ESP2

ENSONIQ SIGNAL PROCESSOR 2

Part I

Instruction and Hardware Specification

1995

David Andreas
Jon Dattorro
J. William Mauchly

MISSING INFORMATION:

Serial, external memory, and host interfaces; setup & hold times.
Recommend pull-up resistor values.

0. Introduction

The demands of digital audio require particular features from DSP (Digital Signal Processing) chips for sound-effect design. The semiconductor industry serves a broader customer base, however. While several commercial processor chips are more or less suitable to the audio technical community, this is not what drives digital audio-based companies into the intense and expensive design cycle of custom VLSI. The principal driving force is the competitive edge gained by proprietary hardware which stifles reverse engineering. The primary end-product, of course, is software; the algorithms which drive the chips.

To remain competitive with contemporary products, a typical commercial sound-effects box needs to execute roughly 30 different algorithm <u>types</u>, each satisfying certain artistic standards of quality. While in the throes of an audio signal processor design, a DSP guru from CCRMA, Stanford University, theorized that most DSP algorithms can be formulated in terms of the digital filter. [Moorer] Reverberation algorithm design, however, remains more of an artistic endeavor than a science; this elusive specialty is still dominated by a few companies having the benefit of an early start. [Blesser/Bader] [Griesinger]

Having this mandate, three design engineers were given carte blanche in 1990 to improve upon an existing and proven chip design known as ESP. Rarely does an audio-product specification call for the algorithm designers to engineer the computer as well. The outcome is the fixed-point ESP2 which, as it turns out, exceeds the capabilities of commercial DSP chips with regard to audio processing efficiency.

The purpose of Part I, the *Hardware (or Chip) Specification*, is to explain the ESP2 chip design philosophy as it pertains to DSP applied to audio. The ESP2 programming Language is discussed in Part II, often referred to as the *Software Specification*. We discuss effect design and present algorithms for several practical and real Audio Applications in Part III. We do this, first, to highlight the efficiency of the ESP2, second, as an exposition of how fundamental DSP operations are accomplished in real-time, third, to present some new results in the field of audio and DSP.

1. Chip Overview

Part I fully explains the functionality of the Digital Signal Processor chip called ESP2. To assist the programmer, this *Chip Spec.* explains the implementation of all the ESP2 instructions. The information required by the engineer to design this chip into some system is also found here, and is explained so as to be accessible to programmers having a limited hardware background.

The information regarding the instructions is supplemented in Part II, the Language and Software Spec., which explains nuances of the syntax which is the ESP2 assembly language. The ESP2 language resembles some aspects of the C programming language. The language is easy to learn, intuitive, and truly a step up from the standard practice of manipulating unmeaningful register names.

The Software Spec. should be read by anyone using the ESP2 for intensive operations involving external memory access (e.g., Reverberator design). Other engineers wishing to write simpler test programs can get by with Part I and an example of a complete ESP2 program from which they may derive a shell to work within. Programming examples can be found in the Applications section, Part III. The References are found thereafter.

The ESP2 chip architecture, shown in Figure 1, is optimized for the processing of audio signals. The demands of audio dictate a minimum single precision bit-width of 24 bits. The most prominent feature of this architecture is the three parallel function units: Address Generator (**AGEN**), Arithmetic Logic unit (**ALU**), and Multiplier/Accumulator/shifter unit **(MAC unit)**.

Each function unit is complemented by extra source/destination registers called **SPR (**Special Purpose Register**)**. These registers all have unique purposes specialized to the unit that they support. In many instances SPRs increase the number of operands employed by a single instruction.

The AGEN is supported by a distinct block of registers called **AOR (**Address Offset Register**)**, which facilitate random access of off-chip data on each instruction cycle. These registers provide address offsets to AGEN's modulo address calculation mechanisms. Unlike conventional DSP chips, the AORs facilitate the design of sparse digital networks which is a requirement of audio signal processing; e.g., digital reverberators.

Both the ALU and the MAC unit perform their three-operand instructions directly on all the internal registers including **GPR (**General Purpose Register**)**. MAC unit and ALU direct access of SPR is useful to control the many specialized processes. The AORs are directly accessible from the ALU and MAC unit for the purpose of modulating addresses or for any general purpose. This feature is useful for time-varying processes.

The instruction memory is 96 bits in width and completely internal; i.e., there is no provision for off-chip program memory. This large instruction word supports the parallel architecture such that all three units can function in parallel while the instruction set supports this. While the ESP2 also supports standard program control operations such as branching and calls to subroutines, there are hardware provisions for program space conservation. This conservation includes a low-overhead looping mechanism and conditionally executable instructions. The latter feature eliminates the overhead associated with branching around status dependent code, and improves efficiency threefold through the selective conditional execution of any or all of the parallel function unit operations.

Although the ESP2 is inherently a parallel/pipeline design, all ESP2 instructions individually execute in the same amount of time (one instruction cycle, four system clocks) under all circumstances.

1.1. Chip Architecture

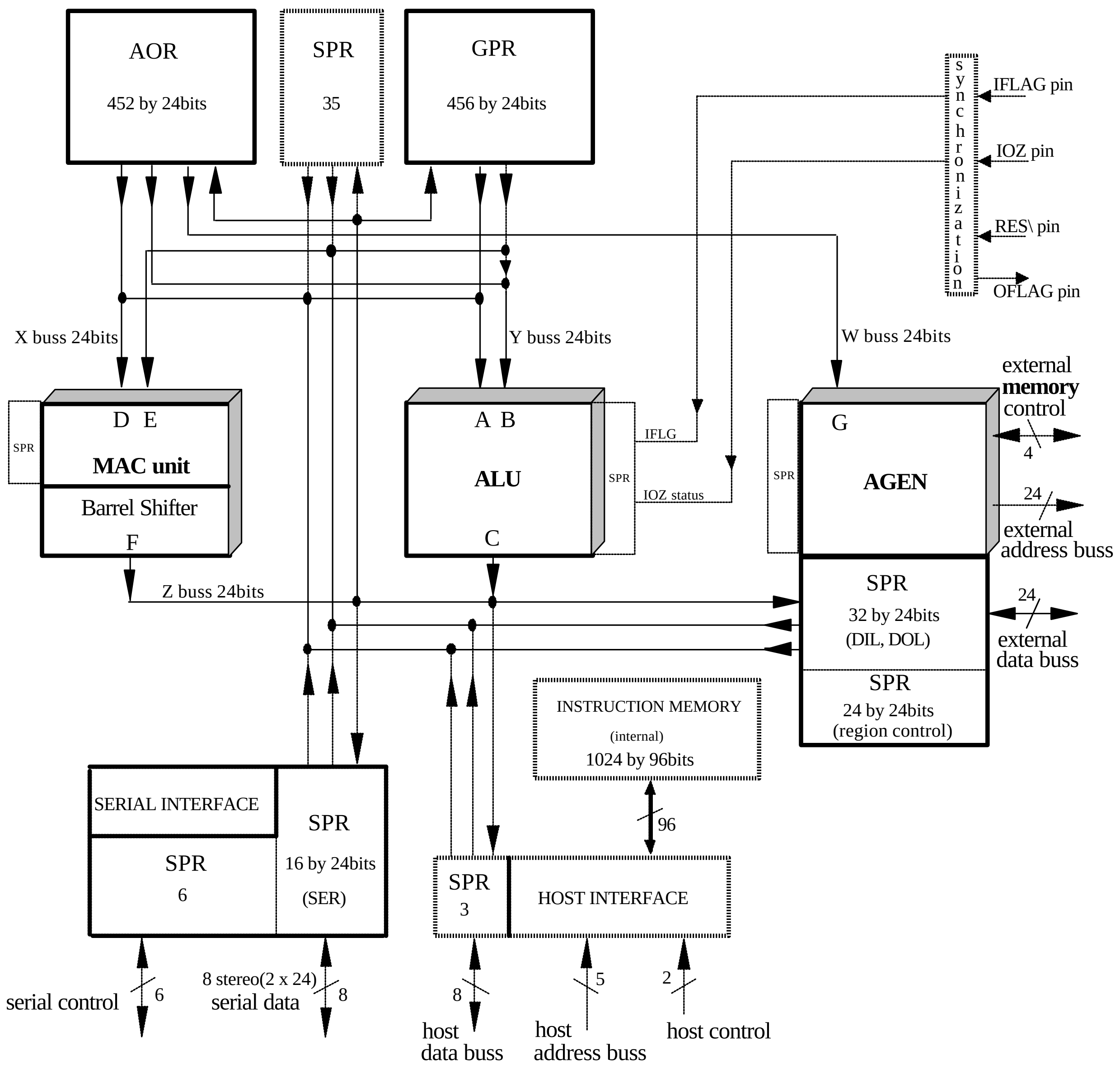


Figure 1. ESP2 Chip Architecture

1.1.1. Function Units: MAC, ALU, AGEN

The three main function units are designed to execute in parallel employing an instruction set which supports the parallelism. The internal *pipeline* design dictates a specific hardware ordering-in-time which sees the MAC unit first, followed mid-cycle by the ALU. The AGEN timing exactly parallels that of the MAC unit. Some of the inter-unit pipeline latencies discussed **(**see the Instruction Cycle Timing diagram**)**, will be more easily understood if this ordering-in-time is kept in mind.
The order of each instruction *field*, on the programmer's single line of code **(**one **instruction cycle**, one program line, one instruction line**)**, within which each unit's instruction will be found is:

MAC unit	*ALU*	*AGEN*

The latencies are few, while easy-to-understand charts are presented in the Software Specification to assist the programmer. It is important that the programmer recognize that the speed and efficiency of the ESP2 processor for most all DSP tasks is facilitated by this pipeline design.
One benefit of the parallelism is that the programmer will find many common instructions amoung the units; e.g., both the MAC unit and ALU have identical MOV and identical shift (ASH) syntax, so the programmer can simply cut and paste into the appropriate field.

The MAC unit is a three operand device (two source, one destination), plus a seed source. The MAC unit instruction set comprises 20 fundamental instructions, 10 variations, and an assortment of pseudo instructions providing a large palette of multiply/accumulate/shift operations. A prominent feature of the MAC unit is the ability to selectively inhibit latching of the accumulator result while sending it to some destination register. A Barrel Shifter is integral to the MAC unit, available on a per-instruction basis, and can be accessed by the ALU. The Barrel Shifter can be used to shift either the input to the accumulator or the accumulator output.

The ALU instruction set consists of 32 standard, non-standard, and Boolean instructions. All of these instructions can take two source operands and another destination operand. The non-standard operations are used for such things as FFTs, envelope generators, stereo-to-mono signal conversion, etc. The ALU instruction set is augmented with a wide assortment of pseudo instructions. Program control instructions which allow conditional branching are found in the ALU. While branching allows the programmer to jump over parallel instructions to the ALU, MAC unit, and AGEN, a separate mechanism allows conditional execution of individual instructions to any or all of the three function units.

The AGEN performs modulo addressing of 8 distinct regions of data located nearly anywhere in external physical memory and of any size. Within each region can be defined numerous delaylines whose region address offsets are determined by the multiplicity of AORs employed by AGEN. AGEN features include a Plus-One addressing mode, and UPDATE of the region BASE under program control. The AGEN can also perform absolute addressing as would be required for peripheral I/O.

1.1.2. Internal Registers.

Operand addresses are 10 bits allowing up to a total of 1024 internal registers. This register space is apportioned as follows:

Table 1. Internal Register Address Map

$000 - $1c7	General Purpose Registers (GPRs)

\$1c8 - \$1ff	Special Purpose Registers (SPRs)
\$200 - \$3c3	Address Offset Register (AORs)
\$3c4 - \$3ff	Special Purpose Register (SPRs)

The ESP2 instruction set operates directly on these registers given meaningful names by the programmer.
The MAC unit (with one restriction) and ALU can utilize all the registers as operands.

1.1.3. GPR and AOR

GPRs (General Purpose Registers) and AORs (Address Offset Registers) are 24-bit wide registers implemented as large dynamic ram arrays. GPRs and AORs have two read ports and one write port. Each of the ports is accessed twice per instruction cycle. This gives a virtual set of four read ports and two write ports per instruction cycle for each of the arrays.
The GPR read ports allow two operand fetches for the ALU and two for the MAC unit in an instruction cycle. The two GPR write ports allow one result-write by the ALU and one by the MAC unit.
The AOR array is used to hold address offsets for the AGEN. When not used for holding address offsets, <u>AORs may be used just like GPRs</u>. The four AOR read ports allow one offset fetch for the AGEN, one operand fetch for the MAC unit, and two operand fetches for the ALU. The two write ports of the AOR allow one result-write by the ALU and one by the MAC unit.
Host access to GPR and AOR is governed by the ALU.

Because these registers are dynamic RAM they must be refreshed to maintain the data. The mechanism for refreshing these registers is somewhat transparent and built into the instruction set. GPR and AOR power-up in their lower power state; i.e., when these registers read as logical 1, they are in their lower power consumption state.

The address map allows 456 GPRs and 452 AORs. Due to die size limitations, it is not at first planned to include all of these registers. The initial revisions will have 256 GPRs and 256 AORs.

1.1.4. SPR

SPRs (Special Purpose Registers) are static registers for holding data specific to a particular operation of a function unit, for interfacing to the external ports of the chip, for controlling certain operating modes, or for chip hardware configuration and status. Internal chip control and status registers, such as the Program Counter (PC) or the Condition Code Register (CCR), are mapped as SPRs to provide access via the system host. Host access to SPR is governed by the ALU.

All the function units (ALU, AGEN, MAC) have supporting SPRs. In some instances, one or several SPRs effectively behave as extra **s**ource/destination operands to a particular function unit. Unless stated otherwise, these extra source/destination SPRs are subject to the same inter-unit latencies as any conventional source/destination register. **(**See the section on Inter-Unit Latency.**)**

SPRs are accessible as operands in all of the same modes as GPRs, although some of the SPRs are read-only. SPRs are distributed over the GPR and AOR address space so as to balance the number of GPRs against the number of AORs.

1.1.5. Internal Register Usage

The rules governing the use of the three types of registers as operands for the three function units are indicated in Table 2. (The symbol * in Table 2 denotes a valid usage.) Notice that an AOR is not a valid E source operand in the MAC unit because one of the AOR's four virtual read ports is usurped by AGEN.

Table 2. Registers as Operands

	ALU			**MAC unit**			**AGEN**
	A	B	C	D	E	F	G
GPR	*	*	*	*	*	*	
AOR	*	*	*	*		*	*
SPR	*	*	*	*	*	*	
Buss	X	Y	Z	X	Y	Z	W
Source	*	*		*	*		*
Destination			*			*	

1.1.6. Instruction Memory

The instruction memory is presently a 300 by 96-bit dynamic memory array. These 300 ESP2 instructions are equivalent to 900 instructions of a more conventional architecture because of the three parallel function units that constitute the ESP2. Future expansion sets the maximum possible number of ESP2 instructions at 1024 (3072 conventional). At a sample rate of 44.1 kHz and using a system clock of 40 MHz, ESP2 can execute 226 instruction cycles (678 conventional) per sample period.

The instruction memory cell has one write port and one read port and cycles at twice the instruction rate. This allows one cycle for instruction fetching, and a read/write cycle for refresh or host access to the instruction memory array. Refresh of instructions is transparent to the programmer. Instruction memory powers-up in its lower power state; i.e., when instruction memory reads as logical 1, it is in its lower power consumption state.

Instruction memory can be downloaded, overlaid, or uploaded by the system host at any time at the instruction rate. This is because the host interface is dichotomized between internal register and instruction memory.

1.1.7. Future Expandability

It should be possible to shrink the chip to denser process geometries. This would improve performance by allowing operation at higher clock rates. Since the chip is designed with a 10-bit Program Counter and 10-bit operand (address) **field**s, we can add GPR, AOR, and instruction memory with minimal layout effort.

1.1.8. Internal Operand Busses

The ESP2 contains four 24-bit data busses, W, X, Y, Z. The X and Y busses are time-multiplexed for fetching ALU and MAC unit source operands. These busses connect to the GPR memory array, the AOR memory array, the ALU, the MAC unit, and to all SPRs. The source register (GPR, AOR, or SPR) specified as the ALU's A operand is always fetched on the X buss, as is the MAC unit's D source operand. The register (GPR, AOR, or SPR) specified as the ALU's B source operand is fetched on the Y buss. The register (GPR or SPR, but no AOR) specified as the MAC unit's E source operand is fetched on the Y buss as well. The AOR addressed as the AGEN's G source operand is fetched on the W buss. The Z buss is used to deliver results to the registers (GPR, AOR, or SPR) designated by destination operands, C and F, from the outputs of the ALU and the MAC unit respectively.

The MAC unit and ALU share the X, Y, and Z busses by relinquishing their use on different phases of the same instruction cycle. Instruction Cycle Timing is covered in the section of the same name.

1.1.9. External Interfaces

There are four mechanisms for interfacing to external memory and devices: the external memory interface, the serial interface, the host interface, and chip synchronization.

The **external memory interface** is a 24-bit address, 24-bit data buss for random access of delaylines and tables stored in external memory. This **external memory buss** can also be used to access memory-mapped I/O devices. Addresses for this buss are calculated on every instruction cycle by the AGEN, under program control. The code which drives AGEN can be generated automatically for the programmer by the assembler, if desired. The DIL (Data Input Latch) and DOL (Data Output Latch) SPRs provide the interface to the external memory data buss for incoming and outgoing data as they are accessed as normal sources and destinations of instructions, under program control. The external memory buss cycles at the instruction rate.

The **serial interface** consists of 8 serial stereo data lines. Each of the lines can be configured as input or output. There are two fully programmable sets of clocks for controlling the timing of data transfers on the serial data lines, and each data line can be assigned to either set of clocks. The serial clocks may be disabled to allow exogenous devices to dictate the serial timing. The SER data SPRs (e.g., SER0L) provide the interface to incoming and outgoing serial audio data as they are accessed as normal sources and destinations of instructions, under program control.

The asynchronous **host interface** offers an 8-bit data exchange on the host side, but 24-bit on the ESP2 internal register side. It is described more fully in the later section, Host/ESP2 Interface. The host/ESP2-register interface timing internally parallels that of the ALU; the system host transfers data to/from GPR/AOR/SPR registers using normal ALU data paths (the Y and Z busses). Transfers are <u>completely</u> under ESP2 program control, however, the exact time of the register transfer governed by the ALU HOST and BIOZ instructions. The host will not hang waiting for an acknowledge because built-in semaphores are polled by the host. When the chip is halted, it continually executes HOST instructions by design.
Instruction memory is always accessible to the system host at the instruction rate regardless of the state of the chip. Instruction memory access does not usurp internal chip resources because there is a dedicated 96-bit buss for this purpose (8 bits wide on the host side).

The chip **synchronization interface** includes the IOZ input pin, the IFLAG input pin, the OFLAG output pin, and the RES\ (reset) pin:
The IOZ input pin is most often tied to the system sample rate signal called LRCLK. This signal is asynchronous and comes from <u>any</u> desired source including the ESP2 chip itself. The IOZ pin is indirectly monitored by the ALU BIOZ instruction to synchronize the running ESP2 program to the sample rate.
The IFLAG input pin is uncommitted and can be used for any desired purpose. Its intended purpose is for use as a semaphore in a rapid-transfer DMA scheme. It is visible through the CCR in the ALU as IFLG. Other transfer schemes to external memory are discussed in the Applications section.
The OFLAG output pin is also uncommitted and can be used for any desired purpose. It is connected to the OFLG bit in the HARD_CONF SPR.
The RES\ pin impact is discussed in the Chip Reset and Initialization section. It can be used in a multi-processor environment to initially synchronize several ESP2 chips running

in parallel. **(**Provision has been made in the external memory interface for sharing external memory.**)**

1.2. Instruction Cycle Timing

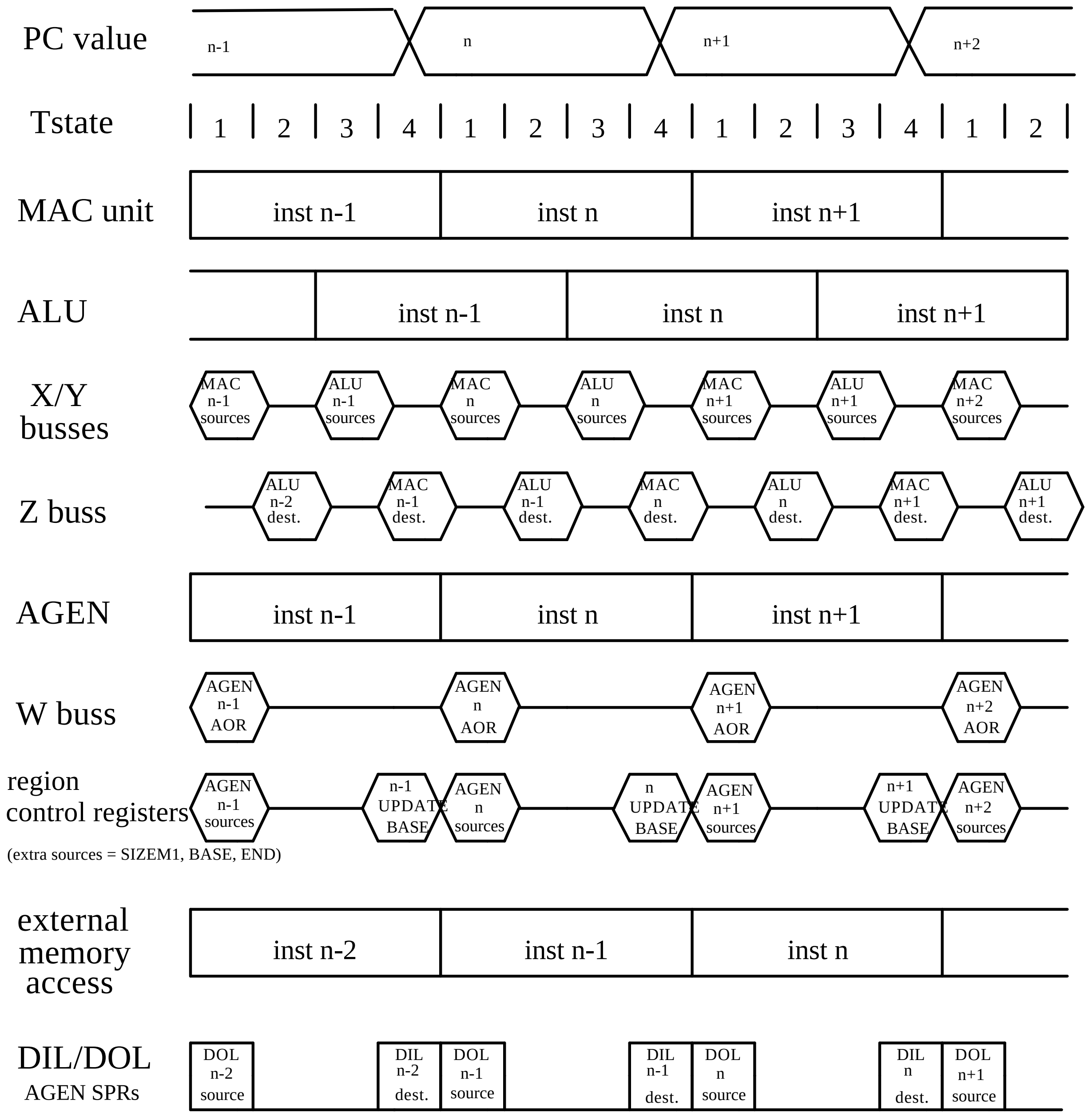


Figure 2. Instruction Cycle Timing diagram

All ESP2 instructions execute in one instruction cycle. The Instruction Cycle Timing diagram shown in Figure 2 illustrates the relative timing of instruction execution for each of the function units (MAC, ALU, AGEN), and their timing relationship with the access to external memory. Notice that the external memory access lags the AGEN instruction requesting it.

The access to internal source/destination operands is also indicated. Execution timing in the MAC unit and ALU are interleaved to allow four operand-fetches and two operand-stores by multiplexing the X, Y, and Z busses. By operating these busses at twice the instruction rate, the MAC unit and ALU can each be supplied with two source operands from GPR/AOR/SPRs and have their results stored back into GPR/AOR/SPRs on every instruction cycle.

1.3. Latency

There are five categories of execution latency due to the internal pipeline architecture: 1**)**inter-unit, 2**)**external memory access, 3**)**ALU instruction cycle execution, and 4**)**serial data access, 5**)**pointer register access for indirect register addressing. The last three categories are discussed in the pertinent sections. With few exceptions, when latencies arise, the latency across the categories is one instruction cycle. This regularity makes for a highly orthogonal design.

1.3.1. Inter-Unit Latency

The interleaving of the MAC unit and ALU timing produces some latency in the availability of the result of the ALU instruction with respect to the MAC unit instruction. As illustrated in Figure 2, the MAC unit fetches its source operands for the $(n+1)^{th}$ queued instruction before the ALU has stored operation results to its destination from the n^{th} instruction. Similar latencies arise between the AGEN and the ALU since the AGEN fetches are coincident with those of the MAC unit. The AGEN source operands **(**which include the AOR (address offset), and BASE, SIZEM1, and END region control registers**)** are acquired at the beginning of an instruction cycle, while there is an optional UPDATE of the region BASE at the end of the cycle.

1.3.2. Latency of External Memory Access via AGEN

AGEN must calculate an address during one instruction cycle while the external memory access at that address physically takes place during the next instruction cycle. The external memory data-interface registers, the SPRs called DILs and DOLs, look like any other register to the MAC unit and ALU. In the case of external memory reads (RD), this pipeline design requires the assembler to schedule a RD at least two instruction cycles before the data is actually supplied to the ALU or MAC unit via **DIL** (Data Input Latch). In the case of external memory writes (WR), the pipeline delays the usage of the MAC unit and ALU results written out via **DOL** (Data Output Latch). The pipeline design allows a WR from the MAC unit to external memory to be scheduled as early as the same instruction cycle as the MAC unit instruction which requested it (or any time thereafter). But the pipeline design allows a WR from the ALU to be scheduled no sooner than one instruction cycle after the requesting ALU instruction.

The following manifest quantifies the inter-unit latencies and the AGEN scheduling rules:

The interleaving of the MAC unit and ALU operations and the alignment of the AGEN timing with the MAC unit timing have important ramifications from a programming point of view. The following rules summarize the register data access latencies between function units.

Nonlatent operations

1. The result of a MAC unit operation is available for use as a source operand by the MAC unit no sooner than the next instruction cycle (next queued program line).

2. The result of a MAC unit operation is available for use as a source operand by the ALU no sooner than the next instruction cycle.

3. The result of an ALU operation is available for use as a source operand by the ALU no sooner than the next instruction cycle.

4. The result of a MAC unit operation written to an AOR or AGEN region control register is available to the AGEN no sooner than the next instruction cycle.

Latent operations

5. **The result of an ALU operation is available for use as a source operand by the MAC unit no sooner than the second instruction cycle following the ALU instruction.**

6. The result of an ALU operation written to an AOR or AGEN region control register is available to the AGEN no sooner than the second instruction cycle following the ALU instruction.

The **scheduling** of DIL/DOL RD/WR from/to external memory, relative to the timing of MAC unit and ALU operand access, follows these rules:

<u>A</u>latent operation

7. External memory writes of MAC unit results to DOL can be scheduled on the same program line **(**or on any line thereafter**)** as the instruction which generates the data.

Nonlatent operation

8. External memory writes of ALU results to DOL can be scheduled no sooner than one instruction cycle after the instruction which generates the data.

Latent operation

9. The fetch of data from external memory for use by either the ALU or the MAC unit as a DIL source operand must be scheduled at least two instruction cycles prior to the sourcing instruction.

For a nice programmer's chart, see the ESP2 Language and Software Specification in the Pipeline section.

2. Multiplier/Accumulator/Shifter

2.1. Architecture

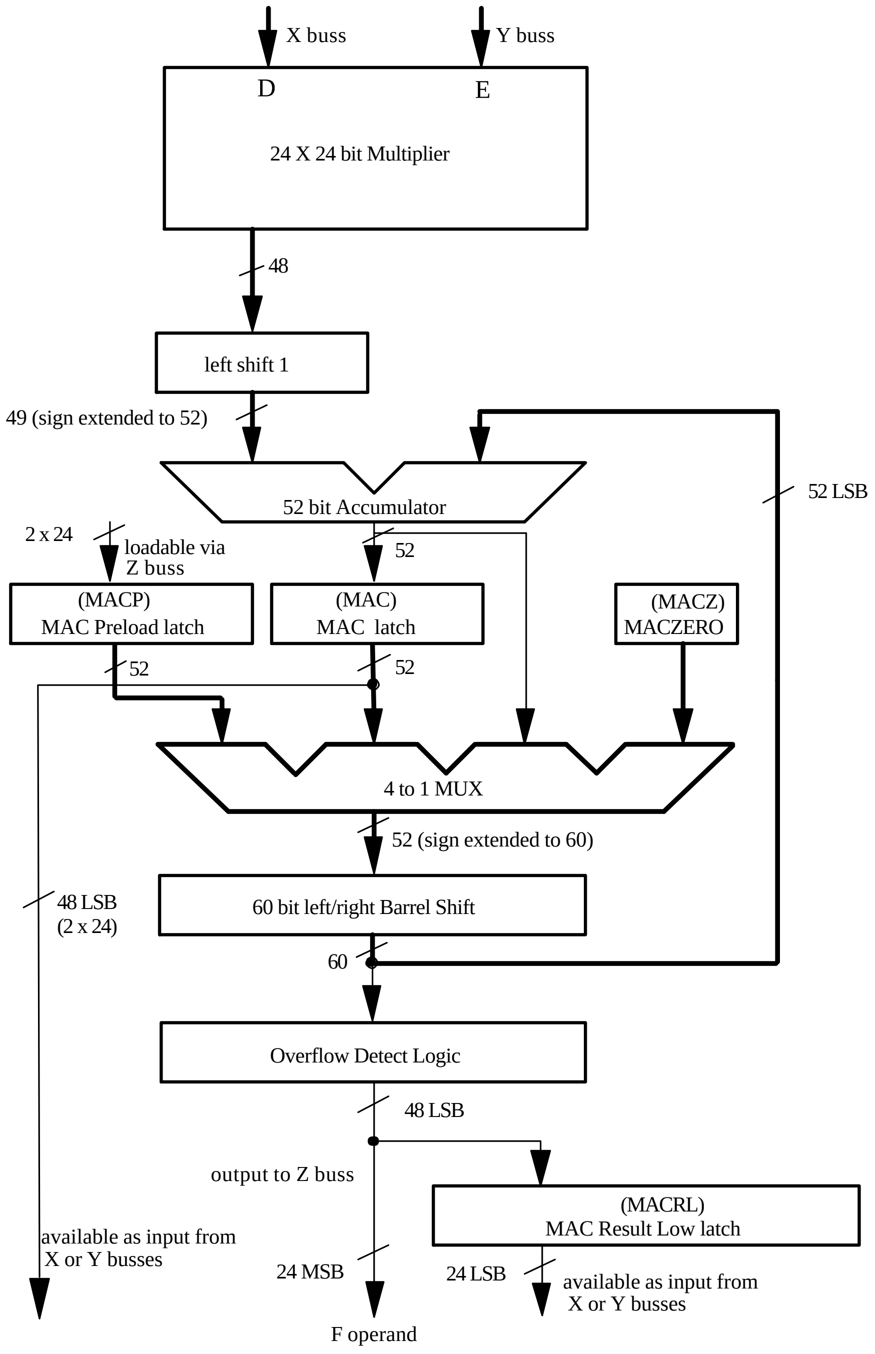


Figure 3. MAC unit architecture. Bold buss shows first half instruction cycle.

The multiplication unit is shown in Figure 3. It consists of a 24 by 24-bit **signed** multiplier, a 52-bit accumulator, a 60-bit left/right arithmetic Barrel Shifter, overflow detect logic, a 4 to 1 MUX and latches. The multiplier sources are fetched on the X and Y buss from registers specified by the D and E operand fields of the instruction. Since the 48-bit product of the 24 by 24-bit fixed-point multiply has two sign bits, a fixed normalizing shift left of 1 bit is designed permanently into the product path. The accumulator adds or subtracts the left-shifted multiplication product, having appended 3 bits of sign extension, to the 52-bit value from the feedback path (bold buss). This allows 4 guard bits for use in detecting overflow in the accumulated result. The accumulator result can then be selectively stored in the MAC latch, or out to a destination, or both. (The symbol labeled 'ACCUMULATOR' has no internal output latch.)

During the first half of an instruction cycle, when the right-side accumulator input is being loaded, the 4 to 1 MUX will select one of three *seed* sources; either the **MAC** latch, the constant MACZERO (**MACZ**), or the MAC Preload latch (**MACP**). These three input options allow:
1**)** accumulation of the multiplier product with the MAC latch,
2**)** multiplication without accumulation,
3**)** accumulation of the multiplier product with a sign-extended (to) 52-bit Preload value (MACP).

Positioning the Barrel Shifter in the feedback path allows shifting of the MAC or MACP latch seeds prior to accumulation.

In the second half of the instruction cycle, there are four destination options:
1**)** 24 MSBs of the double precision accumulator can be written out to a single precision destination register (via the Z buss), and not to the MAC latch,
2**)** to a single precision destination register, and 52-bits to the MAC latch,
3**)** double precision shifted to a single precision destination register, but not shifted to the 52-bit MAC latch,
4**)** double precision shifted to a single precision destination register, but not shifted and not to the MAC latch.
The accumulator result must propagate through the 4 to 1 MUX, the 60-bit shifter, and the overflow detect logic to the reach the Z buss for writing to a destination register. The shifter and overflow detect logic must be separate so that only the final result of a series of intermediate multiply/accumulate/shift operations becomes conditionally saturated before being written to a destination.

The Barrel Shifter always performs a 60-bit arithmetic shift of 0 to 8 bits left or 0 to 7 bits right. The instruction contains only one shift amount, and the shifter can be used to shift the accumulator right-side input or the accumulator output to a destination register, but not both in the same instruction cycle.

The sign extension to 60 bits of the 52-bit MUX output is required because of the maximum possible shift left of 8 bits. The overflow detect logic checks the 13 MSBs of the 60-bit shifter output for overflow. Overflow exists if those 13 bits are not all in the same state. When overflow has occurred the MAC unit output is saturated to either most positive $7FFFFF,FFFFFF if the MSB of the shifter input is a 0, or most negative $800000,000000 if the MSB of the shifter input is a 1. Conditional saturation occurs only at the MAC unit output; there is no saturation of intermediate accumulator results stored in the MAC latch.

Neither is there any saturation of the left-shifted MAC or MACP latch when used as the accumulator seed input. But when the left-shifted MAC or MACP is used as an accumulator input and then the conditionally saturated result is sent to a destination register, it is possible to detect overflow of these two seed inputs if the overflow is not in excess of 4 bits. This reduction in overflow detection capability for shifts left of the MAC and MACP latch inputs is because of truncation on the feedback path in the MAC unit.

2.1.1. Exception Processing

There are two special cases that must be handled by the MAC unit:

Table 3. Multiplier Exceptions

		MAC latch		Destination: **F,MACRL**
$800000 X $800000	=	$0,800000,000000	=	$7FFFFF,FFFFFF
-$800000 X $800000	=	$f,800000,000000	=	$800000,000000

2.2. MAC unit Barrel Shifter

The Barrel Shifter residing within the MAC unit performs only arithmetic shifts. The Barrel Shifter performs two functions:

1**)** It can be used on the accumulator output path to perform a shift of +8 (left) to -7 bits of the 52-bit accumulated sign-extended (to 60 bit) result.

2**)** The Barrel Shifter can be applied to the MAC latch as seed-source, or to Preload values in the MACP latch as seed-source, for double precision shifting by +8 to -7 bits.

If it is desired to examine the 4 MAC latch guard bits, this can be done by first right-shifting them in place back into the MAC latch. They can then be examined by sourcing directly from the MAC latch (the SPR MACH).

ASSEMBLER NOTE: The shift code stored in the instruction is an unsigned 4-bit value. The value is derived by the following equation:

Shift code = desired shift amount + 7

Since the desired shift amount is in the range from +8 to -7 bits, the result of the Shift code equation falls into the range 0 to 15.

2.3. Accessing the MAC unit

The MACP latch is accessible as two pairs of 24-bit write-only SPRs: MACP_H and MACP_L, and MACP_HC and MACP_LS. *The ESP2 assembler generally disallows any reference to MACP as a source operand which is not the seed source in the MAC unit.* When MACP is used as a single precision destination, this is synonymous with the SPR, MACP_HC.

The unsaturated MAC latch is directly accessible as a pair of 24-bit read-only SPRs, MACH and MACL, for use as source operands in the ALU or MAC unit with normal latencies. *From the ALU, using the MAC latch as a destination is disallowed by the assembler.* When MAC is used as a single precision source operand, this is synonymous with the SPR, MACH.

2.3.1. Writing the MACP (Preload) latch

MACP is used in accumulation, instead of the MAC latch or MACZ, under control of the program. The MACP latch is nonvolatile and will retain a value written to it until written again. Because of the interrelationship between the ALU and MAC unit instruction timing, the value written by the ALU into the high or low-order bits of MACP during the n^{th} instruction line will be available for accumulation 2 instruction cycles later at the $(n + 2)^{th}$ queued instruction line. The MAC unit can also initialize the high or low-order MACP latch, just as it can load any other SPR. In this case, MACP is available to the MAC unit on the next instruction cycle. The ALU and the MAC unit can be used in conjunction to initialize the full 52 (high and low-order) bits of the MACP latch.

When the SPRs, MACP_H or MACP_L, are written, the 24-bit value is written into the indicated half of the MACP latch.

When the SPR, MACP_HC, is written, 24 bits are written into the high-order half of the MACP latch while the low-order half is cleared to all zeros. When SPR, MACP_LS, is written, the 24-bit value is written into the low-order half of MACP while the high-order half is written with the sign-extension of the value in the low-order half. These allow MACP initialization to the full 52-bit accumulator width in one instruction cycle.

Any write to the high-order half of the MACP latch is sign-extended into the 4 guard bits to create a full 52-bit word.

2.3.2. Reading the MAC latch

Since the MAC latch is located before the overflow detect in the circuit topology, values read from the MAC latch (using the MAC unit SPRs, MACH and MACL, as sources) will <u>not</u> be conditionally saturated. Our intention was to provide unsaturated MAC unit results as source operand for the MAC unit itself as well as for the ALU.

Also note from observation of the fundamental instructions, that the MAC latch will be **un**shifted with regard to the last executed MAC unit instruction if it was <u>not</u> of the form:

MAC(P) >>n +/- D X E > MAC(,F)

That is to say for many instructions, the MAC unit output is shifted but the MAC latch acting as extra destination is not.

2.3.3. MAC Result Low latch (MACRL)

Examination of the fundamental MAC unit instructions shows that the MAC unit has a destination on every instruction cycle. If the programmer does not specify one, the assembler chooses the read-only ZERO SPR as the destination. On every instruction cycle, the low 24 bits of the conditionally saturated MAC unit output will be written to the MAC Result Low latch because it is in the output path. Since MACRL is mapped as an SPR, it is readable by the MAC unit and ALU as a source operand. Storing the low word allows double precision arithmetic using all 48 bits of the final result out of the MAC unit. It is necessary to read the MAC Result Low latch before it is overwritten by the MAC unit in the next instruction cycle. The only exception is when the MAC unit is executing NOPs in which case the contents of MACRL will be preserved (see the MAC unit NOP pseudo instruction).

2.4. MAC unit Instructions

The fundamental instructions executed by the MAC unit are two-source one-destination instructions, having an optional seed source and an optional MAC latch destination, of the form:

(MAC(P)**)** +/- D X E > (MAC,)F

When a seed source is not specified, the assembler inserts MACZERO (MACZ). The D and F operands can be any GPR, SPR, or AOR. The E operand can be any GPR or SPR. The F operand acts as the register-type destination. At least one of the two destinations must be specified by the programmer.

When the only destination specified is MAC, the assembler substitutes the read-only SPR called ZERO for the F operand. Unsaturated results of the accumulation are stored in the MAC latch (the SPRs: MACH, MACL) for use in subsequent accumulations.

When the only destination specified is F, the MAC latch is inhibited as a destination. This inhibition is a means to preserve the previous MAC latch contents.

Table 4 lists the fundamental instructions of the MAC function unit:
(Parenthesis not required; it only serves to clarify the operation.)

Table 4. MAC unit List of Instructions

MACZERO + D X E > MAC >>n	> F
MACZERO - D X E > MAC >>n	> F
MACZERO + D X E >>n	> F
MACZERO - D X E >>n	> F
MAC + D X E > MAC >>n	> F
MAC - D X E > MAC >>n	> F
(MAC + D X E) >>n	> F
(MAC - D X E) >>n	> F
MAC >>n + D X E	> MAC, F
MAC >>n - D X E	> MAC, F
MAC >>n + D X E	> F
MAC >>n - D X E	> F
MACP + D X E > MAC >>n	> F
MACP - D X E > MAC >>n	> F
(MACP + D X E) >>n	> F
(MACP - D X E) >>n	> F
MACP >>n + D X E	> MAC, F
MACP >>n - D X E	> MAC, F
MACP >>n + D X E	> F
MACP >>n - D X E	> F

In six of the instructions in Table 4, notice the MAC latch gets the unshifted unsaturated result while the destination operand, F, gets the shifted and conditionally saturated result. In four other cases, the MAC or MACP latch as seed source is shifted, but the MAC latch as extra destination remains unsaturated. But in all cases, the destination, F, receives the shifted and conditionally saturated result. The shift amount *n*, as specified in Table 4, can be +7 to -8 bits; specified as <<n it can be -7 to +8 bits. The shift amount *n* is a constant expression that is encoded into the micro-instruction word.

2.5. MAC unit Pseudo Instructions

The MAC unit pseudo instructions are designed to preserve the MAC latch where possible. Therefore most pseudos do not provide MAC as a destination. Note that the MAC latch is neither a valid destination from the ALU (but MACP is).
Generally speaking, only the MAC unit pseudo, NOP, preserves MACRL.

ADD D, MAC > MAC, F = MAC - D X MINUS1 > MAC, F ! destination is MAC or F or both
ADD D, MAC > MAC >>n > F = MAC - D X MINUS1 > MAC >>n > F
ADD D, MAC = MAC - D X MINUS1 > MAC
ADD D, MAC >>n > MAC = MAC >>n - D X MINUS1 > MAC

ADD D, MACP > MAC, F = MACP - D X MINUS1 > MAC, F ! destination is MAC or F or both
ADD D, MACP > MAC >>n > F = MACP - D X MINUS1 > MAC >>n > F
ADD D, MACP = MACP - D X MINUS1 > MACP
ADD D, MACP >>n > MAC = MACP >>n - D X MINUS1 > MAC

The MAC latch as a destination is unsaturated, double precision. MACP as destination is single precision, conditionally saturated.

ASH D >>n > F = - D X MINUS1 >>n > F
ASH some_reg >>n = - some_reg X MINUS1 >>n > some_reg

This pseudo instruction performs an arithmetic shift *n* places of the 24 bit D operand. The value *n* which is encoded in the micro-instruction is the shift range; it is a constant expression which can take any value in the range +7 to -8 bits.

ASH D >>8 > F = D X HALF >>7 > F
ASH some_reg >>8 = some_reg X HALF >>7 > some_reg

These two extra pseudo instruction definitions make the range of shift for ASH symmetrical. Alternatively the programmer might choose the constant (HALF, MINUS1) to be smaller **(**using the primitive instruction**)** thus extending the range of possible shifts right even further.

In a shift right, the programmer should be aware that the MACRL SPR receives all of the bits shifted out of the single precision D operand. This means that bits of the single precision operand are <u>not</u> lost when shifted across the LSB boundary. Considering these pseudo instruction definitions, the 48-bit concatenation, (some_reg,MACRL), will contain all 24 of the right-shifted bits of some_reg.

PROGRAMMER NOTE: Double precision SHIFTMAC(P) pseudos or the ALU's double precision shift instructions should also be considered. Note that the ALU's ASH pseudo instruction can not incorporate the MACRL SPR to catch the LSBs as explained here for the MAC unit.

CLR F = MACZ + ZERO X ZERO > F

DBL D > F = - D X MINUS1 <<1 > F
DBL some_reg = - some_reg X MINUS1 <<1 > some_reg

EXIT = MACZ + ZERO X ZERO > REPT_CNT

This MAC unit pseudo instruction is used to terminate a repeating block of code, instigated by the REPT instruction, at the end of the current block. The MAC unit EXIT pseudo instruction may be placed anywhere within a repeated instruction block **except** for the last line of the block where it will not work at all. See the description of the ALU REPT instruction and the section on SPR Hazards for more details.

HALVE D > F = - D X MINUS1 >>1 > F
HALVE some_reg = - some_reg X MINUS1 >>1 > some_reg

MOV D > F = - D X MINUS1 > F

This allows the multiplier to do a MOV from operand D to operand F. If the D operand is MAC, then the destination will receive the unsaturated MAC latch (MACH) from the previous queued MAC unit operation. This pseudo instruction does not preserve MACRL.

PROGRAMMER NOTE: This instruction uses the MINUS1 SPR ($800000) as the E operand. MOV to the MAC latch is discouraged because the MACP latch fulfills any role as preload register, and because the MAC latch is not a valid destination from the ALU.

MOVSMAC > F = MAC + ZERO X ZERO > F
MOVSMACP > F = MACP + ZERO X ZERO > F

These pseudo instructions send the double precision conditionally saturated MAC or MACP latch to the single precision destination. If you wish to move the unsaturated MAC latch to a destination, use the MOV pseudo instruction above or the ALU MOV instruction.

NEG D > F = D X MINUS1 > F
NEG some_reg = some_reg X MINUS1 > some_reg

NOP = MACRL X ONE >>1 > ZERO

This NOP for the MAC unit, utilizing the SPR called ONE, preserves MACRL. Since the least significant 24-bits of the MAC unit result are always stored in MACRL, this instruction performs a move of the MACRL SPR to itself. The MAC latch is not written, hence it is preserved. This instruction performs no refresh.

NORFSH Identical to NOP

RFSH = - REF X MINUS1 > REF

This instruction performs a MOV using the REF SPR as source and destination registers. See the section on GPR and AOR Refresh for details on this SPR and its use in refreshing internal DRAM. Execution of this instruction will cause the loss of the contents of MACRL from the previous queued MAC unit instruction.

SHIFTMAC >>n	= MAC >>n + ZERO X ZERO > MAC
SHIFTMAC >>n > MAC	= MAC >>n + ZERO X ZERO > MAC
SHIFTMAC >>n > F	= MAC + ZERO X ZERO >>n > F
SHIFTMACP >>n > MAC	= MACP >>n + ZERO X ZERO > MAC
SHIFTMACP >>n > F	= MACP + ZERO X ZERO >>n > F

The SHIFTMAC and SHIFTMACP pseudo instructions perform an arithmetic shift of the 52 bit MAC latch and MAC Preload latch contents, respectively. The constant expression, n, in the equation is the shift amount; it can take any value in the range -8 to +7 bits. If MAC appears as a destination, it remains unsaturated, but it is shifted with double precision. Unlike the MAC latch, the double precision MACP cannot be shifted in place, which explains why there is no corresponding simplest form of SHIFTMACP. As always, any result written to a destination register (including MACP) is single precision and conditionally saturated.

PROGRAMMER NOTE: The rationale behind the definition of these pseudos is the following:
First, we want double precision results unless a single precision destination register is specified.
Second, consider the construct,

MAC >>n *or* MACP >>n + ZERO X ZERO > MAC,F

F is conditionally saturated but not guaranteed saturated correctly in a shift left because the MAC unit feedback path is truncated. Therefore, we can only use this construct reliably for SHIFTMAC(P) when the only destination is MAC (F is the ZERO SPR), because the MAC latch is never saturated.

SQR some_reg > F = some_reg X some_reg > F
SQR some_reg = some_reg X some_reg > some_reg

This is a short hand way of multiplying a number by itself.

ASSEMBLER NOTE: AORs cannot be squared because the MAC unit can only get one source operand from AORs. This event should be flagged as an error for the user.

SUB D, MAC > MAC, F = MAC + D X MINUS1 > MAC, F ! destination is MAC or F or both
SUB D, MAC > MAC >>n > F = MAC + D X MINUS1 > MAC >>n > F
SUB D, MAC = MAC + D X MINUS1 > MAC
SUB D, MAC >>n > MAC = MAC >>n + D X MINUS1 > MAC

SUB D, MACP > MAC, F = MACP + D X MINUS1 > MAC, F ! destination is MAC or F or both
SUB D, MACP > MAC >>n > F = MACP + D X MINUS1 > MAC >>n > F
SUB D, MACP = MACP + D X MINUS1 > MACP
SUB D, MACP >>n > MAC = MACP >>n + D X MINUS1 > MAC

The MAC latch as a destination is unsaturated, double precision. MACP as destination is single precision, conditionally saturated.

XCH some_reg1, some_reg2
= - some_reg1 X MINUS1 > some_reg2 MOV some_reg2 > some_reg1

The exchange instruction is a macro-pseudo instruction which uses one MAC unit operation and one ALU operation in order to exchange the contents of two registers. During an XCH instruction, the MAC unit executes a MOV (pseudo) instruction as defined above, while the ALU also executes a MOV instruction but in the opposite direction. Since the operation of the MAC unit and ALU is interleaved, the register contents are swapped using fewer instructions than either function unit acting alone would require.

The complete results are discernible to the ALU on the next instruction cycle, but due to inter-unit latency, only some_reg2 is discernible to the MAC unit on the instruction cycle following the exchange; there, some_reg1 still holds its original contents. On the second instruction cycle following the exchange, the complete results are discernible to the MAC unit.

ASSEMBLER NOTE: An XCH specified in the MAC unit instruction field employs both the MAC unit and the ALU, therefore no ALU operation can be specified on that instruction line.
A warning should be issued if indirection is specified because to work properly, the programmer must have set up the INDIRB and/or INDIRC SPRs in the ALU, while it is

the INDIRD and/or INDIRF (<u>not</u> INDIRE) SPRs which must have been set up in the MAC unit.

3. ALU and Instruction Set

The Arithmetic Logic unit can perform a variety of general and special purpose arithmetic, data movement, and logical operations. It incorporates classical zero-overhead saturation arithmetic for handling computational overflow, and can shift double precision signals to the left or right for the purpose of normalization. Instructions exist to perform unsigned arithmetic without saturation.

During every instruction cycle the ALU takes one or two 24-bit source operands and produces a 24-bit result which is sent to a (third) destination register. ALU execution overlaps with the operation of the MAC unit, so the two computation units operate in parallel. Detailed description of the ALU and MAC unit execution cycles may be found in the section on Instruction Cycle Timing.

The instructions executed by the ALU are three-operand instructions of the form:

OPERATION A, B > C

The source A and B operands can be any GPR or SPR or AOR. The C operand is the destination and it can also be any internal register. Since external memory access takes place via the interface SPRs called DIL and DOL, the available operands virtually include external memory data.

Some of the instructions that follow **(**especially program control instructions**)** do not use all of the three available operands. Some of the instructions store data in the operand (address) **field**s themselves. Directions have been included showing how the assembler should regard the operands and operand fields of those instructions.

The mnemonic ZERO, used often as an operand, refers to the read-only SPR whose content is zero.

3.1. ALU Instructions

These are the fundamental instructions of the ALU:

ADD is a saturating 2's complement addition operator used to create the sum of two signals. If the sum cannot be represented in 24 bits, full-scale positive ($7FFFFF) or full-scale negative ($800000) is substituted for the overflowed result. The operation performed is:

C = A + B

ADDC Add with carry is a 2's complement saturating addition operator like ADD, but the carry bit in the Condition Code Register (CCR) is added at the LSB. This is valuable for double precision arithmetic. The operation performed is:

C = A + B + carry

When the ADDC instruction is used in conjunction with a preceding ADDV to perform double precision arithmetic, the ADDC operation can saturate the high 24-bit word of the 48-bit result. Since the low 24-bit word was computed in the preceding ADDV operation, its value will not be conditionally saturated. The low word of the result can be adjusted by the following conditional operation:

```
  ADDV  a_low, b_low > c_low
  ADDC  a_high, b_high > c_high
  IF OV
{XOR    MINUS1, c_high > c_low}
```

(See the section on ALU Pseudo Instructions for the IF pseudo, and see the section on Setting Condition Codes for information on the V flag.)

PROGRAMMER NOTE: Since conditionally executed instructions never modify the Condition Code Register, a double precision addition using the ADDC instruction will <u>not</u> execute properly if the preceding ADDV is conditionally executed {}.

ADDV is an unsaturating addition operator. It operates exactly as ADD, except that it lacks overflow detection. For this reason it is not normally used to add signals together, unless the signals are double precision. However, it can be used for generating ramp signals, for performing unsigned address arithmetic, and for double precision arithmetic. It performs the operation:

C = A + B

AMDF This instruction first subtracts the operands as B - A with conditional saturation, and then takes the (one's complement) absolute value of the result.

Equivalent Pseudo-code:

```
if (B - A) < 0  then C = (B - A) ^ $FFFFFF        /* exclusive OR */
else C = B - A
```

This implementation employing the exclusive OR operation, instead of negation, yields the absolute value of negative numbers which are off by 1, making the magnitude of the destination smaller by one LSB of a 24-bit word in two's complement. This instruction is typically found in pitch detection applications where this error is insignificant. If the error needs to be corrected however, then the state of the N flag in the CCR can be monitored.

AND performs the bit-wise logical AND of the two 24-bit operands:

C = A & B

AS performs an arithmetic shift of B by the contents of A using destination C. Saturation will occur if any of the bits shifted left through the MSB differ from the original sign bit. The shift amount is restricted to the range of +8 to -8 bits. Positive values correspond to left shifts and negative values correspond to right shifts. Zeros enter the LSB during left shifts and the sign enters the MSB during right shifts. (See the ASH pseudo.)

ASDH Arithmetic Shift Double High performs a double precision arithmetic shift using the B operand as the high word and the A operand as the low word of a 48-bit input. Saturation will occur if any of the bits shifted left through the MSB differ from the original sign bit. The shift amount for this operation comes from the ALU_SHIFT SPR and its range is restricted to +8 through -8 bits as in the AS instruction. Zeros enter at the low-word LSB in a left shift, and the sign enters at the high-word MSB in a right shift. The result is the high 24-bit word of the conditionally saturated 48-bit shift output.

ASSEMBLER NOTE: **Switch the operands** on this instruction so that ASDH gpr1,gpr2 gets assembled as gpr2 being the A operand and gpr1 being the B operand. This allows the two halves of a double precision word to appear in proper order. In this particular example, gpr2 is the destination (the C operand).

PROGRAMMER NOTE: See the assembler note above. This switch of the operands hobbles indirection. To indirect on gpr1 the programmer must write to INDIRB, and vice-versa. The programmer must always use the verbose form of the instruction explicitly declaring the destination to successfully implement indirection.

ASDL Arithmetic Shift Double Low performs a double precision arithmetic shift using the B operand as the high word and the A operand as the low word of a 48-bit input. The shift amount for this operation comes from the ALU_SHIFT SPR and its range is restricted to +8 through -8 bits as in the AS instruction. Zeros enter at the low-word LSB in a left shift, and the sign enters at the high-word MSB in a right shift. The result is the low 24-bit word of the 48-bit shift output.

The ALU performs this and the ASDH instruction by intelligently extracting a 32-bit field from the 48-bit input based on shift direction and whether the low or high word is the desired result. When the low word is desired, the MSBs of the input are lost before the shift and are, therefore, not available for detecting overflow during a shift.

ASSEMBLER NOTE: **Switch the operands** on this instruction so that ASDL gpr1,gpr2 gets assembled as gpr2 being the A operand and gpr1 being the B operand. This allows the two halves of a double precision word to appear in proper order. In this particular example, gpr2 is the destination (the C operand).

PROGRAMMER NOTE: See the assembler note above. This switch of the operands hobbles indirection. To indirect on gpr1 the programmer must write to INDIRB, and vice-versa. The programmer must always use the verbose form of the instruction explicitly declaring the destination to successfully implement indirection.

PROGRAMMER NOTE: The result of this instruction will saturate to $FFFFFF or $000000 when the V flag in the Condition Code Register is true from the **previous** queued ALU instruction. The direction of overflow will be determined by the N flag of the Condition Code Register from the previous queued ALU instruction.

The N and Z flags will then be set based on the 24-bit result from <u>this</u> instruction, the V flag will not be modified.
If the saturation feature is undesirable, use the LSDL instruction instead.

PROGRAMMER NOTE: Since conditionally executed instructions never modify the Condition Code Register, a double precision shift using the ASDL instruction will not conditionally saturate reliably based on the previous queued ALU instruction if that previous instruction is conditionally executed {}.

AVG takes the average of two operands:

$$C = (B + A) >> 1$$

The guard bit of the addition result is included in the shift, therefore overflow is not possible. This means that two full scale signals may be used as inputs without a saturated result. Often used for stereo to monophonic signal conversion.

PROGRAMMER NOTE: The operation, (B - A) >>1 , can be coded using the AVG instruction as follows:

```
AVG somereg_a, somereg_b > somereg_c
SUB somereg_a, somereg_c > some_other_reg
```

This method stores (B+A) >>1 in somereg_c. This method also avoids the possibility of saturation in the intermediate result that could occur in the more obvious coding that follows:

```
SUB somereg_a, somereg_b > somereg_c
AS #-1, somereg_c > some_other_reg
```

BIOZ is a sample-rate synchronization instruction which conditionally sets the BIOZ bit in the HOST_CNTL register. A high BIOZ bit suspends the chip. The BIOZ bit is set when this instruction is encountered and the IOZ status bit of the HOST_CNTL register is found low. The chip will stay in a state of suspension while the IOZ status bit remains low. Execution will resume when the IOZ status bit is set, for then the BIOZ bit will be automatically cleared. If the IOZ status bit is set before the BIOZ instruction is encountered, then the BIOZ bit cannot go high, hence no suspension will occur. The IOZ status bit is automatically set by a low to high transition of the IOZ input pin while the IOZ_EN bit in the HOST_CNTL register is high. The IOZ pin is an **a**synchronous input which is synchronized to the instruction cycle by the synchronization interface. It is most often tied to the sample rate signal called LRCLK **(**which is also allowed to be **a**synchronous with regard to serial data transfer I/O**)**.

Taking the IOZ_EN bit low will disable the detection of subsequent IOZ input pin transitions, hence disabling the subsequent setting of the IOZ status bit, and will therefore hold the chip in BIOZ suspension **indefinite**ly **(**assuming that a BIOZ instruction was encountered in the running program**)**. If IOZ_EN goes low after the IOZ status bit was set, the subsequent BIOZ instruction will observe a high IOZ status bit. Hence, indefinite suspension will occur the next time around. When IOZ_EN is again set high, the next low to high transition on the IOZ pin sets the IOZ status bit.

All this can be more easily understood by observing the schematic below:

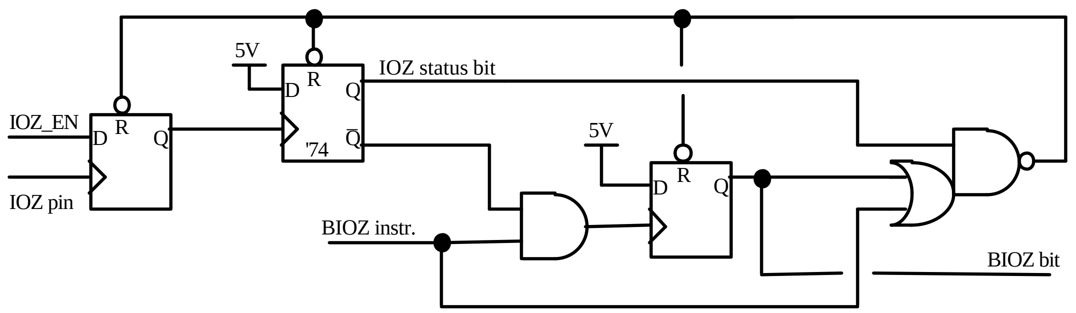


Figure 4

The IOZ status bit appears in the CCR, the HOST_CNTL interface register, and in HOST_CNTL_SPR. The BIOZ instruction automatically monitors the bit which appears in the CCR and which is updated on a per instruction basis. Unlike the IFLAG pin, nowhere does an image of the IOZ input pin exist. The system host has read/write access to the IOZ status bit, the IOZ_EN bit, and the BIOZ bit through the HOST_CNTL interface register.

In order to allow run-time host access to internal registers, the ALU will execute HOST instructions but only **while** in suspension; as it does during chip halt **(**see the section on Halt and Suspension States**)**. The example program and its Equivalent Pseudo-code shows that suspension is not guaranteed by the mere execution of a BIOZ instruction. We see that the instruction cycle corresponding to the BIOZ instruction itself is used to perform one internal register refresh. Keep in mind that all instructions in the example program are executed only once.

```
example program: NOP      BIOZ      NOP
                 next  queued  instr.
                 2nd   queued  instr.
```

Equivalent Pseudo-code:

```
example program: NOP        MOV  REF > REF      NOP    /* BIOZ bit is always clear
coming in. */
                           /* These B and C operands are supplied by the assembler. */
               if (IOZ status bit)                /* Set on low to high transition of IOZ
pin. */
                       clear IOZ status bit
               else
                       set BIOZ bit
               execute   next queued instr.    /* Execution latency */
               if (IOZ status bit) {
                       clear BIOZ bit
                       clear IOZ status bit   /* Zero it for detection of next sample
period. */
               }
               while (BIOZ bit) {                /* Suspension. */
                       HOST                      /* Auto refresh or host access. */
                       /* B and C operands for HOST are supplied by the  hardware as REF. */
                       if (IOZ status bit) {     /* Set on low to high transition of IOZ
pin. */
                               clear BIOZ bit
                               clear IOZ status bit        /* Zero for start of next
period. */
                       }
               }
               execute   2nd  queued  instr.      /* Resume program. */
```

The BIOZ instruction does not always cause chip suspension. Suspension will not occur when a program main loop equals or exceeds the sample period. BIOZ has an instruction cycle execution latency of 1. If the chip shall enter into a state of suspension then there is a one instruction cycle latency before so. Therefore the instruction line **queued** for execution following BIOZ (which is not necessarily the instruction line at PC value + 1 (See the section on Instruction Cycle Execution Latency.)) will execute before the suspension becomes effective. When suspension terminates, execution resumes with the *2nd instr*uction line queued for execution following BIOZ.

PROGRAMMER NOTE: If the number of instruction lines in every program main loop is precisely equal to the sample period, this means that the BIOZ instruction will never allow host access because the chip never goes into suspension. In this case, the programmer must include HOST instructions elsewhere in the code if host access is desired at run-time.
If an occasional program loop exceeds the sample period (which is allowed by the chip synchronization interface), then BIOZ will not allow host access on that particular loop for the same reason.

The BIOZ instruction always performs at least one refresh of internal registers as evidenced by the first line of Pseudo-code. When the application program spends time in suspension (in the while-loop inside the BIOZ instruction) more internal registers are refreshed. The HALT_REF_DIS bit in the HARD_CONF SPR must be low during suspension (or halt) for the HOST instruction in the while-loop to perform refresh when no host access is pending. During halt or suspension, the MAC unit can be forced to do refresh, effectively doubling the refresh rate, if the HALT_MAC_REF bit in the HARD_CONF SPR is set. This doubling comes at the expense of the loss of the contents of the MACRL SPR **(**the low-order MAC unit result from the queued instruction line executed prior to halt or suspension**)**.

ASSEMBLER NOTE: It is critical that BIOZ be used be perform refresh by setting the B and C operands to the SPR called REF for the MOV operation. The A operand is ZERO.

BREV performs a classical bit reverse operation on the full 24 bits of the B operand. This instruction is used in a radix-2 FFT and can be used for FFTs of any binary size up to 2**24. The usage in a loop is the same as for DREV. The operation on the bits is as follows:

23 > 0
22 > 1
21 > 2
.
.
.
1 > 22
0 > 23

The BREV instruction can also be used to reverse the order of bits emerging or received from the serial interface data lines; e.g.,

```
BREV  some_reg > SER2L       /* instead of  MOV */
```

ASSEMBLER NOTE: The A operand of this instruction is a don't-care. For consistency the ZERO SPR should be used for the A operand.

DREV performs a digit reverse operation on the full 24 bits of the B operand. The operation on the bits is as follows:

23 > 1
22 > 0

21 > 3
20 > 2

19 > 5
18 > 4
.
.
.
1 > 23
0 > 22

This instruction is used in a radix-4 FFT of any quaternary size up to 2**24. Example of usage in a loop:

```
        ADDV #(2**24)/N, index              ! where N = size of FFT
        DREV index > drev_index              ! index = 0  ->  N-1
```

ASSEMBLER NOTE: The A operand of this instruction is a don't-care. For consistency the ZERO SPR should be used for the A operand.

HOST provides host access to GPR/SPR/AOR via the host interface registers. The HOST_GPR_PEND bit of the HOST_GPR_CNTL interface register is automatically checked by the ESP2 for a host access request. If an access is pending, one MOV instruction is automatically executed which transfers data between the HOST_GPR_DATA SPR and internal register memory in the direction specified by the HOST_GPR_RW\ bit of the HOST_GPR_CNTL interface register. The MOV executes using normal ALU timing and the standard Y and Z busses. The address of the GPR/AOR/SPR to be accessed resides in the HOST_GPR_ADDR1,0 interface registers. When the MOV is complete the HOST_GPR_PEND bit will automatically clear.

When no host access is pending, this instruction performs refresh of one of the internal GPR/AOR registers.

The BIOZ instruction contains the HOST instruction within it. These two instructions provide the only mechanism for host access to internal registers at run-time. But if the number of instruction lines in every program main loop equals or exceeds the sample period, this means that the BIOZ instruction will <u>never</u> allow host access because the chip never goes into suspension. In that case, the programmer must include HOST instructions elsewhere in the code if host access is desired at run-time.

ASSEMBLER NOTE: If **no** host access is pending, one MOV of B to C automatically occurs in the ALU along the normal data paths using the operands found in the instruction. The B and C operands, then, are both the REF SPR to perform refresh of GPRs/AORs automatically when no host access is pending. (See also the NOP pseudo instruction.)
The A operand of this instruction is a don't-care. For consistency the ZERO SPR is used for operand A.

Jcc Conditional Jump moves the value in the B operand **field** of the instruction into the PC. The A operand **field** of the instruction holds a *condition* mask which controls conditional execution of the instruction by always unconditionally preloading the CMR whenever this instruction is encountered. There is a 1 instruction cycle latency before the PC is modified, therefore the instruction line queued for execution following Jcc is always executed before the jump is made. Conditional execution applies to all instructions. Conditional jumps are performed by conditionally executing the Jcc instruction. If the skip bit is not set for the ALU **(**i.e., no curly braces in the assembler syntax**)**, the jump is always taken regardless of the preloaded mask. **(**See the description of the Conditional Execution Mechanism.**)** *condition* can be GT, GTE, EQ, LT, etc.

ASSEMBLER NOTE: A MOV operation executes along the normal ALU data path. The C operand should be assigned as the ZERO SPR to insure that the MOV is benign. The moves to the CMR and PC use **special** reduced-latency hardware apart from the normal ALU data path. The A operand field of this instruction is set to the ALW Condition Mask if **not** specified.

PROGRAMMER NOTE. The Jcc mnemonic is not recognized by the assembler. Use instead:

```
{JMP  label,  condition > CMR}
  JMP  label                              ! By default the assembler supplies ALW as the  condition
```

We strongly discourage the practice of modifying the PC from the ALU using any instruction ***not*** *from the JMP class (Jcc, JScc, RScc) nor REPT, such as MOV. The reason for this is that a sequence of instructions involving MOV to PC and one of the latent instructions, such as BIOZ for example, can have indeterminate outcome. In that example the desired PC value, as vector, varies dependent upon precisely how many instruction cycles the chip remains in suspension.*

JScc Conditional Jump to Subroutine executes like Jcc, except the value of the PC for the second instruction line **queued** for execution following JScc **(**which is not necessarily the instruction at PC value + 2 (See the section on Instruction Cycle Execution Latency.) **)** is pushed onto a 4-deep hardware stack. The B operand **field** holds the new value of the PC, while the A operand **field** holds a *condition* mask which always unconditionally preloads the CMR whenever this instruction is encountered. Like Jcc, the instruction cycle execution latency of JScc is 1, so the instruction line queued for execution after JScc is always executed before the subroutine is entered.

ASSEMBLER NOTE: A MOV operation executes along the normal ALU data path. The C operand should be assigned as the ZERO SPR to insure that the MOV is benign. The moves to the CMR and PC use **special** reduced-latency hardware apart from the normal ALU data path. The A operand field of this instruction is set to the ALW Condition Mask if **not** specified.
Although the PCSTACK0,1,2,3 SPRs are writable by the MAC unit, a hazard will occur if the MAC unit is writing to one of these registers in the same instruction line as a JScc or RScc instruction. It is desirable to detect these events and prevent the programmer from doing this.

PROGRAMMER NOTE. The JScc mnemonic is not recognized by the assembler.

Use instead:

```
{JS  label,  condition > CMR}          ! Conditional.
  JS  label                             ! By default the assembler supplies ALW as the  condition
```

Unlike RScc, there is no programmer specified movement from B to the C operand.

Example of stack management:

```
CODE  /* push onto PCSTACK */
        NOP    JS  here
        NOP    MOV  #some_PC_value > PCSTACK0
here:   NOP                                           /* any MAC/ALU/AGEN
instruction */
```

LIM is a special operation for checking a ramping value to a upper or lower limit. This instruction is often used for envelope generation. It accepts a limit value as the A operand and a ramping value as the B operand. The instruction also looks at the NA Condition flag to determine the ramp direction. The NA flag in the CCR holds the sign of the A operand from the <u>previous queued</u> ALU operation.

The instruction executes as follows:

```
if (NA == 0)              /* If the previous A operand is greater than or equal to zero (POS)
... */
        MIN  A, B > C
else
        MAX  A, B > C
```

This instruction is typically used, for example, in a two instruction block as follows:

```
ADD  increment, current_value > current_value
LIM    limit,     current_value > current_value
```

The ADD instruction in this example will set the NA flag with the sign of the increment, which is the direction of the ramp.

PROGRAMMER NOTES:

Since conditionally executed instructions never modify the CCR, the LIM instruction in the example above will not work properly if the preceding ADD instruction is conditionally executed {}.

To conditionally saturate to arbitrary +/- values, use the individual MAX and MIN instructions.

The LIM instruction can be used to implement the Signum function **(**sgn()**)**. In the following applications the negative Signum function is produced:

```
                TEST  A
                LIM  #$7FFFFF, #-$7FFFFF > C          ! no zero produced
```

With another instruction, we can get the zero.

```
                 IF  NZ
                 TEST  A > C                           ! zero case
                {LIM  #$7FFFFF, #-$7FFFFF > C}
```

LS performs a logical shift of B by the contents of A using destination C. The shift amount is restricted to the range of +8 to -8 bits. Zero will be shifted into the LSB or MSB depending on the direction of shift. No saturation will occur. (See the LSH pseudo.)

LSDH Logical Shift Double High performs a double precision logical shift using the B operand as the high word and the A operand as the low word of a 48-bit input. The shift amount for this operation comes from the ALU_SHIFT SPR and its range is restricted to +8 through -8 bits as in the LS instruction. Zeros enter vacated bits in left and right shifts of the 48-bit double word. The result is the high 24-bit word of the 48-bit unsaturated shift output.

Application example (double precision rotate left):

LSDH high, low > high
LSDH low, high > low

ASSEMBLER NOTE: **Switch the operands** on this instruction so that LSDH gpr1,gpr2 gets assembled as gpr2 being the A operand and gpr1 being the B operand. This allows the two halves of a double precision word to appear in proper order. In this particular example, gpr2 is the destination (the C operand).

PROGRAMMER NOTE: <u>See the assembler note above</u>. This switch of the operands hobbles indirection. To indirect on gpr1 the programmer must write to INDIRB, and vice-versa. The programmer must always use the verbose form of the instruction explicitly declaring the destination to successfully implement indirection.

LSDL Logical Shift Double Low performs a double precision logical shift using the B operand as the high word and the A operand as the low word of a 48-bit input. The shift amount for this operation comes from the ALU_SHIFT SPR and its range is restricted to +8 through -8 bits as in the LS instruction. Zeros enter vacated bits in left and right shifts of the 48-bit double word. The result is the low 24-bit word of the 48-bit unsaturated shift output.

Application example (double precision rotate right):

LSDL high, low > low
LSDL low, high > high

ASSEMBLER NOTE: **Switch the operands** on this instruction so that LSDL gpr1,gpr2 gets assembled as gpr2 being the A operand and gpr1 being the B operand. This allows the two halves of a double precision word to appear in proper order. In this particular example, gpr2 is the destination (the C operand).

PROGRAMMER NOTE: <u>See the assembler note above</u>. This switch of the operands hobbles indirection. To indirect on gpr1 the programmer must write to INDIRB, and vice-versa. The programmer must always use the verbose form of the instruction explicitly declaring the destination to successfully implement indirection.

MAX takes the greater in two's complement of the A and B operands:

C = Maximum(A, B)

The comparison is performed as C = A - B.

ex.: To eliminate the $800000 code in a signal for purposes of symmetry,

```
   MAX  #$800001, some_signal > some_signal
```

PROGRAMMER NOTE: The state of the flags in the CCR upon equality are consistent with the SUB instruction.

MIN takes the lesser in two's complement of the A and B operands:

C = Minimum(A,B)

The comparison is performed as C = A - B.

ex.: To conditionally saturate to arbitrary high and low limits,

```
     MIN    upper_limit, value         !destination is 'value'
     MAX  lower_limit, value
```

PROGRAMMER NOTE: ditto

MOV performs the fundamental data movement of B to C, as in:

MOV B > C

There is no alteration of the CMR as in MOVcc.

We strongly discourage the practice of modifying the PC from the ALU using any instruction ***not*** *from the* ***JMP class*** *(Jcc, JScc, RScc) nor REPT.*

ASSEMBLER NOTE: The A operand of this instruction is a don't-care. For consistency the ZERO SPR should be used for operand A.

MOVcc Conditional move executes like MOV but the A operand **field** holds a *condition* mask which always unconditionally preloads the CMR whenever this instruction is encountered. If this instruction is conditionally executed {}, then *condition* will be used in the decision to execute or not. Like the other cc-class instructions, *condition* also applies to all conditionally executed operations in the other function units appearing on the same program line as MOVcc and on all subsequent queued lines until another *condition* is preloaded.

PROGRAMMER NOTE: Specifying CMR as the destination (the C operand) of the MOVcc will overwrite the load of the CMR from the A operand field. The CMR used to conditionally execute the MOVcc in this unusual usage is the one from the A operand field.

PROGRAMMER NOTE: MOVcc is not recognized by the assembler. Use instead:

```
  MOV B > C,  condition > CMR
{MOV B > C,  condition > CMR}          ! curly braces denote a  conditionally executed
instruction
  MOV B > C                            ! assembler substitutes MOV instruction
```

ASSEMBLER NOTE: The move to the CMR is always unconditional and uses **special** reduced-latency hardware apart from the normal ALU data path. If no *condition* mask is specified, substitute the MOV instruction.

OR performs the bit-wise logical OR of the two 24-bit operands:

C = A | B

RECT performs the operation:

```
        if (B < 0)   C = A - B          /* with conditional saturation */
        else         C = B
```

Two special cases of this instruction exist: If A = 0 the result is the absolute value of B (full wave rectification; see ABS pseudo). If A = B the result is B if B is positive, or 0 if B is negative (half wave rectification; see HWR pseudo). Regarding the CCR, the 'result' of this instruction is the C operand.

REPT This instruction provides a low-overhead looping mechanism through which a *block* of MAC/ALU/AGEN instructions can be made to **repeat** a specified number of times. This means that the block is always executed at least once. The minimum number of instruction lines constituting the block is one.
The instruction's A operand **field** always represents the PC value of the last instruction line of the block. The B operand field or the B operand represents the number of times to repeat the block. This duality gives rise to two forms of the REPT instruction.

Three SPRs play a role in this looping mechanism: The REPT instruction (both forms) always loads the value in the A operand field into the REPT_END SPR. REPT always automatically loads the value of the PC for the next instruction line queued for execution **(**which is not necessarily the instruction line at PC value + 1, so the block can be disjunct. See the section on Instruction Cycle Execution Latency.**)** into the REPT_ST SPR. REPT (first form) loads the value of the B operand **field** into the REPT_CNT SPR. A nonzero value in REPT_CNT causes a jump to the PC value contained in REPT_ST **whenever** the instruction at the PC value contained in the REPT_END SPR is executed. REPT_CNT post-decrements at the end of each pass through the block if it is nonzero.

The hardware for automatically setting up a loop is separate from the normal ALU data path. The normal data path executes a MOV B > C during this instruction, however. To make this MOV benign, the C operand is the ZERO SPR. The instruction then has the following first form:

first_form: REPT last_instruction, *countm1*

```
/* REPT equivalent Pseudo-code */
REPT_CNT = countm1,  REPT_ST = start_block,  REPT_END = last_instruction;
/*  countm1 is a constant expression stored in the B operand field */
do {start_block:          MAC/ALU/AGEN instr.;
                                :
      last_instruction:     MAC/ALU/AGEN instr.} while (REPT_CNT--);
```

This first form of the REPT instruction is useful for looping 1024 times and less, and is recommended for repeating an instruction block as short as one instruction in length. last_instruction is the PC value of the last instruction of the block and gets loaded into the REPT_END SPR via the A operand **field** of the instruction. The constant expression, *countm1*, gets loaded into the REPT_CNT SPR via the B operand **field** of the instruction; GPRs are not allocated to hold either last_instruction nor *countm1*. *countm1* is the number of times to repeat the block. *countm1* is unsigned, representing the number of loops minus 1, and so the assembler allows a *countm1* of zero. This first form of the instruction always executes a block of any length at least once.

ASSEMBLER NOTE: The C operand is ZERO, for consistency, in the first form of this instruction.

For the REPT instruction, the normal ALU data path movement of B to C occurs after the REPT_CNT SPR is loaded from the B operand field. Knowing this, the ALU data path MOV operation can be employed to move a repeat count value from the contents of some chip register (GPR/AOR/SPR) into the REPT_CNT SPR. The second form of the instruction is then:

second_form: REPT last_instruction, *countm1_gpr* > REPT_CNT

```
/* REPT equivalent Pseudo-code, second form */
REPT_CNT = address of  countm1_gpr,  REPT_ST = start_block,  REPT_END = last_instruction;
do {start_block:       MAC/ALU/AGEN instr.,     if (first loop) REPT_CNT =  countm1_gpr; /* transparent */
                       MAC/ALU/AGEN instr.;
                                 :
      last_instruction: MAC/ALU/AGEN instr.                           } while (REPT_CNT--);
```

This second syntax for the REPT instruction allows counts greater than 1024 and allows computed loop counts. In this second form of the REPT instruction, *countm1_gpr* is a register (GPR/AOR/SPR) containing an unsigned value specifying the number of times to repeat a block of code. **(***countm1_gpr* holds the number of loops minus 1 and cannot be interpreted, by the hardware, to hold a negative value.**)** So the sequence of events first moves the register address of *countm1_gpr* into the SPR, REPT_CNT, via the B operand field (as in the first form). A short time later, the contents of *countm1_gpr* (the B operand) overwrites the contents of REPT_CNT via the normal ALU data path because REPT_CNT is the C operand in this form.

ASSEMBLER and PROGRAMMER NOTE for second form:
Because of pipeline latency issues here, although the REPT_CNT SPR will get the contents of *countm1_gpr* on the next instruction cycle, the REPT looping mechanism will not be able to use it for two instruction cycles. This is fine if the block to be repeated is at least two instructions long. But if the block is only 1 instruction in length, the loop will execute correctly if and only if the **address** of *countm1_gpr* is nonzero, and the contents of *countm1_gpr* is the number of repeats minus one **(**this is the same as saying the number of loops minus two**)**! The assembler will check that the address of *countm1_gpr* is nonzero.

Using the second form of the REPT instruction, the block of instructions will loop at least once if the block is two or more instructions in length, but it will loop at least twice if the block is only one instruction in length. **(For this reason, we do not recommend the use of the second form of REPT for one-line loops.** See the section on Latent Instructions.**)** Therefore, if the block length is two instructions or more, *countm1_gpr* may rightly hold the value zero indicating 1 loop, no repeat.

Jumps and subroutines (JScc, RScc, or Jcc) are allowed within a loop, but these instructions must not immediately precede the last instruction of the repeated block. This hazard is observed by the assembler in the contiguous case.
One must also be careful on a jump which is intended to abort the execution of the loop. The REPT looping mechanism will remain functional as long as REPT_CNT is nonzero. That mechanism will attempt to continue the loop if the last instruction of the last repeated block is reached by any means. So to be safe, zero the REPT_CNT SPR using the ALU or MAC unit pseudo instruction called EXIT before aborting loop execution.
Likewise, if it becomes necessary to terminate a loop at the end of the current

block, this can also be done via the pseudo instruction, EXIT. An EXIT must occur at least one instruction prior to the last instruction of the block in order for the loop to stop at the end of the current block.

Because there is only one set of REPT_ST, REPT_END, and REPT_CNT registers, multiple REPT instructions cannot be nested. Since these registers are SPRs, it is possible to set up loops manually by writing directly into these registers.

ASSEMBLER NOTE: Since the REPT_CNT, REPT_END, and REPT_ST SPRs are written as a result of REPT instruction execution, a hazard will occur if one of these registers is used as the destination of a MAC unit operation on the same instruction line as the REPT, or on the last line of a repeated block.

RScc Conditional Return from Subroutine pops the value of the PC from the 4-deep hardware stack, PCSTACK0,1,2,3. The A operand **field** of the instruction holds a *condition* mask which controls conditional execution by always unconditionally preloading the CMR whenever this instruction is encountered. The RScc instruction always moves the B operand to the C operand just as in a MOVcc instruction. Since RScc modifies the PC, (as do the other members of the JMP class, Jcc and JScc) its instruction cycle execution latency is 1. So, the instruction line queued for execution following RScc will always execute as the last instruction line of the subroutine.

PROGRAMMER NOTE. RScc is not recognized by the assembler. Use instead:

```
{RS B > C, condition > CMR}      ! curly braces denote a conditionally executed instruction.
  RS B > C                        ! by default the assembler inserts the ALW condition.
{RS, condition > CMR}             ! Note the required comma in this syntax.
  RS                              ! by default the assembler inserts the ALW condition.
```

PROGRAMMER NOTES:
Specifying CCR as the destination (the C operand) of the RScc instruction is a very good way to restore the state of the Condition Codes before the return home.
Specifying PCSTACK0,1,2, or 3 as the destination (the C operand) of the RScc will replace the specified PCSTACK SPR contents after the pop. The original contents of PCSTACK0 are always used to load the PC for the return to the calling routine, in any case.
Specifying CMR as the destination (the C operand) of the RScc will overwrite the load of the CMR from the A operand field. The CMR used to conditionally execute the return in this unusual circumstance is the one in the A operand field.

Example of stack management:

```
CODE  /* pop PCSTACK */
        NOP    MOV  #here > PCSTACK0
        NOP    RS
        NOP                                /* any MAC/ALU/AGEN instructions */
here:   NOP
```

ASSEMBLER NOTE: Although the PCSTACK SPRs are writable by the MAC unit, a hazard will occur if the MAC unit is writing to one of these registers on the same instruction line as a JScc or RScc instruction. It is desirable to detect these events and prevent the programmer from doing this.
A MOV operation always executes along the normal ALU data path. If the programmer does **not** specify B and C operands, they both should be the ZERO SPR for consistency. The moves to the CMR and PC use **special** reduced-latency hardware apart from the normal ALU data path. The A operand field of this instruction is set to the ALW Condition Mask if **not** specified.

SUB is a 2's complement saturating subtraction operator which yields the difference of two signals, analogous to ADD. It performs the operation:

$$C = B - A$$

The statement SUB A, B > C is read: '*subtract A from B and put the result in C* '.

SUBB Subtract with borrow is a 2's complement saturating subtraction operator which yields the difference of two operands. It departs from SUB in that the carry bit of the CCR is subtracted from the LSB position. This operator is useful for double precision arithmetic. It performs the operation:

C = B - A - carry

When the SUBB instruction is used in conjunction with a preceding SUBV to perform double precision arithmetic, the SUBB operation can saturate the high 24-bit word of the 48-bit result. Since the low 24-bit word was computed in the preceding SUBV operation, its value will not be conditionally saturated. The low word of the result can be adjusted by the following conditional operation:

```
  SUBV  a_low, b_low > c_low
  SUBB  a_high, b_high > c_high
  IF OV
{XOR    MINUS1, c_high > c_low}
```

(See the section on ALU Pseudo Instructions for the IF pseudo, and see the section on Setting Condition Codes for information on the V flag.)

PROGRAMMER NOTE: Since conditionally executed instructions never modify the Condition Code Register, a double precision subtraction using the SUBB instruction will not execute properly if the preceding SUBV is conditionally executed {}.

SUBREV is a 2's complement saturating subtraction operator which yields the difference of two signals, just like SUB. But the position of the source operands with respect to the minus sign is reversed. It performs the operation:

C = A - B

SUBV is an unsaturating subtraction operator, analogous to ADDV. Like SUB, it performs the operation:

C = B - A

SUBV (like ADDV) can be used for double precision math and unsigned address arithmetic. (See SUBB instruction.)

XOR performs the bit-wise logical exclusive OR of the two 24-bit operands:

C = A ^ B

Example:
The following code exchanges the contents of two registers without requiring a third register:

```
XOR  some_reg1, some_reg2
  XOR  some_reg2, some_reg1
XOR  some_reg1, some_reg2
```

3.2. ALU Pseudo Instructions

Pseudo instructions make use of the fundamental instructions in an unusual or limited way. Since ALU pseudo instructions are assembled as one of the fundamental ALU instructions, they affect the CCR just the same.

ABS B > C

performs the operation:

RECT ZERO, B > C

This is the absolute value operation and it is a special case of the RECT instruction is which the A operand is the ZERO SPR.

ASSEMBLER NOTE: The ABS instruction requires that the A operand be the ZERO SPR.

ASH B >>n > C

performs the operation:

AS #-n, B > C

If a computed shift is required, use the primitive AS instruction. The syntax here is the same as for the MAC unit pseudo.

```
ASH  some_reg >>n       ! some_reg is also destination.
ASH  some_reg <<n       ! shift left for positive  n
```

n is a constant expression (no # sign allowed); -8 <= n <= 8 bits.

ASSEMBLER NOTE: The syntax above is equivalent to AS A, B > C where the A operand holds the shift amount. A-operand negative means shift right; this is derived from exponentiation of 2.

BIOZNORFSH
BIOZNORFSH B > C
This pseudo performs a BIOZ instruction with the exception that the B and C operands of the indicated MOV operation **(**see BIOZ, Equivalent Pseudo-code**)** are **not** set to the REF SPR. This instruction **always** moves B to C. The first form of the instruction uses the ZERO SPR as the defaults for B and C.

The standard BIOZ instruction uses MOV REF > REF to accomplish at least one refresh of internal registers. The BIOZNORFSH pseudo does not use the REF SPR thereby inhibiting only that first refresh. To inhibit subsequent refresh **(**which is **not** desirable, by the way**)** during a BIOZ (while-loop) suspension, the HALT_REF_DIS bit in the HARD_CONF SPR **must** be set. That bit controls internal register refresh during the halt and suspension states of the chip. In the suspension state, it disables refresh when set by effectively changing the HOST instruction, embedded within the BIOZ instruction, to a HOSTNORFSH. The HALT_REF_DIS bit overrides the HALT_MAC_REF bit, but it can **never** override the usage of the REF SPR, used as an instruction source <u>and</u> destination, as a means to effect refresh. See the description of Internal Register Refresh for more details on the use of this bit.

ASSEMBLER NOTE: If the programmer does not specify B and C operands, they should be the ZERO SPR for consistency. The A operand of this instruction is a don't-care, but it should be the ZERO SPR for consistency.

CLR C

performs the operation:

MOV ZERO > C

This is a shorthand way to clear a register.

CMP A, B
CMP A, B > C
is a pseudo instruction created from the SUBREV instruction using the destination ZERO. It performs the **comparison** operation:

ZERO = A - B

The assembler supplies the destination ZERO when C is not specified. Used in this manner, the CMP pseudo constitutes an arithmetic test which results in a setting of the CCR that is used in subsequent instruction lines by conditionally executed instructions.
The statement, CMP A, B is read: '*compare A to B and set the CCR accordingly*'.
CMP A, B > C performs the same operation but has a destination that is not ZERO.

DBL some_reg > some_other_reg
DBL some_reg

This pseudo instruction doubles a register's contents.
The second form performs the operation:

```
ADD some_reg, some_reg > some_reg
```

DEC some_reg = SUB ONE, some_reg > some_reg
DECV some_reg = SUBV ONE, some_reg > some_reg

DIFF B > C

performs the operation:

C = $7FFFFF - B

ASSEMBLER NOTE: This instruction requires the constant $7FFFFF be stored in a GPR or AOR for use as the A operand. It also assumes that the ALU SUBREV instruction is used.

EXIT

This ALU pseudo instruction is used to terminate a repeating block of code, instigated by the REPT instruction, <u>at the end</u> of the block. Although the ALU EXIT pseudo instruction may be placed anywhere within a repeated block of instructions, it will <u>not</u> terminate the current block if placed on the last line of the block; in that case it will, however, terminate the block repeat after the next time around. The operation is:

REPT ZERO, 0 > ZERO

(The A operand is a don't-care, the B operand **field** is 0.**)**

HALT

ESP2 program halt of the chip can be achieved by the operation:

OR #$2, HOST_CNTL_SPR > HOST_CNTL_SPR

This instruction sets the ESP_HALT bit in HOST_CNTL_SPR without affecting the remaining bits of the register. (See the section on Halting the Chip.) To successfully halt ESP2 under program control, the ESP_HALT_EN bit of the HOST_CNTL interface register must also be set, otherwise HALT is ignored.

The chip can be taken out of the halt state by having the system host clear the ESP_HALT and HOST_HALT bits of the HOST_CNTL interface register.

There is an execution latency of 1 instruction cycle for HALT to take effect, therefore the
instruction line queued for execution following HALT will execute before the chip enters into the halt state.

ASSEMBLER NOTE: To use this instruction, a GPR must be allocated to hold the value $2.

HALVE B > C = AS #-1, B > C

HOSTNORFSH

HOSTNORFSH B > C

This pseudo allows host interaction, as in the HOST instruction, when host access to an internal register is pending. If **no** host access is pending, the instruction performs one MOV B > C where B and C are not the REF SPR. This instruction inhibits the internal register refresh which is normally part of the standard HOST instruction when no host access is pending, by purposely not referencing the REF SPR in the B and C operands. In general, it is not a good idea to inhibit refresh.

ASSEMBLER NOTE: If the programmer does not specify B and C operands, they both should be the ZERO SPR for consistency. The A operand of this instruction is a don't-care, but it should be the ZERO SPR for consistency.

HWR B > C

performs the half-wave rectification operation:

```
if B < 0
        C = A - B = 0
else
        C = B
```

HWR is a special case of the RECT instruction where the A operand equals the B operand:

RECT some_reg, some_reg > some_other_reg.

ASSEMBLER NOTE: The HWR instruction requires that the A operand and the B operand be the same register.

IF *condition* = MOV ZERO > ZERO, *condition* > CMR

{**IF** *condition*} = {MOV ZERO > ZERO, *condition* > CMR}

These pseudo instructions utilize the MOVcc instruction to always unconditionally preload the Condition Mask Register with a *condition* mask. Unlike the CMP pseudo, **IF does not constitute a test** of any sort. In fact, the CCR is largely unaffected by these pseudos. **(**Only the IFLG and IOZ flags of the CCR are updated in the first form.**)** The {second form} is used when it is desired that all the flags remain unmodified in the CCR. Like MOVcc, the new *condition* mask will apply to all conditional operations on the same program line containing the IF, and on all subsequent queued lines until another *condition* is preloaded.

PROGRAMMER NOTE: *condition* is a constant expression. The assembler recognizes EQ, Z, GTE, etc. (see the section on Arithmetic Condition Masking). All cc-class instructions always unconditionally preload the CMR.

ASSEMBLER NOTE: The B and C operands for this instruction should be assembled as the ZERO SPR, while the A operand **field** holds the preload *condition* mask which gets sent to the CMR.

INC	some_reg	= ADD ONE, some_reg > some_reg
INCV	some_reg	= ADDV ONE, some_reg > some_reg

LSH B >>n > C

performs the operation:

LS #-n, B > C

and uses a specified shift amount; -8 <= n <= 8 bits (right) .
If a computed shift is required, use the primitive LS instruction.

```
LSH some_reg >>n        ! some_reg is also destination.
LSH some_reg <<n        ! shift left for positive  n
```

n represents a constant expression (no # sign allowed).

ASSEMBLER NOTE: The syntax above is equivalent to LS A, B > C where the A operand holds the shift amount. A-operand negative means shift right; this is derived from exponentiation of 2.

NEG B > C

performs the operation:

ZERO - B > C

This is a special case of the SUBREV instruction in which the A operand is supplied by the assembler as the ZERO SPR.

NOP

performs a refresh of the internal register indicated by the REFPT SPR. The instruction is assembled as:

MOV REF > REF

This refresh (and the MAC unit RFSH pseudo), when successfully executed (not skipped), cannot be overridden by any means. See the section on GPR and AOR Refresh for more information on the use of the REF SPRs for internal register refresh.

ASSEMBLER NOTE: The A operand of this instruction is a don't-care, but it should be the ZERO SPR for consistency.

NORFSH

This no-operation instruction is assembled as:

MOV ZERO > ZERO

ASSEMBLER NOTE: This NORFSH pseudo requires that the C operand be the ZERO SPR. The A and B operands are don't-cares, but they should be ZERO for consistency.

RFSH identical to NOP pseudo instruction.

ROL some_reg
ROL some_reg > some_other_reg = LSDH some_reg, some_reg > some_other_reg
This is a 24-bit rotate left (having no other bits outside the register involved). The ALU_SHIFT SPR provides the shift amount for the instruction.

ROR some_reg
ROR some_reg > some_other_reg = LSDL some_reg, some_reg > some_other_reg
This is a 24-bit rotate right (having no other bits outside the register involved). The ALU_SHIFT SPR provides the shift amount for the instruction.

TEST A performs: ADD A, ZERO > ZERO
TEST A > C performs: ADD A, ZERO > C
This test sets the flags in the Condition Code Register, appropriate to the arithmetic status of operand A, so that they can be used in subsequent instruction lines by conditionally executed instructions. The second form of this pseudo also moves operand A to some destination.

3.3. Condition Code Register

Associated with the ALU is a Condition Code Register, the CCR, which is used for the conditional execution of instructions. Seven bits (*flags*) from this register reflect the arithmetic status of the chip, particularly the result of the previous queued ALU operation. One bit (IFLG) reflects the state of a pin, while the remaining bit (IOZ) reflects the state of the internal IOZ status bit located in HOST_CNTL_SPR and the HOST_CNTL interface register. The 9 LSBs of this 24-bit CCR are:

bit: 8	7	6	5	4	3	2	1	0
IOZ	IFLG	NB	NA	N	C	V	LT	Z

The CCR contains 5 flags whose states are derived in most ALU operations from basic arithmetic results:
Z (zero result)
C (carry out)
N (negative result)
V (result overflow)
The LT (less than) flag is generated by the logic:

```
LT = V  ^  N'            /* exclusive OR */
```

where N' is the sign of the result before saturation. Having the LT flag allows simple detection of a Less-Than Condition.

The sixth and seventh flags, NA and NB, hold the sign of the A and B operands, respectively, from the previous queued ALU operation.

The eighth flag is IFLG; it is an image of the IFLAG pin signal which is part of the synchronization interface. The IFLAG pin is an asynchronous uncommitted input to the ESP2 which is internally synchronized by the ESP2. The synchronization forces the state of IFLG to update on a per instruction basis along with the ALU's update of the flags 0 through 6.

The nineth flag, IOZ, reflects the state of the internal IOZ status bit. The IOZ status bit is a function of the IOZ input pin signal which is part of the synchronization interface. IOZ is employed by the instruction BIOZ to automatically synchronize a running program to the sample-rate signal at the IOZ pin. (See the BIOZ instruction for more details.) The asynchronous IOZ pin signal is internally synchronized by ESP2 to force the IOZ flag to update on a per instruction basis along with the ALU's update of the remaining flags.

The upper bits of the 24-bit CCR SPR are read as zeros as are all SPRs less than 24 bits in width. The CCR is mapped in the SPR address space and is read-writable by the instructions as is any other SPR. IFLG and IOZ, however, are read-only in the CCR SPR. The IOZ status bit can be manually modified by writing to the HOST_CNTL (host) interface register or to HOST_CNTL_SPR.

PROGRAMMER NOTE:
The programmer may have occasion to restore the chip state after returning from some subroutine. *The programmer should be wary of using the CCR as the destination of any instruction other than an ALU MOV, MOVcc, or RScc* because the outcome would be

unpredictable since many ALU instructions automatically manipulate bits of the CCR. Using the ALU MOV, MOVcc, or RScc to restore the CCR, the CCR bits become active during the next instruction cycle for the purpose of conditional execution. **(**Moving to the CCR from the MAC unit produces a hybrid combination of the MAC unit supplied data with only the automatically updated CCR bits from the ALU; interesting but dangerous. See SPR Hazards.**)**

3.3.1. Setting Condition Codes

Condition Codes of the CCR are generated as follows for each ALU operation:

Table 5. CCR Setting by Instruction

OPERATION	**IOZ**	**IFLG**	**NB**	**NA**	**N**	**C**	**V**	**LT**	**Z**
ADD, ADDV, ADDC	X	X	X	X	X	X	X	X	X
SUB, SUBV, SUBB, SUBREV	X	X	X	X	X	X	X	X	X
AND, OR, XOR	X	X	X	X	X	-	-	-	X
MIN, MAX	X	X	X	X	X	X	X	X	(3)
AVG	X	X	X	X	X	-	X	-	X
RECT	X	X	X	X	X	-	(2)	(2)	X
AMDF	X	X	X	X	(4)	X	X	X	X
AS, ASDH	X	X	X	X	X	(1)	X	-	X
ASDL, LS, LSDH, LSDL	X	X	X	X	X	(1)	-	-	X
BREV, DREV	X	X	X	X	X	-	-	-	X
LIM	X	X	-	-	X	X	X	X	(3)
MOV, HOST, BIOZ, REPT	X	X	-	-	-	-	-	-	-
MOVcc, Jcc, JScc, RScc	X	X	-	-	-	-	-	-	-
{*any instruction*}	-	-	-	-	-	-	-	-	-

N' is defined to be the MSB of the adder/subtractor output prior to saturation, shifting, MUXing, XORing, etc.

X in the NB and NA flags indicates that the flag is set to the sign of the corresponding source operand, two's complement. NB and NA, as other CCR bits, are active for the purpose of conditional execution on the next instruction cycle.

X in the IOZ flag indicates that the flag is set to the value of the IOZ status bit. The IOZ status bit is in turn a function of the IOZ input pin from the synchronization interface.

X in IFLG indicates that the flag is set to the value of the IFLAG input pin from the synchronization interface.

X in the N flag indicates that the flag is set to the sign of the result of the operation, two's complement.

X in the C flag indicates that the flag is set to the value of the carry output of the MSB of the adder/subtractor as in classical 2's complement arithmetic.

X in the V (overflow) flag indicates that the flag is set to the value G ^ N' where G is the guard bit (beyond the MSB) of the adder/subtractor. (The ^ symbol indicates the exclusive OR binary operator.)

X in the LT flag indicates that the flag is set to the value N' ^ V .

X in the Z flag indicates that the flag is set to a 1 if the result of the operation is a zero.

(-) Indicates that this flag is not altered by the operation.

(1) C receives the last bit shifted off the left of the 24-bit result in a left shift, or off the right of the 24-bit result in a right shift (before saturation is performed in the cases of arithmetic shifts).
If the shift amount is zero the C flag gets the bit to the immediate left of the MSB:

In the case of AS or ASDH this would be a sign extension of the MSB.
In the case of LS or LSDH this would be zero.
In the case of ASDL or LSDL this is the LSB of the B operand.

(2) V and LT are modified based on the result of the subtraction if the B operand is less than zero. If the B operand is greater than or equal to zero, the subtraction is not performed, therefore the V and LT flags will be cleared.
(3) Z asserted based on the result of the comparison not the result of the instruction.
(4) The result of an AMDF operation always has a positive sign. Therefore, the N flag for this operation is set to N' as defined above, which is indicative of whether the exclusive OR operation was performed.

Only the ALU automatically sets the CCR. CCR bits will be active for the conditional execution of ALU, and/or MAC unit, and/or AGEN operations during the next instruction cycle. CCR bits become valid at the end of the ALU instruction **(**see Instruction Cycle Timing diagram**)**; thus, they are available (as source operand) for explicit reading by the next queued ALU instruction, but by the second queued MAC unit instruction.

If the ALU instruction is conditionally executed {}, the CCR will remain unchanged from the previous queued instruction.

3.3.2. Conditional Execution Mechanism

In addition to the program control instructions (e.g., the JMP class), the ESP2 chip has a mechanism that allows individual function unit operations to be conditionally executed. Any or all instruction fields may be marked by the programmer as conditionally executed {}, meaning that the corresponding function unit's operation will only be executed when the ALU's Condition Code Register (CCR) is in a specified state. That state constitutes the skip Condition, and is specified by loading a particular mask value into the Condition Mask Register (CMR) under program control. In this manner, the individual MAC unit, and/or ALU, and/or AGEN instructions can be conditionally executed according to a variety of ALU Conditions. The Conditions are formulated from combinations of the nine CCR flags:
IOZ, IFLG, NB, NA, N, C, V, LT, and Z.

All ESP2 instructions execute in one instruction cycle whether or not conditionally executed. When the Condition Mask and Condition Code Register do **not** correspond, a function unit instruction having its skip bit set {} will have its destination inhibited and **all** its functionality disabled (with caveat under Instructions Not Skippable). The ALU, MAC unit, and AGEN each have their own skip bit so this mechanism can be applied independently to each.

Recall that only the ALU automatically sets the CCR. <u>ALU instructions which are conditionally executed never set the CCR, regardless of whether they were executed or not</u>. This provision allows for the CCR to be set once before a block of conditionally executed instructions. Then, that CCR is automatically held the same across the entire length of the block. In this way, an entire block of instructions may be conditionally executed based on the same Condition. <u>The Condition Mask Register (CMR) can be initialized at the beginning of the block, but it is unconditionally subject to change if a cc-class instruction is encountered</u>. To keep the CMR the same across the block, any cc-class instruction within the block must reload the CMR with the desired mask. **(**Note that if a block of MAC unit and/or AGEN instruction fields are conditionally executed while the corresponding block of ALU instructions is not, then the CCR will be changing on a per instruction basis, but in accordance with the specific instructions executed in the ALU.**)**

To detect a skip Condition during the instruction cycle of any conditionally executed instruction field, the following logic is performed:

$$\text{skip}\backslash \;=\; \text{NOT}_M \wedge \mathbf{(}(\text{NB}\&\text{NB}_M)|(\text{NA}\&\text{NA}_M)|(\text{N}\&\text{N}_M)|(\text{Z}\&\text{Z}_M)|(\text{V}\&\text{V}_M)|(\text{C}\&\text{C}_M)|(\text{LT}\&\text{LT}_M)|(\text{IFLG}\&\text{IFLG}_M)|(\text{IOZ}\&\text{IOZ}_M)\mathbf{)}$$

The subscript M in the skip \ equation indicates the corresponding bit of the Condition Mask Register. Each Mask bit in the CMR (except NOT) is ANDed with the corresponding bit in the CCR, and the results are ORed together. This result is then inverted (or not) as per the status of the NOT bit, to account for the extra negated cases. If the final result is false (equal to 0) then the current instruction field is skipped; that is to say, the field becomes a NOP in the case of the MAC unit and AGEN and a NORFSH in the case of the ALU by disabling the output along any destination buss.

Note that a NEV (execute never) Condition, where instructions will be unconditionally skipped, can be formulated by clearing all Condition Mask bits to 0 except for a 1 in the NOT bit. Likewise, ALW (execute always) can be formulated by clearing all bits in the CMR including NOT.

3.3.3. Instructions Not Skippable

All instructions (including REPT, BIOZ, HOST, and their related pseudos) are conditionally executable {} with few qualifications:

All cc-class instructions (Jcc, JScc, RScc, and MOVcc) always unconditionally preload the CMR whenever these instructions are encountered ({conditional} or not). This action cannot be inhibited. Fortunately, the MOV instruction has two primitive forms.

Blocks of conditionally executed {} ALU instructions warrant consideration regarding the impact upon the conditional execution of several instructions, ADDC, SUBB, LIM, ASDL, which input Condition Code Register bits as part of their computation within the block. **(**See the description of each instruction named for more details.**)** Double precision operations are impacted; the programmer should consider the use of branching instructions instead of conditionally executed instructions.

ASSEMBLER NOTE: Warnings should be issued when the instructions, ADDC, SUBB, LIM, ASDL, are found conditionally executed.

When the SPR called REF is used as a source operand in either the MAC unit or the ALU, the contents of the REFPT SPR are incremented. This action cannot be inhibited through conditional execution. See the section on GPR and AOR Refresh.

3.3.4. Arithmetic Condition Masking

There are 16 commonly used arithmetic Conditions which can be tested based on the 5 CCR flags:
N, C, V, LT, and Z. The programmer loads a mask (having 5 corresponding bits and a NOT bit) into the Condition Mask Register to select the Condition which will cause instructions to be conditionally executed. The mask values for the 16 Conditions are listed in Table 6. Note that mask values are not copies of the bit patterns produced by the CCR when these Conditions exist; they are merely the values that are necessary and sufficient for detecting the desired Condition.
In addition to these Conditions, the flag IFLG can be used for conditional execution based upon the state of the input pin signal, called IFLAG, which it reflects. Similarly, the flag IOZ can be used for conditional execution as it reflects the state of the internal signal called the IOZ status bit (see BIOZ instruction), which is in turn a function of the input pin signal called IOZ.

The CMR can be written by a number of means:
1**)** Special instructions in the ALU always **unconditionally** move the value of the A operand field of the instruction into the CMR; These are the cc-class instructions, consisting of Jcc, JScc, RScc, and MOVcc. When any one of these instructions is encountered ({conditional} or not), a new Condition Mask is preloaded and becomes effective for all conditional operations (MAC, ALU, and AGEN) on the same and subsequent queued instruction lines, until a new Mask is loaded.
2**)** A second method for writing the Condition Mask is to use the ALU or MAC unit to MOV the contents of some register (GPR/SPR/AOR) into the CMR along the normal data paths. Under this circumstance, the new Condition Mask will be effective for all conditional operations on the queued instruction lines following the MOV, until a new Mask is loaded.

PROGRAMMER NOTE: If both function units specify the CMR as their destination operand on the same instruction line, the ALU write to the CMR will supersede the MAC unit write to the CMR. Conditional Execution Latencies of the CMR are discussed in the Software Spec. in the section having that name.

ASSEMBLER NOTE: A hazard will occur if the MAC unit instruction specifies the CMR as a destination on the same instruction line as an ALU operation which is a Jcc, JScc, RScc, or MOVcc. In this case, both function units are trying to write the CMR simultaneously with unpredictable results.

Table 6. Arithmetic Condition Code Masks

Condition	Mnemonic	NOT	IOZ	IFLG	NB	NA	N	C	V	LT	Z	Mask Value
Equal (Zero)	EQ=Z	1	0	0	0	0	0	0	0	0	1	$201
Not Equal (NonZero)	NEQ=NZ	0	0	0	0	0	0	0	0	0	1	$001
Negative	NEG	1	0	0	0	0	1	0	0	0	0	$210
Positive (>= 0)	POS	0	0	0	0	0	1	0	0	0	0	$010
Overflow	OV	1	0	0	0	0	0	0	1	0	0	$204
No Overflow	NV	0	0	0	0	0	0	0	1	0	0	$004
Lower (Carry Set)	LO=CS	1	0	0	0	0	0	1	0	0	0	$208
Higher or Same (Carry	HS=CC	0	0	0	0	0	0	1	0	0	0	$008
Clear)	LS	1	0	0	0	0	0	1	0	0	1	$209
Lower or Same	HI	0	0	0	0	0	0	1	0	0	1	$009
Higher	LT	1	0	0	0	0	0	0	0	1	0	$202
Less Than	GTE	0	0	0	0	0	0	0	0	1	0	$002
Greater Than or Equal	LTE	1	0	0	0	0	0	0	0	1	1	$203
Less Than or Equal	GT	0	0	0	0	0	0	0	0	1	1	$003
Greater Than	ALW	0	0	0	0	0	0	0	0	0	0	$000
Always	NEV	1	0	0	0	0	0	0	0	0	0	$200
Never	IFLG	1	0	1	0	0	0	0	0	0	0	$280
IFLG set	NIFLG	0	0	1	0	0	0	0	0	0	0	$080
IFLG clear	IOZ	1	1	0	0	0	0	0	0	0	0	$300
IOZ status bit set	NIOZ	0	1	0	0	0	0	0	0	0	0	$100
IOZ status bit clear	BNEG	1	0	0	1	0	0	0	0	0	0	$240
B operand negative	BPOS	0	0	0	1	0	0	0	0	0	0	$040
B operand positive	ANEG	1	0	0	0	1	0	0	0	0	0	$220
A operand negative	APOS	0	0	0	0	1	0	0	0	0	0	$020
A operand positive												

HI, LO, HS, LS can be used for unsigned comparisons such as for address arithmetic.

3.4. Instruction Cycle Execution Latency (Latent Instructions)

This advanced topic may be skipped by the novice or the first-time reader.

This category of latency regards the latency of the instructions themselves. All the latent ESP2 instructions are from the ALU. No instruction has an execution latency which exceeds 1 instruction cycle. The **latent instruction** is one whose impact is not effective until the second queued program line following. The reason we study this topic is because we must be able to predict the effect of cascades of instructions, some of which may be latent.

The latent instructions in the ESP2 are: Jcc, JScc, RScc, and BIOZ.
The latent pseudo instructions are: BIOZNORFSH and HALT.

The **JMP class** (Jcc, JScc, RScc) instructions always see the next queued instruction line execute prior to the actual branch; this is a prominent case.
The **HALT class** (HALT, BIOZ, BIOZNORFSH) instructions see the next queued instruction line execute prior to freezing the PC; most all audio programs will employ the BIOZ instruction.

In the following examples, the labels, identified by colons, each denote a PC value corresponding to a particular program line number. Many of the ancillary instructions are NOPs to make the examples simpler, but they generally represent any nonlatent instruction.

CASE: JMP class

An important instruction sequence involves the nonlatent low-overhead loop instruction, REPT, which is not from the JMP class. Specifically, we consider:

```
CODE
I1:     NOP    REPT label2, countm1

I2:     NOP    JMP label1                 /* CMR unconditionally preloaded with ALW */
I3:     NOP    /* executed */
label1: NOP
label2: NOP
```

The outcome is that the REPT_ST SPR receives the value, I2, as expected. The loop spans the program lines I2 through label2. But now consider:

```
CODE
label1: NOP
label2: NOP

        :
        :
I1:     NOP    JMP label1                 /* CMR unconditionally preloaded with ALW */
I2:     NOP    REPT label2, countm1
```

The outcome has changed such that the REPT_ST SPR receives the value, label1. This happens because when REPT is executed, the program line at label1 is queued next in line for execution; this is what is meant by the phrase, ***queued for***

execution. The loop now spans only label1 and label2. This feature permits the use of disjunct REPT blocks.

CASE: HALT class

```
CODE
I1:      NOP      HALT
I2:      NOP      REPT  label1,  countm1_gpr > REPT_CNT

label1: NOP     /* loop */
```

This important exception is for <u>one-line loops</u> using the second form of the REPT instruction. The previous halt state disrupts the normal latency of the load to the REPT_CNT SPR. So our one-line loop rule for the second form of REPT reverts to the rule for the first form, and the concern about the address of *countm1_gpr* disappears. Since this is too difficult to remember, **we recommend using only the first form of REPT for one-line loops** (see REPT instruction description).

ASSEMBLER NOTE: A warning should be issued under the circumstance that a HALT instruction precedes the second form of the REPT instruction used for a one-line loop. The warning states that *countm1_gpr* should **not** contain the number of repeats less one (i.e., **not** the number of loops desired less two); but rather, the number of repeats desired.
If the instruction preceding REPT is BIOZ (or BIOZNORFSH) for this case, the outcome depends upon whether suspension actually occurs. Hence, this alternate warning should recommend <u>not using the construct</u> unless the programmer knows that suspension will always occur or always not occur.

CASE: JMP class, JMP class
Consider the example of <u>two</u> JMP class instructions, one after the other:

```
CODE
I1:      NOP     JMP  label1            /* CMR unconditionally preloaded with ALW */
I2:      NOP     JMP  label2            /* CMR unconditionally preloaded with ALW */
I3:      NOP      /* hurdle 1 */
label1: NOP
I5:      NOP      /* hurdle 2 */
label2: NOP
```

According to the prescribed latency of these instructions above, the sequence of executed program lines will be as follows: I1, I2, label1, label2.

```
CODE
I1:      NOP     JS  label1             /* CMR unconditionally preloaded with ALW */
I2:      NOP     JMP  label2            /* CMR unconditionally preloaded with ALW */
I3:      NOP      /* hurdle 1 */
         :
         :
label1: NOP
I5:      NOP      /* hurdle 2 */
label2: NOP
```

Here we find that the PC value I3 is pushed onto the hardware stack for the instruction sequence above; this is as expected.

```
CODE
I1:       NOP      JMP  label1          /* CMR unconditionally preloaded with ALW */
I2:       NOP      JS  label2           /* CMR unconditionally preloaded with ALW */
I3:       NOP       /* hurdle 1 */
label1: NOP
I5:       NOP       /* hurdle 2 */
         :
         :
label2: NOP
```

But now we find that the instruction sequence above pushes I5 onto the stack. Again, this is because when JScc is executed, the instruction line at label1 is queued next in line.

A number of academic examples can be constructed from a sequence of JScc/RScc instructions, but they are perhaps not very useful. The predominant feature of this particular sequence is the aerobatic trajectory of the PC. We give one such example:

```
CODE
           NOP          JS  label               /* CMR unconditionally preloaded with ALW */
           NOP          RS              /* CMR unconditionally preloaded with ALW */
pcstack0:  instruction0        /* hurdle */
label:      instruction1
               :
```

In the example above, pcstack0 is pushed onto the hardware stack by JS. The program line holding RS is executed, then *instruction1* is executed, and then *instruction0* is executed. Then *instruction1* is executed again before the program continues on. *instruction1* is executed a total of two times, but *instruction0* is executed only once. By inserting NOPs and/or moving the label around, many more such exercises can be generated.

CASE: HALT class, HALT class

Generally speaking, this type of instruction sequence is not useful and should not be programmed. For example, two HALT pseudo instructions cannot be queued because the impact of the second HALT will be lost upon restart by the system host. As a second example, consider a HALT/BIOZ sequence. The BIOZ instruction loses its execution latency as the chip becomes effectively 'suspended' indefinitely. When the ESP_HALT_EN bit is not set, the execution latency of BIOZ returns hence producing a different outcome. There is no useful purpose for this.

CASE: JMP class, HALT class
The following instruction sequence is a useful programming paradigm **(**see the Applications section.**)**:

```
CODE
top_of_program:  NOP                          /* executes prior to suspension. */
I1:              NOP                       /* PC frozen here. */
                  :
                  :
                 NOP    JMP  top_of_program /* CMR unconditionally preloaded with ALW */
                 NOP    BIOZ
I2:                 ...   /* never executed */
```

Since both JMP and BIOZ are latent, the instruction line queued for execution following each is always executed before impact. The contiguous program line at I2 is never executed. The PC becomes frozen at the second queued instruction line following BIOZ if suspension occurs. When suspension terminates, execution will resume at instruction line I1.

CASE: HALT class, JMP class
This instruction sequence is automatically detected by the hardware as a special case. The HALT_JUMP monitor bit in the HOST_CNTL register goes high when the normal run-time latency of a JMP class instruction is being enforced following a HALT class instruction. The execution latency of a JMP might be absorbed by a preceding HALT class instruction were it not for this enforcement. So, from a programmer's point of view, <u>a JMP class instruction has the same execution latency under all circumstances</u> regardless of what instruction precedes it. We present this case here only in the interest of being complete. As an example:

```
CODE
                 NOP    BIOZ
I0:              NOP    JMP  I2            /* HALT_JUMP bit stays high for this cycle. */
I1:              NOP                   /* PC frozen here if it freezes. */
                   :      /* hurdle */
                   :
I2:              ...
```

Here the JMP instruction line at PC location I0 always executes prior to any amount of suspension time (including none) due to BIOZ. The instruction line at location I1 will execute following any suspension. But I1 <u>always</u> executes prior to the branch regardless of whether the state of suspension actually occurs; this is because of latency enforcement. After I1 executes, the PC will relocate to I2. Because a feature of the chip synchronization interface allows individual program loops to occasionally exceed the sample period, it is not desirable that the BIOZ instruction unequivocally cause suspension. It is for this reason that the enforcement is necessary.

4. Indirect Register Addressing

A mechanism is provided for indirect addressing of internal register-type operands for the MAC unit (D, E, F), the ALU (A, B, C), and AGEN (G). Seven SPRs, corresponding to the six operands of the ALU and MAC unit and the one AOR operand of the AGEN **(**INDIRA, INDIRB, INDIRC, INDIRD, INDIRE, INDIRF, and INDIRG**)**, act as *pointer registers* holding operand addresses. This mechanism indirectly accesses internal register data with normal latency once the pointer registers are loaded.

To proceed with indirect register addressing, we must first load some internal register address into one (or more) of the pointer registers corresponding to the desired operand(s). To activate the indirection mechanism, an instruction is written substituting the INDIRECT SPR as the desired operand for which indirection should occur. This substitution can be for any (or all) of the seven instruction operands. The hardware will recognize this special INDIRECT SPR and substitute the contents of the corresponding pointer register **(**INDIRA,B,C,D,E,F,G**)** for the operand address.

The INDIRECT SPR is not a physical register; it is a reserved address in the SPR space which is used by the programmer to activate indirection on a particular operand. Two more reserved addresses, INDIRINC and INDIRDEC, are provided which also accomplish indirection via the pointer registers. INDIRINC has the augmented functionality of post-incrementing the corresponding pointer register's contents. INDIRDEC post-decrements the corresponding pointer register's contents. (This can be very useful in looping on internal array elements.) When automatically inc[dec]remented in that manner, the INDIRA, INDIRB, and INDIRC pointer SPRs each become an extra source and destination whose timing follows the ALU source and destination in the Instruction Cycle Timing diagram shown in Figure 2. Likewise, automatic inc[dec]rement of INDIRD, INDIRE, and INDIRF follow the MAC unit source and destination timing, while INDIRG inc[dec]rement follows AGEN source and UPDATE BASE timing. The inc[dec]remented pointer register contents will be available for the purpose of indirection in the next instruction cycle; i.e., these automatic inc[dec]rement operations are nonlatent, and normal inter-unit latencies apply.

Miscellaneous

If the function unit instruction whose indirection operand is specified as INDIRINC or INDIRDEC is actually skipped (as in conditional execution), then the automatic increment or decrement will not take place.

We see in Figure 1 that the MAC unit's E operand has no access to AORs. Similarly, if INDIRE points to an AOR, the outcome is indeterminate.

Indirection within the REPT instruction repeated block is not prohibited.

4.1. Exceptions to Indirection

Since the ALU instructions, MOVcc, Jcc, JScc, RScc, and REPT, use the source operand fields of the instruction for storing Condition Mask and repeat count values, the indirection mechanism is disabled in the hardware for those particular source types to avoid unexpected results. This could hypothetically occur if one of the afflicted source operand fields were to contain a value which matched the INDIRECT, INDIRINC, INDIRDEC, or REF reserved addresses.

Table 7 lists the afflicted instructions showing the operands for which indirection is available * :

Table 7. Indirection Operand Availability

Instruction	**Operand A**	**Operand B**	**Operand C**
MOVcc		*	*
Jcc		*	n/a
JScc		*	n/a
RScc		*	*
REPT	*		n/a

ASSEMBLER NOTE: The assembler should detect the use of the REF, INDIRECT, INDIRINC, and INDIRDEC **keywords** as the B operand of REPT, or the A operand of MOVcc, Jcc, JScc, and RScc and then warn the programmer that indirection is disabled for those operands.

ASSEMBLER NOTE: In the * cases of Jcc, JScc, and REPT, there are four high instruction addresses which cannot be **direct**ly referenced as PC values. These illegal instruction addresses correspond to the register addresses of INDIRECT, INDIRINC, INDIRDEC, and REF. In these cases it is the PC value that is detected as an error, while the use of the four keywords is legal; the REF keyword draws an 'outcome indeterminate' warning, however.

ASSEMBLER NOTE: The values for the INDIRECT, INDIRINC, and INDIRDEC reserved addresses are all 10-bit values having 1 in the MSB. The G operand field of the instruction is only 9-bits wide. But, since the G operand field always refers to AORs, the 10th bit of this field is implicitly a 1.

PROGRAMMER NOTE: In order to use indirection effectively, **always use the verbose form of the instructions using no implicit destinations**.
ASDH, ASDL, LSDH, and LSDL require special consideration regarding indirection. See those instructions for information.
Admittedly our method of indirection is contrived. The motivation was the primary design objective that makes all instructions execute in one cycle.

4.2. Pointer Register Latencies

Since the contents of the INDIRA,B,C,D,E,F,G pointer SPRs are used as operand addresses rather than as operands, the latencies encountered when these registers are modified as normal destinations differ from the inter-unit latencies described elsewhere. These latency rules are as follows:

Nonlatent

1. The result of a MAC unit write to INDIRA,B,C is available for use as an indirect address no sooner than 1 instruction cycle (1 queued program line) later.

Latent

2. The result of a MAC unit write to INDIRD,E,F,G is available for use as an indirect address no sooner than 2 instruction cycles later.

3. The result of an ALU unit write to INDIRA,B,C,D,E,F,G is available for use as an indirect address no sooner than 2 instruction cycles later.

A programmer's chart can be found in the ESP2 Language and Software Specification.

5. Internal Memory Refresh

The majority of internal memory is implemented in Dynamic RAM (DRAM) and, therefore, requires a periodic refresh of its contents. This consideration applies to the instruction memory, the GPRs, and the AORs. Memory refresh might interfere with write access from the host interface or with memory writes made in the course of executing some typical ESP2 program were it not for dedicated hardware that transparently detects and prevents such collisions.

The decision to use DRAM for selected memory functions greatly increased the amount of internal memory. The SPRs (Special Purpose Registers) are implemented as Static RAM (SRAM), so they do not require refresh.

5.1. Instruction Refresh

Collision free instruction refresh is transparent to the programmer insofar as it happens nearly all the time; including suspension and halt. There are 4 memory accesses available in the instruction array per instruction cycle; these consist of two reads and two writes. Only 1 read is required for instruction fetching based on the PC, leaving three accesses available. 1 read and 1 write are used to allow host access to instructions when a host access is pending, or to perform refresh when no host access is pending. The remaining write access is unused.

Refresh is performed based on the REFINST SPR which is a 10-bit counter. Although the current implementation of the chip has 300 instructions, the counter is allowed to count from 0 to 1023 for future expandability of the instruction array. Since each instruction is refreshed once every 1024 instruction cycles, the refresh period at 100nS/instruction is 102.4 uS.

A bit in the HARD_CONF SPR can be used to enable or disable instruction refresh; the INST_REF_DIS bit disables instruction refresh when set (this is not desirable).

5.2. GPR and AOR Refresh

The GPR array allows 6 accesses per instruction cycle: 4 for reading source operands to the ALU and MAC unit, and 2 for writing results of ALU and MAC unit operations.
The AOR array allows 6 accesses per instruction cycle: 1 for reading a source operand to the AGEN,
3 for reading source operands to the ALU and MAC unit, and 2 writes of ALU and MAC unit results.
In both the GPR and AOR array there are no available accesses for refreshing these internal registers. Host access is governed by the ALU timing.

Refresh of internal registers at run-time is accomplished via common instructions invoking SPRs tied to hardware support of the refresh operation. (Recall that SPRs are static.) Those instructions provide refresh as an ALU function, NOP, BIOZ, HOST, and as a MAC unit function, RFSH, which each employ the indirect addressing hardware support provided by the REF and REFPT SPRs. We conservatively and empirically estimate a refresh rate of 5 kHz to be sufficient.

The refresh mechanism should be transparent to the programmer because it is built in to the most commonly used instructions, but some unusual user programs may warrant its consideration. In normal audio signal processing applications, use of the BIOZ instruction for sample rate synchronization, and available ALU NOP pseudo instructions should provide sufficient refresh. Additional refresh can be provided through use of the HOST instruction or by using the MAC unit RFSH pseudo instruction instead of NOP in places where preservation of MACRL is unnecessary.

The SPRs, REF and REFPT, control the refresh mechanism as follows: The reserved SPR address called REF provides an indirect addressing mechanism. By addressing REF in the operand field of an instruction, the content of REFPT is substituted as the operand address. REFPT is a 10-bit counter which post-increments whenever the REF reserved address is used as an ALU or MAC unit source operand address in an instruction, regardless of whether that instruction is actually skipped. Any successful (not skipped) move of REF to REF via either the ALU or MAC unit accomplishes one refresh that cannot be inhibited. The newly incremented REFPT value is not available as the destination address until the time at which the current destination address is decoded for the function unit that caused the increment. The assertion of HALT_REF_DIS in the HARD_CONF SPR can **never** override the use of the REF SPR, used as source and destination, as a means to effect internal register refresh. Two moves of REF to REF on the same instruction line using both the ALU and MAC unit doubles the refresh rate.

Because of the address mapping of GPRs from 0 to 455, and AORs from 512 to 963, the LSB of the REFPT counter serves as the MSB of the address so that refreshes alternate between GPR and AOR. Although REFPT is a 10-bit counter, the count is limited to the number of GPRs and AORs physically residing on the chip in the current version.

5.2.1. Internal Register Refresh during Suspension/Halt

The MAC unit and the ALU can both be used to perform internal register refresh operations simultaneously which doubles the refresh rate.

When the chip is in the state of suspension (BIOZ bit of the HOST_CNTL register is set) or in the halt state (because of a high halt bit in the HOST_CNTL register), the MAC unit executes NOPs by default (designed to preserve MACRL) while the ALU executes the HOST instruction which allows host access or refresh when no access is pending. In these two states, the MAC unit can be forced to do refresh automatically by setting the HALT_MAC_REF bit in the HARD_CONF SPR. This has the effect of doubling the refresh rate, but the contents of the MACRL SPR will not be preserved. When the ESP2 program resumes (i.e., comes out of halt or suspension) the MACRL will subsequently be preserved, assuming no RFSH instructions in the MAC unit code.

The HALT_REF_DIS bit in the HARD_CONF SPR can be set to disable internal register refresh by both the MAC unit and ALU during halt or suspension (this is not desirable). The assertion of HALT_REF_DIS always overrides the HALT_MAC_REF bit.

5.2.2. Internal Register Refresh Collision

Due to the interleaved nature of the MAC unit and ALU operand fetching, one of these function units can be writing a new result to some internal register at the same time that the other unit is being used to refresh it. This consideration is also applicable to the case of host access, whose buss timing is governed by the ALU, either through the HOST instruction or during the suspension or halt states. The chip contains logic to prevent a refresh operation from writing old data over newly created register contents.

Whenever REF is used as the destination in the MAC unit, the chip compares the contents of REFPT to the destination address of the ALU from the preceding instruction cycle. If the values are equal and the ALU instruction is not actually skipped (as in a conditionally executed instruction), a NOP is performed by the MAC unit instead of RFSH.
Likewise, when REF is used as the destination in the ALU, the chip compares the contents of REFPT to the destination address of the MAC unit from the same instruction cycle. If the values are equal and the MAC unit instruction is not actually skipped, a NORFSH pseudo is substituted in place of NOP in the ALU.

6. SPR Hazards

Many of the SPRs in the chip provide special dedicated services which require modification of the SPR contents as a result of ALU or AGEN operations. In those cases these registers are being modified by means other than their usage as the destination operand of some instruction. All SPRs can be read and written by the MAC unit and the ALU **(**with the exception of a handful of registers which are read-only or write-only**)**. A conflict (a hazard) can occur at instances in which a particular SPR is modified in association with its special service at the same time that it is written as the destination of some MAC unit or ALU operation. The cases are all listed below:

ASSEMBLER NOTE: The assembler should detect these hazards and issue errors (or warnings) because the outcome of these cases is indeterminate.

1. Specifying the CMR as the MAC unit destination on the same instruction line as an ALU Jcc, JScc, RScc, or MOVcc.

2. Specifying the PC as the MAC unit destination at any time.

3. Specifying REPT_ST, REPT_END, or REPT_CNT as the MAC unit destination on the same instruction line as an ALU REPT (or EXIT pseudo) instruction, or on the last line of a REPT block.

4. Specifying the PCSTACK0,1,2,3 SPRs as the MAC unit destination on the same instruction line as a JScc or RScc.

5. Specifying a JScc, RScc, or Jcc on the penultimate queued instruction line of a REPT block.

6. Specifying a region BASE SPR as a MAC unit destination on the same instruction line as the AGEN performing an UPDATE BASE operation on the same register.

7. Specifying either SPR, HOST_GPR_DATA or HOST_ESP_FACE, as a MAC unit destination at any time should be cited because it could cause conflict with a host write to that register as a host interface register. (warning)
   There is a conflict specifying HOST_CNTL_SPR as a MAC unit source at any time because the host may be writing to it as an interface register at the same time that the MAC unit is reading from it. (warning)

8. Specifying CCR as the MAC unit destination when the ALU operation on the same instruction line is <u>not</u> MOV (or NOP), MOVcc, Jcc, JScc, RScc, HOST (HOSTNORFSH), BIOZ (BIOZNORFSH), or REPT. (warning)
   Specifying CCR as the ALU destination of an instruction which is <u>not</u> MOV, MOVcc, RScc, HOSTNORFSH, or BIOZNORFSH. (warning)

9. The indirection mechanism provides an automatic increment or decrement of the pointer SPRs via the INDIRINC and INDIRDEC SPRs. Consequently, the following indirection hazards can occur:
   1**)** MAC unit write to INDIRG on the same program line as an AGEN reference to INDIRINC or INDIRDEC.

2**)** MAC unit write to INDIRD when the D operand is INDIRINC or INDIRDEC, or to INDIRE when the E operand is INDIRINC or INDIRDEC. ALU write to INDIRA when the A operand is INDIRINC or INDIRDEC, or to INDIRB when the B operand is INDIRINC or INDIRDEC.
3**)** MAC unit write to INDIRINC or INDIRDEC when INDIRF points to itself. ALU write to INDIRINC or INDIRDEC when INDIRC points to itself. (This last MAC and ALU hazard is not detectable by the assembler, in general.)

7. External Data-Memory Interface

The interface to external memory is comprised of two pieces:
1) the Address Generator function unit, AGEN, which performs modulo address calculation in its fundamental mode,
2) the **external memory buss** comprising an address and data buss, which provides the physical connection to external memory (not shown in Figure 5).
This memory port contains the DIL and DOL SPRs which act as the data interface between external memory and the ALU and MAC unit. (See Chip Architecture.)

7.1. Address Generator (AGEN) Architecture

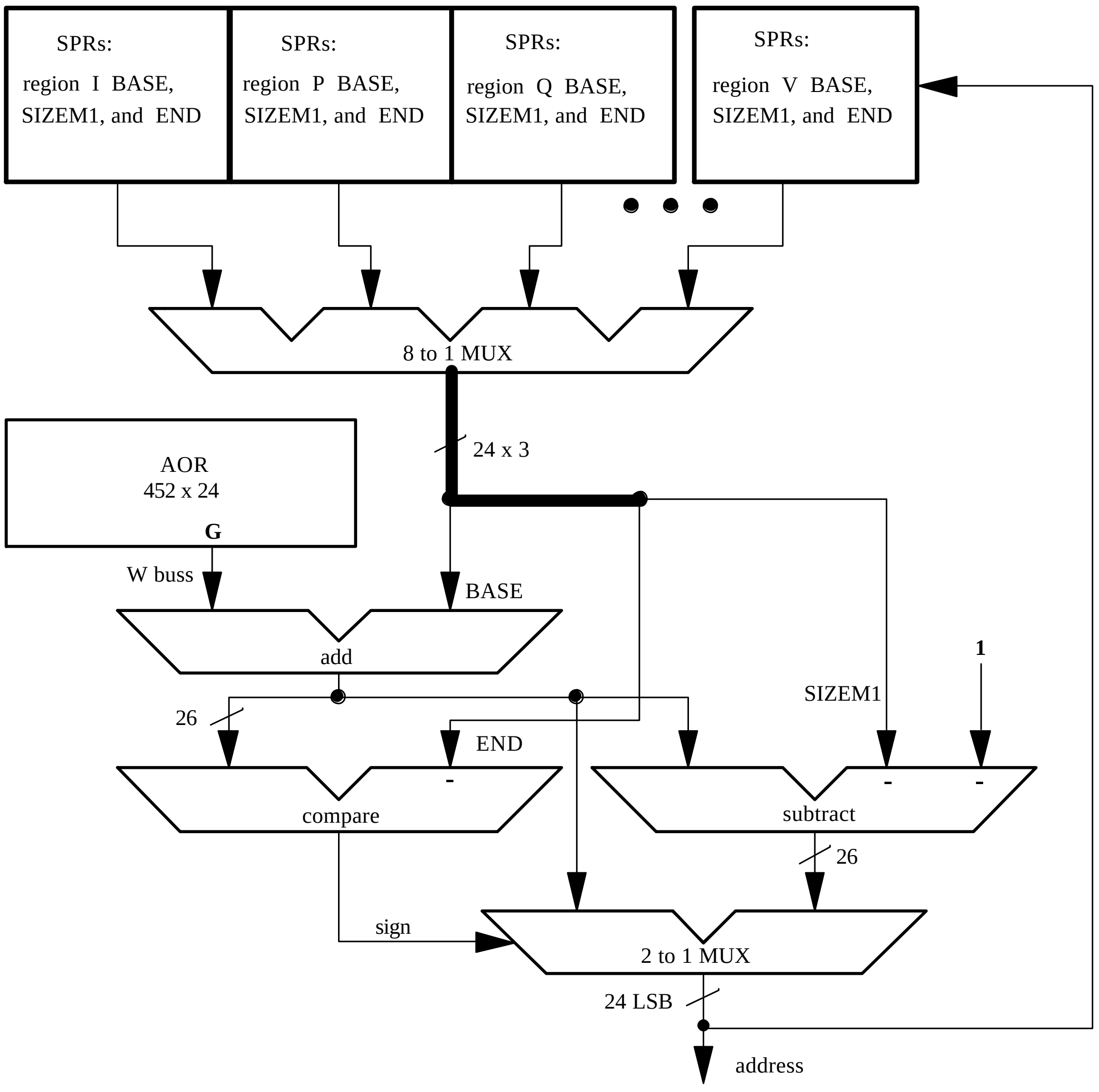

SPRs:
region I BASE, SIZEM1, and END
SPRs:
region P BASE, SIZEM1, and END
SPRs:
region Q BASE, SIZEM1, and END
SPRs:
region V BASE, SIZEM1, and END
8 to 1 MUX
24 x 3
AOR
452 x 24
G
W buss
BASE
add
1
SIZEM1
26
END
-
compare
subtract
-
-
26
sign
2 to 1 MUX
24 LSB
address

Figure 5. Address Generator

A partial diagram of the Address Generator unit is shown in Figure 5. The Address Generator calculates absolute addresses for the external memory interface once every instruction cycle (100nS). It consists of three 26-bit adders, eight BASE SPRs, eight END SPRs, and eight SIZEM1 SPRs, the 24-bit unsigned SPRs serving to partition eight addressing regions. Relative address offsets held in 24-bit unsigned AORs are selected by the G operand field of the instruction. The programmer dereferences an AOR to make an indirect access of external memory.

AGEN code can be automatically generated for the programmer by the assembler. Three bits in the external-memory control-field of the micro-instruction determine which of the eight regions is accessed. The programmer's code either states the desired region, or it is implicit from the declaration of the AOR. The memory location and purpose of the eight regions are each determined by the contents of the corresponding **region control registers (**BASE, SIZEM1, and END**)**. The region control registers can be automatically initialized by the assembler from values implicit in the programmer's region declarations. These regions can each be configured as delaylines, tables, or I/O space through the appropriate setting of the region control registers. This becomes apparent by examining the nature of the calculations automatically performed in the AGEN:

7.1.1. AGEN Address Calculation

At every instruction cycle the following external memory address calculations are performed using the specified address offset register and the region control registers from the region specified in the micro-instruction.

```
address = AOR + BASE
if address > END
    address = address - (SIZEM1 + 1)
```

'address' is an absolute external memory address. AOR denotes one of the Address Offset Registers. The SIZEM1 SPR contains the size of the region - 1; this deficit is necessary in order to represent a table which is the full size of physical memory, and accounts for the correction factor '1' appearing in Figure 5. Although the BASE, SIZEM1, END, and AOR are 24-bit unsigned, the calculations are carried out at 26-bit two's complement to prevent the first equation from overflowing and to guarantee the correct sign in the result of the comparison.

7.1.2. Plus-One Addressing Mode

The first calculation can also be executed as

address = AOR + BASE + 1

under instruction control. This addressing mode (not shown in Figure 5) is useful in acquiring neighboring addresses as for interpolation operations. No register is permanently modified as a result of this calculation which occurs prior to external memory access.

7.1.3. Extent of the Modulo

These equations allow circular addressing within a predefined region because of the region END detection and modulo addressing. This modulo scheme of addressing restricts the AOR offset range to 0 through SIZEM1 + 1, for any value of BASE within the modulus. Since the modulo does not go to infinity, it is easy for computed AOR offsets to land outside of the modulus, so programmers should beware.

Regions can reside anywhere in physical memory based on the contents of the region control SPRs. Regions cannot span absolute address zero.

Multiple delaylines coexist in the same region by stringing them end to end and addressing them using different offsets within the modulus. Delaylines will move only within the extent of the defined region by decrementing the region BASE in a modulo fashion described in the section UPDATE region BASE. Therefore, the operation of delaylines will not impact data in other regions of memory.
Modulo addressing is always with respect to the region, not to the individual delayline.

7.1.4. Other External Memory Configurations

By setting the END SPR to the value of the maximum physical memory location, the region simplifies to a table not having modulo addressing. Tables are distinguished from delaylines by fixing the BASE SPR. In this case the BASE always points to the first location of the table. Tables can certainly be implemented, however, without defeating the modulo addressing hardware scheme.

Setting BASE to zero and END to the value of the maximum physical location defeats modulo addressing while admitting absolute addressing, in the AORs, suitable for peripheral I/O devices. The recommended use of AORs is to hold relative address offsets with respect to the associated region. I/O addressing can certainly be implemented in a relative fashion without the need to defeat the modulo addressing hardware. In any case, for absolute addressing the MUX_ADDR bit in the HARD_CONF SPR must be low (linear addressing). All these configurations can be determined by the programmer in the declarations. (See the Software Spec. for more.)

7.1.5. UPDATE region BASE

A built-in method for updating the region BASE SPR under program control is also provided via the AGEN. This mechanism is especially useful for **dec**rementing the BASE register every sample period in a modulo fashion. This is desirable when programming delaylines. As before, the AGEN calculates a new address using the contents of the AOR **(**pointed to by the G operand field**)** as the offset amount. The only change is that the AGEN opcode field of the instruction demands that the address be written to the external address buss as well as back into the BASE SPR. The AOR, in this addressing mode, becomes a post-increment to the region BASE SPR. This explains the feedback path in Figure 5.

```
address = AOR + BASE
if address > END
    address = address - (SIZEM1 + 1)
BASE = address
```

The new BASE value is available to the MAC unit, ALU, and AGEN on the next instruction cycle for use both in AGEN addressing and as a source SPR. To modulo decrement the BASE by 1, set AOR equal to contents of SIZEM1. To decrement the BASE by 2, set AOR equal to SIZEM1 - 1, and so on. Decrementing can be performed indefinitely, the BASE remaining within the modulus.

ASSEMBLER NOTE: If BASE updating is specified in a given instruction, a collision could occur between the AGEN and the MAC unit trying to write to the same region BASE SPR because the two units are operating with the same timing. **This hazard should be recognized and flagged as an error by the assembler; chip operation is unpredictable.**

7.2. AGEN Instructions

The AGEN opcode field consists of six instruction bits. Three of the bits are used for selecting one of eight regions. The remaining three bits are used to select from the following operations:

NOP cycle on the external memory buss.
External memory RD.
External memory RD, then UPDATE BASE.
Plus-One addressing, external memory RD.
External memory WR.
External memory WR, then UPDATE BASE.
Plus-One addressing, external memory WR.
UPDATE BASE.

During an AGEN NOP, the external memory buss is tristated. The last operation (UPDATE BASE) allows UPDATE of the region BASE SPR while tristating the external memory buss.
See the ESP2 Language and Software Specification for the proper syntax of AGEN instructions.

7.3. Accessing the Region Control Registers and AORs

The region control registers (BASE, SIZEM1, and END) are mapped as SPRs for use as source/destination operands by the ALU and MAC unit. This can be useful for having more than eight physical regions of external memory active at a time, since the contents of these registers can be modified under program control.

In the assembler, the regions are called I,P,Q,R,S,T,U,V. These are respectively assigned in microcode to: 0,1,2,3,4,5,6,7. So references to BASEP, SIZEM1U, or ENDT, for example, denote a region control SPR in a particular region.

The AORs can also be accessed by the ALU and MAC unit for computing address offsets in the running program. The ALU provides the ADDV and SUBV instructions for unsigned arithmetic operations. Because of bandwidth limitation on the AOR memory, AORs cannot be used as the E operand of the MAC unit. (Refer to the section on Register Usage.) When AORs are not being used to hold address offsets, they can be used for any general purpose.

See the Pipeline section in the Software Spec. for a simple chart of latencies in the access of the registers discussed above.

7.4. External Memory Access

The external memory interface port consists of a 24-bit address buss, a 24-bit data buss, the pin signals Row Address Strobe (RAS \), Column Address Strobe (CAS \), Memory Read/Write (**MR/W\**), and Memory Request (**MEM_REQ\**). The MEM_REQ\ pin is asserted by the ESP2 to signal an upcoming external memory cycle. When the MEM_REQ\ signal is asserted, the CAS \ and RAS \ signals of the ESP2 will assert during the <u>next</u> memory cycle. The MEM_REQ\ and MR/W\ pins each require a pull-up resistor.

ESP2 External Memory Access Timing

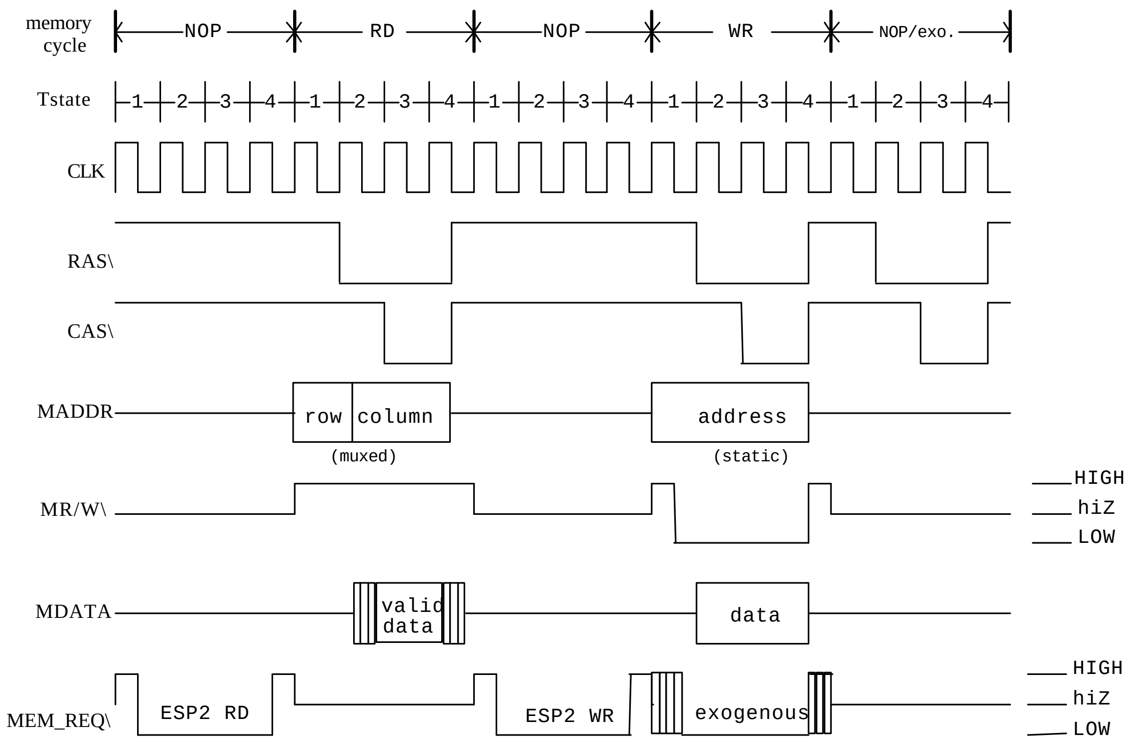


Figure 6. Timing with refresh disabled. The MUX_ADDR bit selects the multiplexed or static addressing modes on-the-fly.

The timing diagram in Figure 6 illustrates activity on the memory buss over five instruction cycles for several operations. One instruction cycle corresponds to four Tstates which corresponds to one memory cycle. Since addressing can be either multiplexed (DRAM) or linear (SRAM), the diagram shows both forms. The signals RAS\ and CAS\ are asserted regardless of whether multiplexed or linear addressing is performed. This is because the RAS\ signal is sometimes used for chip-enable while CAS\ is sometimes used for output-enable of static RAM chips. In the event that the RAS low time provides insufficient access time for SRAMS, an upper address line with a resistor pullup can serve as a chip select providing a 3.5 clock cycles worth of access time. Notice how the MR/W\ and MEM_REQ\ signals are driven high by ESP2 before and after there assertion by the ESP2. Driving high at the end of the cycle ensures good rise time characteristics as opposed to a resistor pullup. These signals should have resistor pullups to hold them high while they are tristate by the ESP2

The ESP2 will tristate (release) the pin signal MR/W\ and the external memory buss (address and data) during AGEN NOP or UPDATE BASE (no access) instruction cycles, and also during halt state and BIOZ suspension where the AGEN is designed to execute NOPs. This feature can be used to allow exogenous hardware or other running ESP2 chips access to the same external memory. Since the MEM_REQ\ pin tristates during unused cycles, it can be driven by exogenous hardware to force the ESP2 to generate a RAS\/CAS\ sequence on the very next memory cycle. **The assertion of MEM_REQ\ by an exogenous source does <u>not</u> force the ESP2 to release the external memory**

buss, so buss contention is possible. For this reason it is recommended to allow access by an exogenous source only during the halt and indefinite suspension states.

When multiple ESP2s are set up to share the same external memory, however, one of the chips can automatically drive the RAS \ and CAS \ pins by watching its MEM_REQ \ input pin which is wire-ORed between all the running ESP2s. The memory accesses must be scheduled in the AGEN code of each chip so as to avoid buss contention. This presumes that the chips are running in synchronization with one another which can be accomplished by a system reset.

7.4.1. External Address Buss

The external memory address buss is controlled by the ESP2 for the purpose of supplying a 24-bit address to external memory and peripheral I/O devices. A bit-selectable mode is available on-the-fly which provides multiplexed addressing (RAS \/CAS \) for accessing dynamic RAM (DRAM), in sizes from 64K to 16M, when the **MUX_ADDR** bit in the HARD_CONF SPR is high (**set**). Otherwise (**reset**), linear addressing for static RAM (SRAM) or peripheral I/O is selected. When the MUX_ADDR bit is asserted from the MAC unit, the new AGEN addressing mode is activated immediately (i.e., in the same instruction cycle). If asserted from the ALU, the new addressing mode is activated on the next instruction cycle.

Multiplexing is illustrated in Table 8 for DRAMs of various sizes. The Table shows what actually <u>appears</u> at the address pins when either mode is selected. For example; when a 256K by *n*-bit DRAM is connected, what actually appears at the ESP2 address <u>pin</u> called A8 is the address <u>bit</u> A0 when RAS \ (Row Address Strobe \) is asserted, and address <u>bit</u> A16 when CAS \ (Column Address Strobe \) is asserted.

The wordlength, *n*, of the physical DRAM does not impact the address pin connections. Further, **DRAMs of various size may coexist in a system**.

The address pin names, MADDR[23:0], have been shortened in the Table.

<u>Table 8</u>. External Address Pin Connection

PIN NAME	**S**RAM **mode:** linear	**D**RAM multiplexed	DRAM multiplexed	DRAM multiplexed	DRAM multiplexed	DRAM multiplexed	DRAM multiplexed
A23	A23	A23	A23	A23	A23	A23	A23
A22	A22	A22	A22	A22	A22	A22	A22
A21	A21	A21	A21	A21	A21	A21	A21
A20	A20	A20	A20	A20	A20	A20	A20
A19	A19	A19	A19	A19	A19	A19	A19
A18	A18	A18	A18	A18	A18	A18	A18
A17	A17	A17	A17	A17	A17	A17	A17
A16	A16	A16	A16	A16	A16	A16	A16
	RAS\CAS\ **size:** any	RAS\CAS\ any	RAS\CAS\ 64K	RAS\CAS\ 256K	RAS\CAS\ 1M	RAS\CAS\ 4M	RAS\CAS\ 16M
A15	A15	A11/A23					A11/A23
A14	A14	A3/A22					A3/A22
A13	A13	A10/A21				A10/A21	A10/A21
A12	A12	A2/A20				A2/A20	A2/A20
A11	A11	A9/A19			A9/A19	A9/A19	A9/A19
A10	A10	A1/A18			A1/A18	A1/A18	A1/A18
A9	A9	A8/A17		A8/A17	A8/A17	A8/A17	A8/A17
A8	A8	A0/A16		A0/A16	A0/A16	A0/A16	A0/A16
A7	A7	A7/A15	A7/A15	A7/A15	A7/A15	A7/A15	A7/A15
A6	A6	A6/A14	A6/A14	A6/A14	A6/A14	A6/A14	A6/A14
A5	A5	A5/A13	A5/A13	A5/A13	A5/A13	A5/A13	A5/A13
A4	A4	A4/A12	A4/A12	A4/A12	A4/A12	A4/A12	A4/A12
A3	A3	A3/A11	A3/A11	A3/A11	A3/A11	A3/A11	
A2	A2	A2/A10	A2/A10	A2/A10	A2/A10		
A1	A1	A1/A9	A1/A9	A1/A9			
A0	A0	A0/A8	A0/A8				

The DRAM sizes shown in Table 8 are typical of the industry standards. Since DRAM address pins are multiplexed, every pin added increases memory size by a factor of 4. Typical DRAM chips may not be fast enough to interface to the ESP2 running at full speed. The DRAM access time must be at most 50 nS when running the ESP2 at the system clock rate of 40 MHz (10 MHz instruction rate). Slower DRAM demands a slower clock rate. Very high speed SRAM is commonly available, hence the two address buss modes.

The external memory buss is asserted for every AGEN instruction type excepting a NOP and an UPDATE BASE (no access) instruction. Since those two operations do not access external memory, the address and data buss are tristated. The external memory buss is also tristated during halt or when the chip is in suspension due to the BIOZ instruction, for then the AGEN is performing NOPs.

7.4.2. External Memory Data-Interface

The interface from the chip's internal computation units (MAC and ALU) to external memory is supplied through the AGEN's Data Input Latches (**DIL**s) and Data Output Latches (**DOL**s). A 4-bit field in the micro-instruction along with a read/write bit identifies one of these registers to AGEN. The assembler can organize the use of these registers into FIFO and cache structures.
There are 16 DILs and 16 DOLs, all 24 bits wide and mapped as SPRs. These registers are accessed by the computation units as sources and destinations like any other SPR.

The AGEN provides a feature called **magnitude truncation** (to 16 or 24 bits) of DOL out to external memory. The 24-bit DOL is assumed to hold the MSBs of a larger word in two's complement. Magnitude truncation to 24 bits or less is desired. This feature is controlled by two bits in the HARD_CONF SPR: The **MAG_TRUNC** bit enables magnitude truncation when it is high. When set from the MAC unit, magnitude truncation becomes active on the same program line. When set from the ALU, magnitude truncation is activated on the next queued program line. The TRUNC_WIDTH bit sets the truncation width to 24-bits when it is high, or 16-bits when it is low.

In the case of 24-bit truncation width, the magnitude truncation algorithm is as follows:

```
if (DOL < 0)  external memory = DOL + $1
else  external memory = DOL
```

In the case of 16-bit truncation width, the algorithm is:

```
if (DOL < 0)  external memory = (DOL + $100)  &  $FFFF00
else  external memory = DOL  &  $FFFF00
```

For 16-bit truncation width, the 8 LSBs of the 24-bit result are set to $00 which allow development of algorithms for 16-bit target systems within development systems having 24-bit external memory.

The selected algorithm is performed in the AGEN as DOL is sent out to external memory; the DOL is not permanently modified. For magnitude truncation somewhere in between 16 and 24-bits, one can employ the various shifting mechanisms in the computation units prior to magnitude truncation.

7.5. External DRAM Refresh

ESP2 External Memory Refresh Timing

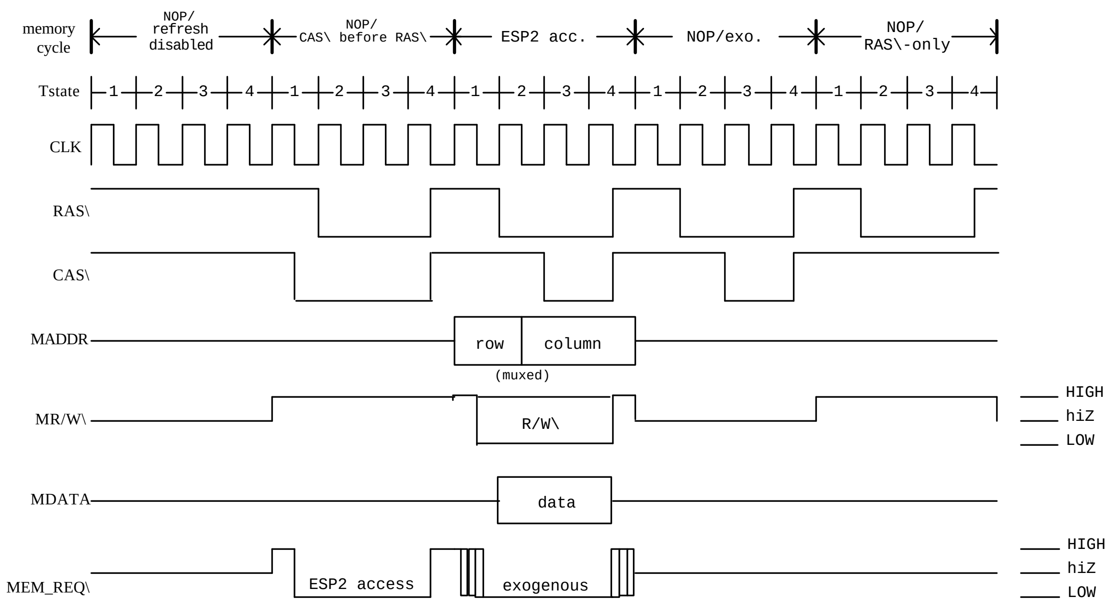


Figure 7. Refresh enabled after first cycle. Refresh is associated with AGEN NOP when there is no MEM_REQ\ on the previous cycle.

The ESP2 supports both DRAM and SRAM as external memory, **and both memory types may be coexistent in a system design**. There exists an automatic external DRAM refresh mechanism built into the ESP2. This mechanism acts in conjunction with the address counters built into most all commercial DRAM chips. The DRAM's address counter controls a refresher, also inside the DRAM chip, which becomes activated by the appearance of the pin signal CAS\ before the pin signal RAS\. ESP2 supports the CAS\ before RAS\ refresh mode for total refresh. Notice in Figure 7 how the MR/W\ signal is driven high by ESP2 during a refresh cycle. Depending upon the system design when SRAM is present, be aware that the refresh mechanism may cause a read of SRAM, putting data on the external memory buss.

A second mode of refresh, RAS\-only refresh, is <u>not</u> fully supported by ESP2 because it requires a refresh address to appear on the external memory buss. The RAS\-only refresh signal can be made to emanate from the ESP2 RAS\ pin, however. Do not allow the address buss to become undefined during a RAS\-only refresh as this can corrupt DRAM regardless of the state of the MR/W\ or CAS\ lines.

These two modes of refresh are determined by the settings of two bits in the HARD_CONF SPR. The XMREF_CASDIS bit, when set, yields RAS\-only refresh. When the XMREF_DIS bit is set, both modes of refresh are disabled. **(**XMREF_DIS overrides XMREF_CASDIS.**)** Both bits low yield the default mode, CAS\ before RAS\ refresh. All settings are independent of the MUX_ADDR bit.

When activated, the ESP2 external DRAM refresh mechanism operates transparently at run-time, halt, or during BIOZ suspension. At run-time the refresh mechanism engages only when the AGEN unit is executing NOP or UPDATE BASE (no access) instructions **and** the MEM_REQ\ pin signal was **not** asserted by any source (exogenous or ESP2 itself) on the previous memory cycle. During halt or suspension, recall that the AGEN executes NOPs, so refresh will automatically occur on every memory cycle as long as the MEM_REQ\ signal was not asserted on the previous cycle. **Recall that the act of reading or writing external memory at *N* sequential locations over a specified time period also accomplishes total refresh, where *N* is the square root of the external DRAM size.**

Note that if an exogenous device takes over the external memory buss and exercises MEM_REQ\, it will prevent ESP2 from refreshing DRAM. If the external memory buss is usurped for extended periods, the exogenous device must be responsible for refreshing DRAM.

7.6. Initializing or Accessing External Memory from the System Host

The ESP2 does not provide internal hardware for direct access of external memory by the system host through the ESP2. The ESP2 tristates its external memory buss during the states of halt and BIOZ suspension, and during NOP or UPDATE BASE (no access) instruction execution in the AGEN. It is at these times that exogenous hardware (e.g., the host itself) can be given direct access to external memory over that same buss.

If we allow the host to take-over the ESP2 external memory buss, then when external **D**RAM is used and there is no MEM_REQ\ pin signal assertion from either the ESP2 or the host, **the DRAM will always be automatically refreshed by ESP2** on the next memory cycle if the CAS\ before RAS\ refresh mode is activated in the HARD_CONF SPR. To avoid buss contention with a running ESP2 program making random access, it may be preferred to transfer external memory data to/from the host (or some DMA mechanism) during halt or indefinite suspension. But if the transfer needs to occur at run-time, various methods may be employed which require a very simple ESP2 program to supervise indirect transfer through the ESP2:

One such procedure might incorporate the ALU BIOZ instruction. This instruction indirectly monitors the IOZ input pin from the synchronization interface. The IOZ pin will typically be hardwired to the sample rate clock (LRCLK). The implication here is that were the IOZ pin involved in the transfer of audio data to/from external memory, then the transfer could occur no faster than the system sample rate. Under these assumptions, the standard host/ESP2-register interface would most likely be used. In that case, some GPR would be designated to hold the transferred data while some AOR would be designated to hold the external memory address offset. Of course some protocol would need to be established between the host and the ESP2 program, but this protocol is malleable.

A second procedure might incorporate the IFLAG and OFLAG pins from the synchronization interface. There is no dedicated ALU instruction to monitor that input pin, but an image of the IFLAG pin appears in the CCR. There it is called IFLG and is updated on a per instruction basis. Therefore the IFLAG pin can be easily monitored by the ESP2 program based on the IFLG and NIFLG Conditions. Using this scheme, data would be transferred to/from the 8-bit HOST_ESP_FACE_B2,1,0 host interface registers. The ESP2 would transfer data directly into or out of the concatenated 24-bit SPR image (HOST_ESP_FACE) of these registers, bypassing the standard host/ESP2-register interface. The ESP2 could then signal back to the host via the OFLAG pin. Of course, some software protocol between the host and the ESP2 program would need to be established. This second procedure has the advantage that data transfer occurs at the instruction rate.

It may be likely that these two procedures would find use while the chip is executing some application program. So, some scheme must be invented to vector the running ESP2 program to this external memory host-access routine after it has been overlaid. **Vectoring by host load of the PC at run-time, using the standard host/ESP2-register interface, is <u>not</u> recommended because of indeterminate results.** An alternative vectoring procedure is given in the Applications section.

8. Serial Interface

The asynchronous serial interface consists of 16 24-bit serial data SPRs (called SER0L, SER0R, SER1L, SER1R, ..., SER7R), each L/R pair of which can be used for input or output to A/Ds, D/As, other DSP chips, etc. These SPRs are grouped into eight stereo pairs, each pair multiplexed on its own serial data pin (SER[0:7]). The SER data SPRs are double buffered so that both left then right channel data coming into the chip in one sample period are accessible to the ESP2 at the start of the next sample period. Likewise, data written to the left and right SER data SPRs by ESP2 during one sample period will be sent out in the next sample period, left channel followed by right. That is to say, there exists a latency equal to one sample period for serial data input or output. This design allows ESP2 access to SER data SPRs at any time throughout the sample period with no hazards.

Each of the individual serial data pins (SER[0:7]) can be programmed as input or output, and each can be assigned to either of two sets of **a**synchronous serial clocks **(**LRCLK, WCLK, and BCLK**)** via the SER_CONF SPR. This accommodation for simultaneous communication between devices having different serial timing requirements opens the door to applications such as sample rate conversion.

8.1. Serial Interface Control Registers

Because of the great diversity of definitions for serial clock timings amoung the commercially available D/As and A/Ds, the best approach is to make the serial interface fully programmable. The serial interface can be made to conform with I^{2}S justification or the popular 'right justified' format. Format programmability can be achieved using a BCLK *counter* and the following *serial interface control registers*:

Table 9. Serial Interface Control Registers

WCLKL_ST	Defines the start of WCLK and left serial data.
WCLKL_END	Defines the end of WCLK and left serial data.
LRCLK_END	Defines the end of LRCLK.
WCLKR_ST	Defines the start of WCLK and right serial data.
WCLKR_END	Defines the end of WCLK and right serial data.
LRCLK_MOD	Defines the count **modulus** less 1 of the BCLK counter.

See the example for settings rules. By duplicating the BCLK counter and serial interface control registers, two independent timing sets are created with each serial data line assigned to either timing set. These 8-bit serial interface control registers are concatenated into SPRs called SCLK**0**_REG0 and SCLK**0**_REG1 both defining timing set 0, and SCLK**1**_REG0 and SCLK**1**_REG1 defining timing set 1; see those SPRs for the exact format.

Observing the layout of the HARD_CONF SPR (see the section on Special Purpose Registers), any of the BCLK[0:1], and/or WCLK[0:1], and/or LRCLK[0:1] serial clock signals may be selectively enabled to drive the associated pins. When any of those signals are not enabled, the associated pin becomes a high impedance input. Any subset of the ESP2 serial clock pin signals may be selected as inputs independent of all the other clock pins.

8.2. LRCLK

The LRCLK (left right clock) signal determines the channel, left or right. If the ESP2 is driving the LRCLK pin, it will assert (rise) after the BCLK counter reaches the value in LRCLK_MOD (see the example). Due to that assertion the BCLK counter resets and begins the count again, counting the number of bit clocks since the beginning of the sample period marked by the rising edge of LRCLK. The falling edge of LRCLK is defined by LRCLK_END.

The BCLK counter will reset whenever the LRCLK signal transits high no matter who is driving the associated LRCLK pin. The SER data SPRs are latched as well on the positive transition of LRCLK.

The system sample rate can be defined by LRCLK as generated by ESP2. If this is desired, then the exact rate can be expressed in terms of: 1)the system clock (CLK), 2)the BCLK modulus (LRCLK_MOD+1), and 3)the BCLK divide rate set in the HARD_CONF SPR. See Figure 8.

When some other device is driving the LRCLK pin, it is not a requirement to load the LRCLK_END and LRCLK_MOD serial interface control registers. When some other device is driving LRCLK and the BCLK and WCLK pins as well, it is not important how any of the serial interface control registers are set.

8.3. WCLK

The WCLK (word clock) signal always frames the data (see the example). The justification and duration of the WCLK pin signal output (hence SER[0:7] pin data) is determined by comparing the unsigned contents of WCLKL_ST, WCLKL_END, WCLKR_ST, and WCLKR_END to the BCLK counter. This scheme gives flexibility on the separation of right and left channels, the justification of the data transfer with respect to LRCLK transitions, and the number of bits of data transfer up to the full 24-bit width of the SER data SPRs.

If no other chip in the system requires WCLK, the ESP2 may generate its own. As an example in the case that some exogenous device provides LRCLK and BCLK but does not provide WCLK, the ESP2 can generate WCLK for itself (and any other chip requiring it) while its own LRCLK and BCLK pins are configured as inputs. This is achieved through the proper setting of the serial interface control WCLK registers and by configuring the WCLK pin signal as output via the HARD_CONF SPR. This extended functionality is eminently useful while the concept is applicable to other cases.

8.4. BCLK

The BCLK (bit clock) counter is zeroed by the positive transition of the LRCLK signal regardless of whether the associated LRCLK pin is being driven by an exogenous source or by the ESP2 itself. The BCLK counter is also zeroed when the chip is reset via the RES \ pin. The BCLK pin is disabled (made an input) by RES \. When the BCLK pin is enabled, oscillations at the BCLK pin output are always observed. The BCLK divide rate, shown in Figure 8, is set in the HARD_CONF SPR (see the SPR Descriptions).

The asynchronous BCLK pin signal controls the clocking-in of SER[] data to internal shift-registers. Data is valid on the rising edge of the BCLK signal for input or output data. Because the serial interface is asynchronous, the number of bit clocks per sample period need **not** be an exact integer multiple of the sample period, but the prescribed number of bit clocks <u>must</u> not exceed the actual sample period.

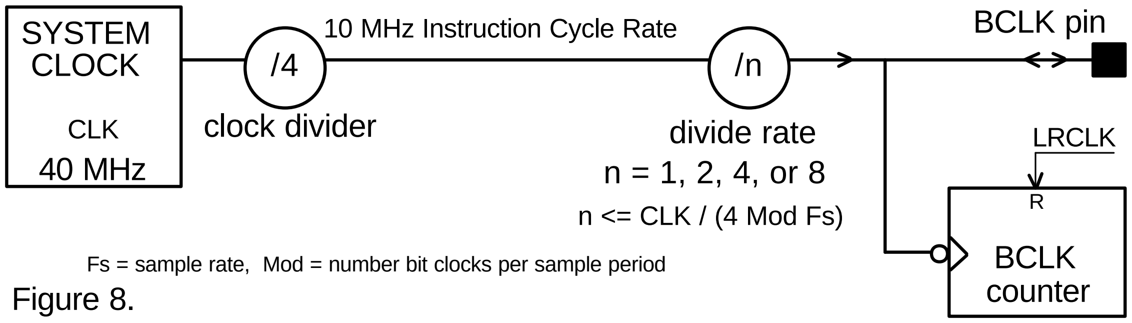


Figure 8.

8.5. An Example of Serial Interface Control Register Settings

It must be emphasized that the ESP2 serial interface is designed to be **a**synchronous. In light of that, the example timing diagram in Figure 9 might be used to understand how serial data is transmitted or received under any I/O configuration of the serial clock pin signals, LRCLK, WCLK, and BCLK.

The example in Figure 9 is also intended to provide rules for setting the serial interface control registers. Remember that when serial clock pins are driven by an exogenous device, the settings of the corresponding serial interface control registers become irrelevant.

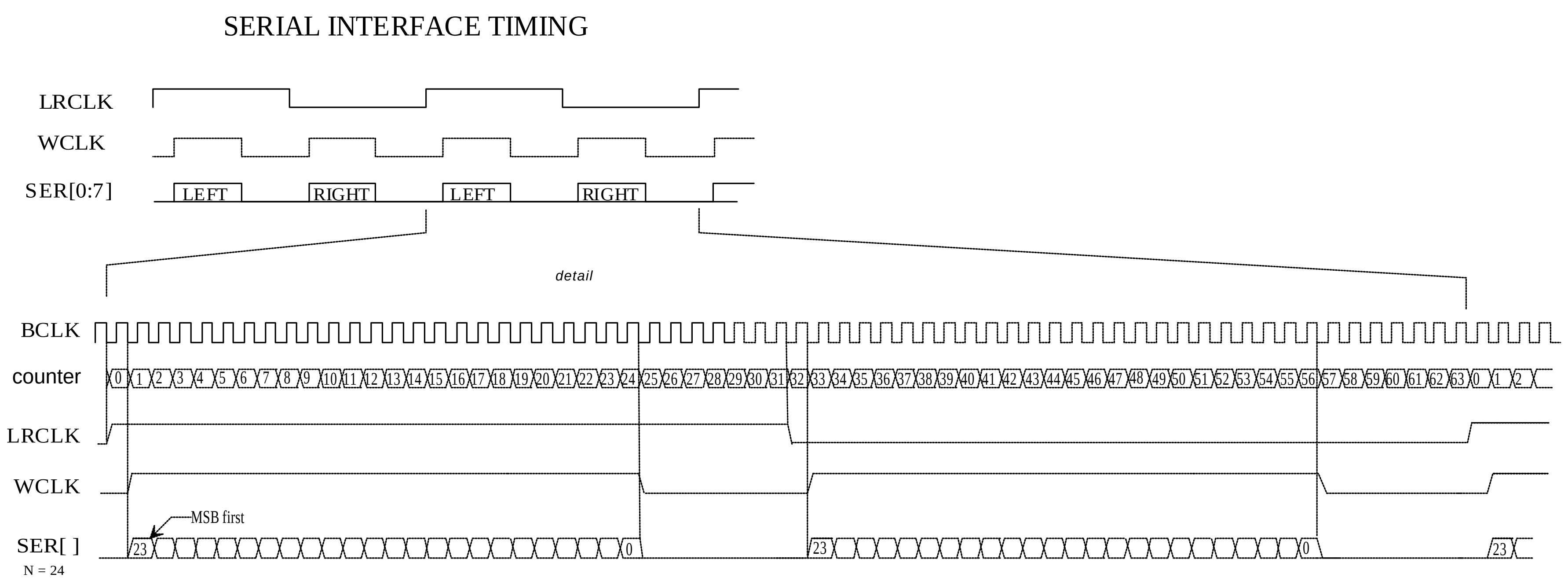


Figure 9. Example Serial Interface Clock Cycle.

Given: 64 (**Mod**) bit-clocks (BCLK) per sample period, N-bit data, LRCLK output 50% duty cycle, I^2S justification.

Serial interface control register values = BCLK counter value before desired transition.

0 <= all serial interface control register values < Mod . Mod >= 2 N + 2.

Mod bit clocks are <= actual sample period.

To reverse the SER[] bit stream, use ALU's BREV instruction.

```
WCLKL_ST     = 0,  (not equal to Mod - 1)
WCLKL_END   = WCLKL_ST + N  <=  LRCLK_END
LRCLK_END   = 31,  (N  <=  LRCLK_END  <  LRCLK_MOD)
WCLKR_ST    = 32,  (LRCLK_END  <  WCLKR_ST  <=  LRCLK_MOD - N)
WCLKR_END   = WCLKR_ST + N  <=  LRCLK_MOD
```

LRCLK_MOD = Mod - 1

8.6. Serial Interface Rules

- It is recommended that SER data access **not** be programmed on the queued instruction line following a BIOZ instruction. This is because the IOZ input pin is often connected to the signal LRCLK which latches the SER data SPRs. The queued instruction line following BIOZ is executed prior to suspension; this means that a SER access there might get old data.

- It is further recommended that all SER data access occur near the top of the program. The synchronization instruction, BIOZ, allows program main loops to occasionally exceed the sample period. SER data access at the end of a main loop might miss the launch window under that circumstance.

- Serial data is always left-justified within 24-bit SER data SPRs.

- The WCLK signal <u>always</u> frames the SER[] data. The ESP2 <u>always</u> requires the WCLK signal, either from an exogenous source or self-generated.

- In general, the transitions of LRCLK, WCLK, the BCLK counter, and SER[] data all occur on the negative transition of BCLK. Each SER[] data bit is latched on the positive transition of BCLK.

- If the serial interface control registers are set up for an N-bit transmission or reception frame, then
24-N LSBs of the SER data SPRs will be transmitted or received as logical zero regardless of their actual value.

8.7. Serial Reset and Synchronization

In revision three of the chip, two bits are added to the hard_conf register. These bits are the SERIAL_RESET, and SERIAL_SYNC bits. The bits are added to facilitate the initialization and synchronization of the two independent timing generators when either one or both are driven by the ESP2.

The SERIAL_RESET bit can be set to put the serial timing generator into a known state in the event when the serial timing control registers are modified. When this pin is set, the bit clock counters are reset to 0 and the clocks signals are driven in the corresponding state, LRCLK[0:1] = 1, WCLK[0:1] = 0. The BCLK[0:1] pins are also held low. Once the bit is cleared, normal functionality is resumed. This bit can be very useful for production testing of the chip.

The SERIAL_SYNC bit can be used to synchronize serial timing generator 1 to the low to high transition of serial timing generator 0. This is very useful in the event that timing generator 1 is running at a multiple of timing generator 0's rate, especially if timing generator 0 is driven from an external source. Whenever it is desired that one timing generator run at a multiple of the base sample rate, timing generator 0 should be set to the base sample rate, timing generator 1 should be set to the multiple rate, and the SERIAL_SYNC bit should be set to synchronize the two generators.

9. Halt and Suspension States

9.1. Halting the Chip

The chip can be halted unconditionally by writing the HOST_HALT bit of the HOST_CNTL interface register. The HOST_CNTL interface register is accessible directly from the system host, and is also accessible (in a limited way) to the ALU and MAC unit as an SPR (HOST_CNTL_SPR).

A mechanism for host-supervised halting under ESP2 program control exists through the HALT pseudo instruction of the ALU. That instruction is used to set the ESP_HALT bit in HOST_CNTL_SPR. If the ESP_HALT_EN bit (located in the HOST_CNTL interface register) is set by the system host, the chip will enter into a halt state when a HALT pseudo is encountered. If the ESP_HALT_EN bit is low, then the ESP_HALT bit is ignored. Program debugging is facilitated by placing HALT pseudo instructions throughout a program as breakpoints. The breakpoints can be activated by the host setting the ESP_HALT_EN bit, or they can be deactivated to allow continuous execution of the program. Recall from the ALU's HALT pseudo instruction description that there is a 1 instruction cycle latency in the execution of the HALT. This results in the execution of the queued instruction line following HALT before the chip enters into the halt state.

The chip can be taken out of the halt state by having the host clear the ESP_HALT and HOST_HALT bits of the HOST_CNTL interface register.

At run-time, when the IOZ_EN bit of the HOST_CNTL interface register is high, the BIOZ instruction typically causes some amount of time spent in suspension depending upon the program's length with respect to the presumably longer sample period. A less obvious method for 'halting' the chip, then, is to bring the IOZ_EN bit low. By doing so, the host disables ESP2's observation of IOZ input pin activity subsequent to this IOZ_EN event, hence suspending program execution indefinitely after a BIOZ instruction is encountered. This **indefinite suspension** occurs after a BIOZ instruction because the IOZ status bit in the CCR will never <u>transit</u> high, when the enable bit IOZ_EN is low, to allow the program to resume. **(**The actual suspension takes place after either the first or second BIOZ instruction following the event, depending upon whether the IOZ status bit was already set at the time of the event.**)**

The chip can be taken out of an indefinite suspension by having the host set the IOZ_EN bit of the HOST_CNTL interface register.

9.2. Chip State during Halt and Suspension

To preserve the state of the chip during a halt or BIOZ suspension, the MAC unit, ALU, and AGEN must perform operations which are nondestructive to the state of GPRs, SPRs, and AORs. As long as the INST_REF_DIS bit of the HARD_CONF SPR is <u>not</u> asserted, micro-instructions will always be refreshed regardless of whether the chip is in a halt/suspension state. The SPRs are fabricated using static memory, so refresh is not an issue there. The GPRs and AORs, however, are fabricated using dynamic memory, so refresh of those registers must be considered.

While halted or suspended the PC is frozen, therefore execution will resume at the next unexecuted instruction line in the queue. The HALT_JUMP bit, in the HOST_CNTL interface register and in HOST_CNTL_SPR, monitors the enforcement of normal run-time latency when the ALU instruction queued for execution following halt/suspension **(**HALT

(ESP_HALT), HOST_HALT, or BIOZ**)** is from the JMP class (Jcc, JScc, RScc). In these cases this HALT_JUMP monitor bit will go high. When the program resumes, HALT_JUMP will automatically clear. While the chip is halted or indefinitely suspended, the PC may be loaded via the standard host/ESP2-register interface for the purpose of vectoring the program coming out of halt/suspension. If the PC is loaded via the interface at this time, the HALT_JUMP bit is automatically cleared.

System host loading of the PC does not take the chip out of the halt nor the suspension states. It is **not** recommended to load the PC from the host interface at run-time because the results are indeterminate. It is neither recommended to load the PC from the host interface during suspension unless the suspension is indefinite.

The MAC unit executes its NOP or RFSH pseudo instruction based on the HALT_MAC_REF bit of the HARD_CONF SPR (0 or 1 respectively). Only when the MAC unit executes NOPs is the state of MACRL **(**the low 24 bits of the 48-bit MAC unit result**)** preserved; when the MAC unit executes RFSH, MACRL is destroyed. It is not critical during halt or suspension for the MAC unit to be involved with internal register refresh.

The ALU executes either a HOST instruction or a HOSTNORFSH pseudo instruction based on the state of the HALT_REF_DIS bit in the HARD_CONF SPR (0 or 1 respectively). Both of these instructions allow host access to internal registers during halt or suspension. HOST provides internal register refresh when no access is pending. The assertion of HALT_REF_DIS disables internal register refresh and always overrides HALT_MAC_REF. (This is not desirable.)

The AGEN executes its NOP instruction which tristates the external memory buss and the pin signal, MR/W\ . Refresh of external DRAM occurs if the XMREF_DIS bit in the HARD_CONF SPR is low.

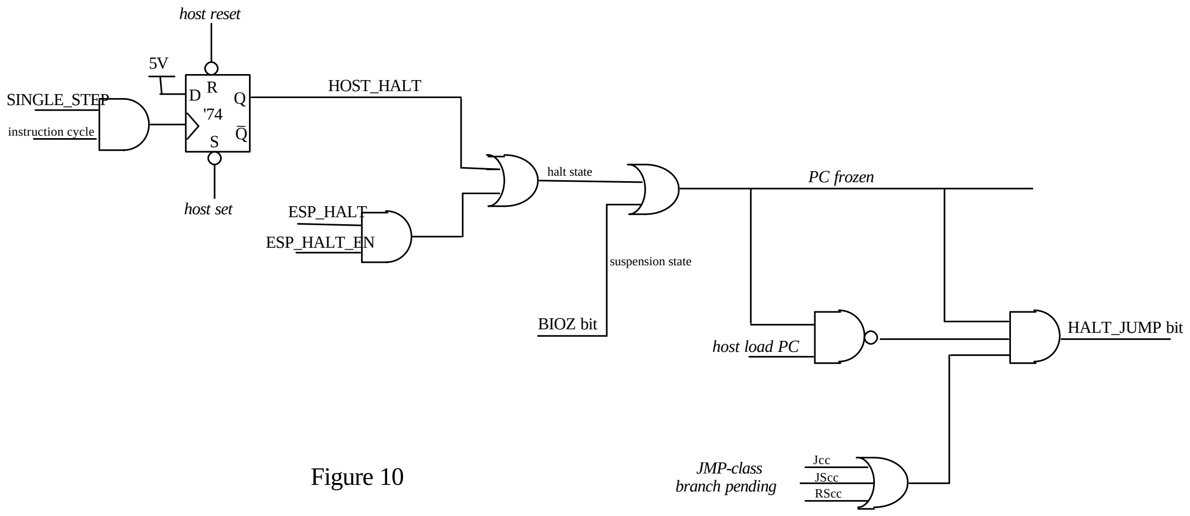


Figure 10

9.3. Single Step Mode

A program debugging tool exists which allows Single Stepping of programs. When the SINGLE_STEP bit of the HOST_CNTL interface register is set, the ESP2 will execute the next queued single instruction line and then enter a halt state by setting the HOST_HALT bit of the HOST_CNTL register. The host must then clear the HOST_HALT bit in order to have the chip execute the next queued instruction line.

Single Stepping can yield erroneous results since the normal inter-unit **latencies will disappear.**

The disappearance of latency warrants consideration of the impact upon all latent instructions:

Normal run-time instruction cycle execution latency of the JMP class (Jcc, JScc, RScc) instructions allows the next queued instruction line to be executed prior to the actual branch. The HALT_JUMP bit of HOST_CNTL_SPR and the HOST_CNTL interface register comes into play here monitoring the enforcement of normal run-time execution latency during Single Step. So, there is no difference in the behavior of JMP class instructions during Single Step.

Recall that normal run-time execution latency of BIOZ (or BIOZNORFSH) and HALT dictates that the following queued instruction line execute prior to suspension or halt. During Single Step, the instruction cycle execution latency of BIOZ will disappear. This means that the instruction line queued for execution following BIOZ will **not** execute prior to suspension. In other words, if suspension occurs, the PC will stay frozen for subsequent Steps at the location of the first instruction line queued following BIOZ. When suspension terminates, the program proceeds from that location. This is in opposition to the normal run-time execution latency of BIOZ where the PC becomes frozen at the **second** queued line during suspension.
The same considerations apply to the HALT pseudo.

There will be a problem if the Single Step encounters a REPT instruction of the second form having a one-line loop. See the section on Latent Instructions for more.

9.4. Single Pass Operation

An extremely useful tool for ESP2 program debugging is Single Pass execution. 'Single Pass' refers to all those instructions that would normally execute during one sample period. This method of debug is recommended in preference to the Single Step mode. Single Pass can be accomplished quite easily using the IOZ status and IOZ_EN bits of the HOST_CNTL interface register.

Assuming that the program has a BIOZ instruction for maintaining sample rate synchronization, the chip can be made to perform a Single Pass by holding the IOZ_EN bit low. When this bit is cleared, the ESP2 will ignore <u>subsequent</u> activity on the IOZ input pin. Therefore, after the BIOZ instruction is executed, ESP2 will remain in a suspension state indefinitely. The host can then set the IOZ status bit of the HOST_CNTL interface register, which has the same effect as a low to high transition at the IOZ input pin, which will cause the program to resume execution. The IOZ status bit will then be automatically cleared by the ESP2 so that the chip will again enter and remain in suspension the next time the BIOZ instruction is reached.

The chip is taken out of suspension by a high IOZ status bit in the HOST_CNTL interface register.

10. Chip Reset, Initialization, and Synchronization

10.1. Reset

The chip has a reset pin (RES \) used for power-up initialization of the chip. Reset will have the following effects on the chip state:

1. The HOST_CNTL interface register and HOST_CNTL_SPR are set to $04 (chip halted).
   This action clears the IOZ status bit, the BIOZ bit, and the IOZ_EN bit.

2. HARD_CONF SPR is reset to zero to:
   1**)** make all serial interface clock pins **(**LRCLK[0:1], BCLK[0:1], and WCLK[0:1]**)** go into high
   impedance input mode, set BCLK divide rates to /2,
   2**)** enable instruction refresh, and enable GPR/AOR refresh from the ALU only,
   3**)** select linear (SRAM) addressing mode, and disable magnitude truncation to external memory,
   4**)** CAS \ before RAS \ external memory refresh enabled.
   5) The OFLAG pin is tristated to be compatible with pre revision 3 chips.
   6) SERIAL_RESET and SERIAL_SYNC are off (low) to be compatible with pre-revision 3 chips.

3. External memory buss (address and data) and pin signal, MR/W \, enter tristate.

4. SER_CONF SPR, all bits set to logical 1. This places all SER[0:7] lines in the high impedance (input) state, and selects timing-set number 1 for all.

5. Serial interface control's BCLK counters reset.

6. The HOST_GPR_PEND bit of the HOST_GPR_CNTL interface register, and the HOST_INST_PEND bit of the HOST_INST_CNTL interface register are cleared.

7. The PC is cleared.

8. The internal clock divider (/4 in Figure 8) circuit is initialized on the low to high transition of RES \. This allows synchronization to tester hardware for manufacturer chip debug.

9. The REPT_CNT SPR is cleared.

10.2. Initialization

Some SPRs may require initialization, while others power-up in unknown states. All SPR/GPR/AOR initialization can be done in the ESP2 program declarations; this is the recommended way although the system host could also initialize the same registers. SPRs associated with the internal memory refresh mechanism (REFPT, REF, or REFINST) need not be initialized, except for manufacturer test purposes.

1)SER0L, SER0R, SER1L, ..., SER7R

Clear the serial data registers being used.

2)SCLK**0**_REG0,1, SCLK**1**_REG0,1

Settings of various serial interface clock timings. These registers require initialization only if any of the serial clock pins will be activated.

3)SER_CONF

Configures serial interface data lines, direction and timing set association.

4)HARD_CONF

Configures serial interface clock pins (activation and BCLK divide rate), external memory type and truncation, and internal/external refresh modes.

When changing serial pins from input to output at initialization (SER_CONF SPR), it is recommended that LRCLK and BCLK first be activated with WCLK held low. This ensures that the cleared data in the SER0L, SER0R, ..., SER7R SPRs is transferred into the output buffers at the low to high transition of LRCLK before the contents of the buffers is output onto the serial pins during WCLK high time. WCLK can be held low by setting the WCLK_ST and WCLK_END values greater than the LRCLK_MOD value (SCLK0_REG0,1 SCLK1_REG0,1 SPRs) and enabling the word clock output pins (HARD_CONF SPR). Once a couple of sample periods have passed, WCLK can be programmed for normal operation.

SERIAL_RESET, and SERIAL_SYNC can be used to synchronize the start of serial timing generators.

5)REPT_CNT

This SPR **must** forever be initialized to **zero** or bad stuff will happen (*JonD*). Whenever a program is overlaid, started, restarted, etc., the REPT_CNT SPR should be cleared. This is easily accomplished sometimes simply by declaration. REPT_CNT controls the automatic looping mechanism for the REPT instruction, and can become activated even if that instruction is never utilized.

6)PC

The Program Counter should be initialized to the desired value by the programmer in the declarations as any other SPR. The PC is in the zero state at reset (RES \). The PC can be initialized by the system host during halt or indefinite suspension. The PC should **not** be initialized at run-time by the host because the results are indeterminate. To vector a running program at run-time, see the Applications section.

7)GPR/AOR

The programmer must zero all application-critical registers in the declarations.

8)HOST_CNTL host interface register.

This control register is customarily initialized from the host side. IOZ_EN is usually set, to activate observation of the IOZ input pin by the ALU BIOZ instruction. Typically, ESP_HALT_EN is set to allow the ESP2 to exercise the HALT pseudo instruction. ESP_HALT and HOST_HALT are customarily cleared to start the processor.

10.3. Synchronization.

In systems using multiple ESP2s it may be necessary to ensure that their internal clock dividers which divide the input clock into the 4 clock instruction cycle are synchronized to one another. This is absolutely necessary if multiple ESP2s are communicating on a common external memory bus. For this reason, the IFLAG and OFLAG pins have dual functionality which can be activated via the SYNC_MODE[1:0] bits of the HARD_CONF register. The following truth table indicates the various configurations of the IFLAG and OFLAG pins.

Table 10. SYNC_MODE bit functionality

sync_mode[1:0]	OFLAG pin function	IFLAG pin function	Description
0	Tristate output	IFLAG input	Rev 2 compatibility mode
1	OFLG bit	IFLAG input	Semaphore mode
2	Sync pulse	IFLAG input	Sync master mode
3	Sync pulse	Sync input	Sync slave mode

The OFLG bit is bit 23 of the HARD_CONF SPR. Therefore, the state of the OFLAG pin can be altered by write to the HARD_CONF SPR. As discussed in the previous section on reset, the HARD_CONF is cleared by the RESB pin. This places the IFLAG into standard input mode, and tristates the OFLAG for backward compatibility with earlier versions of the chip. Once the chip has been reset, the functions of these pins can be changed in order to synchronize multiple ESP2s. For this to be accomplished, one ESP2 will serve as the sync master and the remainder will serve as sync slaves. The SYNC_MODE bits of HARD_CONF register of the master are set to 2. This will cause a sync pulse to be output on the OFLAG pin. The SYNC_MODE bits of the slaves are set to 3 which will cause them to synchronize their internal divider to transitions on their IFLAG pin. By wiring the OFLAG pin of the master to the IFLAG pin of the slave, and programming the registers as mentioned, synchronization can be accomplished. The registers should be held in the above states for a minimum of 2 to 3 instructions cycles after which, the SYNC_MODE pins can be changed to other modes. When taking the chips out of synchronization mode, it is suggested that all slaves be switched from mode 3 to either modes 0 or 1, before the master is switched out of mode 2.

Since mode 3 synchronizes to transitions on the IFLAG and also outputs a sync pulse matching the state of its internal divider, daisy chaining of ESP2s together is also possible. The intent is to accomodate synchronization in whatever wiring configuration is deemed to be most useful for semaphore communication using IFLAG and OFLAG under normal operating conditions. In daisy chaining configurations where one slave will sync to the master and then pass a corrected sync pulse to the next slave, more than 2 to 3 instruction cycles may be necessary to ensure that the whole chain has locked to the master. As a general rule, 2 to 3 cycles per link in the chain are probably sufficient.

11. Special Purpose Registers

All SPRs are read-writable unless specified otherwise. All registers less than 24 bits in width are right justified having the unused bits read as zero. Reading a 'reserved' register returns unspecified data. Following is a map of all the SPRs:

Address	SPR Name	Width	Register Purpose
$3ff	ZERO	24	Zero Constant. Read-only.
$3fe	REFINST	10	Instruction Refresh counter
$3fd	REFPT	10	GPR and AOR Refresh Pointer.
$3f1	REF	n/a	Indirect refresh address via refresh counter REFPT
$3fc	INDIRA	10	Indirect address pointer, A source operand
$3fb	INDIRB	10	Indirect address pointer, B source operand
$3fa	INDIRC	10	Indirect address pointer, C destination operand
$3c7	INDIRD	10	Indirect address pointer, D source operand
$3c6	INDIRE	10	Indirect address pointer, E source operand
$3c5	INDIRF	10	Indirect address pointer, F destination operand
$3c4	INDIRG	10	Indirect offset pointer, AGEN G-operand
$3f9	INDIRECT	n/a	Indirect address mode
$3f8	INDIRINC	n/a	Indirect address mode with post-increment
$3f0	INDIRDEC	n/a	Indirect address mode with post-decrement
$3f7	SCLK**0**_REG0	24	Serial interface control registers combined to define set **0** left channel timing and LRCLK0 end. bits: 23:16 LRCLK_END**0**; 15:8 WCLKL_END**0**; 7:0 WCLKL_ST**0**
$3f6	SCLK**0**_REG1	24	Serial interface control registers combined to define set **0** right channel timing and BCLK0 modulus-1. bits: 23:16 LRCLK_MOD**0**; 15:8 WCLKR_END**0**; 7:0 WCLKR_ST**0**
$3f5	SCLK**1**_REG0	24	Serial interface control registers combined to define set **1** left channel timing and LRCLK1 end. bits: 23:16 LRCLK_END**1**; 15:8 WCLKL_END**1**; 7:0 WCLKL_ST**1**
$3f4	SCLK**1**_REG1	24	Serial interface control registers combined to define set **1** right channel timing and BCLK1 modulus-1. bits: 23:16 LRCLK_MOD**1**; 15:8 WCLKR_END**1**; 7:0 WCLKR_ST**1**

$3f3	HARD_CONF	24	Hardware Configuration	
			bit:	
			1:0	BCLK**0** divide rate
			2	BCLK**0** enable (I/O, 0/1)
			3	WCLK**0** enable (I/O, 0/1)
			4	LRCLK**0** enable (I/O, 0/1)
			6:5	BCLK**1** divide rate
			7	BCLK**1** enable (I/O, 0/1)
			8	WCLK**1** enable (I/O, 0/1)
			9	LRCLK**1** enable (I/O, 0/1)
			10	MUX_ADDR (high active)
			11	MAG_TRUNC (high active)
			12	TRUNC_WIDTH (high = 24-bit, low=16-bit)
			13	HALT_REF_DIS (high active)
			14	INST_REF_DIS (high active)
			15	HALT_MAC_REF (high active)
			16	XMREF_DIS (high active)
			17	XMREF_CASDIS (high active)
			18	SERIAL_RESET (high active)
			19	SERIAL_SYNC (high active)
			20	reserved
			21	SYNC_MODE[0]
			22	SYNC_MODE[1]
			23	OFLG bit

$3f2	SER_CONF	16	Serial Data Line Configuration bit: (logical 0 = timing set number zero) 0 data line 0 timing select 1 data line 1 timing select 2 data line 2 timing select 3 data line 3 timing select 4 data line 4 timing select 5 data line 5 timing select 6 data line 6 timing select 7 data line 7 timing select 8 data line 0 direction (0 = output) 9 data line 1 direction 10 data line 2 direction 11 data line 3 direction 12 data line 4 direction 13 data line 5 direction 14 data line 6 direction 15 data line 7 direction	
$3ef	SER0L	24	Serial data line zero left channel (SER data SPR)	
$3ee	SER0R	24	Serial data line zero right channel	
$3ed	SER1L	24	Serial data line one left channel	
$3ec	SER1R	24	Serial data line one right channel	
$3eb	SER2L	24	Serial data line two left channel	
$3ea	SER2R	24	Serial data line two right channel	
$3e9	SER3L	24	Serial data line three left channel	
$3e8	SER3R	24	Serial data line three right channel	
$3e7	SER4L	24	Serial data line four left channel	
$3e6	SER4R	24	Serial data line four right channel	
$3e5	SER5L	24	Serial data line five left channel	
$3e4	SER5R	24	Serial data line five right channel	
$3e3	SER6L	24	Serial data line six left channel	
$3e2	SER6R	24	Serial data line six right channel	
$3e1	SER7L	24	Serial data line seven left channel	
$3e0	SER7R	24	Serial data line seven right channel	
$3df	BASEI	24	Region I base	
$3de	SIZEM1I	24	Region I size less 1	
$3dd	ENDI	24	Region I end	
$3dc	BASEP	24	Region P base	
$3db	SIZEM1P	24	Region P size less 1	
$3da	ENDP	24	Region P end	
$3d9	BASEQ	24	Region Q base	
$3d8	SIZEM1Q	24	Region Q size less 1	
$3d7	ENDQ	24	Region Q end	
$3d6	BASER	24	Region R base	
$3d5	SIZEM1R	24	Region R size less 1	
$3d4	ENDR	24	Region R end	
$3d3	BASES	24	Region S base	
$3d2	SIZEM1S	24	Region S size less 1	

$3d1	ENDS	24	Region S end
$3d0	BASET	24	Region T base
$3cf	SIZEM1T	24	Region T size less 1
$3ce	ENDT	24	Region T end
$3cd	BASEU	24	Region U base
$3cc	SIZEM1U	24	Region U size less 1
$3cb	ENDU	24	Region U end
$3ca	BASEV	24	Region V base
$3c9	SIZEM1V	24	Region V size less 1
$3c8	ENDV	24	Region V end
$1ff	DIL0	24	Data Input Latch SPR
$1fe	DIL1	24	Data Input Latch SPR
$1fd	DIL2	24	Data Input Latch SPR
$1fc	DIL3	24	Data Input Latch SPR
$1fb	DIL4	24	Data Input Latch SPR
$1fa	DIL5	24	Data Input Latch SPR
$1f9	DIL6	24	Data Input Latch SPR
$1f8	DIL7	24	Data Input Latch SPR
$1f7	DIL8	24	Data Input Latch SPR
$1f6	DIL9	24	Data Input Latch SPR
$1f5	DILA	24	Data Input Latch SPR
$1f4	DILB	24	Data Input Latch SPR
$1f3	DILC	24	Data Input Latch SPR
$1f2	DILD	24	Data Input Latch SPR
$1f1	DILE	24	Data Input Latch SPR
$1f0	DILF	24	Data Input Latch SPR
$1ef	DOL0	24	Data Output Latch SPR
$1ee	DOL1	24	Data Output Latch SPR
$1ed	DOL2	24	Data Output Latch SPR
$1ec	DOL3	24	Data Output Latch SPR
$1eb	DOL4	24	Data Output Latch SPR
$1ea	DOL5	24	Data Output Latch SPR
$1e9	DOL6	24	Data Output Latch SPR
$1e8	DOL7	24	Data Output Latch SPR
$1e7	DOL8	24	Data Output Latch SPR
$1e6	DOL9	24	Data Output Latch SPR
$1e5	DOLA	24	Data Output Latch SPR
$1e4	DOLB	24	Data Output Latch SPR
$1e3	DOLC	24	Data Output Latch SPR
$1e2	DOLD	24	Data Output Latch SPR
$1e1	DOLE	24	Data Output Latch SPR
$1e0	DOLF	24	Data Output Latch SPR

$1df	CCR	9	Condition Code Register	
			bit:	
			0	Zero flag
			1	Less Than flag
			2	oVerflow flag
			3	Carry flag
			4	Negative flag
			5	A operand Negative flag
			6	B operand Negative flag
			7	IFLG (read-only)
			8	IOZ status bit (read-only)
$1de	CMR	10	Condition Mask Register	
			bit:	
			0	Zero mask
			1	Less Than mask
			2	oVerflow mask
			3	Carry mask
			4	Negative mask
			5	A operand Negative mask
			6	B operand Negative mask
			7	IFLG mask
			8	IOZ mask
			9	NOT bit
$1dd	PC	10	Program Counter (writable with caveat)	
$1dc	PCSTACK0	10	PC stack, top	
$1db	PCSTACK1	10	PC stack, 1 below top	
$1da	PCSTACK2	10	PC stack, 2 below top	
$1d9	PCSTACK3	10	PC stack, 3 below top (bottom).	
$1d8	REPT_ST	10	Holds PC value for start of repeat block	
$1d7	REPT_END	10	Holds PC value for end of repeat block.	
$1d6	REPT_CNT	24	Repeat counter for REPT instruction.	
$1d5	ALU_SHIFT	24	ALU shift amount (only the five LSBs are used to control double precision shifts in the ALU).	

$1d4	MACRL	24	MAC Result Low latch (read-only)
$1d3	MACP_H	24	MACP latch high word (write-only)
$1d3	MACH	24	MAC latch high word (read-only)
$1d2	MACP_HC	24	MACP latch high word - clear low word (write-only)
$1d1	MACP_L	24	MACP latch low word (write-only)
$1d1	MACL	24	MAC latch low word (read-only)
$1d0	MACP_LS	24	MACP latch low word - sign extend into high word (write-only)
$1cf	HOST_GPR_DATA	24	GPR/SPR/AOR data access by host
$1ce	HOST_CNTL_SPR	8	Image of the HOST_CNTL host interface register for internal access to status and control bits. All high active. bit: 0 ESP_HALT_EN (read-only) 1 ESP_HALT 2 HOST_HALT (read-only) 3 IOZ status bit 4 IOZ_EN (read-only) 5 BIOZ bit (read-only) 6 SINGLE_STEP (read-only) 7 HALT_JUMP (read-only)
$1cd	HALF	24	Read-only constant register containing the hex value $400000. This value is necessary for some MAC unit pseudo instructions.
$1cc	ONE	24	Read-only constant register containing the hex value $000001.
$1cb	MINUS1	24	Read-only constant register containing the hex value $800000.
$1ca		n/a	reserved address available for future use
$1c9		n/a	reserved (tied in with $1c8 since HOST_ESP_FACE is accessed over the GPR busses which accesses two registers at a time and column multiplexes the busses).
$1c8	HOST_ESP_FACE	24	SPR mapping of the HOST_ESP_FACE_B2,1,0 host interface registers. This register is uncommitted.

11.1 SPR Descriptions

ALU_SHIFT is an SPR which supplies a shift amount (an extra source operand) to the ALU during the execution of the ASDH, ASDL, LSDH, and LSDL instructions. Regarding its availability to the ALU for its prescribed purpose, loading ALU_SHIFT follows the same inter-unit latency rules as any other GPR/AOR destination.
ALU_SHIFT is read/writable and could be used like a GPR in programs where the double precision shift instructions are not required.

BASE, END, and SIZEM1 registers are used by the Address Generator for accessing up to eight regions in external memory. Each of the regions has a programmable size and location in physical memory. See the AGEN description for further information.

CCR and CMR are the Condition Code and Condition Mask Register and are concerned with conditional execution of instructions. The CCR is set automatically only in the ALU. These two SPRs are readable and writable with the exception of the IOZ status bit and IFLG which are read-only in the CCR. For further information see the description of the Conditional Execution Mechanism in the section on the Condition Code Register.

The DIL and DOL registers are the data interface to external memory. Data coming into the chip is latched in the designated DIL register. Data to be written by the chip must be placed in one of the DOL registers. These SPRs are readable and writable like GPRs. See the sections on the External Memory Interface for more information on the use of these registers.

The HALF register is a read-only constant containing the value $400000. This value is necessary for some MAC unit pseudo instructions and has therefore been hardwired into an SPR.

The HARD_CONF register is an 18-bit SPR used to configure multifarious aspects of the chip hardware. Upon reset, this register will initialize to 0.
The LSBs of HARD_CONF determine whether each serial clock pin acts as input or output, and determine the divide-down rates of BCLK0 and BCLK1 (00 corresponds to /2, 01 <=> /4, 10 <=> /8, 11 <=> /1). When the individual serial clock pins are <u>not</u> enabled, they are in the high impedance input mode.
The MUX_ADDR, MAG_TRUNC, and TRUNC_WIDTH bits are readable and writable in this register. When the MUX_ADDR bit is low, this yields the linear external addressing mode suitable for SRAM or peripheral I/O. When MUX_ADDR is high, multiplexed addressing suitable for DRAM is activated. Assertion of MAG_TRUNC invokes the magnitude truncation mode of WR to external memory. TRUNC_WIDTH low selects magnitude truncation to 16 bits, high selects to 24 bits.
The INST_REF_DIS (instruction refresh disable), HALT_REF_DIS (halt refresh disable), and HALT_MAC_REF (halt MAC unit refresh) bits are useful for controlling the automatic refresh of instructions and internal registers. See the section on Internal Memory Refresh for details.
The XMREF_DIS (external memory refresh disable) and XMREF_CASDIS (external memory refresh CAS \ disable) bits control the refresh mode of external memory. The default mode provides CAS \ before RAS \ refresh. Setting XMREF_CASDIS provides RAS \-only refresh. Setting XMREF_DIS disables external memory refresh altogether regardless of the setting of XMREF_CASDIS.
Setting the SERIAL_RESET bit will keep the serial clocks in a reset state in which both WCLK pins are low, both LRCLK pins are high, and both BCLK pins are low. This can be

useful for resetting the clocks after changing registers that control the period and duty cycle of the WCLK and LRCLK pins. Setting SERIAL_SYNC causes serial timing generator 1 to reset on the low to high transition of LRCLK[0] as well as LRCLK[1]. This can be useful for synchronizing the two timing generators when timing generator 1 is running at an integer multiple of the sample rate set by timing generator 0.

SYNC_MODE[1:0] and OFLG are used in conjunction with the OFLAG and IFLAG pins to provide instuctions synchronization between multiple ESP2s. (See the section on synchronization for a description of the function of these bits).

HOST_CNTL_SPR is an SPR image of the HOST_CNTL host interface register. All of the bits of this register can be read by the ESP2 program to allow access to these hardware status and control bits. For a description of this register refer to the section on Halting the Chip; also the description of the HOST_CNTL interface register.

The ESP_HALT bit can be written. This is necessary for the proper execution of the HALT pseudo instruction in the ALU.

The IOZ status bit appears in the CCR and can easily be polled. Detecting and clearing a set IOZ status bit can be accomplished automatically by executing the BIOZ instruction. Alternately, the IOZ status bit can be written via this SPR to allow manual program control.

The HOST_ESP_FACE register is a concatenation of three 8-bit host interface registers which are made accessible as an SPR to ESP2 instructions. Like HOST_GPR_DATA, this SPR is unusual in the respect that it is directly accessible from both the host interface and the ESP2 without the need to execute HOST or BIOZ instructions. What differentiates HOST_ESP_FACE from HOST_GPR_DATA is that it is uncommitted; meaning, the programmer is free to utilize this resource in any opportune manner. Since one of the purposes of the host interface is host access of GPRs/AORs/SPRs, this register could be accessed like any other SPR using the mechanism described in the Host/ESP2 Interface description. But this is an idiosyncrasy of the design which is not the intended use of this register.
One possible use of HOST_ESP_FACE would be in a DMA (direct memory access) scheme where the ESP2's external memory were being quickly loaded (or read). This SPR would be written by some exogenous mechanism, then a tight ESP2 loop would write that data out. The IFLAG and OFLAG pins might act as the semaphores in this scheme.

The HOST_GPR_DATA register is mapped as an SPR to provide the link from the outside world **(**the system host**)** to all the internal ESP2 registers (GPR/SPR/AOR). This register is the 24-bit SPR mapping of the concatenated 8-bit HOST_GPR_DATA_B2,1,0 host interface registers. The ESP2 automatically moves pending host data to and from this SPR during ALU HOST instruction execution.

The INDIRA,B,C,D,E,F,G SPRs hold indirect addresses. Writing these SPRs allows modification of the indirect address. See the section on Indirect Register Addressing.

The INDIRDEC address is used in the same manner as INDIRECT, but it causes a post-decrement of the corresponding (INDIR) pointer register. The decremented value is ready for the next instruction cycle. If INDIRDEC is used in a skipped instruction, the address will not be decremented.

The INDIRECT address is used to initiate indirect addressing. When this address is used as the A operand address, for example, the contents of the INDIRA pointer register is substituted for the address of that operand. The same happens for the remaining operands substituting the INDIRB, INDIRC, INDIRD, INDIRE, INDIRF and INDIRG SPRs, respectively.

The INDIRINC address is used in the same manner as INDIRECT, but it causes a post-increment of the corresponding (INDIR) pointer register. The incremented value is ready for the next instruction cycle. If INDIRINC is used in a skipped instruction, the address will not be incremented.

MACH and MACL are the most significant and least significant halves of the MAC unit's unsaturated MAC read-only latch.

MACP_H and MACP_L are the most significant and least significant halves of the MAC unit's write-only Preload latch. Since these two registers are write-only and the MACH and MACL registers are read-only, they have been mapped to the same addresses.

MACP_HC is a secondary method of writing the high word of the MAC Preload latch (MACP) which simultaneously clears the MACP_L register.

MACP_LS is a secondary method of writing the low word of the MAC Preload latch (MACP) which simultaneously sign-extends the MSB of the data entering the MACP_L register into the 24-bit MACP_H register.

MACRL is the low 24 bits of the MAC unit result after shifting and saturation.

The MINUS1 register is a read-only constant containing the value $800000. This value is required by some MAC unit pseudo instructions.

The ONE register is a read-only constant containing the value $000001. This value is required by the MAC unit NOP pseudo instruction and has, therefore, been hardwired into an SPR.

PC is the Program Counter. This SPR is readable and writable. At run-time there are severe hazards associated with writing to the PC from the host or as destination of the MAC unit or ALU. But the PC can be written by the host with no caveat while the chip is halted or indefinitely suspended, to initialize the start point of a program. The PC is cleared on chip reset.

PCSTACK0,1,2,3 compose a four deep stack for keeping track of the PC during subroutine calls. JScc pushes the PC value + 2 onto the top of the stack (PCSTACK0), and RScc pops the new PC value from the top of the stack.

REF is a reserved address used during internal register refresh. Specifying this address as a source or destination operand address causes indirect use of the REFPT contents as the operand address. When REF is used as the source and destination operand so as to accomplish refresh, the same contents of REFPT is used for both source <u>and</u> destination. Specifying REF as the source operand of an instruction, REFPT is incremented in preparation for use as a source operand for another refresh operation. The newly incremented value is not available as the destination address until the time at which the current destination address is decoded for the function unit that caused the increment. See section on GPR and AOR Refresh for additional information on the use of these addresses.

REFINST is the 10-bit refresh counter for refreshing instructions. See the section on Internal Memory Refresh for more information.

REFPT is the 10-bit refresh counter for refreshing internal registers, GPR/AOR. See the section on GPR and AOR Refresh for more information.

REPT_ST, REPT_END, and REPT_CNT are used as extra sources by the REPT instruction. When REPT is executed, REPT_ST always gets loaded with the value of the PC for the next instruction line queued for execution. REPT_END always gets loaded with the value from the A operand field of the instruction. REPT_CNT gets loaded with the value from the B operand field or the B operand depending upon the form of the instruction.

SCLK**0**_REG0, SCLK**0**_REG1, SCLK**1**_REG0, and SCLK**1**_REG1 are concatenated registers each containing three 8-bit serial interface control registers used for setting serial clock timings. The SPR map lists the control registers contained within each SPR, while the Serial Interface section describes how those control registers are used.

SER0L, SER0R, SER1L, ... Repositories for serial data transmission to and from other ESP2 chips, A/D and D/A converters, and other serial devices. There are 8 serial stereo data lines (physical pins) having a left and right channel-data SPR associated with each. See the section on the Serial Interface.

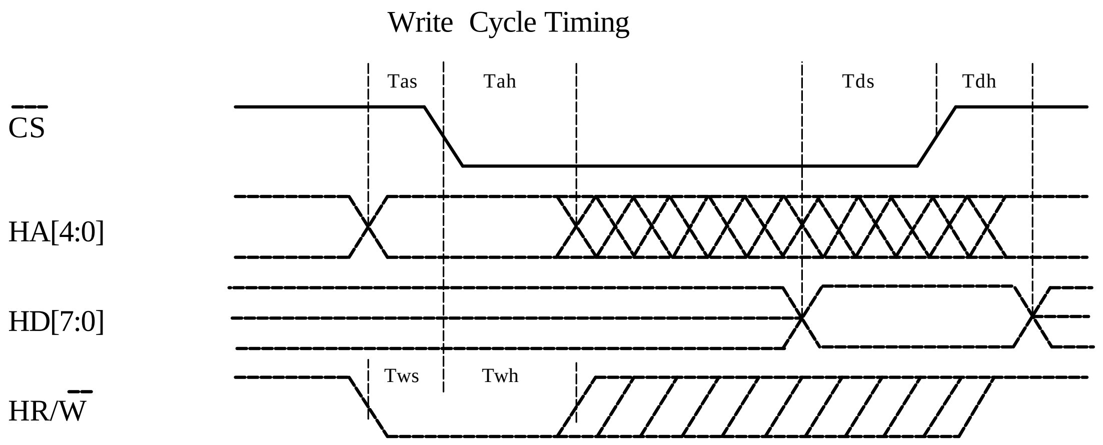
The serial data line configuration register, SER_CONF, is a 16-bit register whose upper byte has one bit for each serial data line, which designates that data line as input or output. A logical 1 in the associated bit indicates that the data line is an input. In the lower byte there is a bit per data line which associates one of two timing sets with that data line: 0 = timing set number 0, 1 = timing set 1. See the SPR map in this section on Special Purpose Registers for the exact definition of bits.

ZERO is the constant value $000000 hardwired into the chip. Since this value is required by many of the pseudo instructions, it has been included as an SPR. It is also used as the *null* destination in cases where the result of an instruction is either undefined or not of interest.

12. Host/ESP2 Interface

The purpose of the host interface is to access all the ESP2 internal registers and instruction memory. There are separate mechanisms for accessing each which are described herewith.

The host interface consists of five address pins HA[4:0], eight data pins HD[7:0], a read/write pin HR/W\, and a chip select pin CS\. HA[4:0] and HR/W\ are latched on the falling edge of CS\. In a read cycle, the ESP2 will assert the HD[7:0] pins with the read-data while CS\ is low. In a write cycle, the HD[7:0] pins will propagate into the addressed register and be latched on the rise of CS\.



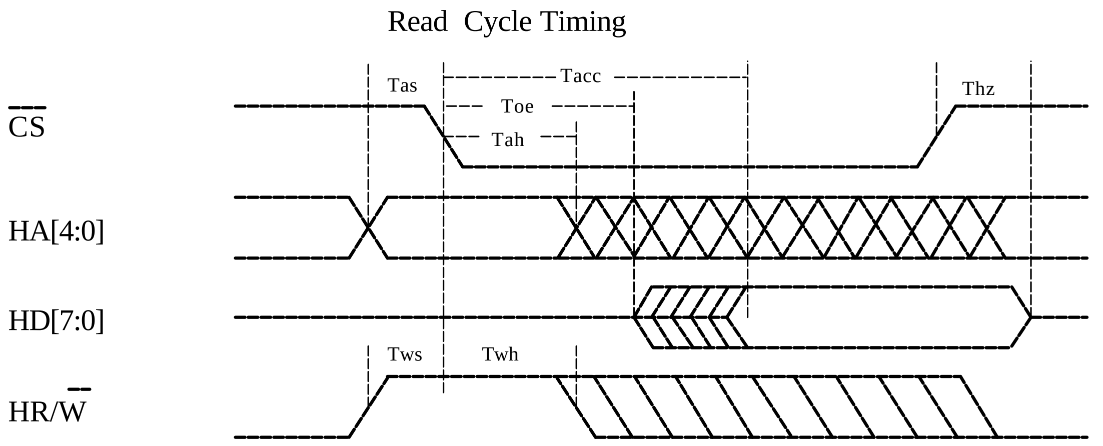


Figure 11.

Tas = Address Setup to CS\ falling edge
Tah = Address Hold from CS\ falling edge
Tws = HR/W\ Setup to CS\ falling edge
Twh = HR/W\ Hold from CS\ falling edge
Tds = Data Setup to CS\ rising edge (write cycle)
Tdh = Data Hold from CS\ rising edge (write cycle)
Toe = CS\ falling edge to Data Active delay (read cycle, output enable)

Tacc = CS \ falling edge to Data Valid (read cycle, access time)
Thz = CS \ rising edge to Data Tristate (read cycle)

12.1. Host/ESP2-Register Interface

The HOST_GPR_PEND bit of the HOST_GPR_CNTL host interface register is at once the semaphore and the command to read or write an internal register (GPR/AOR/SPR). When there is no internal register access pending, internal register refresh typically occurs.

When the chip is not halted and if an access is pending, data will be automatically moved from/to HOST_GPR_DATA **(**the SPR image of the HOST_GPR_DATA_B2,1,0 interface registers**)** to/from an internal register during the first available HOST or BIOZ instruction. In this manner the ALU grants host access along normal ALU data paths. The direction of transfer is specified by the HOST_GPR_RW\ bit of the HOST_GPR_CNTL interface register. Depending on the location and frequency of HOST or BIOZ instructions in the executing program, there is typically some delay before the write to or read of the GPR/AOR/SPR actually takes place. The worst case delay under typical operating conditions is one sample period. When the access has been completed, the HOST_GPR_PEND bit will be cleared. The host will not be forced to hang waiting for the semaphore because the host can poll it.

When the chip is halted or indefinitely suspended and an access is pending, the ALU is continuously executing HOST instructions by design. Therefore there is minimum delay for the transfer from/to the HOST_GPR_DATA SPR. Since the system host presumably runs asynchronous to the internal instruction cycle, the HOST_GPR_PEND bit must still be polled.

12.1.1. Writing GPR/AOR/SPR

A write of a GPR/AOR/SPR via the host interface involves the following steps:

1. Check the HOST_GPR_PEND bit of the HOST_GPR_CNTL interface register to see that it is low meaning that there is no register transfer pending.

2. Write three bytes of data to the HOST_GPR_DATA_B2,1,0 interface registers. **(**If these registers already hold the desired data because of a previous GPR/AOR/SPR-write, this step can be ignored**)**.

3. Write the GPR/AOR/SPR address to the HOST_GPR_ADDR1,0 interface registers.

4. Write $80 into the HOST_GPR_CNTL interface register initiating the write. The HOST_GPR_PEND bit when set constitutes the command to write when the HOST_GPR_RW\ bit is low. By observing this bit, the write event can be monitored because the bit goes low when the write occurs.

12.1.2. Reading GPR/AOR/SPR

A read of a GPR/AOR/SPR via the host interface involves the following steps:

1. Check the HOST_GPR_PEND bit of the HOST_GPR_CNTL interface register to see that it is low meaning that there is no register transfer pending.

2. Write the GPR/AOR/SPR address to the HOST_GPR_ADDR1,0 interface registers.

3. Write $81 to the HOST_GPR_CNTL interface register, setting the HOST_GPR_PEND bit. The HOST_GPR_PEND bit when set constitutes the command to read when the HOST_GPR_RW\ bit is high.

4. Once the HOST_GPR_PEND bit of the HOST_GPR_CNTL interface register has been automatically cleared, read three bytes of data from the HOST_GPR_DATA_B2,1,0 interface registers.

12.2. Host/ESP2-Instruction Interface

The HOST_INST_PEND bit of the HOST_INST_CNTL host interface register is at once the semaphore and the command to read or write one 96-bit instruction. When there is no instruction access pending, instruction refresh typically occurs.

Host access to instruction memory is always allowed on every instruction cycle whether or not the chip is halted; there are no hazards. When an access is pending, one instruction will be automatically moved from/to the HOST_INST_DATA_B11,...,0 interface registers along a separate 96-bit wide data path in a direction specified by the HOST_INST_RW\ bit of the HOST_INST_CNTL interface register. When the access has been completed, the HOST_INST_PEND bit will be automatically cleared. The host will not be forced to hang waiting for the semaphore because the host can poll it. Since the system host presumably runs asynchronous to the internal instruction cycle, it is always necessary to poll the HOST_INST_PEND bit.

12.2.1. Writing Instruction Memory

A write of a single instruction via the host interface involves the following steps:

1. Check the HOST_INST_PEND bit of the HOST_INSTR_CNTL interface register to see that it is low meaning that there is no instruction memory transfer pending.

2. Write twelve bytes of data to the HOST_INST_DATA_B11,...,0 interface registers. **(**If these registers already hold the desired data because of a previous instruction-write, this step can be ignored**)**.

3. Write the instruction address to the HOST_INST_ADDR1,0 interface registers.

4. Write $80 to the HOST_INST_CNTL interface register initiating the write. The HOST_INST_PEND bit when set constitutes the command to write when the HOST_INST_RW\ bit is low. By observing this bit, the write event can be monitored because the bit goes low when the write occurs.

12.2.2. Reading Instruction Memory

A read of a single instruction via the host interface involves the following steps:

1. Check the HOST_INST_PEND bit of the HOST_INST_CNTL interface register to see that it is low meaning that there is no instruction memory transfer pending.

2. Write the instruction address to the HOST_INST_ADDR1,0 interface registers.

3. Write $81 to the HOST_INST_CNTL interface register, setting the HOST_INST_PEND bit. The HOST_INST_PEND bit when set constitutes the command to read when the HOST_INST_RW\ bit is high.

4. Once the HOST_INST_PEND bit of the HOST_INST_CNTL interface register has been automatically cleared, read twelve bytes of data from the HOST_INST_DATA_B11,...,0 interface registers.

12.3. Host Interface Registers

'Reserved' registers are nonexistent; reading from them returns unspecified data.

12.3.1. Testing

For test purposes, one may be interested in writing then verifying host interface registers to determine whether the ESP2 is alive and powered. The host interface registers are designed to be accessed asynchronously from the system host. This design demands a special procedure for testing some of the host interface registers. The HOST_ESP_FACE_B2,1,0 registers can be tested in a straight-forward manner.

When testing HOST_INST_CNTL and HOST_GPR_CNTL, keep in mind that the associated **_**PEND bit is self-clearing. In halt state, the HOST_GPR_PEND bit clears itself quickly.

When testing

HOST_INST_DATA_B11,...,0
HOST_INST_ADDR1,0
HOST_GPR_DATA_B2,1,0
HOST_GPR_ADDR1,0

one must employ the following procedure:
-Write the desired **_**INST**_** or **_**GPR**_** host interface registers from the list above.
-Set the **_**PEND bit in the HOST_INST_CNTL or the HOST_GPR_CNTL register, respectively.
-Wait for the associated **_**PEND bit to clear, then verify data previously written to the selected host interface registers.

Thorough testing of GPRs/AORs, some SPRs, and the instruction (program) memory, themselves, involves the erasure of some hidden internal registers. The correct procedure follows the section Host/ESP2 Interface, with the following augmentation: After the test data is written to some GPR/AOR/SPR or instruction memory, the test data being held in the interface registers must be wiped using different data, and then a write to the read-only SPR called ZERO must occur **(**it is not sufficient to simply wipe out the host interface register data**)**. Then, the test data can be read back from the memory-under-test and verified.

The HOST_CNTL register is tricky to test.

```
/********************************** test HOST_CNTL register **********************************/
error1 = error2 = error = 0;
/*test the halt bit by itself*/
for(i=0; i<LOOPS; i++) {
    host_inyaface->host_cntl = 0;
    temp = host_inyaface->host_inst_data_b4;  /* electrical flack */
    if((host_inyaface->host_cntl & HOST_HALT) != 0) error = 1;

    host_inyaface->host_cntl = HOST_HALT;
    temp = host_inyaface->host_inst_data_b5;  /* electrical flack */
    if((host_inyaface->host_cntl & HOST_HALT) != HOST_HALT) error = 1;
}
mask = 0x7f; /*mask off HALT_JUMP bit*/
logic = 0xd7;
temp1 = (0x55 | HOST_HALT) & mask;
temp2 = (0xaa | HOST_HALT) & mask; /*test performed in halt mode*/
for(i=0; i<LOOPS; i++) {
    host_inyaface->host_cntl = temp1;
    temp = host_inyaface->host_inst_data_b2;  /* electrical flack */
    while((temp = host_inyaface->host_cntl & mask) != temp1) {
        if(error1) break;
        error1 = 1;
        printf("\nerror: host_cntl = %02x\n", temp);
        printf("expected %02x above\n", temp1);
        printf("For this test to work in Rev 1 or higher, the IOZ pin must be jumpered either to MPU_IOZ or GND\n");
    }
    host_inyaface->host_cntl = temp2;
    temp = host_inyaface->host_inst_data_b3;  /* electrical flack */
    while((temp = host_inyaface->host_cntl & mask) != (temp2 & logic)) {
        if(error2) break;    /*when both IOZ and BIOZ bits are high, they reset themselves. */
        error2 = 1;
        printf("\nerror: host_cntl = %02x\n", temp);
        printf("expected %02x above\n", temp2 & logic);
    }
}
if(!error1 && !error2 && !error)printf("HOST_CNTL host interface register is good.\n");
else if(error)printf("\nERROR! HOST_CNTL host interface register has faulty HOST_HALT bit.\n");
else if(error1 || error2)printf("\nERROR! HOST_CNTL host interface register is faulty.\n");
/*********************************************************************************************/
```

Address	**Host Interface Register Name**	Register Usage
$0	HOST_INST_DATA_B11	Byte 11 of instruction
$1	HOST_INST_DATA_B10	Byte 10 of instruction
$2	HOST_INST_DATA_B9	Byte 9 of instruction
$3	HOST_INST_DATA_B8	Byte 8 of instruction
$4	HOST_INST_DATA_B7	Byte 7 of instruction
$5	HOST_INST_DATA_B6	Byte 6 of instruction
$6	HOST_INST_DATA_B5	Byte 5 of instruction
$7	HOST_INST_DATA_B4	Byte 4 of instruction
$8	HOST_INST_DATA_B3	Byte 3 of instruction
$9	HOST_INST_DATA_B2	Byte 2 of instruction
$a	HOST_INST_DATA_B1	Byte 1 of instruction
$b	HOST_INST_DATA_B0	Byte 0 of instruction
$c	HOST_INST_ADDR1	2 MSBs of instruction address
$d	HOST_INST_ADDR0	8 LSBs of instruction address
$e	reserved	
$f	HOST_INST_CNTL	bit 7 - HOST_INST_PEND bit 6:1 - Reserved for use as chip revision number. Reads 2 in Revision 2 of the chip. bit 0 - HOST_INST_RW\

$10	reserved	
$11	HOST_GPR_DATA_B2	High byte of GPR/AOR/SPR register data
$12	HOST_GPR_DATA_B1	Mid byte of GPR/AOR/SPR register data
$13	HOST_GPR_DATA_B0	Low byte of GPR/AOR/SPR register data
$14	HOST_GPR_ADDR1	2 MSBs of GPR/AOR/SPR register address
$15	HOST_GPR_ADDR0	8 LSBs of GPR/AOR/SPR register address
$16	reserved	
$17	HOST_GPR_CNTL	bit 7 - HOST_GPR_PEND bit 0 - HOST_GPR_RW\

$18	reserved	
$19	HOST_CNTL	Host Control register. All bits are active in the high state, readable and writable, except for the HALT_JUMP bit which is read-only. bit: 0 ESP_HALT_EN 1 ESP_HALT 2 HOST_HALT 3 IOZ status bit 4 IOZ_EN 5 BIOZ bit 6 SINGLE_STEP 7 HALT_JUMP (read-only)

$1a	reserved	
$1b	HOST_ESP_FACE_B2	High Byte of HOST_ESP_FACE SPR

$1c	HOST_ESP_FACE_B1	Mid Byte of HOST_ESP_FACE SPR
$1d	HOST_ESP_FACE_B0	Low Byte of HOST_ESP_FACE SPR

12.3.2. Some Host Interface Register Descriptions

The host control register, **HOST_CNTL**, is the only register for controlling the operation of the ESP2 that is directly accessible from the host processor. This read/write host interface register allows the host to recall the hardware status, and provides direct control of the ESP2 regardless of the state of its internal function units. This register is also directly accessible by ESP2 as an SPR in a limited way (refer to the HOST_CNTL_SPR description).

The ESP_HALT and ESP_HALT_EN bits in conjunction with the HALT pseudo instruction provide a means of break-pointing during algorithm execution (refer to the section on Halting the Chip).

The HOST_HALT bit provides a means of unconditionally halting the ESP2 by the host processor regardless of the state of the internal function units.

The BIOZ bit, the IOZ status bit, and the IOZ_EN bit operate in conjunction with the ALU BIOZ instruction and are described in detail in the description of that instruction.

The SINGLE_STEP bit provides a mechanism for the successive execution of single queued instruction lines.

HALT_JUMP is a read-only monitor-bit. It goes high when the chip enters the halt or suspension state **(**HALT (ESP_HALT), HOST_HALT, or BIOZ**) if** the ALU instruction queued for execution is from the JMP class (Jcc, JScc, RScc). When this monitor bit goes high, normal run-time instruction cycle execution latencies are being enforced coming out of halt/suspension. The bit automatically clears when the program resumes.

These bits are discussed further in the section on Halt and Suspension States.

HOST_ESP_FACE_B2,1,0 are uncommitted host interface registers **(**mapped into three consecutive bytes from the host side**)** which are also directly accessible by ESP2 as one (24-bit) SPR called HOST_ESP_FACE.
The HOST_GPR_DATA_B2,1,0, HOST_ESP_FACE_B2,1,0, and HOST_CNTL host interface registers are distinguished in so far as they are simultaneously mapped as SPRs, so can therefore be accessed directly by ESP2. This means that there is <u>no</u> need for the execution of BIOZ or HOST instructions in order that the host be able to communicate with a running ESP2 program via these registers; and vice versa. This is advantageous for fast intercommunication. In the host register space, these registers are mapped as consecutive bytes, whereas in the SPR space they are concatenated into single 24-bit registers.

See the SPR Descriptions for a bit more about **HOST_GPR_DATA_**B2,1,0, the internal register link to the outside world.

13. Pin List

Pin Name	**Description**	**Function**	**Number**
External Memory Interface			
MADDR[23:0]	Address Buss	Output/Tristate	24
MDATA[23:0]	Data Buss	I/O	24
MR/W\	Write Enable	Output/Tristate	1
RAS\	Row Address Strobe	Output	1
CAS\	Column Address Strobe	Output	1
MEM_REQ\	Memory Cycle Request	I/O Tristate	1
VSS_A	Ground to address	Ground	1
VSS_D	buffers	Ground	1
VDD_A	Ground to data buffers	Power	1
VDD_D	Supply to address buffers	Power	1
	Supply to data buffers	*Subtotal*	**56**
Host Interface			
HA[4:0]	Host Address	Input	5
HD[7:0]	Host Data	I/O	8
CS\	Chip Select	Input	1
	(edge sensitive)		
HR/W\	Write Enable	Input	1
VSS_H	Ground to buffers	Ground	1
VDD_H	Supply to buffers	Power	1
		Subtotal	**17**
Serial Interface			
BCLK[1:0]	Bit Clocks	I/O	2
	(edge sensitive)		
WCLK[1:0]	Word Clocks	I/O	2
	(edge sensitive)		
LRCLK[1:0]	Left/Right Clocks	I/O	2
SER[7:0]	Serial Data Lines	I/O	8
VSS_S	Ground to buffers	Ground	1
VDD_S	Supply to buffers	Power	1
		Subtotal	**16**
Miscellaneous			
CLK	System Clock	Input	1
	(four times instruction		
	rate,		
IOZ	40 MHz nominal)	Input	1
	Sample Rate		
	Synchronization (rising		
	edge sensitive, refer to		
	BIOZ instruction)		
IFLAG	Input Flag	Input	1
OFLAG	Output Flag	Output/Tristate	1
RES\	Chip Reset	Input	1
VSS[3:0]	Ground to chip internals	Ground	4
VDD[1:0]	Supply to chip internals	Power	2
		Subtotal	**11**

Grand total	**100**

Table 11. ESP2 Chip Pinout

PIN NUMBER	NAME	FUNCTION	comment
1	resb	Input	
2	clk	Input	
3	iflag	Input	
4	VSS[0]	Ground	Connect to VSS frame
5	oflag	Output/Tristate	
6	VDD_S	Power	
7	bclk0	I/O	
8	wclk0	I/O	
9	lrclk0	I/O	
10	bclk1	I/O	
11	wclk1	I/O	
12	lrclk1	I/O	
13	ser7	I/O	
14	ser6	I/O	
15	ser5	I/O	
16	ser4	I/O	
17	ser3	I/O	
18	ser2	I/O	
19	ser1	I/O	
20	ser0	I/O	
21	VSS_S	Ground	
22	VSS[1]	Ground	Connect to VSS frame
23	VDD[0]	Power	
24	VDD_A	Power	
25	mrwb	Output/Tristate	
26	maddr[23]	Output/Tristate	
27	maddr[22]	Output/Tristate	
28	maddr[21]	Output/Tristate	
29	maddr[20]	Output/Tristate	
30	maddr[19]	Output/Tristate	
31	maddr[18]	Output/Tristate	
32	maddr[17]	Output/Tristate	
33	maddr[16]	Output/Tristate	
34	maddr[15]	Output/Tristate	

35	maddr[14]	Output/Trist ate
36	maddr[13]	Output/Trist ate
37	maddr[12]	Output/Trist ate
38	maddr[11]	Output/Trist ate
39	maddr[10]	Output/Trist ate
40	maddr[9]	Output/Trist ate
41	maddr[8]	Output/Trist ate
42	maddr[7]	Output/Trist ate
43	maddr[6]	Output/Trist ate
44	maddr[5]	Output/Trist ate
45	maddr[4]	Output/Trist ate
46	maddr[3]	Output/Trist ate
47	maddr[2]	Output/Trist ate
48	maddr[1]	Output/Trist ate
49	maddr[0]	Output/Trist ate
50	VSS_A	Ground

PIN NUMBER	NAME	FUNCTION	comment
51	VSS_D	Ground	
52	mdata[23]	I/O	
53	mdata[22]	I/O	
54	mdata[21]	I/O	
55	mdata[20]	I/O	
56	mdata[19]	I/O	
57	mdata[18]	I/O	
58	mdata[17]	I/O	
59	mdata[16]	I/O	
60	mdata[15]	I/O	
61	mdata[14]	I/O	
62	mdata[13]	I/O	
63	mdata[12]	I/O	
64	mdata[11]	I/O	
65	mdata[10]	I/O	
66	mdata[9]	I/O	
67	mdata[8]	I/O	
68	mdata[7]	I/O	
69	mdata[6]	I/O	
70	mdata[5]	I/O	
71	mdata[4]	I/O	
72	mdata[3]	I/O	
73	mdata[2]	I/O	
74	mdata[1]	I/O	
75	mdata[0]	I/O	
76	VDD_D	Power	
77	rasb	Output	
78	casb	Output	
79	mem_reqb	I/O	
80	ioz	Input	
81	csb	Input	
82	hrwb	Input	
83	VSS[3]	Ground	Connect to VSS frame
84	VSS[2]	Ground	Connect to VSS frame
85	VDD[1]	Power	
86	VDD_H	Power	
87	ha[4]	Input	
88	ha[3]	Input	
89	ha[2]	Input	
90	ha[1]	Input	
91	ha[0]	Input	
92	hd[7]	I/O	
93	hd[6]	I/O	
94	hd[5]	I/O	
95	hd[4]	I/O	

96	hd[3]	I/O
97	hd[2]	I/O
98	hd[1]	I/O
99	hd[0]	I/O
100	VSS_H	Ground

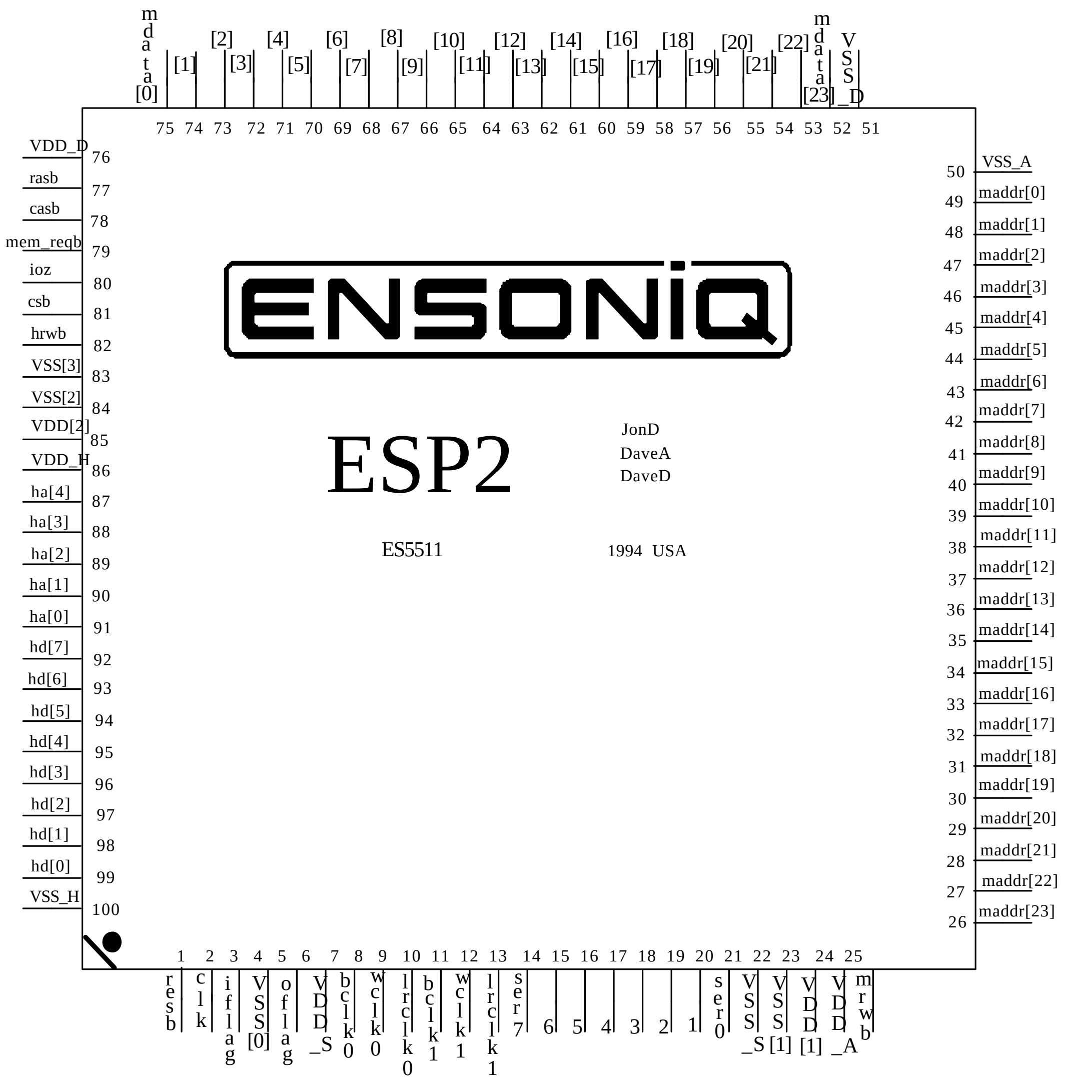


Figure 12.

Figure 13. ESP2 die photograph, revision 1.

ESP2

Ensoniq Signal Processor 2

Part II

Language and Software Specification[1]

Jon Dattorro
J. William Mauchly

[1] © 1995, Jon Dattorro

0. Introduction

In Part II we describe the syntax and assembler interpretation of the language used for writing programs for the ESP2 digital audio signal processing computer chip. The chip architecture and instruction set are described in Part I (often referred to as the *Chip Spec*.). Some instruction features are highlighted, elaborated, or augmented here in Part II, however.

1. Architectural Overview

The ESP2 chip utilizes a very long instruction word. It shares some of the properties of a Reduced Instruction Set Computer (RISC) where all hazards are accountable in the assembler. Every ESP2 instruction takes the same amount of time (four system clocks) and the same amount of space (96 bits).

In the ESP2 there are three distinct function units: called MAC, ALU, and AGEN.
A short (2-deep) Pipeline design strategy is incorporated for each function unit to speed up instruction execution. There is one Program Counter (PC). All three function units operate in parallel on a common pool of registers, and the instruction set fully supports the parallelism. Only one of the function units, the Address Generator (AGEN), can access data memory which is external to the chip.

1.1. Program Instruction Memory

The instruction word is 96 bits wide. All instructions are stored inside the chip in DRAM for ultimate speed of execution. In the first chip revision are planned 300 instructions, with provisions for 1024. Each ESP2 instruction is, because of parallelism, equivalent to 3 instructions on other popular DSP chips.

The design constraint which makes all instructions execute in the same amount of time was imposed because of its importance to the programmer. The programmer’s job often consists of cramming the most functionality into a period of time dictated by the audio sample rate. Instructions which would have had conditional execution times are excluded from the ESP2 design as are instructions whose execution time would have deviated from the norm. The enforced regularity eases the programmer’s job of budgeting execution time.

1.2. Internal Registers

The chip contains three types of registers: General Purpose Registers (GPR), Address Offset Registers (AOR), and Special Purpose Registers (SPR). The GPRs, AORs, and SPRs can be utilized as sources and destinations by the MAC unit and the ALU. All registers have a unique address in the range of [0...1023]; i.e., there are 1024 on-chip registers. Except for a few SPRs, all registers are 24 bits wide. SPRs which are not 24 bits in width have unused bits read as logical 0.

The AORs are usually associated with the AGEN and are utilized as sources there, but when not being used by the AGEN they are free to double as general purpose registers. So, the AORs can be used like GPRs in the MAC unit and ALU; the syntax supports this.

The SPRs have a wide variety of purposes. Many of them are tied directly into the operations of the three function units. As such, they often act as extra source or destination registers. For example, one of the 16 SPRs called DIL is always the destination of the AGEN for an external memory read. Another example is the ALU_SHIFT SPR which acts as an extra source register, holding the shift amount and direction for the ALU’s double precision shift instructions (ASDH, ASDL, LSDH, LSDL).

The SPRs are implemented using Static RAM, so we may associate the words: **s**pecial, **s**ource/destination, and **s**tatic with the **S** in SPR as a mnemonic aid to understanding.

1.3. Function Units

The function units operate directly on the registers. The three function units are the Multiplier/Accumulator/shifter (MAC unit), the Arithmetic Logic unit (ALU), and the Address Generator (AGEN). All three units fetch their individual instruction information (opcode) from different parts of the same long instruction word. The instruction word also provides the source and destination register addresses for the operations.

The MAC unit and ALU routinely utilize three operands; two sources, one destination.[2]

AGEN code can be automatically generated for the programmer by the assembler if desired. The AGEN routinely utilizes one source operand (an AOR) plus three or four extra source operands.[3] The AGEN utilizes one destination operand[4] and/or one or no extra destination operand.[5]

[2]Many SPRs are dedicated to particular function units bringing the maximum utilization up to five operands in at least one case.

[3]; i.e., the three SPRs known as the region control registers, and one of the 16 Data Output Latch SPRs in the case of an external memory write.

[4]; i.e., the region BASE*r* SPR in the case of an UPDATE BASE operation,

[5]; i.e., one of the 16 SPRs known as Data Input Latch in the case of an external memory read.

1.3.1. MAC unit
The MAC unit takes two registers as sources and writes another register as destination. It also has two internal latches, called MAC and MACP (P for Preload), which can be involved in the computation as *seed* sources and are also accessible via SPRs. The MAC latch can also act as an extra destination holding intermediate accumulated products, but this destination can be selectively inhibited under program control. Internal to the MAC unit is also a versatile Barrel Shifter which can be used to shift (both ways) an accumulated result out to some destination, or to shift the contents of the seed sources prior to the next accumulation. The MAC unit has 20 distinct **fundamental** (primitive) instructions and 10 variants, plus about 17 pseudo instructions.

1.3.2. ALU
The ALU usually takes two registers as sources and writes another register as destination. One of the sources, of course, may also be specified as the destination if desired; the same is true of the MAC unit. The ALU controls all branching and subroutine calls. The ALU has the widest assortment of fundamental instructions numbering 32, plus about 23 pseudo instructions.

1.3.3. AGEN
The AGEN routinely takes five registers as sources for an external memory write, or four source registers and one destination register for an external memory read. The AGEN unit is the most unusual function unit; it both calculates addresses and uses those addresses to read or write external memory. AGEN has several flexible modes of address calculation. The AGEN is a modulo address calculation unit by design. It employs a multiplicity of address offsets into one or several user-specified circular regions of *physical* (absolute) address space. This is its fundamental mode of calculation but it can easily be coerced into accessing fixed external table arrays or into absolute addressing of peripheral I/O devices. The AGEN has 8 distinct instructions.

1.4. External Memory

24 address and 24 data lines on-chip provide access for up to 16 Mega-words of 24-bit data memory off-chip. All external memory accesses are controlled by the AGEN. One physical read or write can occur per **instruction cycle (**one program line, one instruction line, one line of code, or four system clocks; all the same meaning**)**.

1.4.1. AGEN Access of External Memory

The address calculator within AGEN takes as source operand, one Address Offset Register (AOR) having an associated region specifier. The physical address automatically calculated is a function of the selected addressing mode, the specified AOR, and the contents of the **region control registers (**the extra source SPRs: BASE*r*, SIZEM1*r*, END*r***)** for the specified region. The AGEN then uses that address to read or write external memory.

A word of data read from external memory is transferred to one of a set of 16 SPRs called Data Input Latches (DIL). Likewise, a word of data written to external memory comes from one of a set of 16 SPRs called Data Output Latches (DOL). The AGEN's DILs and DOLs are then accessible by the MAC unit and ALU as operands.

1.4.2. Regions

A **region** is a programmer-specified absolute range of external memory. Having a total of eight hardware-supported regions, each region can be used for any desired purpose. For example, one region could hold a multiplicity of **delayline**s,[6] while another holds one or more tables, while yet another is used for peripheral I/O. Regions can reside anywhere in physical memory and be of any size, so long as they do not span the physical limits. Every external memory access is associated with one of the eight provided regions: I,P,Q,R,S,T,U,V. Regions are defined by their associated region control registers: BASE*r*, SIZEM1*r*, and END*r*.

1.4.3. Offset Addressing

An external memory address is automatically calculated using offset addressing into a region, followed by modulo arithmetic to keep the address within the region boundaries. An absolute address is automatically calculated by AGEN on every instruction cycle; the calculation is begun by adding an Address Offset Register (AOR) to a region BASE*r* register. An absolute address which is beyond the region END*r* is wrapped around the modulus by automatically subtracting the actual region size from it, so as to remain inside the region. The specific AOR and region used for each AGEN operation are encoded in the instruction and implicitly or explicitly determined by the programmer.

The address offset held within an AOR is regarded as 24-bit unsigned by the AGEN unit. But since AORs can also be used for general computations outside the realm of address generation, the 24-bit contents of AORs are treated as standard two's complement by the ALU and MAC unit.

[6]A delayline implements the DSP function, z^{-N}, and is often a constituent of a circular buffer. (See Preface.)

1.4.4. AGEN Utilization

Artificial reverberation requires perhaps 100 separate delaylines. In some other more traditional processor architecture, at every sample period the read and write address of every delayline would be decremented and individually subject to modulo arithmetic.

Using the ESP2, the individual delayline address offsets each reside in a separate Address Offset Register. A region is set up which contains all the delaylines in their entirety, and whose size is at least the total size of all the delaylines.[7] The assembler determines that total and initializes the region control registers:

SIZEM1*r* is set by the assembler to the total region size, less 1.

The END*r* SPR points to the last absolute memory location in the region.

BASE*r* initially points to the start of the region in absolute memory, as determined at assembly.

The BASE*r* register is decremented once per sample under program control. Each delayline address <u>offset</u>, held in an individual AOR, is with respect to the region BASE*r*. A single decrement of the region BASE*r*, then, will decrement the absolute address of <u>every</u> delayline. The modulo arithmetic required to keep BASE*r* within the modulus **(**the region, or the *circular buffer***)** happens automatically in the hardware.

In other words, all modulo arithmetic is performed with respect to the <u>region</u> as modulus, not to the individual delayline. All the delaylines reside within the chosen region. As the region BASE*r* is decremented, all the delaylines effectively move at once around the circular region following that region BASE. This is further discussed in the section, UPDATE region BASE.

[7]The assembler first augments the declared size of each individual delayline (or table) by 1.

2. Assembly Language

This section presents the skeletal parts of the ESP2 syntax, and introduces concepts of truncation mathematics which are indigenous to fixed-point machines. The exposition of the greater part of the AGEN unit syntax is postponed until the section devoted to the AGEN unit itself; likewise for the MAC unit and the ALU.

2.1. Assembler Directives

Keywords found in a source file are called *declarators.* They determine the interpretation of statements following, until another declarator is encountered. No commas are required to separate multiple declarations in a block of like declarations.

2.1.1. Declarator: PROGRAM *program_name*

program_name becomes a global symbol in the object and header files. *program_name* must appear on the same line as the declarator, and may not be a reserved keyword.

2.1.2. Declarator: PROGSIZE = *n*

or PROGSIZE <= *n*

The programmer issues one of these statements, where *n* is some constant expression denoting the number of desired lines of code. The assembler issues an error or a warning if the equality or inequality is not respectively true. This declaration is not mandatory.

2.1.3. Constant Declaration: DEFCONST

This declarator is typically found towards the beginning of all the declarations. Use the = sign, no comma required. Multiple DEFCONST are allowed.

```
DEFCONST
        Sintable_Size = 256     Region_Size = 256 * $100
        Costable_Size = 256     Table_Size = 512          tblSize1 = 513
```

2.1.4. Register Declaration: DEFSPR, DEFGPR

SPRs and GPRs are respectively declared and optionally initialized here. GPR register addresses may also be determined by the programmer if desired using @ . For example,

```
DEFGPR
      some_gpr = 1024 @$3f          ! $ means hexadecimal
```

The # symbol extracts the *value* of all symbols following in an expression.[8] The value of a number is the number itself, so the following declaration of some_gpr is equivalent.

```
DEFGPR
      some_gpr = #1024 @$3f         ! initialization and/or address optional
      some_other  @$40
```

No comma is required. Multiple DEFSPR, DEFGPR are allowed. <u>All</u> register names must be universally unique.

In general, in a DEFSPR, it is true that any SPR explicitly initialized should override any assembler determination. Should the programmer wish to <u>reserve</u> any of the AGEN DILs or DOLs for any purpose, the act of declaring them within DEFSPR has the desired

[8]See Value (extracted by #) Summary, or section on Fixed-Point Mathematics.

effect. The assembler then relinquishes those particular resources during scheduling. Programmer override of region BASE, SIZEM1, and END initialization is not observed by the assembler when determining AOR assignments.

2.1.5. AOR Declaration/External Memory Allocation: DEFREGION
This declaration is used for defining tables, delaylines, and absolute I/O regions of external memory. Declaration of a particular region is allowed only once, no commas required. AORs are declared and optionally initialized here. <u>All</u> names must be universally unique. AORs become implicitly associated with the region under which they were declared by default. Their register addresses may be optionally declared just as they are for GPRs. For example:

```
DEFREGION    Q[$10000]  @256
        mydelay[3032]
        yourdelay[999]  @3033
        myOffsetRegister = $a70000
        somePointer = 0 @$200       ! initialization and/or address optional

DEFREGION    R[Sintable_Size+1+Costable_Size+1]
        sin[Sintable_Size]
        cos[Costable_Size]
        table_pointer = &cos[0]        !AOR initialized to root of cosine table
```

Declaration of the region size in DEFREGION *r*[*region size*] is optional[9] as is the physical location (@256) of the region start, but region size minimum is 1. Region specifier, *r*, is required: I,P,Q,R,S,T,U, or V (= 0,1,2,3,4,5,6, or 7).

Declaration of the relative address offset of the beginning (the **root**) of an external memory array (@3033), with respect to the physical start of the region, is also optional. In the declarations above it should be clear that &yourdelay[0] = 3033 **(**& as in the C programming language**)** is the root address offset of yourdelay[999], but its absolute (physical) address is 256 + 3033.

Table and delayline sizes are not optional. If the region size, or table or delayline size **(**specified within [ ]**)** is written as a floating-point expression, it will be rounded to the nearest integer by default. The programmer may explicitly indicate various math functions (see Fixed-Point Mathematics) other than rounding.

[9]The assembler would set the SPR, SIZEM1*r* = *region size* **-** 1. If region size is not specified, the assembler will automatically determine it. BASE*r* and END*r* would also be determined automatically, whether or not region size is specified. BASE*r* is usually initialized to the region start. All these region control register settings can be overridden in a DEFSPR.

External Memory Initialization
An extension of the DEFREGION syntax[10] facilitates initialization of external memory by a *relocating downloader*.[11] The assembler issues directives to a downloader by encoding the assembly **.**o and **.**bin object output files. For example:

```
    DEFREGION  P[Region_Size]=0
         sigmoid[Table_Size]  :  "filename"
         tblName1[tblSize1]  :  3
```

This DEFREGION declaration shows an assignment (=0) that causes the assembler to issue a directive to a downloader to initialize the entirety of region P in external memory to 0. Any constant expression could have been used in place of 0 for the initialization value.

The declaration of the external memory array, sigmoid[Table_Size], has an assignment syntax (**:**) which causes the assembler to encode the ASCII name ID *filename* into its **.**o and **.**bin output files. This becomes a directive to a downloader to use the contents of the quoted file as the initialization of the external memory array. The array size can be as small as 1 or as large as physical memory.

Likewise, the declaration of the external memory array, tblName1[tblSize1], has the assignment syntax (**:**) which causes the assembler to encode an array identifier (ID=3) into its **.**o and **.**bin output files. This becomes a directive to a downloader to use some array (array number 3 in this case) from an established library as the initialization contents of the external memory array. Any constant expression could have been used in place of 3 for the array identifier.

2.1.6. Declarator: CODE
This keyword must precede all instruction lines. Register, constant, and region declarations are allowed within the body of the code. In that case, the keyword CODE must appear again immediately following these interspersed declarations.

2.1.7. Attributes: LOCAL, GLOBAL
These two keywords may occur anywhere within the body of a DEFCONST, DEFSPR, DEFGPR, or DEFREGION declaration, and with any frequency. GLOBAL will cause all subsequent symbol declarations in the respective body to appear in the **.**hdr file assembler output having the format of #define statements as in the C programming language; LOCAL turns that off. Every new declaration defaults to LOCAL.

[10]As always under DEFREGION, @ is optionally available syntax following any assignment (**:** , =).
[11]See the External Memory Host Access Application.

2.1.8. Declarators: FAMILY, MEMBER
Each declarator encodes a following constant expression into a reserved field of the **.**o and **.**bin assembler output files; the default encoded value is 255 if the declarator is not employed. They are useful for making efficient parameter-control structures for the system host and to organize multifarious ESP2 programs constituting a single commercial product.

2.2. Instruction Syntax

An instruction line must contain a MAC unit operation.

MAC-operation

An ALU operation may follow a MAC unit operation on an instruction line. If none is specified by the programmer, then the assembler will insert an ALU NOP.

MAC-operation ALU-operation

An AGEN operation may follow an ALU operation on an instruction line. If none is specified by the programmer, then the assembler may schedule an AGEN operation there. If no AGEN operations are required, it will insert an AGEN NOP.

MAC-operation ALU-operation AGEN-operation

One line of code is typically referred to as an *instruction cycle*.

Curiously, both the ALU and MAC unit NOPs are pseudo instructions, while the AGEN has a dedicated NOP instruction. **(Pseudo instructions** are formulated within the assembler as source/destination variations of the fundamental instructions.**)**

2.2.1. The Predominant Destination to Source Latency

ALU results have an inter-unit latency of one when sourced from the MAC unit. This means that the ALU results become available to the MAC unit on the <u>second</u> queued line of code following the ALU instruction. But MAC unit results are available to the ALU on the first queued line of code following the MAC unit instruction; i.e., on the next queued program line. **(**See the section called **Pipeline** for all cases by example.**)**

Whenever SPRs act as extra source/destination operands, they are accessed having the same latency as the established source/destinations unless indicated otherwise.

2.2.2. Line Continuation

A single tripartite instruction line **(**one line of code, one instruction cycle**)** may be made to occupy more than one printed line by placing a continuation symbol **(**the back-slash, \ **)** as the <u>first</u> character of text on a line.

```
parry: MACP  +  coef3  X  delayline[333]  >  MAC  >>3  >  temp
\                                                ADD  this, that  >  there
\                                                         RD  *chorusline  >  DIL0
```

2.2.3. Labels

An instruction line may contain a label. A label must be the first thing on a line and must be followed by a colon.

```
mysubroutine: coef1 X temp > MAC        MOV  temp  >  lasttemp
              NOP                       MOV  #mysubroutine  >  INDIRB
```

The value of the label is the number of the program line upon which it resides; the **line number** corresponds to the Program Counter value when it hits that line of code. **(**The # symbol tells the assembler to allocate a GPR whose contents is the value of the Program Counter denoted by *mysubroutine*.**)**

2.2.4. Comments Within a Program

There are three styles of comments available within ESP2 programs and declarations:

1) A comment may start with an ! and end with another ! or end-of-line/carriage-return; i.e., one ! will comment out everything to the end of the line. A second ! will delimit the comment as in the C-style /* */ .

2) The C-style comments /* */ are also available. In this case the right side delimiter is required, unlike ! style comments.

3) The ANSI-C notation, // , comments from that point to the end-of-line/carriage-return.

2.3. Fixed-Point Mathematics

At assembly, the assembler evaluates expressions much like they are evaluated in the C programming language. Specifically, there is the same dichotomy between integer and floating-point math; viz: i%k vs. MOD(,) . Integer expressions are promoted to floating-point expressions also as in C. Scientific notation is supported, and a special binary-radix (**q**) notation is introduced to facilitate the assignment of floating-point values to fixed-point 24-bit registers.

In declarations involving mathematical expressions, the programmer must bear in mind that the math is being performed (assuming two's complement) at the full precision of the machine within which the assembler is executing. But when ESP2 register contents are used within expressions, they are assumed two's complement at 24 bits. This means that register contents will be sign-extended to machine long-word precision before being used in subsequent expressions. For example:

```
DEFCONST
    Big_Negative      =    $800000              ! (positive in 32 bits)
    Negative          =  $ff800000              ! (negative in 32 bits)
    Calculus1         =   0.5 * Big_Negative    ! =    $400000    (positive in 32 bits)
    Calculus2         =   0.5 * Negative        ! = $ffc00000     (negative in 32 bits)
DEFGPR
     some_gpr    = Big_Negative  @5         ! register gets $800000 (negative 24 bits)
     some_other  =  0.5 * Big_Negative      ! register gets $400000  (positive 24 bits)
     gpr2        =  0.5 * some_gpr          ! but this register gets $c00000 (negative 24 bits)
     gpr3        = #0.5 * some_gpr          ! register gets 3 (2.5 rounded)
     gpr4        = COS(some_gpr)            ! legal here in declarations. Contents used.
     gpr
```

The declaration of the hexadecimal ($) constant Big_Negative is assumed positive in the calculation of Calculus1 because the assembling machine long-word size is 32 bits. But notice that once the register some_gpr gets loaded, subsequent math employing the contents of that GPR **(**as in the calculation of gpr2**)** sign-extends its contents before performing the calculation.

We can also perform math using the ***value*** of symbols. The # symbol is used to indicate the intent to use value rather than content.[12] For example, the value of some_gpr is its register address, 5 .

# **precedes all the elements of an expression in ESP2 syntax**;

i.e., we do not allow any appearances of # within an expression. This means that any expression preceded by # is assumed to use only symbol *values* in its evaluation.

[12]See the **Value (extracted by #) Summary** under the ***delayspec*** **Chart** in the AGEN chapter.

Continuing from the declarations above:

```
CODE
NOP             MOV  #0.5 * some_gpr  >  gpr4      ! gpr4 gets 3  .

      error
NOP             MOV    0.5 * some_gpr  >  gpr5      ! error here but not in declarations.
```

This last example shows that register contents are <u>not</u> allowed as operands in expressions appearing in the code as they are allowed in the declarations. There are two reasons for this:
1) In-line expression operators make the code unreadable because the functions of the ESP2 chip itself become confused with the operators in the expression. **(**In the expression above, one might conclude that there were a multiplier in the ALU.**)**
2) From the expressions, it is not clear whether the operations are being performed on the register values (their addresses) or their contents. If contents were assumed, only initialization contents are known to the assembler. Yet the code seems to call for current contents, hence making it unreadable.

It is imperative to distinguish between in-line math expressions appearing in the code, and ESP2 chip operations. The in-line math is performed <u>at assembly time</u>, and is provided as a convenience to the programmer who may prefer in-line expressions to constants declared in a DEFCONST.

Continuing...

```
NOP              ADD  #(1+gpr), B             ! legal (gpr address used).

     error
NOP             ADD COS(#gpr), B              ! illegal because # must be at beginning.
NOP             ADD #COS(gpr), B              ! legal (gpr addr. used). Cosine function.

      error
NOP              ADD  COS(gpr), B              ! illegal because implies contents used.

      error
NOP              ADD 1+gpr, B                  ! illegal because implies contents used.
```

This last example is ambiguous because it is not clear whether it is the contents of that GPR whose register address is one past gpr which is desired, or 1 added to the contents of gpr which is desired. In a running program, the contents of gpr are likely to change. If we had established the convention that contents were to be used, then the assembler would only have knowledge of initialization contents. The ESP2 does not have the means, in general, to change the contents of the A-operand prior to the operation with the B-operand.[13]

[13]For treatment of gpr as the base of some internal register array, the syntax provides a register indexing construct.

2.3.1. Assembly-Time In-Line Math Functions

Excerpts from the C programs which constitute the math portions of the assembler program itself follow. These enumerate the available functions while showing how they are internally defined.

```
/*-----------------------------------
 Math Functions (from espdata.c)
 -------------------------------------*/
 GRefstruct fun_data[ ]= /* math-function mnemonics */
{                          /*    (see also espdata.h)       */
  {"SIN",        AP_SINX},
  {"COS",        AP_COSX},
  {"TAN",        AP_TANX},
  {"ASIN",       AP_ASINX},
  {"ACOS",       AP_ACOSX},
  {"ATAN",       AP_ATANX},
  {"SINH",       AP_SINHX},
  {"COSH",       AP_COSHX},
  {"TANH",       AP_TANHX},
  {"EXP",        AP_EXPX},
  {"LN",         AP_LNX},
  {"LOG",        AP_LOGX},
  {"SQRT",       AP_SQRTX},
  {"CEIL",       AP_CEILX},
  {"FLOOR",      AP_FLOORX},
  {"ABS",        AP_ABSX},
  {"INT",        AP_INTX},             /* integer result */
  {"BTRUNC", AP_BTRUNCX},              /* integer result */
  {"ROUND",  AP_ROUNDX},               /* integer result */
  {"ATAN2",  AP_ATAN2YX},              /* 2 arguments */
  {"MOD",        AP_MODXY},            /* 2 arguments */
  {"RSH",        AP_RSHXY},            /* 2 integer arguments; integer result */
  {"LSH",        AP_LSHXY}             /* 2 integer arguments; integer result */
};
```

Exponentiation is supported via the old ****** Fortran notation. The C-language boolean operators, such as **&** , **|**, **~** , and **^** , **(**AND, OR, one's complement (bit flip) operator, and XOR**)** are also recognized in expressions at assembly. & also has meaning in the AGEN unit syntax and is widely used there, so context is critical for its successful use.

2.3.2. Truncation Mathematics.

Assembly-Time Math Function Definitions:
Magnitude Truncation, Rounding, and Truncation

```
/* double result, val1, val2; *//* (from espeval.c) */
/*-----------------------------------------------------------------
   Integer functions:  These functions return an integer-type
   value.  RSH and LSH require integer-type arguments.
   -----------------------------------------------------------------*/
    case AP_INTX:    /* INT    ( CExpr )         */
    case AP_ROUNDX:  /* ROUND  ( CExpr )         */
    case AP_BTRUNCX: /* BTRUNC ( CExpr )         */
    case AP_RSHXY:   /* RSH    ( CExpr , CExpr ) */
    case AP_LSHXY:   /* LSH    ( CExpr , CExpr ) */
    {
     switch (root->CHILD->usem.ival)
     {
      case AP_INTX:       /* magnitude truncation */
       result = floor(fabs(val1)) * SIGNUM(val1);
       break;
      case AP_ROUNDX:     /* rounding */
       result = floor(fabs(val1) + 0.5) * SIGNUM(val1);
       break;
      case AP_BTRUNCX:    /* truncation */
       result = floor(val1);
       break;

      case AP_RSHXY:
       if (!int_op) error (ERR_OPNDINT);
       result = (long) val1 >> (long) val2;
       break;
      case AP_LSHXY:
       if (!int_op) error (ERR_OPNDINT);
       result = (long) val1 << (long) val2;
       break;
     }
     rtn = new_sem (FIXED);
     rtn->usem.ival = (long) result;
     break;
    }
```

Shown are the C-program definitions of the three truncation functions and left/right arithmetic shift functions recognized at ESP2 program assembly. The **truncation functions (**magnitude truncation, rounding, truncation**)** operate on floating-point numbers and produce integer results. Magnitude truncation and rounding are symmetrical functions. On many machines, it is helpful to keep in mind that the **(int)** or **(long)** C-cast of a floating-point number actually performs a magnitude truncation. The SIGNUM() macro returns (1., 0., -1.) for strictly positive, zero, and negative arguments, respectively. Recall that the floor(x) function, defining truncation, returns the largest integer not greater than x ; truncation simply throws away or masks off the fractional part.

Real-Time Compute Statistics

Although the defined truncation functions are only used at assembly, it would be interesting to analyze the statistical impact of each function type when used in real-time computation; e.g., say in an ESP2 program executing some digital filter algorithm. For that application, the meaning of the truncation functions regards the handling of what is considered to be the fractional part **(**the LSBs to the right of the radix point**)** of some binary word. An example might be the conversion from a double precision (48-bit) result to single precision (24-bits). <u>The most notable outcome is that the use of magnitude truncation introduces noise into the signal which is statistically 6 dB higher in power than that produced by either rounding or truncation</u>. Further, rounding and truncation produce the same noise power, but truncation, having a DC offset of one-half quantum, is not a zero mean process. These results could be explained by considering both the magnitude and sign of the quantization errors of each truncation function. [Jackson,ch.11.2]

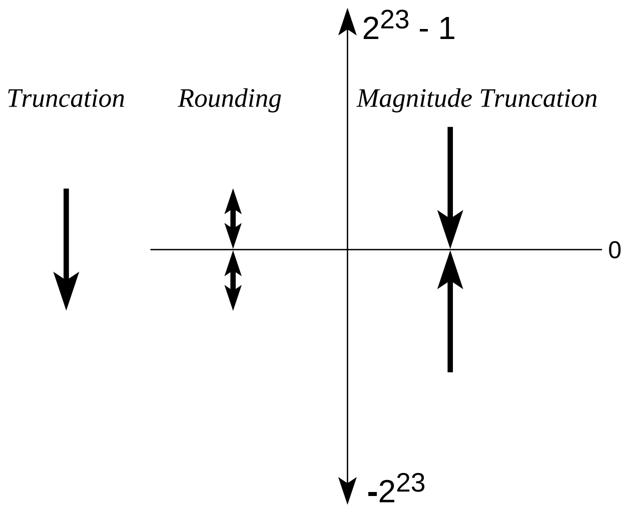


Figure MT. Direction and relative magnitude of change after quantization of any 24-bit two's complement number.

Alternately, in Figure MT the action of each truncation function is represented in terms of the change to some two's complement number. None of the truncation functions can change a positive number to a negative number, or vice versa. The truncation function simply called 'truncation' has no point of symmetry; it causes the same direction of change anywhere in the continuum. This lack of symmetry accounts for the statistical DC offset. Of the three truncation functions, both rounding and truncation have the propensity for returning values which exceed the magnitude of their arguments. This idiosyncrasy has been shown to be one of the causes of limit cycle tones produced in direct form and lattice digital filter topologies. [Jackson,ch.11.5] Magnitude truncation does not share this characteristic and can sometimes remedy limit cycle oscillation. [Smith] **(**See the Reverberation Application.**)**

In Figure MT it is indicated that truncation will increase magnitude of only negative numbers, whereas rounding can increase the magnitude of all numbers. Further, truncation <u>always</u> increases negative number magnitude, but rounding does not always increase magnitude (explaining the bidirectional arrows). On the other hand, magnitude truncation <u>always</u> **de**creases magnitude of both positive and negative numbers.

Real-Time Compute Implementation
We wish to know how the truncation functions are each implemented in the binary domain. For the sake of illustration, let us assume that we are given 24-bit two's complement binary numbers in **q**8 format (discussed shortly):

```
DEFGPR
        val1 = $007FFF                  !  q8  means  $007F.FF  =   127.99609375
        val2 = $FF8001                  !  q8  means  $FF80.01  =  -127.99609375
```

binary magnitude truncation
In the binary domain, the corresponding operation to magnitude truncation is:

```
Pseudo-code:  if(val < 0)  val += $0.FFF... ;        /* binary magnitude truncation */
                    val = binary truncate(val);        /* discard bits to right of binary point */
```

For the two values, after binary magnitude truncation: val1 = $007F00, val2 = $FF8100 . This is how magnitude truncation might be programmed in a real-time computation within the ESP2. In hexadecimal we are adding 0.999... to *val* when it is negative. We conditionally add $0.FFF... to *val*, rather than $1.0, to avoid bumping up a perfect negative integer. This implies that if <u>any</u> of the fractional bits of *val* are nonzero when it is negative, then magnitude truncation will increase *val.*

But as it often happens in ESP2, the given 24-bit binary number may itself be the **MSB**s (most significant bits) of a higher precision 48-bit result. It is likely that the 24 **LSB**s (least significant bits) of the higher precision result were nonzero and the situation is such that we are not sure. Statistically, it is much better to err on the side of caution if we do not know what those higher precision LSBs were. So in this circumstance,[14] we would amend the Pseudo-code to conditionally add $1.0 to *val* instead of $0.FFF... **.**

binary rounding
Curiously, in the case of two's complement binary rounding there is no <u>conditional</u> addition. In the binary domain, the corresponding operation to rounding is:

```
Pseudo-code:  val += $0.8;                             /* binary rounding */
                    val = binary truncate(val);        /* discard bits to right of binary point */
```

For the two given values, after binary rounding: val1 = $008000, val2 = $FF8000 . This is how rounding might be programmed in a real-time computation within the ESP2. In hexadecimal we are unconditionally adding 0.5 to *val* regardless of its sign.[15]

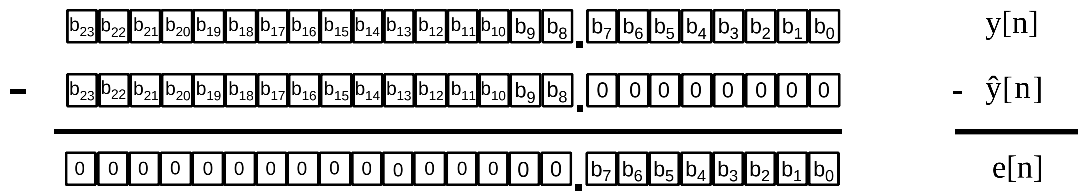
[14]This is, in fact, how the magnitude truncation mode is actually implemented in the ESP2 external memory data interface (see the Chip Spec.). As such it is, theoretically, more applicable to magnitude truncation of double precision results.
[15]When comparing rounding results in the floating point domain with the corresponding results in the binary domain, it is wise <u>not</u> to use test values having a fractional part = 0.5 exactly, because this case produces different results in each domain. To solve this quandary we invoke Reagan's theorem; it 'doesn't matter'. (Appendix II)

binary truncation
In the binary domain, the corresponding operation to truncation is simply:

```
Pseudo-code:  val = binary truncate(val);        /* binary truncation */
                                                  /* discard bits to right of binary point */
```

For the two given values, after binary truncation: val1 = \$007F00, val2 = \$FF8000; the 8 LSBs are masked off, or simply discarded.

One useful fact regarding truncation is that the fractional part **(**rather, the part that is discarded or masked off**)** is always positive in sign. This fact could be used to advantage within a digital filtering circuit having truncation error feedback for the purpose of minimizing **truncation noise**.[16] [Dattorro]



b_{23} b_{22} b_{21} b_{20} b_{19} b_{18} b_{17} b_{16} b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 . b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
y[n]
- b_{23} b_{22} b_{21} b_{20} b_{19} b_{18} b_{17} b_{16} b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 . 0 0 0 0 0 0 0 0
- ŷ[n]
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 . b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
e[n]

$$-2^{15}\, b_{23} + \sum_{m=1}^{23} 2^{-m+15}\, b_{23-m} \;-\; \left(-2^{15}\, b_{23} + \sum_{m=1}^{15} 2^{-m+15}\, b_{23-m}\right) = \sum_{m=16}^{23} 2^{-m+15}\, b_{23-m}$$

Figure TruncPos. Example showing how truncation error is always positive.

Figure TruncPos shows the example of truncating any 24-bit **q**8 two's complement number by taking the 16 MSBs and discarding the 8 LSBs. We are interested in the sign of the error e[n] that results from subtracting the truncated number $\hat{y}[n]$ from the full-precision number y[n]; i.e.,

$$y[n] - \hat{y}[n] = e[n] \geq 0$$

Since the b_i can only take on the value 0 or 1, in two's complement, the stated result follows regardless of the sign of y[n]. By induction, this result extends to other **q** and other truncation widths.

[16]This noise, due to ongoing internal signal quantization error, is also known as roundoff noise. [Jackson] Truncation error feedback is also known to minimize limit cycle oscillation [Laakso], thus providing an alternative to magnitude truncation as a remedy.

2.3.3. Scientific and Binary-Radix Notation

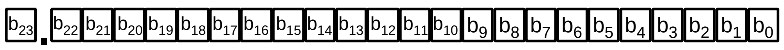
These notations are both of the form:

*mantissa∗radix**exponent*

Scientific notation has radix 10, while binary-radix notation has radix 2.

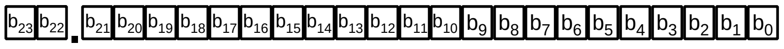
The assembler supports scientific notation of floating-point constants as in, for example, 7e-3 which is mathematically equivalent to 7∗10**-3 . Substituting an *expression* for the mantissa is not allowed in our language; i.e., using the same example, *expression* e-3, is <u>not</u> allowed. **(**An *expression* would consist of arithmetic operations on symbolic names and/or numbers.**)** Blank space is neither permitted before the ’e’.

Binary-radix **q** notation is similar; for example, 7q3 is defined as mathematically equivalent to 7∗2**3 (**q** for *quanta*). This notation comes in handy when we wish to specify the location of the binary point in a fixed-point number which is to be assigned as the contents of some register; i.e., to specify the register format. Figure Twos shows the two most commonly used formats for digital filtering coefficients. In Figure Twos (a), the range of coefficient is [-1., 1.), whereas[17] for (b) it is [-2., 2.).



b_{23} . b_{22} b_{21} b_{20} b_{19} b_{18} b_{17} b_{16} b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0

fixed-point **q**23 register content = $-2^0\, b_{M-1} + \sum_{m=1}^{M-1} 2^{-m}\, b_{M-1-m}$



b_{23} b_{22} . b_{21} b_{20} b_{19} b_{18} b_{17} b_{16} b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0

fixed-point **q**22 register content = $-2^1\, b_{M-1} + \sum_{m=1}^{M-1} 2^{-m+1}\, b_{M-1-m}$

Figure Twos. Two’s complement fixed-point examples: M=24 bits.

Unlike scientific notation, binary-radix notation can substitute a <u>complete</u> mathematical expression for the mantissa, and blank space is permitted before the **q**. **(**The complete expression can even include floating-point numbers in scientific notation.**)**

[17]This notation means that the positive extreme cannot be exactly reached.

When the assembler encounters a floating-point expression by itself (no **q** notation) whose magnitude is less than 1.0, it assumes that the expression will be represented within some assigned 24-bit register as **q**23 . This means that the floating-point value will be multiplied by 2**23, by default, before the register is initialized.[18] That result is rounded (also by default) to the nearest integer and then assigned as the contents of the register. This places the binary point one bit away from the MSB in the 24-bit register, rather, 23 bits away from the LSB.

If a floating-point expression is found by itself whose magnitude is greater than or equal to 1.0, then the assembler assumes **q**0 (=1) as default in that case. This places the binary point, in the 24-bit register, 24 bits away from the MSB (or 0 bits away from the LSB).

One way to look at binary-radix (**q**) notation, then, is as conversion from a floating-point representation, *floating.expr*, to fixed-point representation. In the conversion process, we can change the default scalar via the **q** notation; e.g.,

some_register = *floating.expr* **q**22

This declares the binary point in the 24-bit register to be fixed at two bits away from the MSB (or 22 bits away from the LSB).

Generally speaking, any complete expression followed with binary-radix **q** notation will be multiplied by 2 raised to the indicated power. But we should instead think of **q** notation as a mnemonic tool which provides the **location** of the binary point within some fixed-point register, as in Figure Twos.

[18]**q**23 is the maximum binary radix locator in 24-bit two's complement fixed-point.

2.3.4. Fixed-Point Arithmetic within the ESP2

Now we turn to the subject of numerical computation within the executing ESP2 function units. The rules of fixed-point arithmetic are easy when the **q** notation is employed.

Addition and Subtraction

Rule 1**)** *For ESP2 to add or subtract two fixed-point numbers, they must have the same binary point; i.e., the same* **q**.

If they do not have the same **q** then the various shifters in the ESP2 can be used to bring them into alignment.

Multiplication

Rule 2**)** *Using ESP2, when multiplying a 24-bit (two's complement) number having* **q**N *with another 24-bit number having* **q**M, *the product is a 48-bit number*[19] *having* **q**(N+M+1). *The 24 MSBs of the product is* **q**(N+M+1 - 24).

The extra 1 in Rule 2 comes from the ESP2 signed multiplier having a built-in permanent shift-left-1 of the product to remove the extra sign bit. Most often, only the MSBs of an accumulation of products constitute some desired result. So, for example, suppose we multiply a **q**0 signal at 24-bits **(**typically 16 bits of signal left-justified into a 24-bit word**)** by a **q**23 coefficient at 24 bits. The ESP2 result is a 48-bit product at **q**24. Now, if we truncate the least significant 24 bits storing only the MSBs, we end up with a 24-bit result at **q**0. In this case, the **q** of the result is the same as that of the signal.

2.3.5. Numerical Precision

If the programmer is scrupulous, it is not too difficult to implement *block floating-point* arithmetic. Block floating-point arithmetic is a numerical implementation of an algorithm using a fixed-point processor where the binary-radix point of a block (some collection or group) of operands is fixed, but only over some intermediate portion of a longer computation. A different block (or the same block computed at a different phase of the program) may then have a different binary point location. The block floating-point technique finds use as applied to FFT (Fast Fourier Transform) [Analog Devices] [Kim/Sung] or amplitude compression algorithms implemented on fixed-point processors having barrel shifters. When the fundamental word-size of a fixed-point processor is large enough (24 bits as in ESP2), the need for block floating-point diminishes and fixed-point computations may suffice.[20] An alternative to block floating-point is double precision arithmetic which ESP2 supports.[21]

[19]having 4 bits of sign extension,

[20]See the FFT Radix-4 Application.

[21]See the FFT Radix-2 Application.

2.4. GPR/AOR Array Declaration

The syntax provides a construct useful for manipulating internal register data organized as a register array or vector:

```
DEFGPR
      some_gprs  =  1  @$2...$80     /* initializes entire array to same value */
```

Alternatively,

```
DEFGPR
       some_gprs        = 1  @$2      /* (some_gprs, 0) not allowed here but OK in code. */
      (some_gprs, 1)    = 7
         :                :
         :                :           /* use this method to initialize to different values */
      (some_gprs, $7e) = 40
```

The same rules apply to AORs.

```
DEFREGION   R
       some_aors  =  $100  @$200...$233
```

As shown, an array of GPRs is declared (allocated) and (optionally) initialized. The same is done for a group of AORs. Note the allowable addresses for the respective register arrays in the Chip Spec. (see the section on Chip Architecture). The programmer should be cognizant of the current register allocation through examination the listing (**.**lst) file.

2.4.1. Internal Register Array Reference

```
CODE
MOV  (some_gprs, n) > destin
MOV  (some_aors, 1) > destin2
MOV  (some_aors, 0) > destin3
```

Reference to a GPR and two AORs within their respective array are shown. The GPR whose contents are moved to destin has an address which is that of some_gprs plus *n*, for *n* a constant expression. Similarly, destin2 gets the contents of the AOR at address $201.

This **(***register array name*, *index***)** reference to an AOR may be substituted anyplace where an AOR reference is legal. Double parentheses may be required to reference an AOR having an explicit region specifier; for example:

((some_aors, -7)**)***R*

Note that the assembler recognizes negative indices for this internal register array construct.

2.4.2. Array/Register Clear

It is up to the programmer to zero all critical registers, specific to an application program, in the declarations.

2.5. #include *filename*

The ESP2 assembler has a feature which allows #include statements as in the C programming language. The ESP2 assembler just expects a filename, however; no quotes or triangle brackets are required. The included file is typically used for the declarations of constants, but can be used for any declarations.
A possible usage would be to alias SPR names as follows:

```
DEFCONST
     BASEi  = #BASEI      //  # symbol extracts value which is SPR address this case.
     BASEu  = #BASEU
     YI     = #DILD
     YI2    = #DILC
```

Nested #include is allowed.

2.6. Conditional Execution

Curly braces {} may be placed around any individual instruction in the MAC unit, and/or the ALU, and/or the AGEN unit field to independently specify conditional execution. Further, **all** individual unit instructions can be conditionally executed. The curly braces enclosing a particular instruction field indicates that the corresponding skip bit is set for that function unit. When the skip bit is set, if the **Condition Mask Register (**the SPR called CMR**)** satisfies the **Condition Code Register (**the SPR called CCR**)**[22] at the time the conditional instruction is about to be executed, that instruction <u>will</u> be **executed**; otherwise that instruction will be skipped. A skipped instruction still consumes program time equal to one instruction cycle, however.

The CCR is set automatically only by the ALU on every instruction cycle, and the outcome is discernible to all three function units for the purpose of conditional execution on the next queued line of code. Since the CCR is also mapped as an SPR, it can be manually written to restore its state if necessary. The safest way to do so is by use of the ALU MOV instruction **(**or MOVcc, or RScc**)**. Manually loading the CCR any other way offers some interesting, but not necessarily useful results. **(**See the Chip Spec. for more details.**)**

[22]This concept is found in conventional microprocessor design for decision making based upon arithmetic and chip status. In ESP2, individual bits of the CCR correspond to various states of the chip. Logical combinations of these bits often yields more precise information. The CMR is used to mask the desired combinations.

The proper interpretation of a curly-braced field is that the corresponding unit's instruction becomes **conditionally executed** depending on the status of its skip bit and the CMR's relation to the CCR; i.e., if the associated skip bit is asserted and the Condition specified by the CMR is satisfied. The list of CMR masks can be found in the Chip Spec. and includes the Conditions: NEV (*never*), ALW (*always*), IOZ **(**masks the read-only *IOZ status bit* in the CCR which is a function of the IOZ input pin of the synchronization interface**)**, IFLG **(**masks the read-only *IFLG bit* in the CCR which is an image of the IFLAG input pin hardware semaphore of the synchronization interface**)**,[23] GT, LT, GTE, etc...
For example:

```
NOP                              MOV  #NEG  > CMR       !see  IF  pseudo using MOVcc
NOP                              SUB   non_negative_constant, ZERO  >  somewhere
{this X that > MAC, there}       ADD  mine, yours > ours
```

In this example, only the MAC unit instruction field is conditionally executed based upon the CCR outcome determined in the ALU operation on the previous line of code. The Condition being checked for is negativity, as shown in the first line of code having the load of the CMR from a GPR. That GPR is holding the value of the negativity Condition denoted by the keyword NEG. We discuss a more efficient load of the CMR, shortly.

2.6.1. cc-class Instructions

Some of the ALU's 32 fundamental instructions have a feature which allow the CMR to be unconditionally preloaded and then immediately used to determine whether the instruction on the very same program line will be conditionally executed. Notably, the Jcc, JScc, RScc, and MOVcc instructions[24] have this feature. For example:

```
NOP                SUB   non_negative_constant, ZERO  >  somewhere
NOP               {MOV  gpr1  >  gpr7,  NEG > CMR}           ! use  MOVcc  instr.
```

In this example, gpr1 is conditionally moved to gpr7 if the outcome of the previous ALU operation was negative. This feature of the MOVcc instruction saves us one line of code and one GPR, avoiding an explicit MOV to CMR as in the previous example.

The **mask** we have designated to represent the negativity Condition is itself stored in the A operand (address) **field** of the 96-bit microinstruction; It is not stored in the A operand. In contrast, the B and C operands are gpr1 and gpr7 (i.e., two GPRs). Thus, the assembler must determine what the NEG mask is and then assemble that into the MOVcc instruction's A operand field. The A operand field always gets moved into the CMR by the instruction, prior to the decision to conditionally execute {}. Also, the skip bit corresponding to the ALU instruction field must be asserted by the assembler for that ALU MOV instruction, as requested.

[23]An image of the OFLAG output pin of the synchronization interface is not incorporated into the CCR.
[24]There exist two ALU MOV class instructions: MOV and MOVcc (conditional MOV). See the section called MOV Pseudo Instruction.

This leads to another use of the MOVcc, RScc, JScc, and Jcc instruction types:

```
NOP                    MOV  gpr1  >  gpr7,  NEG > CMR        ! use  MOVcc  instr.
```

In this example, the lack of curly braces indicates that the ALU's skip bit is <u>not</u> set. The MOV of gpr1 to gpr7 will, therefore, always take place regardless of the CCR outcome on the previously executed program line in the ALU. The CMR still gets preloaded with the mask we have designated to represent the negativity Condition. This is a great convenience for subsequent conditional queued lines of code as it may save one instruction.

Here is an esoteric example:

```
NOP                     MOV  gpr1  >  gpr7                   ! use  MOV  instr.
NOP                    {MOV  gpr1  >  gpr7,  ALW > CMR}      ! use  MOVcc  instr.
```

Each of these two lines of code always loads gpr7 with the contents of gpr1. Regarding the second program line, since the CMR is unconditionally preloaded with a Condition Mask designated to represent the *always* Condition, the MOV (of gpr1 to gpr7) will unconditionally take place, even though the instruction is made conditionally executable by the curly braces.[25] As there is no ALU operation that could put the CCR in a state that represents the ALW Condition, it is the mask which is designed to satisfy <u>any</u> state of the CCR that determines the ALW Condition.[26]

2.6.2. Conditional Execution Latencies of the CMR

The ALU's IF pseudo instruction is defined by the assembler utilizing the MOVcc instruction; viz,

```
                    IF  Condition
```

is the same as writing

```
            MOV  ZERO > ZERO,  Condition > CMR
```

The IF pseudo accomplishes an unconditional preload of the CMR while using read-only SPR ZERO as the standard destination. The programmer is free to explicitly MOV to CMR, but the use of MOVcc (hence IF) suffers less latency as will be shown by example. For this reason there is no IF pseudo in the MAC unit. **(**Also recall that the CCR is automatically set only by the ALU.**)**

In the following <u>examples</u>, *Condition* connotes some arbitrary Condition that might be desired by the programmer, while we choose POS (specifically, the positivity Condition) as another Condition to play against. We also choose the MOVcc instruction from the cc-class instructions, to preload the CMR in some examples. Substituting another cc-class instruction should yield the same conditional execution latency.

[25]The assembler should respect the programmer's wish to set the skip bit for this instruction.

[26]Likewise, there exists a NEV (*never*) Condition.

```
MAC unit                  ALU                              comment
NOP                       IF  POS                          ! pseudo uses  MOVcc  instr.
NOP                       ADD A, B                         ! CCR set by ALU
{result X gpr3 > gpr4}    {some_instruction}               ! MAC and ALU conditional


NOP                       ADD A, B                         ! CCR set by ALU
NOP                       IF  POS                          ! arithmetic CCR remains valid
{result X gpr3 > gpr4}    {some_instruction}               ! MAC and ALU conditional


NOP                       ADD A, B                         ! CCR set by ALU
{result X gpr3 > gpr4}    IF  POS                          ! MAC executes if previous ALU POS


NOP                       ADD A, B                         ! CCR set by ALU
{result X gpr3 > gpr4}    MOV  gpr1 > gpr7, POS > CMR      ! MAC executes if previous ALU POS


NOP                       ADD A, B                         ! CCR set by ALU
{result X gpr3 > gpr4}    {MOV  gpr1 > gpr7, POS > CMR}    ! MAC and ALU conditional


NOP                       ADD A, B                         ! CCR set by ALU
NOP                       {IF Condition}                   ! leave IFLG and IOZ flags alone
{some_instruction}        MOV  #POS > CMR                  ! MAC conditional on Condition
{result X gpr3 > gpr4}    {MOV  gpr1 > gpr7}               ! MAC and ALU conditional on POS


NOP                       ADD A, B                         ! CCR set by ALU.
{some_instruction}        IF  POS                          ! alatent!
{result X gpr3 > gpr4}    {MOV  gpr1 > gpr7}               ! 2 MACs and ALU conditional on POS
```

In all cases above, the AGEN fulfills the same latency rules as the MAC unit.

```
                                                   AGEN
NOP                     ADD A, B               NOP                   ! CCR set by ALU
NOP                     IF Condition                                 ! IFLG and IOZ flags possibly altered
MOV  #POS > CMR         {some_instruction}     {some_instruction}    ! ALU and AGEN conditional on Condition
{result X gpr3 > gpr4}  {MOV  gpr1 > gpr7}     {some_instruction}    ! MAC,ALU,and AGEN conditional on POS


                  error
MOV  #Condition > CMR   cc-class ALU instruction   ! illegal; NOT ALLOWED BY HARDWARE
```

2.6.3. Exceptions to Conditional Execution

There are four cc-class instructions which incorporate a simultaneous preload of CMR as part of their designed function: these include Jcc, JScc, RScc, and MOVcc. The move to CMR in these cases always takes place and cannot be inhibited. This must be considered in any conditional execution of these instructions.

One immediate consequence regards the IF pseudo as previously defined. Writing

{IF Condition}

accomplishes the same unconditional preload of the CMR as does

IF Condition

But {IF Condition} is useful when it is desired to update none of the bits in the CCR; without {}, the IFLG and IOZ flags will be updated. This is the second reason to use the IF pseudo in preference to an explicit unconditional MOV to CMR.

If it is truly desired that an IF Condition statement be conditional with regard to the loading of the CMR, then the programmer can always resort to the explicit,

{MOV #Condition > CMR}

keeping in mind the increased latency as illustrated in the examples above. As always, any conditionally executed instruction leaves all the CCR bits unchanged.

Instructions which cannot be used reliably within a conditional *block* of code are the LIM, SUBB, ASDL, and ADDC instructions. This is so because these ALU instructions incorporate the CCR flag states in the execution of their prescribed function. Since any previously queued conditionally executed instruction {} does not alter the CCR by design, then these four instructions would not be given their proper requirements. The conditional execution of these instructions is not prohibited by the assembler *although warnings are issued when they are found conditional*. These warnings are reminders that it is the previous queued instructions to look out for.

Conditionally executed AGEN coding in the MAC unit or ALU instruction field **(**see the section ahead on the AGEN**)** is **dangerous**. This is because the scheduled AGEN instruction-field {code} produced by the assembler **(**found in the *AGEN listing***)** is most often not on the same instruction line as the MAC unit or ALU field source code which requested the external memory access. In that case, the CCR is not necessarily in the same state on both the requesting instruction line and the scheduled line. For this reason, conditionally executed AGEN coding in the MAC unit or ALU instruction field should be written carefully, also making sure the CMR is in the desired state in both locations. The assembler does not disallow this type of coding, although appropriate warnings are issued.

2.6.4. Conditional Execution, in Summary:

- Regarding one line of code, any function unit's operation **(**the MAC unit's, and/or the ALU's, and/or the AGEN's**)** can be conditionally executed {} independently of any other but based on the same CMR and CCR.

- The CCR is automatically set only by the ALU.

- All ALU instructions automatically update the IFLG and IOZ flags in the CCR.

- Conditionally executable instructions {} never alter the CCR, regardless of whether they are executed.

- The ALU's CCR result is discernible by all function units on the next queued line of code for the purpose of conditional execution.

- All cc-class instructions unconditionally preload the CMR. The new CMR applies to all conditionally executable operations for all function units appearing on the same program line and on all subsequent queued lines until another Condition is loaded.

- When the programmer does not specify a preload to CMR, the assembler will prefer to utilize the MOV instruction instead of MOVcc.[27]

- If the programmer does not specify a preload of the CMR for JMP class instructions, the assembler will choose the ALW (*always*) Condition.[28]

- Conditionally executable instructions {} always consume program time equal to one instruction cycle.

[27]Discussed under Branching, Moving, and Pseudo Instructions.
[28]ditto

2.7. Shift Pseudos in the MAC unit and ALU / *Constant Expressions*

The ALU has the fundamental AS and LS instructions which store the shift amount in the A operand. That is, a GPR/AOR can be used to hold the shift amount. This is useful for computed shifts. Within the normal MAC unit syntax we have the >> *or* << operator which specifies that a shift amount follows in a constant-expression. This constant is stored in the MAC unit field of the microinstruction 96-bit word.

Both the MAC unit and ALU have shift pseudo instructions as an augmentation to the normal syntax. The ALU has two shift pseudo instructions which allow the specification of constant shift amounts. If a computed amount is desired, then the ALU shift instructions (AS, LS) should be used instead. The syntax for the ALU shift pseudo instructions is:

ASH B >>*const.expr* > C **or** ASH B <<*const.expr* > C ! arithmetic

LSH B >>*const.expr* > C **or** LSH B <<*const.expr* > C ! logical

The value of the constant expression, *const.expr,* is actually stored in the A operand of the AS and LS instructions respectively. The constant expression in the pseudo can be positive or negative.

A *constant expression* is a mathematical statement of constants and/or symbolic constants such as:

INT**(**LOG(N)/LOG(Radix)**)** - 7

Register names are not generally allowed in such expressions because of ambiguities which arise in the syntax. Although the # symbol is often used **(**always placed at the beginning of an expression**)** to extract the value of all following symbols such as register names, it is not allowed where a constant expression is <u>expected</u> by the assembler.

The MAC unit has a shift pseudo instruction which has the same syntax as the ASH pseudo of the ALU. This homogeneity is desirable.

ASH D >>*const.expr* > F **or** ASH D <<*const.expr* > F !arithmetic

The shift amount is, as for the primitive MAC unit syntax, stored within the MAC unit microinstruction <u>field</u>. For both the ALU and MAC unit,

ASH *dest* >>*const.expr*

is an acceptable pseudo instruction syntax implying *dest* as the destination.

Now we give some examples of constant expressions appearing in the code which are allowed or not:

```
    ASH  dest >>3          ! ok
    ASH  dest <<1+16/4     ! ok
    ASH  dest <<-1         ! ok
```

but,

```
           error
    ASH  dest <<gpr        ! illegal
```

This is an error because it might imply that the contents of gpr is used as the shift amount; some GPR name, gpr, was erroneously used in a place where a constant expression is expected. **(**The fundamental AS and LS instruction use the contents of the A operand (the programmer's first operand) as the shift amount.**)**

```
               error
     ASH  dest  >>#(1+gpr)       ! illegal because # disallowed where constant expression expected
```

The # symbol in this expression implies that some GPR would be allocated and initialized to the value of (1+gpr) at that point. **(**The value of the name, gpr, extracted by #, is its register address.**)** But the # is not <u>expected</u> by the assembler in front of this constant expression, so it is an error.

```
             error
     ASH  dest  <<#1          ! illegal  because # disallowed for same reason as before.
```

Wherever constant expressions are expected, the same rules apply as in the previous examples; external memory array indices, for example:

```
               error
      MOV  delay[#(3+gpr)] > dest        ! illegal
but,
      MOV  delay[3 + 7/3] > dest         ! is  legal
```

The # symbol generated an error because it is a directive to allocate some GPR, when encountered in the code, initialized to the value of the succeeding expression. This is certainly <u>not</u> what is expected from a constant expression.

```
                   error
       MOV  delay[3+gpr]  >  dest        !  illegal
```

The expression within [ ] is generating an error because the meaning of the gpr register is ambiguous. This usage might imply that its <u>contents</u> are being used as some index. As this is not within the capabilities of the ESP2, this construct is disallowed by the assembler.

3. MAC unit

The MAC unit has 20 distinct fundamental instructions and 10 variants which provide many combinations of shifting, multiplying, and accumulating(±). The specific MAC unit operation is specified by the placement of mathematical symbols between the operands (unlike the ALU).

3.1. Sources

The first source operand, D, can be nearly any register on chip. The second source, the E operand, is restricted however; it may not access AORs. The reserved upper case X symbol represents the operation multiplication, and must always be present between the two sources; viz,

```
D X E > MAC, F
```

A third source operand (the seed source) may be either the double precision MAC latch, MACP (the MAC Preload latch), or MACZ **(**the MAC unit's internal zero seed source, MACZERO**)**, followed by a + or **-** . For example,

```
    MAC   +  D X E  > MAC, F
    MACP  -  D X E  >  F
    MAC  -  coef1 X lastx  > MAC
    MACZ  -  D X E  > MAC, F          same as        -D X E > MAC, F
```

When no third <u>source</u> operand is specified, MACZ is assumed.

3.2. Destinations

The MAC unit destinations <u>must</u> be specified; there is no default destination as there is in the syntax of the ALU. The MAC unit accumulator result can be written to either or both the double precision MAC latch and the single precision destination register specified by the F operand (24 MSBs).

```
    D X E > MAC, lastx          means          MACZ + D X E > MAC, lastx
```

When only the MAC latch is specified as a destination, the F destination operand is assembled to be the ZERO SPR **(**which is read-only, and different from MACZERO**)**.

```
    D X E > MAC                 means          MACZ + D X E > MAC, ZERO
```

In many constructions it is possible to write the result to F and *not to* the MAC latch!

```
    D X E > coef1               means          MACZ + D X E > coef1
```

The previous MAC latch contents are preserved.

No matter what destinations are selected by the programmer, the low-order conditionally saturated MAC unit results (24 LSBs) are sent to the MACRL (MAC Result Low) latch on every instruction cycle.

3.3. The MAC unit Registers

The MACP latch is a Preload register which seeds the accumulator under program control. **(**Review, at this point, the MAC unit architecture diagram in the Chip Spec.**)** The **accumulator** can take as seed-source either the MAC latch, MACP, or MACZ (MACZERO).

Both MAC and MACP are latches internal to the MAC function unit. When directly reading the MAC unit, one is accessing the unsaturated MAC latch. When directly writing to the MAC unit, one accesses the MACP latch. These two latches are accessible via SPRs. One nice consequence of this is that the SPRs associated with the MAC function unit are allowed as sources and destinations to the MAC unit itself, as would be any other SPR, but with two restrictions:

3.3.1. Writing to MACP

When **writing** to the MAC unit, one accesses the MACP latch. The high-order 24 bits can be initialized via loads of the MACP_HC or MACP_H SPRs. When the programmer types the pseudonym, MACP, as a destination, this is defined as equivalent to the SPR, MACP_HC, by the assembler. **(**A load of MACP_HC clears the low 24 bits of the MACP latch while loading the high-order bits. A load of MACP_H is a simple load to the high-order MACP bits.**)** *From the ALU, specifying MAC as a destination is disallowed by the assembler*. This is because one might be led to believe that the MAC latch itself may be preloaded; this is not the case. Only the MACP latch can be initialized. The low-order MACP bits can be initialized via loads of either the MACP_LS or MACP_L registers. The former sign-extends into the upper 24 bits. The four MACP_ SPRs are defined as write-only by the assembler. The MACP pseudonym is similarly defined as write-only except when used as the accumulator seed.

3.3.2. Reading from MAC

When **reading** from the MAC unit, one accesses the unsaturated MAC latch. Here the programmer refers to the SPR, MACH, to access the high 24-bit word, and to the SPR, MACL, to access the low word. Both MACH and MACL are defined by the assembler to be read-only. It does not make sense to use MACP as a source since it is the MAC latch, not the MACP latch, which would be read. *The assembler will disallow any reference to MACP as a source operand which is not the seed source in the MAC unit.* The pseudonym, MAC, when used as a source operand is defined as equivalent to the SPR, MACH, by the assembler. Since when sourcing MAC one refers to the unsaturated MAC latch, most of the time the programmer will prefer to access intermediate MAC unit results from the Z destination buss (see the Chip Spec.), taking advantage of the three-operand architecture.

3.3.3. Reading from MACRL

Low-order MAC unit results can also be obtained from another place which is right outside the MAC unit called the MACRL latch. If the programmer does not specify a destination other than MAC, the assembler substitutes the ZERO SPR. So, this read-only SPR, MACRL, is loaded on every instruction cycle with conditionally saturated low-order 24-bit results from the MAC unit output.[29] The MAC unit NOP has been designed to preserve the contents of MACRL; this SPR is not writable.

Alternatively, **un**saturated low-order MAC unit results can be read directly from the MACL register, which is part of the MAC latch.

3.4. Saturation

When an accumulator result goes to the MAC latch, it is always unsaturated. When the MAC unit writes a result to a destination other than the MAC latch **(**to some GPR or AOR, for example**)**, then and only then will conditional saturation occur. Saturation decisions are based upon the thirteen MSBs of the 60-bit Barrel Shifter output. **(**See the Chip Spec. for more details.**)** There also exists a register for the special purpose of reading conditionally saturated low-order MAC unit results: it is called MACRL.

3.5. Barrel Shifter

The MAC unit multiplies two signed 24-bit operands and produces a signed 48-bit product and four more overflow bits. Analytically, in fixed-point (non-integer) arithmetic, multiplying two **q**23 24-bit numbers would result in a **q**46 48-bit format; i.e., a number having the binary point after the 46th bit counting from the LSB, and having a redundant sign bit. But, *the product in the ESP2 is always shifted left once*, to produce a **q**47 48-bit result in the present case. Usually, the high 24-bit word **(**not including the 4 overflow (guard) bits**)** which is then in **q**23, is taken out to the destination along the Z buss.

Beyond this permanent product shift left, a programmable 60-bit Barrel Shifter is available in the MAC unit at every instruction cycle operating on 52 bits **(**which includes the four guard bits**)** of either a seed source or an accumulated result. This feature allows us to be more selective about the alignment of inputs to the accumulator or about the accumulator bits appearing at the MAC unit output. In other words, the Barrel Shifter facilitates fixed-point arithmetic at various binary points. Note that the binary point in the MAC latch (as output) and the destination register can be in different locations when the Barrel Shifter is used to shift the accumulator output to a destination. This is because of the position of the Barrel Shifter in the MAC unit output path.

A shift may be specified to the right or left using >>***n*** or <<***n*** , respectively. *n* must be a constant expression from 1 through 7 for right shifts, or 1 through 8 for left shifts. *n* can be 0 in which case the shift operation need not be specified. We will indicate just one direction in the remainder of the text for simplicity (*n* may be negative). The placement of the shift operator determines precisely what will get shifted. Three legal placement options are:

[29]This stands in contrast with the 52-bit MAC latch to which the storing of results can be selectively inhibited.

1) MAC >>*n* + D X E > MAC, F !*shifting the accumulator seed source*

2) MAC + D X E > MAC >>*n* > F !*shifting into the destination*
Programmers take note that this particular instruction says that the result left in the MAC latch is not shifted.

3) MACP + D X E >>*n* > F !*means* (MACP + D X E) >>*n* > F
This shifts into the destination but deposits nothing to the MAC latch!

Right shifts are sign extended while left shifts are zero filled and saturated if required; these are double precision arithmetic shifts. Double precision logical shifts may be performed in the ALU if need be.

3.6. MAC unit MOV Pseudo Instruction / Homogeneity

The MAC unit can easily be made to MOV data without alteration; thus there is a very useful MOV pseudo instruction in the MAC unit having the same syntax as the MOV in the ALU. The reader is referred to the Chip Spec.

In our discussion of Shift Pseudos, we encountered the ASH pseudo which is homogenous in the MAC unit and the ALU. Perusal of the MAC unit pseudo instructions in Part I will reveal a high degree of similarity with the instructions and pseudo instructions in the ALU. This is a purposeful design feature of this assembly language whose justification requires an understanding of the programming process in a parallel architecture such as ESP2. Very often the programmer finds it necessary to cram the maximum amount of functionality into a small program space. The available space is dictated by outside constraints such as the given sample period. The optimal efficiency of an ESP2 program comes about when the MAC unit, ALU, and AGEN unit are roughly equalized in the number of instructions executed per sample period.[30] This is more easily accomplished when the language is homogenous among the parallel function units.

3.6.1. MAC unit XCH Macro Pseudo Instruction

Utilizing the MOV pseudo instruction in the MAC unit, and coding a reverse MOV on the same program line in the ALU, then taking advantage of the destination latencies, one can invent a *macro* pseudo instruction which exchanges the contents of two sources. Due to inter-unit latency, the complete result of that exchange is available to the ALU on the next queued line of code, but is available to the MAC unit on the second queued line following the macro. **(**This coding is 33% more efficient, regarding number of instructions, than having either MAC unit or ALU alone do the exchange. The reader is referred to the Chip Spec.**)**

[30]To attain that ideal, the programmer will often play the *chiclets* game; popular around 1960.

3.7. Listing of MAC unit Instructions

Notice that every MAC unit instruction has a destination operand, F. The assembler's default destination F operand is the read-only ZERO SPR. This implies that for those instructions having both MAC and F as allowable destinations, the programmer can effectively indicate only the MAC latch as the desired destination. This yields 10 variants of the existing instructions.

Remember that when MACP is specified as the destination (F) operand, it gets single precision results, whereas the MAC latch destination, as fed from the accumulator, always receives double precision results. Both MAC and MACP as seed source are double precision, however.

Also notice that the MAC latch destination and the destination register do not always receive the same shifted results.

TABLE MACLIST. Fundamental MAC unit Instructions.

instruction	comment
D X E > MAC >>*n* > F	
-D X E > MAC >>*n* > F	*variant*: -D X E > MAC
D X E >>*n* > F	
-D X E >>*n* > F	
MAC + D X E > MAC >>*n* > F	
MAC - D X E > MAC >>*n* > F	
MAC + D X E >>*n* > F	(MAC + D X E) >>*n* > F
MAC - D X E >>*n* > F	*ditto* -
MAC >>*n* + D X E > MAC, F	
MAC >>*n* - D X E > MAC, F	
MAC >>*n* + D X E > F	
MAC >>*n* - D X E > F	
MACP + D X E > MAC >>*n* > F	
MACP - D X E > MAC >>*n* > F	
MACP + D X E >>*n* > F	(MACP + D X E) >>*n* > F
MACP - D X E >>*n* > F	*ditto* -
MACP >>*n* + D X E > MAC, F	
MACP >>*n* - D X E > MAC, F	
MACP >>*n* + D X E > F	
MACP >>*n* - D X E > F	

4. ALU

The ALU is the most conventional and general-purpose of the three function units. It can conditionally affect the Program Counter resulting in branching and calls to subroutines. It can execute 32 different opcodes, but the programmer's assembly language will contain many more instructions. Since the ALU is a three-operand device (as is the MAC unit), flexibility in the choice of these operands admits many variants of the fundamental ALU instructions. These variants are called *pseudo instructions* and account for the excess of instructions beyond 32. The ESP2 Chip Spec. explains all the instructions and how the pseudo instructions are constructed from the 32 fundamental instructions.

4.1. Three Operand Instructions

Most ALU instruction have three operands, two sources, A and B, and one destination, C. The sources are separated by commas, the destination by a > (right chevron).

OPERATION A , B > C

Examples of instructions having three operands are:
ADD, ADDV, ADDC, SUB, SUBV, SUBB, SUBREV, MAX, MIN, AND, OR, XOR, RECT, AVG, AMDF, LIM, AS, LS

When the destination is not supplied, the C operand will be assumed to be the same as B.

ADD this, that *is short for* ADD this, that > that

This particular convention is germane only to the ALU.

4.1.1. Compare

The CMP (*compare*) pseudo instruction is derived from the instruction SUBREV by utilizing the read-only SPR called ZERO as the destination:

CMP *this*, *that*

The ALU will perform the following operation:

ZERO = *this* - *that*

The programmer can think of it as 'compare *this* to *that*'. Later in the code where a conditionally executable instruction { } is encountered, if GT were active in the CMR, for example, then the instruction would be read: 'if *this* is greater than *that*, then <u>do</u> this conditionally executable instruction'.

Three SUB-class instructions (SUB, SUBB, SUBV) are reversed in operand order with regard to the SUBREV instruction. For example,

SUB *this*, *that* > *there*

should be thought of as 'subtract *this* from *that* and put the result in *there*'. The ALU performs the following operation:

there = *that* - *this*

4.2. Two Operands: One Source, One Destination

Some instructions use only one source operand. The destination is separated by a right chevron. The most common is:

MOV B > C

The instructions, MOV, BREV, and DREV, are examples of instructions having only one source, one destination.

For one-source instructions, when the destination is not supplied, the source will also be interpreted as the destination.

BREV gpr *is short for* BREV gpr > gpr

where gpr is assigned to both the B and C operands.

Again, this convention is germane to the ALU.

4.3. Fundamental ALU Instructions Having No Operands

BIOZ instruction

The operands of the BIOZ and HOST instructions (and the ALU NOP pseudo) are usurped because of the need to refresh internal registers transparently. BIOZ always performs at least one refresh, while HOST performs refresh only if no host access is pending.

The purpose of the ALU's BIOZ instruction is to conditionally suspend program execution, but it has an execution latency of one instruction cycle. This means that two lines of code will be executed once whenever this instruction is encountered; the instruction line queued for execution following the BIOZ, is executed prior to suspension. When suspension occurs, the Program Counter (PC) becomes frozen at the second instruction line queued for execution following the BIOZ. When the PC freezes due to BIOZ, we say that the chip is in *suspension*, as opposed to a halt. This is because the suspension only lasts until a positive transition occurs (at the sample rate) at the IOZ input pin, which is part of the synchronization interface. When the program resumes, it begins at the second queued instruction line following BIOZ.

The Principle of Average Excess

Suspension is, therefore, a means of synchronizing program execution to the sample rate. If the positive transition at the IOZ pin has already occurred by the time BIOZ is reached, then there will be no suspension. We can use this fact to squeeze a lot more performance out of the programs that we write.

The ESP2 synchronization interface allows the run-time of a cyclic program to momentarily ***exceed*** *the sample period so long as the* ***average*** *run-time is less than the sample period.*

Stated differently; The average excess time spent by the main program cycle (the main loop) beyond the sample period must be ≤ 0 in a sample synchronous ESP2 system.[31] The caveat here is that the cumulative excess beyond the sample period can never exceed one more sample period; not even momentarily. Otherwise the SER data SPR transfers, latched by the transit high of the serial interface pin signal called LRCLK, will miss their launch window. Because LRCLK is often tied to the IOZ pin, this is why we recommend SER access towards the beginning of a program.[32]

The Programmer's Scope Loop

The parallel programming paradigm

MAC-operation   ALU-operation   AGEN-operation

allows the programmer to place a dummy external data-memory (AGEN unit) access on any available program line; say, alongside some chosen MAC unit or ALU instruction deemed a landmark to the program's cyclic operation. By placing such an access on the same program line as BIOZ, for example, the programmer can observe excess run-time with respect to the sample period, as discussed above. The programmer might schedule a dummy read to physically non-existent external memory;[33] say, at \$800000. Address bit 23 on pin maddr[23] would then be asserted every time the dummy read were executed.[34] That pin would be monitored on a dual-trace oscilloscope, the second trace monitoring the signal at pin LRCLK for the sample period. By visual comparison, the programmer would then be able to easily discern whether the average main loop duration were less than the sample period.[35] Ordinarily, the programmer would have foreknowledge of this by design. Only the most intricate programs might require such monitoring.

[31]We guise the Principle in terms of excess time to emphasize that many contemporary chip designs do not allow such excursions beyond the sample period. Negative excess time spent is interpreted as a main loop of average duration shorter than the sample period.

[32]but **not** on the next queued program line following BIOZ.

[33]The ESP2 is designed for 24-bit addressing to 16 Mega-word (24-bit) external data-memory.

[34]Recall that BIOZ executes once and then suspends the ESP2 if necessary; i.e., the chip is not designed to 'loop' on the BIOZ instruction.

[35]This method of observation was invaluable during the development of a product based upon this Principle. [Dattorro2400] The Lexicon Model 2400 was indeed designed to execute a cyclic program whose run-time often exceeded the sample period.

We elaborate on this debugging technique a little:[36]

```
              .
              .
DEFREGION  I  @$800000
    dummyAor = 0
DEFSPR
    DIL0                                    ! reserve a DIL for personal use
              .
              .
CODE
              .
              .
NOP        BIOZ        RD *dummyAor > DIL0
              .
              .
```

The partial program above shows one way to accomplish the proposed objective which is to assert a synchronous pulse on external data-memory address line 23. This method is independent of the type of external memory; i.e., DRAM/SRAM. The upper address pins, maddr[16] through maddr[23], do not multiplex (row/column) when in DRAM mode. Therefore the observed state of an individual pin will be the same regardless of the setting of the MUX_ADDR bit in the HARD_CONF SPR.

If all legitimate accesses to external data-memory are to regions where the maddr[23] pin is logical 0, then the dummy access will place a logical 1 **(**+5V**)** on the pin for one instruction cycle. If any AGEN NOP cycles immediately follow in the program, the pin will remain high until the next scheduled external memory cycle, because the external memory address pins tristate on NOP cycles. It is prudent, then, to attach a pull-down resistor to that pin so that the line is pulled low at tristate. The pull-down resistor will typically cause slower (more rounded) transitions, thereby allowing one to distinguish pin-driven transitions from pin tristate transitions.

If this feature is desired within a development environment, the system designer has the option of placing pull-down or pull-up resistors on the busses that tristate.

Since BIOZ is an ALU instruction, scheduling an external memory operation on the same program line will cause the external memory buss cycle to straddle the second half of the BIOZ and the first half of the instruction cycle following BIOZ.

HOST instruction
The BIOZ instruction includes within it another ALU instruction called HOST. HOST is itself an independent fundamental ESP2 instruction synchronizing run-time host access, to/from any one internal 24-bit register, using the standard host/ESP2-register interface. **(**Instruction memory is **always** accessible at the instruction rate, and internal register memory is accessible at the instruction rate during halt.**)** If multiple HOST instructions are executed **(**as in the BIOZ instruction during suspension**)**, then it becomes possible, for example, to update all the coefficients of an **IIR** (infinite impulse response) filter at once, at the instruction rate, at run-time.

[36]The reader may first want to review the section covering AGEN unit programming before delving further into this scheme. Familiarity with the hardware configuration register (HARD_CONF from the Chip Spec.) and its control over the external data-memory interface would be helpful.

In the event that an ESP2 program is written without the use of the BIOZ instruction, the programmer must remember to include at least one HOST instruction somewhere in the program for internal register access at run-time from the system host. Since BIOZ executes the HOST instruction only during suspension, if the number of instructions in every program main loop equals or exceeds the sample period, then BIOZ will not allow host interaction.

4.4. Branching, Moving, and Pseudo Instructions

It is very important to distinguish fundamental ESP2 instructions from *pseudo* instructions which are formulated within the assembler as source/destination variations of the fundamental instructions. There are 32 ALU and 20 MAC unit fundamental instructions while the number of useful variations and pseudo instructions number much more.

4.4.1. MOV Pseudo Instruction

There exist two MOV instructions in the ALU: MOV and MOVcc. MOV does not alter the CMR, while MOVcc always preloads it. In either MOV, the B operand always goes to the destination register specified by the C operand, as in

MOV B > C

All the cc-class instructions utilize the A operand **field** and a special reduced-latency data path to make the unconditional preload of the CMR. *So, strictly speaking, the following cc-class MOV instructions are pseudo instructions because the arrangement of operands does not correspond to the instruction primitive*:

unconditional move with unconditional preload of CMR:

MOV B > C, *cc* > CMR

conditional move (based on new CMR) with unconditional preload of CMR:

{MOV B > C, *cc* > CMR}

where *cc* can be: EQ, NEQ, GT, LT, GTE, LTE, HI, LO, HS, LS, POS, APOS, BPOS, NEG, ANEG, BNEG, OV, NV, CC, CS, ALW, NEV, IOZ, NIOZ, IFLG, NIFLG, Z, NZ **(**note: CS = LO, CC = HS, Z (the *zero* Condition) = EQ, NZ = NEQ**)**.

When the programmer does not specify a load to CMR, the assembler will utilize the MOV instruction instead of MOVcc. This is necessary so as to avoid modifying the CMR unintentionally with this ubiquitous instruction.

It is interesting to note at this point that none of the fundamental cc-class instructions **(**Jcc, JScc, RScc, MOVcc**)** are themselves recognized by the assembler; only the pseudos presented herein are recognized.[37]

[37]The MAC unit has a MOV pseudo instruction having the same syntax as MOV in the ALU.

4.4.2. JMP class Pseudo Instruction

In the ESP2, the branching instructions are three operand ALU cc-class instructions. The JMP class (Jcc, JScc, RScc) instruction operands are arranged as follows:

```
        Jcc  Condition, label  >  ZERO              !  =  Jcc A, B > C    ;syntax not used
```

The destination register, the read-only SPR ZERO, is supplied by the assembler in the cases of Jcc and JScc. In actuality, the destination could be any register. Were the assembler to assign some other register as the destination, then the contents of a register, whose address is the same as the value of *label*, would be written to the assigned destination. Since it would be of little utility to load some other destination register, the destination ZERO is the most logical choice. The reason that this move to ZERO happens is because the new PC location denoted by *label* is assembled into the B operand **field** of the microinstruction 96-bit word. Although the B field is not a GPR address, a MOV of the B operand to the C operand occurs along the normal ALU data path by design. Most ESP2 instructions act this way.

The B operand field, holding the value *label*, is forced into the PC along a special reduced-latency data path if the branch is taken. This special path provides a quicker update of the PC. There are, so far, two destinations: the ZERO register and the PC.

The Condition is hard-coded into the A operand **field** of the microinstruction by the assembler, and **always** gets moved into the CMR prior to any decision to branch; this, in fact, makes a third destination! The CMR is unconditionally preloaded by a cc-class instruction whether or not conditionally executed.

The skip bit is discussed in the section on Conditional Execution. Any instruction field can be conditionally executed if the current Condition Code, the CCR set only by the ALU, satisfies the CMR, and if the skip bit associated with that instruction field is asserted by the programmer. Branch instructions constitute special moves to the SPR called PC. Since any instruction can be conditionally executed, then in this manner we accomplish conditional branching.

The programmer does not use the syntax above because the JMP class instructions are difficult to interpret. The syntaxes shown below are more explicit, indicating first where the branch is to, and then indicating that there is a preload of the SPR called CMR (Condition Mask Register) on the same program line. The CMR is then used immediately in a decision {} to branch.

Since there is no primitive instruction corresponding to the arrangement of operands as shown below, these branch instructions, strictly speaking, are pseudo instructions because their operands must be rearranged by the assembler into the format specified above (in Jcc):

***JMP Pseudo Instructions*:**

unconditional branch:

```
    JMP  label
    JMP  label,  cc > CMR              !Jcc having skip bit not asserted.
   {JMP  label,  ALW > CMR}            !skip bit asserted
```

conditional branch (based on unconditionally preloaded new CMR):

```
   {JMP  label,  cc > CMR}             ! meaning: branch if Condition (cc) is true
```

unconditional branch to subroutine:

```
    JS  label
    JS  label,  cc > CMR               ! JScc  having skip bit not asserted
```

conditional branch to subroutine:

```
   {JS  label,  cc > CMR}              ! unconditional preload of CMR
```

unconditional return from subroutine:

```
    RS
    RS,  cc > CMR                      ! Note use of comma for preload to CMR.
```

conditional return from subroutine:

```
   {RS,  cc > CMR}                     ! unconditional preload of CMR
```

If the skip bit is <u>not</u> set by the programmer, then all JMP class instructions will be unconditional regardless of the particular Condition sent to the CMR.
Whether or not the skip bit is set in the ALU, there will <u>always</u> be a preload of the CMR by the JMP class instructions. If the programmer does not specify a preload of the CMR, the assembler will choose the ALW (*always*) Condition.

The line of code queued for execution following Jcc, JScc, and RScc will always be executed; i.e., these instructions have an execution latency of one instruction cycle.

All JMP class instructions (Jcc, JScc, RScc) and REPT always leap all three function unit parallel instructions. **The programmer must be cognizant of any AGEN instructions scheduled by the assembler appearing on any instruction lines surrounding the branch (see the .agn listing), because they may be unintentionally missed or erroneously executed.**

In general, MOV to the PC along the normal ALU data path is ill-advised.

4.5. Low-Overhead Looping

The ALU REPT instruction is a three operand instruction, having extra destinations, designed to provide an efficient looping mechanism. The label whose value is the PC location of the last instruction in the block, always gets coded into the A operand **field**; <u>it is not the address of a GPR</u>, so no GPR must be allocated to hold that value. The second operand or operand field is the repeat count, the number of times a block of code will be **repeated**. This means that the number of loops is always 1 more than the repeat count, and the block of code following REPT will always be executed at least once.

```
first_form:        NOP              REPT  last_instruction, NUM_REPEATS
loop_start:        somegpr X result > somewhere
last_instruction:  somegpr X someother > someother
```

The destination operand, if not specified, is supplied by the assembler as ZERO. When specified as in this example, the <u>constant expression</u>, NUM_REPEATS, gets coded into the B operand **field** of the instruction and determines the number of repeats. Therefore <u>no</u> GPR must be allocated by the assembler to hold that value.

When the B operand field holds the number of repeats, the expected constant value has a maximum of 1023 since it is a 10-bit field. This means that looping more than 1024 times must use a second form of the instruction. When the destination (the SPR called) REPT_CNT is specified, then the contents of the GPR named in the B operand determines the number of repeats;

```
second_form:    NOP              REPT  last_instruction, repeat_count > REPT_CNT
                                   :
```

The user-named GPR called repeat_count holds the number of repeats, which has a maximum of 2^{24} - 1, and is moved to REPT_CNT along the normal ALU data path. No other destinations are allowed by the assembler.

*This interpretation of the operands is dictated by the elegant REPT microinstruction which first loads the B operand **field** into the SPR REPT_CNT, and then loads the register contents pointed to by the B operand field (the B operand) into the destination denoted by the C operand. If the destination operand is specified as REPT_CNT, then it gets loaded twice in succession!*

There are some special rules that must be observed by both the programmer and the assembler regarding successful implementation of one-line loops. For a more thorough description of the REPT instruction, see the Chip Spec.

The repeated block of code covers all three function units. **The programmer must be aware of any AGEN instructions scheduled by the assembler appearing on any program line in the block (see the .agn listing), because they will be repeated too. If a *delayspec* appears in the block but it is scheduled outside the block, the block may need to be extended.**

5. AGEN

The purpose of the AGEN is to read and write external data-memory. The AGEN utilizes the G operand, which is always of type AOR, to calculate a new external memory address on every instruction cycle. The fundamental AGEN address calculations are subject to modulo arithmetic by design, but AGEN easily adapts to addressing in absolute or relative fashion.

While the AGEN has several powerful modes of address calculation, the programmer is free to employ GPRs or AORs to perform more intensive address computations in either the MAC unit or the ALU. Both the instruction set and the language syntax support programmed address computations. The final result of an address computation must ultimately reside in an AOR.

An external memory array (delayline, peripheral I/O, table) must reside in some region. A region is typically a circular buffer in absolute address space having associated with it BASE*r*, SIZEM1*r*, and END*r* region control registers. The region control registers, mapped as SPRs, serve to bound the region for automatic modulo addressing with respect to the region modulus.[38] That external memory array must be associated with one of the eight hardware-supported regions in the chip (which are named I, P, Q, R, S, T, U, and V) by declaring it so in a DEFREGION statement.

The address generator (AGEN) operations can be specified by the programmer in two ways:
1**)**by writing AGEN code directly on a program line in the AGEN instruction field, or
2**)**by the use of external memory references as sources or destinations of the ALU or MAC unit.

In the first case, the programmer is assisting in the automatic scheduling functions built into the assembler. In the latter case, **(**called 'AGEN coding in the MAC unit or ALU instruction field'**)** the assembler will automatically generate and schedule the appropriate AGEN code for the programmer which will appear in the *AGEN listing* (**.**agn).

<u>Whenever</u> AGEN code is generated, either by the programmer or the assembler, this is called *scheduling*.

[38]The size of the region is arbitrary.

5.1. AGEN Scheduling

Let ***delayspec*** represent any external address offset expression, yet to be defined. The assembler must resolve all the *delayspec*s appearing in the MAC unit and ALU instructions fields by scheduling the requested AGEN operations on external memory, if required.[39] More accurately, each *delayspec* specifies an address offset into external memory. These offsets must be held in AORs which either the assembler or the programmer can allocate and initialize. The process called **scheduling** determines the access sequence of each *delayspec* requesting access to external memory; both the assembler and programmer may take part in the generation of the required AGEN code. More than one access may be requested on each line of code. Only one access per line of code can be scheduled, however, indirecting the assigned AOR to RD external memory contents into an SPR called DIL, or to WR the contents of an SPR called DOL out to external memory. The DIL or DOL then replaces the programmer's reference to the requesting *delayspec* in the MAC unit or ALU instruction field after assembly. This action is seen in the AGEN listing.

The assembler will not wrap schedules it generates beyond user-defined program boundaries. Scheduling is always performed on program lines contiguous with the requesting *delayspec*. The assembler will complain if insufficient program space at the top or bottom prevents scheduling. The program boundaries are determined primarily from the existing lines of source code.

5.1.1. Meaning of *delayspec*

A chart of *delayspec* syntax is given later, and we will see that *delayspec* has many forms. The purpose of using a *delayspec* as an operand might be to access data residing off-chip in a large external memory. In other uses of *delayspec*, an external address offset may be in the process of being computed by the program, requiring no external access for the time being; i.e., just an AOR reference. The most common *delayspec* looks like an array reference in the C programming language. For example:

```
DEFREGION   Q[$10000]  @256
            mydelay[3032]
            mypointer = &mydelay[1]
CODE
NOP         NOP             RD  mydelay[123] > DILn
```

In our example, mydelay is the name of a *delayline* (type of external memory array) declared to reside in a specific region (Q) of an optionally specified size ($10000), which is in turn declared <u>or</u> determined by the assembler to start at a specific absolute location in external physical memory (256). This particular *delayspec*

mydelay[123] = *(&mydelay[123])

is requesting an external memory access to a certain element of the delayline in region Q. The delayline must have been previously declared while the *delayspec* need not have been. The name itself, mydelay **(**defined equivalent to the *delayspec* &mydelay[0] as in the C programming language**)**, connotes the relative address offset of the beginning

[39]Not all *delayspec*s demand external memory access. A *delayspec* can be an AOR reference, for example, hence no schedule is required. The *delayspec* Quote Scheme conserves external memory bandwidth by eliminating redundant accesses.

(the ***root***) of that delayline with respect to the physical start of the associated region. The index in the square brackets is expected to be a constant-expression and is usually meant to be the time delay in units of number-of-samples.

In the coding example given above, when mydelay[123] is encountered, the quantity,

```
                    address offset = root address offset of  mydelay[3032],  +  123
                                   = &mydelay[0] + 123
                                   = 123
```

is evaluated by the assembler.[40] The assembler must then allocate an Address Offset Register (AOR) implicitly associated with region Q by default, and initialize it to hold that quantity.

The *delayspec* in our example above, then, causes an AOR to be initialized and used by the AGEN unit to fetch the desired word of external memory data. The AGEN deposits that data into some DIL. The contents of that DIL will, most likely, later be used by the program. These are the actions and meaning of the *delayspec.*

Another useful form of *delayspec* utilizes an explicit reference to an AOR. For example:

```
      CODE
      NOP        NOP               RD  *mypointer > DILn
```

This *delayspec* requests an external memory access (* as in C programming). From the declarations above we see that *mypointer = *&mydelay[1] which we will find to be the same as mydelay[1]. So, in this second manner we can make reference to an element of a previously declared delayline. Unlike the case for the array form of *delayspec*, mydelay[123], <u>the AOR</u> mypointer <u>must have been declared</u>. The region with which it becomes associated by default is determined by the region in which it is declared.

[40]The <u>value</u> of the delayline name itself, mydelay, (i.e., the value extracted using #mydelay or #&mydelay[0]) is actually the register address of the AOR holding &mydelay[0] (the root address offset) in the same region as the delayline. How such an AOR comes into existence is the topic of more discussion. In any case, #mydelay[0] also yields the root.

5.1.2. Root Address Offset of an External Memory Array

Our earlier definition of *root* rightly expresses it as the relative delayline beginning, with respect to the start of the region. This is because the assembler determines it. When the assembler performs the initialization of the region control registers, BASE*r* begins life pointing to the start of the region in physical memory. At that time, when all the delayline (or table, or peripheral I/O) roots are being determined, the root has two identical derivations: as with respect to the physical region start, or with respect to the region BASE.

If the programmer overrides the region BASE initialization in a DEFSPR, the assembler will still determine roots with respect to the physical region start. More generally, programmer override of region BASE, SIZEM1, and END initialization is not observed by the assembler in its determination of AOR assignments.

It will become desirable to move BASE*r* around the region as discussed in the section called UPDATE region BASE. As BASE moves under program control, we consider it the new start of the region modulus (a circular buffer). Under this circumstance it is preferable to refine our definition of root to mean with respect to the region BASE. We can do this because all AGEN address calculations add offsets held in AORs to region BASE register contents to derive an absolute address. In any case, the value of the root is the same using either interpretation.

5.2 AGEN Instruction Field Coding

The AGEN field is optional for the programmer, and must be placed after the ALU operation on a program line. This code is typically generated (and scheduled) by the assembler in response to *delayspec*s found in the MAC unit and ALU instruction fields. The programmer who requires a more direct view of the scheduling for a piece of source code having intricate branching would be found typing code into the AGEN instruction field.[41] When the programmer performs the scheduling by typing AGEN code directly, we call this *programmer assisted scheduling.* The AGEN scheduler is sophisticated, however, and the programmer will find it difficult to exceed its efficiency.

The AGEN uses 16 Data Input Latches (DIL*n* = DIL0, DIL1, ..., DILF) and 16 Data Output Latches (DOL*n* = DOL0, DOL1, ..., DOLF) as 24-bit data buffers. These SPRs are the repository of incoming or outgoing data from/to external memory. Each one of these SPRs can be reused during the course of an ESP2 program, and so there is no limit imposed by the Data Latch resource upon the number of external memory accesses per sample period. There is a limit, imposed by other hardware constraints, of one external memory access per line of ESP2 code. **(**This does not prevent the programmer, however, from requesting multiple accesses in the various fields of the same program line.**)**

This is how AGEN code might appear in the AGEN instruction field:
The read syntax in the AGEN field is

RD *delayspec* > DIL*n*

whereas the write syntax is

WR DOL*n* > *delayspec*

5.2.3. DIL/DOL Reservation

Should the programmer wish to reserve any of the DILs or DOLs for private use, the act of declaring them within DEFSPR has the desired effect. The assembler then relinquishes those particular resources during scheduling.

If the programmer chooses to specify the DIL/DOL in the AGEN instruction field without having first reserved it in the declarations, the outcome can be unpredictable. This is because the assembler will consider all unreserved DIL/DOL SPRs as an available resource during scheduling.

[41]The AGEN listing shows all the AGEN code generated by the assembler and the programmer. Keep in mind that the AGEN listing (**.**agn) assembler output can be modified by the programmer and reassembled.

5.3. AGEN Coding in the MAC unit or ALU Instruction Field

This type of AGEN coding occurs when a *delayspec* is used as a source or destination in the MAC unit or ALU instruction fields of a program line. We are discussing AGEN programming which does <u>not</u> appear in the AGEN instruction field. For example:

```
CODE
NOP             ADD  coef1, coef2 > delayspec
```

An Address Offset Register would be allocated if this *delayspec* were not as yet encountered. If *delayspec* is requesting an <u>external memory write</u>, then this instruction would also cause the assembler to allocate the next available DOL*n*. The result of the ALU addition would then be placed into that DOL**.** On the following program line or any line thereafter, the AGEN could be scheduled to WR from DOL*n* to external memory as addressed by *delayspec*; this action would be seen in the AGEN listing.

In other words, *delayspec*, as specified by the programmer, comprises an address offset into a particular region of absolute memory. That region is also denoted[42] by *delayspec* and hard-coded as part of the instruction word which would perform the actual WR to external memory. Some AOR, typically determined by the assembler, is allocated to hold that offset into the specified region.[43] If an external memory access is requested by *delayspec,* then the AGEN uses that Offset Register operand to indirect the access. The DOL*n* SPR which then replaces *delayspec* as the ALU destination, would hold the write-data used in that access.

In the case that there is an external memory access request, if we were to examine the **.**agn listing produced by the assembler, we would find some AGEN dereference of that Address Offset Register which was allocated to *delayspec* in the assembly of our one-line program above. That dereference would of course reside on some line of code following the ADD, in the AGEN instruction field where the external memory access is scheduled. This latency is, in fact, the reason for the existence of multiple DOL SPRs; a multiplicity of writes to external memory can be queued-up if necessary, to occur at some later more opportune time.

The situation for <u>external memory reads</u> is quite similar. For example,

```
NOP             ADD  coef, delayspec > destreg
```

If *delayspec* is requesting an external memory access, then a DIL*n* is allocated. Here the assembler recognizes *delayspec* as an external memory reference and (if not already) an Address Offset Register is typically allocated. That AOR holds an address offset[44] into the region *r* in absolute memory. The particular region and AOR content become known either through the declaration of *delayspec*, or through the appearance of *delayspec* in the code alone.

[42]perhaps implicit by association with the region via declaration,

[43]**The programmer also has the means to explicitly declare, allocate, initialize, and name that associated Address Offset Register, by the way.**

[44]with respect to the corresponding region BASE*r* register,

At some time at least two lines prior to this program line, the assembler needs to have scheduled (generated AGEN instructions to perform) a RD from external memory to that allocated DIL*n*. That scheduled AGEN code employs the region and AOR denoted by *delayspec*. Since the word of data read from external memory has been deposited into the assigned DIL ahead of time, our line of code would be assembled substituting that DIL SPR in place of *delayspec*.

5.3.1. Example: AGEN Coding in the MAC unit Instruction Field

The following program lines show AGEN coding in the MAC unit field. The particular form of *delayspec* used is an explicit AOR dereference:

```
DEFREGION   T    @regionTstart
            sinctable[1024]
            sigmoidtable[1024]
            tablejump = 2
DEFREGION   S    @regionSstart
            sinc2table[1024]
            sigmoid2table[1024]

CODE
NOP                                 ADDV &sigmoidtable[0], #regionTstart > BASET
NOP                                 ADDV &sigmoid2table[0], #regionSstart > BASES
NOP                 !wait for BASET to get to AGEN, the first read should get scheduled here.
NOP
*tablejump        X coef1 > MAC                           !tablejump connotes region T.
*(tablejump)T   X coef2 > MAC
*(tablejump)S   X coef3 > MAC                             !region-S override.
```

The following shows another way of doing exactly the same thing, but this time by coding directly in the AGEN instruction field:

```
DEFSPR  DIL0       !reserve DIL0
CODE
NOP                                 ADDV  &sigmoidtable[0], #regionTstart  > BASET
NOP                                 ADDV  &sigmoid2table[0], #regionSstart  > BASES
NOP                                 NOP              RD *(tablejump)T  > DIL0
NOP                                 NOP              RD *(tablejump)T  > DIL0
DIL0  X coef1 > MAC                 NOP              RD *(tablejump)S  > DIL0
DIL0  X coef2 > MAC
DIL0  X coef3 > MAC
```

!But the CODE is identical to the **AGEN listing** (**.**agn) of the first way!

5.4. AGEN Listing (.agn)

The assembler produces a special listing of the programmer's original source code, but having appended in the AGEN instruction field the assembler-generated, assembler-scheduled AGEN instructions. This AGEN code consists of references to AORs as pointers to external memory, and references to DIL SPRs and DOL SPRs acting as repositories of external memory data. In place of the programmer's *delayspec*s in the MAC unit and ALU fields would appear the DILs and DOLs assigned and allocated by the assembler.

The AGEN listing can itself be assembled! Were we to reassemble this AGEN listing, an identical listing would be generated. See the Applications section for an example.

5.5. DIL/DOL Conservation; The *delayspec* Quote Scheme

The assembler will observe attempts by the programmer to assist in the scheduling if the conventions outlined here are followed:
We need a way to inform the assembler that programmer code is specifically written so as to inhibit the scheduling of some *delayspec* which appears as an operand in the MAC unit or ALU instruction fields. We adopt the convention that

"*delayspec*"

(in quotes in the code) appearing in only the MAC unit or ALU fields, specifies the same DIL/DOL as did the first previous reference to the same unquoted *delayspec* to the left or above in any instruction field.

If the first previous reference in the source code resides in a MAC unit or ALU field then we have accomplished *DIL/DOL conservation.* By conservation we mean that the number of external memory accesses has been reduced by reusing the contents of a previously loaded DIL/DOL. Since under these circumstances the assembler chooses the particular DIL/DOL used by the AGEN, then in this case it is up to the assembler to preserve the contents of that DIL or DOL for all occurrences of "*delayspec*" following *delayspec.*

Conservation can become critical during intensive external memory access where it is desired to reduce the required external memory bandwidth.

If the first previous reference in the source code to the same unquoted *delayspec* resides in the AGEN instruction field, then we have programmer-assisted scheduling. Actually, programmer assisted scheduling results whenever the programmer types code directly into the AGEN instruction field. The programmer must reserve the DIL/DOL used (via DEFSPR) in order to be absolutely sure that it is not trashed by a subsequent assembler schedule.

In either case, when the "*delayspec*" is encountered, the assembler inhibits a schedule for it. An assembly error is generated if there is no previous unquoted reference.

We refrain from dichotomizing between the treatment of *delayspec*s associated with RD operations and *delayspec*s associated with WR operations within our search pattern **(**to the left or above**)**. We do this because of a Reverb-type instruction sequence which is often useful:

```
                coef    X   signal  >  MAC, delay[0]
     MAC   +  coef2   X   signal2  > MAC
     MAC   -   coef3   X  "delay[0]"  >  delay2[100]
```

Since the *delayspec,* delay[0], connotes a specific DOL, say DOL*n*, then after assembly DOL*n* replaces delay[0] as the F operand in the first line of code and is reused in the third line of code in place of "delay[0]". This instruction sequence conserves external memory bandwidth and also preserves an intermediate result[45] in a conditionally saturated 24-bit destination, DOL*n*. In this particular case the quotes also prevent an erroneous external memory access of a location which has not yet been written with the desired contents.

[45]Recall that reference to the MAC latch as one of the multiplier operands (D or E), would call for the unsaturated MAC latch.

5.6. AGEN *delayspec* Syntax

```
DEFCONST
        N = 256
        RADIX = 4
        n = INT(LOG(N)/LOG(RADIX))
DEFREGION   T
        d[N]
        dp = &d[n]  @$2a3      !dp is an AOR assigned to hold address offset of d[n]
```

Given these prototype declarations we establish the following generally applicable chart of *delayspec*, where each individual column holds equivalent syntax:

--------------------------------------- AGEN Syntax ---

| |
| |

-- ***delayspec* chart** ---------------------------------

--MAC unit scope --

---ALU scope ---

----------------------AGEN scope --------------------

column: 1	2	3	4
External memory contents	External memory address offset		AOR address
(AOR indirection)	(in AOR)	(in GPR)	(in GPR)
d[n]	&d[n]	#d[n]	#&d[n]
*&d[n]	-	#*&d[n]	-
*dp	dp	-	#dp

------------------------------------ equivalences ---

```
d      =    &d[0]              (Note 1)
&d[n]  =    (&d[n])
dp     =    (dp)

*&  cancel each other          (Note 2)

[ ]    ≠    [0]                (Note 3)
```

------ optional explicit region specifier and/or Plus-One addressing mode ----------

```
d[n(+)]r                       (Note 4)
*&d[n(+)]r
*(dp(+))r              ! r  requires parenthesis
```

| |
| |

Note 1: *delayspec* chart simplification for case of root of external memory array.

```
-------------------------------------------------MAC unit scope ----------------------------------------
---------------------------------------------------ALU scope -------------------------------------------
----------------------AGEN scope --------------------
```

column: 1	2	3	4
External memory contents	External memory address offset		AOR address
(AOR indirection)	(in AOR)	(in GPR)	(in GPR)
d[0]	&d[0]	#d[0]	#&d[0]
*&d[0]	-	#*&d[0]	-
*d	d	#*d	#d

```
----------------------------------------------------------------------------------------------------
```

Note 2: The *delayspec* *(&d[n]) finds use in the AGEN instructions, RD/WR then UPDATE (discussed under UPDATE region BASE).

When * or & follow # in an operand field, the *delayspec* chart indicates the precise meaning. Therefore, the following two lines of code are identical:

```
NOP          ADDV  #10173 +    d[n], lfo > dpointer
NOP          ADDV  #10173 + *&d[n], lfo > dpointer
```

Since the *delayspec* #*&d[n] is verbose, its equivalent form #d[n] is most often used instead. Since d = &d[0] , however, the *delayspec* #*d is more efficient for determining the root of a delayline in a DEFGPR, DEFCONST, or DEFSPR.

Note 3: d[ ] is distinguished from d[0] by the assembler; these two *delayspec*s are not equivalent. Occurrences of each in the code connotes a separate Address Offset Register. d[ ] would be found in computed addressing schemes, hence the associated AOR requires no initialization. **(**The same holds for: &d[ ] , *&d[ ] **)**

Note 4: *delayspec* may optionally have an explicit region reference, *r* = I,P,Q,R,S,T,U, or V .

The Plus-One addressing mode hardware feature is discussed later.

***delayspec* Chart Interpretation**

In the **declarations**:

All columns are available for use in DEFREGION, DEFSPR, DEFGPR, and DEFCONST, subject to interpretation:

column 1) Referencing column 1 would be an error because the contents of external memory are generally not known at time of assembly; e.g.,

```
DEFGPR
        gpr1 = d[100]                           !error
        gpr2 = *dp                              !error
```

column 2) Referencing column 2 presumes the existence of a previously allocated AOR. Row 1 (in column 2) presumes existence in the same region as the *delayspec*, but row 1 can always be used in the assignment to an AOR under any DEFREGION; e.g.,

```
DEFGPR
        gpr3 = &d[100]                          !error, no AOR was allocated.
        gpr4 = &d[n]                            !ok, gets contents of AOR, dp.
        gpr5 = dp                               !ok,          ditto
DEFREGION  r
        aor1 = &d[107]                          !ok, gets address offset.
```

column 3) Referencing column 3 does not presume the existence of a previously allocated AOR. Neither is an AOR is allocated; e.g.,

```
DEFGPR
        gpr6 = #d[100]                          !ok, gets address offset of d[100].
        gpr7 = #*d                              !ok, gets address offset of d[0].
```

column 4) Referencing column 4 does indeed presume the existence of a previously allocated AOR, whose register address is desired. Row 1 (in column 4) presumes existence in the same region as the *delayspec*; e.g.,

```
DEFGPR
       gpr8   = #&d[99]                         !error, no AOR was allocated.
       gpr9   = #&d[n]                          !ok, gets register address of AOR, dp.
       gpr10 = #dp                              !ok,              ditto
```

When **&** or # are employed in the declarations, their appearance alone does not cause allocation of registers. Under DEFREGION, & has like meaning to # ; i.e., to extract the value of the external memory-array symbols following. Elsewhere, & refers to existing AORs. Refer to the *delayspec* chart for the specific meanings.

delayspec Chart Interpretation

In the **code**:

column 1) The first column indirects an AOR via the AGEN unit to acquire the content of external memory in a specified region. All rows except row 3 (in column 1) cause allocation and initialization of an AOR if not yet allocated in the associated region.

column 2) The second column is a direct reference to an existing AOR which holds an external memory address offset. But row 1 (in column 2) causes allocation and initialization of an AOR if not yet allocated in the associated region.

column 3) The third column is a direct reference to a GPR which holds an external memory address offset. All rows (in column 3) cause allocation and initialization of a GPR if not yet allocated. No pre-existing AOR is presumed, no AOR is allocated.

column 4) The fourth column is a direct reference to a GPR which holds a pre-existing AOR address. All rows (in column 4) cause allocation and initialization of a GPR if not yet allocated. No AOR is allocated.

AOR Allocation

In the code, any reference to one from the *delayspec* set: d[n], &d[n], or *&d[n], will cause allocation of the same AOR, allocation occurring only once. Once allocated, reference to the set calls out an existing AOR. But if an AOR assignment of identical initialized contents, in the same region, pre-exists in the declarations, then no new allocation will occur.
In an assignment within DEFREGION the &, as in &d[n], indicates that an address offset is desired, while in the code &d[n] would require the contents of the associated Address Offset Register. Since that AOR contains the required address offset, the meaning to the programmer is the same in both cases.

GPR Allocation

As before, # *must always appear as the first character in all syntax involving its use.*
In general, the appearance of # indicates that it is the value of the symbols following (i.e., the argument of #) that is desired. But in the code, # also allocates a GPR initialized to the extracted value. An allocation will occur if reference was not previously made in the code to the same-valued argument. If allocation has previously occurred, then # will refer to that GPR. Unlike the case for pre-existing AORs, the assembler will **not** substitute SPRs or declared GPRs of the same initialized value.

Value (extracted by #) Summary:

-The value of a number is the number itself.

-The value of a label is the same as the Program Counter (PC) value when it hits the corresponding line of code. In the declarations, an assignment to #*label* would be the same as the assignment to *label* .

-The value of a register name is its register address.

-The value of an external memory-array name is the register address of an AOR.

-The value of an indexed external memory-array is the (relative) address offset of the array at that index. For example, #d[0] is equal to the root of the external memory array d[N] .

5.7. Referencing an AOR by its Initialized Contents

We will adopt the convention that in the code &myline[118] is a direct reference to that Offset Register, in the same region as the delayline called myline, assigned to hold the relative address offset of the delayline at index 118. If &myline[118] appears in an assignment to an AOR in the declarations like so,

```
DEFREGION S
            myline[N]
            some_aor  =  &myline[118]
            relayer[1000]
            rpointer = &relayer[335]
```

then the & calls out the value of the root address offset ascribed to myline[N] by the assembler, plus 118. Since the root is a relative address offset of the beginning of an external memory array with respect to the physical start of the region, *S* in this case, it is independent of the absolute address in BASE*S*.

When &myline[118] appears in the code, it is a direct reference to some_aor. If there were another *delayspec* appearing in our code like so: myline[127], this would need a separate AOR allocated in the same region, and initialized to hold <u>its</u> address offset. That allocation would normally be performed by the assembler when myline[127] **(**or the *delayspec* &myline[127] or *&myline[127]**)** were first encountered in the code.

Hence, both the register contents <u>and</u> the associated region serve to uniquely specify an AOR. These two factors thereby make it possible to reference an AOR by its contents alone.

5.7.1. Example: Referencing an AOR by Contents

```
*(rpointer)  X  coef1 > MAC
```

It is important to note that in this code, rpointer must be an AOR because the asterisk denotes an external memory reference. But, any legal reference to an AOR could, in principle, be used. Another valid *delayspec* might be:

```
*(&relayer[335])  X  coef1 > MAC
```

Since &relayer[335] identifies the Address Offset Register called rpointer, the two lines of code above are identical.[46] It is equally important to note that the validity of this latter line of code is not dependent upon the declaration of rpointer. Had rpointer not been declared, a new AOR would have been allocated in the same region when relayer[335] **(**or &relayer[335], or *&relayer[335]**)** were first encountered in the code.

The code above is, of course, also equivalent to:

```
relayer[335] X coef1 > MAC
```

5.8. Individual Table or Delayline size

<u>The size of any delayline is always one plus its declared size</u>. This assembler convention is adopted so that *delayline*[0], acting as the delayline input buffer, can be written to at any time during one sample period of a running program. If we consider *delayline*[1] as the first sample in a delayline for the current sample period, then we need never worry that perhaps it was written by some previously executed line of code; since it will not change contents mid-period we can rely on the original contents being there. In this manner, *delayline*[0] acts as a buffer to the delayline input, while the one-sample-period latency is tolerated. Of course, this convention need not be followed. The programmer must simply be aware that the assembler is augmenting the requested delayline sizes.[47]

Since the physical address increases with increasing delayline indices, it is apparent that the programmer needs to decrement the associated region BASE*r* SPR once per sample period. This function is not automatically supplied by the hardware, nor is it desirable to be as such. When the decrement occurs, *delayline*[1] becomes the previous sample period's *delayline*[0]. **(**The AGEN's UPDATE instruction is useful for accomplishing the decrement, discussed under UPDATE region BASE.**)**

Since we do not differentiate tables and delaylines in the declarations, then declared table sizes also become augmented by 1 during assembly. This assembler convention can also be interpreted as the simultaneous support of the two most common ranges of index: 0...N-1 and 1...N . Tables should be placed in a different region by the programmer, because the BASE*r* SPR is usually fixed for table access.

[46]This differs from the assembler handling of GPRs, where constant GPRs allocated by first discovery in the code are <u>not</u> resolved with SPRs or declared GPRs of identical content.
[47]This has never been a problem.

5.9. Computed Addresses

5.9.1. AOR used in Delayline Access

The next most common *delayspec* is for a computed address into a delayline during program execution.

```
    DEFGPR              lfo   coef1   coef2   output   input
    DEFREGION  S        chorus[1000]
    CODE
          .
          .
    MOV input  >  chorus[0]                           ! MAC unit MOV
          .
```

The modulating address offset is computed and written to an (undeclared) AOR, &chorus[ ], whose name is derived from the declared delayline (to be interpolated), chorus[1000], and so is associated with the same region by default.

```
          .
    NOP          ADDV &chorus[100], lfo >  &chorus[ ]     !this is address computation
          .
```

In this case, the AOR **(**the *delayspec*, &chorus[100], obviously allocated by the assembler in this circumstance**)** which contains the address offset of chorus[100], is added to a bipolar low frequency oscillator output and then written to the AOR called &chorus[ ] . The name of the latter Offset Register could then be used in a statement to interpolate the delayline like so:

```
          .
          .
              *&chorus[ ]    X  coef1 > MAC              ! external memory requests.
    MAC  +   *&chorus[(+)]  X  coef2  >  output          ! (+) see Plus-One Addressing Mode
```

where coef1 and coef2 are derived from the fractional portion of lfo (not shown).

This C-like notation indicates that the *delayspec*, &chorus[ ], is a computed external memory address offset with respect to BASE*S*, while the AGEN indirects two accesses * to external memory via the AOR, &chorus[ ], which are scheduled by the assembler.

In this simplified example we do not show the circular movement of the region BASE in a modulo fashion, although this is covered in the section, UPDATE region BASE. See the ESP2 Applications section for real examples of interpolation.

5.9.2. External Memory Latency

Examination of the Instruction Cycle Timing diagram in the Chip Spec. reveals that external memory access lags the AGEN instruction requesting that access. This fact explains the following latencies:

In the case of an external memory read, the contents of DIL*n* first become available as sources to the MAC unit and ALU two program lines following the AGEN code.
In the case of an external memory write, the ALU must deposit its data into DOL*n* at least one program line prior to the AGEN code which sources that DOL*n*, but the MAC unit can write to DOL*n* as late as the same program line!

These latencies are illustrated in the section on **Pipeline**. Recognize that these latencies are the <u>minimum</u> latencies; i.e., if the traffic to/from external memory is light, then the automatic scheduling will achieve optimum performance. As soon as the number of memory accesses exceeds one per program line, then the scheduling may bottleneck and the minimum latencies may not be achieved. For this reason the *delayspec* Quote Scheme syntax has been provided to minimize the traffic, while the AGEN listing has been provided for the programmer to peruse post-assembly.

5.9.3. Inter-Unit Latency

The question naturally arises at this point regarding how many lines of code must exist between an external memory reference and its <u>computed</u> address, so that the reference uses the current address. In our circumstance above where an external memory read is requested by the MAC unit, the AOR, &chorus[ ], is first computed then written out of the ALU as the destination of the instruction ADDV. This Address Offset Register is available as a source to the AGEN unit <u>two</u> program lines later; this is the first time that AGEN has access to this information. Assuming no other pending external memory accesses, AGEN takes the offset information at that time and makes the access, loading the word of external memory data into some DIL*n*. But that DIL*n* does not become available as source to the MAC unit (or ALU) for another two program lines due to pipeline latencies. DIL*n* then replaces the *delayspec*, *&chorus[ ], in the program above. This makes a total of three intervening lines of code; rather, *&chorus[ ] must be referenced **(**desiring minimum latency in this case**)** no earlier than the fourth line of code after the computation of the address; viz.:

```
NOP      NOP                         WR  DOL0 > chorus[0]S      ! input
NOP      ADDV &chorus[100], lfo > &chorus[ ]            !this is address computation
NOP
NOP      NOP                         RD  chorus[ ]S  > DIL1       !access external memory
NOP      NOP                         RD  chorus[(+)]S  > DIL1
      DIL1  X  coef1 > MAC
MAC + DIL1  X  coef2 > output
!This is a partial AGEN listing of the earlier program.
```

5.9.4. Optimal AGEN Coding

If the computed AOR, &chorus[ ], were written out of the MAC unit, unlike our example above, the latency as source to the AGEN would have been <u>one program line less</u>. See the section called **Pipeline** for all minimum inter-unit latencies by example; other cases can be figured from the chart there.

There is nothing preventing the programmer from performing the address computation of &chorus[ ] <u>after</u> the external memory request. The external memory access would employ the old address in that case, but the coding efficiency gained is often well worth the added latency.

5.9.5. Computed Addresses and Table Lookup

The second case of computed addressing that we will consider is for table lookup; one or multiple tables occupying an entire region. This can be handled in nearly the same way as for the computed delayline address offset. Here we place the table index in an AOR, and add to that the table root in the multiple table case. But the absolute region start address remains unmodified in the region BASE*r* SPR.

Tables employ region memory using the same AGEN modulo addressing mechanisms as delaylines. Regions must therefore be dichotomized by the programmer as those which are utilized for tables and those which are utilized for delayline accesses; i.e., <u>tables and delaylines should not reside in the same region</u>. The reason for this is that a programmer is required to modulo decrement the BASE*r* register associated with a region of delaylines, once per sample period. Since a table lookup requires a fixed region BASE*r* reference, a region of delaylines offers a moving target.

5.9.6. Computed Addresses and Structured Table Lookup

In yet a third case of computed addressing schemes, we have many different tables in memory all residing in the same region. If we now put the <u>absolute</u> pointers (addresses) for the beginning of each table into AORs, the desired table index (the relative offset) for each lookup could then be written to the region BASE*r* register in this reversal of roles! The utility of this <u>unorthodox</u> method comes about when multiple tables have like-information at the same index; analogous to a structure-type in the C programming language.

One way to compute the absolute table pointers would be as follows:

```
DEFREGION   R    @location_zero
        base1 =  location_zero                          ! AOR declaration
        first_table[length1]
        base2 = #location_zero + first_table[length1] + 1
        second_table[length2]
        base3 = #location_zero + second_table[length2] + 1
        third_table[length3]
        !...and so on
```

The chip architecture allows other means of accomplishing the same end, but this is, of course, up to the programmer.

5.10. Defeating the Modulo Addressing Hardware

By setting the region END*r* SPR to the maximum physical address, the programmer can defeat the automatic modulo addressing, and the content of SIZEM1*r* becomes irrelevant. This setting can be accomplished via either a DEFSPR directive or as an instruction in the code. This setup of the region control registers is employed when it is desired to unequivocally address external memory as one or several tables.

It is not critical, however, to manually set the region END*r* register to implement tables; i.e., modulo addressing need not be defeated to successfully address a region as a table. It is probably more important not to decrement the region BASE in a region used for holding tables.[48]

5.10.1. Peripheral I/O

When absolute addressing is required as for I/O devices, the region BASE*r* could be set to zero while the region END*r* is set to the maximum physical address. The desired absolute address[49] could then be placed in some AOR. Again, the region BASE*r* would not be expected to undergo modification during program execution. The programmer **must** reset the MUX_ADDR bit in the HARD_CONF SPR which selects the linear addressing mode (SRAM mode); this can be done either in the declarations or dynamically in the program. This setup of the region control registers is employed when it is desired to unequivocally access external memory in absolute terms.

Once again, it is not critical to manually set the region BASE*r* and END*r* SPRs to implement absolute addressing. For example;

```
DEFREGION  I[$100000] @$D00000
        portIO[1] @0
        parallel_ADC[1] @$40000
        parallel_DAC[1] @$50000
```

These declarations establish a region of size $100000 starting at $D00000. Whenever portIO[0] is referenced in the code, the external memory address buss asserts absolute $D00000. Whenever parallel_ADC[0] is referenced, the address buss asserts $D40000 . parallel_ADC[1] asserts $D40001, and so on. This brings us back to the recommended usage of AORs as relative address offsets into a region. The advantage of this scheme is that we need not concern ourselves with the assignments of the region control registers.

[48]as one would for delaylines, discussed shortly in **UPDATE region BASE**.

[49]This would be a departure from the standard practice of using AORs to hold address offsets.

5.10.2. Region Configuration Summary

As it stands, neither delaylines, nor tables, nor peripheral I/O space have distinct declarators; only regions are declared. The subtle differences between table, delayline, and I/O references lay in how the associated regions are utilized by the programmer. It is important that however the region control registers (BASE*r*, SIZEM1*r*, END*r*) get set, the programmer always has the final say regarding their contents. So, in general, it is true that <u>any SPR explicitly declared and initialized in a DEFSPR should override any assembler determination of its initialization contents</u>. But, programmer override of region BASE, SIZEM1, and END initialization is not observed by the assembler when it is making AOR assignments.

5.11. Plus-One Addressing Mode

The AGEN supports a special addressing mode in which 1 is added to the address offset, obtained from a specified AOR, <u>before</u> it is used in an external memory access. This addressing mode is useful for interpolation. (See the ESP2 Applications section.)

```
           *choruspointer  X coef1 > MAC
    MAC  - *choruspointer(+)  X coef2 > output
```

The second program line uses the syntax (+) to mean:

address offset, in the AOR choruspointer, plus 1 .

This syntax is **not** a C-style operator because <u>neither the region BASE*r* nor the AOR are permanently modified using this addressing mode</u>.

Plus-One addressing may also be used with an array-type *delayspec* as in:

myline[118(+)]

5.12. Forced WR

Normally, the assembler schedules all the external memory-reads first. Once this is done, the closest following available AGEN instruction slots are used to schedule whatever external memory-writes were requested by the program. Under some circumstances, however, it is desirable for the programmer to insure that particular writes take place with minimum holdoff. For this purpose the destination syntax

=> *delayspec*

is used in place of > *delayspec* for AGEN coding in the MAC unit or ALU instruction fields.

5.13. UPDATE region BASE

The AGEN has one last feature which enables the programmer to optionally write an address offset plus region BASE*r* back into the BASE*r* register after an external memory access (RD or WR). This allows for modulo incrementing or decrementing the region BASE*r* by specified amounts. Using the pointer-type expression, *dp, in the AGEN instruction source operand field, we have the new *delayspec*:

```
...    RD  *(BASE  +=  tablejump)  >  DILE            !UPDATE region BASE after read
```

This AGEN instruction says that the external memory to which the AOR, tablejump, points, is read and placed in the Eth DIL. The absolute external memory address is, as always, the modulo sum of the region BASE*r* and the specified AOR contents.[50] In this addressing mode, however, the modulo sum is deposited back into BASE*r* after the external memory access; i.e., the region BASE*r* register is post-incremented by the contents of tablejump, thus the contents of BASE*r* is updated.

The increment may be chosen effectively negative. For example, to decrement the region BASE*r* by 1 in a modulo fashion, the contents of the AOR tablejump must be set to the contents of the region SIZEM1*r* SPR. To decrement by 2, the contents of tablejump should be set to (SIZEM1*r* - 1), and so on.
For example:

```
DEFSPR  DIL0                          ! reserve DIL0
DEFREGION  S  @0
    diffusion_line[477]
    wait_line[412000]
    tablejump = SIZEM1S               ! declaration must be at end of DEFREGION

CODE
NOP
NOP
NOP          NOP                         UPDATE  BASE += tablejump          ! pure form
//NOP        NOP                         RD  *(BASE += tablejump) > DIL0
//NOP        MOV  *(BASE += tablejump) > DIL0
//NOP        NOP                          RD  *(BASE  +=  &wait_line[412000]) > DIL0
```

The UPDATE of BASE*S* is identical using any one of these lines of code, but the commented lines (//) would each perform a read from external memory too. Notice how an UPDATE is worked into the AGEN coding in the ALU instruction field. That *delayspec* syntax is available in the MAC unit instruction field too. The *pure form* of the AGEN UPDATE instruction, having no external memory access, is shown in the example program. This pure UPDATE form is allowed only in the AGEN instruction field.

[50]the modulus automatically determined by the size of the region, the particular region connoted by the specific AOR.

The *delayspec* from the example,

*(BASE += &wait_line[412000])

is interesting because the * operates on the entire contents of the parenthesis. This makes sense because it comes from the established syntax, *&d[n] = d[n] . In our example, &wait_line[412000] calls out an AOR which when added to BASE yields an absolute external memory address. The * operator then requests the contents at that address. Access of external memory is always accomplished by modulo summing an AOR with a region BASE register. In this syntax, we explicitly show the sum to engage the AGEN post-UPDATE mode.

5.13.1. Use of UPDATE

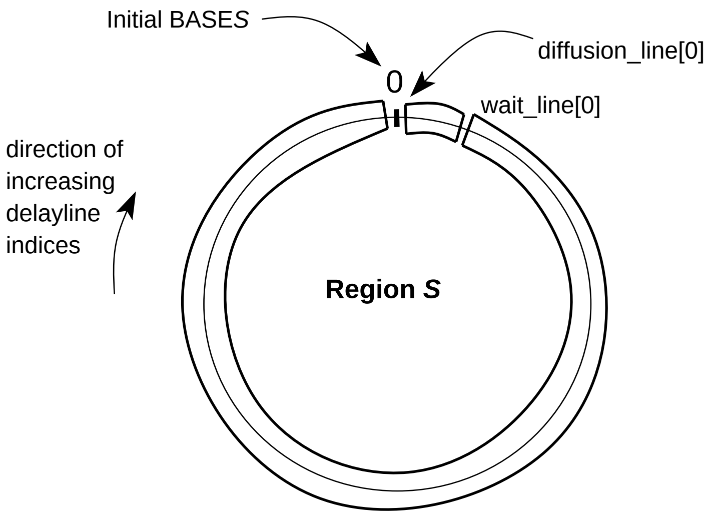


Figure O. Region *S* at initialization. (Delaylines not drawn to scale.)

At download time, BASE*S* identifies the start of the region. The two declared delaylines are of fixed size. We need to feed the delayline inputs **(**[0]**)** with a new sample while we discard the oldest sample. The discard is primarily achieved by decrementing the region BASE under program control. The discard is the last thing done in a sample synchronous loop, so an UPDATE instruction will typically be found at the end of the program main loop. To decrement the region BASE by 1 we <u>add</u> the size of the region less 1. Referring to Figure O, it is easy to see that adding the contents of SIZEM1 (as explained above) to the initial region BASE, moves it one place to the left in the modulus, the modulus being the region size. The automatic modulo arithmetic in the AGEN keeps all subsequent decrements of BASE*S* within the modulus.

Somewhere in the body of the program should be found writes to the delayline inputs. Because the start (input, root) of each delayline can be interpreted as a relative address offset with respect to the region BASE, all the delaylines will have shifted one place left in the modulo memory, each following the movement of the region BASE. All delayline address arithmetic in the AGEN is subject to the same modulo math. When each delayline input is written, it will write over the contents of the oldest sample of the preceding delayline in this modulo queue.

5.13.2. Explicit Region

The particular region BASE*r* is connoted through the declaration of the AOR called tablejump in the earlier example, but the region can be explicitly indicated if desired. Should you want to use the same offset into a different region, then that new region may be specified using I, P, Q, R, S, T, U, or V in place of *r* :

```
...    UPDATE   BASEr += tablejump              !UPDATE region BASEr
...    UPDATE   BASE += (tablejump)r            ! ditto
...    UPDATE   (BASE += tablejump)r            ! ditto
```

BASE*r* means one from: BASEI, BASEP, BASEQ, BASER, BASES, BASET, BASEU, BASEV.

The hardware does **not** support the simultaneous increment of BASE in one region while an external memory access is made from another. So for the sake of clarity, **one should write only one explicit region specifier**.

5.14. Region Override

In general, any *delayspec* may include an optional region specifier. In some circumstances, it may be advantageous to override the default region which is implicit to each *delayspec.* For example, the identical code statements,

```
        DEFGPR   coef1

        CODE
          *&wait_line[ ]S   X  coef1 > MAC
```

or

```
             wait_line[ ]S   X  coef1 > MAC
```

are each redundant in the specification of the region *S* in light of the declarations in the example above. But if another region were specified instead of region *S*, they would remain valid *delayspec*s because this would then be a directive to use the same offset into a different region of external memory. That ’different region’ need not have been declared.

5.14.1. List of AGEN Instructions and AGEN Syntax

delayoffset is a declared Address Offset Register (AOR). &*delayline*[*n*] refers to the same Offset Register. Several variations of the same instruction are shown.

TABLE AGENLIST. AGEN unit Instructions.

Instruction	Operation
NOP	***the only fundamental nop***
RD *delayline*[*n*]*r* > DIL*i*	*read*
RD *&*delayline*[*n*]*r* > DIL*i*	*read*
RD *(delayoffset)*r* > DIL*i*	*read*
RD *delayline*[*n*(+)]*r* > DIL*i*	*read Plus-One*
RD *&*delayline*[*n*(+)]*r* > DIL*i*	*read Plus-One*
RD *(delayoffset(+))*r* > DIL*i*	*read Plus-One*
RD *(BASE += &*delayline*[*n*])*r* > DIL*i*	*read then UPDATE*
RD *(BASE += delayoffset)*r* > DIL*i*	*read then UPDATE*
WR DOL*i* > *delayline*[*n*]*r*	*write*
WR DOL*i* > *&*delayline*[*n*]*r*	*write*
WR DOL*i* > *(delayoffset)*r*	*write*
WR DOL*i* > *delayline*[*n*(+)]*r*	*write Plus-One*
WR DOL*i* > *&*delayline*[*n*(+)]*r*	*write Plus-One*
WR DOL*i* > *(delayoffset(+))*r*	*write Plus-One*
WR DOL*i* > *(BASE += &*delayline*[*n*])*r*	*write then UPDATE*
WR DOL*i* > *(BASE += delayoffset)*r*	*write then UPDATE*
UPDATE BASE += &*delayline*[*n*]*r*	*UPDATE, no memory access*
UPDATE BASE += (delayoffset)*r*	*UPDATE, no memory access*

6. Pipeline

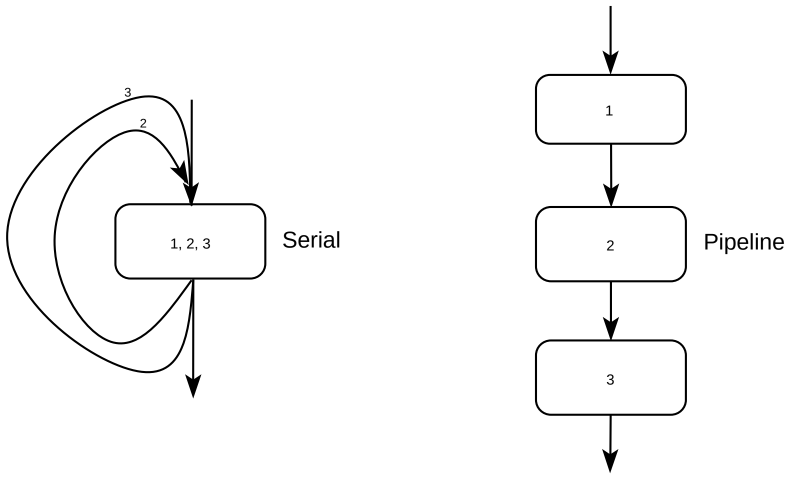


Figure Pipe. Serial vs. Pipeline architecture.

We have not yet touted the benefits of a pipeline architecture, which is ESP2. We claim that the pipeline affords an internal computational efficiency; the disadvantage is that the programmer is required to be cognizant of various types of small latencies which are listed and explained thoroughly in the Chip Spec.

We give an example of the same computation within two fictitious architectures requiring M=3 different operations to complete, shown diagrammatically in Figure Pipe. We wish to perform this computation on each of a sequence of, say, N=1024 numbers. We assume that the operation cycle time is the same for every functional block in Figure Pipe. In the serial architecture the computation time is M*N = 3072 cycles, but in the pipeline architecture the computation time is only N + (M-1) = 1026 having an execution latency=(M-1)=2.[51] That is, the pipeline architecture is nearly 3 times faster than the equivalent serial architecture. This achievement comes at a cost of 3 times the hardware in this example, however.

This pipeline idea has been exploited in a number of commercial DSP architectures. [Dattorro2400] One notable example is the WE DSP32 from AT&T. There the pipeline is quite deep and non-orthogonal across the various categories of latency, making the programmer's job quite difficult. This problem is alleviated for the programmer somewhat by adding a layer of abstraction in the form of a C-language compiler whose output is translated into DSP32 code. The compiler output is often full of inefficiency in the form of DSP32 NOPs to satisfy the various latencies.

Clearly, there is some tradeoff which balances computational efficiency with programmability. We feel that we have found a good middle ground in the parallel/pipeline architecture of ESP2.

[51]The latency in the pipeline causes us to refine our thinking of the 'next operation' to the 'next **queued** operation'.

6.1. Minimum Inter-Unit Destination-to-Source Register Latency (through example)

REGISTER to REGISTER

MAC unit to ALU

```
gpr1 X gpr2 > result          NOP
NOP                           ADD  result, gpr3 > gpr4
```

ALU to MAC unit

```
NOP                           ADD  gpr1, gpr2 > result
NOP                           NOP
result X gpr3 > gpr4                            ! result first becomes available as MAC source.
```

MAC unit to AGEN (address offset or region control registers)

```
gpr1 X gpr2 > my_offset       NOP                 NOP
NOP                           NOP                 RD  *my_offset > DIL3
```

ALU to AGEN (address offset or region control registers)

```
NOP              ADDV  gpr1, gpr2 > my_offset     NOP
NOP              NOP                              NOP
NOP              NOP                              RD  *my_offset > DIL3
```

AGEN (UPDATE region BASE*r*) to MAC unit

```
NOP                           NOP                 RD  *(BASE += my_offset)r  > DIL3
-BASEr  X  #$800000 > MACP    NOP
```

AGEN (UPDATE region BASE*r*) to ALU

```
NOP                           NOP                 RD  *(BASE += my_offset)r  > DIL3
NOP                           ADDV  SIZEM1r, BASEr
```

EXTERNAL MEMORY to/from DATA LATCH

MAC unit to DOL

```
reg1 X reg2 > DOL4            NOP                 WR  DOL4 > *&delayline[200]
```

ALU to DOL

```
NOP                       ADD  reg1, reg2 > DOL4  NOP
NOP                       NOP                     WR  DOL4 > *&delayline[200]
```

MAC unit from DIL

```
NOP                           NOP                 RD  *&delayline[200] > DIL3
NOP                           NOP                 NOP
DIL3 X reg4 > reg4            NOP
```

ALU from DIL

```
NOP                           NOP                 RD  *&delayline[200] > DIL3
NOP                           NOP                 NOP
NOP                           ADD  DIL3, reg4 > reg4
```

7. Indirection

The chip hardware provides an indirection mechanism for accessing internal registers. Recall that the ALU employs the operands named A, B, and C, while the MAC unit employs D, E, and F, and the AGEN unit has the G operand (which is an AOR). The ALU MOV instruction, for example, moves the source operand B to the destination operand C, while the MAC unit moves D to F.

```
NOP                          MOV  B > C
MOV  D > F
```

As is always the case for indirection in the ALU, the INDIRA and INDIRB *pointer* SPRs hold the register address of the A and B source operands, respectively, while the INDIRC pointer SPR holds the register address of the C destination operand. If we want to indirect on the C operand, for example, we must first place the desired destination address into the pointer register, INDIRC. We might accomplish this as follows:

```
DEFGPR          gpr    source=17    dest
DEFREGION  R    aor
CODE
MOV  #gpr > INDIRC                                    !move register address
NOP                     MOV  source > INDIRECT
```

When the ALU sees the INDIRECT SPR in the C operand field, it substitutes the contents of INDIRC for the C operand address. So in this example, the contents of source (=17) are moved to the GPR called gpr, because INDIRC holds the register address of gpr.

We point out a potential pitfall here which arises in the coding of abbreviated instructions employing implied destinations. This occurs, for example, when the ALU construct,

```
                    OPERATION  A, INDIRECT              ! not recommended
```

is used in place of the verbose construct,

```
                    OPERATION  A, INDIRECT > INDIRECT
```

The first construct implies that the destination register address, inside INDIRC, is the same as the second source register address, inside INDIRB. Since INDIRB cannot hold the destination address of any instruction, then the first construct can only work as implied if the programmer has preloaded both INDIRB and INDIRC with the same address.

So, for indirection to work properly with no caveat, only the following constructs should be used in the ALU:

```
OPERATION  INDIRECT, B
OPERATION  INDIRECT, B > C
OPERATION  A, INDIRECT > INDIRECT
OPERATION  INDIRECT, INDIRECT > INDIRECT
```

It is much easier to remember that for indirection to work as intended, use the verbose ALU construct:

OPERATION   A, B > C

substituting INDIRECT where desired. An exception occurs for the use of indirection in conjunction with the ALU's ASDH, ASDL, LSDH, and LSDL double precision shift instructions. Consult the Chip Spec.

Of course, the INDIRINC and INDIRDEC SPRs may be substituted for any INDIRECT SPR. The use of these two SPRs as operand automatically post-increment/decrements the corresponding pointer register.

The MAC unit has its own pointer registers corresponding to its D, E, and F operands: INDIRD, INDIRE, and INDIRF. The AGEN unit has INDIRG. In general, there is less abbreviation allowed by the assembler in the MAC unit than in the ALU, and there is no abbreviation in the AGEN. So, the programmer is less likely to run into trouble coding indirection in the MAC unit or AGEN.[52]

Pseudo instructions are plentiful in both the ALU and MAC unit, so the programmer must consult the Chip Spec. to determine how operands get mapped for each individual pseudo. In some cases, the programmer may need to resort to the instruction primitives to get the desired result.

To reiterate the general rule: *always use verbose constructs when coding indirection.*

[52]Recall that the E operand in the MAC unit cannot be an AOR; likewise INDIRE cannot point to an AOR.

7.1. Indirection, Minimum Latencies (through example)

MAC unit to MAC unit source

```
MOV  #gpr > INDIRD                              ! same latency for INDIRE
NOP
MOV  INDIRECT > dest
```

MAC unit to MAC unit destination

```
MOV  #gpr > INDIRF
NOP
MOV  source > INDIRECT
```

ALU to ALU source

```
NOP        MOV  #gpr > INDIRB                   ! same latency for INDIRA
NOP
NOP        MOV  INDIRECT > dest
```

ALU to ALU destination

```
NOP        MOV  #gpr > INDIRC
NOP
NOP        MOV  source > INDIRECT
```

MAC unit to ALU source or destination

```
MOV  #gpr > INDIRB                              ! same latency for INDIRA or INDIRC
NOP        MOV  INDIRECT > dest
```

ALU to MAC unit source or destination

```
NOP        MOV  #gpr > INDIRF                   ! same latency for INDIRD or INDIRE
NOP
MOV  source > INDIRECT
```

MAC unit to AGEN G operand (RD or WR)

```
MOV  #aor > INDIRG
NOP
NOP        NOP        RD  *(INDIRECT)r  > DILn
```

ALU to AGEN G operand (RD or WR)

```
NOP        MOV  #aor > INDIRG
NOP
NOP        NOP        RD  *(INDIRECT)r  > DILn
```

ESP2

Ensoniq Signal Processor 2

Part III

Fundamental Audio Applications[53]

Jon Dattorro

[53] © 1995, Jon Dattorro

1. Linear and Allpass Interpolation
for Application to Chorus, Flanging, and Vibrato Effects

1.1. Audio Applications of Interpolation

In this section, we present the topic of delayline interpolation from the more intuitive point of view of the required fractional sample delay; i.e., a time domain viewpoint. The classical derivation, called **sample rate conversion**, [Schafer/Rabiner] [Vaidyanathan] is more a frequency domain formulation. In a musical context, sample rate conversion ratio inverse (M/L in Appendix IX) corresponds to Pitch Change or Shift ratio. We synopsize the classical results in Appendix IX where will be found a schematic translation of the fundamental algorithms we discuss to the more traditional DSP nomenclature. That should serve to bridge the two viewpoints.

The technique of delayline interpolation is employed when it is desired to delay a signal by some number of samples expressible as a whole plus some fractional part of a sample. This way, the delay is not restricted to sample boundaries hence avoiding signal discontinuities when the delay time is swept. Delayline interpolation is indigenous to Pitch Change and Pitch Shift[54] algorithms which are themselves integral to numerous other effects; e.g., Doppler, and Leslie Rotating Speaker emulation. Delay *modulation* forms the basis of the Chorus effect and its close relative, the Flanger.[55] Delay modulation alone (with no mix) yields Vibrato when the modulation is sinusoidal. Use of delay modulation typically entails a nominal signal delay because the modulation spans some desired range.

The interpolation methods we seek are computationally simple and inexpensive by necessity. The interpolation algorithm may be executed many times in one sample-synchronous audio processing program. Playing a subsidiary role, we typically cannot afford interpolation routines that consume a large percentage of the allotted execution time.

1.2. !*** Linear interpolation ***

We show the kernel of the ESP2 code[56] for Linear interpolation from Appendix XI. The principles discussed in this section are applied therein.[57]

```
!********************* Linear interpolation ***************************
MOV nominal_delay > MACP
MACP + y_q n X chorus_width > &VoiceL[ ]                          !address, integer part
          frac  X  *&VoiceL[(+)] > MAC        LSH  MACRL >>1  > frac      !fractional part (positive)
MAC + frac_u  X  *&VoiceL[ ] > vibrato      DIFF  frac > frac_u
!*************************************************************
```

[54]Pitch Change and Pitch Shift are distinguished later on.

[55]These two effects come about when the output signal is made to be a linear sum (mix) of the original (dry) input and the dynamically delayed (wet) input signal. The Chorus and Flanger are distinguished primarily by the minimum delay in their delay range. (The minimum is less for the Flanger.)

[56]ESP2 is the second-generation Digital Signal Processing chip for audio from Ensoniq Corp., Malvern, Pennsylvania, USA. Finished in 1995, it was designed to supersede the Motorola 56000 series.

[57]The complete ESP2 program in Appendix XI is fully tested and in production.

The process called Vibrato[58] is shown diagrammatically in Figure L. The local index *m* is defined as the current computed relative index into our delayline with respect to its root. The local index, *m* , requires computation because we want it modulated by the LFO (**low frequency oscillator**), $y_q n$, oscillating as a function of time, n. The range of *m* , ±CHORUS_WIDTH, is centered about the nominal tap point into the delayline, NOMINAL_DELAY.

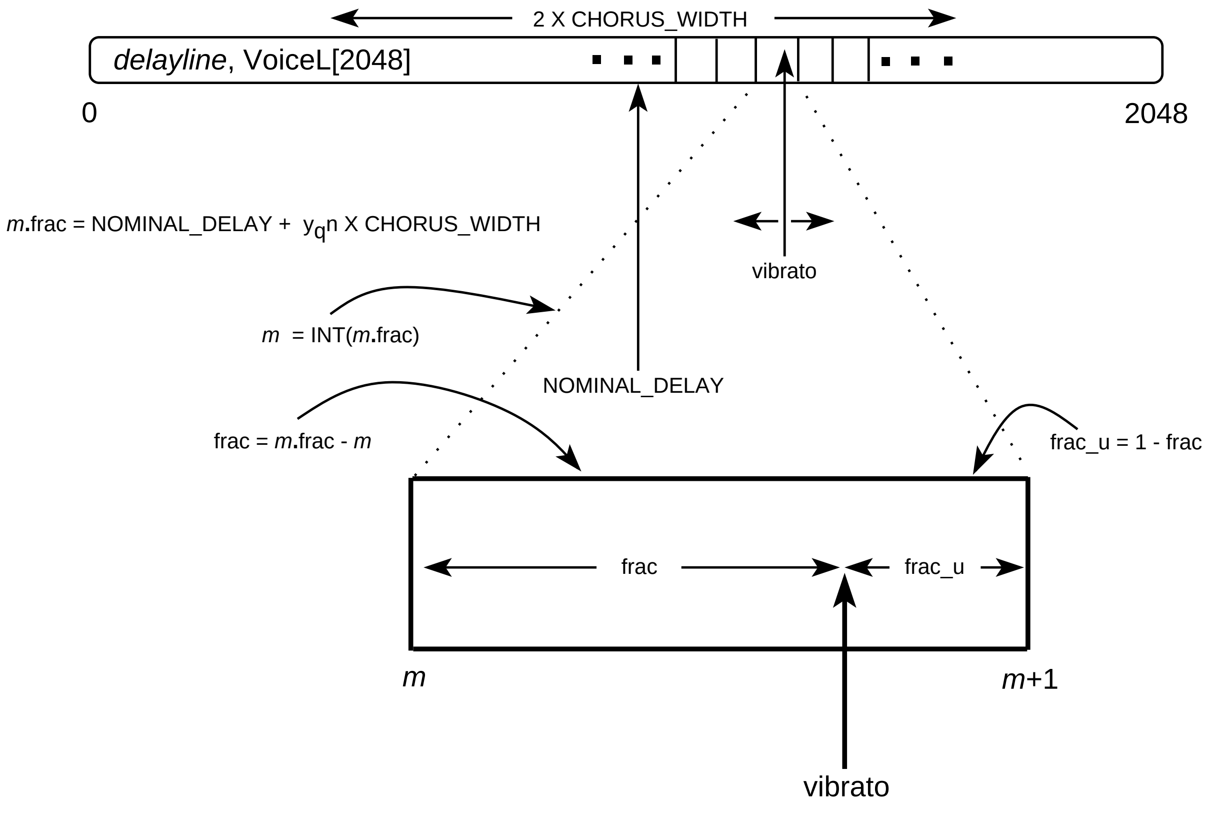


Figure L. vibrato = frac X VoiceL[*m*+1] + (1 - frac) X VoiceL[*m*]
This is the desired analytical result, not ESP2 syntax.

Note from Figure L that because the time-varying fixed-point expression ***m*.**frac is always a positive index into the delayline, then the coefficient, frac, is always positive. The time-varying positive coefficients, frac and frac_u, are 23-bit resolution because chorus_width **(**assigned as CHORUS_WIDTH in the declarations**)** is scaled by the oscillator, $y_q n$ (q for quadrature), which is in **q**23 format.[59]

58In the context of audio effect design, Vibrato is most often defined to be undulating pitch <u>change</u> although it can certainly be accomplished by other means.
59See Scientific and Binary-Radix Notation.

In the program, the role of m is fulfilled by the AOR, &VoiceL[ ]. The brackets of &VoiceL[ ] are empty, in the program, to highlight the fact that the index into the delayline, VoiceL[2048], is being computed by the program. Unlike m, &VoiceL[ ] must be computed with respect to the region BASE; i.e., the root address offset of VoiceL[2048] must be taken into account.

$$\&\text{VoiceL[ ]} = m + root$$

where $root$ = &VoiceL[0]. The (non-fractional) root is extracted for us by the assembler in the declaration of nominal_delay, which is apparently used in the programmed computation of &VoiceL[ ]. That declaration of nominal_delay accounts for the fact that VoiceL[2048] may not be the first or only delayline existing in the region (although in our example it is). Declared as such, we handle the general case.

1.3. Linear Interpolation as a Polyphase Digital Filter

We are probably not accustomed to think of Linear interpolation as a filtering process. This is because it is such an intuitively simple algorithm in the time domain performing variable, fractional sample delay. Yet Linear interpolation is used routinely to perform sample rate conversion in contemporary sampler-type music synthesizers,[60] and we know that there is a vast amount of literature on the subject which poses the conversion problem in the frequency domain.

Linear interpolation as applied in the Chorus, Flanger, and Vibrato effects, makes small undulating changes in pitch. If we listen closely to these effects, we will also discover a significant perceived loss of high frequency content not attributable to comb filtering.[61] It is this observation that compels a frequency domain analysis.

[60]There, it is most often manifest as a constant Pitch Change at playback dependent upon the key struck. (We distinguish Pitch Change from Pitch 'Shift'. [Dattorro2400]) The technique used to accomplish Pitch Change by fixed amounts is called the **phase-accumulating oscillator.** [Moore,ch.3]
The Ensoniq *Mirage*, introduced back in 1984, employed **zeroth-order interpolation** (choosing the nearest sample in time) in conjunction with 8 phase-accumulating oscillators. The *Mirage*, one of the earliest sampler-type music synthesizers, is still in commercial use because of its characteristic sound quality due primarily to its digital encoding scheme. Its hybrid floating-point design was based on an 8-bit mantissa and an 8-bit exponent. Recorded sound samples employed only the mantissa. Subsequent enveloping applied the 8-bit exponent to the analog reference on a D/A converter. That way, the signal to noise ratio of the original sample was not compromised when the signal was dynamically scaled. -Robert Yannes
[61]When a signal is added to a delayed replica of itself, comb filtering results.

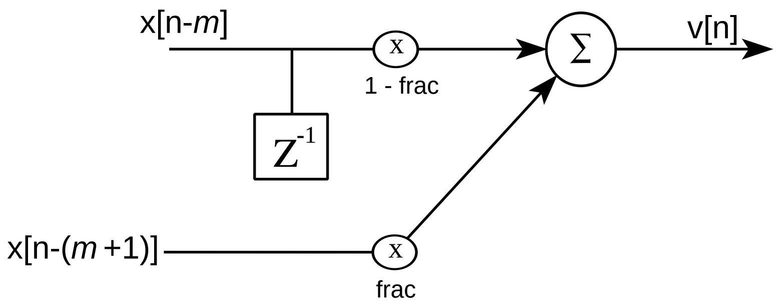


Figure PL. Linear interpolation circuit.

In Figure PL we show the actual two-input linear time-varying digital circuit (in DSP format) which implements standard Linear interpolation. We have drawn it in a strange way to emphasize the non-sequential access of the required input samples. The time index, n, steps through discrete time in a sequential fashion; $0 \le n < \infty$. The index, n, always refers to the current sample.

In the circuit, the z^{-1} (unit delay) is not terminated because the local index, *m* , is time-varying **(**it changes at each time step, n**)**, which is to say that it can take on any value within bounds.[62] From the declarations in our program we have implicitly established the parameters: $0 \le m < 2048$. The consequence of these considerations is that x[n-1-*m*] is not necessarily the *old* value of x[n-*m*].

Ideally what we want is for v[n] to approximate the value of the continuous signal x(t) at points in time between sample instants; i.e., we would like

$$v[nT] \approx x((n - m\mathbf{.}\mathrm{frac})T) \qquad \text{(lipdf)}$$

where T is the sample period. The Linear interpolation circuit makes this approximation (lipdf), but it is a time-varying circuit[63] because its coefficients are a function of *m***.**frac as outlined in Figure L. The circuit of Figure PL is **polyphase** because whenever an output sample v[n] is computed, a new pair of coefficients is fetched yielding a different phase response from one of the filters in an ordinal set.[64] [Vaidyanathan,pg.166] Notice that when frac=0, which is not unusual, the circuit performs no filtering action; i.e., allpass.

[62]Strictly speaking, *m* is a function of n; i.e., *m*(n). Its non-sequential nature demands random access of external memory.
[63]It is possible for a polyphase network to be time-**in**variant even when the constituting circuits are time-varying. This happens when the output signal is a replica of the input signal to within a constant and/or a delay term. [Vaidyanathan]
[64]The number of possible coefficients is related to their resolution; there are $L=2^{23}$ possible pairs of coefficients, hence that many polyphase filters. [Crochiere/Rabiner,ch.4.3.11]

To make the connection from the circuit to our program, we make the analytical identification:

$x[n-m]$ = VoiceL[m] !**not** ESP2 syntax[65]

This identification locates the requested sample in our delayline. It is clear that x[n] always refers to VoiceL[0] as this is the current sample. This is true because BASEV is decremented in the program, and positive *m* indexes older samples in our delayline.

1.3.1. High-Frequency Loss of the Linear Interpolator

To facilitate the investigation, we will approximate the actual circuit used (shown in Figure PL) with the polyphase filter circuit in Figure PL2. We can do this because the two inputs in Figure PL are separated by one sample in the delayline. We need not assume constant *m* for the analysis because we will view the **instantaneous transfer function** of this non-recursive network,[66] represented by Figure PL2.

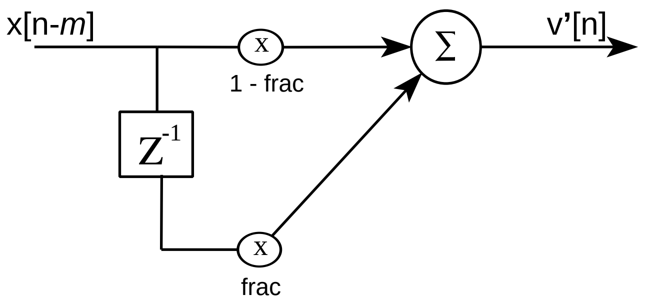


Figure PL2. Linear interpolation analytical circuit approximation.

Because of the constant interrelationship of the time-varying coefficients, the circuit approximation has an instantaneous frequency response that traverses a domain having an allpass transfer function **[**1**,** z^{-1}**)** at either extreme of the coefficients, to an averaging transfer $(1 + z^{-1}) /2$ in the middle. This large set of transfers are the frequency responses of the polyphase filters. In Figure PLS we show several of these transfers, corresponding to different values of frac.

[65]*Delayspec*s in the form of external memory array references expect constant expressions within the brackets.

[66]We freeze time and then determine the transfer function at that moment. This analytical device is justifiable if *m* changes slowly in time. The circuit approximation in Figure PL2 is only used for analysis; it is not used in the actual implementation.

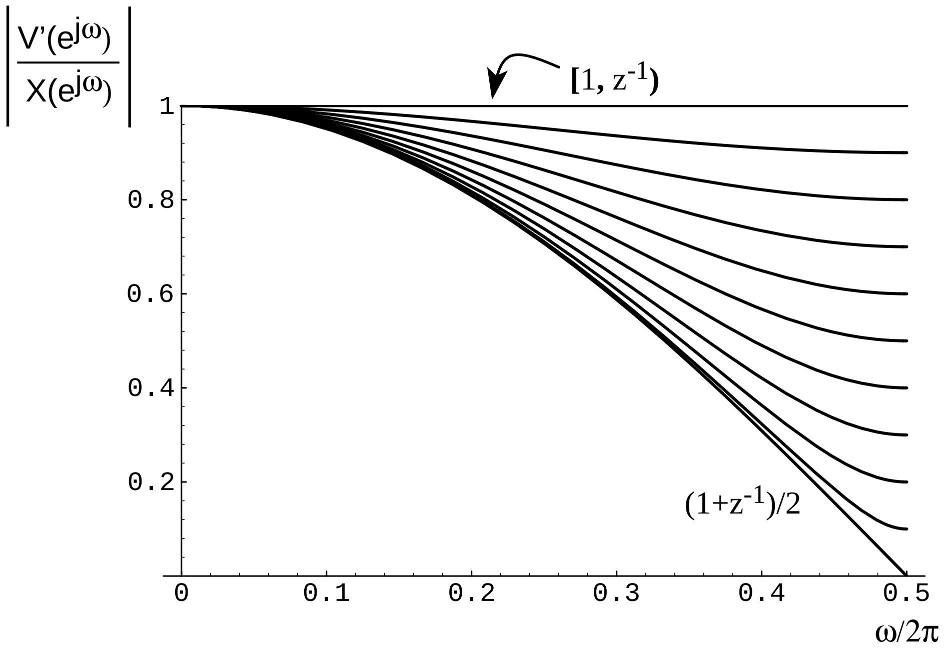


Figure PLS. Spectra of Linear interpolation polyphase filters.

It is this set of transfer functions in Figure PLS that we hear in our effects.[67] In some cases, flutter is clearly audible and objectionable. In other cases, a veil seems to have been placed over the sound source.

We could eliminate this artifact if the polyphase filters were allpass. So we need to know if it is possible to make the individual polyphase filters have allpass transfers while still performing interpolation. The answer is in the affirmative [Vaidyanathan,pg.166] [Renfors/Saramäki] and we explore this in the section, Allpass Interpolation.[68]

[67]Each curve below the single allpass is duplicated since there are two sets of coefficients corresponding to each curve in the case of Linear interpolation (for example, $(0.25 + 0.75\, z^{-1})$ and $(0.75 + 0.25\, z^{-1})$).

[68]The reason that the polyphase filters can have allpass transfer is because:

> The formal frequency domain formulation of interpolation derives the polyphase filters from what is called the **prototype interpolation filter**. The prototype filter is simply a successively delayed sum of the L polyphase filters
>
> $$H(z) = \sum_l e^{-j\omega l}\, E_l(e^{j\omega L})$$
>
> constituting the complete set. [Vaidyanathan,ch.4.3] The prototype filter response must not be allpass, by definition, but no such restriction is placed upon the individual polyphase filters! In fact, for the idealized formulation of interpolation by a rational factor, each polyphase filter is exactly allpass; (See Appendix IX for interpretation.)
>
> $$E_l(e^{j\omega}) = e^{j((\omega-2\pi m)l/L)}$$
>
> where l is the polyphase filter number, L is the upsampling rate, m is the prototype replication number or the frequency-band number of an Lth-band (Nyquist(L)) prototype. [ibid.,pg.168,109,124] [Crochiere/Rabiner,ch.4.2.2]

When we sum all the successively delayed non-allpass polyphase filters of Linear interpolation we

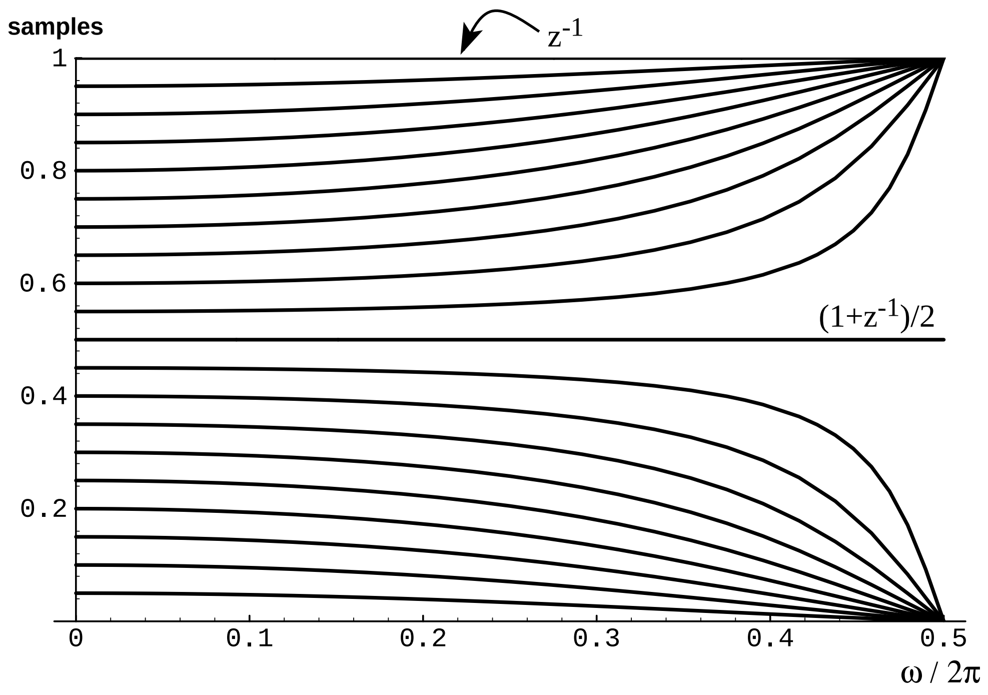


Figure PLD. Delays of Linear interpolation polyphase filters.
The extreme polyphase filter z^{-1} is not used.

Figure PLD shows the delay transfer as a function of frequency introduced by the polyphase filters corresponding to Figure PLS. The fractional sample delay is figured as the negative of the phase response divided by the radian frequency; i.e., the **phase delay**. This signal delay is with reference to a steady state sinusoid of infinite duration.

The delay transfers of the polyphase filters are as important as their frequency transfers. When the coefficient, frac, calls for a quarter-sample delay, for example, we would like a constant quarter-sample delay for all frequencies. The only delay transfer of Linear interpolation that perfectly meets this criterion is the one in the middle, $(1 + z^{-1}) /2$; i.e., it is the only one, aside from the trivial case of frac=0, that we will see having **linear phase**, hence a constant delay (half-sample, this case). It is observed from Figure PLS and Figure PLD, that when the filtering is at its worst the delay is perfect.

Notice that the delay for each individual polyphase filter in Figure PLD is fairly constant in the low frequency region, however. If we regard only DC, we make the further observation that the delay perfectly tracks the polyphase filter coefficient, frac, as desired. This property is a major determinant in the performance of the Linear interpolator from the standpoint of ***THD+N*** of low frequency signals.

get the transform of a sampled triangular finite impulse response, which is what we expect from the prototype. [Crochiere/Rabiner] [Opp/Sch,pg.109]

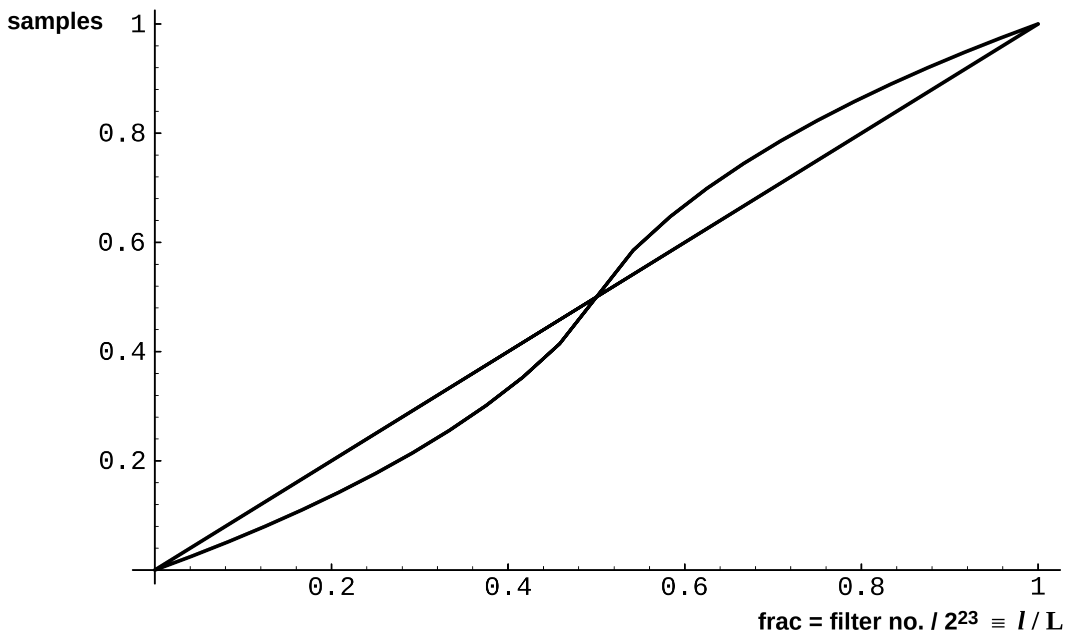


Figure PLA. Average Delay of each Linear interpolation polyphase filter.

Figure PLA shows the average delay (over frequency) of each individual polyphase filter for the Linear interpolator. There are $L=2^{23}$ different polyphase filters because that is how many different non-negative coefficients there are in 24-bit two's complement, **q**23 format. [Crochiere/Rabiner,ch.4.3.11] The straight line is the desired average.

1.4. Allpass Interpolation

The implementation cost vs. performance of Linear interpolation is difficult to beat, especially as the nominal sample rate is increased. It does have significant drawbacks, however, which motivates us to look for better methods.

The drawbacks to the use of Linear interpolation are:
1**)** it is a lowpass process having a dynamic zero at the Nyquist frequency creating muffled sounds and unaccounted damping in signal paths employing this process,
2**)** the dynamic zero of its polyphase digital filters introduces audible flutter (amplitude modulation) which is quite objectionable near unity Pitch Change Ratio for sounds having much high frequency content,
3**)** an exceptional amount of aliasing occurs and is worst for large pitch change upwards, corresponding to the case of sample rate reduction (**decimation**).[69]

Other companies [Rossum] report alternative interpolation strategies.[70] The author recommends Lagrange interpolation [Crochiere/Rabiner] [Schafer/Rabiner] [Ramstad] [Laakso/Välimäki] which is an analytical extension to Linear interpolation.[71] But non-recursive **FIR** (finite impulse response) techniques such as these have a high computational cost. Nonetheless, FIR filters dominate contemporary sample rate conversion practice because from the point of view of internal truncation noise, it is difficult to mess up an FIR implementation. [Dattorro89] [DattorroPat.] [Andreas] Also, FIR filters offer linear phase.[72]

[69]This last problem comes about because the prototype Linear interpolation filter has (one-sided) bandwidth π/L in the Vaidyanathan sense; the same is true for Allpass interpolation. The bandwidth is not a design parameter, but falls out as a result of the implementation. When the decimation rate M exceeds the interpolation rate L, aliasing is traditionally tolerated for these two techniques. When our musical applications of interpolation operate only over a microtonal conversion ratio, then aliasing is less of an issue.

[70]*The E-mu* Proteus *sampling music synthesizer (1989) and its relatives all employ 7th-order interpolation polynomials. They aren't exactly 'Lagrange' though. They use a technique in which a Remez Exchange is applied to an 'ideal' filter response similar to that of Lagrange, but having lower maxima in the stopband. This gives the deep notches advantageous in the Lagrange approach, but also the superior stopband rejection of a sinc-based design. For more information, see the US patent on the fundamental E-mu G-chip interpolator; No.5,111,727.* -David Rossum

[71]i.e., a higher-order polynomial curve fit using more signal values, and which is maximally flat in the frequency domain while suppressing ripple in the time domain. The two-point Lagrange interpolator is equivalent to Linear interpolation.

[72]Generally speaking, a linear phase prototype interpolation filter does not guarantee linear phase polyphase filters, and vice versa. Linear interpolation, for example, is a classical case of an FIR prototype that is linear phase **(**having a symmetrical triangular impulse response of length 2L spanning two original samples by design [Opp/Sch,pg.109] [Crochiere/Rabiner]**)** but whose polyphase filters are not. But if a linear phase prototype is perfectly bandlimited to π/L, for L a rate conversion factor, then all its L polyphase filters will also be linear phase; ideally of the form $E_l(e^{j\omega}) = e^{j((\omega-2\pi m)l/L)}$ where m is the frequency-band number of an Lth-band **(**Nyquist(L)**)** prototype. [Adams,pg.545] [Vaidyanathan,pg.168] [Opp/Sch,ch.5.7] (See Appendix IX for interpretation.) Crochiere [Crochiere/Rabiner,ch.4.3.6-4.3.10] gives explicit general design procedures for simultaneously linear phase FIR polyphase and prototype interpolation filters.

Recursive polyphase filters have not been popular because they are not linear phase, in general.[73] The practice of recursive digital filtering requires an understanding of fixed-point arithmetic, truncation noise recirculation, [Dattorro] and transient phenomena.

We present here the simple recursive technique of Allpass interpolation which is useful primarily for microtonal changes in pitch (less than ±one semitone). Linear interpolation will outperform it from the standpoint of the Total Harmonic Distortion plus Noise[74] (***THD+N***). Otherwise, Allpass interpolation eliminates the drawbacks of Linear interpolation in this microtonal region, and makes the interpolation sound analog.

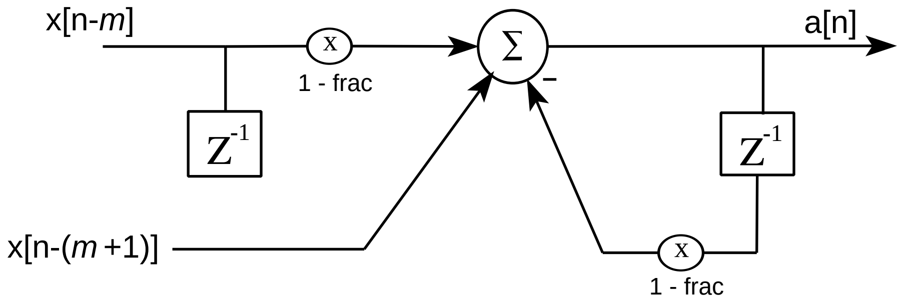


Figure PA. Allpass interpolation circuit. ©JonD 1994

Figure PA, a modification to Figure PL, shows the actual circuit used to implement Allpass interpolation. [Smith/Friedlander] Our application of Figure PA employs time-varying filter coefficients. The formal derivation of the ideal network [Vaidyanathan] [Renfors/Saramäki] requires as many recursive memory elements as there are coefficients **(**$L=2^{23}$ in 24-bit (**q**23 format) two's complement**)**.[75] In the ideal network, the filter coefficients are fixed. That is one reason why this simpler network in Figure PA (employing only one recursive element with a time-varying coefficient) only performs well (in terms of ***THD+N***) for small pitch changes. Making the same analytical circuit approximation as before, we can see that connection of the node at x[n-(*m* +1)] instead to the unit delay would make the instantaneous transfer function of the resulting polyphase filter, allpass. This means that the polyphase interpolation circuit in Figure PA has a frequency response which is approximately allpass; $|A(e^{j\omega})/X(e^{j\omega})| \approx 1$.

[73]Renfors' IIR design offers nearly linear-phase recursive polyphase and prototype interpolation filters. [Renfors/Saramäki]

[74]This is a measure of signal purity which aggregates everything that is not signal, and then relates that to some purified reference which is usually the signal itself. ***THD+N*** distortion is inherent to either interpolation process.

[75]Each recursive memory element resides in a structure like Figure PA having the feedforward delay element connected. See Appendix IX. Our simulations have shown that as few as $L=2^8$ recursive elements work quite well to make constant or sweeping Pitch Change over a large range. For the analogous case in Linear interpolation, also see [Crochiere/Rabiner,pg.81,ch.4.3.11] which shows that Linear interpolation is ideally implemented because of the lack of any long term memory.

As shown in Figure PA, the time-varying coefficients (frac_u=1-frac) are easily derived from ***m*.**frac as in Figure L. We find that, subjectively, the circuit of Figure PA sounds quite smooth in musical applications. But, analytically speaking, the desired signal-delay introduced by the circuit approximation does not track frac as well as it does for Linear interpolation.

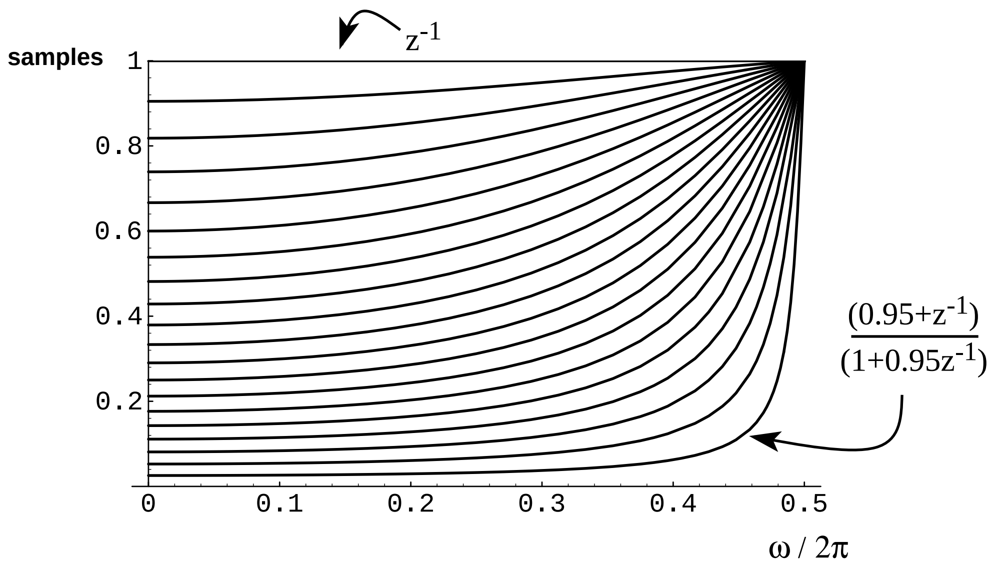


Figure PAD. Delays of Allpass interpolation polyphase filters.
The extreme polyphase filter z^{-1} is not used.

Figure PAD is the plot corresponding to Figure PLD. Notice how the delay between filters is spaced unevenly; especially noticeable at DC. But for each individual filter, the delay is still fairly constant in the low frequency region.

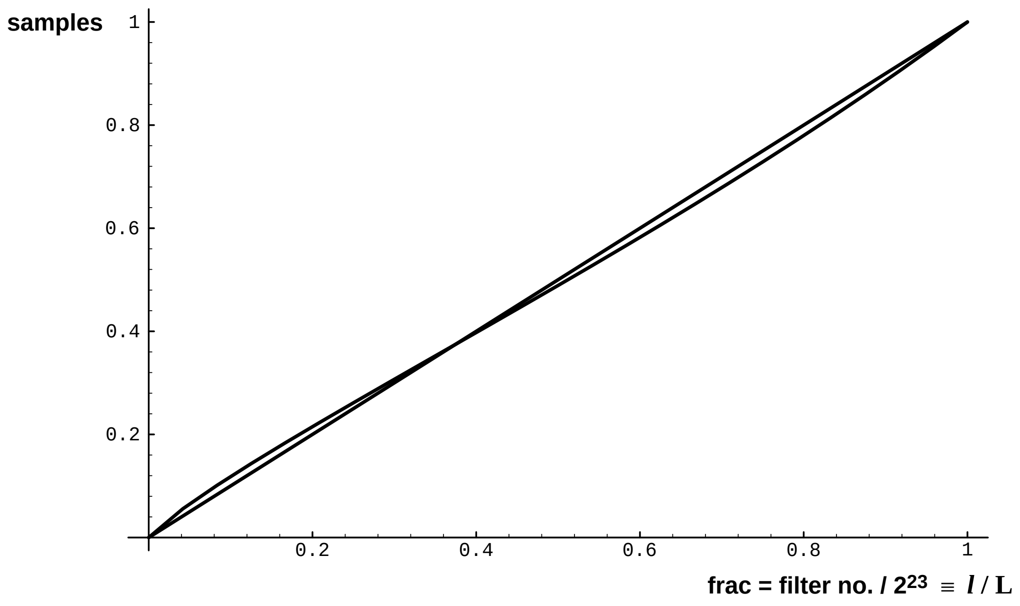


Figure PAA. Average delay of each Allpass interpolation polyphase filter.

Surprisingly, the average delay (over frequency) of each polyphase filter, shown in Figure PAA, is better than that for Linear interpolation (compare to Figure PLA).

It is reasonable to expect that we could force the delay of each polyphase filter to be equal to any desired value at a particular frequency by appropriately altering the polyphase filter coefficients. This would be beneficial were we concerned with ***THD+N*** performance in a particular frequency region.

Indeed, the exact equation to warp the allpass circuit coefficients in Figure PA is [SmithIII,pg.178]:

$$1 - \text{frac} \;\rightarrow\; \frac{\sin\left(\frac{\omega}{2}(1-\tau)\right)}{\sin\left(\frac{\omega}{2}(1+\tau)\right)}$$

where τ is the desired fractional sample delay at ω, the desired normalized radian frequency. We can eliminate the dependency of the equation upon signal frequency ω when we are primarily concerned with ***THD+N*** performance at low frequencies (ω near 0). For then the coefficient **warp equation** simplifies to:

$$1 - \text{frac} \;\rightarrow\; \approx \frac{(1-\tau)}{(1+\tau)} = 1/3 + \sum_{i=1}^{\infty} \frac{(-2)^{i+2}}{3^{i+1}} (\tau - 1/2)^i \qquad \text{(sw.eq)}$$

This is a reasonable substitution for audio signals. From Figure L we can see that the desired fractional sample delay is frac, so we make the identification, τ = frac .

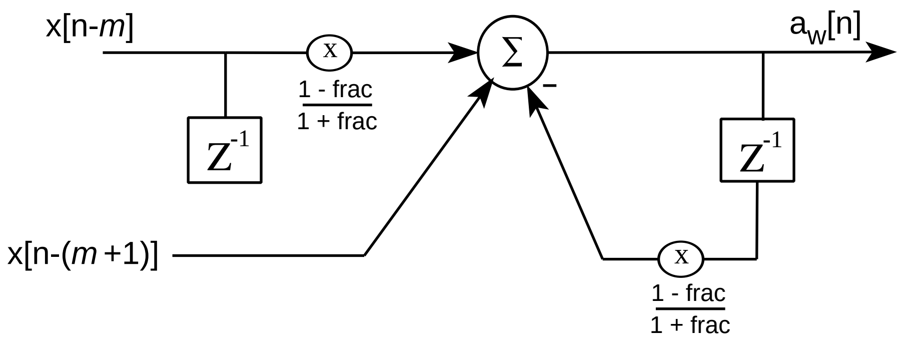


Figure WPA. Warped Allpass interpolation circuit. ©JonD 1994

Figure WPA shows the actual time-varying circuit used to implement warped Allpass interpolation using the simplified warp equation (sw.eq) in place of the allpass coefficients shown in Figure PA.

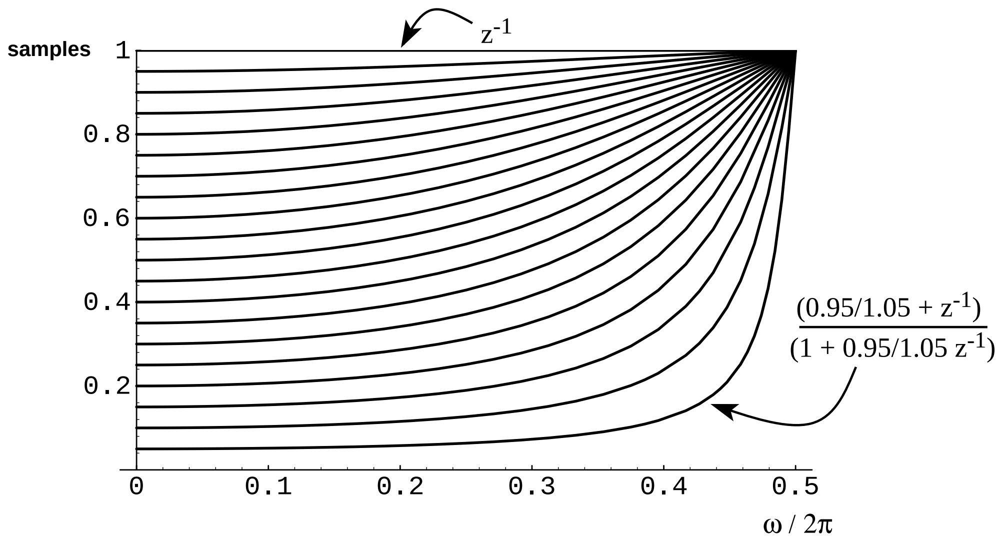


Figure PADW. Delays of Allpass interpolation warped polyphase filters.
The extreme polyphase filter z^{-1} is not used.

Figure PADW demonstrates the even distribution of delay across the polyphase filters at low frequency using the simplified warp equation (sw.eq) as shown in Figure WPA. This closer tracking between frac and signal delay will help improve somewhat the ***THD+N*** performance at low frequencies.

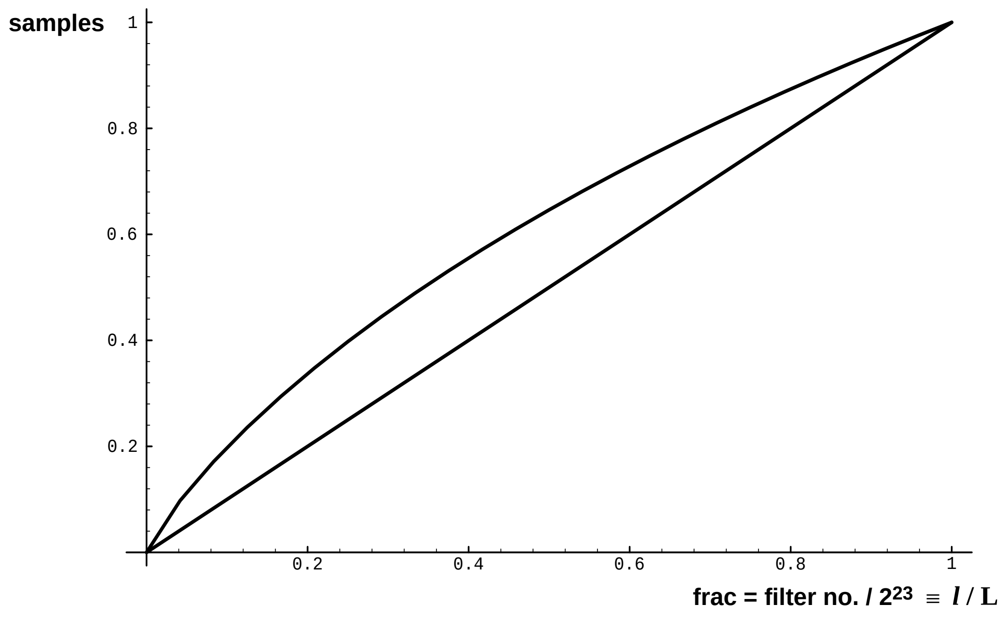


Figure PAAW. Average delay of each Allpass interpolation warped polyphase filter.

But the improvement to delay distribution in the low frequency range causes the average delay shown in Figure PAAW to suffer as we might expect (compare to Figure PAA). We find, analytically, that for Allpass interpolation of low frequency sinusoids on up to a few kHz, use of the simplified coefficient warp equation (sw.eq) is desirable from a ***THD+N*** standpoint. But for some of our musical applications we have not found it absolutely necessary to implement the circuit using different coefficients than those shown in Figure PA. Within an ESP2 implementation, however, several computed terms from the Taylor series expansion (sw.eq) of the simplified warp equation (or a table lookup routine) would easily map 1-frac to the warped coefficients of Figure WPA. We demonstrate this shortly.

Distortion Analysis

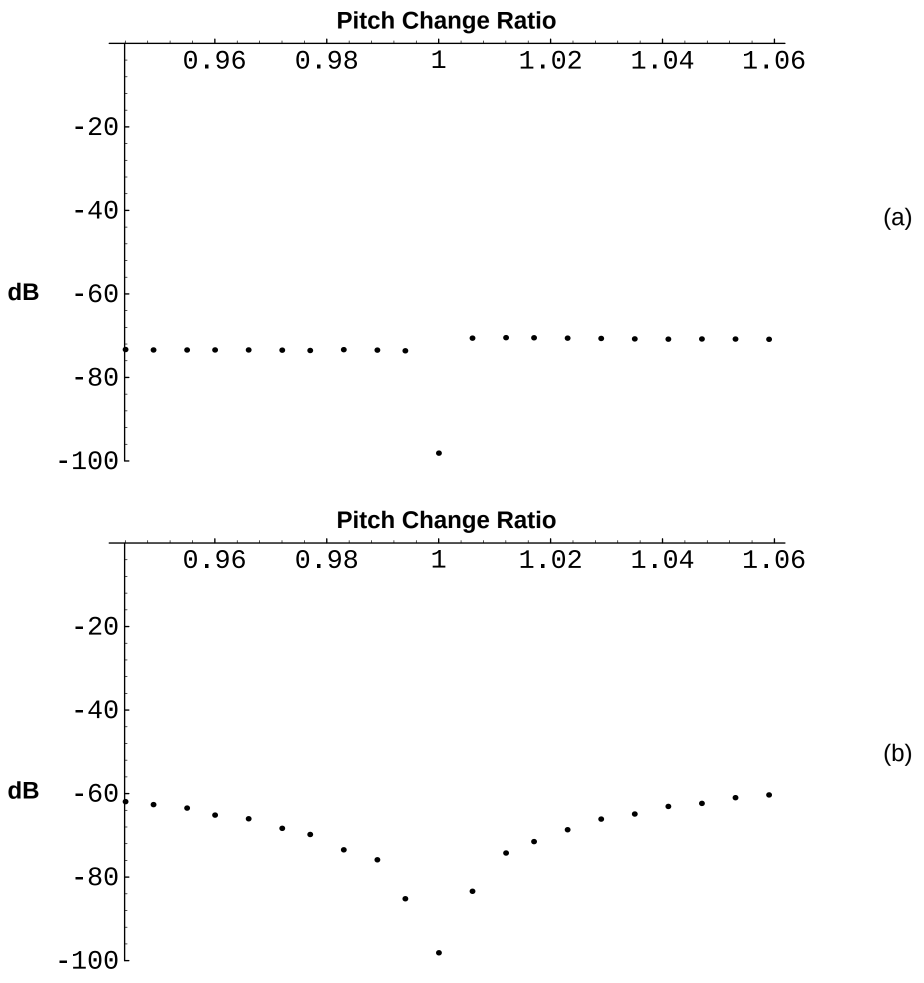


Figure THD. Simulated ***THD+N*** of pitch-changed sinusoid due to:
(a) Linear interpolation,
(b) Allpass interpolation.

Figure THD estimates the Total Harmonic Distortion plus Noise of a 16-bit 401 Hz sinusoid, sampled at 44.1 kHz, for various constant pitch changes spanning ±one semitone. The C-program simulation which produced the ***THD+N*** estimates in Figure THD (b), employs the simplified warp equation (sw.eq) and 16-bit polyphase filter coefficients. Without (sw.eq), the ***THD+N*** in (b) would be about -44 dB. The polyphase filter coefficients used to make Figure THD (a) are 8-bits in width.

The employment of Allpass interpolation now warrants consideration of transient phenomena as well as truncation error recirculation: The predominant artifact exposed in Figure THD (a) (Linear interpolation) is harmonic distortion due to the facts: 1**)** the two-tap FIR magnitude transfer is modulated by the time-varying coefficients, and 2**)** the actual delay of the two-tap FIR deviates from the desired time-varying delay. The predominant artifact exposed in Figure THD (b) (Allpass interpolation), however, is distortion due to transients arising from the time-varying polyphase filter coefficients. Even though the filter coefficients are updated at the sample rate, the nature of the Pitch Change/Shift algorithm (as outlined in Figure L) demands a disjunct sequence of desired fractional-sample delay. While truncation noise is a second-order artifact in both cases, we mention that truncation post-accumulation, as can be seen in the ESP2 programs, is emulated.

1.5. ESP2 Programs for Allpass Interpolation

We show the required changes to the ESP2 Linear interpolation program to implement Allpass interpolation by two of the methods discussed above. The following sections of code would each replace the corresponding section labelled !*** Linear interpolation *** . This first implementation does not employ coefficient warping.

```
!***************** Allpass interpolation ******************JonD 12/30/94***********
NOP                                    MOV #2.**-8. q24 > MACP_L     !1/256th sample offset
MOV nominal_delay > MACP_H             MOV *&VoiceL[(+)] > MACP      !to second seed
MACP + yqn X chorus_width > &VoiceL[ ]
MACP + frac_u X *&VoiceL[ ] > MAC      LSH MACRL >>1 > frac          ! q23
MAC  - frac_u X vibrato > vibrato      DIFF frac > frac_u
!*************************************************************************
```

The code above implements the digital circuit in Figure PA having time-varying coefficients derived as in Figure L. The total cost of this Allpass interpolator is one more line of ESP2 code over that for the Linear interpolator.

We require the implementation to prevent prolonged Nyquist ($F_s / 2$) oscillation that comes about when frac=0. This happens whenever chorus_width is set to zero by the system host. We minimize Nyquist oscillation by forcing the circuit to interpolate by a constant fraction of a sample at all times. (That is the purpose of the first line of code above.) This technique introduces a constant fractional offset into the equation for *m*.frac in Figure L (not shown there), and has little deleterious audible consequence.

Next we consider coefficient warping to improve the processing ***THD+N***:

```
DEFGPR      Taylor      nest
!***************** warped Allpass interpolation ************JonD 5/22/95************
MOV  nominal_delay > MACP                         MOV  *&VoiceL[(+)]  > MACP
MACP  +  yqn X chorus_width  >  &VoiceL[ ]
MACP  +  frac_u  X  *&VoiceL[ ]  > MAC            LSH  MACRL >>1 > frac
MAC    -  frac_u  X  vibrato > vibrato            SUB  HALF, frac > Taylor

MOV  #-32./81. > MACP                             MOV  #16./27. > MACP
MACP  +  #64./243. X Taylor > nest                ASH  #-8./9. >>1 > MACP
MACP  +  nest X Taylor >>1 > nest                 ASH  #1./3. >>1 > MACP
MACP  +  nest X Taylor > nest
MACP  +  nest X Taylor <<1 > frac_u
!*****************************************************************************
```

This Allpass interpolator code employs the first five terms of the Taylor series expansion of (sw.eq) to map frac into (1-frac)/(1+frac). It implements Figure WPA via the map:

frac_u = 1/3 + (frac-1/2)**[**-8/9 + (frac-1/2)**[**16/27 + (frac-1/2)**[**-32/81 + (frac-1/2)**[**64/243**]]]]**

Since the map is only approximate, <u>it is no longer necessary to prevent Nyquist oscillations as before</u>. We measure an improvement of 26 dB in ***THD+N*** over that of the previous Allpass interpolator code,[76] attributable to the coefficient warp. Linear interpolation exceeds this particular ***THD+N*** performance, but only by a few dB; viz.,

Measured Linear interpolation:	-78 -> -88 dB
Measured warped Allpass interpolation:	-77 -> -85 dB

These ***THD+N*** measurements are time-varying because the sinusoid frequency is slowly modulated by the LFO, $y_q n$, in the code. That is why they are indicated as a range. All polyphase filter coefficients are 24 bits, the sample rate is 44.1 kHz. As is apparent from the ESP2 program, truncation is post-accumulation.

Having reached parity between the two processes in terms of ***THD+N***, we would preferentially choose Allpass interpolation because it eliminates the drawbacks stated at the outset when employed in a microtonal region.

1.6. Theoretical Extensions

[Laakso/Välimäki] broadens the scope of this Allpass approach to interpolation. They provide formulae for allpass polyphase filters of higher order that possess a more linear delay vs. frequency, thereby providing a prototype interpolation filter (see Appendix IX) having higher stopband rejection. The delay linearization as follows filter order is not quite as fast as we might like, however. [Välimäki/Laakso] deals with transient phenomena.

[76]having declared the LFO Freq=0.1 Hz in a real-time ESP2 hardware development system, frequency modulating a 24-bit, 400 Hz sinusoid into an Audio Precision, Inc., analog signal analyzer.

2. The White Chorus Effect

Any complete discussion of the Chorus effect must consider its relative, the Flanger. The intended goal of Chorusing is to emulate the independence of multiple like-voices playing in unison. But the goal of Flanging is to jolt the ear's time-correlation mechanism by offering a signal replica delayed by an amount that is within the integration time constant of the hearing system.[77] We must first note that there is a very strong bond between the implementation of the Chorus effect and of the sonically radical Flange effect. What has gestated into the industry standard for this species condenses simply to a sum of the original input signal with a delayed replica. In either effect, the replica delay is modulating and never static. The consequence of summing the two signals is to introduce moving troughs into the input signal spectrum. In the case of Flanging, that is the desired result; the deeper and more selective the troughs are, the better.[78] In the case of Chorusing, the troughs are undesirable and an effort is made to limit their depth by summing unequal amounts of original signal and delayed replica. But the primary distinguishing design feature of the Choruser is that the <u>minimum</u> of the modulated delay time is much greater than that of the Flanger. The principal reason is so as to avoid flanging, by the Choruser, which becomes subjectively more pronounced for small delays. Indeed, the best Flangers can sweep the delay all the way to absolute zero; no delay.

In Figure Chorus we present a modification to the industry standard two-voice Chorus effect, that attempts to compensate for the spectral aberration caused by the many troughs in the feedforward sum. The modification is the introduction of a feedback path into the delayline, whose tap point is separate from that of the feedforward path but fixed at the center of the modulating delay in the feedforward path. Hence, the Chorus circuit approximates an allpass filter when the changing delay in the feedforward path is proximal with the fixed delay in the feedback path. We prefer not to feed back a modulating signal because the modulation induces pitch change. Feeding back a pitch changed signal produces more pitch change and becomes objectionable quickly for either intended effect. Feedback is gainfully employed in the Flanger in an inverse sense to make a comb filter that <u>increases</u> the perceived depth of the troughs; i.e., the same circuit can be used for both effects.

---

[77]The familiar thunder of a jet aircraft often reaches our ears by combination in air of the direct and reflected engine backwash. As the aircraft changes position, the reflection time changes and Doppler due to the aircraft's recession is introduced into both paths. The roar is more interesting as the reflection time is swept; that introduces more Doppler pitch change into the reflected path.

[78]The Flange effect gets its name from a studio technique that sums two synchronous magnetic-tape recorder signals playing identical material. The recording engineer places the thumb on one tape flange to bring the two recorders slightly out of synchronization thus creating the effect. In the 70's the Flange effect was emulated by analog phase shifting networks consisting of a cascade of allpass filters having time-varying elements. Implemented in this manner, the problem of delaying a signal by brute force was overcome. These devices were called Phasers and remain popular because the spectral troughs are not spaced harmonically, in general. While second-order allpass filter sections of the cascade offer more control [J.O.Smith] over trough frequency and selectivity, the Phaser is well emulated in DSP using only first-order sections [Hartmann] [Beigel] (which may be quieter in terms of truncation noise performance). The spectral troughs are harmonically stretched in the first-order case. In any case, global feedback enhances the effect, affecting perceived trough depth.

Table Chorus

Effect	*blend*	*feedforward*	*feedback*
industry standard Chorus	1.0	0.7071	0.0
White Chorus	0.7071	1.0	0.7071
Flanger	0.7071	0.7071	-0.7071

The settings in Table Chorus refer to Figure Chorus; they are given as typical values and are not unique, although the White Chorus settings approximate an allpass response. The maximum magnitude of *feedback* is 0.9999999 (**q**23) for stability. When the circuit performs Flanging, the *blend* and *feedforward* coefficients must be equal for maximum trough depth.

For the White Chorus effect, introduced here, a typical value of the *blend* coefficient in Figure Chorus would be 0.7071 and remains fixed, for the most part. A typical delay center (NOMINAL_DELAY) would be 600 samples at 44.1 kHz, and a typical peak delay excursion (CHORUS_WIDTH) of the modulation about the delay center would be approximately 550 samples. A typical rate of modulation would be about 0.15 Hz.

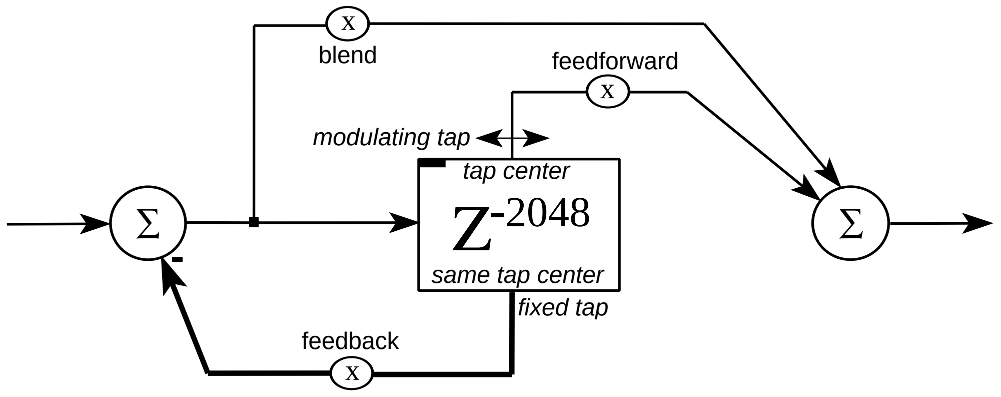


Figure Chorus. The industry standard Chorus effect circuit with feedback.
The strong flange zone is indicated; the leftmost (0^+ delay) portion of the delayline.

Design
This circuit in Figure Chorus produces a brilliant tone quality having pleasing movement with a little ambience. Used singly, this circuit's effect is sometimes called *Doubling*. Typical Chorus design configurations see two such circuits, however, a quadrature oscillator providing the modulation; each circuit having 90^o relative phase displacement.[79] Each circuit signal would typically be output to separate channels; this configuration is successfully applied to mono, stereo, or panned input signals. Time delay differences between two output channels carrying coherent signals emit localization cues known as the Haas effect. While musically interesting, it is a persistent source of irritation for recording engineers attempting to accurately place a musical instrument into a mix. For this reason it is prudent to place a stereo field control (or a panning circuit) at the output of any Chorus algorithm.

[79]The section on Sinusoidal Oscillators discloses highly efficient quadrature designs.

Flanger design would normally incorporate modulation of the same phase in each channel. The strong flange zone occupies approximately the first 1 ms of the delayline. The viable regions of delay excursion for the Chorus and Flanger effects overlap, however, and can easily be determined by ear.

It is important to keep the Chorus circuit free of nonlinearity for those many professional guitarists who want this effect as clean and simple as possible. For those particular guitarists, simplicity of the Chorus design is a virtue as they use this effect most all of the time. Allpass interpolation[80] for delay modulation becomes critical to the transparency of any Chorus design. Recall that Linear interpolation is a time-varying lowpass filtering operation. Indeed, a multi-voice Chorus design employing Linear interpolation subjects the signal to significantly audible amounts of low-pass filtering. We term the Chorus *White* when both negative feedback and Allpass interpolation are employed to minimize the spectral aberration that would be consequent in the absence of these two signal processing techniques.

Flangers, on the other hand, can benefit from a mild nonlinearity introduced into the signal path in front of the effect, so that the induced troughs see a more rich signal source. But Allpass interpolation is also critical to successful Flanger design so that the dynamically delayed signal remains unfiltered. Lowpass filtering of the delayed signal via Linear interpolation would reduce the depth of the high frequency troughs; that is undesirable for a good Flanger.

Caveat

For particular types of input material (e.g., percussive signals) and subjectively long nominal delays (10 ms), the feedback path can introduce echoes which are objectionable.[81] The level of objection is somehow related to the particular form of interpolation. The use of the Allpass interpolator with coefficient warping (sw.eq) seems to mitigate the echoes' aural impact.[82] In any event, one can always turn down the *feedback* knob in this circumstance.

[80]as discussed in the Application of the same name,

[81]The old bathroom reverberator.

[82]We apologize for being unable to characterize this problem more accurately at this time.

Vibrato
One might be interested to know the amount of pitch change when the *blend* and *feedback* coefficients are 0; i.e., when the circuit is made to produce Vibrato.[83] Without loss of generality we may consider the Vibrato applied to a sinusoid of arbitrary amplitude, A, and radian frequency, ω. Using terminology established for the Interpolation Applications, from (lipdf) we write,

$$\begin{aligned} x((n - m.\mathrm{frac})T) &= A \sin\big(\omega (n - m.\mathrm{frac})\, T \big) \\ &= A \sin\big(\omega (n - (\mathrm{NOMINAL\_DELAY} + y_q n \ \mathrm{CHORUS\_WIDTH}))\, T \big) \end{aligned}$$

where, $y_q n = \sin(\omega_\varepsilon n T)$, is the LFO and ω_ε is the radian rate of modulation (2π Freq; see the interpolation program called *Linear*). NOMINAL_DELAY and CHORUS_WIDTH are expressed in units of samples.

The instantaneous radian frequency is:

$$\begin{aligned} \Omega &= \partial\big(\omega (n - (\mathrm{NOMINAL\_DELAY} + y_q n \ \mathrm{CHORUS\_WIDTH}))\, T \big) / (T\, \partial n) \\ &= \omega \big(1 - \mathrm{CHORUS\_WIDTH}\ \omega_\varepsilon T \cos(\omega_\varepsilon n T)\big) \end{aligned}$$

Note that if the LFO were triangular, then the instantaneous frequency would be piecewise constant which is unnatural, hence not desirable.

$$\begin{aligned} \text{Pitch Change Ratio} &= \Omega / \omega \\ &= 1 - \mathrm{CHORUS\_WIDTH}\ \omega_\varepsilon T \cos(\omega_\varepsilon n T) \end{aligned}$$

This tells us that the pitch change is time-varying and proportional to the modulation frequency and the sample period, T.

$$\text{Pitch Change Ratio extrema} = 1 \pm \mathrm{CHORUS\_WIDTH}\ \omega_\varepsilon T$$

[83]Note that this implementation of Vibrato maintains the signal's macro-temporal features.

Detune Effect
There is yet another related effect which is outside the scope of the present discussion. Yet, its sonic impact is so powerful that it deserves mention; that is the Detune effect. It is accomplished in DSP employing the class of algorithm known as the Pitch Shifter. [Dattorro2400] The Detune algorithm class employs a splicing mechanism[84] (when implemented in the time domain; see Appendix IX) to maintain the macro-temporal signal features while shifting the pitch by a fixed microtonal amount using some form of interpolation. The pitch shifted signal is then mixed with the original. The result is often described by musicians as subjectively 'fattening' the sound.

This Detune effect occurs naturally and is built-in to instruments such as the pianoforte, mandolin, and twelve-string guitar. It accounts for one salient character of each instrument's sound. The Chorus and Detune effects are sonically quite different, the latter algorithm being more difficult to implement properly. The primary distinguishing feature of the two is that the pitch is necessarily modulating in the Chorus effect because of the means of implementation.

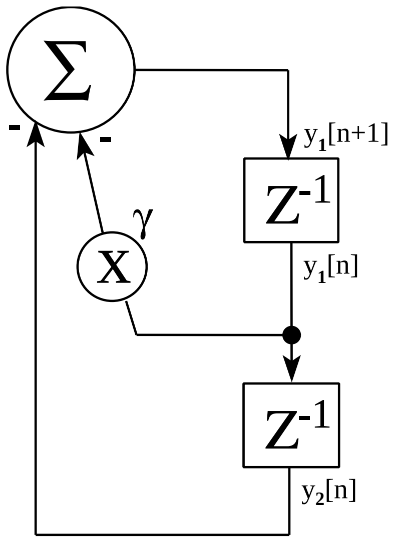
The reader should also be aware that contemporary sampling music synthesizers[85] often emulate Detuning through the use of Pitch Change, which in many cases is undesirable.

[84]unlike Pitch <u>Change</u> algorithms which have been the focus of our look into interpolation, and which are actually the topic of [Rossum] despite its title. The temporal features of the original signal are maintained in a pitch shifted replica; that is what distinguishes **Pitch Shifting** from Pitch Changing by fixed amounts. While many Pitch Shift algorithms employ delayline interpolation, the Lexicon Model 2400 Stereo Audio Time Compressor/Expander **(**designed by the author in 1986 and still in production [Dattorro2400]**)** sidestepped the need for interpolation by incorporating a variable-rate A/D conversion system. In that design, the D/A circuitry is fixed rate, the whole instrument operating in real time on pre-recorded material.
[85]rather sampler-type, which we distinguish from wavetable-type synthesizers,

3. The Low-Frequency Sinusoidal Oscillator

The low frequency sinusoidal oscillator (**LFO**) is ubiquitous in effect design. There are several implementation options:
1) direct form,
2) coupled form,
3) first modified coupled form,
4) second modified coupled form,
5) normalized waveguide.

The **direct form** oscillator is the most efficient option requiring only one multiply, but it is noisy unless truncation error feedback is used. [Haija/Ibrahim] [Dattorro] The error feedback can be implemented using only one or two adds, so it is attractive. The single coefficient γ requires high resolution for very low frequencies of oscillation, however.



$$\begin{cases} y_1[n+1] = -\gamma\, y_1[n] - y_2[n] \\ y_2[n+1] = y_1[n] \end{cases}$$

$$y_1[n] = \frac{1}{\sin(\omega)} \big(y_1[0] \sin(\omega + n\omega) - y_2[0] \sin(n\omega) \big)$$

$$y_2[n] = \frac{1}{\sin(\omega)} \big(y_1[0] \sin(n\omega) + y_2[0] \sin(\omega - n\omega) \big)$$

$$\boxed{\gamma = -2\cos(\omega) \quad ; 0 < \omega < \pi} \qquad \text{poles} = \frac{-\gamma}{2} \pm j\sqrt{1 - \frac{\gamma^2}{4}}$$

Figure DF1. Direct Form Sinusoidal Oscillator.

The direct form oscillator was analyzed using State-Variable theory,[86] the results of which are shown in Figure DF1. This type of analysis excels at finding the zero input response (the **ZIR**). (See Appendix X.) The selected output is expressed in terms of the initial conditions of all the memory elements, the state variables. This method of analysis differs considerably from the technique of solving the recursive difference equation for the selected output, in so far as the initial conditions required by the latter are of the output itself. In the case of the direct form, these two methods coincide.

Obviously, by choosing the initial states $y_1[0] = 0$ and $y_2[0] = -\sin(\omega)$, we get

$$y_1[n] = \sin(n\,\omega)$$

There is no quadrature sinusoid at any node. But by choosing, for example, $y_1[0] = 1$, and $y_2[0] = \cos(\omega)$, we get

$$y_1[n] = \mathbf{(}1/\sin(\omega)\mathbf{)}\ \mathbf{(}\sin(\omega + n\,\omega) - \cos(\omega)\sin(n\,\omega)\mathbf{)} = \cos(n\,\omega)$$

The direct form oscillator is *hyperstable*[87] under coefficient quantization as proven by the equation for the pole locations in the **z**-plane in Figure DF1. Regardless of the quantization of γ, the pole radii are exactly 1 as determined from the pole magnitude. <u>Any instability in the sinusoidal waveform can only be attributed to signal quantization effects</u>, primarily in the form of truncation error in this recursive topology.

One must be cognizant of the relationship between oscillation frequency and amplitude when using the different oscillators presented in this section. The equations of Figure DF1 predict that when frequency is <u>changed</u> via γ after the oscillator has been running for some time, amplitude will deviate, in general. One theoretically overcomes this problem by simultaneously updating the memory elements (the *states*), but this can be difficult in practice depending upon the particular oscillator topology.

The direct form oscillator circuit can be derived from one trigonometric identity;[88] viz.,

$$\sin((n+1)\omega) = \cos(\omega)\sin(n\,\omega) + \sin(\omega)\cos(n\,\omega)$$
$$\sin((n-1)\omega) = \cos(\omega)\sin(n\,\omega) - \sin(\omega)\cos(n\,\omega)$$

where ω is the normalized radian frequency of oscillation ($2\pi f\,T$) controlled by γ which must be in **q**22 binary-radix format for ESP2. Summing the two equations yields:

$$\sin((n+1)\omega) = 2\cos(\omega)\sin(n\,\omega) - \sin((n-1)\omega)$$

[86]following the analysis presented in [Gordon/Smith] for the coupled form and the second modified coupled form,
[87]We shall define 'hyperstable' in this context to mean stability of oscillation in the face of coefficient quantization. The effect of signal quantization in networks is not included in this definition.
[88]This was pointed out to the author by Michael Chen.

Making the substitution, $y_1[n] \leftarrow \sin(n\,\omega)$, we get

$$y_1[n+1] = 2\cos(\omega)\; y_1[n] - y_1[n-1]$$

which is the difference equation for the circuit.[89]

The **coupled form** is an established topology known by several names including the Rader/Gold, and normal form. The coupled form is a state-space digital filter structure and is one of the foremost contributions of the branch of Linear System theory known as State-Variable analysis. The coupled form has many attributes including low truncation noise and low coefficient sensitivity, respectively due to signal and coefficient quantization within a finite precision machine. [Jackson,ch.4.4] The latter attribute makes tuning easier. Its primary detriment is that four multiplys are required in its unmodified form.

The unmodified coupled form in Figure X (a) shows the four multiplys required for oscillation, including two $\cos(\omega)$ coefficients. Using all four coefficients, the pole locations are ideally on the unit circle. It can be deduced from the equations given in Figure X (a) that for any initial states ($y[0]$, $y_q[0]$) the outputs at $y[n]$ and $y_q[n]$ yield <u>quadrature</u> sinusoids. But when the ideal filter coefficients become quantized to their representation in a finite precision machine, the pole locations are perturbed in a direction dependent upon the polarity of the coefficient quantization error.[90] The impact of this is that the pole radii are no longer exactly 1, so the oscillator amplitude either decays to zero or clips (assuming automatic saturation arithmetic) after some time has past. We do not relish the four multiply requirement and we recall that the direct form does not suffer from this particular problem. This is because the direct form pole radii can be fixed by the second-order recursive coefficient [Dattorro,(30)] which is always precisely set to 1 for our present application.

The coupled form has a simpler trigonometric derivation, but uses two identities; viz.,

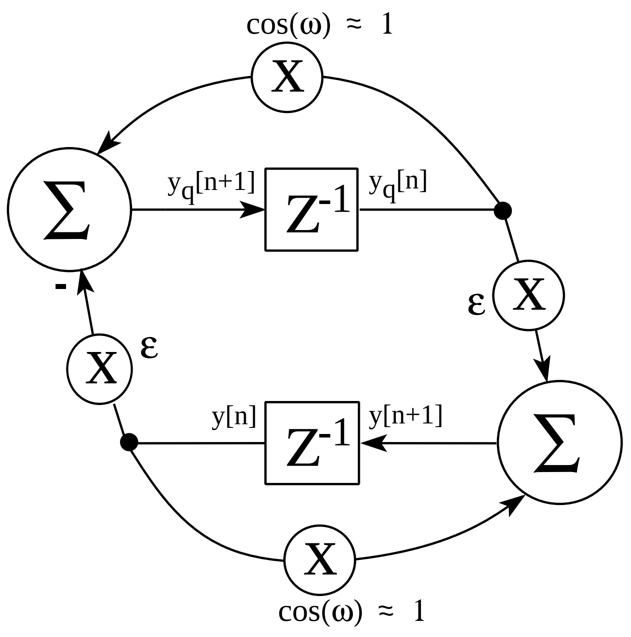
$$\cos((n+1)\omega) = \cos(\omega)\cos(n\,\omega) - \sin(\omega)\sin(n\,\omega)$$
$$\sin((n+1)\omega) = \sin(\omega)\cos(n\,\omega) + \cos(\omega)\sin(n\,\omega)$$

Making the substitutions, $y[n] \leftarrow \sin(n\,\omega)$, $y_q[n] \leftarrow \cos(n\,\omega)$, we get

$$y_q[n+1] = \cos(\omega)\; y_q[n] - \sin(\omega)\; y[n]$$
$$y[n+1] = \sin(\omega)\; y_q[n] + \cos(\omega)\; y[n]$$

[89]This also happens to be the generating recurrence relation of the Chebyshev polynomials; $T_{n+1}(x) - 2x\,T_n(x) + T_{n-1}(x) = 0$. [Hamming,pg.473]

[90]The availability of the ROUND(), INT(), or BTRUNC() in-line assembler truncation functions now warrants appreciation.



$$\begin{cases} y_q[n+1] = \cos(\omega)\; y_q[n] - \varepsilon\; y[n] \\ y[n+1] = \varepsilon\; y_q[n] + \cos(\omega)\; y[n] \end{cases}$$

$$y_q[n] = y_q[0]\cos(n\,\omega) - y[0]\sin(n\,\omega)$$

$$y[n] = y_q[0]\sin(n\,\omega) + y[0]\cos(n\,\omega)$$

$$\boxed{\varepsilon = \sin(\omega) \quad ; \omega < \pi} \qquad \text{poles} = \cos(\omega) \;+/-\; j\,\varepsilon$$

(a)

$$\begin{cases} y_q[n+1] = y_q[n] - \varepsilon\; y[n] \\ y[n+1] = \varepsilon\; y_q[n] + y[n] \end{cases}$$

$$y_q[n] = \left(\frac{1}{\cos(\omega)}\right)^n \left(y_q[0]\cos(n\,\omega) - y[0]\sin(n\,\omega) \right)$$

$$y[n] = \left(\frac{1}{\cos(\omega)}\right)^n \left(y_q[0]\sin(n\,\omega) + y[0]\cos(n\,\omega) \right)$$

$$\boxed{\varepsilon = \frac{\sin(\omega)}{\cos(\omega)} \quad ; \omega < \pi/2} \qquad \text{poles} = 1 \;+/-\; j\,\varepsilon$$

(b)

Figure X. (a) Coupled Form (Four Multiplier) Sinusoidal Oscillator.
(b) First Modified Coupled Form (Two Multiplier) Low Frequency Oscillator.

We will make a modified coupled form digital filter behave as an oscillator by placing its poles either slightly beyond or precisely on the unit circle in the **z**-plane, even in the presence of coefficient quantization error. We will see that both of these choices of pole locations can be achieved using only two multiplys, and both have useful purposes.

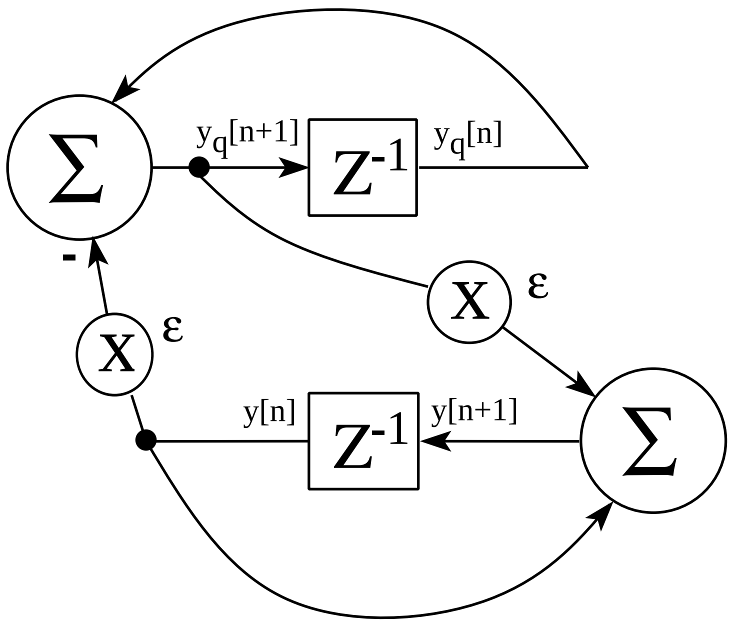
If the coupled form oscillator is employed as an LFO, we make the observation that the two $\cos(\omega)$ coefficients are close to 1. If we set them to 1 permanently, we eliminate two multiplys and arrive at the **first modified coupled form** in Figure X (b). Using the first modified coupled form, the quadrature sinusoids at $y[n]$ and $y_q[n]$ will always clip somewhat (assuming saturation arithmetic) because the pole radii are in excess of unity. When the oscillator frequency is very low, the clipping will be slight.

The advantage of this first modified coupled form implementation is that its output amplitude is at least unity, even after frequency is abruptly changed. The oscillator never blows up, as the equations in Figure X (b) would predict, because of the saturation nonlinearity built into the ESP2 computation units. The detriment to the use of this oscillator is that the tuning frequency is practically restricted to a portion of the first quadrant in the **z**-plane.[91] This modified coupled form of the oscillator would be chosen when the criteria for selection of an LFO does not include sinusoid purity, but full amplitude stable quadrature signals are a must.

The **second modified coupled form**, first presented in [Gordon/Smith], is shown in Figure X+1. This oscillator produces a high purity sinusoid at the two outputs, $y[n]$ and $y_q[n]$, whose noise floor is low enough to characterize D/A converters. The two outputs are no longer exactly in quadrature as before,[92] but the oscillator is hyperstable because the pole radii are exactly 1 even when epsilon is quantized.[93] The tradeoff for this stability is the linking of amplitude to frequency of oscillation as seen in the equations in Figure X+1. By observation of the poles, the tuning frequency now spans DC to Nyquist as epsilon goes from 0 to 2, and so the range of oscillation is not restricted to low frequencies as in the first modified coupled form.

We have found as predicted, using the second modified coupled form, that when epsilon (the frequency control) is abruptly changed, the oscillator amplitude will change itself to a new value. This is because we ignore the new initial states, $y_q[0]$ and $y[0]$, at the time of the frequency change; i.e., no attempt is made to adjust them. We observed these consequent deviations in amplitude and empirically found them to be less than a decibel over a very useful range of frequency for an LFO. It was never found necessary, in any of our applications, to compensate for these small changes in amplitude. The built-in saturation arithmetic within the ESP2 computation units eliminates any possibility of long-term clipping, completely.

[91]This restriction is overcome in practice by running the oscillator twice as fast, which means twice the code.
[92]From the equations in Figure X+1, it can be shown trigonometrically for any initial states that the two sinusoids are in a near quadrature relationship; being off by exactly 1/2 sample at any frequency of oscillation. This was first pointed out to the author by Timothy S. Stilson.
[93]Like the direct form, any instability in the sinusoidal waveform can only be attributed to signal quantization effects, primarily in the form of truncation error in this recursive topology.



$$\begin{cases} y_q[n+1] = y_q[n] - \varepsilon\, y[n] \\ y[n+1] = \varepsilon\, y_q[n+1] + y[n] \end{cases}$$

$$y_q[n] = \frac{1}{\cos(\frac{\omega}{2})}\left(y_q[0]\cos(n\omega - \frac{\omega}{2}) - y[0]\sin(n\omega)\right)$$

$$y[n] = \frac{1}{\cos(\frac{\omega}{2})}\left(y_q[0]\sin(n\omega) + y[0]\cos(n\omega + \frac{\omega}{2})\right)$$

$$\boxed{\varepsilon = 2\sin(\frac{\omega}{2}) \;\; ; \omega < \pi}$$

poles = $1 - \frac{\varepsilon^2}{2} +/- \; j\,\varepsilon\sqrt{1 - \frac{\varepsilon^2}{4}}$

Figure X+1. Gordon/Smith Sinusoidal Oscillator.
a.k.a. Second Modified Coupled Form (Two Multiplier) Oscillator.

The ideal digital **integrator**s, $1/(1 - z^{-1})$, have never presented a problem in practice because there is a zero of transmission at $z=1$ (DC) across each embedded integrator from its input to its output. This comment applies to both modified coupled forms.

The second modified coupled form oscillator ESP2 code, shown within the Linear Interpolation Application, is written such that if several oscillator routines were cascaded, the interleaved code resulting would be most efficient in terms of program space. The cascaded routines would then average about two program lines per oscillator. That code for one oscillator is reproduced below:

```
!****** hyperstable near-quadrature oscillator (second modified coupled form) ******
NOP                            MOV yqn > MACP
NOP                            MOV yn > MACP
MACP - epsilon X yn  > yqn
MACP + epsilon X yqn > yn
```

If only one oscillator is required, the code shown may be compressed to occupy only three program lines as demonstrated in the FFT Radix-4 Application.

Real-Time Measure of Sinusoid Purity
We made measurements of ***S/(THD+N)*** for the direct form and the second modified coupled form oscillator, each running at a sample rate of 69818.181 Hz in a real-time ESP2 hardware development system. The measurements were taken using an Audio Precision, Inc., analog signal analyzer. For each oscillator, the two parameters are frequency of oscillation and the number of bits in the two's complement signal path. Binary truncation post-accumulation is used to limit the path-width. The results are tabulated in Table OSCTHD:

Table OSCTHD. Measurement of oscillator ***S/(THD+N)*** .

	direct form				**second modified coupled form**		
no. bits	freq.			no. bits	freq.		
	20Hz	100Hz	1000Hz		20Hz	100Hz	1000Hz
24	60 dB	75	86 dB	24	88 dB	86	86 dB
23	58	75	86	23	88	86	86
22	53	73	85	22	88	86	86
21	45	70	84	21	87	86	86
20	37	64	84	20	87	86	86
19	flatline	60	79	19	86	85	85
18		58	74	18	85	84	85
17		50	73	17	79	80	84
16		40	67	16	70	73	82
15		25	61	15	67	68	78
14		flatline	56	14	59	65	73
13			54	13	47	61	68
12			45	12	40	60	62
11			39	11	25	50	56
10			37	10	flatline	45	48
9			23	9		30	43
8			19 dB	8		21	42
7			flatline	7		flatline	35
6				6			34
5				5			28
4				4			24 dB
3				3			flatline
2				2			
1				1			

The hardware system total noise was roughly -86 dB, relative to full scale, which prevented measurements significantly below that level. The readings at -88 dB are probably due to a system signal-transfer anomaly. The term *flatline* in Table OSCTHD refers to a complete lack of any form of oscillation as viewed on an oscilloscope.

purity of direct vs. coupled form
We now postulate the reason pertaining to the foregone conclusion from the empirical data that the direct form oscillator is inferior to the second modified coupled form: It is that the latter has truncation error feedback built into its topology. We state this in an equivalent way; <u>each noise transfer of the second modified coupled form has zeroes of transfer</u>, whereas that of the direct form has not. Figure X+1n shows how truncation noise sources ($e[n]$ and $e_q[n]$) are conceptually inserted into a circuit in a linear fashion. [Jackson] [Dattorro] Each deterministic noise source in a contemporary DSP chip resides in front of a multiplier because that is the only location where truncation is demanded by the architecture.[94]

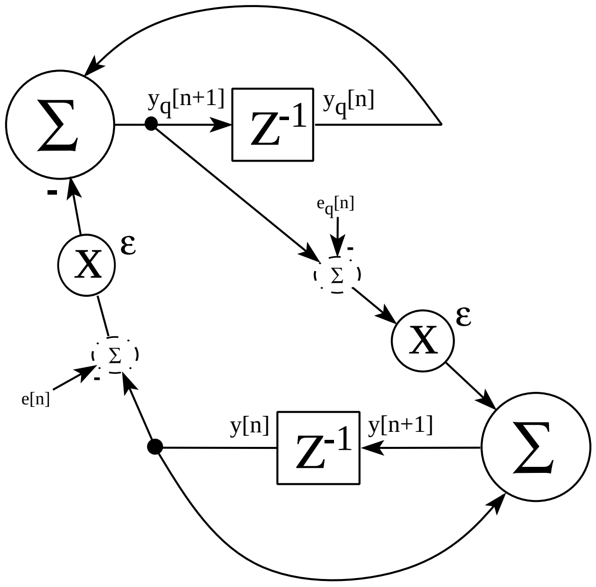


Figure X+1n. Second Modified Coupled Form Sinusoidal Oscillator showing noise sources.

Each noise source can make its way to either of two outputs for this coupled topology in Figure X+1n: $y_q[n]$ or $y[n]$. In each case, the noise transfer either picks up a zero at DC, or is multiplied by ε^2 which yields the same effect. In order for the direct form oscillator to perform as well, truncation error feedback must be employed to introduce a zero into its deterministic noise transfer, as stated at the outset.

[94]Were we to instead place the noise sources following the accumulators, for truncation post-accumulation, we would reach similar conclusions in the analysis of this topology. Again, the ideal digital integrators would pose no practical problem.

purity vs. frequency
The data in Table OSCTHD indicates that signal purity is a function of oscillator frequency for both topologies. Binary truncation noise can be modelled like quantization noise. [Opp/Sch,pg.353] Gray demonstrates [Gray,ch.6.3] how the quantization noise of a pure sinusoid is <u>not</u> characteristically white regardless of amplitude or frequency. He explains that the deterministic noise spectrum is in fact discrete, because the signal is sinusoidal,[95] and that the noise spectral components exist at odd harmonics of the sinusoid frequency including the fundamental.

Gray's references indicate that these results have been known for decades. The conclusions drawn can be generalized to the present truncation noise situation depicted in Figure X+1n:

$$e[n] = \sum_{\substack{m=-\infty \\ m \text{ odd}}}^{\infty} e^{jm\omega n}\ b_m \qquad \text{(en0)}$$

where ω is the (fundamental) frequency of oscillation. Equation (en0) has the form of a continuous-time complex Fourier series, then sampled in time. The Fourier series coefficients, b_m, form a conjugate-symmetric set, thus $e[n]$ is real-valued. The expression for $e_q[n]$ is similar. Each b_m will be some function of weighted ordinary Bessel functions of integer order m and real argument.

Thus far, we have been presenting all the equations for oscillation in terms of the zero input response (ZIR). But the noise model in Figure X+1n suggests that the noise sources, $e[n]$ and $e_q[n]$, are subject to the zero state response (**ZSR**) of the network. The ZSR of the oscillator is precisely that of an integrator centered (in the frequency domain) at the frequency of oscillation. Any noise energy in the vicinity of the oscillation frequency should cause the oscillator to blow up. This does not happen in practice because of the saturation nonlinearity built into most contemporary DSP chips. The injected noise just causes phase jitter and amplitude perturbations which decrease signal purity.[96] Signal quantization in a recursive network, then, is a secondary form of instability. The primary determinant of stability is pole location, which is why we have been concerned with the quantization of filter coefficients.

[95]to be more precise; because the analog signal, from which the sampled signal is derived, has a discrete spectrum,
[96]For this reason it is recommended to run all the oscillators at full-scale amplitude and to place a volume knob at the output.

The data in Table OSCTHD suggests that either implementation of the oscillator/integrator has constant (noise equivalent)[97] bandwidth, independent of center frequency. We intuit this because it appears that the quantity of noise goes up as center frequency moves down. Perhaps when the oscillation frequency is low, the more powerful harmonics of the noise spectrum are clustered closer together in the vicinity of the center frequency. Hence the integrator becomes more disturbed, thus the ***THD+N*** will be worse. A simpler explanation might be frequency domain foldover of the integrator transfer near DC.

chaotic behavior

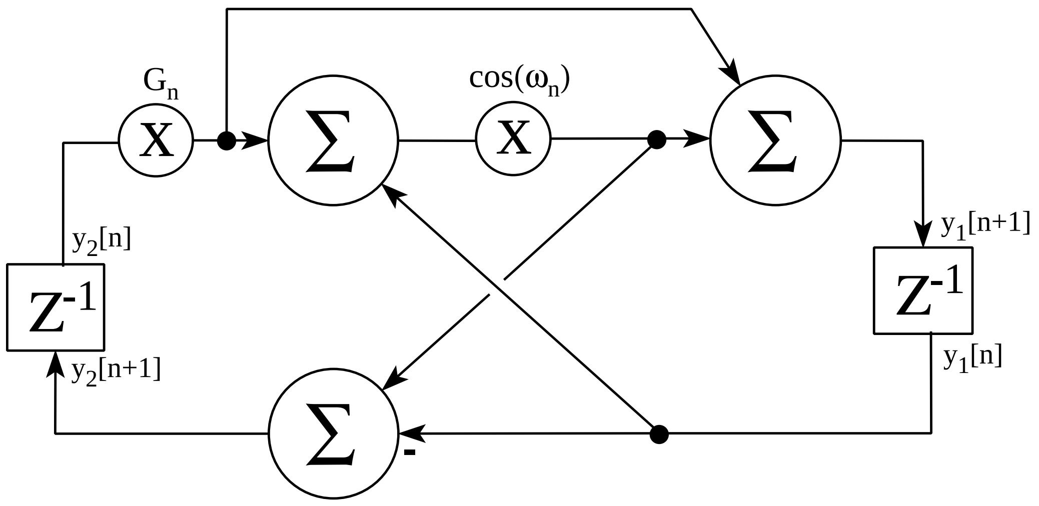
The direct form oscillator at 20 Hz is fascinating to view, however, when the signal path bit-width is 20 bits (just before flatline). The oscillation metamorphoses from sinusoidal to nearly triangular, and back; this chaotic but stable process occurs over periods of real time on the order of minutes. During each epoch, the oscillation frequency can be observed to change by as much as 1/2 the desired frequency.

It should be pointed out that although we show no data for very low frequencies in Table OSCTHD, the first and second modified coupled form sinusoidal oscillators are routinely called upon to produce frequencies less than 1 Hz (0.1 Hz typical).

[97]defined in [Harris] [Cooper]

More Recent Developments

Smith and Cook subsequently disclosed a lattice topology for sustained oscillation in [Smith/Cook] which is claimed more suitable for VLSI implementation. The oscillator shown in Figure SC is called the **normalized waveguide** oscillator because it was derived as a spin-off from Smith's results in the theory and implementation of digital waveguides.



$$\begin{cases} y_1[n+1] = \cos(\omega_n)\ y_1[n] + (\cos(\omega_n)+1)\ G_n\ y_2[n] \\ y_2[n+1] = (\cos(\omega_n)-1)\ y_1[n] + \cos(\omega_n)\ G_n\ y_2[n] \end{cases}$$

$$\begin{aligned} y_1[n] &= y_1[0]\ \cos(n\,\omega_n) - y_{2G}[0]\ \cot(\omega_n/2)\ \sin(n\,\omega_n) \\ y_2[n] &= y_1[0]\ \tan(\omega_n/2)\ \sin(n\,\omega_n) + y_{2G}[0]\ \cos(n\,\omega_n) \end{aligned} \qquad \text{(oscSC)}$$

$$G_n = \frac{\tan(\omega_n/2)}{\tan(\omega_{n-1}/2)} \quad ; 0 < \omega_n < \pi/2 \qquad \text{poles} = \cos(\omega_n)\ +/-\ j\sqrt{1 - \cos^2(\omega_n)}$$

Figure SC. Smith/Cook Normalized Waveguide Sinusoidal Oscillator.

Although this design recaptures the quadrature relationship of the sinusoids appearing at the circuit outputs, the primary contribution of this topology to the field of oscillator design is that it solves the amplitude deviation problem.[98] It is a remarkable distinction that this sinusoidal oscillator possesses: This normalized waveguide oscillator is designed for instantaneous change in frequency <u>without</u> concomitant change in amplitude,[99] hence the new notation (ω_n) showing frequency as a function of the time index. The frequency change must be performed properly, however. Toward this end, the amplitude coefficient G_n is introduced, and specified to deviate from 1 only at the time of the occurrence of the frequency change;[100] i.e., it is an amplitude compensation factor which is engaged for one sample period. The elegance of the Smith/Cook solution rests in the fact that knowledge of the state time (the precise value of n) at the occurrence is not required, neither is knowledge of the state values. All that is required is knowledge of the previous frequency.

<u>Whenever frequency is changed</u>, <u>the control</u> G_n <u>is all that is required to maintain constant amplitude in the oscillator circuit</u>; i.e., the memory elements need never be adjusted in the implementation. For the analytical equation (oscSC) to remain valid, however, $y_{2G}[0]$ and $y_1[0]$ must be re-evaluated whenever G_n deviates;

$$y_1[0] \;\; \text{<-} \;\; y_1[n_0]$$
$$y_{2G}[0] \;\; \text{<-} \;\; y_2[n_0]\, G_{n_0} = y_{2G}[n_0] \qquad \text{(oscG)}$$

where n_0 is the time index at the change; $n_0 \neq 0$. It is important that we interpret equation (oscSC) properly so that we may show the technique on paper. We will now demonstrate the validity of these assertions by example:

98 It is evident from the equations in Figure SC that sinusoid amplitude is linked to the frequency of oscillation. This is similar to the situation for all the oscillators presented, except for the Rader/Gold coupled form.

99 Because we are dealing with the ZIR, frequency change can be instantaneous in all the oscillators presented. But the Smith/Cook oscillator is the first one to deal effectively with amplitude correction at the instant of the change.

100 G_n is not at all involved with frequency tuning. The special case $G_0 = 1$. But G_n can be greater than 1, albeit momentarily, which will necessitate at least **q**22 arithmetic.

example

Suppose that at absolute time $n=0$, we are given $\omega_0=\omega_{-1}$, $y_1[0]=1$, and $y_{2G}[0]=0$. Then while G_n remains static we expect for $n = 0 \rightarrow \infty$,

$$y_1[n] = \cos(n\,\omega_0) \qquad \text{(osc0)}$$
$$y_2[n] = \tan(\omega_0/2)\ \sin(n\,\omega_0)$$

Suppose we suddenly freeze time at $n=n_{0+}$. At this time we desire a change in frequency which is effectively to take place at $n=n_0$. We do this by changing the one tuning coefficient from $\cos(\omega_0)$ to $\cos(\omega_{n_0})$ at time n_0, and by letting G_n deviate from 1 but only for one sample time at n_0;

$$G_{n_0} = \tan(\omega_{n_0}/2)\ /\ \tan(\omega_{n_0\text{-}1}/2) \qquad \text{(oscG2)}$$

We then have the new initial states from (osc0) and (oscG):

$$\omega_{n_0\text{-}1} = \omega_0$$
$$y_1[n_0] = \cos(n_0\,\omega_{n_0\text{-}1})$$
$$y_{2G}[n_0] = \tan(\omega_{n_0}/2)\ \sin(n_0\,\omega_{n_0\text{-}1}) \qquad \text{(osc1)}$$

Using the new initial states (osc1), we find from (oscSC) that,

$$y_1[n] = \cos(n_0\,\omega_{n_0\text{-}1})\ \cos\big((n\text{-}n_0)\omega_{n_0}\big) - \sin(n_0\,\omega_{n_0\text{-}1})\ \sin\big((n\text{-}n_0)\omega_{n_0}\big)$$
$$y_2[n] = \cos(n_0\,\omega_{n_0\text{-}1})\ \tan(\omega_{n_0}/2)\ \sin\big((n\text{-}n_0)\omega_{n_0}\big) + \tan(\omega_{n_0}/2)\ \sin(n_0\,\omega_{n_0\text{-}1})\ \cos\big((n\text{-}n_0)\omega_{n_0}\big)$$
$$;\text{for } n=n_0 \rightarrow \infty \qquad \text{(osc2)}$$

Equation (osc2) simplifies by trigonometric identity to,

$$y_1[n] = \cos\big(n_0\,\omega_{n_0\text{-}1} + (n\text{-}n_0)\omega_{n_0}\big)$$
$$y_2[n] = \tan(\omega_{n_0}/2)\ \sin\big(n_0\,\omega_{n_0\text{-}1} + (n\text{-}n_0)\omega_{n_0}\big) \qquad ;\text{for } n=n_0 \rightarrow \infty \qquad \text{(osc3)}$$

In both outputs the phase is correct in light of the new starting time. Comparing (osc3) to (osc0), we see that the amplitude of $y_1[n]$ remains as it was which is the desired result. To successfully change the frequency again and again, the same procedure can be repeated to derive similar results. But the amplitude of $y_2[n]$ has changed from its original value of $\tan(\omega_0/2)$. So, in this particular example we have not maintained the amplitude of $y_2[n]$ although we have managed to keep the amplitude of $y_1[n]$ constant throughout the process of changing the frequency of oscillation.

It is interesting to note that Smith and Cook originally derived the same result using principles of energy conservation across transformer-coupled waveguides.

stability
The Smith/Cook oscillator is hyperstable because there is only one tuning coefficient, $\cos(\omega_n)$. The equation for the poles in Figure SC shows that even under coefficient quantization, the pole magnitude is always unity.[101] Like the direct form, however, the single tuning coefficient requires high resolution at very low frequencies of oscillation. Inaccuracy in the representation of G_n when it deviates cannot affect the tuning frequency.

truncation noise
We will only permit speculation regarding the noise performance of the Smith/Cook oscillator as compared to the (Gordon/Smith) second modified coupled form, as we have no actual measurements of ***THD+N*** to bolster any analytical findings.

Recalling our discussion '*purity of direct vs. coupled form*', there the importance of zeroes in the deterministic noise transfer function was revealed. We have not been able to devise an implementation that would place a zero into the steady state truncation noise transfer (amplitude coefficient G_n set to 1) while maintaining freedom from amplitude deviation across a change in frequency.[102]

Hence we speculate that the noise performance of the Smith/Cook sinusoidal oscillator is not as good as that of the Gordon/Smith.

miscellany
Some other articles relevant to the field of oscillator design are [Fliege/Wintermantel] and [Thoen].

[101]Like the direct form, any instability in the sinusoidal waveform can only be attributed to signal quantization effects, primarily in the form of truncation error in this recursive topology.
[102]This topic is worthy of further research, however. One complicating circumstance is the potentially large disparity in amplitude between the two outputs, as shown by (oscSC).

4. External Memory Host Access (Paradigm and Utility)

The ESP2 does not provide direct access of external memory data by the system host at run-time, nor during halt, nor suspension. We present here a typical scheme for vectoring a running ESP2 application to a simple utility program which makes the desired access at the sample rate.[103] The ESP2 acts as intermediary.

We create a new program paradigm for a sample-synchronous application beyond which the utility will be overlaid at run-time by a relocating downloader driven by the system host.[104] We need a relocating downloader in order to select the start of the utility anywhere within instruction memory.

We write the utility so that it needs to know as little as possible about the running application. But we do need to know the relative address offset within the particular region of external memory to which to begin access, and we need a few registers globally reserved for the utility program. This information can be shared by all running applications, and we conveniently place it in an include file.

```
!************************ #include  file.h  contents ***************************
DEFCONST   GLOBAL
        CLK = 40.e6                         ! ESP2 system clock.
        Fs = 44100.                         ! system sample rate.
        T_LOCALE = $800000                  ! fixed but arbitrary, agreed on by convention.
        OVERLAY = 10                        ! overlay-instruction budget .
        AUDIO = 3*Fs                        ! 3 seconds.
        ESP_HALT = $2                       ! see HOST_CNTL_SPR in Chip Spec.

DEFREGION  T   @T_LOCALE   GLOBAL           ! region desired for host/external memory transfer.
    capture[2*AUDIO]                        ! A few seconds of stereo audio.
    sigmoid_table[257]                      ! Table used by a lot of programs.
    inverse_table[512]
    xmp  @$2ff                              ! external  memory  pointer used by utility.

DEFGPR  GLOBAL
     host_cntl_msk = ESP_HALT  @$ff         ! used to mask  HOST_CNTL_SPR.
     indirb  @$fe                           ! used to save  INDIRB  of application.
!*********************************************************************
```

[103]Note that the elimination of the BIOZ instruction from the application program paradigm, presented shortly, would permit the host external memory access at a much higher rate.
[104]The download, by the system host, of new instructions into ESP2 instruction memory always occurs at the instruction rate.

Resource Management
We assume that any DIL or DOL required by the utility's AGEN operations will be free at the instant that the utility is executed. This is a reasonable assumption and allows sharing of that resource.

The utility can use a few instructions in any unused instruction space. Recall that at a sample rate of 44.1 kHz, only 226 ESP2 instruction lines can be executed per sample period. So, there will likely be some instructions free that we will not need to take away from the application resource.

We will reserve two GPRs; one to hold a semaphore mask (host_cntl_msk) and one to save the application's return address (indirb). Finally, we will need a dedicated and reserved AOR (xmp) which, together with the region BASE*r*, will point to the desired location in external memory. The AOR, xmp, is a relative address offset into the region, T in this case. It can be initialized, by the host, to the root address offset of any one of the declared external memory arrays.
These three registers will only need to be initialized once throughout the use of the utility.

4.1. ESP2 External Memory Host Access Application Program Paradigm

```
! JonD, ESP2, 1/1/95.

PROGRAM  Transfer

#include  file.h

PROGSIZE  <= INT(CLK/(4.*Fs)) - OVERLAY    !compensation for execute time, not size

DEFSPR     LOCAL
  INDIRB = application
  PC = init
  REPT_CNT = 0
  SER_CONF = $007fff                                        ! see the Chip Spec.
  HARD_CONF = $008400

CODE
!********************* begin application ***************************************
init:           NOP
                 .
application: NOP                                            ! JMP and return here
                 .
                 .
                 .
                NOP                 JMP  INDIRECT
                NOP                 BIOZ
!******************* end application program ************************************
```

Procedure
We assume that this application has been running for some time. The SPR INDIRB holds the PC value of the label called ***application*** as evidenced by the declarations. The host queries this register and saves its contents in the global GPR called indirb. The host takes the ESP_HALT_EN and ESP_HALT bits, of the HOST_CNTL interface register, low.

After the host overlays the utility, it overwrites INDIRB with the desired PC location of the start of the utility. When the JMP is encountered thereafter, it will be vectored to our little utility. The host initializes the AOR, xmp, to the desired address offset into the region T in external memory (see the chart in the Discussion).

When a semaphore is taken high by the host, the utility will be activated for a single stereo access. The utility resets the semaphore. When the host is finished accessing external memory, it will set INDIRB back to its original contents thus bypassing the utility.

4.2. ESP2 External Memory Host Access Utility

```
! JonD, ESP2, 1/2/95.

PROGRAM  Utility

#include  file.h

PROGSIZE  <= OVERLAY

CODE
!************************** begin WR utility *******************************
utility:NOP                                AND host_cntl_msk, HOST_CNTL_SPR > ZERO
        NOP                                JS  push, NZ > CMR
        MOV  indirb > PCSTACK0            {ADDV  host_cntl_msk, xmp}
push:{MOV  HOST_GPR_DATA > *xmp(+)}        RS, NZ > CMR
     {MOV  HOST_ESP_FACE > *xmp}          {XOR  host_cntl_msk, HOST_CNTL_SPR}
!**************************** end utility **********************************
```

Discussion of Utility

The JS in conjunction with the load of PCSTACK0 constitutes a push of indirb onto the hardware stack. The GPR, indirb, holds the PC value of the label ***application***, and was loaded by the system host during initialization of this procedure. This is done so that RS will know where to return to within the application. JS and RS each have one instruction cycle execution latency, and they are unconditionally executed. This little utility is called at the sample rate as part of the running application which is never stalled.

The only unconventional usage within the utility is of HOST_CNTL_SPR. This register is unusual in so far as it appears simultaneously in the host interface register space **(**as HOST_CNTL**)** and in the ESP2 SPR space. This fact makes that register desirable for fast inter-communication. We sacrifice one of the bits of that register, namely the ESP_HALT bit, for use as the semaphore. This is possible if the host holds the ESP_HALT_EN bit low and if we agree that there are no HALT pseudo instructions, within the application, that we need activated during this time.

The HOST_GPR_DATA and HOST_ESP_FACE SPRs are the only other registers sharing the attribute of simultaneous appearance in the SPR and host interface register spaces. This design yields the ultimate speed in register data transfer because the host does <u>not</u> need to go through the standard host/ESP2-register interface to transfer data somewhere that the ESP2 can see it directly; namely, from/to these two SPRs. This means that there is no need for the execution of BIOZ or HOST instructions in order that the host be able to communicate with a running ESP2 program via <u>these</u> three registers; and vice versa.[105]

The HOST_ESP_FACE SPR is uncommitted, but the HOST_GPR_DATA SPR is an integral part of the host interface for internal register access. We employ both these registers in our utility because we want stereo data access. Use of HOST_GPR_DATA here does not preclude standard use of the host/ESP2-register interface, however; this is because of the semaphore protocol.

As shown, the utility writes stereo data out to external memory. The stereo data are presumed stored in an interleaved fashion **(**Left channel, Right channel, Left...**)**, so we employ to our advantage the AGEN's Plus-One addressing mode (+). The two conditional external memory WR are scheduled on the same respective instruction lines in the AGEN as the lines of requesting MAC code.[106] This means that the scheduled conditional AGEN code experiences the same CCR and CMR as the requesting MAC code.

[105]Therefore, we could eliminate the synchronization of this utility to the BIOZ suspension in the application program so that external memory data transfer would exceed the sample rate.

[106]We could insure this via the assembler's forced WR directive, => , but that is not necessary for these isolated five lines because there is little external memory traffic.

The GPR, host_cntl_msk, performs double duty as a semaphore mask and as the increment for our stereo data pointer, the AOR called xmp.   The increment to xmp will become effective for the second current external memory WR because of the assembler scheduling in the AGEN.

All four last lines of code in the utility, in fact, see the same CCR and CMR. For this reason, the utility doubles for both RD and WR access. To make the utility perform RD access, the source and destination in each of the last two lines of MAC unit code are interchanged. Because of the scheduling, in that case, xmp will not increment in time to affect the two current external memory RD in the AGEN.

Given these considerations we have the following chart for the proper use of this utility:

	HOST_GPR_DATA	**HOST_ESP_FACE**	**xmp** initialization
WR	Left channel data	Right channel data	address offset - 1
RD	Right channel data	Left channel data	address offset

There is a potential **hazard** with the host interface when using either of these two SPRs as MAC unit destination in the RD utility. This hazard comes about because of each SPRs' simultaneous appearance in two register spaces. Once again, the hazard is avoided by the semaphore protocol; the conditionally executed instructions inhibit the destination unless the semaphore bit is high. Notice the efficiency we have gained through the use of conditionally executable code, denoted by {}. Without it, we would have the overhead of jumping around it.

5. FFT Radix-4

5.1. FFT Radix-4 C-Program Model

```
/* This is a validation test of the [Burrus/Parks,pg.113] Radix-4 FFT - JonD 1/15/93 */
/* This is an in-place algorithm. */
/* Modified for zero-indexing. */

#include <stdio.h>
#include <math.h>

#define FREQ  4                             /* power of two < 64, to get integral period */
                                            /* and to keep epsilon q23 */
#define N     256
#define RADIX   4
#define TWIDDLE_SIZE    2*N

double roundint();
double pow2to23, RoundInt24(), TruncInt24();

void main() {

/* FFT */
int N1, N2, M, IE, IA1, IA2, IA3, I1, I2, I3, I, J, K;
long templ;
double CO1, CO2, CO3, SI1, SI2, SI3, XT, YT, R1, R2, R3, R4, S1, S2, S3, S4;
double R1b, R2b, R3b, S1b, S2b, S3b;
double X[N], Y[N];
/* TWIDDLE */
double twopi, Wreal[N], Wimag[N];
/* INPUT SIGNAL */
double epsilon, yn, yqn;
double pi;

/* init */
M = roundint(log((double)N)/log((double)RADIX));
twopi = 8.*atan(1.);
pi    = 4.*atan(1.);
pow2to23 = pow(2.0, 23.0);
```

```
/**************** GENERATE TWIDDLE FACTORS ***************/
/* first value is: 0x7FFFFF */ /* first interleaved value is: 0x000000 */
/* last value is: 0x7FF622 */ /* last interleaved value is: 0xFCDBD5 */
printf("Twiddle factor table size is: %d\n", TWIDDLE_SIZE);
/*for(K=0; K<N; K++)   {
        Wreal[K] = RoundInt24(cos((K*twopi)/N));
        templ = pow2to23*Wreal[K];
        if(templ >= 0x00800000L) {
                Wreal[K] = TruncInt24(1.0 - pow(2., -23.));
                templ = 0x7fffffL;
        }
        printf("%0.6lx\n", templ & 0xffffffL);

        Wimag[K] = RoundInt24(sin((K*twopi)/N));
        templ = pow2to23*Wimag[K];
        if(templ >= 0x00800000L) {
                Wimag[K] = TruncInt24(1.0 - pow(2., -23.));
                templ = 0x7fffffL;
        }
        printf("%0.6lx\n", templ & 0xffffffL);
}*/  /* 24-bit fixed-point, q23 */

for(K=0; K<N; K++) {  /* floating-point */
        Wreal[K] = cos((K*twopi)/N);
        Wimag[K] = sin((K*twopi)/N);
}
/*************** end TWIDDLE ********************************/

/*** GENERATE INPUT SIGNAL ********************************/
epsilon = 2.*sin((pi*FREQ)/N);
y_qn = 0.0;
yn  = -cos((pi*FREQ)/N);

for(K=0; K<N; K++) {
      X[K] = y_qn/2.;  /* fixed-point overflow prevention */
      Y[K] = 0.0/2.;
      y_qn -= epsilon*yn;
      yn  += epsilon*y_qn;
}
/*************** end INPUT *******************************/
/*
for(I=0; I<N; I++)
        printf("X[%d] = %lf\t  Y[%d] = %lf\n", I, X[I], I, Y[I]);
*/
```

```
/********************* THE FFT *******************************/
N2 = N;
IE = 1;
for(K=0; K<M; K++) {
        N1 = N2;
        N2 = N2 >> 2;
        IA1 = IA2 = IA3 = 0;
        for(J=0; J<N2; J++) {
                CO1 = Wreal[IA1];
                CO2 = Wreal[IA2];
                CO3 = Wreal[IA3];
                SI1 = Wimag[IA1];
                SI2 = Wimag[IA2];
                SI3 = Wimag[IA3];
                I1 = J  + N2;
                I2 = I1 + N2;
                I3 = I2 + N2;
                for(I=J; I<N; I+=N1) {
                        R1 = (X[I] + X[I2])/(RADIX/2);
                        R3 = (X[I] - R1);
                        S1 = (Y[I] + Y[I2])/(RADIX/2);
                        S3 = (Y[I] - S1);
                        R2 = (X[I1] + X[I3])/(RADIX/2);
                        R4 = (X[I1] - R2);
                        S2 = (Y[I1] + Y[I3])/(RADIX/2);
                        S4 = (Y[I1] - S2);
                        R2b   = R1 - R2;
                        S2b   = S1 - S2;
                        R1b   = R3 - S4;
                        S1b   = S3 + R4;
                        R3b   = R3 + S4;
                        S3b   = S3 - R4;
                        X[I2] = (CO2*R2b + SI2*S2b)/(RADIX/2);
                        Y[I2] = (CO2*S2b - SI2*R2b)/(RADIX/2);
                        X[I3] = (CO3*R1b + SI3*S1b)/(RADIX/2);
                        Y[I3] = (CO3*S1b - SI3*R1b)/(RADIX/2);
                        X[I1] = (CO1*R3b + SI1*S3b)/(RADIX/2);
                        Y[I1] = (CO1*S3b - SI1*R3b)/(RADIX/2);
                        X[I] = (R1 + R2)/(RADIX/2);
                        Y[I] = (S1 + S2)/(RADIX/2);
                        I1 += N1;
                        I2 += N1;
                        I3 += N1;
                }
                IA1 += IE;
                IA2 = IA1 + IA1;
                IA3 = IA2 + IA1;
        }
        IE = IE << 2;
}       /************** end FFT *****************/
```

```
/******************** Digit Reversal Address/Input Scaling Compensation *******/
J = 0;
N1 = N - 1;
for(I=0; I<N1; I++) {
                if(I >= J) goto label101;
                XT   = X[J];
                X[J] = X[I];
                X[I] = XT;
                YT   = Y[J];
                Y[J] = Y[I];
                Y[I] = YT;
label101:       K = N/RADIX;
label102:       if(K*3 > J) goto label103;
                        J = J - K*3;
                        K /= RADIX;
                        goto label102;
label103:       J += K;
}

for(I=0; I<N; I++) {    /* compensation for earlier fixed-point overflow prevention */
         X[I] *= 2.;
         Y[I] *= 2.;
}
/*********************************** end BIT SWAP ****************/
for(I=0; I<N; I++) {
        if((fabs(X[I]) >= 1e-16) || (fabs(Y[I]) >= 1e-16))
            printf("X[%d] = %.16lf\t  Y[%d] = %.16lf\n", I, X[I], I, Y[I]);
}
}

/******************** subroutines **********************************/
/******** replacement for rint() on NeXT O.S. v2.1 *******/
double roundint(double x)
{
if(x >= 0.)return((double)((int)(x + 0.5)));
return((double)((int)(x - 0.5)));
}

double TruncInt24(x)
double x;
{
return(floor(pow2to23*x)/pow2to23);
}

double RoundInt24(x)
double x;
{
return(roundint(pow2to23*x)/pow2to23);
}
```

5.2. FFT Radix-4 ESP2 Program

```
! FFT Radix-4, single precision.  - JonD February 18, '93 .
! 53 ESP2 instructions.  1024 point complex input @3.245 ms @10 MHz instruction rate.
! Execution 7 times faster than real time @Fs = 44100 Hz.
! THD+N = approx 110 dB regardless of input signal amplitude @ normalized frequency = 4/N .
! Butterflys are scaled so that FFT of full-scale sinusoid produces +/- $400000
! System host must set the ESP_HALT_EN bit of the HOST_CNTL interface register.

PROGRAM  FFT4

PROGSIZE = 64          !  - 11

DEFCONST
     XI =  #DILF                YI3 = #DIL8
     XI2 = #DILE                CO1 = #DIL7
     YI =  #DILD                CO2 = #DIL6
     YI2 = #DILC                CO3 = #DIL5
     XI1 = #DILB                SI1 = #DIL4
     XI3 = #DILA                SI2 = #DIL3
     YI1 = #DIL9                SI3 = #DIL2
     RADIX = 4
     N = 256        ! can't be 128
     M = INT(LOG(N)/LOG(RADIX))
     FREQ = 4
     Pi = 4.0*ATAN(1.0)
     EPSILON = 2.0*SIN((Pi*FREQ)/N)
     BIG_ONE = (2**24)/N
     REGION_R_LOC = $800000
     REGION_I_LOC = $800400               REGION_T_LOC = $800800

DEFSPR
   PC = begin
   REPT_CNT = 0                       DILA   DILB   DILC   DILE   DILF
   SER_CONF = $007fff                 DIL6   DIL7   DIL8   DIL9
   HARD_CONF = $008400                DIL2   DIL3   DIL4   DIL5

DEFGPR
   GLOBAL
       epsilon = EPSILON
       yqn = 0
       yn  = -COS((Pi*FREQ)/N)
   LOCAL
       J      IE     R1  R2  R3  R4   S1  S2  S3  S4   R1b  R2b  R3b   S1b  S2b  S3b
       N2     count
       Irev   IA1  IA2  IA3
```

```
DEFREGION  T  @REGION_T_LOC  ! TABLE REGION
        Twiddle[2*N]                ! holds interleaved single precision (cos(), sin()) table;
                                    ! loaded by system host at download.  0 root required.
        IA1_doub                    ! AORs
        IA2_doub
        IA3_doub

DEFREGION  R  @REGION_R_LOC  ! DELAYLINE  REGION
        XinArray[N]                 ! real input, single precision, q23.  0 root required.
        Xoutput[N]                  ! output array, q23.  Loaded post digit-reversal.
        XoutPointer
        RevPointer                  ! digit reverse pointer.
        K  @$2f9                    ! default region is R; used in region I
        I        I1        N1       ! don't init any AORs to 0; confusion later with XinArray[0]
        I2       I3
        Ir       I1r                ! to avoid wasting another 2 regions
        I2r      I3r

DEFREGION  I  @REGION_I_LOC
        YinArray[N]                   ! imaginary input, q23.  0 root required.
        Youtput[N]                    ! output array, q23.  Loaded post digit-reversal.
                                      ! Must be in same relative position as Xoutput[N]
 !********************** END declarations ******************************************!

CODE
!******************* generate input signal *****************************!
begin: CLR K                        CLR I

!*** initialize input arrays ***
signal_loop: CLR *(K)I              HALVE  yqn  > *(K)R   !overflow prevention in first butterfly

!****** hyperstable near-quadrature oscillator (second modified coupled form) ******
MOV  yqn > MACP                     MOV yn > MACP
MACP - epsilon X yn  > yqn
MACP + epsilon X yqn > yn
!*** end oscillator ***

!*** go around loop simple way ***
NOP                                 ADDV  ONE, K
NOP                                 CMP  K, #N
NOP                                {JMP signal_loop, NEQ > CMR}
NOP
!****************************** end input signal generation ************!
```

```
!************************** begin the FFT ********************************
fft: MOV ONE > IE                              MOV #N > N2
MOV #REGION_R_LOC > BASER                      CLR K

K_loop:MOV N2 > N1                             CLR IA1
CLR J                                          LSH  N2 >>2
CLR IA2                                        CLR IA3

   J_loop:
   MOV N2 > MACP                 MOV #REGION_I_LOC > BASEI
   ADD N2, MACP  > I2            ADDV  J, BASEI               RD *(I )R  > XI          !XinArray
   ADD I2, MACP  > I3            MOV N2 > I1                  RD *(I2)R > XI2
   NEG ONE > MACP                SUBV N1, SIZEM1R > MACP      RD *(I )I   > YI         !YinArray
   ADD IE, MACP  > count         AVG XI, XI2   > R1           RD *(I2)I  > YI2
   ADD ONE, MACP                 SUB R1, XI     > R3          RD *(I1)R > XI1
   NOP                           AVG YI, YI2   > S1           RD *(I3)R > XI3
   NOP                           SUB S1, YI      > S3         RD *(I1)I  > YI1
   DBL IA2 > IA2_doub            AVG XI1, XI3 > R2            RD *(I3)I  > YI3
   DBL IA3  > IA3_doub           SUB R2, XI1   > R4           RD *(IA2_doub)    > CO2
   DBL IA1  > IA1_doub           AVG YI1, YI3 > S2            RD *(IA2_doub(+)) > SI2
   NOP                           SUB S2, YI1    > S4          RD *(IA3_doub)    > CO3
   NOP                           SUB R2, R1 > R2b             RD *(IA3_doub(+)) > SI3
   NOP                           SUB S2, S1  > S2b            RD *(IA1_doub)    > CO1
   ADD I, MACP  > Ir             SUB S4, R3 > R1b             RD *(IA1_doub(+)) > SI1
   ADD I1, MACP  > I1r           ADD S3, R4 > S1b             RD *(BASE += N1)R  > XI        !ok
   ADD I2, MACP  > I2r           ADD R3, S4 > R3b             RD *(I2)R > XI2
   ADD I3, MACP  > I3r           SUB R4, S3  > S3b            RD *(BASE += N1)I   > YI       !ok

      NOP                          REPT I_loopEnd, count > REPT_CNT
      I_loop:
               R2b X CO2  > MAC             AVG R1, R2 > *(Ir)R      RD *(I2)I  > YI2
      MAC + S2b X SI2  >>1 > *(I2r)R        AVG S1, S2  > *(Ir)I     RD *(I1)R > XI1
              S2b X CO2  > MAC              AVG XI, XI2   > R1       RD *(I3)R > XI3
      MAC - R2b X SI2  >>1 > *(I2r)I        SUB R1, XI     > R3      RD *(I1)I  > YI1
               R1b X CO3  > MAC             AVG YI, YI2   > S1       RD *(I3)I  > YI3
      MAC + S1b X SI3  >>1 > *(I3r)R        SUB S1, YI      > S3
              S1b X CO3  > MAC              AVG XI1, XI3 > R2
      MAC - R1b X SI3  >>1 > *(I3r)I        SUB R2, XI1   > R4
               R3b X CO1  > MAC             AVG YI1, YI3 > S2
      MAC + S3b X SI1  >>1 > *(I1r)R        SUB S2, YI1    > S4
              S3b X CO1  > MAC              SUB R2, R1 > R2b
      MAC - R3b X SI1  >>1 > *(I1r)I        SUB S2, S1  > S2b
      NOP                                   SUB S4, R3  > R1b
      NOP                                   ADD S3, R4 > S1b         RD *(BASE += N1)R  > XI
      NOP                                   ADD R3, S4 > R3b         RD *(I2)R > XI2
      I_loopEnd:NOP                         SUB R4, S3  > S3b        RD *(BASE += N1)I   > YI
   MOV IA1 > MACP                                   ADDV ONE, J
   ADD IE, MACP  > MAC, IA1                         CMP  J, N2
   SHIFTMAC <<1  > IA2                             {JMP J_loop, LT > CMR}
   ADD IA2, MAC  > IA3                              ADDV J, #REGION_R_LOC > BASER

MOV #REGION_R_LOC > BASER            CMP  K, #M-1
ASH IE  <<2                         {JMP K_loop, LT > CMR}
MOV #REGION_I_LOC > BASEI            ADDV ONE, K

!************* digit reversal / undo overflow prevention in first butterfly ***********************!
MOV &XinArray[0] > RevPointer                 MOV &Xoutput[0] > XoutPointer
CLR Irev                                      REPT  reversal, N-1
            NOP                               ADDV #BIG_ONE, Irev       RD *(RevPointer)R > XI
            NOP                               DREV Irev > RevPointer    RD *(RevPointer)I > XI2
            DBL XI   => *(XoutPointer)R
reversal: DBL XI2  => *(XoutPointer)I         ADDV ONE, XoutPointer
!******************************** end digit rev **************************************!

!*************************** end FFT *************************************!
NOP                                           HALT
NOP
!*****************************************************************************!
```

5.3. FFT Radix-4
C-Program Simulation of ESP2 Program

```
/* reference file: fft4.c */
/* cd /jond/esp2/Csource */
/* This is a single precision fixed point (48-bit) rendering for ESP2 simulation. -Jon Dattorro 1993 */
/* Each stage of butterflies is scaled by 4 to prevent overflow.  A sinusoid
   will then have standard power level as output from scaled DFT (scaled by factor N).
   The complex input signal is stored in a single precision array.
*/

#include <stdio.h>
#include <math.h>
double pow(), atan(), log(), log10(), sin(), cos(), sqrt(), floor();

double pow2to46, select_lower24(), lower24();
double pow2to47, RoundInt48(), TruncInt48(), TruncInt48MAC();
double pow2to23, RoundInt24(), TruncInt24();
double pow2to15, TruncInt16(), RoundInt16(), roundint();
double myfmod();

#define NMAX  4096
#define FREQ  4                       /* integer, empirical value. Period is always N/FREQ */
#define RADIX 4
#define TWIDDLE_LOC 0x0
#define TWIDDLE_SIZE 2*N

void main() {

/* FFT */
char buf[32];
int N1, N2, M, IE, J, K, N;
int I, I1, I2, I3, IA1, IA2, IA3;
double CO1, SI1, CO2, SI2, CO3, SI3;
double R1, R2, R3, R4, S1, S2, S3, S4;
double R2b, R3b, R1b, S2b, S3b, S1b;
double X[NMAX], Y[NMAX], preswapX[NMAX], preswapY[NMAX], XT;
double noise_power;
/* TWIDDLE */
long templ;
double twopi, WR[NMAX], WI[NMAX];
/* INPUT SIGNAL */
double pi, amplitude, sinamp, sinamp_dB;
```

```
/*printf("Input number of points: ");
gets(buf);  sscanf(buf, "%d", &N);

printf("Input amplitude of sinusoid (0. -> -90. dB): ");
gets(buf);  sscanf(buf, "%lf", &sinamp_dB);
sinamp = pow(10., sinamp_dB/20.);
*/
sinamp = 1.0;
N = 256;
M = roundint(log((double)N)/log((double)RADIX));
twopi = 8.*atan(1.);
pi    = 4.*atan(1.);
pow2to47 = pow(2.0, 47.0);
pow2to46 = pow(2.0, 46.0);
pow2to23 = pow(2.0, 23.0);
pow2to15 = pow(2.0, 15.0);

/**************** GENERATE TWIDDLE FACTORS ********************************/
/*printf("%0.6lx\t\t", TWIDDLE_LOC);*/
/*printf("%d\n", TWIDDLE_SIZE);*/
for(K=0; K<N; K++)
                   {
                   WR[K] = RoundInt24(cos((K*twopi)/N));
                   templ = pow2to23*WR[K];
                   if(templ >= 0x00800000L) {
                         WR[K] = TruncInt24(1.0 - pow(2., -23.));
                         templ = 0x7fffffL;
                                       }
                   /*printf("%0.6lx\n", templ & 0xffffffL);*/  /* redirect std output to get */
                                                           /* ascii file */
                   WI[K] = RoundInt24(sin((K*twopi)/N));
                   templ = pow2to23*WI[K];
                   if(templ >= 0x00800000L) {
                         WI[K] = TruncInt24(1.0 - pow(2., -23.));
                         templ = 0x7fffffL;
                                       }
                   /*printf("%0.6lx\n", templ & 0xffffffL);*/
                   }
/*************** end TWIDDLE *********************************/
```

```
/*** GENERATE INPUT SIGNAL ********************************/
amplitude = (pow2to23-1.)/pow2to23;
for(K=0; K<N; K++)
                   {
                   X[K] = TruncInt24(0.5*sinamp*amplitude*sin(myfmod((twopi*FREQ*K)/N, twopi)));/* scale ovf in 1st butterfly.*/
                   Y[K] = 0.0;                                                        /* Use of RountInt yields better S/N */
/*                 printf("%0.6lx\n", (long)(pow2to23*X[K]) & 0xffffffL);*/  /* redirect std output to get */
                                                                        /* ascii file */
                   }
/*************** end INPUT *********************************/
/********************* THE FFT *******************************/
N2 = N;
IE = 1;
for(K=0; K<M; K++)
                   {
                   N1 = N2;
                   N2 = N2 >> 2;
                   IA1 = IA2 = IA3 = 0;
                   for(J=0; J<N2; J++)
                         {
                         CO1 = WR[IA1];
                         CO2 = WR[IA2];
                         CO3 = WR[IA3];
                         SI1 = WI[IA1];
                         SI2 = WI[IA2];
                         SI3 = WI[IA3];
                         I1 = J  + N2;
                         I2 = I1 + N2;
                         I3 = I2 + N2;
                         for(I=J; I<N; I+=N1)
                               {
                               R1 = TruncInt24((X[I] + X[I2])/(RADIX/2));   /* scale by 4 for */
                               R3 = TruncInt24(X[I] - R1);                  /* each           */
                               S1 = TruncInt24((Y[I] + Y[I2])/(RADIX/2));   /* stage of       */
                               S3 = TruncInt24(Y[I] - S1);                  /* butterflies    */
                               R2 = TruncInt24((X[I1] + X[I3])/(RADIX/2));
                               R4 = TruncInt24(X[I1] - R2);
                               S2 = TruncInt24((Y[I1] + Y[I3])/(RADIX/2));
                               S4 = TruncInt24(Y[I1] - S2);

                               R2b = TruncInt24(R1 - R2);
                               S2b = TruncInt24(S1 - S2);
                               R1b = TruncInt24(R3 - S4);
                               S1b = TruncInt24(R4 + S3);
                               R3b = TruncInt24(S4 + R3);
                               S3b = TruncInt24(S3 - R4);
/*                          if (K == 1)
                            {
                                  printf("x[%d] = %lx x[%d] = %lx x[%d] = %lx x[%d] = %lx\n",I,(int)floor(pow2to23*X[I]),
                                  I1,(int)floor(pow2to23*X[I1]),I2,(int)floor(pow2to23*X[I2]),I3,(int)floor(pow2to23*X[I3]));
                                  printf("y[%d] = %lx y[%d] = %lx y[%d] = %lx y[%d] = %lx\n",I,(int)floor(pow2to23*Y[I]),
                                  I1,(int)floor(pow2to23*Y[I1]),I2,(int)floor(pow2to23*Y[I2]),I3,(int)floor(pow2to23*Y[I3]));
                            }
*/
                            X[I2] = TruncInt24((TruncInt48MAC(CO2*R2b) + TruncInt48MAC(SI2*S2b))/(RADIX/2));
                            Y[I2] = TruncInt24((TruncInt48MAC(CO2*S2b) - TruncInt48MAC(SI2*R2b))/(RADIX/2));

                            X[I3] = TruncInt24((TruncInt48MAC(CO3*R1b) + TruncInt48MAC(SI3*S1b))/(RADIX/2));
                            Y[I3] = TruncInt24((TruncInt48MAC(CO3*S1b) - TruncInt48MAC(SI3*R1b))/(RADIX/2));

                            X[I1] = TruncInt24((TruncInt48MAC(CO1*R3b) + TruncInt48MAC(SI1*S3b))/(RADIX/2));
                            Y[I1] = TruncInt24((TruncInt48MAC(CO1*S3b) - TruncInt48MAC(SI1*R3b))/(RADIX/2));

                            X[I ] = TruncInt24((R1 + R2)/(RADIX/2));
                            Y[I ] = TruncInt24((S1 + S2)/(RADIX/2));

/*                          if (K == 1)
                            {
                            printf("xout[%d] = %lx xout[%d] = %lx xout[%d] = %lx xout[%d] = %lx\n",I,(int)floor(pow2to23*X[I]),
                                  I1,(int)floor(pow2to23*X[I1]),I2,(int)floor(pow2to23*X[I2]),I3,(int)floor(pow2to23*X[I3]));
                            printf("yout[%d] = %lx yout[%d] = %lx yout[%d] = %lx yout[%d] = %lx\n",I,(int)floor(pow2to23*Y[I]),
                                  I1,(int)floor(pow2to23*Y[I1]),I2,(int)floor(pow2to23*Y[I2]),I3,(int)floor(pow2to23*Y[I3]));
                            }
*/                                I1 += N1;
                                  I2 += N1;
                                  I3 += N1;


                                  }
                            IA1 += IE;
                            IA2 = IA1 + IA1;
                            IA3 = IA2 + IA1;
                            }
                      IE = IE << 2;
        }

/************************************ end FFT *********************/
for(I=0;I<N;I++)
{
                      preswapX[I] = X[I];
                      preswapY[I] = Y[I];
}
```

```
/********************* BIT SWAP/INPUT SCALING COMPENSATION *******/
J = 0;
N1 = N - 1;
for(I=0; I<N1; I++)
                   {
                    if(I >= J) goto label101;
                    XT   = X[J];
                    X[J] = X[I];
                    X[I] = XT;
                    XT   = Y[J];
                    Y[J] = Y[I];
                    Y[I] = XT;
label101:K = N/RADIX;
label102:if(K*3 > J) goto label103;
                          J -= K*3;
                          K /= RADIX;
                          goto label102;
label103:J += K;
                   }

for(I=0; I<N; I++)
                   {
                    X[I] *= 2.;
                    Y[I] *= 2.;
                   }
/*********************************** end BIT SWAP ***************/

/**************************************** STATISTICS ********************/
noise_power = 0;
for(I=0; I<N; I++)
                   {
                   if((fabs(X[I]) < pow(2., -23.)) && (fabs(Y[I]) < pow(2., -23.)));
                   else  {
                         printf("X[%d] = %.16f\t  Y[%d] = %.16f\n", I, X[I], I, Y[I]);
                         templ = floor(X[I]*pow2to23);
                         printf("integerX = %x\t\t\t", templ & 0xfffffeL);
                         templ = floor(Y[I]*pow2to23);
                         printf("    integerY = %x\n", templ & 0xfffffeL);
                   }
                   if((I == FREQ) || (I == (N-FREQ)))    /* sine (not cosine) wave is assumed */
                      noise_power += X[I]*X[I];
                   else noise_power += X[I]*X[I] + Y[I]*Y[I];
                   }
noise_power += pow((0.5*amplitude*sinamp + Y[FREQ]), 2.) + pow((0.5*amplitude*sinamp - Y[N-FREQ]), 2.);
printf("signal/noise w/r full scale is: %.13f dB\n", -10.*log10(noise_power/(0.5*amplitude*amplitude)));
/**********************************************************************/
```

```
for(I=0; I<N; I++)
                   {
/*                        templ = floor(preswapX[I]*pow2to23);
                          printf("integerX[%d] = %x\t\t\t",I, templ & 0xffffffL);

                          templ = floor(preswapY[I]*pow2to23);
                          printf("    integerY[%d] = %x\n",I+513, templ & 0xffffffL);

                          templ = floor(X[I]*pow2to23);
                          printf("integerX[%d] = %x\t\t\t",I, templ & 0xffffffL);

                          templ = floor(Y[I]*pow2to23);
                          printf("    integerY[%d] = %x\n",I, templ & 0xffffffL);
*/
                   }
/**************************************************************************************************************/
}

/****************************** SUBROUTINES ****************************/
double RoundInt48(x)
double x;
{
return(roundint(pow2to47*x)/pow2to47);
}

double RoundInt24(x)
double x;
{
return(roundint(pow2to23*x)/pow2to23);
}

double RoundInt16(x)
double x;
{
return(roundint(pow2to15*x)/pow2to15);
}

double TruncInt48(x)
double x;
{
return(floor(pow2to47*x)/pow2to47);
}

double roundint(double x) /*** replacement for rint() ***/
{
if(x >= 0.)return((double)((int)(x + 0.5)));
return((double)((int)(x - 0.5)));
}
```

```
double TruncInt48MAC(x)
double x;
{
return(floor(pow2to46*x)/pow2to46);
}

double TruncInt24(x)   /* This emulates initialization in esp2 assembler */
double x;
{
return(floor(pow2to23*x)/pow2to23);
}

double TruncInt16(x)
double x;
{
return(floor(pow2to15*x)/pow2to15);
}

double select_lower24(x)  /* also loses the LSB; the 48th bit */
double x;
{
double TruncInt24(), tempd;

tempd = x - TruncInt24(x);
return(floor(pow2to46*tempd)/pow2to46);
}

double lower24(x)
double x;
{
double TruncInt24(), tempd;

tempd = x - TruncInt24(x);
return(floor(pow2to47*tempd));
}

/****************** MYFMOD function ******************************************/
/*** used to increase accuracy of sin routines ***/
/*** fmod() uses floor() instead of (int). ***/
/*** fmod() also failing to return values within |modulo| ***/
/*** Want (int), which is magnitude truncation, for negative arguments. ***/
double myfmod(argument, modulo)
register double argument, modulo;
{
return(argument - (int)(argument/modulo)*modulo);
}
/********************************** END subroutines ****************************/
```

5.4. Comments on the FFT Radix-4 Program

The *Twiddle factor* [Opp/Sch] [AnalogDevices] table must be loaded into external memory before the ESP2 program is run. To load external memory, another ESP2 Application must be run. See the External Memory Host Access Application. The C code for the Twiddle factor table generation is given within the C-program model for the FFT itself. The complex Twiddle factor table is interleaved; cosine then sine, cosine, sine, etc. This is done so as to use to advantage the AGEN's automatic Plus-One addressing mode (+). As a check, the C-program comments include the values of the first and last interleaved Twiddle factors.

Notice that in many instances, the programmer has chosen to explicitly write the AGEN code. The programmer has also managed to give the various DILs meaningful names through a DEFCONST declaration. It is important to declare those DILs in a DEFSPR so that the assembler relinquish those particular resources. If the programmer has designed the algorithm so that those DILs are free when the assembler might need them, then the declaration might be unnecessary.

The FFT portion of the ESP2 program is a translation of the C-program model. The separate C-program simulation estimates the ***S/N*** (signal to noise power ratio) of this single precision FFT Radix-4 circuit itself to be in excess of 110 dB as implemented within the ESP2.[107]

The conventional DFT (Discrete Fourier Transform [ibid.]) inherently amplifies all signals by a factor N, the DFT length. The conventional IDFT (Inverse DFT) must scale all the bin values by 1/N to recover the original input signal. In the algorithms[108] that we present, the responsibility of scaling is transferred to the FFT because of a high likelihood of bin overflow when using a fixed-point processor. The scaling of the complex input signal is distributed over each stage of *butterflys* to minimize truncation noise. This successive scaling yields a remarkable robustness, even for low-level input signals.

Another approach to scaling employs the block floating-point technique which can be implemented within a fixed-point processor, such as ESP2, having barrel shifters. This technique performs a signal-dependent scaling. [ibid.] [Kim/Sung]

[107]The sinusoid frequency used in the estimate is centered on an FFT *bin* frequency. [ibid.] Using an irrational frequency should not change the total noise power, but it will impact the character of the noise power spectrum.
[108]The FFT algorithm, in general, is simply a faster DFT. (But not fast enough!)

5.5. Changes Required to ESP2 Program to make the IFFT (Radix-4)

Complex Swap

We would like the simplest conversion possible; this means using the same Twiddle factor table. We will use a DSP trick; define the *swap* operation on the real and imaginary parts of a complex quantity:

$$\text{swap}(z) = j\,z^* = y + j\,x \qquad ; z = x + j\,y,\ \ j = \sqrt{-1}$$

The asterisk denotes complex conjugation. Then, using the classical definitions of the transform pair,

$$\text{IDFT}\{X[k]\} = (j / N)\,\big(\text{DFT}\{j\,X^*[k]\}\big)^*$$

$$= \text{swap}\big(\,(1/N)\ \ \text{DFT}\{\text{swap}(X[k])\}\,\big)$$

The proof of this follows from the fact that we can show that

$$\text{IDFT}\{X[k]\} = (1/N)\,\big(\text{DFT}\{X^*[k]\}\big)^*$$

This means that to get an IFFT (Inverse FFT) using an FFT algorithm, one must first swap the real and imaginary parts of the complex input record. After the swapped input has passed through the FFT, we then simply swap the complex parts of the FFT output to get the desired result.

Remove Scaling

The required modification to our FFT algorithm, for conversion to the IFFT, is to eliminate the scaling at each butterfly stage in the program core.
Recall that the classical definition of the discrete transform pair [Opp/Sch,pg.532] shows scaling by 1/N appearing in only the **I**DFT. We previously decided to put the scaling into the FFT algorithm instead, because the classical DFT of a real full-scale sampled sinusoid can produce a bin output magnitude as high as N/2. Any value whose magnitude is beyond 1.0 cannot be represented in the **q**23 output format we have chosen for the FFT implementation. Because of the scaling, however, any full-amplitude sinusoid can be represented in our FFT.

Because the first stage of butterflys is topologically prior to scaling, we always cut the input signal amplitude by 1/2 before passing it on to the FFT algorithm. We do this to prevent overflow in the first stage of butterflys. Likewise for the IFFT; even though there is no required scaling following our butterfly stages, the possibility of overflow still exists in the first butterfly of our IFFT. So, the input amplitude halving and its concomitant post-digit-reversal amplitude compensation remains in both the FFT and IFFT algorithms.

This swapping is not constituent to the core of the IFFT algorithm that follows. The required changes with regard to scaling by (1/N) appear in bold type:

```
!************************** begin the IFFT Radix-4 ********************************
ifft: MOV ONE > IE                                  MOV #N > N2
MOV #REGION_R_LOC > BASER                           CLR K

K_loop:MOV N2 > N1                                  CLR IA1
CLR J                                               LSH  N2 >>2
CLR IA2                                             CLR IA3
   J_loop:
   MOV N2 > MACP                 MOV #REGION_I_LOC > BASEI
   ADD N2, MACP  > I2            ADDV  J, BASEI              RD *(I )R  > XI        !XinArray
   ADD I2, MACP  > I3            MOV N2 > I1                 RD *(I2)R > XI2
   NEG ONE > MACP                SUBV N1, SIZEM1R > MACP     RD *(I )I   > YI       !YinArray
   ADD IE, MACP  > count         ADD XI, XI2    > R1         RD *(I2)I  > YI2
   ADD ONE, MACP                 SUB XI2, XI     > R3        RD *(I1)R > XI1
   NOP                           ADD YI, YI2    > S1         RD *(I3)R > XI3
   NOP                           SUB YI2, YI     > S3        RD *(I1)I  > YI1
   DBL IA2 > IA2_doub            ADD  XI1, XI3  > R2         RD *(I3)I  > YI3
   DBL IA3  > IA3_doub           SUB XI3, XI1  > R4          RD *(IA2_doub)    > CO2
   DBL IA1  > IA1_doub           ADD  YI1, YI3  > S2         RD *(IA2_doub(+)) > SI2
   NOP                           SUB YI3, YI1   > S4         RD *(IA3_doub)    > CO3
   NOP                           SUB R2, R1 > R2b            RD *(IA3_doub(+)) > SI3
   NOP                           SUB S2, S1  > S2b           RD *(IA1_doub)    > CO1
   ADD I, MACP  > Ir             SUB S4, R3  > R1b           RD *(IA1_doub(+)) > SI1
   ADD I1, MACP  > I1r           ADD S3, R4 > S1b            RD *(BASE += N1)R  > XI      !ok
   ADD I2, MACP  > I2r           ADD R3, S4 > R3b            RD *(I2)R > XI2
   ADD I3, MACP  > I3r           SUB R4, S3  > S3b           RD *(BASE += N1)I   > YI     !ok

      NOP                        REPT I_loopEnd, count > REPT_CNT
      I_loop:
            R2b X CO2  > MAC              ADD R1, R2 > *(Ir)R      RD *(I2)I  > YI2
      MAC + S2b X SI2  >>0 > *(I2r)R      ADD S1, S2  > *(Ir)I     RD *(I1)R > XI1
            S2b X CO2  > MAC              ADD XI, XI2   > R1       RD *(I3)R > XI3
      MAC - R2b X SI2  >>0 > *(I2r)I      SUB XI2, XI   > R3       RD *(I1)I  > YI1
            R1b X CO3  > MAC              ADD YI, YI2   > S1       RD *(I3)I  > YI3
      MAC + S1b X SI3  >>0 > *(I3r)R      SUB YI2, YI   > S3
            S1b X CO3  > MAC              ADD  XI1, XI3 > R2
      MAC - R1b X SI3  >>0 > *(I3r)I      SUB XI3, XI1 > R4
            R3b X CO1  > MAC              ADD  YI1, YI3 > S2
      MAC + S3b X SI1  >>0 > *(I1r)R      SUB YI3, YI1 > S4
            S3b X CO1  > MAC              SUB R2, R1 > R2b
      MAC - R3b X SI1  >>0 > *(I1r)I      SUB S2, S1  > S2b
      NOP                                 SUB S4, R3  > R1b
      NOP                                 ADD S3, R4 > S1b          RD *(BASE += N1)R  > XI
      NOP                                 ADD R3, S4 > R3b          RD *(I2)R > XI2
      I_loopEnd:NOP                       SUB R4, S3  > S3b         RD *(BASE += N1)I   > YI
```

6. Double Precision FFT Radix-2

6.1. Double Precision FFT Radix-2 C-Program Model

```
/* This is a validation test of the [Burrus/Parks,pg.108] Radix-2 FFT */
/* Modified for zero-indexing. - JonD May'92 */
/* This is an in-place algorithm. */

#include <stdio.h>
#include <math.h>

/* Identifies bin. Use power of two for integral number periods within record. */
#define FREQ      4

#define N       256
#define RADIX     2

double roundint();

void main() {

/* FFT */
int N1, N2, M, IE, IA, I, J, K, L;
double COS, SIN, XT, YT;
double X[256], Y[256], Wreal[256], Wimag[256];
/* TWIDDLE */
double P, A, twopi;
/* INPUT SIGNAL */
double epsilon, yn, yqn;
double pi;


M = roundint(log((double)N)/log((double)RADIX));
twopi = 8.*atan(1.);
pi    = 4.*atan(1.);

/*** GENERATE TWIDDLE FACTORS ********************************/
P = twopi/N;
for(K=0; K<N; K++) {
     A = K*P;
     Wreal[K] = cos(A);
     Wimag[K] = sin(A);
}
/*************** end TWIDDLE ********************************/
```

```
/*** GENERATE INPUT SIGNAL *******************************/
epsilon = 2.*sin((pi*FREQ)/N);
yqn = 0.0;
yn  = -cos((pi*FREQ)/N);

for(K=0; K<N; K++) {
     X[K] = yqn/2;   /* fixed-point overflow prevention */
     Y[K] = 0.0/2;
     yqn -= epsilon*yn;
     yn  += epsilon*yqn;
}
/************** end INPUT ********************************/
/* for(I=0; I<N; I++)
     printf("X[%d] = %lf\t  Y[%d] = %lf\n", I, X[I], I, Y[I]);
*/

/******************** THE FFT ******************************/
N2 = N;
for(K=0; K<M; K++) {
     N1 = N2;
     N2 /= RADIX;
     IE = N/N1;
     IA = 0;
     for(J=0; J<N2; J++) {
          COS = Wreal[IA];
          SIN = Wimag[IA];
          IA += IE;
          for(I=J; I<N; I+=N1) {
               L = I + N2;

               XT   = (X[I] - X[L])/RADIX;
               X[I] =  XT + X[L];
               YT   = (Y[I] - Y[L])/RADIX;
               Y[I] =  YT + Y[L];

               X[L] = COS*XT + SIN*YT;
               Y[L] = COS*YT - SIN*XT;
          }
     }
}
/********************************** end FFT ******************/
```

```
/*************************** BIT SWAP ***************************/
J = 0;
N1 = N - 1;
for(I=0; I<N1; I++) {
          if(I >= J) goto label101;
          XT   = X[J];
          X[J] = X[I];
          X[I] = XT;
          XT   = Y[J];
          Y[J] = Y[I];
          Y[I] = XT;
label101: K = N/RADIX;
label102: if(K > J) goto label103;
             J -= K;
             K /= RADIX;
             goto label102;
label103: J += K;
}

for(I=0; I<N; I++) {   /* compensation for earlier fixed-point overflow prevention */
      X[I] *= 2.;
      Y[I] *= 2.;
}
/*********************************** end BIT SWAP ****************/
for(I=0; I<N; I++) {
     if((fabs(X[I]) >= 1e-16) || (fabs(Y[I]) >= 1e-16))
         printf("X[%d] = %.16lf\t  Y[%d] = %.16lf\n", I, X[I], I, Y[I]);
}
}

/************* replacement for rint() *****************************/
double roundint(double x)
{
if(x >= 0.)return((double)((int)(x + 0.5)));
return((double)((int)(x - 0.5)));
}
```

6.2. Double Precision FFT Radix-2 ESP2 Program

```
! FFT Radix-2, double precision signal - JonD June'92.
! Ref.file: fft2sim.c
! System host must set the ESP_HALT_EN bit of the HOST_CNTL interface register.

PROGRAM  FFT2

PROGSIZE <= 69

DEFCONST
      RADIX = 2
      N = 256
      FREQ = 4
      Pi = 4.0*ATAN(1.0)
      M = INT(LOG(N)/LOG(RADIX))
      EPSILON = 2.0*SIN((Pi*FREQ)/N)
      BIG_ONE = (2**24)/(RADIX**M)
      REGION_T_LOC = $800800
      REGION_U_LOC = $800000

DEFSPR
      DILF  DILE  DILD  DILC
      REPT_CNT = 0
      PC = init
      SER_CONF = $007fff
      HARD_CONF = $008400

DEFGPR
   GLOBAL
      epsilon = EPSILON
      yqn = 0
      yn  = -COS((Pi*FREQ)/N)

      K    J     I     IE
      N2   COSI  YT_low           YT_hi      YTV_low
      N1   SINE  XT_low           XT_hi      XTV_low
      temp  Irev

DEFREGION T @REGION_T_LOC         ! TABLE REGION
      Twiddle[2*N]                ! holds interleaved single prec. cos() and sin() table,
                                  ! loaded by system host at download.
      Index                       ! AOR
```

```
DEFREGION U @REGION_U_LOC        ! DELAYLINE REGION
      Xarray[2*N]                ! real input double precision interleaved. 0 root required.
      Yarray[2*N]                ! imaginary input double precision interleaved.
      Xpointer @$2fd             ! AOR declarations
      Ypointer @$2fc             ! ditto
      XIpointer  XLpointer       ! used in butterfly
      YIpointer  YLpointer       ! ditto
      Xrev_pointer
      Yrev_pointer
!************************************************************************!

CODE
!******************** generate input signal ******************************!
init:
MOV &Yarray[0] > Ypointer         MOV &Xarray[0] > Xpointer
NOP                               CLR K

signal_loop:
!*** initialize double precision input arrays ***
CLR *(Ypointer)                   HALVE yqn > *(Xpointer)   ! overflow prevention in first butterfly.
CLR *(Ypointer(+))                CLR *(Xpointer(+))        ! low word!

!*** oscillator ***
MOV yqn > MACP                              MOV yn > MACP
MACP - epsilon X yn  > yqn
MACP + epsilon X yqn > yn
!*** end oscillator ***

!*** go around loop simple way ***
NOP                                             ADDV ONE,K
NOP                                             CMP  K,#N
ASH ONE <<1 > MACP                             {JMP  signal_loop, NEQ > CMR}
ADD Ypointer,MACP > Ypointer                    ADDV #2,Xpointer
!*************** end input signal generation *****************************!
```

```
!************************** begin the FFT ********************************!
MOV ONE > IE                      MOV #N > N2
NOP                               CLR K

K_loop:
MOV N2 > N1                       MOV &Twiddle[0] > Index
CLR J                             LSH N2 >>1

      J_loop:
      NOP                         MOV J > I
      NOP                         ADDV IE,Index
      NOP                         ADDV IE,Index
      MOV *(Index) > COSI         MOV *(Index(+)) > SINE

      I_loop:
      NOP               LSH I <<1 > XIpointer
      NOP               ADDV I,N2 > temp
      NOP               LSH temp <<1 > XLpointer               RD *(XIpointer(+)) > DILD
      NOP               MOV #-1 > ALU_SHIFT                    RD *(XIpointer) > DILC
      NOP               ADDV XIpointer,&Yarray[0] > YIpointer  RD *(XLpointer(+)) > DILF
      NOP               ADDV XLpointer,&Yarray[0] > YLpointer  RD *(XLpointer) > DILE
!*** XT = ***
      NOP               SUBV DILF,DILD > XT_low
      NOP               SUBB DILE,DILC > XT_hi                 RD *(YIpointer(+)) > DILD
      NOP               ASDL XT_hi,XT_low > XT_low             RD *(YIpointer) > DILC
      ASH XT_hi >>1     ADDV DILF,XT_low > *(XIpointer(+))     RD *(YLpointer(+)) > DILF
      NOP               ADDC DILE,XT_hi > *(XIpointer)         RD *(YLpointer) > DILE
!*** YT = ***
      NOP               SUBV DILF,DILD > YT_low
      NOP               SUBB DILE,DILC > YT_hi
      NOP               ASDL YT_hi,YT_low > YT_low
      ASH YT_hi >>1     ADDV DILF,YT_low > *(YIpointer(+))
      NOP               ADDC DILE,YT_hi > *(YIpointer)
!*** X[L] = ***
            XT_hi X COSI > MAC                LSH XT_low >>1 > XTV_low
      MAC + YT_hi X SINE > MAC                LSH YT_low >>1 > YTV_low
        XTV_low X COSI > temp
      MAC + ONE X temp > MAC
        YTV_low X SINE > temp
      MAC + ONE X temp > *(XLpointer)
!*** Y[L] = ***
            YT_hi X COSI > MAC                MOV MACRL > *(XLpointer(+))
      MAC - XT_hi X SINE > MAC
        YTV_low X COSI > temp
      MAC + ONE X temp > MAC                  ADDV N1,I
        XTV_low X SINE > temp                 CMP I,#N
      MAC - ONE X temp => *(YLpointer)       {JMP I_loop, LT > CMR}
      MOV MACRL => *(YLpointer(+))
      NOP          ADDV ONE,J
      NOP          CMP J,N2
      NOP         {JMP J_loop, LT > CMR}
      NOP

NOP        ADDV ONE,K
NOP        CMP K,#M
NOP       {JMP K_loop, LT > CMR}
NOP        LSH IE <<1

MOV &Xarray[0] > XIpointer                   JS bit_reversal
MOV &Yarray[0] > YIpointer                   NOP
!*************************** end FFT ************************************

!***************************** END PROGRAM ******************************
outahere:
NOP                               HALT
NOP

!*************** bit reversal / undo overflow prevention in first butterfly ***********************
! This subroutine places the high order results in the Xarray[(+)] and Yarray[(+)]
! addresses; i.e., what we previously used to hold the low order double precision bits.
bit_reversal:
MOV &Xarray[0] > Xrev_pointer     MOV  &Yarray[0] > Yrev_pointer
CLR Irev                          REPT reversal,N-1

NOP                               ADDV #BIG_ONE,Irev           RD *(Xrev_pointer) > DILF
ASH ONE <<1 > MACP                BREV Irev > Xrev_pointer     RD *(Yrev_pointer) > DILE
DBL DILF => *(XIpointer(+))       LSH  Xrev_pointer <<1
DBL DILE => *(YIpointer(+))       ADDV &Yarray[0],Xrev_pointer > Yrev_pointer
reversal:
ADD YIpointer,MACP > YIpointer    ADDV #2,XIpointer

NOP                                RS
NOP

!****************************** end bit rev ********************************
```

6.3. Double Precision FFT Radix-2 C-Program Simulation of ESP2 Program

```
/* reference file: fft2.c - JonD 6/15/92 */
/* This is a reduced precision (48-bit) rendering for ESP2 simulation.*/
/* Each stage of butterflys is scaled by 2 to prevent overflow.  A sinusoid
   will then have standard power level as output from scaled DFT (scaled by
   factor N).
   The complex input signal is stored in a double precision array for maximum
   S/N through the FFT system.  Multiplications with the input signal array
   then, are in double precision.
*/

#include <stdio.h>
#include <math.h>

double pow2to46, select_lower24(), lower24();
double pow2to47, RoundInt48(), TruncInt48(), TruncInt48MAC();
double pow2to23, RoundInt24(), TruncInt24();
double pow2to15, TruncInt16(), RoundInt16(), roundint();
double myfmod();

#define NMAX  4096
#define FREQ  4                    /* integer, empirical value. Period is always N/FREQ */
#define RADIX 2
#define TWIDDLE_SIZE 2*N

void main() {

/* FFT */
char buf[32];
int N1, N2, M, IE, INDEX, I, J, K, L, N;
double COS, SIN, XT, YT;
double X[NMAX], Y[NMAX];
double noise_power;
/* TWIDDLE */
long templ;
double twopi, Wreal[NMAX], Wimag[NMAX];
/* INPUT SIGNAL */
double epsilon, yn, yqn;
double pi, amplitude, sinamp, sinamp_dB;
```

```
/*printf("Input number of points: ");
gets(buf);  sscanf(buf, "%d", &N);
printf("Input amplitude of sinusoid (0. -> -90. dB): ");
gets(buf);  sscanf(buf, "%lf", &sinamp_dB);*/
N=256; sinamp_dB=0.;
sinamp = pow(10., sinamp_dB/20.);

M = roundint(log((double)N)/log((double)RADIX));
twopi = 8.*atan(1.);
pi    = 4.*atan(1.);
pow2to47 = pow(2.0, 47.0);
pow2to46 = pow(2.0, 46.0);
pow2to23 = pow(2.0, 23.0);
pow2to15 = pow(2.0, 15.0);

/**************** GENERATE TWIDDLE FACTORS *******************/
/* first value is: 0x7FFFFF */ /* first interleaved value is: 0x000000 */
/* last value is: 0x7FF622 */ /* last interleaved value is: 0xFCDBD5 */
printf("Twiddle factor table size is: %d\n", TWIDDLE_SIZE);
for(K=0; K<N; K++) {
      Wreal[K] = RoundInt24(cos((K*twopi)/N));
      templ = pow2to23*Wreal[K];
      if(templ >= 0x00800000L) {
            Wreal[K] = TruncInt24(1.0 - pow(2., -23.));
            templ = 0x7fffffL;
      }
      /*printf("%0.6lx\n", templ & 0xffffffL);*/  /* redirect std output to get */
                                                /* ascii file */
      Wimag[K] = RoundInt24(sin((K*twopi)/N));
      templ = pow2to23*Wimag[K];
      if(templ >= 0x00800000L) {
            Wimag[K] = TruncInt24(1.0 - pow(2., -23.));
            templ = 0x7fffffL;
      }
      /*printf("%0.6lx\n", templ & 0xffffffL);*/
} /* 24-bit fixed-point, q23 */
/*************** end TWIDDLE ********************************/
```

```
/*** GENERATE INPUT SIGNAL ********************************/
/*epsilon = RoundInt24(2.*sin((pi*FREQ)/N));*/
/*                                           *//* FREQ/N identifies bin. Use power of two */
/*amplitude = (pow2to23-1.)/pow2to23;        *//* for integral */
/*yqn = RoundInt24(0.0);                     *//* number of periods within record. */
/*yn  = RoundInt24(-amplitude*cos((pi*FREQ)/N));*/
/*for(K=0; K<N; K++) {                       */
/*    X[K] = TruncInt24(yqn*0.5);            *//* scale to prevent overflow in 1st butterfly.*/
/*    Y[K] = TruncInt24(0.0*0.5);            *//* Use of RountInt yields better S/N */
/*    templ = pow2to23*X[K];                 */
/*    printf("%d \t %0.6lx\n", 2*K,templ & 0xffffffL); */ /* redirect std output to get */
                                                           /* ascii file */
/*                                                         */
/*    yqn += TruncInt24(-epsilon*yn);                      */
/*    templ = pow2to23*yqn;                                */
/*    if(templ >= 0x00800000L) {                           */
/*          yqn = TruncInt24(1.0 - pow(2., -23.));         */
/*    }                                                    */
                 /* NOP */
/*    yn  += TruncInt24(epsilon*yqn);                      */
/*    templ = pow2to23*yn;                                 */
/*    if(templ >= 0x00800000L) {                           */
/*          yn = TruncInt24(1.0 - pow(2., -23.));          */
/*    }                                                    */
/*}                                                        */
amplitude = (pow2to23-1.)/pow2to23;
for(K=0; K<N; K++) {
      X[K] = RoundInt24(0.5*sinamp*amplitude*sin(myfmod((twopi*FREQ*K)/N, twopi)));
                                                    /* 0.5 for overflow in 1st butterfly.*/
      Y[K] = RoundInt24(0.5*sinamp*amplitude*0.0); /* Use of RountInt yields better S/N */
/*    printf("%d \t %0.6lx\n", 2*K, (long)(pow2to23*X[K]) & 0xffffffL); */
}                                                   /* redirect std output to get ascii file */
/*************** end INPUT ********************************/
```

```
/******************** THE FFT *******************************/
N2 = N;
IE = 1;
for(K=0; K<M; K++) {
      N1 = N2;
      N2 = N2 >> 1;
      INDEX = 0;
      for(J=0; J<N2; J++) {
            COS = Wreal[INDEX];
            SIN = Wimag[INDEX];
            INDEX += IE;
            for(I=J; I<N; I+=N1) {
                  L = I + N2;
                                                        /* scale by 2 for */
                  XT   = TruncInt48((X[I] - X[L])/RADIX);/* each           */
                  X[I] = TruncInt48(XT + X[L]);          /* stage of       */
                                                        /* butterflys     */
                  YT   = TruncInt48((Y[I] - Y[L])/RADIX);
                  Y[I] = TruncInt48(YT + Y[L]);

            X[L] = TruncInt48MAC(COS*TruncInt24(XT)) + TruncInt48MAC(SIN*TruncInt24(YT))
          + TruncInt48MAC(COS*select_lower24(XT)) + TruncInt48MAC(SIN*select_lower24(YT));

            Y[L] = TruncInt48MAC(COS*TruncInt24(YT)) - TruncInt48MAC(SIN*TruncInt24(XT))
          + TruncInt48MAC(COS*select_lower24(YT)) - TruncInt48MAC(SIN*select_lower24(XT));
            }
      }
      IE = IE << 1;
}
/*********************************** end FFT ********************/
```

```
/********************** BIT SWAP/INPUT SCALING COMPENSATION *******/
J = 0;
N1 = N - 1;
for(I=0; I<N1; I++) {
      if(I >= J) goto label101;
      XT   = X[J];
      X[J] = X[I];
      X[I] = XT;
      XT   = Y[J];
      Y[J] = Y[I];
      Y[I] = XT;
label101:K = N/RADIX;
label102:if(K > J) goto label103;
            J -= K;
            K /= RADIX;
            goto label102;
label103:J += K;
}

for(I=0; I<N; I++) {  /* undo overflow prevention */
      X[I] *= 2;
      Y[I] *= 2;
}
/********************************** end BIT SWAP ***************/

/*************************************** STATISTICS ********************/
noise_power = 0;
for(I=0; I<N; I++) {
     if((fabs(X[I]) < pow(2., -23.)) && (fabs(Y[I]) < pow(2., -23.)));
     else       {
          printf("X[%d] = %.16lf\t  Y[%d] = %.16lf\n", I, X[I], I, Y[I]);
          templ = floor(X[I]*pow2to23);
          printf("integerX = %x\t\t\t", templ & 0xfffffeL);
          templ = floor(Y[I]*pow2to23);
          printf("    integerY = %x\n", templ & 0xfffffeL);
      }
     if((I == FREQ) || (I == (N-FREQ)))    /* sine (not cosine) wave is assumed */
          noise_power += X[I]*X[I];
     else noise_power += X[I]*X[I] + Y[I]*Y[I];
}
noise_power += pow((0.5*amplitude*sinamp + Y[FREQ]), 2.)
                      + pow((0.5*amplitude*sinamp - Y[N-FREQ]), 2.);
printf("signal/noise through FFT w/r full scale is: %.13lf dB\n",
                      -10.*log10(noise_power/(0.5*amplitude*amplitude)));
/***************************************************************************/
```

```
/************************** print out ********************************/
/*for(I=0; I<N; I++) {
           templ = floor(X[I]*pow2to23);
           printf("integerX[%d] = %x\t\t\t",2*I + 1, templ & 0xfffffeL);

           templ = floor(Y[I]*pow2to23);
           printf("    integerY[%d] = %x\n",2*I+514, templ & 0xfffffeL);
}
*/
/*************************************************************************/
}

/****************************** SUBROUTINES ****************************/
double RoundInt48(x)
double x;
{
return(roundint(pow2to47*x)/pow2to47);
}

double RoundInt24(x)
double x;
{
return(roundint(pow2to23*x)/pow2to23);
}

double RoundInt16(x)
double x;
{
return(roundint(pow2to15*x)/pow2to15);
}

double TruncInt48(x)
double x;
{
return(floor(pow2to47*x)/pow2to47);
}

/******** replacement for rint() on NeXT O.S. v2.1 *******/
double roundint(double x)
{
if(x >= 0.)return((double)((int)(x + 0.5)));
return((double)((int)(x - 0.5)));
}
```

```
double TruncInt48MAC(x)
double x;
{
return(floor(pow2to46*x)/pow2to46);
}

double TruncInt24(x)
double x;
{
return(floor(pow2to23*x)/pow2to23);
}

double TruncInt16(x)
double x;
{
return(floor(pow2to15*x)/pow2to15);
}

double select_lower24(x)  /* also loses the LSB; the 48th bit */
double x;
{
double TruncInt24(), tempd;
tempd = x - TruncInt24(x);
return(floor(pow2to46*tempd)/pow2to46);
}

double lower24(x)
double x;
{
double TruncInt24(), tempd;
tempd = x - TruncInt24(x);
return(floor(pow2to47*tempd));
}

/****************** MYFMOD function ********************************************/
/*** used to increase accuracy of sin routines ***/
/*** fmod() uses floor() instead of (int). ***/
/*** fmod() also failing to return values within |modulo| ***/
/*** Want (int), which is magnitude truncation, for negative arguments. ***/
double myfmod(argument, modulo)
register double argument, modulo;
{
return(argument - (int)(argument/modulo)*modulo);
}
/********************************* END subroutines ****************************/
```

6.4. Comments on the Double Precision FFT Radix-2 Program

The complex Twiddle factor table must be loaded into external memory before the ESP2 program is run. The Twiddle factor table used here is the same as the one used before. To load external memory, another ESP2 Application must be run. See the External Memory Host Access Application.

The other FFT ESP2 program was optimized for speed, whereas this FFT Radix-2 ESP2 program is optimized for accuracy. This ESP2 program demonstrates the ability of ESP2 to perform double precision multiplys, accumulations, and arithmetic. The complex input signal is generated at 24 bits, **q**23, but it is maintained at double precision (48 bits, **q**47) as it passes through the FFT stages. This shows that an unsigned multiplier is not necessary for double precision math.[109] Although the double precision results are available, the final output only extracts single precision results after the bit reversal subroutine.

If we eliminate the rounding of the high precision input signal to 24-bits in the C-program simulation, we will find the noise introduced by the FFT circuit itself. The simulation estimates the total noise power introduced by the FFT to be about 132 dB below unity; the noise power spectrum is well below that level. If we eliminate the rounding of the high precision Twiddle factors to 24-bits, the total noise power falls to about -242 dB.

6.5. Changes to ESP2 Program to make the IFFT (Radix-2)

The same considerations apply as before.

```
      I_loop:
      NOP          LSH I <<1 > XIpointer
      NOP          ADDV I,N2 > temp
      NOP          LSH temp <<1 > XLpointer               RD *(XIpointer(+)) > DILD
      NOP          NOP                                    RD *(XIpointer) > DILC
      NOP          ADDV XIpointer,&Yarray[0] > YIpointer  RD *(XLpointer(+)) > DILF
      NOP          ADDV XLpointer,&Yarray[0] > YLpointer  RD *(XLpointer) > DILE
!*** XT = ***
      NOP          SUBV DILF,DILD > XT_low
      NOP          SUBB DILE,DILC > XT_hi                 RD *(YIpointer(+)) > DILD
      NOP          ADDV DILF,DILD > *(XIpointer(+))       RD *(YIpointer) > DILC
      NOP          ADDC DILE,DILC > *(XIpointer)          RD *(YLpointer(+)) > DILF
      NOP          NOP                                    RD *(YLpointer) > DILE
!*** YT = ***
      NOP          SUBV DILF,DILD > YT_low
      NOP          SUBB DILE,DILC > YT_hi

      NOP          ADDV DILF,DILD > *(YIpointer(+))
      NOP          ADDC DILE,DILC > *(YIpointer)
```

[109]This fact was also demonstrated in [Dattorro] where double precision coefficients were applied to a recursive digital filter structure using the technique called *residual coefficient coding.*

7. Reverberation

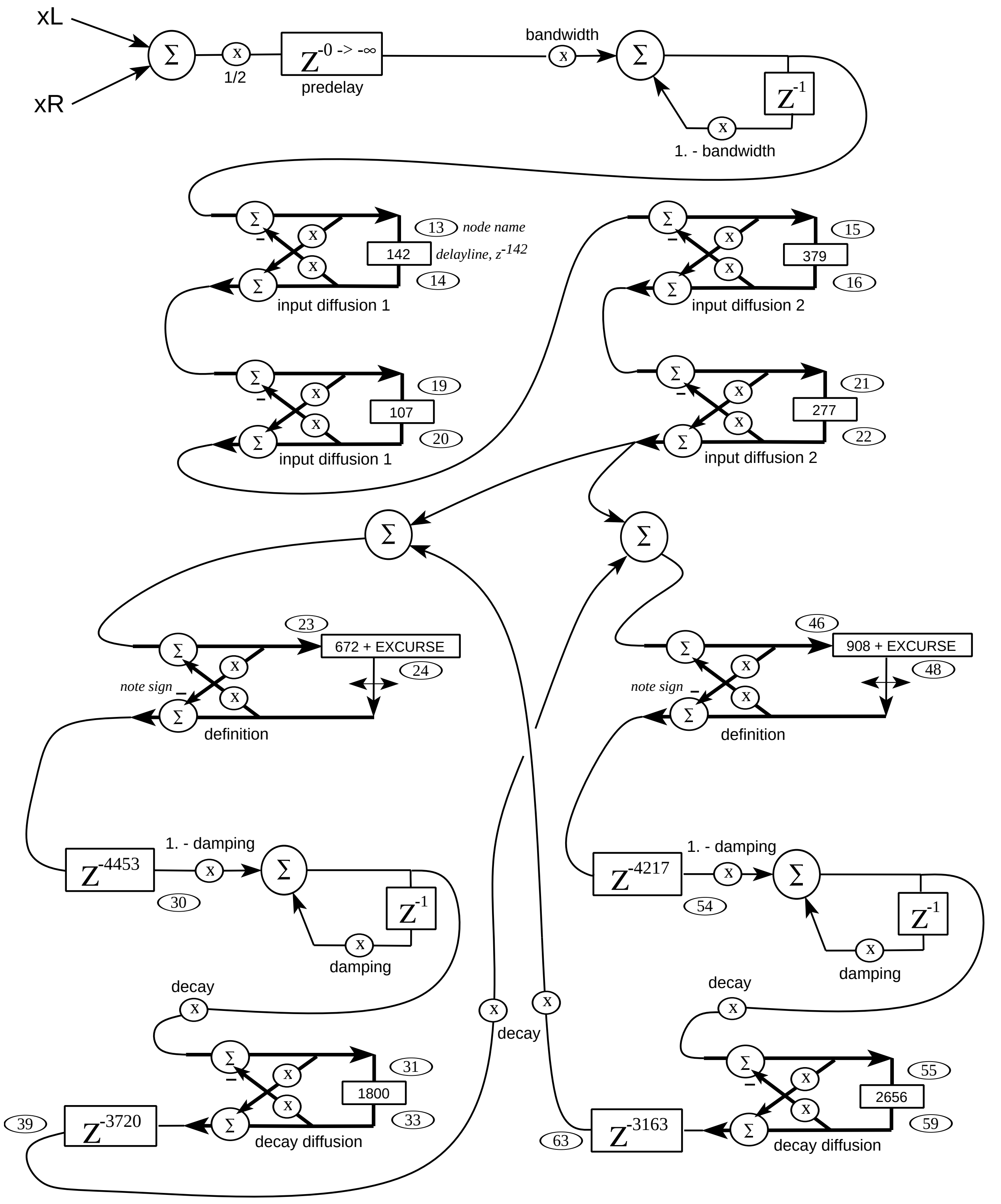


Figure R. Simplified Small Plate-class Reverberation topology due to Griesinger. For the output tap structure (yL, yR) see the application program. Delayline taps at nodes 24 and 48 are modulating. ©JonD 1994

7.1. ESP2 Reverberation Program

```
! TUSCON SMALL PLATE REVERB - JonD July'94 .

!*********************** assembler directives *****************************
PROGRAM VOCPLATE

DEFCONST
    Fs                    = 29761.894531
    MAG_TRUNC16   = $000800
    MAG_TRUNC24   = $001800

PROGSIZE <= 103

DEFSPR
        SER_CONF        = $007fff
        HARD_CONF     = $008400 | MAG_TRUNC16             ! turn on 16-bit  magnitude truncation for low noise
        REPT_CNT        = 0
        PC                     = bioz_loop

DEFGPR  GLOBAL
xL=0 @$80   xR=0 @$81  yL=0 @$82   yR=0 @$83

DEFCONST  GLOBAL
      DEFAULT_DECAY  = 0.50
      DECAY_DIFFUS      = 0.50
      DIFFUS1                  = 0.750
      DIFFUS2                  = 0.625
      DEFINITION            = 0.70
      HFBW                      = 0.9995
      DAMPING                = 0.0005
      OUTMIX                  = 0.60
      EXCURSE                = 16 + 1              !Maximum peak sample excursion of delayline tap modulation.
                                                   !Not used.  + 1 for Plus-One addressing mode when using interpolation.

DEFGPR  LOCAL
   save=0         lp_in=0
   dampL=0    dampR=0
   damping_u    = 1. - DAMPING  @$6c
   bandwidth_u = 1. - HFBW @$74
   outmix = OUTMIX     ! output tap level
```

```
DEFGPR  GLOBAL          /* The Knobs */
   ! input diffusion
   diffus1 = DIFFUS1 @$75
   diffus2 = DIFFUS2 @$77

   ! decay time
   decay = DEFAULT_DECAY @$6e
   ! decay diffusion (decorrelates tank signals)
   ! Host computes: decay_diffus = decay + 0.15 q23, having: floor = 0.25, ceiling = 0.50
   decay_diffus = DECAY_DIFFUS @$6f

   ! decay definition (controls density of tails)
   definition = DEFINITION @$71

   ! High frequency damping
   damping = DAMPING @$6d

   ! high frequency attenuation on input
   bandwidth = HFBW @$73

DEFREGION  V  @$800000
        node13_14[141]
        node19_20[106]
        node15_16[378]
        node21_22[276]
        node23_24[671+EXCURSE]
        node24_30[4452]
        node31_33[1799]
        node33_39[3719]
        node46_48[907+EXCURSE]
        node48_54[4216]
        node55_59[2655]
        node59_63[3162]
        predelay[42000]                              !optional
        minus_one = SIZEM1V
     GLOBAL
        predelay_offset = &predelay[1]  @$2de
!********************** end declarations ********************************
```

```
CODE
!******************************** ESP2 Code *****************************
!********* housekeeping ************
bioz_loop: NOP                                  DIFF  damping > damping_u
           NOP                                  DIFF  bandwidth > bandwidth_u

!****************** a/d/a ************************************
ASH  SER0L >>0 > xL   !network has some gain!   MOV  yL > SER7L
ASH  SER0R >>0 > xR                             MOV  yR > SER7R

!************** load predelay ***************
NOP                                             AVG  xL, xR > predelay[0]

!************************* lowpass filter input ********************
        bandwidth X  *(predelay_offset)  > MAC
MAC + bandwidth_u  X  lp_in > MAC, lp_in

!********** mono input 4 single stage guides ************************
NOP                                             MOV node13_14[141] > MACP
MAC    - diffus1  X "node13_14[141]" > node13_14[0]
MACP + diffus1  X "node13_14[0]" > MAC          MOV node19_20[106] > MACP
MAC    - diffus1  X "node19_20[106]" > node19_20[0]
MACP + diffus1  X "node19_20[0]" > MAC          MOV node15_16[378] > MACP
MAC    - diffus2  X "node15_16[378]" > node15_16[0]
MACP + diffus2  X "node15_16[0]" > MAC          MOV node21_22[276] > MACP
MAC    - diffus2  X "node21_22[276]" > node21_22[0]
MACP + diffus2  X "node21_22[0]" > MAC, save

!*************** decay ***************
MAC  +  decay X node59_63[3162] > MAC

!************** allpass 1 Left *****************************************
NOP                                             MOV node23_24[671] > MACP
MAC   + definition X "node23_24[671]" > node23_24[0]
MACP - definition  X "node23_24[0]" > node24_30[0]

!************************** lowpass Left *************************
        damping_u  X  node24_30[4452] > MAC
MAC + damping     X  dampL  >  dampL

!*************** decay ***************
         decay X dampL > MAC
```

```
!************************************ allpass 2 Left ************************
NOP                                         MOV node31_33[1799] > MACP
MAC   -  decay_diffus  X "node31_33[1799]" > node31_33[0]
MACP + decay_diffus  X "node31_33[0]" > node33_39[0]

!*************** decay ***************
MOV  save > MACP
MACP + decay X node33_39[3719] > MAC

!************** allpass 1 Right ******************************************
NOP                                         MOV  node46_48[907] > MACP
MAC   + definition X "node46_48[907]" > node46_48[0]
MACP - definition  X "node46_48[0]" > node48_54[0]

!************************** lowpass Right ***************************
       damping_u  X  node48_54[4216] > MAC
MAC + damping     X  dampR  >  dampR

!*************** decay ***************
        decay X dampR > MAC

!************************************ allpass 2 Right ************************
NOP                                         MOV node55_59[2655] > MACP
MAC   -  decay_diffus  X "node55_59[2655]" > node55_59[0]
MACP + decay_diffus  X "node55_59[0]" > node59_63[0]

!****************************** Left Out *********************************
       outmix X node48_54[266]   > MAC
MAC + outmix X node48_54[2974] > MAC
MAC  - outmix X node55_59[1913] > MAC
MAC + outmix X node59_63[1996] > MAC
MAC  - outmix X node24_30[1990] > MAC
MAC  - outmix X node31_33[187]   > MAC
MAC  - outmix X node33_39[1066]  > yL                          !all wet
!***************************** Right Out *********************************
       outmix X node24_30[353]   > MAC
MAC + outmix X node24_30[3627] > MAC
MAC  - outmix X node31_33[1228] > MAC
MAC + outmix X node33_39[2673] > MAC
MAC  - outmix X node48_54[2111] > MAC
MAC  - outmix X node55_59[335]   > MAC
MAC  - outmix X node59_63[121]   > yR                          !all wet
!************************************ end Reverb ****************************!
NOP                              JMP  bioz_loop
NOP                              BIOZ                   UPDATE  BASEV += minus_one
!*************************************************************************!
```

7.2. Discussion of the Reverberation Program

The given ESP2 program is not the most efficient coding of the Reverb network possible. We have chosen to optimize the program for readability. The program as shown can be compressed by at least 12 instruction lines.

The Allpass Lattice Topology

The eight *lattices* shown in the Reverb schematic are used in this effect as allpass filters, each having long impulse response time. The two coefficients within each individual lattice must remain identical to maintain the allpass transfer which is insensitive to coefficient quantization. The recommended range of these coefficients is from 0.0 to 0.9999999 (**q**23). Taking them both negative changes the character of the impulse response but does not destroy the allpass transfer. This change in character is exploited in the lattices having the coefficients called *definition* in the schematic. If the lattice coefficients should exceed 1.0, instability will result.

Allpass response is the forced (steady state) response of the chosen lattice output. Because the impulse response of each individual lattice within the Reverb schematic is so long, in some cases the integration time constant of the human hearing system is exceeded. This means that the perception of the allpass filter output may be as discretized events; i.e., not allpass.

This allpass lattice topology tends to clip prematurely at internal nodes, so the input to each lattice cannot be presented with full-scale signal at all frequencies. We like this allpass lattice because it requires only two lines of ESP2 code to implement; observe the coding of the four input diffusers.

Magnitude Truncation

Lattices produce distinct low-level tones, after input signal has been removed, known as zero-input limit cycles. The origin of these tones stems from ongoing signal quantization in a recursive topology. The spontaneous tones can be eliminated through the use of magnitude truncation (truncation towards zero) of the double precision intermediate results written out to single or lower precision external memory. Magnitude truncation is well known to subdue limit cycles in digital networks composed of ladders and lattices.[110] [Smith]

In this ESP2 program, magnitude truncation at the 16-bit level is activated in the declarations section since that is the presumed width of external memory. This means that every WR to external memory is automatically magnitude truncated to 16-bits. Alternatively, the magnitude truncation could be dynamically activated by the running program itself so that only selected portions of the code utilize this feature. Only the recursive portions of the network require magnitude truncation; In Figure R, the WR to the predelay does not require magnitude truncation.

[110]Magnitude truncation is never applied to data read from external memory in the ESP2.

The data is magnitude truncated only on its way out to external memory; i.e., the DOL SPRs are not permanently modified. This is advantageous when the 24-bit precision DOL contents are reused as in the *delayspec* Quote Scheme. If external memory is 24 bits in width (the maximum width supported by ESP2) then the need for magnitude truncation is lessened. The ESP2 still provides for automatic magnitude truncation at the 24-bit level, however. Using a manual shifting scheme, magnitude truncation at other bit levels can also be accommodated.

Magnitude truncation, in the specific case of Reverberator tank topologies employing lattice or ladder allpass networks, can reduce the circuit noise floor by 12 to 24 dB after input signal is removed. The reason that this is true is because the predominant noise mechanism is zero-input limit cycle oscillation,[111] a multiplicity of which being perceived as a whooshing, shushing noise floor. The magnitude truncation makes the Reverb output eventually go to absolute zero, two’s complement. The disadvantage to its use is that the ***THD+N*** **(**Total Harmonic Distortion plus Noise**)** of a steady state sinusoid through the linear Reverb network can be increased by anywhere from 0 to 6 dB.

Delayline Tap Modulation

In the ESP2 program given, we do not show any delayline tap modulation. Linear interpolation or, better yet, Allpass interpolation (as demonstrated in the Application of the same name) can be efficiently employed to slowly modulate[112] the nominal tap point of the two indicated delaylines in the schematic. The modulation will introduce undulating pitch change into the **tank**, the recirculating four lowest lattices in Figure R. As explained in the Linear Interpolation Application, Linear interpolation will introduce time-varying lowpass filtering as an artifact thus supplying some unaccounted damping. Allpass interpolation overcomes this particular problem and is perfectly applicable to Reverb because the required pitch change is microtonal. The sinusoidal LFO driving the modulator is economical requiring only two lines of ESP2 code, while the LFO rate of update is the same as the sample rate. For signals with much high frequency content, such as drums, these built-in modulators serve to break up some pretty audible modes.

There is no analogue to this modulation process in a real room **(**unless the walls are moving**)**. Without the modulation, we may well describe the imaginary space emulated by the given digital circuit as being enclosed by a picket fence. The slow modulation serves to effectively increase the sheer number of resonances **(eigentones**, modes of oscillation, picket density**)** in the tank. The number of resonances in a real room, hall, or plate is probably far beyond what is existent in our little (non-modulating) Reverb network. In the case of drums, the modulation is a godsend. In the case of piano, the modulation, though slight, may be objectionable because of a perceived vibrato.

[111]Here we use the term ’limit cycle’ in the classical DSP sense.

[112]at a rate on the order of 1 Hz, and at a peak excursion of about 8 samples for a sample rate of about 29.8 kHz,

Input Diffusers
The purpose of the four input diffusers is to quickly decorrelate the incoming signal before it reaches the tank. The tank recirculation can sometimes become perceptible as cyclic events if the input signal is not conditioned in this manner. This function becomes especially important for the successful reverberation of percussive sounds.

No diffusion corresponds to zero-valued allpass coefficients, while coefficient magnitudes close to unity produce buzzing local to the afflicted allpass. Optimal diffusion for the first-order allpass lies somewhere in a region closer to $|0.5|$ than to the extreme values of the coefficients. The preset values given in the declarations were arrived at by cut-and-try.[113]

Filters
The three single pole direct form I lowpass filters, used for input signal bandwidth control and Reverb tank damping, will not clip prematurely at any node [Dattorro,pg.857] [Jackson,ch.11.3] as implemented. The *bandwidth* control tracks cutoff frequency, while the *damping* control is high when the damping filter cutoff frequency is low.

Each filter requires only two lines of ESP2 code. Because they are first-order <u>lowpass</u> filters, any low-level zero-input limit cycles they might produce would be at DC; i.e., they will **not** produce tones like the lattices. [Jackson,ch.11.5] Magnitude truncation cannot be engaged here because the 24-bit filter memory is internal. Any signal truncation noise power spectrum generated by the filters themselves will be centered at DC, since it follows the pole frequency. The noise power spectrum peak gain is not great because the one pole is typically relatively far from the unit circle.[114]

Output Tap Points
As given in the ESP2 code, note that the tap structure forming the stereo output signal, yL and yR, is all wet (reverberated) signal. This particular tap structure is characteristic of the Plate-emulation class of Reverb networks.[115] Also note that the output tap structure produces a <u>synthetic</u> stereo image because the stereo input is converted to a monophonic signal[116] at the Reverberator input in this particular topology. Normally, the **desired output** is a mix of the reverberated signal, yL and yR, with the original (dry, full bandwidth) stereo input signal, xL and xR. But we do not show a mixer in the *a/d/a* section of code.

[113]That Reverb is in commercial production.

[114]Were the filters instead high-pass, limit cycle tones might be produced at Nyquist while the truncation noise spectrum would also be concentrated there.

[115]The physical ’Plate’, actually resident in some contemporary recording studios, fills a small room in some embodiments and is sometimes gold-plated. The input signal is typically injected onto the plate via one or two transducers while each output is a sum of multifarious signal taps, each tap transduced at a different location on the plate.

[116]The ALU’s AVG instruction is useful here.

Simple Reverb Networks[117]

We show, in Figure R, one particular network for producing reverberation. We believe that there must be a limitless variety of such networks. The question naturally arises as to why the simple digital circuit shown produces a convincing reverberation. Consider the plucked string of the violin; its envelope may be described as having a coherent exponential decay. It is this character which is theorized to be one of the primary discriminants of non-reverberated sound. Reverberating this sound, on the other hand, would tend to randomize the string envelope and phase producing a bumpier, more dynamic decay.

Long before DSP chips could be integrated into sampler-type synthesizers, reverberated sampled sound was simulated by altering the decay characteristics of recorded dry samples by randomizing an overlaid envelope applied at playback. While not absolutely convincing, it was enough to cause pioneers [Griesinger] [Blesser/Bader] to question the premise of precipitative work [Schroeder] at Bell Labs during the early 60's. One can deduce from that work that to achieve the ideal of colorless reverberation, the eigentone density needs to approach 3 per Hz. It can also be theorized that the limit on the number of achievable eigentones is proportional to the total delayline memory. From our current perspective we know that emulation of physical spaces can be convincingly performed using signal processing bandwidths as low as 10-12 kHz. This is true because of typically rapid acoustical absorption in the high frequency region, and because the desired output is a mix. This bandwidth would then require about 30 thousand eigentones, hence about 64k of delayline memory. In the 1960's, this amount was not economical.[118]

In Reverberator design, while a good general rule regarding delayline memory is certainly 'the more the better' [Griesinger], the efficient Reverb network shown herein[119] stands as testimony that the eigentone density criterion, predicting about 88k memory, is not a hard and fast rule. Of at least equal importance is the decorrelation of the decay.

[117]This discussion is adapted from [Blesser] and it is supplemented by Appendix III.

[118]The Lexicon Model 224 Digital Reverberation System introduced in 1979, possessed only 16k words of memory operating at a sample rate of 20 kHz. The Elecktromesstechnik, Wilhelm Franz K.G., EMT-250 Digital Reverberator distributed in the USA by Gotham Audio Corp. beginning in 1977, operated at a sample rate of 32 kHz having only 8k words of memory. The precursor to this machine is described in [Oppenheim,ch.2].

[119]given a 15 kHz processing bandwidth and having only 22k words of memory, not including predelay,

Color
On the other hand, our Reverb network signal response is not colorless. Empirically we find that some of the most sought commercial Reverbs are somewhat colored in their frequency responses. This means that they impose some audible resonances upon the input signal. It is not unusual to find as many musicians and recording engineers who like a particular Reverb as those who do not. We find that some recording engineers do **not** want accurate emulation of a physical space because the reflection density takes too long to build; they, in fact, sometimes want instantaneous high density reflections with smooth exponential decay of the envelope having randomization in only the phase trail.

Choosing a particular Reverb for a particular application is commonplace, and purveyors of such equipment have been known to purchase an audio signal processing box just to acquire one particular algorithm.[120] At some level, choice of Reverb becomes a matter of taste much like art. There is no one universal reverberation network that satisfies everyone for each and every application; we speculate that there never will.

Design
A technical chronicle of developments in the art of Reverberator design can be found in [Gardner]. That treatise surveys the very latest techniques. Gardner provides a translation (from the French language) of the vanguard, Jot.

[120]much like buying a Compact Disc because one likes the title track.

8. Musical Filtering

Smith gives a good introduction to classical digital filter theory in [Strawn,ch.2], requiring only basic knowledge of math from the reader. Here we discuss filtering requirements for musicians whose criteria are quite different from those of the electronics engineer.

8.1. Filter (Q) Selectivity

Electronics engineers are accustomed to think of digital filters analytically in terms of pole/zero number and locus, cutoff frequency, passband ripple, transition band or slope, stopband attenuation, etc. Musicians and recording engineers are more comfortable thinking in terms of filter parameters: gain or cut, center frequency, and filter Q (selectivity) or bandwidth. Formally, filter Q is defined as the positive quantity:

$$Q = \omega_c / \Delta\omega = \omega_c / (\omega_2 - \omega_1) \qquad \text{(qqq)}$$

;i.e., the center frequency divided by the bandwidth. The bandwidth is determined from the definition of the cutoff frequencies (ω_1 and ω_2). Traditionally, cutoff frequencies occur at an absolute half-power level. In the prototypical case of a steep unity-gain (0 dB) lowpass filter, we recall this level as corresponding to the frequency location where the magnitude-square transfer reaches -3.01 dB (=10 $\log_{10}(1/2)$).

But shallow audio filters may not have a 3 dB transfer excursion, so we must refine the definition of **cutoff frequency** in terms of half-power *excursion*; **not** an absolute level.

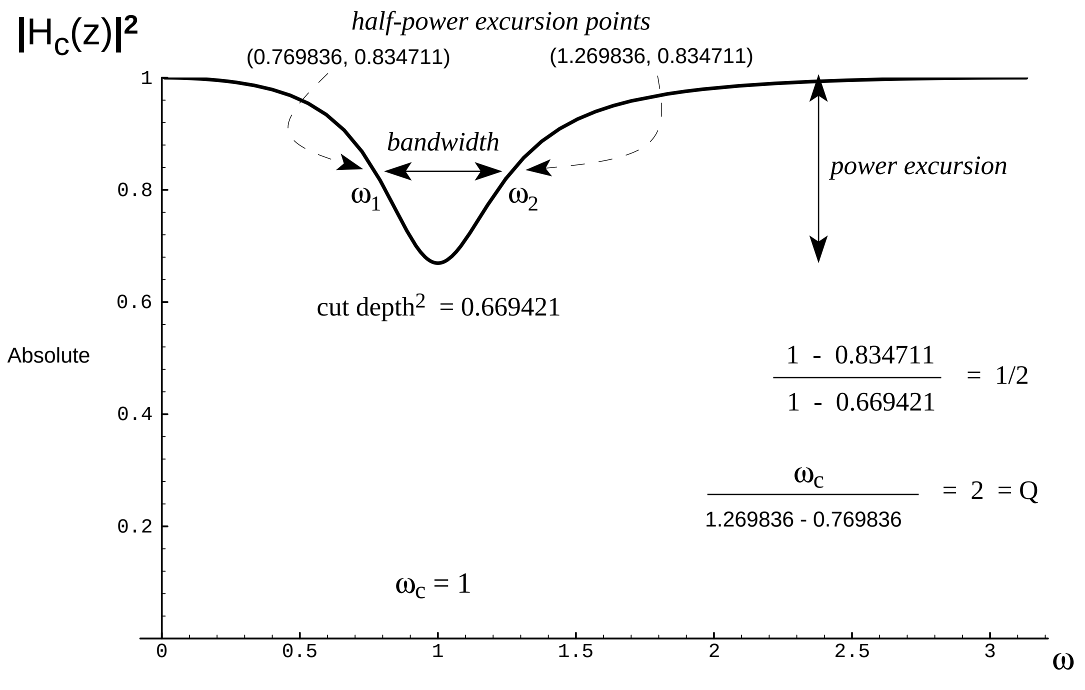


Figure Cut. Cut filter excursion approx. 1.7 dB.

Take for example the *cut filter* magnitude-square transfer function shown in Figure Cut. This example transfer has a Q of 2. We define the two musical cutoff frequencies as corresponding to the level at which

$$(1 - |H_c(e^{j\omega})|^2) \; / \; (1 - |H_c(e^{j\omega_c})|^2) = 1/2 \qquad \text{(NNN)}$$

We must solve this equation for ω; there are two solutions, ω_1 and ω_2. Referencing Figure Cut, this equation instructs us to measure the bandwidth half-way down the trough of the magnitude square transfer. This makes intuitive sense. We cannot use the traditional definition of cutoff frequency for this example because the trough is not deep enough. But note that when $|H_c(e^{j\omega_c})|^2 = 0$ (the notch filter), the solution to (NNN) corresponds to the classical definition of cutoff frequency.

The situation is pretty much the same for the *resonator.* Whereas the cut filter asymptotes to unity at $z = \pm 1$, the resonator is loosely defined as a second-order peaked filter having a peak gain normalized to unity at its center frequency. The resonator can be formulated such that its magnitude square transfer is an exact flip of the corresponding cut filter about the horizontal half-power excursion line; i.e., symmetrical with the cut filter. (This is why many of the numbers are exactly the same in Figure Reso as they are in Figure Cut.) We shall see how shortly.

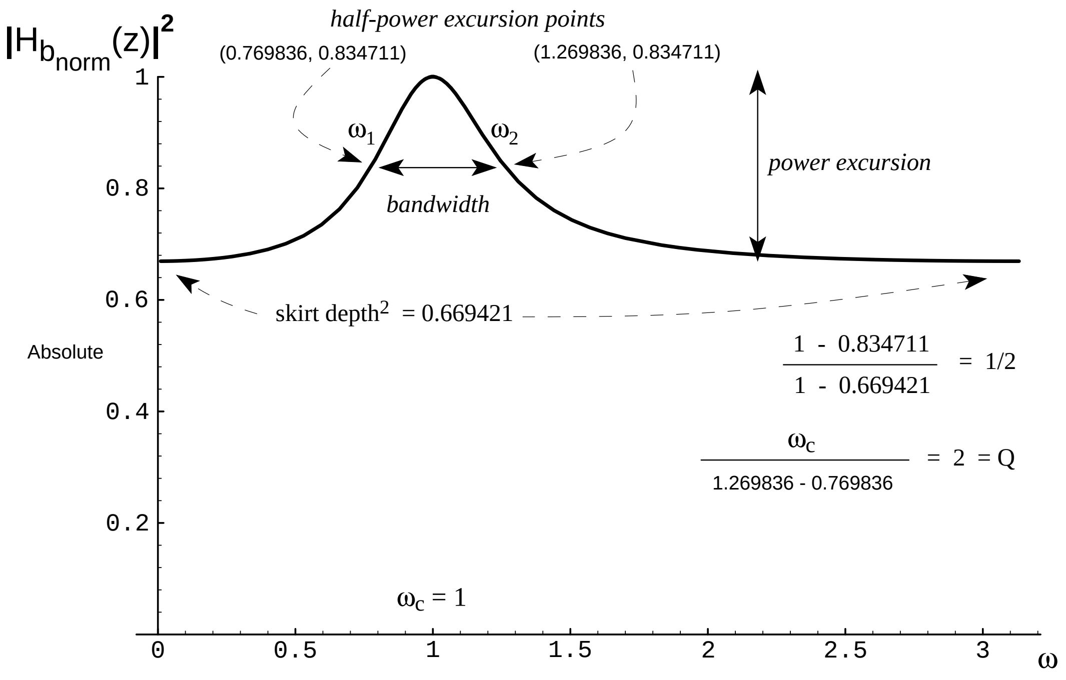


Figure Reso. Resonator excursion approx. 1.7 dB.

For the resonator (the normalized boost filter) we acquire the two musical cutoff frequencies, ω_1 and ω_2 , solving the slightly different equation:

$$(1 - |H_{b_{norm}}(e^{j\omega})|^2) \;/\; (1 - |H_{b_{norm}}(\pm 1)|^2) = 1/2 \qquad \text{(RRR)}$$

As before, the bandwidth is measured half-way <u>up</u> the peak of the magnitude <u>square</u> transfer. Again we note that when $|H_{b_{norm}}(\pm 1)|^2 = 0$ (the perfect resonator), the solution to (RRR) corresponds to the classical definition of cutoff frequency.

Having gained an understanding of musical filter Q, we begin with two unique and musically useful digital filter transfer functions which just happen to fit our definition of filter selectivity:

8.2. The Cut Filter

When constructing a notch filter, we expect there to be an absolute zero of transmission at a chosen place in the frequency-domain transfer. If we use a filter that just has zeroes (i.e., no poles), we can indeed make a notch. The problem is that the rest of the transfer function will not be very flat as we might like it to be. We might also like a surgical notch; one that has high selectivity. For example, here is the magnitude transfer of a notch filter evaluated at $z = e^{j\omega}$, zero-radius $R = 1$, and zero-angle $\theta = 1$ radian:

$|1 - 2 R \cos(\theta) z^{-1} + R^2 z^{-2}|$

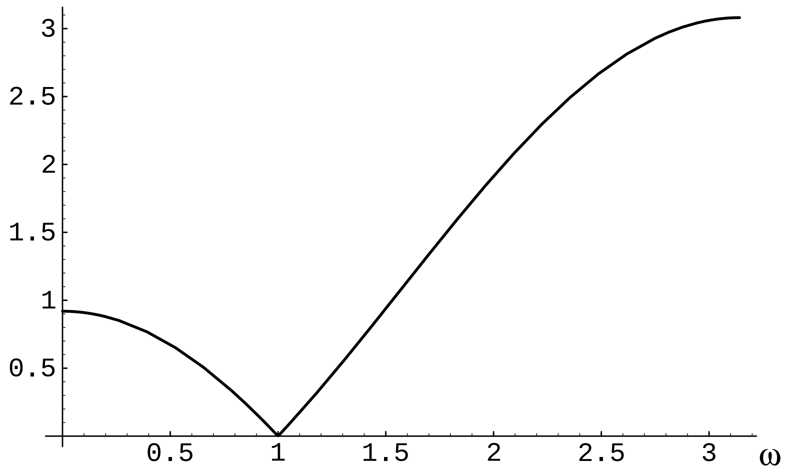


Figure BN. A poor notch filter.

This transfer in Figure BN would pretty much obliterate a musical signal; especially noting the gain at high frequencies. Also, when the notch is moved to a new fixed location, the rest of the transfer changes its shape in an undesirable way. Hence, this particular notch filter is not very useful.

In [Regalia/Mitra][121] it is shown how to make the passband portion of the notch filter flat, and to achieve high selectivity; this is accomplished by adding poles! This result is illustrated in the transfer function of equation (hnz).

$$H_n(z) = (1/2)(1+\beta)\,\frac{1 + 2\gamma z^{-1} + z^{-2}}{1 + \gamma(1+\beta) z^{-1} + \beta z^{-2}} \qquad \text{(hnz)}$$

$|H_n(z)|$ @Q=2

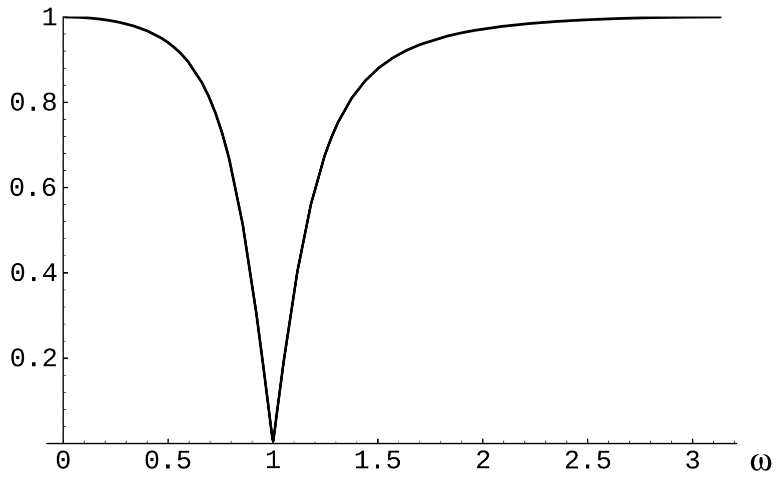


Figure Notcher. The notch filter.

This **notch** filter, (hnz) in Figure Notcher, will have an absolute zero at its center frequency having controllable selectivity, while its magnitude at DC and Nyquist is always 1 regardless of the center frequency. We must determine how to obtain a trough of arbitrary depth while maintaining the other attributes; that would be called a parametric **cut** filter. Before we do that, we look at the resonator which is a perfect power-wise flip of this notch filter.

[121][Regalia/Mitra] also discusses the construction of shelving filters using the same concepts.

8.3. The Resonator

We discuss the use of a resonator as a filter. It is easy to construct a resonator using only poles. But such a transfer has problems similar to those we encountered with the all-zero notch filter; particularly with the shape, selectivity, and magnitude evaluated along the unit circle in the **z**-plane (i.e., at $z = e^{j\omega}$). In particular, the peak magnitude will vary as center frequency is changed to new fixed values.

In [Jackson,ch.4.3] it is shown how to normalize the height of the resonator peak as the center frequency changes, by adding two zeroes; one at Nyquist and the other at DC! This musically useful result is distilled in equation (hrz).

$$H_r(z) = (1/2)(1 - \beta) \frac{1 - z^{-2}}{1 + \gamma(1+\beta) z^{-1} + \beta z^{-2}} \qquad \text{(hrz)}$$

$|H_r(z)|$ @Q=2

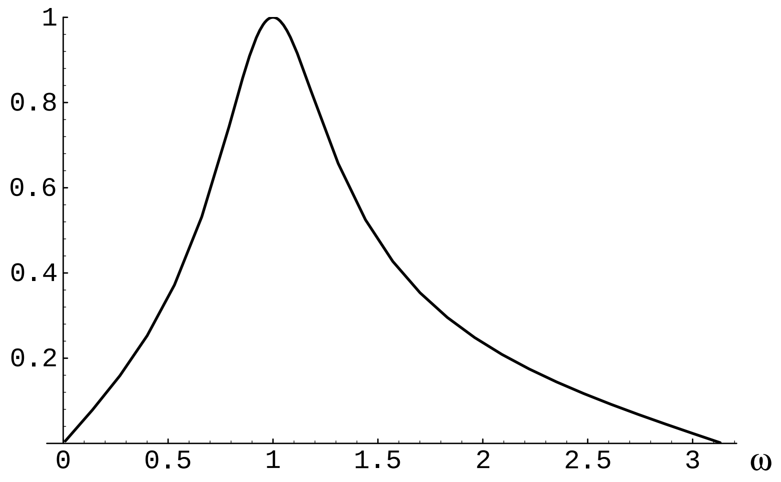


Figure Peaker. The perfect resonator.

This filter has a peak gain which is always precisely 1, regardless of the center frequency. This is characteristic of a resonator. The two zeroes make the skirts of the magnitude response asymptote to zero at the extremities. When the extremities reach zero we call this the **perfect resonator**, (hrz) shown in Figure Peaker. We must determine how to make skirts of arbitrary depth; the **resonator**. We must also determine how to place the skirts at absolute magnitude 1 while achieving arbitrary peak heights; this would be called a parametric **boost** filter.

8.4. Allpass Filter-Topology

These two transfers, $H_n(z)$ and $H_r(z)$, have desirable theoretical and practical properties which we will now expose. First, there is a strong bond between (hnz) and (hrz). Because their denominators are identical, there is one circuit that can generate both.

Consider the allpass lattice topology:

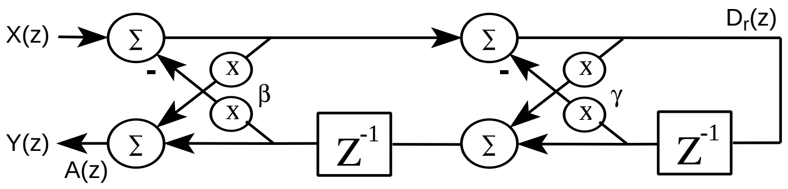


Figure Lattice. Lattice 2nd Order Allpass Filter

The lattice in Figure Lattice has the allpass transfer function:

$$A(z) = \frac{Y(z)}{X(z)} = \frac{\beta + \gamma(1+\beta)z^{-1} + z^{-2}}{1 + \gamma(1+\beta)z^{-1} + \beta z^{-2}} \qquad \text{(ar0)}$$

Some characteristics of the allpass are summarized:

$$|A(z)| = 1, \qquad A(\pm 1) = 1, \qquad A(e^{j\omega_c}) = -1. \qquad \text{(ar1)}$$

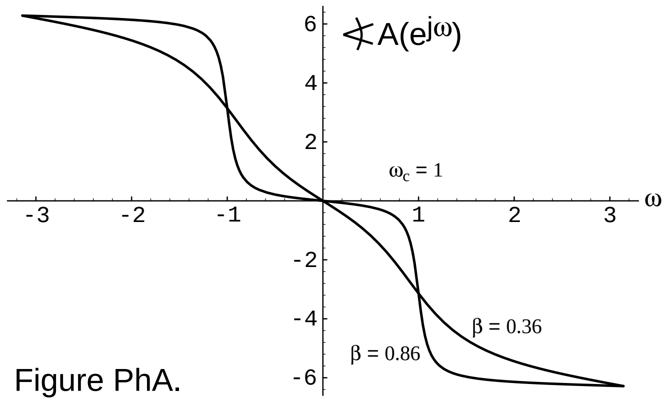


Figure PhA.

It is interesting that the allpass filter will shortly become integral to a parametric filter that is a minimum phase design. We also note in passing that the transfer to $D_r(z)$ from the input comprises only the denominator (the poles) of A(z):

$$D_r(z) = \frac{X(z)}{1 + \gamma(1+\beta)z^{-1} + \beta z^{-2}}$$

Back to the problem at hand, it is easily proven that

$$H_r(z) = \bigl(1 - A(z)\bigr) / 2 \qquad \text{(hrz a)}$$

and

$$H_n(z) = \bigl(1 + A(z)\bigr) / 2 \qquad \text{(hnz a)}$$

Substituting (ar0) into (hrz a) and (hnz a), we can respectively derive (hrz) and (hnz). This means that we can construct notch and perfect resonant filters from an allpass filter. We only have left to show that using the allpass we can construct cut, resonant, and boost filters as well. We will use the fact (ar1) that at the critical frequency,

$$\omega_c = \arccos(-\gamma) \qquad \text{(wc1)}$$

the allpass filter output is 180^o out of phase with respect to a steady state sinusoid at its input. This critical frequency becomes the normalized center radian frequency $\omega_c = 2\pi f_c T$ (for T the sample period) for all filter types employing the allpass filter-topology shown in Figure APDF1.

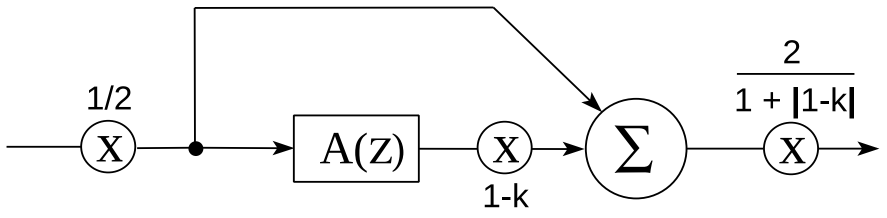


Figure APDF1. Cut, Notch, or Resonator Type Filter

We have introduced a new control coefficient, k. When k=0 this circuit in Figure APDF1 implements the notch (hnz a) exactly, and when k=2 this same circuit implements the perfect resonator (hrz a) exactly. Within these bounds, this control (for k<1) gives us the ability to specify the depth of the cut, leaving the magnitude at the extremal frequencies equal to 1. Using the same circuit for the resonator, we can control the depth of the skirts (when k>1) leaving the absolute peak frequency magnitude at precisely 1. These actions explain the unusual looking normalization at the output.

Table APDF1T

$k = 0$		notch, $H_n(z)$
$0 < k < 1$		cut, $H_c(z)$
$k = 1$	*yields*	input signal
$1 < k < 2$		resonator, $H_{b_{norm}}(z)$
$k = 2$		perfect resonator, $H_r(z)$

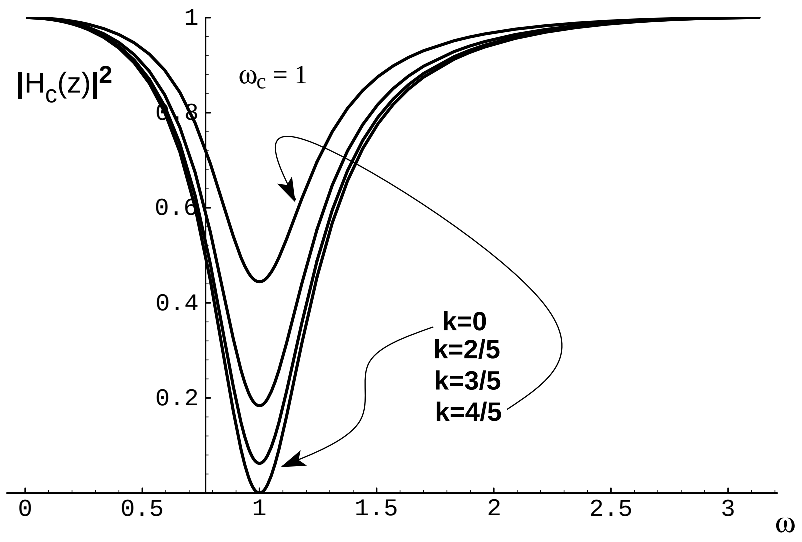


Figure Cutter. Cut Transfer for various values of k, Q=2.

The absolute cut depth = **(**1 **-** (1-k)**)** / (1 + |1-k|) = k / (2 - k) ;$0 \le k < 1$ (cd1)

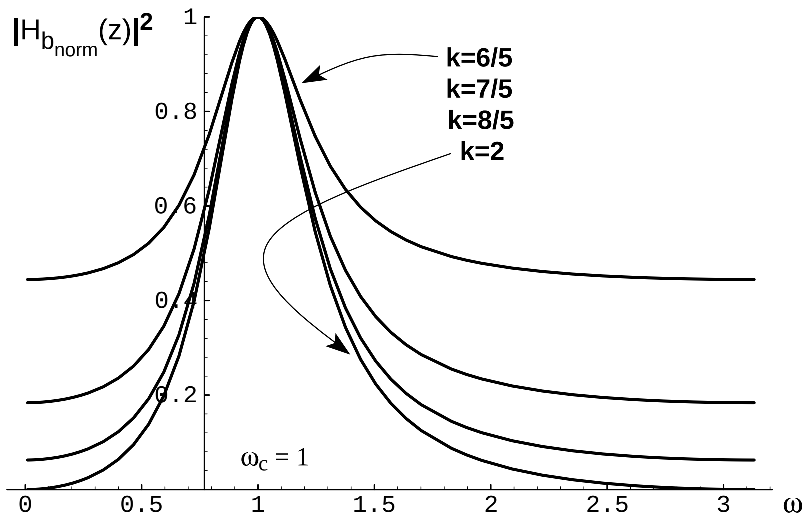


Figure Booster. Resonator Transfer for various values of k, Q=2.

The absolute skirt depth = **(**1 + (1-k)**)** / (1 + |1-k|) = (2 - k) / k ;$1 < k \le 2$ (sd1)

The cut and skirt depths must each be squared to resolve with Figure Cutter and Figure Booster. These depth equations are easily deduced from (wc1) and (ar1) respectively, and are independent of center frequency.

The **center frequency** is unequivocally determined by (wc1) for these cut, notch, resonator, and perfect resonator filters. This center frequency corresponds to the peak or trough extremum of the magnitude transfer evaluated on the unit circle in the **z**-plane.

The ordinate axis is drawn at the lower half-power excursion frequency in the two graphs of Figure Booster and Figure Cutter. The **half-power excursion frequencies (**the two cutoff frequencies**)** are given for the cut, notch, resonator, and the perfect resonator by

$$\cos(\omega_{2,1}) = \frac{(1+\beta)^2 \cos(\omega_c) \ \pm \ (\beta-1) \sqrt{\left(2(1+\beta^2) - (1+\beta)^2 \cos^2(\omega_c) \right)}}{2(1+\beta^2)} \qquad \text{(w21)}$$

$$\beta = \frac{1 - \tan\left(\omega_c / (2\,Q) \right)}{1 + \tan\left(\omega_c / (2\,Q) \right)} \qquad \text{(wbeta)}$$

Given a particular center frequency, the allpass lattice coefficient β precisely controls selectivity (the filter Q) for these cut, notch, resonator, and perfect resonator filters.[122] Whereas the lattice coefficient γ is a function only of ω_c as we see from inspection of (wc1), here we see that β is a function of both ω_c and Q as per our new definition, (RRR) and (NNN). On one hand, it is very good that we have discovered closed form mathematical relationships describing how to modify the two lattice coefficients to control the musical filter parameters. But from a musical control standpoint, we would like to have a way to decouple the filter coefficients so that only one of them controls the center frequency while the other controls only the selectivity parameter. (We almost have that in γ.) Later on, we will see another circuit topology which nearly reaches that ideal.

[122]This equation for β is exact in terms of the selectivity definition herein.

Regalia k Coefficients

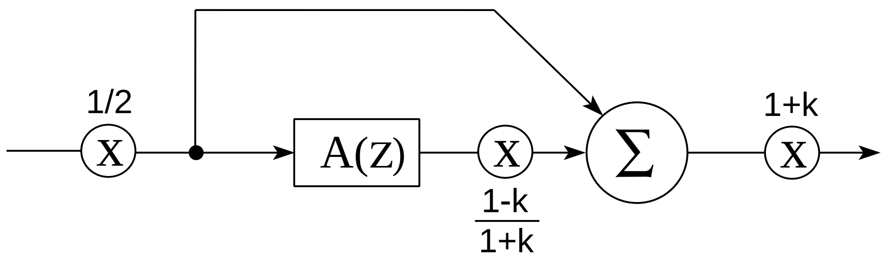


Figure APDF2. Cut, Notch, or Boost Type Filter

In [Regalia/Mitra] it was understood that a simple change of coefficient would result in a design which substitutes the parametric boost filter for the resonator, hence incurring the loss of the perfect resonator. Employing the same allpass filter-topology as before, the coefficients in Figure APDF2 are derived from those in Figure APDF1 via the substitution

$$(1-k) \;\to\; (1-k)\,/\,(1+k) \qquad \text{(sub1)}$$

and via a scaling by the boost factor k on the output, but only when k>1.

Table APDF2T

$k = 0$		notch, $H_n(z)$
$0 < k < 1$	*yields*	cut, $H_c(z)$
$k = 1$		input signal
$1 < k < \infty$		**boost**, $H_b(z)$

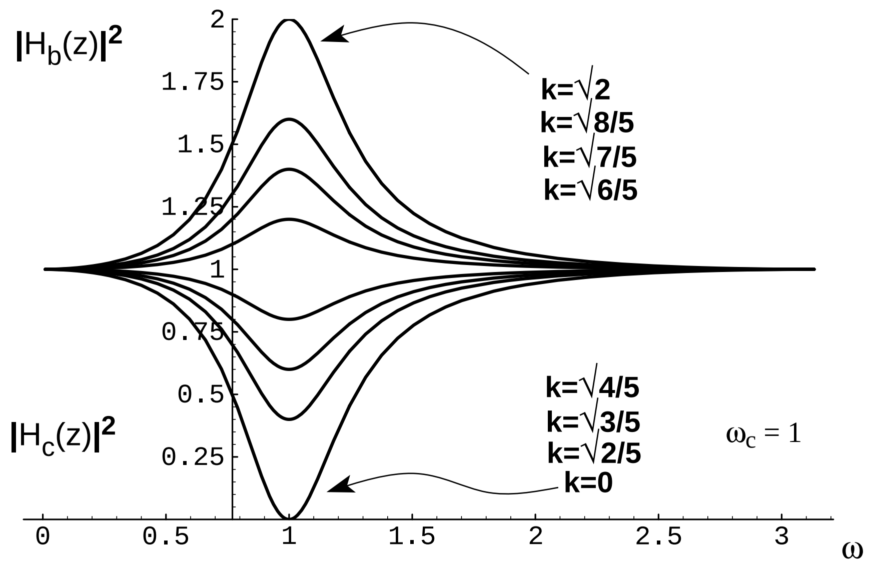


Figure Regal. Cut and Boost Transfers for various values of k. Q=2.

Regalia absolute cut depth $= k \qquad ;0 \le k < 1 \qquad$ (cd2)

Regalia absolute boost $= k \qquad ;1 < k < \infty \qquad$ (sd2)

The cut depth and boost must each be squared to resolve with Figure Regal. These results can be derived by substituting (sub1) into (cd1) and (sd1). As before these results are independent of center frequency. The combined plot in Figure Regal highlights the symmetry of the cut with the boost filters, hence the symmetry of Q. In general, the filters of Figure APDF1 and Figure APDF2 are minimum phase.

From the allpass characteristics (ar1) it can also be deduced that, independent of center frequency:

Regalia absolute skirt depth of boost filter = 1 $\quad ;1 < k < \infty$ (sd2a)

The ordinate axis is again drawn at the lower half-power excursion frequency (the lower cutoff frequency) in Figure Regal. The two cutoff frequencies $\omega_{2,1}$ for the boost filter are derived using a slightly different equation as compared with that for the resonator, (RRR), but it can be shown that the results are the same as before; i.e., (w21) remains valid under:

$$(|H_b(e^{j\omega})|^2 - 1) \ / \ (|H_b(e^{j\omega_c})|^2 - 1) = 1/2 \qquad \text{(NBNB)}$$

Equation (NBNB) does not reduce to the classical definition of cutoff frequency, however, because $H_b(e^{j\omega_c})$ by definition is never zero. Because we were able to derive the Regalia **k** coefficients (in Figure APDF2) from the circuit in Figure APDF1 while retaining the same topology, all the equations thus far remain applicable; i.e., for β, γ, ω_c , and $\omega_{2,1}$.

Lattice Topology in Practice

The foregoing allpass filter-topology constructed from the lattice suffers two drawbacks to its implementation: 1**)** The lattices produce spontaneous low-level audible zero-input limit cycle tones, 2**)** Lattice topologies are prone to signal overflow **(**hence conditional saturation in ESP2**)** at internal nodes before the allpass output has reached full scale.[123]

The first problem is solved by magnitude truncation of all lattice memory elements to 24 bits. [Smith] Since the automatic magnitude truncation feature of the ESP2 can only be invoked upon WR to external memory, it is clear that for these digital filters one does not want to use internal memory; rather it is highly desirable to use 24-bit external memory for lattice memory storage.[124]

[123]In [Jackson,ch.4.3], a novel topology for the perfect resonator is shown.

[124]If only 16-bit external memory is available, one is better off using 24 bit internal memory without magnitude truncation.

The second problem (overflow) is solved by scaling the lattice input as already shown in Figure APDF1 and Figure APDF2. When premature internal overflow persists **(**which is more likely for high Q, cut or boost**)**, it becomes necessary to provide a user controlled input-signal level adjustment.[125] Compensation will be required at the filter output, under separate user-control. Keep in mind that the cost of any output compensation is the concomitant amplification of the filter's internal signal truncation noise floor, so this input scaling process should be limited.

As an alternative to the use of the allpass lattice, we recall that the direct form I filter topology does not suffer from internal signal overflow because of its single accumulator having infinite headroom. [Dattorro,pg.857&pg.875][126] [Jackson,ch.11.3] In Figure DirectLattice, we show an implementation of the second-order allpass filter (ar0) comprising embedded direct form I first-order allpass sections.[127] This topology retains the dichotomy of the musical filter-coefficients as in the lattice, while employing the same coefficients. Empirically, we observe that the limit cycle tones produced by the direct form I are much quieter than those produced by the corresponding lattice in Figure Lattice, in general.[128] Truncation error feedback (not shown) in the direct form is known to further minimize limit cycle oscillation [Laakso], thus providing an alternative to magnitude truncation as a remedy.

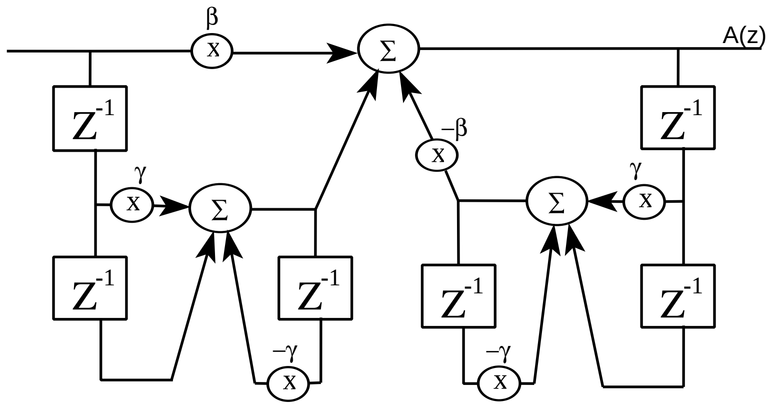


Figure DirectLattice. Direct Form I, 2nd Order Allpass Filter

For both the lattice and embedded direct form, stability is assured by $|\gamma|<1$ and $|\beta|<1$.

[125]This knob is probably required anyway to compensate for filter boosts that the user requests.

[126]Overflow is not always a bad thing.

[127]Conversion to direct form II using Rossum's technique [RossumPat.] would eliminate some memory elements while providing automatic input scaling. The scaling is necessary to prevent internal overflow in that topology.

[128]The direct form I may require error feedback [Dattorro] to be truncation noise competitive with the lattice, however.

8.5. The Chamberlin Filter Topology

Now we consider high fidelity musical filtering using a different topology and the musician's **all-pole** lowpass filter type. The musician's all-pole filter has antecedents in the electronic music industry appearing in currently renowned and vintage music synthesizers.[129] [Curtis] The filters we previously considered had zeroes in the transfer. We were concerned about the control of those filters as a musician might like to control them. Here we present an additional goal; i.e., to come up with filter coefficients each of which individually control only center frequency or selectivity (filter Q). To do so, we re-derive the Chamberlin all-pole (two-pole) lowpass filter topology entirely from the perspective of the discrete-time domain.[130]

The electronics engineer's lowpass has zeroes in the stopband and is very flat in the passband.[131] The stopband zeroes serve to provide high attenuation there. In contrast, musicians have a taste for peaked filters, even when the desired filter is of the lowpass variety. Because the musician's peak-center frequency is typically quite low (requiring poles closer to the unit circle), zeroes are largely unnecessary due to the relatively high attenuation at frequencies far away from the poles. When the peak-center frequency is high, on the other hand, the excursion of the all-pole filter magnitude transfer may not reach 3 dB; in fact, <u>when the peak-center frequency reaches</u> $\pi/2$ <u>the all-pole lowpass filter ceases being lowpass</u> because the magnitude transfer at π starts to exceed the transfer at DC.

Due to the fact that the Chamberlin filter is all-pole, there is little control over the rate of transition from pass to stop band. To increase the transition rate of the lowpass filter, the accepted solution is to cascade an identical all-pole filter. This works in practice because the musician's working range of the lowpass peak-center frequency is much less than $\pi/2$ for reasonable sample rates. Zeroes placed at the Nyquist frequency, for example, would have little impact over the musician's working range. Therefore the cascade is preferred to zeroes at Nyquist. Zeroes elsewhere in the stopband region would entail more computation, hence they are undesirable. In this development, we will consider only a single filter section.

[129]The classic Moog analog synthesizers, for example, employed fourth-order all-pole Voltage Controlled Filters (**VCF**). His constant-Q design was also known as the *Moog ladder*, after the appearance of the schematic. The Chamberlin all-pole design is reputed resident within the contemporary digital synthesizers by Peavey. A cascade of two Chamberlin filters can be considered as the digital counterpart to the Moog VCF because many of the same characteristics are shared; they are both all-pole constant-Q designs tuned by a single sweepable parameter.

[130]This filter was originally derived from an analog State-Variable filter by application of the Impulse-Invariant transformation. [Chamberlin] points out that this circuit topology simultaneously possesses a highpass and bandpass output at the nodes labelled *hp* and *bp*, respectively. We discuss only the lowpass filter function of this circuit in detail here.

[131]The Butterworth filter for example **(**which is a good choice for audio with regard to minimal ringing**)**, has all its zeroes at Nyquist.

Once again, we must refine our notion of cutoff frequency by relating it to half-power excursion, as before. We expect some kind of boosting transfer as shown in Figure Chlp. Notice that the filter is normalized to unity at DC.[132] For this filter type, we define the **passband excursion** from the value of the power transfer at DC to the peak value of the power transfer. Reminding ourselves that this transfer function is periodic in 2π, we then similarly define the **stopband excursion** from the peak to the value at Nyquist.[133] In Figure Chlp the half-power excursion points are indicated defining the musician's bandwidth of the all-pole lowpass filter.

We find the frequencies of the half-power excursion points **(**the **musical cutoff frequencies)** here much like we did before: The passband half-power excursion frequency is found solving (NCHL) for ω; we call this frequency ω_1.

$$\left(|H_{chx}(e^{j\omega})|^2 - 1 \right) \,/\, \left(|H_{chx}(e^{j\omega_c})|^2 - 1 \right) = 1/2 \qquad \text{(NCHL)}$$

Similarly, the solution to (NCHR) for the half-power excursion in the stopband we call ω_2.

$$\left(|H_{chx}(e^{j\omega})|^2 - |H_{chx}(-1)|^2 \right) \,/\, \left(|H_{chx}(e^{j\omega_c})|^2 - |H_{chx}(-1)|^2 \right) = 1/2 \qquad \text{(NCHR)}$$

Neither of these two cutoff frequency definitions, (NCHL) and (NCHR), reduce to the classical definition because none of the terms can go to zero in this all-pole design.

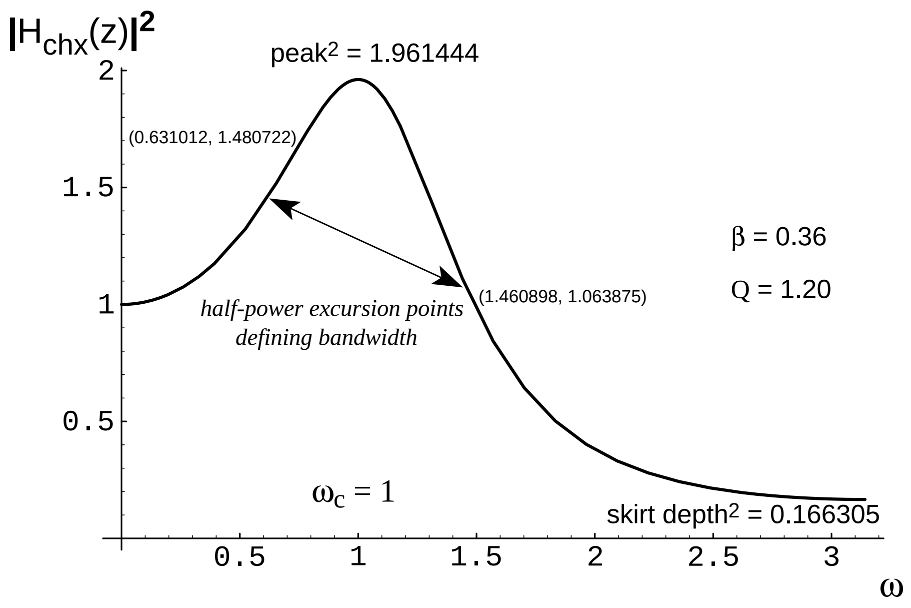


Figure Chlp. All-pole lowpass transfer.

[132]To bring the boost at the peak-center frequency ω_c down to unity, additional scaling is required beyond what we recommend in this exposition.

[133]The electronics engineer's transition band and stopband are merged in this development. Because of the lack of zeroes here, the electronics engineer's boundaries are not as clear. Also, the electronics engineer would measure bandwidth from DC, unlike our measurement.

We begin with a simpler transfer function having no zeroes, so we can expect some of the previously discovered equations to be different.

$$H_{chx}(z) = \frac{\alpha}{1 + \lambda z^{-1} + \beta z^{-2}} \qquad \text{(hchz)}$$

We seek the relationship of the coefficients to center frequency, ω_c, and Q.

$$\omega_c = \arccos\left(-(1 + \beta)\,\lambda / (4\,\beta) \right) \qquad \text{(wcx)}$$

If we express λ as

$$\lambda = 4\,\beta\,\gamma / (1 + \beta)$$

then we find the simpler expression for peak-center frequency.

$$\omega_c = \arccos\left(-\gamma \right) \qquad \text{(wc1)}$$

This equation for center frequency is the same as before, and both (wc1) and (wcx) are exact.

At the peak-center frequency, the magnitude square transfer reaches its peak height; exactly:

$$\max\left(|H_{chx}(e^{j\omega})|^2 \right) = \frac{4\,\alpha^2\,\beta}{(\beta - 1)^2\ (4\,\beta - \lambda^2)}$$

$$= \frac{\alpha^2\,(1 + \beta)^2}{(\beta - 1)^2 \left(1 + \beta^2 - 2\,\beta\,\cos(2\,\omega_c) \right)} \qquad \text{(maxchx)}$$

For the lowpass filter, we normalize the transfer to unity at DC, so α becomes:

$$\alpha = 1 + \lambda + \beta$$

The two musical cutoff frequencies were determined exactly using *Mathematica* [Wolfram][134] as:

$$\cos(\omega_{2,1}) = \cos(\omega_c) \ +,-\ \frac{(\cos,\sin)^2(\omega_c/2)\ \ (\beta-1)\ \sqrt{(2(1+\beta^2-2\beta\cos(2\,\omega_c)))}}{\sqrt{(1+\beta+8\beta^2+\beta^3+\beta^4 - \beta(1-6\beta+\beta^2)\cos(2\omega_c)\ +,-\ 4\beta(1+\beta)^2\cos(\omega_c))}}$$

(w21c)

[134]The extensions of *Mathematica* [Evans] to analog and digital signal processing are highly recommended.

We were not able to determine an exact expression for the β coefficient in terms of ω_c and Q as we did for γ, but the following guess turns out to be a good approximation:

$$\beta \approx \mathbf{(}\, 1 - \sin(\omega_c / (2Q)) \,\mathbf{)} \;/\; \mathbf{(}\, 1 + \sin(\omega_c / (2Q)) \,\mathbf{)} \qquad \text{(bapp)}$$

The plot in Figure Sheet shows that our expression for β is good over the recommended operating peak-center frequency range of $\omega_c = 0$ to $\pi/2$. To make this plot, we substitute the desired Q into the exact equations for ω_2 and ω_1 (w21c) using the approximation to β, (bapp), and then we sweep over ω_c .

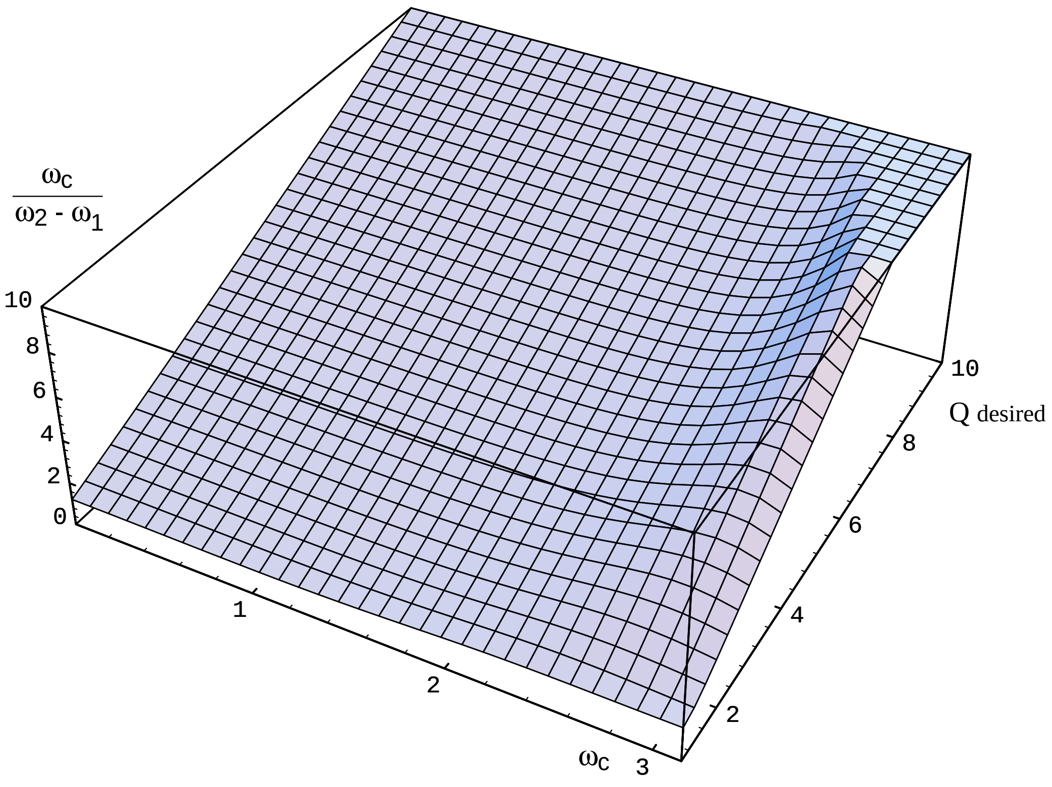


Figure Sheet. Actual all-pole filter-Q as function of center frequency and desired Q.

Had we the exact expression for β, then the sheet would be a taut plane having unit slope with respect to Q desired. But this approximation (bapp) is far better than some others in the literature. [Martin/Sun] [Petraglia] [Kwan/Martin] [Strawn,pg.123] The largest percentage errors in the recommended frequency range is for a desired Q of 1, having maxima from 21.8% to 27.6%.

Approximations

To achieve our stated goal of obtaining filter coefficients which individually control center frequency or filter selectivity, we now make series approximations to our expressions for the filter coefficients we have found thus far. The first several terms of the equivalent Maclaurin series for the z^{-1} coefficient in (hchz) are:

$$\lambda \approx -\left(2 - \omega_c/Q - \omega_c^2 + \omega_c^3 \left(1/(24\,Q^3) + 1/(2\,Q)\right) + \omega_c^4/12 + \ldots\right)$$

Using the good approximation (bapp) we find that the first several terms of the Maclaurin series for the z^{-2} coefficient in (hchz) are:

$$\beta \approx 1 - \omega_c/Q + \omega_c^2/(2\,Q^2) - 5\omega_c^3/(24\,Q^3) + \omega_c^4/(12\,Q^4) + \ldots$$

Figure CT shows the musical filter topology [Chamberlin][135] which implements a truncated series approximation to the desired filter coefficients, hence decoupling somewhat the control of ω_c and Q.

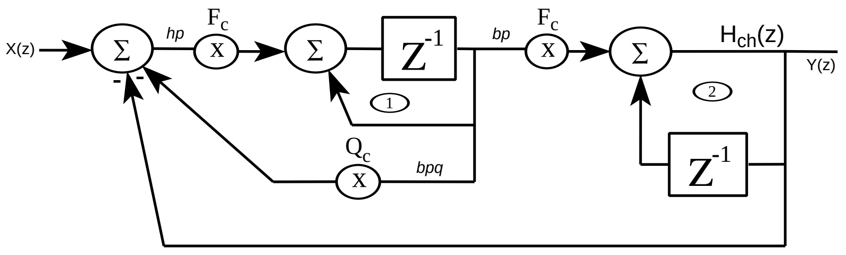


Figure CT. Chamberlin Topology, All-Pole Filter.

Trivial input/output compensation for internal overflow not shown.

See *Signal Overflow Analysis*.

The all-pole lowpass transfer function of this further approximation to (hchz) in Figure CT is:

$$H_{ch}(z) = \frac{Y(z)}{X(z)} = \frac{F_c^2\, z^{-1}}{1 - (2 - F_c\, Q_c - F_c^2)\, z^{-1} + (1 - F_c\, Q_c)\, z^{-2}} \approx z^{-1} H_{chx}(z) \qquad \text{(hchs)}$$

redefining: $\lambda = -(2 - F_c\, Q_c - F_c^2)\,, \qquad \beta = 1 - F_c\, Q_c\,,$ (hcsubs)

where, $F_c \approx 2 \sin(\omega_c/2)$ radian, $\qquad Q_c \approx 1/Q$

The transmogrified numerator of (hchz) now shows a delay operator in (hchs). This comes about because of the need to eliminate an otherwise delay-free loop in the circuit of Figure CT.[136] The numerator coefficient α has become F_c^2 to force (hchs) to unity at DC ($z=1$) as stipulated. This better approximation to F_c in (hcsubs) is from [Chamberlin] [Martin/Sun]. It becomes more exact for high Q where it is approximately ω_c when peak-center frequency is low. **(**An exact expression is given by (wcsx).**)**

[135]We adopt Chamberlin's notation.

[136]It is remarkable that the delay-free loop is eliminated without compromise to the digital filter coefficients, because delay-free loops can be troublesome when it is desired to maintain autonomy of the analog coefficients in the transformation.

Stability/Parameter Decoupling

Stability of complex conjugate poles demands the constraint:

$$0 \le (1 - F_c Q_c) < 1$$

;restated: $$0 < F_c \le 1/Q_c \qquad \text{(stab0)}$$

This condition is ascertained from (hchs) by demanding a pole radius[137] of magnitude less than one.

From previous considerations we presume that the **tuning range** for the all-pole lowpass filter is: $\pi/2 > \omega_c \ge 0$. If we substitute (hcsubs) into the equation for actual peak-center frequency ω_c (wcx) in this range, we discover that F_c and Q_c are not completely decoupled[138] except for very high selectivity ($Q_c \approx 0$):

$$0 < \cos(\omega_c) = \frac{4(1 - F_c Q_c) - F_c^2 (2 - Q_c^2) + F_c^3 Q_c}{4(1 - F_c Q_c)} \le 1 \qquad \text{(wcsx)}$$

We also find on the right-hand side that: $F_c \le 2/Q_c - Q_c$

and on the left-hand side: $F_c < \left(-Q_c + \sqrt{(8 + Q_c^2)} \right) / 2$

From the stability condition, (stab0), the minimum value of F_c is zero. This is achieved for the right-hand side inequality above when Q_c reaches $\sqrt{2}$. Thus we have an upper bound on Q_c to keep the actual peak-center frequency within the prescribed tuning range;

$$0 < Q_c < \sqrt{2}.$$

To maintain **complex conjugate poles** in (hchs)[139] we find that the following inequality holds:

$$0 \le (F_c + Q_c) \le 2$$

This is found by rooting the denominator of (hchs):

$$H_{ch}(z) = \frac{F_c^2 z^{-1}}{(1 - a z^{-1})(1 - a^* z^{-1})} \qquad \text{(hchsroot)}$$

where, $$a = 1 - F_c (F_c + Q_c)/2 + j F_c \sqrt{(1 - (F_c + Q_c)^2/4)}$$

Using the upper bound we found for Q_c we see that there will always be some finite range of F_c over which the poles will be complex conjugate.

[137]See (hrz) in Appendix V to see how to find the pole radius.

[138]A similar conclusion can be reached by solving (bapp) for Q in terms of F_c and Q_c via (hcsubs). We leave this as an exercise. Note: (bapp) is an approximation while (wcsx) is exact.

[139]i.e., for peak-center frequency away from but asymptotically including DC,

Combining all three criteria we finally conclude that to maintain a stable lowpass filter in the form of (hchs) having complex conjugate poles conforming to the prescribed tuning range, then the constraint must hold:

$$0 < F_c < \min\left(1/Q_c\,,\ 2 - Q_c\,,\ 2/Q_c - Q_c\,,\ \left(-Q_c + \sqrt{(8 + Q_c^2)}\right)/2 \right) \qquad \text{(stab1)}$$

We learn from (stab1) that an artificial upper bound on the value of Q_c equal to 1 yields a universal upper bound on F_c equal to 1 as well (corresponding to ω_c of about $\pi/3$). <u>We conclude that we can guarantee stability</u> of complex conjugate poles for any value of either coefficient as long as they individually remain within the range 0 -> 1.

Peak Gain

We examined the actual peak gain of $H_{ch}(e^{j\omega})$ over the prescribed ranges of F_c and Q_c (both, 0 -> 1) substituting the truncated series approximation coefficients (hcsubs) into (maxchx) then taking the square root. We found the peak gain to be greater than but approximately equal to Q_c^{-1}. The largest excess beyond this estimate occurs for low frequency and low Q, or for high frequency and high Q. At $F_c = 0.000001$ and $Q_c = 1$ we found the greatest excess at about 15.5% over Q_c^{-1}.

There is no separate control over peak gain in the Chamberlin topology; it is controlled indirectly through Q_c. We recommend a maximum peak gain of 24 dB, for musical purposes, corresponding to a minimum Q_c of about 0.0625 (filter Q=16).

8.5.1. Performance of the Chamberlin Filter

Now we wish to know whether our approximations are good enough. To do this, an engineer might calculate the root locus of the Chamberlin poles to see how closely their trajectory matches that of a second-order constant-Q filter. Instead we will repeat the musical analysis, as in Figure Sheet, relating Q and center frequency; but this time we will not use the ideal coefficients, rather we use the actual filter coefficients given by the truncated series approximations in (hcsubs).

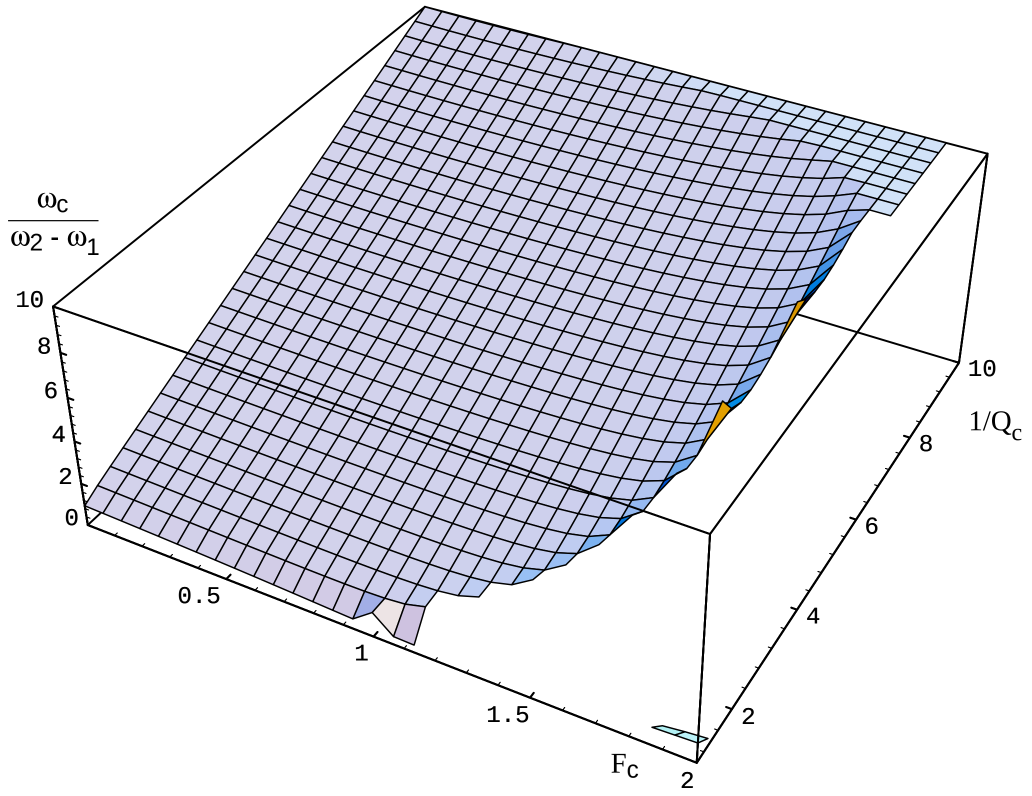


Figure ChamSheet. Actual Chamberlin circuit Q as function of F_c and Q_c^{-1}.

ω_c, ω_2, and ω_1 in Figure ChamSheet are calculated using the actual filter coefficients; evaluating (wcsx) and by substituting (hcsubs) into (w21c). Figure ChamSheet tells the whole story by relating actual circuit Q, (qqq), to the filter coefficients. Ideally, we are looking for a planar relationship. Nonetheless, the sheet is fairly unwrinkled up to $F_c \approx 1$, corresponding to a tuning range of peak-center frequency up to $\pi/3$. Further, it appears that for our purposes the selectivity parameter control, Q_c, is sufficiently decoupled from the tuning frequency control, F_c. Hence we can expect a good relationship of theory with practice in that region.[140]

[140]At a sample rate of 44.1 kHz, $\pi/3$ corresponds to a bandwidth of 7350 Hz. Considering that the topmost note of the pianoforte reaches only 4186 Hz, that tuning range is good enough for musical purposes.

Integrator Analysis

Generally speaking, it is not a good idea to implement an ideal digital integrator unless it can be guaranteed that there exists a zero of transmission across it at DC. This is certainly the case for integrator number 1 in Figure CT which has the required zero across it, but integrator number 2 has no such zero. In that case, one must then prove that there can exist no signal from any source having DC content upon arrival at the input to the integrator under scrutiny. Audio signals normally enter the digital circuit at the designated input node, but noise having DC content is routinely generated in any practical implementation at every node where signal truncation occurs. These noise sources most often appear in ESP2 at the input to each multiplier because multiplier inputs cannot accommodate double precision operands like the accumulators can.[141] The noise is accurately modelled as a deterministic signal, input to a fictitious adder resting in front of the multiplier. Figure CTN demonstrates the application of the noise model to one of the noise sources (e_2) on its way to integrator 2:

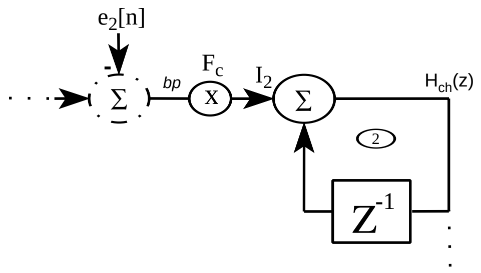


Figure CTN. Truncation Noise Source Model.

For the Chamberlin topology we have the remarkable result that the input to integrator number 2 never sees any signal having DC content. For verification, we now look at the most interesting signal which is the noise source at the input to the multiplier at node *bp*. There we have,

$$I_2(z) / e_2(z) = -F_c \,\mathbf{(}1 - (2 - F_c\, Q_c)\, z^{-1} + (1 - F_c\, Q_c)\, z^{-2}\mathbf{)} / \Delta \qquad \text{(nn2)}$$

where Δ is the denominator of (hchs). The transfer (nn2) has a zero of transmission at DC which can be proven by substituting z=1. The three other possible signal sources **(**at nodes *hp*, *bpq*, and the input**)** acquire a simple zero of transfer at DC in the form, $1\text{-}z^{-1}$, by the time they arrive at I_2.

[141]We presume no truncation post-accumulation for these integrators. This presumption is justified based on the alternative which is a leaky integrator, requiring a multiply in its loop.

Truncation Noise

The object of our noise analysis is to find the noise <u>generated by the circuit itself</u> which then appears at the observed output. The design goal of fidelity might be stated:

Criterion 1**)** *It is desired that the filter circuit generate no noise which would fall above the spectral noise floor due to quantization of the original input signal.*

Under this spectral interpretation of fidelity, the filter is then termed *transparent* to the signal. This one is the more conservative of the two criteria for fidelity that we will consider in this section.

Because the noise we are studying is deterministic (the sources are known), then when it occurs early in a filter circuit topology it undergoes some filtering as does the input signal itself. Using the truncation noise model described for the integrator analysis, then, we wish to know the transfer from each of the noise sources to the lowpass output. Having obtained this information, we can predict the frequency dependent amplification of each presumably wide-bandwidth noise source. Suppose, for example, that noise due to truncation were being generated at the 20-bit spectral level. Then <u>any given noise transfer would need to possess at least a 24 dB boost beyond unity</u> (in any frequency region) <u>before Criterion 1 were violated</u>, assuming 16-bit signal fidelity.[142]

Like the one shown in Figure CTN, there are three noise sources arising due to the truncation at each of the respective multiplier inputs; these are labelled *hp*, *bpq*, and *bp* in Figure CT:

1**)** The noise transfer from *hp* to the output is $-H_{ch}(z)$. This means that the noise transfer from *hp* is the same as the signal transfer (with sign inversion), which is certainly not a bad situation. This source is the largest of the noise contributors.

2**)** The noise transfer from *bpq* to the output is $Q_c\, H_{ch}(z)$. We know that the peak gain of $H_{ch}(z)$ is approximately Q_c^{-1}, which means that the peak of this noise transfer from *bpq* is about 1. This is good.

3**)** Lastly, we consider the noise transfer from *bp* to the output which can be written:

$$\frac{1 - z^{-1}}{-F_c\, z^{-1}}\, H_{ch}(z) \;-\; Q_c\, H_{ch}(z)$$

The first term introduces a zero at DC (but only to the noise), which is exactly what happens when we employ first-order truncation error feedback. [Dattorro] The zero squelches the noise in the low frequency region but boosts it in the high frequency region. Using (hcsubs) we find that the crossover point **(**the point at which $|1-z^{-1}|/F_c=1$**)**, above which the $1 - z^{-1}$ term begins to boost, is approximately ω_c. In this particular circumstance,

[142](20 - 16 bits) x 6 dB/bit = 24 dB. Another way of looking at this example would be to say that under fidelity Criterion 1, four extra bits would be required in the signal path to maintain filter transparency if any one of the noise transfers were capable of a 24 dB gain in any frequency region; i.e., one bit for every 6 dB of gain beyond unity transfer. Refer to Appendix IV for supporting noise concepts.

the second-order lowpass filter, $H_{ch}(z)$, kicks in above ω_c to remove the boosted noise. The second term of noise source 3 is like noise source 2 but opposite in polarity.[143]

Truncation Noise Power Gain

What we have thus far are the deterministic noise transfers which were of interest because of the way that the first criterion for fidelity was stated. The design goal of fidelity has a second, quantitative (statistical) interpretation which is stated:

Criterion 2**)** *It is desired that the total noise power generated by the filter circuit be less than the total noise power due to quantization of the original input signal.*

This is the interpretation ubiquitously ascribed as the -6 dB total noise power per individual bit. [Opp/Sch,pg.123] Here we determine the number of extra bits required to maintain signal fidelity using this criterion. To do so we must calculate what is commonly termed the **noise gain**. This is essentially an estimate of the <u>total</u> power boost of the internally generated, presumed spectrally white noise. Any total boost beyond unity is bad news, while any total gain of 1 or less is good.[144] The calculation is performed using the Parseval energy relation which integrates a noise transfer in either the time or frequency domain. Hence the calculation result is unitless since the integrand is a ratio. From the noise transfers, we already know that the worst offender is the noise source at *hp*. Substituting the poles of that transfer (hchsroot) into the integration results from [Opp/Sch,pg.187,357], we calculate the total noise power gain at *hp* as:

$$\boldsymbol{N_{hp}} = \frac{F_c^4\,\mathbf{(}a(1 - a^{*2}) - a^*(1 - a^2)\mathbf{)}}{(a - a^*)(1 - a^2)(1 - a^{*2})(1 - a^*a)} \qquad \text{(nstat)}$$

[143]Unfortunately, they probably will not cancel each other due to the fact that the former must pass through another truncation nonlinearity and a delay before it ever has the opportunity to combine with this one.

[144]A total noise gain of 1 says that the noise transfer will not increase the total amount of noise that it passes, but says nothing about the spectral distribution of the passed noise.

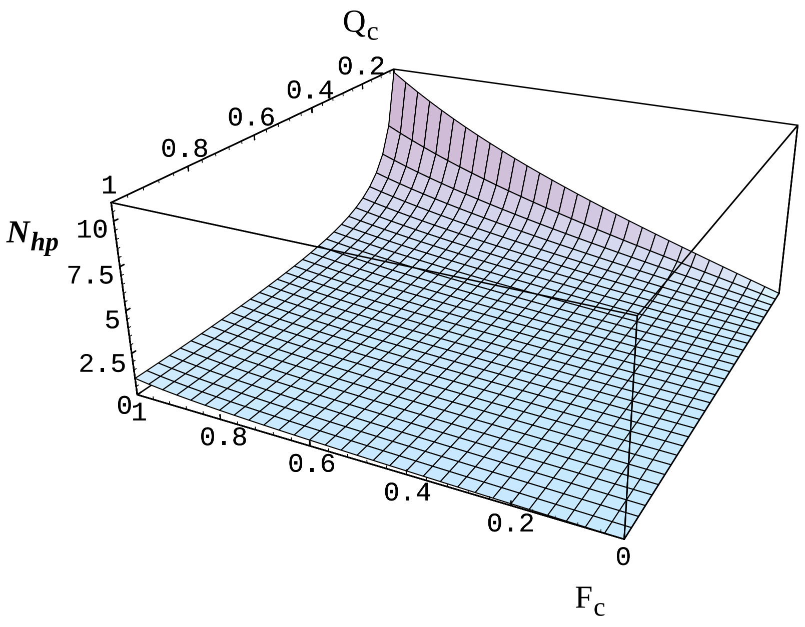


Figure NPlot. Showing $\boldsymbol{N_{hp}}$.

The vertical axis in Figure NPlot represents $\boldsymbol{N_{hp}}$ evaluated over the recommended operating ranges of F_c (0 -> 1) and Q_c (1 -> 0.0625). We see that there is no total noise gain for F_c=0, because the filter is shut down at that point. But we also see that the worst total noise power boost occurs at *hp* for high center frequency and high Q where $\boldsymbol{N_{hp}}$ reaches 10.7826, which translates[145] to 1.715 bits. The aggregate of the total noise power contributions, generated by all the presumably uncorrelated noise sources in the circuit of Figure CT, is calculated simply as $\boldsymbol{N_{hp} + N_{bpq} + N_{bp}}$. This sum likely requires somewhat more than 2 extra bits (beyond the desired fidelity) in the signal path to maintain fidelity using this latter criterion.

All in all, the all-pole lowpass Chamberlin topology looks good from the standpoint of truncation noise performance. This is true because the pole gain, which is the determinant of noise gain in general, does not exceed the desired filter peak-gain for the Chamberlin topology; the maximum pole gain is the maximum peak gain which we recommend to be 24 dB. The most significant improvement in noise performance would come from further minimization of the *hp* noise source, like we saw in the case of the *bp* noise source where truncation error feedback is built in.

[145]10 log(10.7826)/(20 log(2)) = $\log_2(\sqrt{10.7826})$ bits.

Limit Cycle Oscillation

Zero-input limit cycling arises due to ongoing signal quantization within a recursive topology; [Jackson] a nonlinear operation in an otherwise discrete linear system.[146] The filter coefficients are parameters to limit cycles but are not the cause. Limit cycles manifest themselves as annoying low-level tones at a circuit's outputs when no input signal is present. Signal quantization in ESP2 (as in most modern DSP chips) most often takes place at the single precision multiplier inputs where double precision operands cannot be accepted, so must be truncated.[147] The limit cycle tones can therefore be visualized to enter the topology at the same places as the truncation noise. One such input port is shown in Figure CTN. Like the truncation noise sources, if limit cycle tones occur early in a filter topology they will be filtered like the signal (at the same point of entry) itself.

We now know that limit cycle oscillation is minimized by truncation error feedback [Laakso] which was devised to minimize the amplification of truncation noise. [Dattorro] Essentially, error feedback introduces zeroes into the noise transfer function, strategically placed on the unit circle. *A reasonable hypothesis is, therefore, that with or without error feedback, if the noise transfer from a quantizer to the output has a term* $1\text{-}z^{-1}$ *then it provides some immunity to limit cycles as well as some squelching of truncation noise, both artifacts caused by that same quantizer*.[148] From our truncation noise analysis of the Chamberlin topology we see that only one of the truncation noise transfers has such a term. Hence, limit cycle tones cannot be ruled out. Our only recourse is to minimize their amplitude of oscillation by providing internal signal truncation at lower levels. This is tantamount to providing higher precision signal paths.

Signal Overflow Analysis

The study of overflow regards the observation of signal magnitude at sensitive nodes. Typically, one or several sensitive internal nodes may overflow (or underflow) sooner than the output. A saturation nonlinearity clips (appropriately full-scale positive or negative) the overflowed node as this is highly preferable to a two's complement wrap-around nonlinearity. The audible consequence of clipping at internal nodes is much more objectionable than clipping at the filter output, however, so it must be precluded completely. The sensitive internal nodes in ESP2 are, once again, the multiplier inputs because they cannot accept overflowed inputs like the accumulators can.[149] These are labelled *hp* and *bp* in Figure CT.

In our overflow analysis, what we are really interested in is the relationship of the sensitive nodes to the output. So we form a ratio, R, of transfers to the sensitive nodes with respect to the transfer to the output node:

[146]Signal quantization converts a discrete-time system to a *digital* system.

[147]Multipliers then produce double precision results which are usually fed to accumulators which can accept double precision inputs.

[148]In fact, [Laakso92] shows that any zero in the noise transfer provides some limit cycle immunity. The common solution to both artifacts suggests that the two phenomena are homologous.

[149]Recall that most contemporary accumulators are designed to tolerate infinite intermediate output overflow simply by virtue of non-saturating adders. [Jackson,ch.11.3] So saturation at an accumulator output (when necessary) is never performed upon intermediate accumulated results.

1**)** Node *hp*: Formulate $R_{hp}(z) = \mathbf{(}\ hp(z)/X(z)\ \mathbf{)} / H_{ch}(z) = (1-z^{-1})^2 / (F_c^2\ z^{-1})$

$|R_{hp}(z)| = \sin^2(\omega/2) / \sin^2(\omega_c/2)$

This describes a boost over the output at high frequencies. The <u>worst case</u> of overflow comes at the highest frequency ($z=-1$) and for a peak-center frequency ω_c at the top of its recommended range ($\pi/2$), for there the boost over the unity output is by the factor 2.

2**)** Node *bp*: Formulate $R_{bp}(z) = \mathbf{(}\ bp(z)/X(z)\ \mathbf{)} / H_{ch}(z) = (1-z^{-1}) / F_c$

$|R_{bp}(z)| = \sin(\omega/2) / \sin(\omega_c/2)$

Similarly, the worst case boost over the output is absolute $\sqrt{2}$.

<u>Based on this analysis</u>, <u>a simple technique to eliminate internal signal overflow</u>, <u>which we have found works quite well</u>, <u>is to precede the Chamberlin topology with a fixed input level attenuation=1/2 and to follow with a compensation factor of 2 at the output</u>. The output compensation amplifies the filter's internally generated truncation noise, however; i.e., the ratio of filtered signal power to the filter's own total noise degrades by 6 dB.

As shown in Figure Chlp, the Chamberlin lowpass is a boosting filter. For some signals the output may overflow. But overflow at the output will not always occur because the filtered signal may not have significant energy in the frequency region of the boost. Further, some small amount of output clipping is not offensive to the musician. Hence, <u>it is</u> not <u>desirable to automatically normalize the filter peak gain for the musician</u> by attenuating the input signal because there will be an objectionable loss in perceived volume. For then, the musician would demand a knob for output compensation; such a knob is emphatically discouraged because of the consequent amplification of internally generated truncation noise.

The most viable solution to the output overflow problem is to provide a filter input signal-level <u>user control</u>. The user then determines at what input level any clipping at the output becomes offensive. When an input level user control exists for a boosting filter, it becomes unnecessary to provide user-controlled output compensation to maximize the output signal level.

Estimate of Coefficient Width

Regarding F_c , using (hcsubs), resolve 1 Hz in 50 at a sample rate, F_s=44100.

$-\log_2(2(\sin(\pi\ 51/44100) - \sin(\pi\ 50/44100))) \approx$ 13 fractional bits.

In two's complement F_c requires 14 bits, but F_c is never negative.

Q_c minimum is 0.0625 for 24 dB maximum peak gain. This amount of gain requires at least 4 fractional bits; 5 bits in two's complement although Q_c is always positive. More bits are required for increased resolution of selectivity.

Estimate of Minimum Required Internal Signal Path Width

(@24 dB maximum peak gain)

Bit Budget	Attribute
N=16	output fidelity, assumed input signal quantization
n= 2	truncation noise immunity (Criterion 2)
o= 1	internal signal overflow prevention
m= 1	6 dB limit cycle suppression
+ r= max(0, (24/6)-n-m)	signal path LSBs for user-controlled input level attenuation[150]
N+n+o+m+r = 21 bits	total

This estimate maintains signal fidelity of N bits at the Chamberlin lowpass filter output. The filter internal signal path width can be minimized by reducing the maximum peak gain or by compromising the bit-fidelity requirement. It is interesting to note that under the more conservative fidelity Criterion 1, the estimate of the required total number of bits becomes 22; only one more bit (n=4, r=0). We see that the ESP2 single precision path width (24 bits) easily meets either fidelity requirement.

Assuming that the number of bits required to represent Q_c is less than or equal to the number of bits representing F_c , then the integrating accumulators must retain at least 14 + 21 - 1 = 34 bits[151] of precision under fidelity Criterion 2, and 35 bits of precision under Criterion 1.

[150]Assuming that the filter internal signal path resolution exceeds the input signal resolution, then no input signal information will be lost through the use of an input level control, provided that the attenuation is limited to the difference in resolution (6 dB per bit, 24 dB recommended).

[151]The subtraction of 1 is a consequence of the redundant sign bit in the product of a two's complement multiplication.

9. Sonically Pleasant Noise Generation

The technique we use to create white noise is called the *maximal length pseudo-random noise sequence.* [MacWilliams/Sloane] [Golomb] [Recipes][152] The noise is spectrally white because the autocorrelation function of a maximal length sequence is nearly a lone impulse. Once we have noise that is white, it is then filtered to create noise having color.[153] The power of the noise in any given small band of frequencies is low, however, since the total power is spread evenly over all frequencies. We find it necessary to boost small (filtered) bands by as much as 48 dB to achieve a healthy amplitude. Filtering a maximal length sequence using a lowpass filter having a cutoff frequency of approximately 400 Hz, for example, yields a very soothing rumble when amplified. Such narrow-band noise can be instantaneously modulated to get transposition in the frequency domain; achieved through multiplication in the time domain with a sinusoid of the desired transposition frequency, say f_o. This simple effect is reminiscent of gurgling brooks.

A recurring theme throughout this brief is that noise generators are more useful in pairs. Indeed, a simple way to enhance our gurgling brook above would be to transpose two independent generators; one of them to $f_o + f_\varepsilon$, the other to $f_o - f_\varepsilon$, where f_ε is only a few Hz. By so doing, we dismantle the spectral symmetry of the noise that would otherwise exist about f_o. In the spirit of our theme, we present two independent audio noise generators in Figure PN.

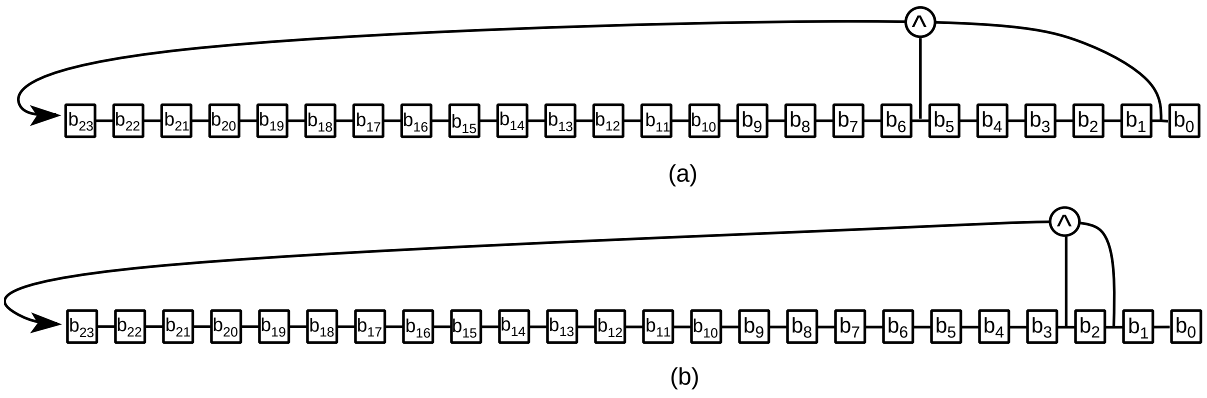

^
b23 b22 b21 b20 b19 b18 b17 b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
(a)
^
b23 b22 b21 b20 b19 b18 b17 b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
(b)

Figure PN. Two maximal length pseudo-random number generators.
For N-bit wide output, take N MSBs.
(a) $b_{23}[n+1] = b_6[n] \wedge b_1[n]$
(b) $b_{23}[n+1] = b_3[n] \wedge b_2[n]$

[152]Often referred to as a maximal length *PN* sequence in the literature.

[153]The term "pink noise" refers to a random process having a power spectrum that falls as 3 dB per octave, thus its power over any octave interval is the same. Often employed in the audio field because it is subjectively white, its power spectrum is proportional to 1/f in the linear frequency variable f. More algorithms and computer programs for colored noise generation and stochastic processes can be found in [Kasdin].

Figure PN (a) and (b) are two of many possible configurations. Examples (a) and (b) each show all the individual bits of one 24-bit register such as would be found inside the ESP2. The caret symbol ^ is C-language notation for exclusive-OR logic. At each sample period, the logic is performed on the two selected bits and then the whole register is shifted right, by one bit, accepting the logical result into the MSB. The 24-bit generator output for each example is simply the succession of 24-bit wide register values.[154] Using the logic shown in (a), a uniformly distributed non-repeating sequence of 23-bit values of length 2^{23}-1 is generated (ignoring b_0). In (b) the 22-bit wide sequence length is 2^{22}-1 (ignoring b_1 and b_0).

The generating logic equations are derived from Table PN; entries 23 and 22, respectively.[155] Table PN notation is for a *word length*-bit register. But Figure PN shows how the implementation is translated to a more suitable left-justified two's complement format. Hence we can fit any of the first 24 equations into a 24-bit register, and so on; we chose equations 23 and 22.

As suggested by the circuit in Figure PN, the value 0 cannot be produced using the Table PN logic. To start a generator, the significant bits of its register are initialized with any nonzero value. Using a different initial register value only shifts the corresponding sequence in time; i.e., the same sequence is started at a different phase of its cycle.

[154]The MSBs would be selected from each respective example in Figure PN for fewer than 24 bits-desired output from a 24-bit register. Smaller or larger registers may use different generator equations, in general.

[155]We did not choose entry 24 because it requires slightly more computation.

Table PN. Generating equations for maximal length sequences.[156]

word length	generator	word length	generator
1	$b_0[n+1] = b_0[n]$	33	$b_{32}[n+1] = b_{13} \wedge b_0[n]$
2	$b_1[n+1] = b_1 \wedge b_0[n]$	34	$b_{33}[n+1] = b_{15} \wedge b_{14} \wedge b_1 \wedge b_0[n]$
3	$b_2[n+1] = b_1 \wedge b_0[n]$	35	$b_{34}[n+1] = b_2 \wedge b_0[n]$
4	$b_3[n+1] = b_1 \wedge b_0[n]$	36	$b_{35}[n+1] = b_{11} \wedge b_0[n]$
5	$b_4[n+1] = b_2 \wedge b_0[n]$	37	$b_{36}[n+1] = b_{12} \wedge b_{10} \wedge b_2 \wedge b_0[n]$
6	$b_5[n+1] = b_1 \wedge b_0[n]$	38	$b_{37}[n+1] = b_6 \wedge b_5 \wedge b_1 \wedge b_0[n]$
7	$b_6[n+1] = b_1 \wedge b_0[n]$	39	$b_{38}[n+1] = b_4 \wedge b_0[n]$
8	$b_7[n+1] = b_6 \wedge b_5 \wedge b_1 \wedge b_0[n]$	40	$b_{39}[n+1] = b_{21} \wedge b_{19} \wedge b_2 \wedge b_0[n]$
9	$b_8[n+1] = b_4 \wedge b_0[n]$	41	$b_{40}[n+1] = b_3 \wedge b_0[n]$
10	$b_9[n+1] = b_3 \wedge b_0[n]$	42	$b_{41}[n+1] = b_5 \wedge b_4 \wedge b_3 \wedge b_2 \wedge b_1 \wedge b_0[n]$
11	$b_{10}[n+1] = b_2 \wedge b_0[n]$	43	$b_{42}[n+1] = b_6 \wedge b_4 \wedge b_3 \wedge b_0[n]$
12	$b_{11}[n+1] = b_7 \wedge b_4 \wedge b_3 \wedge b_0[n]$	44	$b_{43}[n+1] = b_6 \wedge b_5 \wedge b_2 \wedge b_0[n]$
13	$b_{12}[n+1] = b_4 \wedge b_3 \wedge b_1 \wedge b_0[n]$	45	$b_{44}[n+1] = b_4 \wedge b_3 \wedge b_1 \wedge b_0[n]$
14	$b_{13}[n+1] = b_{12} \wedge b_{11} \wedge b_1 \wedge b_0[n]$	46	$b_{45}[n+1] = b_8 \wedge b_5 \wedge b_3 \wedge b_2 \wedge b_1 \wedge b_0[n]$
15	$b_{14}[n+1] = b_1 \wedge b_0[n]$	47	$b_{46}[n+1] = b_5 \wedge b_0[n]$
16	$b_{15}[n+1] = b_5 \wedge b_3 \wedge b_2 \wedge b_0[n]$	48	$b_{47}[n+1] = b_7 \wedge b_5 \wedge b_4 \wedge b_2 \wedge b_1 \wedge b_0[n]$
17	$b_{16}[n+1] = b_3 \wedge b_0[n]$	49	$b_{48}[n+1] = b_6 \wedge b_5 \wedge b_4 \wedge b_0[n]$
18	$b_{17}[n+1] = b_7 \wedge b_0[n]$	50	$b_{49}[n+1] = b_4 \wedge b_3 \wedge b_2 \wedge b_0[n]$
19	$b_{18}[n+1] = b_6 \wedge b_5 \wedge b_1 \wedge b_0[n]$	51	$b_{50}[n+1] = b_6 \wedge b_3 \wedge b_1 \wedge b_0[n]$
20	$b_{19}[n+1] = b_3 \wedge b_0[n]$	52	$b_{51}[n+1] = b_3 \wedge b_0[n]$
21	$b_{20}[n+1] = b_2 \wedge b_0[n]$	53	$b_{52}[n+1] = b_6 \wedge b_2 \wedge b_1 \wedge b_0[n]$
22 (b)	$b_{21}[n+1] = b_1 \wedge b_0[n]$	54	$b_{53}[n+1] = b_6 \wedge b_5 \wedge b_4 \wedge b_3 \wedge b_2 \wedge b_0[n]$
23 (a)	$b_{22}[n+1] = b_5 \wedge b_0[n]$	55	$b_{54}[n+1] = b_6 \wedge b_2 \wedge b_1 \wedge b_0[n]$
24	$b_{23}[n+1] = b_4 \wedge b_3 \wedge b_1 \wedge b_0[n]$	56	$b_{55}[n+1] = b_7 \wedge b_4 \wedge b_2 \wedge b_0[n]$
25	$b_{24}[n+1] = b_3 \wedge b_0[n]$	57	$b_{56}[n+1] = b_5 \wedge b_3 \wedge b_2 \wedge b_0[n]$
26	$b_{25}[n+1] = b_8 \wedge b_7 \wedge b_1 \wedge b_0[n]$	58	$b_{57}[n+1] = b_6 \wedge b_5 \wedge b_1 \wedge b_0[n]$
27	$b_{26}[n+1] = b_8 \wedge b_7 \wedge b_1 \wedge b_0[n]$	59	$b_{58}[n+1] = b_6 \wedge b_5 \wedge b_4 \wedge b_3 \wedge b_1 \wedge b_0[n]$
28	$b_{27}[n+1] = b_3 \wedge b_0[n]$	60	$b_{59}[n+1] = b_1 \wedge b_0[n]$
29	$b_{28}[n+1] = b_2 \wedge b_0[n]$	61	$b_{60}[n+1] = b_5 \wedge b_2 \wedge b_1 \wedge b_0[n]$
30	$b_{29}[n+1] = b_{16} \wedge b_{15} \wedge b_1 \wedge b_0[n]$	62	$b_{61}[n+1] = b_6 \wedge b_5 \wedge b_3 \wedge b_0[n]$
31	$b_{30}[n+1] = b_3 \wedge b_0\ [n]$	63	$b_{62}[n+1] = b_1 \wedge b_0[n]$
32	$b_{31}[n+1] = b_{28} \wedge b_{27} \wedge b_1 \wedge b_0[n]$	64	$b_{63}[n+1] = b_4 \wedge b_3 \wedge b_1 \wedge b_0[n]$

[156] These logic equations are not unique, but all are suitable for audio. The time index, **[n]**, appears once for each logic equation in this table because all time indices are the same on the right hand side. Note that $b_i[n] = b_{i-1}[n+1]$.

rectangular probability density

Using the maximal length generators given in Table PN, the sequence length is then $2^{\textit{word length}}-1$; this is the longest possible for a given word length when the value 0 is excluded. The generating equations[157] given in Table PN each produce a unique independent maximal length sequence having a rectangular (rather 'uniform') probability density function (**PDF**). The probability density function [Cooper] is rectangular because each *word length*-bit value produced by the generator is equi-probable. The sequence is maximal length because every *word length*-bit value is generated just once before the sequence repeats. This zero-mean sequence is bipolar because of its two's complement representation.

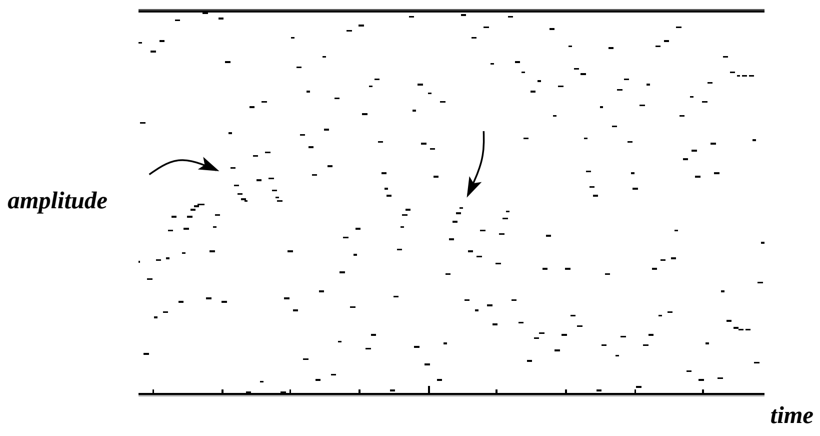


Figure Rect1. Bipolar Rectangular PDF noise showing localized correlation.

Figure Rect1 shows a brief segment of the noise sequence generated by the circuit in Figure PN (a), having variance $\sigma^2 = 1/3$. The plot reveals that maximal length pseudo-random noise generators are fundamentally relaxation-type oscillators. The arrows in the figure point out exponential decays toward zero that are fairly obvious in that microscopic view. This exponential relaxation is easily explained by the right shift by one bit that occurs as part of the algorithm on every sample generated. Since exponential decay is a common physical occurrence in the real world, that may help to explain why these noise sources are perceptually acceptable. The exponential artifacts are not decorrelated simply by choosing a different logic equation from Table PN.

The variance for the rectangular PDF is

$$\sigma^2 = \frac{(x_{max} - x_{min})^2}{12}$$

When $x_{min} = -x_{max}$ and $x_{max} = 1$, then $\sigma^2 = 1/3$.

[157]More generating equations can be found in [Recipes,ch.7.4].

triangular probability density

By linearly adding[158] two sequences from different generators,

$$y = x_1/2 + x_2/2 \qquad \text{(pn0)}$$

their rectangular probability density functions become convoluted; the sum y has a triangular probability density function which is useful for alleviating pseudo-noise amplitude modulation (‘breathing’) in dither applications. [Vanderkooy/Lipshitz]

Triangular probability density sequences generated in this manner remain spectrally white and retain exponential relaxation in time that is fairly obvious to the eye (see Figure Tri1). This zero-mean sequence has variance

$$\sigma^2 = \frac{y_{max}^2 - y_{max}y_{min} + y_{min}^2}{18}$$

When $y_{min} = -y_{max}$ and $y_{max} = 1$, then $\sigma^2 = 1/6$. The variance can be used to approximate the perceived loudness with respect to other PDFs.

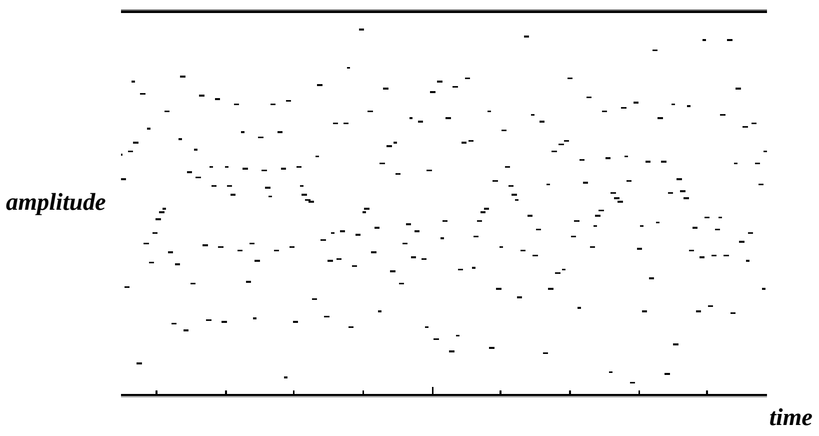


Figure Tri1. Bipolar Triangular PDF noise.

Figure Tri1 shows a segment of a triangular probability density noise process generated via (pn0), having variance $\sigma^2 = 1/6$. (There is no correspondence to the time axis of the other figures.)

[158]The two sequences must each be scaled by 1/2 prior to the addition to avoid clipping. The ESP2 AVG instruction is useful here because it has one extra bit of headroom, hence no incurred loss of the LSB of the sum.

Gaussian probability density

If we continue the process of summing a number of independent sequences, then the *central limit theorem*[159] predicts that the probability density function of the sum approaches Gaussian (rather 'Normal') density as the number gets large.[160] That is why recursive filtering of a rectangular density process also tends towards the Gaussian.

In [Recipes,ch.7.2] is discussed the *Box-Muller* method of transforming rectangular into Gaussian probability density using a fixed and finite amount of computation. As in the triangular case, two independent x sequences having rectangular probability density are first generated: x_1 and x_2. By the principle of transformation of variables we apply,

$$y = \sigma \sqrt{-2 \ln(x_1)} \cos(2\pi x_2) \qquad \text{(pn1)}$$

The x sequences in (pn1) must now span the domain of positive values ($0.0^+ \rightarrow 1.0$). The best way to do this is prior to the transformation, we let

$$x \rightarrow -0.5\, x + 0.5 \qquad \text{(pn2)}$$

This linear operation on the originally bipolar x should not upset its autocorrelation function. Hence the x sequence remains spectrally white in its conversion to a unipolar sequence via (pn2). We wish to avoid an exception process for *ln*(0), hence the negative coefficient in (pn2), assuming **q**23 two's complement format where 1.0 exactly is not expressible.

The bipolar zero-mean sequence y in (pn1) is a Gaussian density sequence having a specified standard deviation σ (and variance σ^2). The y sequence is bipolar because of the $\cos(\cdot)$ evaluation. Only the localized correlation property (see Figure Rect1) of the x sequences carries over to the y sequence.[161] The spectral content of y becomes lowpass; the autocorrelation functions of the x sequences are not preserved in y after the nonlinear transformation (pn1).

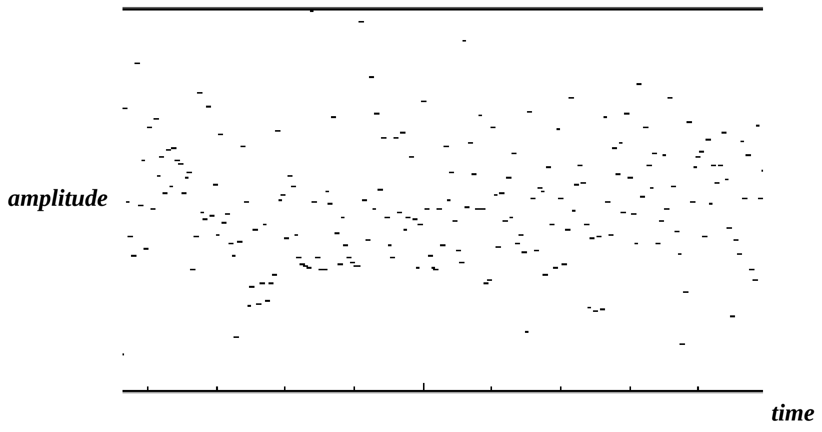


Figure Gauss1. Bipolar Gaussian PDF noise.

[159]from the field of probability and statistics [Cooper] [Recipes],

[160]To implement this procedure we find that it is necessary to scale each sequence by the number of sequences summed if absolutely no clipping is desired.

[161]We wish that were not the case because the localized correlation can be a liability for some sensitive applications.

In [Recipes] it is explained that the $\sin(\cdot)$ function can replace $\cos(\cdot)$ in (pn1), and a technique is shown to eliminate the trigonometric function evaluation altogether.[162] Because $ln(x_1)$ can approach negative infinity, we must choose σ according to the dynamic range of the number system we are working within. But there will always be outliers generated by (pn1). A remedy might be to allow <u>some</u> numerical wrapping of y in those events. The Gaussian probability density function will then become aliased[163] and a tradeoff must be made between the aberration to the density function caused by clipping and that caused by wrapping.

Figure Gauss1 shows a brief segment from a Gaussian noise process having $\sigma = 0.3$ generated via (pn1). Conditional saturation (clipping) was used to handle outliers.

Sonic Musing

The author presently owns about five electronic gadgets for sonic noise generation. These devices are analog and emit electronically amplified sound through a speaker, except for one which consists of an encased fan driven by an induction motor. The fan-type unit was first purchased from Marpac Corp., NC, in 1977. Their analog noise generators, which amplify and filter transistor noise, were found to be suitable substitutes for the earlier electromechanical version. The noise generators are used when sleeping, concentrating, or to block out unwanted ambient disturbances.

Several years ago, the same company announced a new ‘improved’ digital sonic noise generator. Not only is the noise generation digital and algorithmic, but the front panel is now computerized. After living with the unit for a while, the author concluded that the noise was not pleasant, so back it went to the factory for repair. Discussion with the Marpac engineer revealed that the device was operating within normal parameters and was not in need of repair. He confided that he too liked the analog units better. We no longer use that digital device, however, because it sounds bad. Nonetheless, the Marpac company claims growing sales of the digital unit.

The author has put noise generation algorithms into production for commercial musical products, and into the fitting system[164] that supports a customized hearing aid. When evaluating various algorithms, we used our ears. Our purpose for transcribing these events is to convey the knowledge that digital methods for noise generation are <u>not</u> inherently bad, but some algorithms sound better than others. We know that when properly filtered, the fundamental algorithm presented herein sounds as good as any analog method of noise generation; that was our ultimate criterion for choosing it.

[162]The technique to eliminate the trigonometric function evaluation reverts the domain of the x sequences back to (-1.0 -> 1.0), but now entails a new exception process: It must detect when the sum of the squares of x_1 and x_2 is in excess of unity. In that case, the pair is discarded and a new pair is generated. (This fact was first brought to the author’s attention by Paul Hargrove.) This decision process brings with it a <u>variable</u> execution time that may be undesirable in some circumstances.

[163] much like the analog sinc() function becomes the aliased sinc() function (the Dirichlet kernel) in the discrete domain,

[164] *Audio’D*, Hearing Solutions, Paoli PA, USA

References

[Adams] Robert Adams, Tom Kwan, ‘Theory and VLSI Architectures for Asynchronous Sample-Rate Converters’, Journal of the Audio Engineering Society, vol.41, no.7/8, pg.539, 1993 July, AES, 60 East 42nd St., New York NY, 10165 USA

[AnalogDevices] *ADSP-2100 Family - Applications Handbook*, vol.1, 1989, Analog Devices, Inc., DSP Division, One Technology Way, Norwood MA, 02062 USA

[Andreas] David C. Andreas, ‘VLSI Implementation of a One-Stage 64:1 FIR Decimator’, 89th Convention of the Audio Engineering Society, Los Angeles, Preprint 2976 (G-4), 1990 September 21-25, AES, 60 East 42nd St., New York NY, 10165 USA

[Beigel] Michael L. Beigel, ‘A Digital "Phase Shifter" for Musical Applications, Using the Bell Labs (Alles-Fischer) Digital Filter Module’, Journal of the Audio Engineering Society, vol.27, no.9, pp.673-676, 1979 September, AES, 60 East 42nd St., New York NY, 10165 USA

[Blesser] Barry Blesser, personal communication.

[Blesser/Bader] Barry A. Blesser, Karl-Otto Bader, *Electric Reverberation Apparatus*, United States Patent No.4,181,820, January 1, 1980

[Burrus/Parks] C.S. Burrus, T.W. Parks, *DFT/FFT and Convolution Algorithms,* 1985, Wiley-Interscience Publication, John Wiley and Sons, New York NY, USA

[Chamberlin] Hal Chamberlin, *Musical Applications of Microprocessors*, 1980, Hayden Books, 4300 West 62nd St., Indianapolis IN, 46268 USA

[Cooper] George R. Cooper, Clare D. McGillem, *Probabilistic Methods of Signal and System Analysis*, second edition, 1986, Oxford University Press, 2001 Evans Rd., Cary NC, 27513 USA

[Crochiere/Rabiner] Ronald E. Crochiere, Lawrence R. Rabiner, *Multirate Digital Signal Processing,* 1983, Prentice-Hall, Inc., Englewood Cliffs NJ, 07632 USA

[Curtis] *CEM3328 Four Pole Low Pass VCF*, 1983, Curtis Electromusic Specialties, 110 Highland Ave., Los Gatos CA, 95030 USA

[Dattorro] Jon Dattorro, ‘The Implementation of Recursive Digital Filters for High-Fidelity Audio’, Journal of the Audio Engineering Society, vol.36, no.11, pp.851-878, 1988 November, AES, 60 East 42nd St., New York NY, 10165 USA.
---------, Corrections to above, Journal of the AES, vol.37, no.6, pg.486, 1989 June
---------, Addendum to above, Journal of the AES, vol.38, no.3, pp.149-151, 1990 March

[Dattorro2400] Jon Dattorro, ‘Using Digital Signal Processor Chips in a Stereo Audio Time Compressor/Expander’, 83rd Convention of the Audio Engineering Society, New York, Preprint 2500 (M-6), 1987 October 16-19, AES, 60 East 42nd St., New York NY, 10165 USA

[Dattorro89] Jon Dattorro, ‘The Implementation of Digital Filters for High Fidelity Audio, Part II - FIR’, *Audio in Digital Times*, The Proceedings of the Audio Engineering Society 7th International Conference, Toronto, pp.168-180, 1989 May 14-17, AES, 60 East 42nd St., New York NY, 10165 USA

[DattorroPat.] Jon C. Dattorro, Albert J. Charpentier, David C. Andreas, ‘Decimation Filter as for a Sigma-Delta Analog-to-Digital Converter’, United States Patent No.5,027,306, June 25, 1991

[Evans] B. L. Evans, L. J. Karam, K. A. West, J. H. McClellan, ‘Learning Signals and Systems with Mathematica’, IEEE *Transactions on Education*, vol.36, no.1, pp.72-78, 1993 February

[Fliege/Wintermantel] Norbert J. Fliege, Jorg Wintermantel, ‘Complex Digital Oscillators and FSK Modulators’, IEEE *Transactions on Signal Processing*, vol.40, no.2, pp.333-342, 1992 February

[Gardner], W. G. Gardner, ‘Reverberation Algorithms’, in *Applications of Signal Processing to Audio and Acoustics*, M. Kahrs, K. Brandenburg, editors, 1996, Kluwer Academic Publishers, 101 Philip Dr., Assinippi Park, Norwell MA, 02061 USA

[Golomb] Solomon W. Golomb, editor, *Digital Communications with Space Applications*, 1964, Peninsula Publishing, Los Altos CA, 94022 USA

[Gordon/Smith] John W. Gordon, Julius Orion Smith, ‘A Sine Generation Algorithm for VLSI Applications’, Proceedings of the International Computer Music Conference, 1985, pp.165-168, ICMA, 2040 Polk St. #330, San Francisco CA, 94109 USA

[Gray] Robert M. Gray, *Source Coding Theory*, 1990, Kluwer Academic Publishers, 101 Philip Dr., Assinippi Park, Norwell MA, 02061 USA

[Griesinger] David Griesinger, ‘Practical Processors and Programs for Digital Reverberation’, *Audio in Digital Times*, The Proceedings of the Audio Engineering Society 7th International Conference, Toronto, pp.187-195, 1989 May 14-17, AES, 60 East 42nd St., New York NY, 10165 USA

[Haija/Ibrahim] A. I. Abu-el-Haija, M. M. Al-Ibrahim, ‘Improving Performance of Digital Sinusoidal Oscillators by means of Error Feedback Circuits’, IEEE *Transactions on Circuits and Systems*, vol.CAS-33, no.4, pp.373-380, 1986 April

[Hamming] R. W. Hamming, *Numerical Methods for Scientists and Engineers*, second edition, 1973, Dover Publications, Inc., 31 East 2nd St., Mineola, NY 11501

[Harris] Fredric J. Harris, ‘On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform’, *Proceedings* of the IEEE, vol.66, no.1, pp.51-84, 1978 January

[Hartmann] W. M. Hartmann, ‘Flanging and Phasers’, Journal of the Audio Engineering Society, vol.26, no.6, pp.439-443, 1978 June, AES, 60 East 42nd St., New York NY, 10165 USA

[Jackson] Leland B. Jackson, *Digital Filters and Signal Processing*, third edition, 1996, Kluwer Academic Publishers, 101 Philip Dr., Assinippi Park, Norwell MA, 02061 USA

[Kasdin] N. Jeremy Kasdin, ‘Discrete Simulation of Colored Noise and Stochastic Processes and $1/f^{\alpha}$ Power Law Noise Generation’, *Proceedings* of the IEEE, vol.83, no.5, pp.800-827, 1995 May

[Kim/Sung] S. Kim, W. Sung, ‘A Floating-Point to Fixed-Point Assembly Program Translator for the TMS 320C25’, IEEE *Transactions on Circuits and Systems II*, vol.41, no.11, pp.730-739, 1994 November

[Kwan/Martin] T. Kwan, K. Martin, ‘Adaptive Detection and Enhancement of Multiple Sinusoids Using a Cascade IIR Filter’, IEEE *Transactions on Circuits and Systems*, vol.36, no.7, pp.937-947, 1989 July

[Laakso] Timo I. Laakso, *Error Feedback for Reduction of Quantization Errors due to Arithmetic Operations in Recursive Digital Filters*, Thesis for the degree of Doctor of Technology, Report 9, Otaniemi 1991, Helsinki University of Technology, Laboratory of Signal Processing and Computer Technology, Otakaari 5A, SF-02150 Espoo, Finland

[Laakso92] Timo I. Laakso, Paulo S. R. Diniz, Iiro Hartimo, Trajano C. Macedo, Jr., 'Elimination of Zero-Input and Constant-Input Limit Cycles in Single-Quantizer Recursive Filter Structures', IEEE *Transactions on Circuits and Systems*-II, vol.39, no.9, pp.638-646, 1992 September

[Laakso/Välimäki] Timo I. Laakso, Vesa Välimäki, Matti Karjalainen, Unto K. Laine, 'Splitting the Unit Delay - Tools for fractional delay filter design',
IEEE *Signal Processing Magazine*, vol.13, no.1, pp.30-60, 1996 January

[Lee] Francis F. Lee, 'Time Compression and Expansion of Speech by the Sampling Method', Journal of the Audio Engineering Society, vol.20, no.9, pp.738-742, 1972 November, AES, 60 East 42nd St., New York NY, 10165 USA.
---------, also in *Speech Enhancement*, Jae S. Lim, editor, pp.286-290, 1983, Prentice-Hall, Inc., Englewood Cliffs NJ, 07632 USA

[Luthra] A. Luthra, 'Extension of Parseval's Relation to Nonuniform Sampling', IEEE *Transactions on Acoustics, Speech, and Signal Processing*, vol.36, no.12, 1988 December

[MacWilliams/Sloane] F. Jessie MacWilliams, Neil J. A. Sloane, 'Pseudo-Random Sequences and Arrays', *Proceedings* of the IEEE, vol.64, no.12, pg.1715,
1976 December

[Martin/Sun] K. W. Martin, M. T. Sun, 'Adaptive Filters Suitable for Real-Time Spectral Analysis', IEEE *Transactions on Circuits and Systems*, vol.CAS-33, no.2, pp.218-229, 1986 February (Also published in IEEE *Journal on Solid-State Circuits*, vol.SC-21, no.1)

[Moore] F. Richard Moore, *Elements of Computer Music*, 1990, Prentice-Hall, Inc., Englewood Cliffs NJ, 07632 USA

[Moorer] J. Andrew Moorer, personal communication. Also in [Strawn,pg.70].

[Oppenheim] Alan V. Oppenheim, editor, *Applications of Digital Signal Processing*, 1978, Prentice-Hall, Inc., Englewood Cliffs NJ, 07632 USA

[Opp/Sch] Alan V. Oppenheim, Ronald W. Schafer, *Discrete-Time Signal Processing,* 1989, Prentice-Hall, Inc., Englewood Cliffs NJ, 07632 USA

[Petraglia] M. R. Petraglia, S. K. Mitra, J. Szczupak, 'Adaptive Sinusoid Detection Using IIR Notch Filters and Multirate Techniques', IEEE *Transactions on Circuits and Systems II*, vol.41, no.11, pp.709-717, 1994 November

[Ramstad] Tor A. Ramstad, 'Digital Methods for Conversion Between Arbitrary Sampling Frequencies', IEEE *Transactions on Acoustics, Speech, and Signal Processing*, vol.ASSP-32, no.3, pp.577-591, 1984 June

[Recipes] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, *Numerical Recipes in C,* second edition, 1992, Cambridge University Press, 40 West 20th St., New York NY, 10011 USA

[Regalia/Mitra] Phillip A. Regalia, Sanjit K. Mitra, 'Tunable Digital Frequency Response Equalization Filters', IEEE *Transactions on Acoustics, Speech, and Signal Processing,* vol.ASSP-35, no.1, pp.118-120, 1987 January

[Renfors/Saramäki] M. Renfors, T. Saramäki, 'Recursive N^{th}-Band Digital Filters - Parts I and II', IEEE *Transactions on Circuits and Systems*, vol.CAS-34, no.1, pp.24-51, 1987 January

[Rossum] Dave Rossum, 'An Analysis of Pitch Shifting Algorithms', 87th Convention of the Audio Engineering Society, New York, Preprint 2843 (J-6), 1989 October 18-21, AES, 60 East 42nd St., New York NY, 10165 USA

[RossumPat.] David P. Rossum, *Dynamic Digital IIR Audio Filter and Method Which Provides Dynamic Digital Filtering for Audio Signals*, United States Patent No.5,170,369, December 8, 1992

[Schafer/Rabiner] Ronald W. Schafer, Lawrence R. Rabiner, 'A Digital Signal Processing Approach to Interpolation', *Proceedings* of the IEEE, vol.61, pp.692-702, 1973 June

[Schroeder] Manfred Schroeder, ‘Natural Sounding Artificial Reverberation’, Journal of the Audio Engineering Society, vol.10, no.3, 1962 July, AES, 60 East 42nd St., New York NY, 10165 USA

[Slater] Robert Slater, *Portraits in Silicon*, 1987, The MIT Press, Massachusetts Institute of Technology, Cambridge MA, 02142 USA

[Smith] Julius Orion Smith, ‘Elimination of Limit Cycles and Overflow Oscillations in Time-Varying Lattice and Ladder Digital Filters’, *Music Applications of Digital Waveguides*, Report No. STAN-M-39, 1987 May, Center for Computer Research in Music and Acoustics (CCRMA), Dept. Music, Stanford University, Stanford CA, 94305 USA

[Smith/Cook] Julius Orion Smith, Perry R. Cook, ‘The Second-Order Digital Waveguide Oscillator’, Proceedings of the International Computer Music Conference, 1992, pp.150-153, ICMA, 2040 Polk St. #330, San Francisco CA, 94109 USA

[SmithIII] Julius Orion Smith III, *Techniques for Digital Filter Design and System Identification with Application to the Violin*, Report No. STAN-M-14, 1983 June, Center for Computer Research in Music and Acoustics (CCRMA), Dept. Music, Stanford University, Stanford CA, 94305 USA

[J.O.Smith] Julius O. Smith, *An Allpass Approach to Digital Phasing and Flanging*, Report No. STAN-M-21, Spring 1982, Center for Computer Research in Music and Acoustics (CCRMA), Dept. Music, Stanford University, Stanford CA, 94305 USA

[Smith/Friedlander] Julius O. Smith, Benjamin Friedlander, ‘Adaptive, Interpolated Time-Delay Estimation’, IEEE *Transactions on Aerospace and Electronic Systems,* vol.21, no.2, pp.180-199, 1985 March

[Steiglitz] Ken Steiglitz, *A Digital Signal Processing Primer*, 1996, Addison-Wesley Publishing Company, 2725 Sand Hill Road, Menlo Park CA, 94025 USA

[Strawn] John Strawn, editor, *Digital Audio Signal Processing,* 1985, A-R Editions, Inc., 801 Deming Way, Madison Wisconsin, 53717 USA

[Thoen] Bradford K. Thoen, ‘Practical Aspects of Digital Sinewave Generation Using a Second-Order Difference Equation’, IEEE *Transactions on Circuits and Systems,* vol.CAS-32, no.5, pp.510-511, 1985 May

[Vaidyanathan] P.P. Vaidyanathan, *Multirate Systems and Filter Banks*, 1993, Prentice-Hall, Inc., Englewood Cliffs NJ, 07632 USA

[Välimäki/Laakso] V. Välimäki, T. I. Laakso, J. Mackenzie, 'Elimination of Transients in Time-Varying Allpass Fractional Delay Filters with Application to Digital Waveguide Modeling', Proceedings of the International Computer Music Conference, Banff, 1995, pp.327-334, ICMA, 2040 Polk St. #330, San Francisco CA, 94109 USA

[Vanderkooy/Lipshitz] John Vanderkooy, Stanley P. Lipshitz, 'Digital Dither: Signal Processing with Resolution Far Below the Least Significant Bit', *Audio in Digital Times*, The Proceedings of the Audio Engineering Society 7th International Conference, Toronto, pp.87-96, 1989 May 14-17, AES, 60 East 42nd St., New York NY, 10165 USA

[Wolfram] Wolfram Research, Inc., *Mathematica*, Version 2, 1994, Champaign Illinois, 61820 USA

Acknowledgements

The ESP2 chip was designed over a four year period beginning in 1990. The three principal designers were David Andreas, Jon Dattorro, and J. William Mauchly **(**son of John V. Mauchly, inventor of the ENIAC computer [Slater]**)**. Andreas is the principal architect of the circuitry. He made the early computer models so as to simulate the chip prior to fabrication, and he provided the original Hardware Specification. Later in the development, David Dura planned the layout and implemented the chip in its entirety employing custom VLSI. Dura dedicated himself to chip debug and revision. Michael Arnao wrote the assembler. The ESP2 rests on the shoulders of its predecessor, the ESP chip. Although the two chips are divergent, the essence of that first design is retained. In this light, the pioneering work of Stephen Hoge and David DiOrio is recognized.

Appendix I
ESP2 Scheduler Notes: Rules for DIL and DOL Availability

We list the rules for DIL and DOL availability in scheduling external-memory accesses:

Glossary

AGEN read: an AGEN operation that reads the contents of an external-memory location into a DIL

AGEN write: an AGEN operation that writes the contents of a DOL to an external-memory location

defensive criterion: a criterion for determining DIL or DOL availability that tests whether the external-memory access currently being scheduled would be corrupted by a previously-scheduled access

initial operand: the topmost unquoted operand (A, B, C, D, E, F, or G) in a quoted-reference chain; this operand initializes the chain

isolated reference: an external-memory reference that is not the initial operand in a quoted-reference chain

latency: the difference between the number of the instruction where a register (GPR, AOR, or SPR) is written by a function unit (MAC, ALU, or AGEN) and the number of the instruction where the contents of that register become available as a source operand in a particular function unit

protective criterion: a criterion for determining DIL or DOL availability that tests whether a previously-scheduled external-memory access would be corrupted by the access currently being scheduled

quoted-reference chain: one or more quoted similar external-memory references preceded by an unquoted similar external-memory reference

request instruction: an instruction where an external-memory reference appears in the MAC unit or ALU as a source or destination operand

similar external-memory references: two or more external-memory references that refer to the same element of the same external-memory array in the same region and that have the same UPDATE BASE mode and same Plus-One addressing mode

target instruction: an instruction where an AGEN operation is to be coded by the assembler in response to an external-memory request from the MAC unit or ALU

terminal operand: the bottommost operand (A, B, D, or E) in a quoted-reference chain

Variables

’`A_TO_B = k`’ means that the latency from unit `A` to unit `B` is `k` instructions:

```
MAC_TO_MAC       = 1
MAC_TO_ALU       = 1
MAC_TO_AGEN      = 0

ALU_TO_MAC       = 2
ALU_TO_ALU       = 1
ALU_TO_AGEN      = 1

AGEN_TO_MAC      = 2
AGEN_TO_ALU      = 2
```

’`A_PREC_B = k`’ means that if a DOL is written by unit `B` at instruction `n`, and is subsequently written by unit `A` at instruction `n+k`, then at instruction (`n+k+A_TO_AGEN`) the DOL will contain data from the latter write:

```
MAC_PREC_MAC     = 1
MAC_PREC_ALU     = 1

ALU_PREC_MAC     = 0
ALU_PREC_ALU     = 1
```

Scheduling Operations

There are four principal scheduling operations, performed in the following order:

[1] quoted-reference chains
[2] priority external-memory writes
[3] external-memory reads
[4] remaining external-memory writes

Quoted-Reference Chains

Quoted-reference chains are formed starting at instruction `progsize-1` (the last instruction of a program), and working toward instruction `0` (the first instruction). In locating terminal operands, source-code scanning is performed from right to left, traversing operands in the following order: B, A, E, D. The first occurrence of a particular quoted external-memory reference thus encountered becomes the terminal operand of a chain. Further quoted similar external-memory references become links in the chain, which extends to the closest unquoted similar reference for which inter-unit latencies are resolved; this reference becomes the initial operand of the chain.

Priority External-Memory Writes

Priority external-memory writes are scheduled starting at instruction `0` and working toward instruction `progsize-1`. A function unit, `UNIT` (`MAC` or `ALU`), can write DOLn at instruction `request` and schedule an AGEN priority-write at instruction `target` if

[1] `request+UNIT_TO_AGEN = target < progsize`

[2] no AGEN instruction has already been placed at target instruction

and all defensive and protective criteria (as described below) for external-memory writes are true.

External-Memory Reads

External-memory reads are scheduled starting at instruction `progsize-1` and working toward instruction `0`. A function unit, `UNIT` (`MAC` or `ALU`), can read DILn at instruction `request` and schedule an AGEN read at instruction `target` if

[1] `0 <= target+AGEN_TO_UNIT <= request < progsize`

[2] no AGEN instruction has already been placed at target instruction

and all the following defensive and protective criteria are true:

defensive criteria:

[1] the external-memory read to be scheduled is an isolated reference, and (`target == request-AGEN_TO_UNIT`) or neither MAC unit nor ALU has already scheduled an AGEN read using DILn in range

`[target+1, request-AGEN_TO_UNIT]`

or

the external-memory read to be scheduled initializes a quoted-reference chain, and neither MAC unit nor ALU has already scheduled an AGEN read using DILn in range

`[target+1, lastref-AGEN_TO_MAC]` if terminal operand in MAC unit

`[target+1, lastref-AGEN_TO_ALU]` if terminal operand in ALU

protective criteria:

[1] neither MAC unit nor ALU has already scheduled an AGEN read using DILn in range

`[0, target-1]`

from request instruction in range

`[target+AGEN_TO_UNIT, progsize-1]`

[2] neither MAC unit nor ALU has already assigned DILn to a quoted-reference chain whose AGEN read occurs in range

`[0, target-1]`

and that terminates in range

`[target+AGEN_TO_UNIT, progsize-1]`

Note that if AGEN reads are scheduled starting from the bottom of the program and working toward the top, and the closest available AGEN instruction is always chosen as the target instruction, then both protective criteria are always true.

External-Memory Writes

External-memory writes are scheduled starting at instruction `0` and working toward instruction `progsize-1`. A function unit, `UNIT` (`MAC` or `ALU`), can write DOLn at instruction `request` and schedule an AGEN write at instruction `target` if

[1] `0 <= request+UNIT_TO_AGEN <= target < progsize`

[2] no AGEN instruction has already been placed at target instruction

and all the following defensive and protective criteria are true:

defensive criteria:

[1] the external-memory write to be scheduled is an isolated reference, and MAC unit has not already written DOLn in range

```
[request+MAC_PREC_UNIT, target-MAC_TO_AGEN]
```

or

the external-memory write to be scheduled initializes a quoted-reference chain, and MAC unit has not already written DOLn in range

```
[request+MAC_PREC_UNIT,
        MAX(lastref-MAC_TO_MAC, target-MAC_TO_AGEN)]
```
if terminal operand in MAC unit

```
[request+MAC_PREC_UNIT,
        MAX(lastref-MAC_TO_ALU, target-MAC_TO_AGEN)]
```
if terminal operand in ALU

where `lastref` is the number of the instruction of the last quoted reference in the chain

[2] the external-memory write to be scheduled is an isolated reference, and ALU has not already written DOLn in range

```
[request+ALU_PREC_UNIT, target-ALU_TO_AGEN]
```

or

the external-memory write to be scheduled initializes a quoted-reference chain, and ALU has not already written DOLn in range

```
[request+ALU_PREC_UNIT,
        MAX(lastref-ALU_TO_MAC, target-ALU_TO_AGEN)]
```
if terminal operand in MAC unit

```
[request+ALU_PREC_UNIT,
        MAX(lastref-ALU_TO_ALU, target-ALU_TO_AGEN)]
```
if terminal operand in ALU

protective criteria:

[1] MAC unit has not already scheduled an AGEN write using DOLn in range

    [request+UNIT_TO_AGEN, progsize-1]

from request instruction in range

    [0, request-UNIT_PREC_MAC]

[2] ALU has not already scheduled an AGEN write using DOLn in range

    [request+UNIT_TO_AGEN, progsize-1]

from request instruction in range

    [0, request-UNIT_PREC_ALU]

[3] MAC unit has not already assigned DOLn to a quoted-reference chain that terminates in range

`[request+UNIT_TO_MAC, progsize-1]` if terminal operand in MAC unit

`[request+UNIT_TO_ALU, progsize-1]` if terminal operand in ALU

and begins in range

`[0, request-UNIT_PREC_MAC]`

[4] ALU has not already assigned DOLn to a quoted-reference chain that terminates in range

`[request+UNIT_TO_MAC, progsize-1]` if terminal operand in MAC unit

`[request+UNIT_TO_ALU, progsize-1]` if terminal operand in ALU

and begins in range

`[0, request-UNIT_PREC_ALU]`

.

Appendix II
Important Theorems

We present the following without proof as they are evident to the most casual observer:

Reagan’s Theorem
There exist two types of matter in the universe: matter and doesn’t matter.

Lemma predicating existence of the Discrete Time Laplace Transform
There never lived a mathematician named Z.

Quod Erat Demonstrandum.

Appendix III

Reverb Recollections

Ref.: Reverberation Application

December 21, 1994

Dear Jon,

What you wrote was fine but it stimulated my memory of additional snippets. Feel free to use what you want.

I had a personal conversation with Manfred Schroeder in the late 1970's and I asked him the question about what the phrase 'maximal incommensurate' delay values meant, as it appeared in one of his reverberation papers. His answer was particularly interesting. This is a paraphrase based on my tired memory:

"*We did the electronic reverberation for amusement because we thought it would be fun. Since it took the better part of a day to do 10 seconds of reverberation we only ran one sample of music. The notion of delay time selections was random in that we just picked a bunch of numbers and there was no mathematical basis. We just wanted to prove it could be done.*"

He never related this work to his more profound mathematical and perceptual research, specifically the work on the required 3 eigentone per Hz density and the frequency-phase statistics in a random physical space.

The original EMT reverberator, model 250 operating at a 32 kHz sample rate, used a main memory of 8k words and the required eigentone density was emulated entirely by randomizing delaylines. Another interesting fact is that colorless reverberation, using allpass structures, is perceptually not colorless. Even white noise passed through an allpass will not sound like real white noise. When passed through many such allpass structures, it in fact sounds like a machine shop rather than random noise. It still measures spectrally flat. The reason is that frequency regions get bunched in time. It is very much like a chirped sine wave in radar having a purely flat spectrum but is very different from white noise. The 2nd and higher order statistical terms out of an allpass are very, very different from a real random process. The utility of an allpass is to pass all frequencies through so that each allpass can see the same spectral density, otherwise comb peaks would align and dominate. Parallel structures of non-allpass elements achieve a similar issue in that each structure gets fed the full spectrum. Allpass elements are more critical for small delay values. An allpass within a larger loop must be used with great care since it has a sine-like variation in group delay. Hence, the effective loop time, and reverberation time, varies with frequency. After many trips around the loop, the result will be very colored.

Schroeder's had several analyses about reverberation but his 3 eigentone per Hz theory, which maps to 3 seconds of memory, can be looked at in many ways. His result was empirical based on listening tests. Consider two eigentones, or poles, separated by 1 Hz and located in the **s**-plane with a real part of -10 Hz. When excited this will produce two damped exponentially decaying frequencies which differ by 1 Hz. Hence there will be a 1 Hz envelope beat which is clearly audible. Now add other eigentones, randomly spaced but still at a distance of -10 Hz. Assume 10 such eigentone. All of them will beat with each other producing a random envelope having a spectrum which is crudely flat from 0 to 10 Hz. One can do this simulation in closed form with variable excitation of each eigentone. Schroeder's result actually depends on the nominal reverberation time since that determines how many eigentone will get excited by a narrow-band input. In the early reverberation boxes, with only 150 ms of reverberation, typically only a few tones would be excited. The envelope had a clear periodicity of 6 Hz on average. It sounds bad. Some regions had only two eigentones excited with a distance of 2 Hz which was even worse. Development was much more exciting with such limited memory. Today one can use 1 second of DRAM memory. Many simpler structures will thus produce good reverberation.

The perceptual simulations deviated from physical reality in many ways. For example, a natural three dimensional space has an increasing eigentone density which is proportional to the square of frequency. All electronic simulations tend to have a constant density. The reason is that in a three dimensional space, the speed of sound along a dimension is proportional to the sine of the wave front direction, whereas in an electronic structure it is always constant.

Well, that is what I remember so do what you wish with it. Best of luck.

Sincerely yours,

Barry Blesser

Blesser Associates
Electronics & Software Consultants
P.O. Box 155
Belmont MA, 02178 USA

Appendix IV

Truncation Noise Spectral Level vs. Total Noise Power

Ref.: Chamberlin Filter Topology in Musical Filtering

We seek the relationship of truncation noise spectral level, $\boldsymbol{n}(e^{j\omega T})$, to total noise power, $\boldsymbol{N}$, because we wish to prove that for every additional 6 dB of $\boldsymbol{S/N}$, the noise spectral level drops by the same amount. The analysis of truncation noise is much like that of quantization noise. [Opp/Sch,pg.353] It is interesting that the classical derivation of quantization noise [Opp/Sch,pg.119] is statistical and does not include any consideration of sample rate. We therefore expect our results to reflect this.

Here we regard the truncation noise signal as deterministic, and we consider only <u>normalized</u> signal and noise spectra such that a real sinusoid of infinite duration has finite average power $\boldsymbol{S}=1/2$ (and a complex sinusoid has average power $\boldsymbol{S}=1$). We then pose the problem: 10 log $\boldsymbol{S/N}$ = 96 dB for a real sinusoid; find the truncation noise power spectral level. 96 dB is the approximate expected signal to noise ratio for a 16-bit fidelity signal (6 dB per bit), [Opp/Sch,pg.123] and we set $\boldsymbol{S}=1$ to simplify the discussion. We then find the total noise power solving,

$$10 \log \boldsymbol{N} = -96$$

where[165]

$$\boldsymbol{N} = \frac{T}{M}\int_0^{F_s} |\,\boldsymbol{n}(e^{j\omega T})\,|^2 \, df \qquad \text{(IV-n0)}$$

$$\boldsymbol{N} = M\,T\int_0^{F_s} \boldsymbol{n}_1^2 \; df \qquad \text{(IV-n1)}$$

where $T = 1/F_s$, and where M is the number of points in the data record, and where $\boldsymbol{n}_1$ is the normalized average noise spectral magnitude; i.e., the average of $|\,\boldsymbol{n}(e^{j\omega T})\,|\,/\,M$ over frequency. Solving, we find:

$$\boldsymbol{n}_1^{\,2} = \boldsymbol{N}/M \qquad \text{(IV-n2)}$$

which is independent of sample rate, as expected. This equation indicates that noise power spectral level is proportional to total noise power. This means that if total noise power drops by 6 dB, then so will its spectral level (assuming M constant). We needed to know this to justify the claim (supporting fidelity Criterion 1 in our example from the text) that the average difference in truncation noise spectral level between a 20-bit and a 16-bit quantized signal is 24 dB (=(20-16) x 6.02 dB).

[165]There is confusion in the various DSP textbooks regarding expression (IV-n0). We believe that [Jackson] is correct in his expression for the Parseval energy relation, while the modification to his expression to yield the power relation comes from [Luthra].

Appendix V

Filter Errata in the Literature

Ref.: Musical Filtering

A mistake has been perpetuated regarding the center frequency of the second-order digital filter.

The polar representation of complex conjugate filter poles is often found, correctly written, as

$$z_{pole} = R\,e^{\pm j\theta} \qquad \text{(zp1)}$$

The erroneous hypothesis can be recognized wherever filter center radian frequency, ω_c , is ascribed to the radian **pole angle**, θ . Hence, the distinction between <u>center</u> frequency and <u>pole</u> frequency is obscured in the literature.[166] There it is argued that for high selectivity, this distinction is of little practical importance. But that tenuous assumption of practical equivalence has, consequently, promulgated specious theoretical conclusions within the audio community.

One such **erroneous** conclusion is that the perfect resonator transfer (hrz), for any given center frequency, does <u>not</u> have a peak magnitude exactly equal to 1 when evaluated on the unit circle in the **z**-plane. The errant proof evaluates (hrz) at the resonant frequency (i.e., at $z=e^{j\theta}$) in complete disregard of the **pole radius**, R. Evaluation at the true center frequency (i.e., at $z=e^{j\omega_c}$) given by (wc1) shows that conclusion to be false; i.e., it is true that the perfect resonator as given by (hrz) <u>always</u> has a peak gain of exactly unity.

We can establish a correspondence between pole and center frequency by equating the denominator of the general second-order transfer function (written in terms of the pole radius and angle (zp1) [Jackson,ch.4.3] [Dattorro,(27)]) to the perfect resonator (hrz):[167]

$$H_r(z) = \frac{(1/2)(1-\beta)\ (1 - z^{-2})}{1 - 2R\cos(\theta)\,z^{-1} + R^2 z^{-2}} = \frac{(1/2)(1-\beta)\ (1 - z^{-2})}{1 + \gamma(1+\beta)\,z^{-1} + \beta z^{-2}} \qquad \text{(hrz)}$$

[166]The **resonant frequency** is that frequency at which a filter rings when excited by an impulse. The resonant frequency is the pole frequency, which is the same as the pole angle, θ, in the **z**-plane. The center frequency is the frequency at peak magnitude transfer in the steady state; when a filter is excited by a sinusoid of infinite duration. Generally, the center and resonant frequencies are not identical. [Steiglitz,ch.5.5]

[167]The poles occur in complex conjugate pairs when the filter coefficients of z (i.e., γ and β) are real. When the filter coefficients are real, then it is easily shown that the filter's impulse response must also be real.

The identifications we can easily deduce using (wc1) are:

$$\beta = R^2$$

$$\gamma = \frac{-2 R \cos(\theta)}{1 + R^2} = -\cos(\omega_c)$$

This proves that the only instance where center frequency ω_c would be the same as second-order pole (or resonant) frequency θ is for conjugate poles right on the unit circle (R = 1). But in that circumstance one has an oscillator, not a filter.

These results can be extended to the resonator in general. Similar conclusions can be drawn from examination of the second-order all-pole transfer (hchz), and from the second-order all-zero transfer such as the one in Figure BN.

An instance where center frequency is identical to pole frequency is for the case of the first-order resonator. The equivalence is independent of pole radius, R, unlike the second-order case. This instance may be the reason for the propagation of the erratum regarding the second-order case. The transfer of the first-order resonator is:

$$F_r(z) = \frac{1 - R}{1 - R e^{j\theta} z^{-1}}$$

This filter has only one pole. But notice that the one filter coefficient is complex, in general. Hence the impulse response of this filter cannot be real. One may surmise that there must be some interaction among multiple poles in the **z**-plane that destroys radial symmetry.

Appendix VI

Assembly Listing

The **.**lst file output of the assembler (**.**e2 source file input, by convention) shows the assembled code in 'half-human, half-machine' format. Each operand address field (A,B,C,D,E,F,G) and each opcode is shown in a numerical form before it is packed into one large 96-bit microinstruction word. The listing reports statistics such as the program size, number of registers used, their initialization contents, everybody's address, declared but unused stuff, etc.

For a detailed example, see the Linear Interpolation Application.

Assembler Invocation

The command line switch list appears whenever the assembler is invoked without a command line source file (**.**e2) input:

```
ESP2 Assembler Version X.XX [3 June 2054]
usage: esp2asm -abdhlnorsx <src-file>
switches:
-a [<file>]  produce AGEN listing file (.agn)
-b           produce binary object (.bin) instead of C object (.o)
-d           include debug data in object file
-h  <file>   rename header file (.hdr)
-l [<file>]  produce listing file (.lst)
-n           do not inhibit .o, .bin, .hdr, or .asm upon error
-o  <file>   rename object file
-r <rev no.> place chip revision number in .o, .bin, and .asm
-s [<file>]  produce simulator input file (.asm)
-x  <files>  delete named files prior to assembly
```

Regarding only the notation immediately above:
[ ]   means that what is enclosed is optional.
< > means that what is enclosed is non-literal; so <file> means your file name.

The binary object (**.**bin) is in Ensoniq format. The C object (**.**o) is an ASCII file, a C-language compatible declaration of the binary object. Each output file holds assembly information equivalent to the other.

Appendix VII
Assembly Errors and Warnings

Need expansion and explanation of all errors and warning messages in esperrs.h

Appendix VIII
ESP2 Assembler functions requiring implementation or revision beyond Version 0.52

Assembler BUGS

-BUG: (v0.51)

```
error: 'VoiceL': external-memory reference in expression
error: '&': argument must be external-memory reference
error: 'VoiceL': uninitialized AOR (v0.51)
DEFREGION  V
   VoiceL[2048]
   delayL = &VoiceL[350] q8        !this should be legal
```

-BUG: (v0.51)

```
error: 'MINUS1': uninitialized SPR
DEFGPR
      k = MINUS1                      ! cases: ZERO, HALF, MINUS1, ONE
```

-BUG: (v0.51) PROGRAM *name* should appear in header file. Program *name* should be made a unique identifier, as are other user-definable names; hence a reserved symbol.

-BUG: (from tunable speaker algorithm. v0.51) "error: 'm_mid2': syntax error"

```
DEFCONST   Q_MID2 = .5
DEFGPR  Q_mid2 = Q_MID2 q21 @$63     Qm_mid2 = -Q_MID2 q21 @$64
```

-BUG: (v0.52)

```
Preset AORs              Address  Contents  (Base 10)  Region  Reference
MAC $004, line 61 . . . . $2fb     $000000    0          V       VoiceL[ ]
```

AORs allocated for computed addressing [ ] require null *** initialization.

-BUG: (v0.51)
warning: *’gfgf’:* never used

```
        DEFREGION R
              Xoutput[256]
             gfgf = &Xoutput[3]
        CODE
        MOV  #3 > &Xoutput[3]
```

Yet *gfgf* appears in **.**agn .

-BUG: (v0.51)
... RD *(INDIRDEC)R > DILD
Appears as RD INDIRDEC > DILD in **.**agn .

-BUG: (v0.51) Item #7 in **SPR Hazards**, HOST_CNTL_SPR ok as MAC unit destination.

-BUG: (v0.51) from revs.txt :
’***Version 0.45 [24 March 1995]***
6 removed global-SPR references from .hdr file (could have clashed with SPR defines in downloader) [syms.c]’
The above item requires undoing. Proper solution is for particular product/programmer convention which abides by rule not to type GLOBAL within DEFSPR.

-BUG: (v0.51)
MAC error: ’ONE’: illegal type in expression
MOV #ONE > MACP
The value of a register name is its register address.

-BUG: (v0.52) Following should issue warning for never-used GPRs when there is no appearance of some_gprs in code.

```
DEFGPR
       some_gprs          = 1  @$2
      (some_gprs, 1)      = 7
```

Assembler ERRORS and WARNING messages requiring implementation or revision:

1.) RD *(BASE += (aor)P)R > DIL0

This syntax is confusing because it implies that the BASE from one region is updated while the memory access is from a different region.

2.) Negative delayline index now yields error. Calculation should be same as for positive index, no error. If address offset goes out of region modulus, then error.

3.) PROGSIZE <= 1025

Should produce error: *in excess of physical limit* (1024).

9.) Currently there is no mention of quoted destination *delayspec* in **ESP2 Scheduler Notes** or Soft Spec. as an error condition, but assembly error is generated. There should be no error.

10.) Warning for SER SPR input or output on queued line following BIOZ (or BIOZNORFSH) instruction. Warning states: 'serial access may not be current'. Preventing output there avoids the Haas effect. The 'following queued line' is not necessarily contiguous.

11.) Quoting from the section on AGEN:

'*AOR Allocation*

*In the code, any reference to one from the delayspec set: d[n], &d[n], or *&d[n], will cause allocation of the same AOR, allocation occurring only once. Once allocated, reference to the set calls out an existing AOR. But if an AOR assignment of identical initialized contents, in the same region, pre-exists in the declarations, then no new allocation will occur.***'**

When in the code the assembler substitutes a pre-existing AOR from the declarations, a warning should be issued notifying the programmer. This is because programmer initialized AORs are sometimes used for purposes other than addressing. This booby-trap happens often in the case of AORs initialized to 0.

12.) Warning for no refresh of internal registers. This warning would be caused by the complete lack of ALU NOP, BIOZ, or HOST instructions or MAC unit RFSH pseudo instructions. Any one of these instructions executed with a minimum frequency of approximately 5000 Hz is sufficient to refresh all internal registers. (Must <u>not</u> include BIOZNORFSH or HOSTNORFSH in criterion. Conditionally executed instructions should be considered as providing <u>no</u> refresh.)

13.) assembler v0.51, Hardware Spec, section 4.1, second ASSEMBLER NOTE regarding the '***** *cases of Jcc...*'**.** New Warning for improper use of REF SPR, as noted.

13a.) assembler v0.51, amendment to first ASSEMBLER NOTE; '*The assembler should detect the use of the* ***REF****, INDIRECT, INDIRINC, and INDIRDEC...*', Hardware Spec, section 4.1, adds REF to list of keywords. Only instruction impacted is REPT.

14.) Hardware spec., SPR Hazards section, number 9.

<u>Required Assembler Revision</u>

- Want   #&d[n]   to allocate only GPR in code. Reference is to pre-existing AOR address. Want no error upon forward reference in code.

- Want pointer arithmetic within DEFREGION using & **.**   Presently can do math but <u>must</u> use # within DEFREGION.

- **q**N; for N made to be a constant <u>expression</u> (i.e., composed of symbolic constants and/or numbers). **(**Recommend elimination of capital **q** for binary-radix notation; interference with region identifier of same name.**)**

- In **.**lst put & in front of references to Preset AORs where appropriate. Alternately, change the heading 'Reference' to 'Dereference'.

- The relocation bit-map for the relocating downloader in the listing (**.**lst, v0.52) is confusing because it appears that the information is part of the instruction word. This might be remedied by <u>not</u> supplying a header mnemonic, as is presently the case for line numbers appearing at the far left; this is how the listing currently appears:

```
         |   |   |   |  | |  |   |   |   |  |  |   |  | |.--- L:  AGEN DATA LATCH
         |   |   |   |  | |  |   |   |   |  |  |   |  | | .- s:  AGEN SKIP BIT
         |   |   |   |  | |  |   |   |   |  |  |   |  | | |  B:  RELOC BITMAP
         |   |   |   |  | |  |   |   |   |  |  |   |  | | |  |
         D   E   F   OP S s  A   B   C   OP s  G   OP R L s  B
000  72  1d4 1cc 3ff 02 6 0  3ff 3ff 3ff 19 0  2f5 0 7 0 0  01
```

This is the recommended revision:

```
         |   |   |   |  | |  |   |   |   |  |  |   |  | |.--- L:  AGEN DATA LATCH
         |   |   |   |  | |  |   |   |   |  |  |   |  | | .- s:  AGEN SKIP BIT
         |   |   |   |  | |  |   |   |   |  |  |   |  | | |  .   RELOC BITMAP
         |   |   |   |  | |  |   |   |   |  |  |   |  | | |  |
         D   E   F   OP S s  A   B   C   OP s  G   OP R L s
000  72  1d4 1cc 3ff 02 6 0  3ff 3ff 3ff 19 0  2f5 0 7 0 0  01
```

ESP2 ASSEMBLER, REQUIRED ENHANCEMENTS

- AGEN listing should retain fully-commented lines in both declarations and CODE to enhance readability, and for reuse as source. See Linear Interpolation Application AGEN listing, for example.

- Explicit declaration of available physical external memory locations for bounds checking.

- Turn off specific warnings, in declarations.

- If E operand is AOR, while D operand is not, swap the two operands with notification, but not if indirection is involved.

- New switch in command line: -r
  This places chip revision number in microcode (**.**o, **.**bin) and **.**asm. Update assembler invocation.
- Clean up switch list as in section, Assembler Invocation.

- Allow forward reference.

- In DEFREGION, allow declaration of overlapping delaylines with warning.

- Allow declaration of overlapping regions with warning.

- Allow declaration of overlapping register arrays with warning.

- Language simplification of register INDIRECTion.

- Interactive loop analyzer for estimation of program run-time w/r sample period.

- Initialize internal registers from declarations using filename.

- Initialize arrays of internal registers without allocation. In this circumstance there should be no warning for registers never used.

- Make binary-radix notation employ only lower case **q**, while region reference employs only upper case Q (as region reference presently does, v0.52).

Appendix IX

MultiRate Audio Processes

Ref.: Interpolation Applications[168]

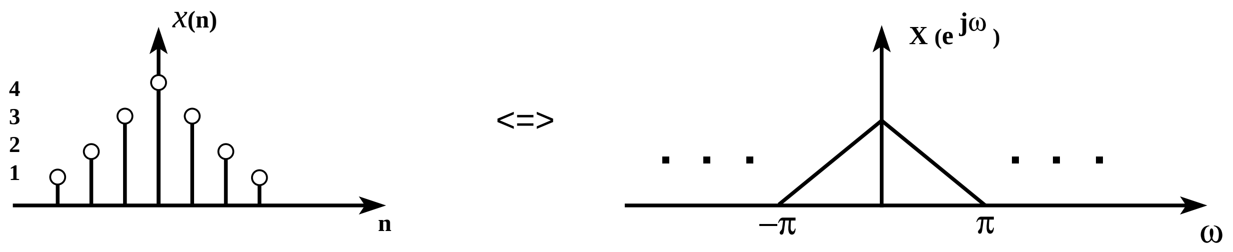
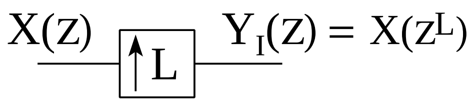
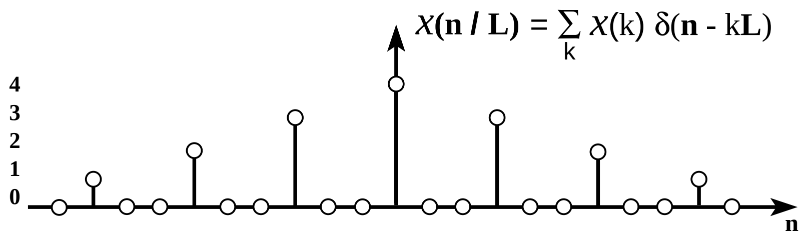


Figure ISS. Time-limited input signal and fictitious real spectrum.

(a)



(b)



<=>

(c)

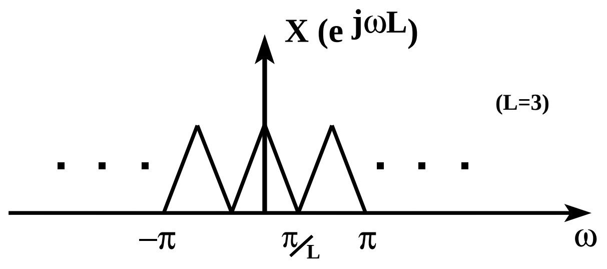


Figure UV. Upsampler, Vaidyanathan.

[168]Thanks go to Scott Levine for his careful review of this Appendix.

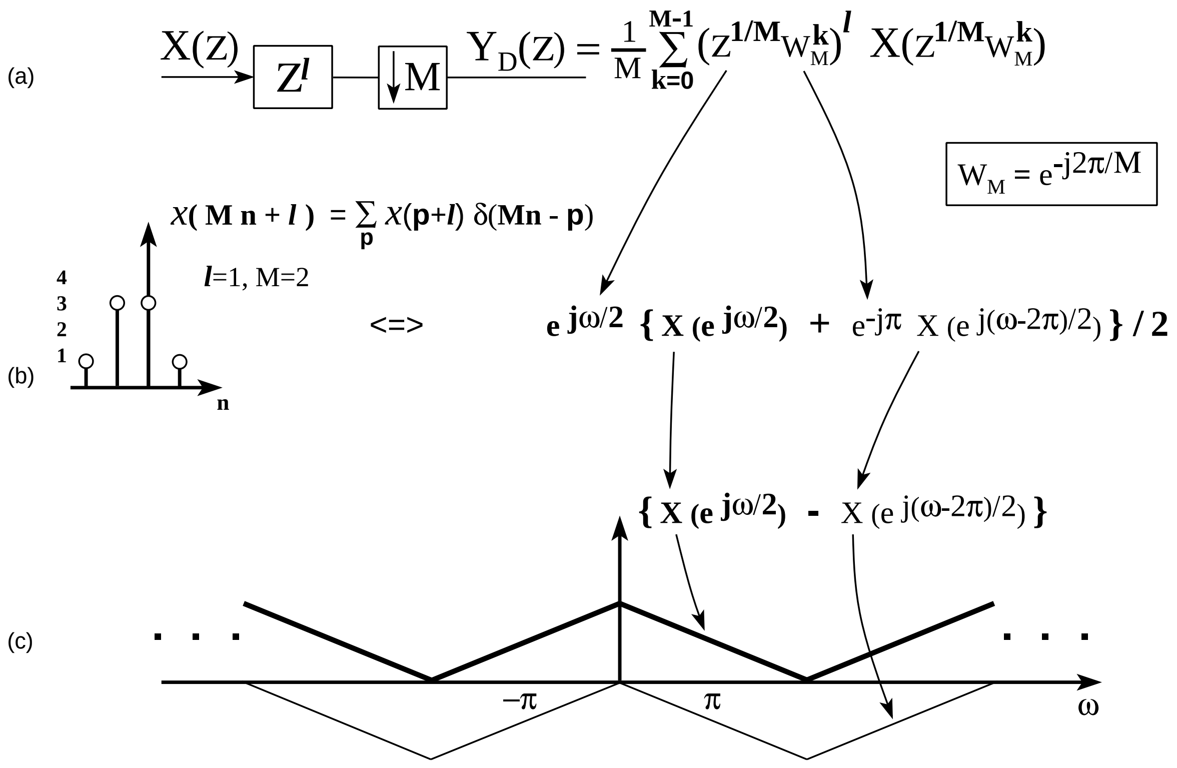

(a)
X(Z)
Z^l
$\downarrow$M
$Y_D(Z) = \frac{1}{M}\sum_{k=0}^{M-1}(Z^{1/M}W_M^k)^l\ X(Z^{1/M}W_M^k)$
$W_M = e^{-j2\pi/M}$
$x(Mn+l) = \sum_p x(p+l)\,\delta(Mn-p)$
l=1, M=2
4
3
2
1
n
(b)
<=>
$e^{j\omega/2}\ \{X(e^{j\omega/2}) + e^{-j\pi}X(e^{j(\omega-2\pi)/2})\}/2$
$\{X(e^{j\omega/2}) - X(e^{j(\omega-2\pi)/2})\}$
(c)
$-\pi$
π
ω

Figure DSB. Generalized downsampler, Vaidyanathan

We call the downsampler in Figure DSB (a) *generalized* because of the preceding advance operator. [Vaidyanathan,pg.122] The integer advance *l* finds use when either positive or negative.

The frequency domain decimation equation in Figure DSB (a) is a true Z transform that is derived from the time domain expression for decimation in (b), once we make the substitution:

$$\sum_{p=-\infty}^{\infty}\delta(\mathbf{M}\mathbf{n}-\mathbf{p}) = \frac{1}{\mathbf{M}}\sum_{k=0}^{\mathbf{M}-1}W_M^{-k\mathbf{p}} \qquad \text{(WID)}$$

for $-\infty < n < \infty$. $\delta(n)$ is the Kronecker delta function. The identity (WID) is the bridge between the two domains.

We now derive the generalized decimation equation for l an arbitrary integer advance, and for M the rate of decimation: Taking the Z transform,

$$Y_D(z) = \sum_{n=-\infty}^{\infty} x(Mn + l)\, z^{-n}$$

First we substitute the equivalent form of the signal x(Mn+l) whose time index remains n. Then we apply (WID).

$$Y_D(z) = \sum_{n=-\infty}^{\infty} \left\{ \sum_{p=-\infty}^{\infty} x(p + l)\, \delta(Mn - p) \right\} z^{-n}$$

$$= \sum_{\frac{p}{M}=-\infty}^{\infty} \left\{ x(p + l)\, \frac{1}{M} \sum_{k=0}^{M-1} W_M^{-kp} \right\} z^{-\frac{p}{M}}$$

$$= \frac{1}{M} \sum_{k} \sum_{p=-\infty}^{\infty} x(p + l) z^{-\frac{p}{M}} W_M^{-kp}$$

Since the sum over k only has value when p=Mn by (WID), then there is no complex root of z taken; hence that exponentiation is unambiguous. Now we let p+l -> r .

$$= \frac{1}{M} \sum_{k} \sum_{r=-\infty}^{\infty} x(r) z^{-\frac{(r-l)}{M}} W_M^{-k(r-l)}$$

$$Y_D(z) = \frac{1}{M} \sum_{k=0}^{M-1} \left(z^{\frac{1}{M}} W_M^{k} \right)^{l}\; X\left(z^{\frac{1}{M}} W_M^{k} \right) \qquad (\text{deci})$$

Equation (deci) is the desired result expressed in terms of the Z transform, X(z).

The two equivalent figures, Figure CMPR and Figure CMPI, tie together the formal analysis in terms of sample rate conversion ratio, [Crochiere/Rabiner,pg.40,pg.81] [Vaidyanathan,pg.131] and our interpretation of the interpolation process[169] in terms of the required fractional sample delay. H(z) in Figure CMPR represents the formal prototype interpolation filter. When the interpolation process is discussed as in the Applications in terms of the required fractional delay, the upsampling rate L is implicit in the sheer resolution of the polyphase filter coefficients. Hence the implied value of L is 2^{23} because we require no less than a 24-bit processor for audio.

[169]We often refer to the complete process depicted in Figure CMPR and Figure CMPI as 'interpolation'. That is an abuse of terminology, strictly speaking, as both the process of interpolation <u>and</u> decimation are carried out in those figures, having H(z) simultaneously serving both purposes. In Figure CMPR, there is an upsampler that serves to insert L-1 zeroes between every incoming sample, and there is a downsampler that discards M-1 out of every M samples. Upsampling and downsampling combined with the appropriate digital filter respectively describe the process of interpolation and decimation. The role of the upsampler is played by the commutator in Figure CMPI.

Each polyphase filter, represented by $E_l(z)$ in Figure CMPI, ideally presents a different fractional sample delay to a signal. More accurately, for the idealized formulation of interpolation by a rational factor, each polyphase filter is exactly allpass;

$$E_l(e^{j\omega}) = e^{j(\omega - 2\pi\, m[((\omega+\pi))_{2\pi}])l/L} \qquad ; m[\cdot] = 0 \to L-1 \qquad \text{(polyphi)}$$

where l is the polyphase filter number, L is the upsampling rate, the integer $m[\cdot]$ is the prototype replication number or the frequency-band number of an Lth-band (Nyquist(L)) prototype. The phase is a disjunct function of $m[\cdot]$ but linear in the baseband ($m[\cdot]=0$) of all the ideal polyphase filters. [Vaidyanathan,pg.168,109,124] [Crochiere/Rabiner,ch.4.2.2] Hence the l^{th} polyphase filter represents an advance of l/L fractional samples in the baseband.

The proper interpretation of this idealized polyphase filter (polyphi) requires m to be a modulo function of ω. For every modulo 2π of ω, $m[\cdot]$ increments within bounds; i.e.,

$$\begin{aligned} m[((\omega+\pi))_{2\pi}] &\equiv \left(\left(\frac{\omega+\pi - ((\omega+\pi))_{2\pi}}{2\pi}\right)\right)_L = \left(\left(\left\lfloor\frac{\omega+\pi}{2\pi}\right\rfloor\right)\right)_L \qquad ; \ \omega+\pi > 0 \\ &\equiv L-1-\left(\left(\frac{|\omega+\pi| - ((|\omega+\pi|))_{2\pi}}{2\pi}\right)\right)_L = L-1-\left(\left(\left\lfloor\frac{|\omega+\pi|}{2\pi}\right\rfloor\right)\right)_L \qquad ; \ \omega+\pi < 0 \end{aligned}$$

where the double parentheses denotes the modulo operator; the real modulus 2π and the integer modulus L .

This idealization presumes that the prototype interpolation filter is perfectly bandlimited. To construct the prototype from these ideal polyphase filters (polyphi) we use (polyh).

$$H(z) = \sum_{l=0}^{L-1} z^{-l} E_l(z^L) \qquad \text{(polyh)}$$

Now, for every modulo 2π/L of ω, $m[\cdot]$ increments. So we have

$$\begin{aligned} H(e^{j\omega}) &= \sum_{l=0}^{L-1} e^{-j\omega l} e^{j(\omega L - 2\pi\, m[((\omega L+\pi))_{2\pi}])\frac{l}{L}} \\ &= \sum_{l=0}^{L-1} e^{-j2\pi\, m[((\omega L+\pi))_{2\pi}]\frac{l}{L}} \end{aligned}$$

From (WID), this only has value for $m[\cdot]=0$. Hence,

$$\begin{aligned} H(e^{j\omega}) &= L \qquad ; \ |\omega| < \frac{\pi}{L} \\ &= 0 \qquad ; \ \frac{\pi}{L} < |\omega| < \pi \end{aligned}$$

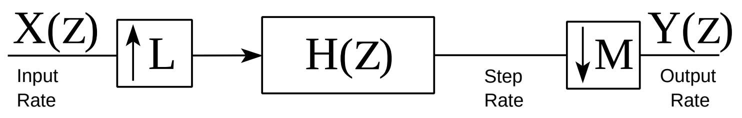


Figure CMPR. General model of interpolation by a rational factor.

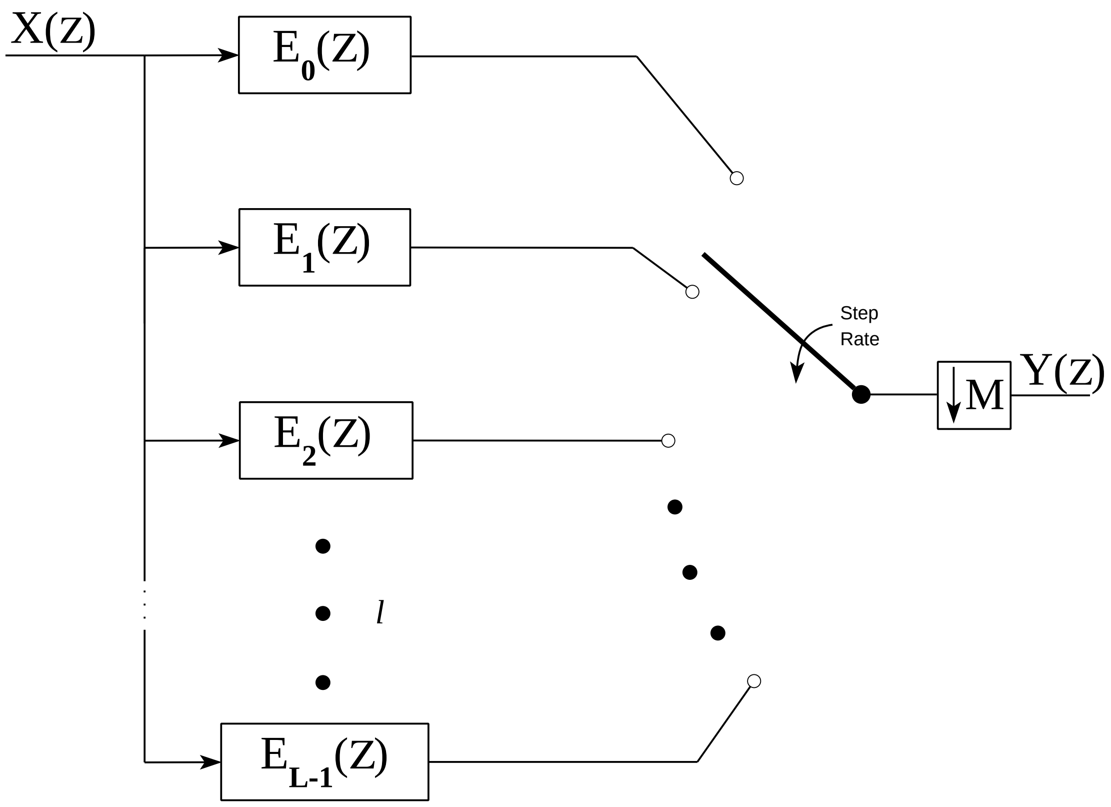


Figure CMPI. Ideal commutator model of interpolation showing the many polyphase filters.

The transfer function of the commutator filter circuit in Figure CMPI (not including the downsampler) can be described using a generalization of (polyh):

$$H(z) = \sum_{l=0}^{L-1} z^{-(l - l_o)}\ E_l(z^L) \qquad \text{(polyhc)}$$

where l_o is the starting phase of the commutator; i.e., the initial position of the brush when the input sample x[n] arrives. The action of the downsampler[170] is synchronized to l_o such that the very first sample found at position l_o is passed. Reversing the brush direction would simply change one sign of the z exponent in (polyhc); i.e., to $z^{(l-l_o)}$.

The individual $E_l(z)$ in Figure CMPI are time-**in**variant and correspond to one of the circuits shown in the Applications in Figure PL2, and Figure PA or Figure WPA (both having the delay element connected), for the respective cases of Linear[171] and ideal Allpass interpolation. Recalling the nomenclature we previously devised for those circuits in (lipdf), we fix the coefficients of the lth filter by making the identification:
$\tau = \text{frac} = l/L$ where τ is the desired fractional sample delay, frac is the polyphase filter coefficient, l is the polyphase filter number, and L is the upsampling rate.[172]

By design, $E_l(z)$ is a first-order two-tap non-causal FIR filter in the case of Linear interpolation;

$$\left.\begin{aligned} P_l(z) &= (1 - l/L) + (l/L)\, z^{-1} \\ E_l(z) &= z\, P_{L-l}(z) \end{aligned}\right\} \text{;Linear interpolation}$$

while in the case of ideal Allpass interpolation it is a first-order non-causal allpass filter;

$$\left.\begin{aligned} P_l(z) &= \frac{(1 - l/L) + z^{-1}}{1 + (1 - l/L)\, z^{-1}} \\ E_l(z) &= z\, P_{L-l}(z) \end{aligned}\right\} \text{;ideal Allpass interpolation}$$

[170]Taking the view of Crochiere, for the moment, the downsampler would operate at the step rate, L F_s .

[171]Because the many polyphase filters of Linear interpolation require no long-term memory, we were able to cheat in the actual implementation by using only the one circuit shown in Figure PL instead of the entirety of Figure CMPI. Hence, we successfully implemented *ideal* Linear interpolation. (We drop the term 'ideal' in the discussion.)

[172]The number of polyphase filters, L, in those applications was determined by the numerical resolution of the registers holding the polyphase filter coefficients.
Also note that because of the method of implementation of all the applications that we discuss, there is no division required in the capture of the coefficient, l/L.

$$\left.\begin{aligned} P_{l}(z) &= \frac{\dfrac{(1-l/L)}{(1+l/L)} + z^{-1}}{1 + \dfrac{(1-l/L)}{(1+l/L)}\, z^{-1}} \\ E_{l}(z) &= z\, P_{L-l}(z) \end{aligned}\right\}$$

;ideal Allpass interpolation
with coefficient warping (sw.eq)

The three $P_l(z)$ are the respective transfer functions of the circuits in Figure PL2, Figure PA (having the delay element connected), and Figure WPA (having the delay element connected).

For the prototype interpolation filter H(z) of Linear interpolation, the length of its triangular impulse response is 2L as illustrated in Figure LinImpulse; that length, by design, spanning only two of the original input sample periods. While in the case of ideal Allpass interpolation, the impulse response is of infinite length as implied in Figure ALLImpulse. Both impulse responses have every Lth sample equal to zero (except the central sample) thus identifying Lth-band (Nyquist(L)) prototype filters. [Vaidyanathan,ch.4.6] [Renfors/Saramäki] Hence the original input samples, $x(n)$, appear unscathed at the input to the downsampler in Figure CMPI regardless of l_o. A second salient characteristic of Lth-band filters is that the frequency-shifted filter sums to level in the frequency domain like so;

$$\sum_{k=0}^{L-1} H\left(zW_L^k\right) = L \qquad \text{(lthb)}$$

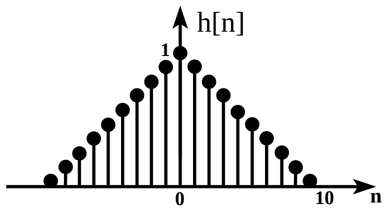


Figure LinImpulse. Impulse response of Linear interpolation prototype, L=10.

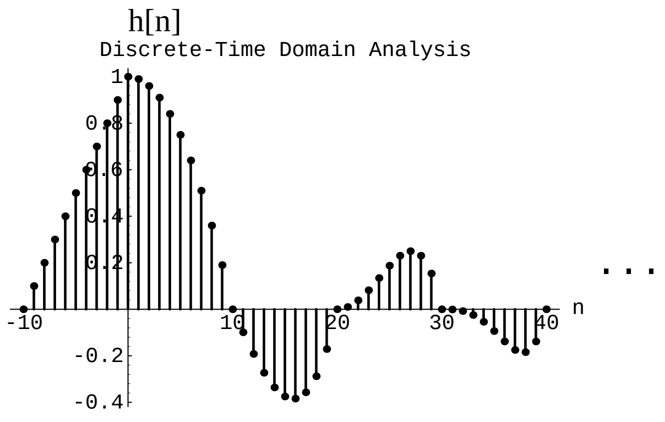


Figure ALLImpulse. Ideal Allpass prototype filter impulse response, L=10. [Evans]

No coefficient warping was used to make Figure ALLImpulse. The initial linear slope of the impulse response for the Allpass interpolation prototype in Figure ALLImpulse is identical to that of the Linear interpolation prototype. This can be explained best by observing Figure CMPI in the case that M=1. The commutator brush visits all the polyphase filters within the same time period of one input sample. If the input signal $x(n)=\delta(n)$ and $l_o=0$, then what we shall see at the brush is the impulse processed by each of the allpass polyphase filters. But on this first sweep, none of the allpass filters has anything stored in its memory, so the filters are effectively scalar multipliers increasing linearly with l .

Now we show those two prototype filters in the frequency domain:

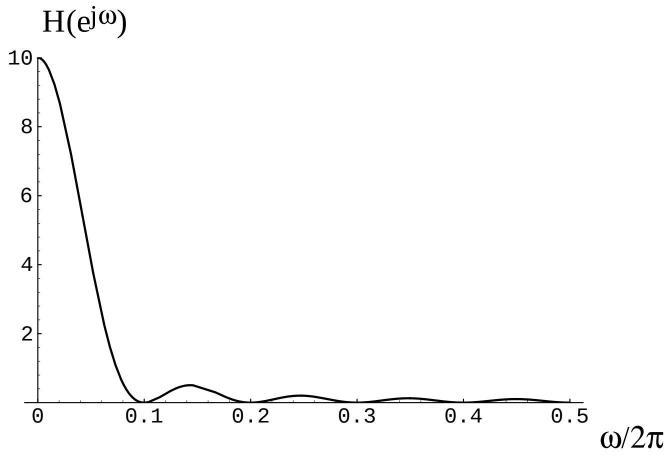


Figure LINPROTO. Linear interpolation prototype filter transfer, L=10.

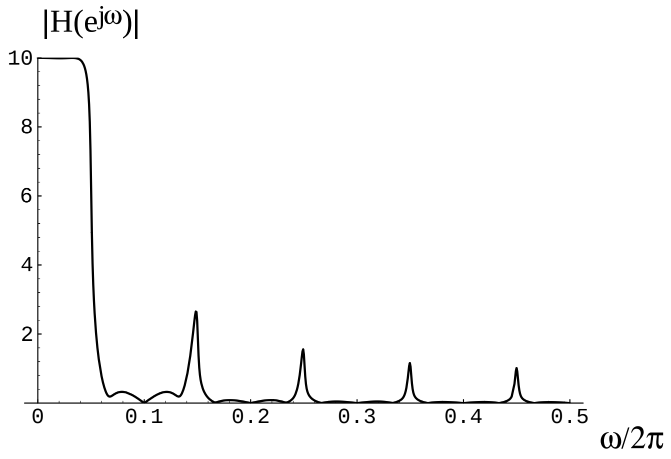


Figure ALLPROTO. Ideal Allpass interpolation prototype filter transfer, L=10.

Figure LINPROTO and Figure ALLPROTO show the prototype filter magnitude for Linear and Allpass interpolation (no coefficient warp), respectively, in the case of L=10. For Linear interpolation we expect a positive real function of the form,

$$H(e^{j\omega}) = \frac{\sin^2(\omega L / 2)}{L \sin^2(\omega / 2)} \qquad \text{(asinc)}$$

The expression for the Allpass interpolation prototype is not so simple; neither is its phase.

Of course, there is a mathematical description of any prototype interpolation filter in terms of its polyphase filters, and vice versa.

$$H(z) = \sum_{\boldsymbol{l}=0}^{\mathbf{L}-1} z^{-\boldsymbol{l}} E_{\boldsymbol{l}}\left(z^{\mathbf{L}}\right) \qquad ; \; \boldsymbol{l}_o = 0 \qquad (\text{polyh})$$

$$E_{\boldsymbol{l}}(z^{\mathbf{L}}) = \frac{1}{L} \sum_{\mathbf{k=0}}^{\mathbf{L-1}} (z\, W_L^{\mathbf{k}})^{\boldsymbol{l}}\; H(z\, W_L^{\mathbf{k}}) \qquad (\text{polye})$$

$$= \sum_{n=-\infty}^{\infty} h(L\,n + \boldsymbol{l})\; z^{-n\mathbf{L}} \qquad (\text{polyeh})$$

Without significant loss of the generality of (polyhc), equation (polyh) describes well the action of the commutator circuit presuming $l_o=0$. Equation (polye), a generalization of (lthb), is an application of the decimation equation (deci) to H(z) for advance l . Equation (polye) applies directly to the polyphase decomposition of either IIR or FIR prototypes. The time domain impulse response h(n) in (polyeh) of course corresponds to H(z).

Having established these definitions, (polye) and (polyh), we can write the equation for the commutator model of Figure CMPI in the manner of Vaidyanathan.

$$Y(z) = \big(X(z^{L})\; H(z)\big)\big\downarrow_M \qquad (\text{polyy})$$

$$= \frac{1}{M} \sum_{\boldsymbol{m}=0}^{\mathbf{M-1}} X(z^{\mathbf{L/M}}\, W_M^{\mathbf{L}\boldsymbol{m}}) \sum_{\boldsymbol{l}=0}^{\mathbf{L-1}} (z^{\mathbf{1/M}}\, W_M^{\boldsymbol{m}})^{-\boldsymbol{l}}\; E_{\boldsymbol{l}}(z^{\mathbf{L/M}}\, W_M^{\mathbf{L}\boldsymbol{m}})$$

This formidable looking equation reduces nicely for special cases of interest:
e.g., when M=L, $Y(z) = X(z)\, E_0(z)$. (Recall that $E_0(z)=1$.)
When M=L/2, $Y(z) = X(z^2)\,\big(E_0(z^2) + z^{-1} E_{L/2}(z^2)\big)$. In this case, the two polyphase filters are attempting to form the half-band filter required of an effective upsampling and interpolation by the factor 2=L/M (pretend that M=1).

The resulting expressions are not as simple when L/M is irreducible. The fascinating aspect of (polyy), however, is that it gives us a fixed closed-form representation for the linear time-varying circuit; that is Figure CMPI. The reader is encouraged to explore that further[173] and thereby get a better feel for the relationship to the commutator model in Figure CMPI.

[173]Keep in mind the identity (WID).

Engineering Approximation to Ideal Allpass Interpolation

Rather than implement the ideal formal circuit in Figure CMPI in the case of Allpass interpolation, our innovation (as proposed in the Applications) is to instead implement the formal circuit in Figure CMPIappx which is an approximation. The recursive elements have been taken out of the polyphase filters in Figure CMPIappx. Now,

$$\left.\begin{aligned} P_{\boldsymbol{l}}(z) &= (1 - l/L) + z^{-1} \\ E_{\boldsymbol{l}}(z) &= z\, P_{\mathbf{L}\text{-}\boldsymbol{l}}(z) \end{aligned}\right\} \text{;Allpass interpolation Application}$$

As we saw in the Applications, this approximation to the ideal formulation is only good,[174] in terms of ***THD+N***, for microtonal changes in pitch; M/L near 1.

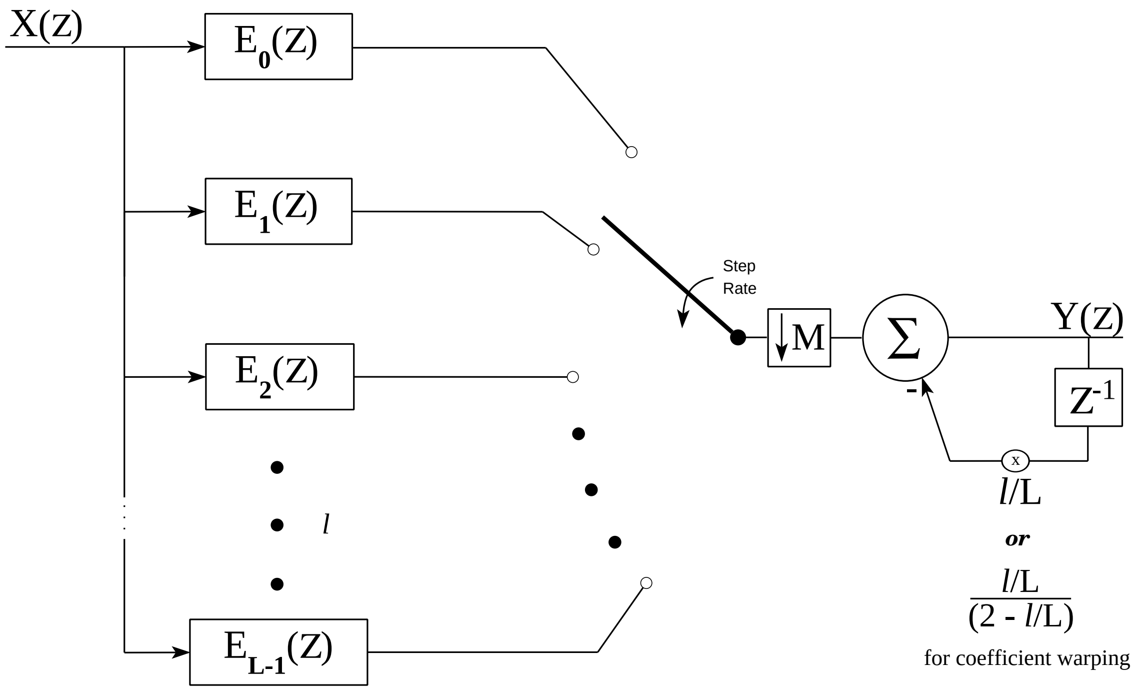


Figure CMPIappx. Commutator model of proposed Allpass interpolator.

[174]This circuit is in production in many successful electronic musical products.

Figure CMPIappx is the equivalent to Figure PA as is; without the feedforward delay element connected. Each $E_l(z)$ in Figure CMPIappx is nonrecursive, having no long-term memory, and time-**in**variant. The idea of moving the recursive part of the polyphase network outside of the commutator is not new. [Crochiere/Rabiner,ch.3.4] proposed separating out the denominator like this, but their denominator was common to all the polyphase filters. Note that the lone time-varying recursive circuit on the right-hand side in Figure CMPIappx only approximates the the L different time-**in**variant recursive circuits that are supposed to be associated with each $E_l(z)$ as in Figure CMPI.

One further refinement is our warping of the polyphase filter coefficients, computed in real-time[175] and employed in the approximation (Figure CMPIappx) to the ideal network.

$$\left.\begin{aligned} P_{\boldsymbol{l}}(z) &= \frac{(1 - \ l/L)}{(1 + l/L)} \ + \ z^{-1} \\ E_{\boldsymbol{l}}(z) &= z\, P_{\mathbf{L}-\boldsymbol{l}}(z) \end{aligned}\right\} \text{;Allpass interpolation Application with coefficient warping (sw.eq)}$$

With warping, Figure CMPIappx becomes equivalent to Figure WPA as is. Using this refinement, we measured an improvement of 26 dB in ***THD+N*** for microtonal pitch changes. (That was discussed in the Applications.)

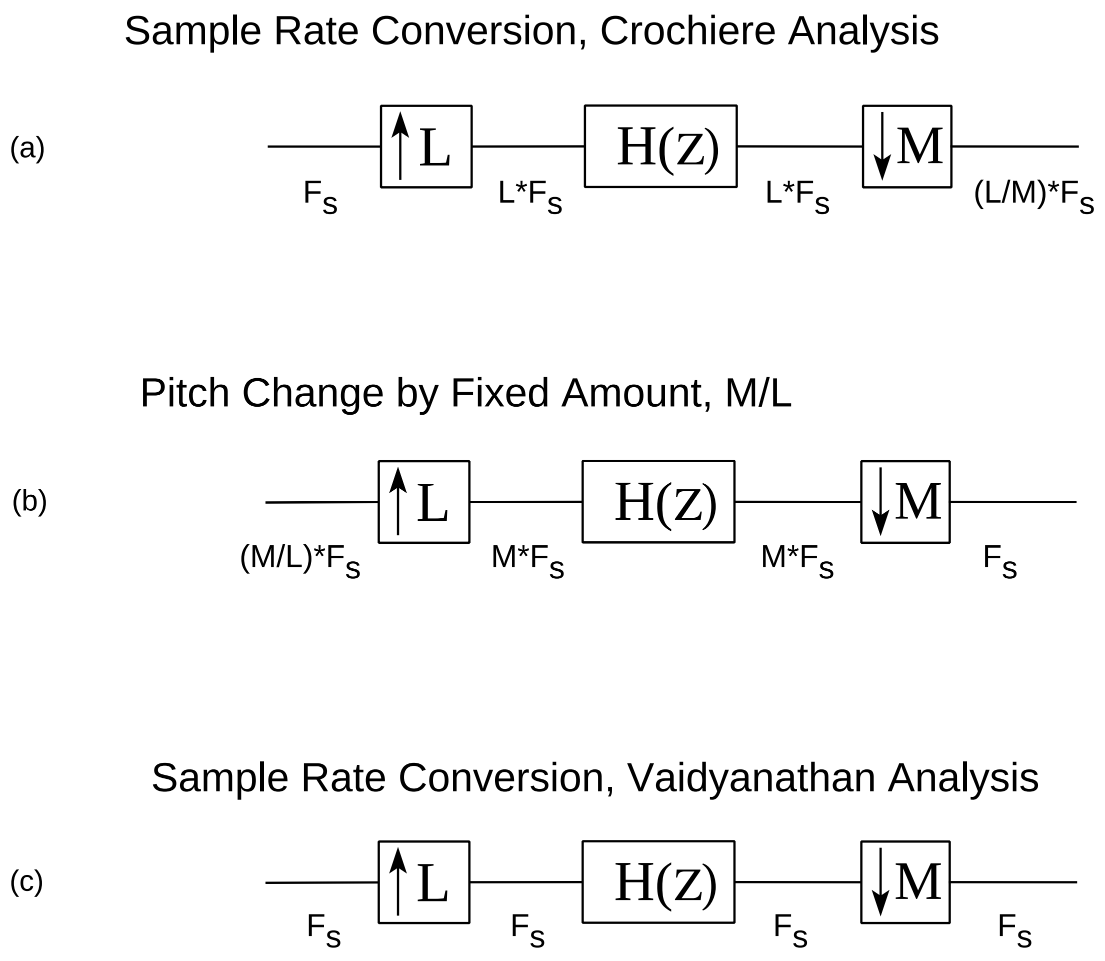
Distortion is inherent to any interpolation process. For large pitch change with little distortion, one's choice (in our context) is to revert to Linear interpolation at a high sampling rate or to implement the network in Figure CMPI for ideal Allpass interpolation. We have simulated Figure CMPI in the C programming language using 16-bit polyphase filter coefficients. To achieve excellent results over a large range of M/L for ideal Allpass interpolation, we find that coefficient warping (sw.eq) is necessary, and that about $L=2^8$ recursive states must be stored. Since there are no time-varying coefficients in the ideal circuit of Figure CMPI, associated transient phenomena are a non-existent component of the signal distortion characteristics. The only transients that arise there are due to the ZSR (Zero State Response) of the recursive polyphase filters of Allpass interpolation. In Pitch Change/Shift applications of the ideal network, no one polyphase filter is accessed more than another, in general. The ZSR transients will be short for polyphase filters having small feedback coefficients, and longer for larger coefficients. So we expect ZSR transients to be evenly distributed except in degenerate cases.

Conversion: Vaidyanathan -> Crochiere
The Vaidyanathan method of analysis is simpler because it ignores changes in sample rate, whereas the method of Crochiere does not. But note that in terms of the insertion or deletion of samples, the action of the upsampler block **(**as shown in Figure UV (a) and (b)**)** and the downsampler block **(**as in Figure DSB (a) and (b)**)** is respectively the same for both the Crochiere and Vaidyanathan methods of analysis. While the two methods of analysis are equally valuable, it is prudent to have a means of converting between the two. Given the Vaidyanathan analysis, one simply substitutes every occurrence of z with z' (Crochiere's notation) which is defined equal to $z^{\mathbf{M/L}}$, and one would re-label any frequency domain graphs substituting ω' (which equals $\omega\mathbf{M/L}$) in place of ω.

[175]This step is optional. Review (sw.eq) in the Applications section.

The conversion becomes exceedingly simple when either L or M is 1. For example, we wish to convert Figure UV (M=1, L=3) to the Crochiere-style upsampler analysis. Then we must re-label the abscissa using ω' in place of ω, and we re-label the ordinate as $X(e^{j\omega' L}) = X(e^{j\omega})$ in place of $X(e^{j\omega L})$. This result is correct because Crochiere always maintains the time period between the original samples; that is true for either upsampling or downsampling.

Polyphase Perspectives



Sample Rate Conversion, Crochiere Analysis
(a)
↑L
H(Z)
↓M
F_S
L*F_S
L*F_S
(L/M)*F_S
Pitch Change by Fixed Amount, M/L
(b)
↑L
H(Z)
↓M
(M/L)*F_S
M*F_S
M*F_S
F_S
Sample Rate Conversion, Vaidyanathan Analysis
(c)
↑L
H(Z)
↓M
F_S
F_S
F_S
F_S

Figure CU. Contrasting the presumed computation rates of the various methods of analysis.

The sample rate conversion ratio, L/M, corresponds to the Pitch Change/Shift ratio inverse. We presume that the Pitch Change process in Figure CU (b) is performed on a stored sample record. Samples appear at the input at a rate different from which they were recorded. Such would be the case in contemporary sampler-type synthesizers.

Application Schema

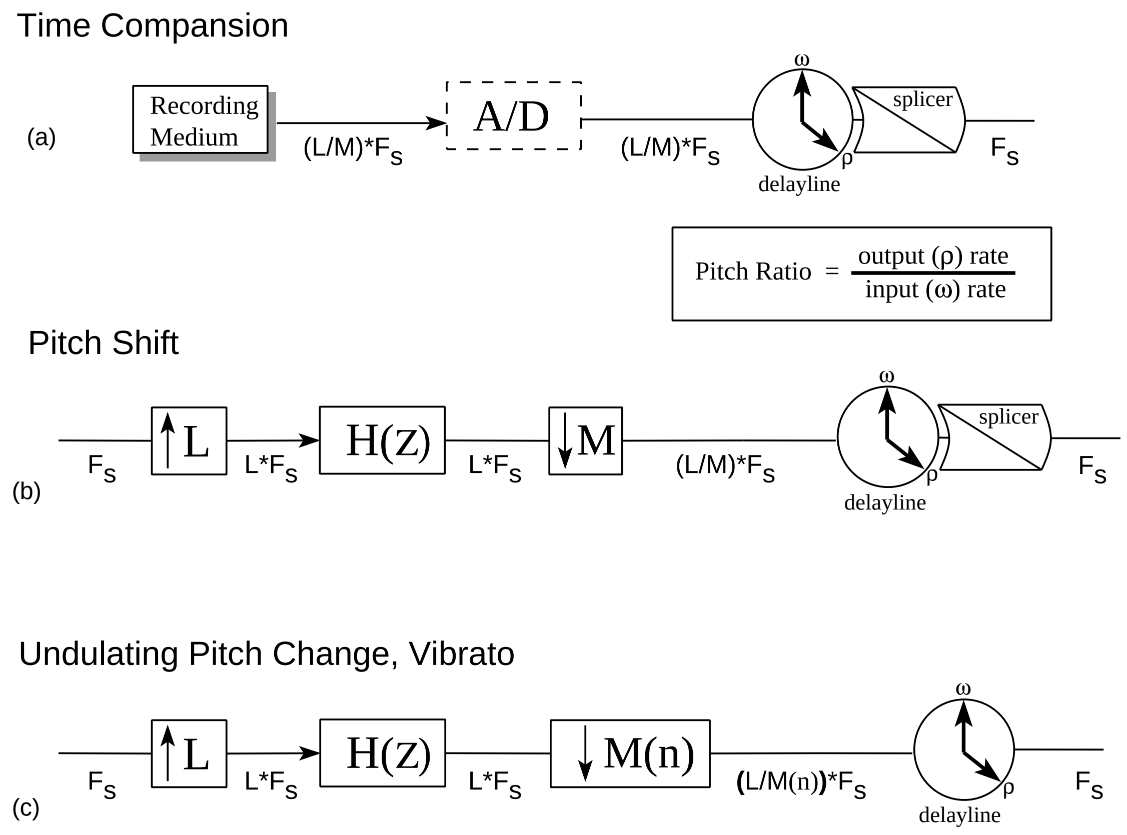


Figure DTI. Applications of Discrete Time Interpolators.

Time *Companion* (compression/expansion) was accomplished before the days of DSP by physically splicing magnetic recording-tape to alter the run-time without the concomitant shift in perceived pitch. [Lee] Contemporary Companion machines [Dattorro2400] actually change the playback speed of the recording medium thereby performing the manually tedious algorithm deftly in real time. The splicer in Figure DTI controls equestrian jumps by ρ-pointer to compensate for the disparity in sample rate between the input and output of the delayline.[176]

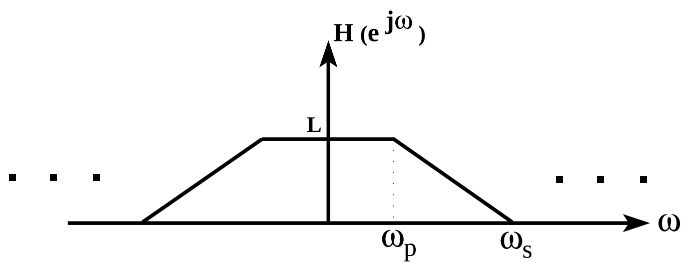
The converse operation, Pitch Shift, alters the perceived pitch with no change to run-time. In Figure DTI, both the Pitch Shift and Vibrato processes are performed in real time, and unlike fixed Pitch Change shown in Figure CU, both maintain the macro-temporal features of the original signal via propitious application of a delayline. The splicer is not required for Vibrato since the mean sample rate across the delayline is F_S by design; i.e., the downsampler M is appropriately time-varying.

All three processes shown require some significant amount of nominal (average) transport delay: To perform well upon polyphonic material, the Time Companion and Pitch Shift

[176]Each jump target is determined by a very high speed autocorrelator seeking periodicity within the delayline contents.

algorithms require as much as 60 ms. That much delay is easily perceptible and a compromise is nearly always necessary. Vibrato, on the other hand, can be performed well using only about 1 ms nominal delay.

Prototype Filter Design



$$\omega_p \leq \min\{\pi/M,\ \sigma/L,\ \pi/L\}$$

$$\omega_s = \begin{cases} \omega_p & ;\pi/M < \sigma/L \\ \omega_p + 2(\pi-\sigma)/L & ;\text{otherwise} \end{cases}$$

Figure HPR. Prototype interpolation filter specs for signal bandwidth σ.

Figure HPR shows the constraints on the design of the prototype filter, in general, when absolutely no aliasing of the signal, having (one-sided) bandwidth σ, is tolerated.[177] Implicit from the axis labeling is that the desired frequency domain filter is real.

The techniques of interpolation that we considered in the Applications section did not allow this level of control over the design of the prototype. We simply accepted what the implementation offered because it was computationally attractive.

[177]Note that by the filter specification in Figure HPR, there can be a discontinuity in the progression of the allowed transition-band slope as soon as σ/L creeps just past π/M. That happens because the role of the prototype switches abruptly, now having the added responsibility of further bandlimiting the signal. Admitting some small amount of aliasing mitigates that radical change in requirement. If $\sigma=\pi$, however, then the desired filter is always square.

Appendix X

Oscillator Equation Derivation

Ref.: Oscillator Applications

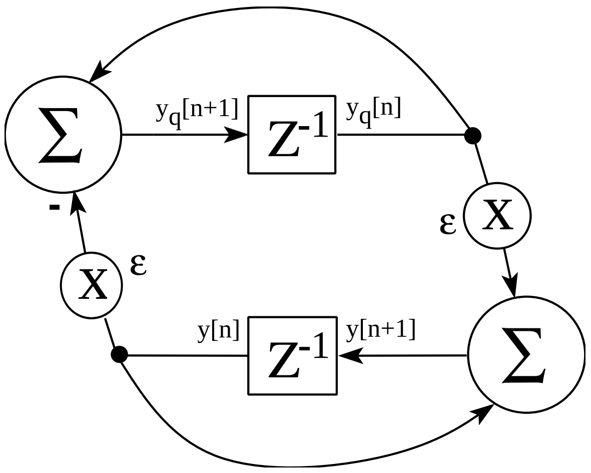


Figure X (b). The first modified coupled form sinusoidal oscillator.

We demonstrate the derivation of the oscillator equations for one case only: the first modified coupled form oscillator. This analysis is adapted from [Gordon/Smith] where the derivation of the coupled form and the second modified coupled form is shown.

$$\begin{cases} y_q[n+1] = y_q[n] - \varepsilon\, y[n] \\ y[n+1] = \varepsilon\, y_q[n] + y[n] \end{cases}$$

These state equations describe the ZIR of the circuit in Figure X (b). From them we read off the matrix,

$$\mathbf{G} = \begin{pmatrix} 1 & -\varepsilon \\ \varepsilon & 1 \end{pmatrix}$$

If we define the vector

$$\mathbf{y}_n = \begin{pmatrix} y_q[n] \\ y[n] \end{pmatrix}$$

then we may write $\mathbf{y}_{n+1} = \mathbf{G}\ \mathbf{y}_n$. This State-Variable description has the solution,

$$\mathbf{y}_n = \mathbf{G}^n\ \mathbf{y}_o$$

for all time $n \geq 0$, where $\mathbf{G}^n$ is called the *state transition matrix*, and where $\mathbf{y}_o$ is the vector of initial states; i.e., $\mathbf{y}_n$ at $n=0$.

If we can find the eigenvectors and eigenvalues of **G**, then we may write equivalently

$$\boxed{\mathbf{y}_n = \mathbf{T}\, \Lambda^n\, \mathbf{T}^{-1}\, \mathbf{y}_o}$$

where **T** holds the conjugate eigenvectors (given by *Mathematica*) in its columns, and Λ holds the corresponding eigenvalues along its diagonal. Specifically,

$$\Lambda = \begin{pmatrix} 1 + j\,\varepsilon & 0 \\ 0 & 1 - j\,\varepsilon \end{pmatrix} \triangleq \begin{pmatrix} \lambda & 0 \\ 0 & \lambda^* \end{pmatrix}$$

$$\mathbf{T} = \begin{pmatrix} j & -j \\ 1 & 1 \end{pmatrix}$$

$$\mathbf{T}^{-1} = \frac{1}{2}\begin{pmatrix} -j & 1 \\ j & 1 \end{pmatrix}$$

By constructing the diagonal matrix Λ, the exponentiation no longer requires matrix multiplication. Hence it is easier to acquire a solution analytically for large n.
The eigenvalues are the poles of the system under study. λ^* denotes the conjugate eigenvalue. If we define each eigenvalue in polar form

$$\lambda \triangleq |\lambda|\; e^{\,j\omega}$$

then we can make the identifications from Λ

$$\Rightarrow \begin{cases} 1 = |\lambda| \cos(\omega) \\ \varepsilon = |\lambda| \sin(\omega) \end{cases}$$

From these we conclude

$$\therefore \quad \begin{aligned} \varepsilon &= \sin(\omega)/\cos(\omega) \\ |\lambda| &= 1/\cos(\omega) \qquad ;\omega < \pi/2 \end{aligned}$$

where the actual oscillator coefficient ε is expressed in terms of the desired frequency of oscillation, ω. It follows that

$$\mathbf{T}\;\Lambda^n\;\mathbf{T}^{-1} = \frac{1}{\cos^n(\omega)} \begin{pmatrix} \cos(n\,\omega) & -\sin(n\,\omega) \\ \sin(n\,\omega) & \cos(n\,\omega) \end{pmatrix}$$

Appendix XI

ESP2 Program for Linear Interpolation

Ref.: Interpolation Applications

```
! Linear interpolation of delayline: JonD, ESP2, 11/19/94.

! VoiceL[2048] (signed, 24-bit q0, left justified) is the delayline we will tap.
! The tap point will undergo modulation.

! chorus_width (unsigned GPR, q0) is the peak excursion (in samples) of the delayline
! modulation about the tap point, nominal_delay.

! nominal_delay (unsigned AOR, q0) is the nominal whole sample delay into the delayline;
! i.e., the center tap point.

! y_qn (signed GPR, q23 (all fractional)) is the full-scale oscillator modulator output.
PROGRAM  Linear

DEFCONST
   CLK = 40.e6
   Fs = 44100.
   Freq = 1.0                              ! Hz
   Pi = 4.0*ATAN(1.0)
   EPSILON = 2.0*SIN((Pi*Freq)/Fs)
   CHORUS_WIDTH = 350                      ! CHORUS_WIDTH < NOMINAL_DELAY.
   NOMINAL_DELAY = 400                     ! In samples.  Altered to suit application.

PROGSIZE  <= INT(CLK/(4.*Fs))              ! assuming one big loop

DEFREGION  V
   VoiceL[2048]
   minus_one = SIZEM1V                     !AOR declaration = BASE modulo decrement
   nominal_delay =  &VoiceL[NOMINAL_DELAY]
   m_aor                                   !see Style section

DEFSPR
   PC = LinInterp
   REPT_CNT = 0
   SER_CONF = $007fff
   HARD_CONF = $008400

DEFGPR
   GLOBAL
      chorus_width = CHORUS_WIDTH
      epsilon = EPSILON
   LOCAL
      y_qn = 0
      yn  = -COS((Pi*Freq)/Fs)
      frac=0   frac_u=0        xL          vibrato=0
```

```
CODE
!****************************************************************************
LinInterp: NOP                                          !executes prior to suspension.  Want no SER access here.

!************** serial I/O *************
MOV SER0L > xL                     MOV vibrato > SER7L                     !coming out of suspension

!********************* feed delayline input ***********************
MOV xL > VoiceL[0]

!********************* Linear interpolation **************************
MOV nominal_delay > MACP
MACP + yqn X chorus_width > &VoiceL[ ]                                  !address, integer part
        frac  X  *&VoiceL[(+)] > MAC        LSH  MACRL >>1  > frac      !fractional part (positive)
MAC + frac_u  X  *&VoiceL[ ] > vibrato      DIFF  frac > frac_u
!****************************************************************

!****** hyperstable near-quadrature oscillator (second modified coupled form) ******
NOP                               MOV yqn > MACP
NOP                               MOV yn > MACP
MACP - epsilon X yn  > yqn
MACP + epsilon X yqn > yn

!***************** sample sync/delayline memory shift ******************************
NOP                               JMP  LinInterp
NOP                               BIOZ                  UPDATE  BASEV += minus_one
!**************************** end program ************************************
```

Discussion of the Linear Interpolator

As written, the effect of this program is to produce Vibrato. For other effects, such as Chorus or Flanging, slight modifications are made as discussed in the Applications.

This prototypical usage of the JMP, BIOZ, and UPDATE instructions serves as a **paradigm** for <u>all</u> sample synchronous programs employing delaylines. The delayline memory shift via the UPDATE instruction was discussed in the section on UPDATE region BASE.

Latency

Code very similar to this Application was previously discussed in the section on Computed Addresses. There a partial AGEN listing was shown, illustrating the various latencies. The AGEN listing for this Application appears later on.

It is strongly recommended to place all SER data SPR access towards the beginning of a program in case a particular program main loop exceeds the sample period. The time margin allowed for a program main loop to momentarily exceed the sample period is a design feature of the BIOZ instruction; in that case there will be no BIOZ suspension on that loop. **(**BIOZ is a sample-rate synchronization instruction discussed at length in the Chip Spec.**)**

The BIOZ instruction[178] has an instruction cycle execution latency of 1. So, the first line of our Linear interpolation program is executed prior to suspension, if suspension is warranted.[179] BIOZ is always executed before the JMP actually takes place because JMP is also one of the latent instructions. The SER data SPR transfers are latched by a transit high of the serial interface pin signal called LRCLK. That signal is likely tied to the IOZ input pin. The IOZ pin transit high will usually take the program out of suspension. **It is for this reason that we do <u>not</u> place <u>any</u> SER data access** such as

MOV SER0L > xL

on the next queued program line following the BIOZ instruction (the first line of our program**)**, for then we would <u>not</u> access the most recent sample.

The first line of code appears to be wasted in our Application program. This is only because of the need to simplify the presentation. It is not too difficult to find something for the first program line to do which is not involved with incoming or outgoing SER audio-sample data.[180]

In the portion of the code commented, !*** Linear interpolation ***, the fractions **(**frac and **frac_u** = 1 - frac**)** and the address offset **(**&VoiceL[ ]**)** are all computed too late to be used in the current sample period. As long as they are all applied in the <u>same</u> sample period, the computations will be valid, but latent.

---

[178]Any concern regarding BIOZ as the last instruction in this program is allayed by the preceding JMP.

[179]If this program executes in an amount of time that is less than the sample period (the period of LRCLK), then there <u>will</u> be suspension due to BIOZ.

[180]The right channel SER0R is conspicuously missing from our program; there is no reason for this.

Style

Instead of the C-like notation, &VoiceL[ ], another programmer may have chosen to explicitly declare an AOR called *m_aor* in region V. This may be easier to conceptualize in some cases. The interpolation code segment above would then have appeared as follows:

```
!********************* Linear interpolation ******************************
MOV nominal_delay > MACP
MACP + y_q n X chorus_width > m_aor                              !address, integer part
        frac  X  *m_aor(+) > MAC            LSH  MACRL >>1  > frac     !fractional part (positive)
MAC + frac_u  X  *m_aor > vibrato           DIFF  frac > frac_u
!*******************************************************************
```

Listing (.lst) of ESP2 Linear Interpolation Program

```
ESP2 Assembler Version 0.40 [29 November 1994]
Program listing: 01/14/95 18:10:58
linear.e2: source lines: 75  microinstructions: 13
no errors, 1 warning

       1  ! Linear interpolation of delayline: JonD, ESP2, 11/19/94.
       2
       3  ! VoiceL[2048] (signed, 24-bit q0, left justified) is the delayline we will tap.
       4  ! The tap point will undergo modulation.
       5
       6  ! chorus_width (unsigned GPR, q0) is the peak excursion (in samples) of the delayline
       7  ! modulation about the tap point, nominal_delay.
       8
       9  ! nominal_delay (unsigned AOR, q0) is the nominal whole sample delay into the delayline;
      10  ! i.e., the center tap point.
      11
      12  ! yqn (signed GPR, q23 (all fractional)) is the full-scale oscillator modulator output.
      13
      14  PROGRAM  Linear
      15
      16  DEFCONST
      17     CLK = 40.e6
      18     Fs = 44100.
      19     Freq = 1.0                                ! Hz
      20     Pi = 4.0*ATAN(1.0)
      21     EPSILON = 2.0*SIN((Pi*Freq)/Fs)
      22     CHORUS_WIDTH = 350                        ! CHORUS_WIDTH < NOMINAL_DELAY.
      23     NOMINAL_DELAY = 400                       ! In samples.  Altered to suit application.
      24
      25  PROGSIZE  <= INT(CLK/(4.*Fs))                ! assuming one big loop
      26
      27  DEFREGION  V
      28     VoiceL[2048]
      29     minus_one = SIZEM1V                       !AOR declaration = BASE modulo decrement
      30     nominal_delay = &VoiceL[NOMINAL_DELAY]
*** linear.e2 31: warning: 'm_aor': never used
      31     m_aor                                     !see Style section
      32
```

```
     DEFSPR
        PC = LinInterp
        REPT_CNT = 0
        SER_CONF = $007fff
        HARD_CONF = $008400

     DEFGPR
        GLOBAL
           chorus_width = CHORUS_WIDTH
           epsilon = EPSILON
        LOCAL
           yqn = 0
           yn  = -COS((Pi*Freq)/Fs)
           frac=0   frac_u=0         xL          vibrato=0


     CODE
     !****************************************************************************
000  LinInterp: NOP                 !executes prior to suspension. Want no SER access here.

     !************** serial I/O *************
001  MOV SER0L > xL                 MOV vibrato > SER7L    !coming out of suspension

     !********************** feed delayline input *************************
002  MOV xL > VoiceL[0]

     !********************* Linear interpolation ***************************
003  MOV nominal_delay > MACP
004  MACP + yqn X chorus_width > &VoiceL[ ]                            !addr,int part
005        frac  X  *&VoiceL[(+)] > MAC       LSH  MACRL >>1  > frac   !fractional part
006  MAC + frac_u  X  *&VoiceL[ ] > vibrato   DIFF  frac > frac_u
     !**********************************************************************

     !****** hyperstable near-quadrature oscillator (second modified coupled form) ******
007  NOP                            MOV yqn > MACP
008  NOP                            MOV yn > MACP
009  MACP - epsilon X yn  > yqn
00a  MACP + epsilon X yqn > yn

     !***************** sample sync/delayline memory shift *****************************
00b  NOP                            JMP  LinInterp
00c  NOP                            BIOZ                   UPDATE  BASEV += minus_one
     !**************************** end program ***************************************
```

```
         .------------------------------------------------ D:  MAC D-OPERAND
         |   .-------------------------------------------- E:  MAC E-OPERAND
         |   |   .---------------------------------------- F:  MAC F-OPERAND
         |   |   |   .------------------------------------ OP: MAC OPCODE
         |   |   |   |  .--------------------------------- S:  MAC SHIFT
         |   |   |   |  | .------------------------------- s:  MAC SKIP BIT
         |   |   |   |  | |  .---------------------------- A:  ALU A-OPERAND
         |   |   |   |  | |  |   .------------------------ B:  ALU B-OPERAND
         |   |   |   |  | |  |   |   .-------------------- C:  ALU C-OPERAND
         |   |   |   |  | |  |   |   |   .---------------- OP: ALU OPCODE
         |   |   |   |  | |  |   |   |   |  .------------- s:  ALU SKIP BIT
         |   |   |   |  | |  |   |   |   |  |  .---------- G:  AGEN G-OPERAND
         |   |   |   |  | |  |   |   |   |  |  |  .------- OP: AGEN OPCODE
         |   |   |   |  | |  |   |   |   |  |  |  |  .---- R:  AGEN REGION
         |   |   |   |  | |  |   |   |   |  |  |  |  | .-- L:  AGEN DATA LATCH
         |   |   |   |  | |  |   |   |   |  |  |  |  | | .- s:  AGEN SKIP BIT
         |   |   |   |  | |  |   |   |   |  |  |  |  | | |
         D   E   F   OP S s  A   B   C   OP s  G  OP R L s
000  51  1d4 1cc 3ff 02 6 0  3ff 3f1 3f1 0b 0  200 6 0 0 0
001  54  3ef 1cb 0f9 03 7 0  3ff 0f8 3e1 0b 0  200 6 0 0 0
002  57  0f9 1cb 1ef 03 7 0  3ff 3f1 3f1 0b 0  2fc 1 7 0 0
003  60  2fe 1cb 1d2 03 7 0  3ff 3f1 3f1 0b 0  2fb 4 7 0 0
004  61  0fd 0ff 2fb 06 7 0  3ff 3f1 3f1 0b 0  2fb 0 7 0 0
005  62  0fb 1ff 3ff 00 7 0  0f7 1d4 0fb 10 0  200 6 0 0 0
006  63  0fa 1ff 0f8 0a 7 0  0f6 0fb 0fa 1a 0  200 6 0 0 0
007  67  1d4 1cc 3ff 02 6 0  3ff 0fd 1d2 0b 0  200 6 0 0 0
008  68  1d4 1cc 3ff 02 6 0  3ff 0fc 1d2 0b 0  200 6 0 0 0
009  69  0fe 0fc 0fd 07 7 0  3ff 3f1 3f1 0b 0  200 6 0 0 0
00a  70  0fe 0fd 0fc 06 7 0  3ff 3f1 3f1 0b 0  200 6 0 0 0
00b  73  1d4 1cc 3ff 02 6 0  000 000 3ff 1c 0  200 6 0 0 0
00c  74  1d4 1cc 3ff 02 6 0  3ff 3f1 3f1 19 0  2ff 7 7 0 0

Integer Constants          Value          (Hex)       Scope
-----------------
CHORUS_WIDTH. . . . . . .  350          $0000015e     local
NOMINAL_DELAY . . . . . .  400          $00000190     local

Real Constants             Value                Scope
--------------
CLK . . . . . . . . . . .  40000000             local
EPSILON . . . . . . . . .  0.000142475857       local
Freq. . . . . . . . . . .  1                    local
Fs. . . . . . . . . . . .  44100                local
Pi. . . . . . . . . . . .  3.14159265359        local
```

```
GPRs                      Address  Contents  (Base 10)  Scope
----
chorus_width. . . . . . . $0ff     $00015e    350       global
epsilon . . . . . . . . . $0fe     $0004ab    1195      global
yqn . . . . . . . . . . . $0fd     $000000    0         local
yn. . . . . . . . . . . . $0fc     $800000   -8388608   local
frac. . . . . . . . . . . $0fb     $000000    0         local
frac_u. . . . . . . . . . $0fa     $000000    0         local
xL. . . . . . . . . . . . $0f9     ***        ***       local
vibrato . . . . . . . . . $0f8     $000000    0         local

Constant GPRs             Address  Contents  (Base 10)  Expression
-------------
ALU $005, line 62 . . . . $0f7     $ffffff   -1         (???)
ALU $006, line 63 . . . . $0f6     $7fffff    8388607   (???)

External Memory Arrays    Root     Absolute   Size      Region  Scope    ID
----------------------
VoiceL. . . . . . . . . . $000000  $000000    2049      V       local    ***

AORs                      Address  Contents  (Base 10)  Region  Scope
----
minus_one . . . . . . . . $2ff     $000800    2048      V       local
nominal_delay . . . . . . $2fe     $000190    400       V       local
m_aor . . . . . . . . . . $2fd     ***        ***       V       local

Preset AORs               Address  Contents  (Base 10)  Region  Reference
-------------
MAC $002, line 57 . . . . $2fc     $000000    0         V       VoiceL[0]
MAC $004, line 61 . . . . $2fb     ***        ***       V       VoiceL[ ]

SPRs                      Address  Contents  (Base 10)  Scope
----
ZERO. . . . . . . . . . . $3ff     ***        ***       local
HARD_CONF . . . . . . . . $3f3     $008400    33792     local
SER_CONF. . . . . . . . . $3f2     $007fff    32767     local
REF . . . . . . . . . . . $3f1     ***        ***       local
SER0L . . . . . . . . . . $3ef     ***        ***       local
SER7L . . . . . . . . . . $3e1     ***        ***       local
BASEV . . . . . . . . . . $3ca     $000000    0         local
SIZEM1V . . . . . . . . . $3c9     $000800    2048      local
ENDV. . . . . . . . . . . $3c8     $000800    2048      local
DIL0. . . . . . . . . . . $1ff     ***        ***       local
DOL0. . . . . . . . . . . $1ef     ***        ***       local
PC. . . . . . . . . . . . $1dd     $000000    0         local
REPT_CNT. . . . . . . . . $1d6     $000000    0         local
MACRL . . . . . . . . . . $1d4     ***        ***       local
MACP_HC . . . . . . . . . $1d2     ***        ***       local
ONE . . . . . . . . . . . $1cc     ***        ***       local
MINUS1. . . . . . . . . . $1cb     ***        ***       local
```

```
Labels                    Value   (Base 10)
------
LinInterp . . . . . . . . $000     0

GPR Memory Map
     0/8      1/9      2/a      3/b      4/c      5/d      6/e      7/f
000  ........ ........ ........ ........ ........ ........ ........ ........
008  ........ ........ ........ ........ ........ ........ ........ ........
                                  .
                                  .
                                  .
078  ........ ........ ........ ........ ........ ........ ........ ........

     0/8      1/9      2/a      3/b      4/c      5/d      6/e      7/f
080  ........ ........ ........ ........ ........ ........ ........ ........
                                  .
                                  .
                                  .
0f0  ........ ........ ........ ........ ........ ........ 7fffff,c ffffff,c
0f8  000000,l ******,l 000000,l 000000,l 800000,l 000000,l 0004ab,g 00015e,g

AOR Memory Map
     0/8      1/9      2/a      3/b      4/c      5/d      6/e      7/f
200  ........ ........ ........ ........ ........ ........ ........ ........
208  ........ ........ ........ ........ ........ ........ ........ ........
                                  .
                                  .
                                  .
278  ........ ........ ........ ........ ........ ........ ........ ........

     0/8      1/9      2/a      3/b      4/c      5/d      6/e      7/f
280  ........ ........ ........ ........ ........ ........ ........ ........
288  ........ ........ ........ ........ ........ ........ ........ ........
                                  .
                                  .
                                  .
2f8  ........ ........ ........ 000000,c 000000,c ******,l 000190,l 000800,l
```

```
Summary:

Declared Regions          Base      Size      Available
----------------
V . . . . . . . . . . . . $000000   2049      0

Total GPRs. . . . . . . . 10  (9 initialized)
Total AORs. . . . . . . . 5   (4 initialized)
External Memory Used. . . 2049 words
Instructions Available. . 213

source lines: 75  microinstructions: 13
no errors, 1 warning
```

AGEN Listing (.agn) of ESP2 Linear Interpolation Program

```
! ESP2 Assembler Version 0.40 [29 November 1994]
! AGEN listing: 01/14/95 18:10:59
! linear.e2: source lines: 75  microinstructions: 13
! no errors, 1 warning

! Linear interpolation of delayline: JonD, ESP2, 11/19/94.

! VoiceL[2048] (signed, 24-bit q0, left justified) is the delayline we will tap.
! The tap point will undergo modulation.

! chorus_width (unsigned GPR, q0) is the peak excursion (in samples) of the delayline
! modulation about the tap point, nominal_delay.

! nominal_delay (unsigned AOR, q0) is the nominal whole sample delay into the delayline;
! i.e., the center tap point.

! yqn (signed GPR, q23 (all fractional)) is the full-scale oscillator modulator output.

PROGRAM  Linear

DEFCONST
   CLK = 40.e6
   Fs = 44100.
   Freq = 1.0                                      ! Hz
   Pi = 4.0*ATAN(1.0)
   EPSILON = 2.0*SIN((Pi*Freq)/Fs)
   CHORUS_WIDTH = 350                              ! CHORUS_WIDTH < NOMINAL_DELAY.
   NOMINAL_DELAY = 400                             ! In samples.  Altered to suit application.

PROGSIZE  <= INT(CLK/(4.*Fs))                      ! assuming one big loop

DEFREGION  V
   VoiceL[2048]
   minus_one = SIZEM1V                             !AOR declaration = BASE modulo decrement
   nominal_delay = &VoiceL[NOMINAL_DELAY]
   m_aor                                           !see Style section

DEFSPR
   PC = LinInterp
   REPT_CNT = 0
   SER_CONF = $007fff
   HARD_CONF = $008400
```

```
DEFGPR
   GLOBAL
      chorus_width = CHORUS_WIDTH
      epsilon = EPSILON
   LOCAL
      y_qn = 0
      yn  = -COS((Pi*Freq)/Fs)
      frac=0   frac_u=0         xL          vibrato=0


CODE
!*****************************************************************************
LinInterp:
NOP
!************** serial I/O *************
MOV  SER0L > xL                         MOV  vibrato > SER7L
!********************** feed delayline input *************************
MOV  xL > DOL0                          NOP                      WR DOL0 > VoiceL[0]V
!********************* Linear interpolation ***************************
MOV  nominal_delay > MACP               NOP                      RD VoiceL[(+)]V > DIL0
MACP + y_qn X chorus_width > &VoiceL[ ]  NOP                      RD VoiceL[ ]V > DIL0
frac X DIL0 > MAC                       LSH  MACRL >> 1 > frac
MAC + frac_u X DIL0 > vibrato           DIFF frac > frac_u
!***********************************************************************
!****** hyperstable near-quadrature oscillator (second modified coupled form) ******
NOP                                     MOV  y_qn > MACP
NOP                                     MOV  yn > MACP
MACP - epsilon X yn > y_qn
MACP + epsilon X y_qn > yn
!***************** sample sync/delayline memory shift ******************************
NOP                                     JMP  LinInterp
NOP                                     BIOZ                    UPDATE  BASEV += minus_one
!**************************** end program ***************************************
```