Programmable DSP Architectures: Part I

Edward A. Lee

This two-part paper explores the architectural features of single-chip programmable digital signal processors (DSPs) that make their impressive performance possible. This part discusses the most basic such feature, the integration of a hardware multiplier/accumulator into the data path, and a more subtle feature, the use of several (up to six) independent memory banks. These features are studied in light of the performance benefit and the impact on the user. Representative DSPs from three manufacturers, AT&T, Motorola, and Texas Instruments, are used to illustrate the ideas. It is not the intent of the author to catalog available DSPs or their features or to endorse particular manufacturers. It is the intent to compare different solutions to the same problems. The second part, to appear in the next issue of *ASSP Magazine,* discusses pipelining and makes some bold predictions about the future of DSPs.

1. INTRODUCTION

PROGRAMMABLE DSPs are specialized microcomputers for real-time number crunching. Target applications require extensive arithmetic computation, usually with hard real-time constraints. This two-part paper studies the architectural techniques used to get the performance that is required for such applications, and especially concentrates on the impact of these techniques on the user. This impact is considerable because DSPs are traditionally designed for performance, not extensive functionality or programmer convenience.

Because of their specialized applications, programmable DSPs have evolved architectures that are significantly different from conventional microprocessors. On certain DSP-related benchmarks, their performance has consistently exceeded that of microprocessors with arithmetic co-processors by more than an order of magnitude throughout their ten year history. This performance advantage is still evident today, although microprocessors have improved dramatically and DSPs have acquired many of their features.

A number of architectural innovations have been used to achieve this impressive performance. The most basic is the integration of fast multiplier/accumulator hardware into the data path; the arithmetic is not done on a co-processor which is separated from the main data path, but rather is an integral part of execution of every instruction. Obviously, this is advantageous when most of the instructions involve arithmetic. With careful organization of the architecture, the instruction cycle time can be made equal to the cycle time of the arithmetic hardware. DSPs that are expected to be shipped within the next year will be capable of a full 32-bit floating-point multiply and add every 60–80 nsec. That is roughly 25 to 33 Mflops, an impressive computational speed comparable to that of today's mainframe computers. But integrated fast arithmetic does not tell the whole story. While necessary for such performance, it is not sufficient. Today's DSPs use extensive pipelining, several independent memories, parallel function units, and hardwired design (not microprogrammed). These architectural features often have a serious detrimental impact on the assembly language, and hence on the user or compiler designer. Fortunately, this situation is improving.

We will illustrate most ideas with widely used DSPs from three manufacturers, AT&T, Motorola, and Texas Instruments, summarized in Table 1. Other important DSPs are listed in Table 2. Most of the architectural features of the DSPs in Table 2 are also represented in Table 1, so their explicit inclusion in this paper would be redundant. The reader is urged to contact the manufacturers for complete and up-to-date specifications and not to rely on the data presented in this paper. Other useful surveys of DSPs are given in [Nis86], [Owe84], and [All85].

1.1. *A Little History.*

DSPs began to appear roughly ten years ago, with the Bell Labs DSP1 [Bod81] and NEC 7720 [Nis81] being the first that qualify. The 7720 architecture is still being updated today (the 77C25 appeared this year), making it the most durable single-chip DSP architecture. The Bell Labs DSP1 was never marketed outside AT&T. The AMI S2811 [Nic78] was announced earlier, but delivery was delayed because of problems with the VMOS technology. All of these DSPs had internal memory, permitting stand-alone system implementations.

A signal processor available before 1979 is the Intel 2920 [Tow79], which included A/D and D/A capability on

The views expressed in this paper are those of the author and do not reflect an endorsement or policy of the ASSP Society, the Publications Board or the *ASSP Magazine* editorial personnel.

0740-7467/88/1000-0004$1.00©1988 IEEE

TABLE 1. This paper uses the DSPs listed in this table for most examples. These DSPs are deemed by the author to be representative of most important architectural techniques. Their selection does not constitute a judgement about their superiority nor an endorsement. In several cases, slower versions are available for lower cost. In all cases, faster versions have appeared or are expected to appear after the original. The number of bits in the operands are represented as x/y, where x is the number of bits in the multiplier operands, y is the number of bits in the adder of ALU operands.

Company	Part	Intro Date	MAC Time (ns)	no. bits fixed pt.	no. bits float. pt.
AT&T	DSP32	84	160	16	32/40
	DSP32C	88	80	16 or 24	32/40
	DSP16	87	55	16/36	
	DSP16A	88	33	16/36	
Motorola	DSP56001	87	74.1	24/56	
	DSP96002	89	75	32/64	44/96
Texas Inst.	TMS32010	82	390	16/32	
	TMS32020	85	195	16/32	
	TMS320C25	87	100	16/32	
	TMS320C30	88	60	24/32	32/40

chip, but lacked a hardware multiplier. A number of microcontrollers are also excluded for the same reason. Also, there were (and still are) multi-chip computers specialized for signal processing, but I again will not classify them as programmable DSPs. Finally, numerous digital filter, modem, speech synthesis, and speech recognition chips have been available since the early 1970s, but because of their limited programmability they are not included in this survey.

In 1982, Texas Instruments introduced the TMS32010 [Mag82], the first member of what was to become the most popular DSP family [Mag85] [Abi86] [Sim87]. Its popularity was due in no small part to TI's emphasis on development software and hardware. At about the same time, Hitachi introduced the first CMOS DSP, the HD61810 (HSP) [Hag83]. It was also the first DSP to use a floating point format (a 12 bit mantissa and 4 bit exponent). One year later, Fujitsu greatly increased the speed of commercially-available programmable DSPs with the MB8764 [Kik83], which has a 120 nsec multiply and accumulate time.

In 1984, the Bell Labs DSP32 appeared on the market [Ker85]. It was a first in two important respects: the first DSP that AT&T would market, and the first 32-bit floating-point DSP. Shortly thereafter, NEC introduced the μPD77230 [Kaw86], a 32-bit floating-point processor with a multiply and accumulate time of 150 ns. NEC achieved this impressive speed partly by separating the normalization of floating-point results into a separate instruction. The first floating point processor using the IEEE standard format is the Fujitsu MB86232, which appeared in 1987 [Gam87]. Motorola is a relative newcomer in the marketplace; they began shipping their first DSP, the DSP56001 [Klo86], in 1987. It is a 24-bit fixed point DSP.

The most recent crop of DSPs, all available or expected in 1988 or early 1989, include both faster fixed-point devices such as the Hitachi DSPi [Kan87] and AT&T DSP16A, and three new floating-point devices, the AT&T DSP32C [Bod88], the TI TMS320C30, and the Motorola DSP96001 and 96002. The Motorola DSP, like the Fujitsu MB82232, uses the IEEE floating-point standard internally, and Zoran plans to join shortly with the 35325.

1.2. Basic Benchmarks.

In both Table 1 and Table 2, the multiply and accumulate (MAC) times are listed. This is one of the most basic statistics for comparing the performance of programmable DSPs. As defined in this paper, the MAC time is the time per tap for a very long FIR filter with in-line code. This generally reflects the maximum rate at which instructions involving both multiplication and accumulation can be issued. For most of the processors it is equal to the minimum instruction cycle time. Using the MAC time, a crude estimate of the ability of a given DSP to handle a certain application can be made as shown in Table 3. As can be seen from this table, quite sophisticated signal processing can be done with today's DSPs in the voiceband range, but real-time video is out of the range of a single DSP.

More meaningful benchmarks would be FFTs, IIR filtering, division, and higher level functions. Comparing these benchmarks, however, is tricky, because the benchmarks are not always completely described and often don't exactly match those of the competition. In this paper, we avoid this quagmire. DSP manufacturers are only now beginning to use standard benchmarks in combination with optimizing C compilers. This will facilitate comparison of the devices in the future.

Programmable DSPs have successfully been applied in voice and audio band real-time DSP products such as voiceband data modems, echo cancelers, voice and digital audio coding, voice synthesis, speech recognition, digital audio equalization, music synthesis, music pro-

TABLE 2. Other important single-chip programmable DSPs.

AMI	S2811 S28211/2	1978. First DSP described. 12/16 bit fixed point. 300 ns MAC time [Nic78]. 1983. Update of 2811.
Analog Devices	ADSP-2100 ADSP-2100A ADSP-2101/2	1986. 125 ns MAC time, 16/40 bit fixed. Memory is off-chip. 1988. Update of 2100, 80 or 100 ns MAC time. 1988. 2100A with internal RAM and peripherals. 2102 has mask-programmable program ROM.
AT&T	DSP1 DSP20	1979. An early DSP, used internally. 800 ns MAC. 16 and 20 bit operands to the multiplier [Bod81]. 1981. Update of DSP1. 400 ns MAC time. Also not marketed.
Fujitsu	MB8764 MB87064 MB86232 MB86220	1983. 100 ns MAC time. 16/26 bit fixed [Kik83]. 1985. Version of 8764 with fewer pins. 1987. 32b floating point with 150 ns MAC (75 ns fixed point). IEEE std. 1989? 24b floating point with 160 ns MAC (80ns fixed point). Similar to 8764.
Hitachi	HD61810 DSPi	1982. 12/16 bit floating point. 250 ns MAC time. 1988. For image processing, very fast I/O. 50 nsec MAC time [Kan87].
IBM	Hermes	1981. 100 ns MAC time accomplished with a clever technique called "ZIPing" [Ung85]. Not marketed outside IBM.
Motorola	DSP56000	1987. Mask-programmed version of 56001.
National	LM32900	1987. 100 ns MAC, 16/32 bit fixed, no internal memory.
NEC	μPD7720 μPD7720A μPD7281 μPD77230 μPD77220 μPD77C25	1980. 250 ns MAC. Heavily used, early DSP [Nis81]. Update of 7720. 244 ns MAC. 1984. Data flow machine for image processing. A very unusual device. 1985. 32b floating point, 150 ns MAC [Kaw86]. 1986. 24/48b fixed, 100 ns MAC, subset of 77230. 1988. Update of 7720A. 122 ns MAC.
Oki	6992	1986. 100 ns MAC time, 22 bit floating point.
Philips/ Signetics	5010 5011	125 ns MAC time, 16 bits words. 5010 with no internal program ROM.
Thomson/ Mostek	68931 68930	1986. Noted for complex data types. 360 ns complex MAC time, 16/32 bit fixed. [Bar88]. 1987. Version of 68931 with fewer pins.
Toshiba	6386/7	1983. 16/31 bit fixed. 250 ns MAC.
Texas Instr.	TMS32011 TMS320C10 TMS320C15 TMS320E15 TMS320C17 TMS320E17	1985. Version of TM32010 with μ-law and A-law conversion. 1986. CMOS version of the TMS32010. 1987. CMOS version of the 32010 with larger memories. 1987. Version of the 320C15 with EPROM. 1987. CMOS version of the 32011 with larger memories. 1987. Version of the 320C17 with EPROM.
Zoran	34161 34322 35325	1986. "Vector signal processor", 100 ns MAC time, 16 bit block floating point. High level DSP instructions (e.g. DFT). 1988. 32 bit block floating point. 1989? 32 bit IEEE floating point.

cessing, sample-rate converters as well as miscellaneous telecommunications applications. They have also been applied to non-real-time applications such as graphics, medical imaging, simulation, and general image processing. The expressive power of their assembly languages is approaching that of microprocessors, so we can expect to see an explosion of applications. There is no fundamental impediment to using these devices for all types of numeric-intensive applications. In the opinion of this author, the only factor holding back such growth is the difficult programming and the poor performance of existing compilers. This is an active area of research and develop-

TABLE 3. A crude assessment of the capability of a programmable DSP to handle a certain application can be made by considering the MAC time and comparing it against the time between samples for the application.

application class	sample rate	sample period	MACs/sample (100 ns MAC)	MACs/sample (50 ns MAC)
voiceband	8 kHz	125 μsec	1,250	2,500
audio	44 kHz	22.7 μsec	227	454
video	5 MHz	200 nsec	2	4

ment, however, so we can expect significant improvement in the near future.

1.3. *Overview of the Paper.*

The first feature that a prospective user needs to understand is the numeric format of the DSP. Consequently, this paper begins with a brief discussion of the tradeoff between fixed and floating-point DSPs. In the process, some data paths are described. After this, we explore a variety of parallel memory organizations derived from the basic Harvard architecture. These parallel memory organizations are crucial to the performance of the DSPs. Part II of this paper, to appear in the next issue of *ASSP Magazine*, describes the pipelining of DSPs. The fundamental differences between programming styles for DSPs are simply different ways of dealing with pipelining. Part II will also discuss the future of programmable DSPs.

2. ARITHMETIC

Programmable DSPs come in two basic flavors, fixed and floating point. The floating-point DSPs are more expensive and generally at least 50% slower than fixed-point devices in comparable technology. The main advantage of floating-point devices is that they free the user from concerns about scaling. The main disadvantage is a speed and cost penalty. Because of these penalties, fixed-point DSPs are here to stay. Consequently, it is useful to understand the implications of programming a fixed-point device.

2.1. *Fixed Point Arithmetic.*

Most of us are accustomed to using floating-point numbers by default, not worrying about word lengths and only minimally worrying about overflow and underflow. When writing programs we typically restrict our use of fixed-point numbers to array indices, loop counters, or other variables that have inherently bounded integer values. Unfortunately, when using a fixed-point DSP, the programmer has to pay more attention to the limitations of the number system. In particular, overflow must be prevented and precision must be carefully conserved by scaling signals. Precision is lost through quantization errors arising from two sources, A/D and D/A conversion and multiplication. The former is well documented elsewhere and does not depend on the DSP architecture, so we will ignore it in this paper. The second, multiplication, is present in all DSPs, and the structure of the DSP has a significant impact.

Quantization errors occur when multiplying two numbers because the number of bits required to specify the product with full precision is equal to the sum of the number of bits in the operands. Discarding any of these bits entails a loss of information. Most DSP architectures permit the user to preserve the full product. Consider the data path in Figure 1. Notice that the operands to the multiplier have 16 bits, while the ALU and accumulator register have 32 bits. Consequently, the programmer can perform many successive multiply-and-add operations (to compute, for example, an inner product) without discarding any bits in the products. If the operands to the multiplier have N bits, it is typical to have at least $2N$ bits in the ALU and accumulators.

In most DSPs, products with $2N$ bits can be stored and manipulated as double-precision numbers. But this is expensive (in time and memory) and is usually not necessary. If only N bits of a $2N$ bit result are kept, it is up to the programmer to determine which N bits to keep. It is desirable to keep the lowest order N bits possible without causing an overflow in order to preserve as much precision as possible. The shifter after the accumulator in Figure 1 is provided for this purpose.

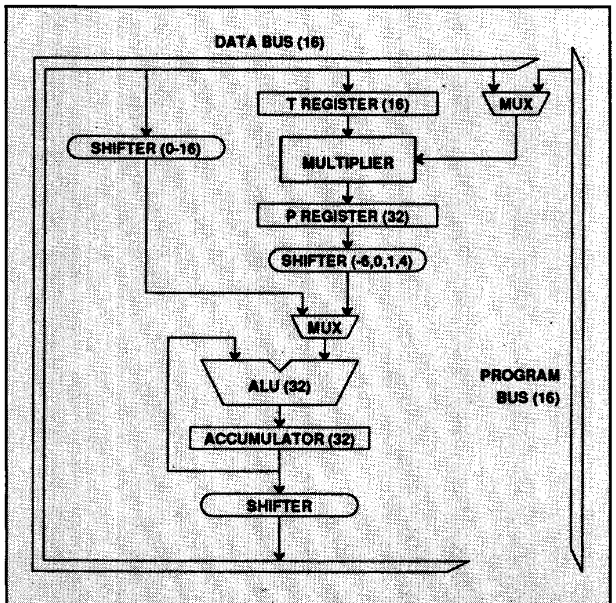


Figure 1. A simplified view of the Texas Instruments TMS320C25 data path.

Overflow occurs in two ways in a fixed-point DSP. Either the accumulator register overflows when too many numbers are added to it, or the program attempts to store N bits from the accumulator and the discarded high order bits are important. The following measures can be taken against overflow:

- **More precision:** For the first type of overflow, a partial solution is to use a larger word size for the accumulator and ALU. Consider, for example, the data path in Figure 2. The accumulators and ALU have 36 bits rather than 32. Four extra bits are provided as headroom against overflow. This permits the programmer to add up to $2^4 - 1 = 15$ 32-bit products to the accumulator with complete confidence that overflow will not occur. Most DSPs do in fact have extra headroom bits in the ALU and accumulators.
- **Saturation arithmetic:** When an overflow occurs, the DSP simply sets the value of the result to the largest magnitude positive or negative number, as appropriate, and proceeds as if nothing had happened. This is far better than simply permitting the overflow to occur, which yields a large magnitude and/or sign error. It is also better than special fault processing, which is usually inconsistent with real-time constraints. In order for this to work for both types of overflow, the ALU should have saturation hardware, and there must be saturation hardware between the accumulator and the data bus. For example, the SHIFT/SAT unit in Figure 2 performs this function.
- **Shifting products:** Before adding a product to the accumulator, it can be shifted down (with sign extension), discarding the low order bits. Of course, this entails a loss in precision. The shifters after the product registers in Figure 1 and Figure 2 can be used for this purpose. Since this shifting is effectively a scaling of the product, the programmer must take it into account.

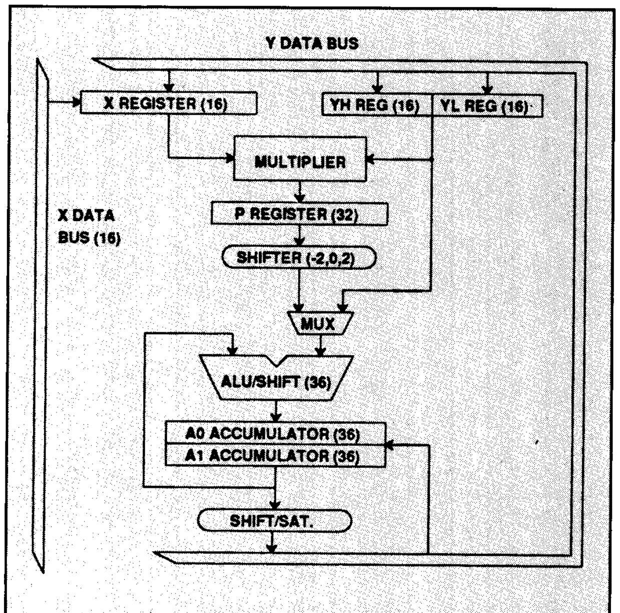


Figure 2. The simplified data path the AT&T DSP16 and DSP16A.

Example 1

All three measures are available in the Motorola DSP56001 (as well as the DSP16). Its data path is shown in Figure 3. The multiplier operands have 24 bits. There is no product register in this data path because the multiply and accumulate operations are integrated into a single indivisible hardware unit, as shown in Figure 3. Accumulators have 56 bits, or 8 bits of headroom above the 48 bit product. A limiter (saturation) is provided between the accumulators (A and B) and the 24 bit busses.

A complete solution to the overflow problem requires that the programmer be aware of the scaling of all variables so that overflow is sufficiently unlikely, where "sufficiently" depends on the application. Of course, the programmer could ensure that the values of all variables are small, but then the quantization error will be large. The objective therefore is to maximize the average magnitude of the variables subject to the constraint that overflow is sufficiently unlikely. This is not always easy to do. The programmer should consider the statistics of the incoming signal(s) and scale intermediate variables to minimize the probability of overflow while conserving as much precision as possible.

The power of the quantization error is roughly independent of the signal level as long as overflow does not occur. Consequently, the signal-to-quantization-noise ratio increases linearly with signal level, as shown in Figure 4a, until overflow occurs, at which time the signal-to-noise ratio degrades rapidly. The *dynamic range* can be defined as the range of signal levels over which the SNR exceeds a minimum acceptable SNR, X dB. With some simple assumptions about the statistics of the quantization errors, we can write the signal to quantization noise ratio as

$$SNR_{dB} = 6N + R + K \tag{1}$$

where we have assumed no overflow. R is the ratio of the input power to full-scale (in dB) and K is a constant that depends on the statistics of the quantization noise. In (1) we see that each additional bit yields an improvement in SNR of 6 dB. Referring to Figure 4a, this is equivalent to stating that the dynamic range is increased by 6 dB for each additional bit. Consequently, a 24-bit DSP such as the DSP56001 has 48 dB more dynamic range than a 16-bit DSP.

2.2. Floating Point Arithmetic.

After performing a fixed-point operation, when the contents of a full-precision accumulator register are to be stored, it would be convenient if the hardware could identify the best bits in the result to store and at the same time keep track of the scaling changes. In effect, this is the role of floating point. A floating-point number x is made up of a mantissa $M(x)$ and an exponent $E(x)$ such that

$$x = M(x) \times 2^{E(x)}. \tag{2}$$

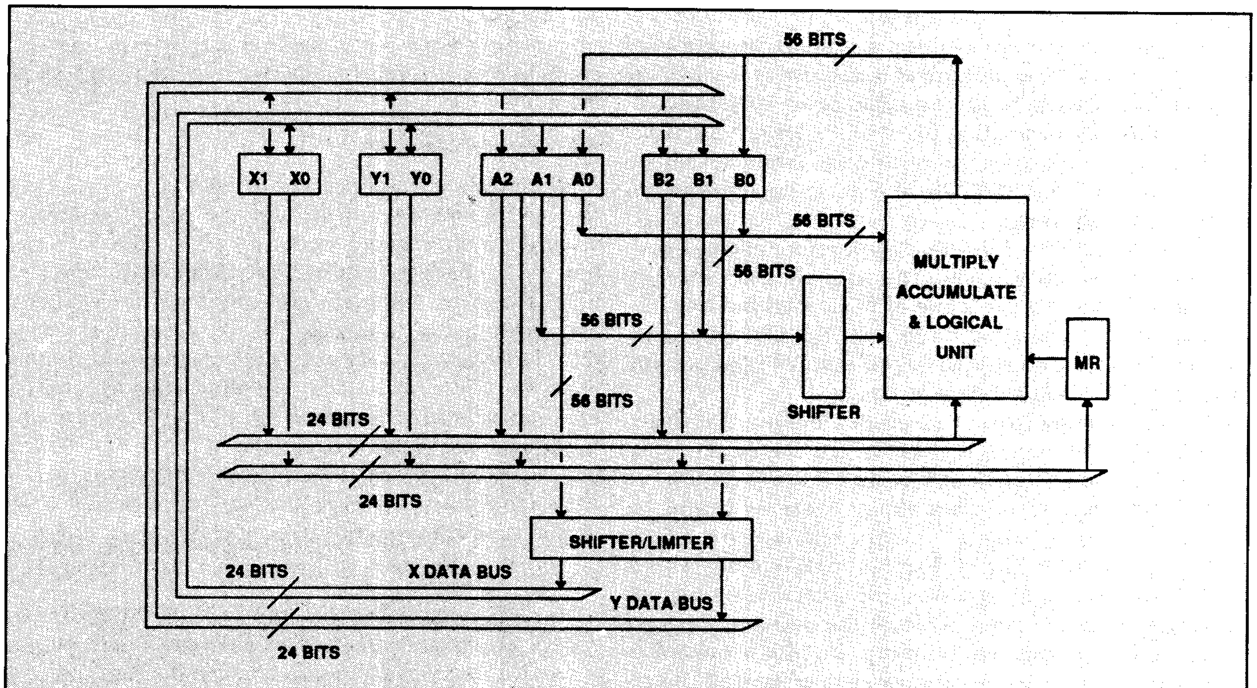


Figure 3. The simplified data path of the Motorola DSP56001.

Suppose two floating-point numbers x and y are to be multiplied. The product is

$$z = M(x) \times M(y) \times 2^{E(x) + E(y)}. \quad (3)$$

A hardware floating-point multiplier must contain both a multiplier for the mantissas and an adder for the exponents. Extra precision is usually provided to store the product of the two mantissas.

Example 2

The DSP32, DSP32C, TMS320C30, and NEC μPD77230 all allocate 8 bits to the exponent and 24 bits to the mantissa. The full-precision product of two mantissas therefore requires 48 bits, of which 32 bits are kept in the DSP32/C and TMS320C30 and 47 bits are kept in the NEC μPD77230.

Example 3

The floating-point multiplier in the DSP96002 can take operands with up to 44 bits (32 bit mantissa and 11 bit exponent). The full precision result (64 bit mantissa, 11 bit exponent) can be stored in 96-bit registers or processed with a 44-bit floating-point adder.

After each floating-point operation, numbers should be renormalized. When two mantissas are multiplied to-

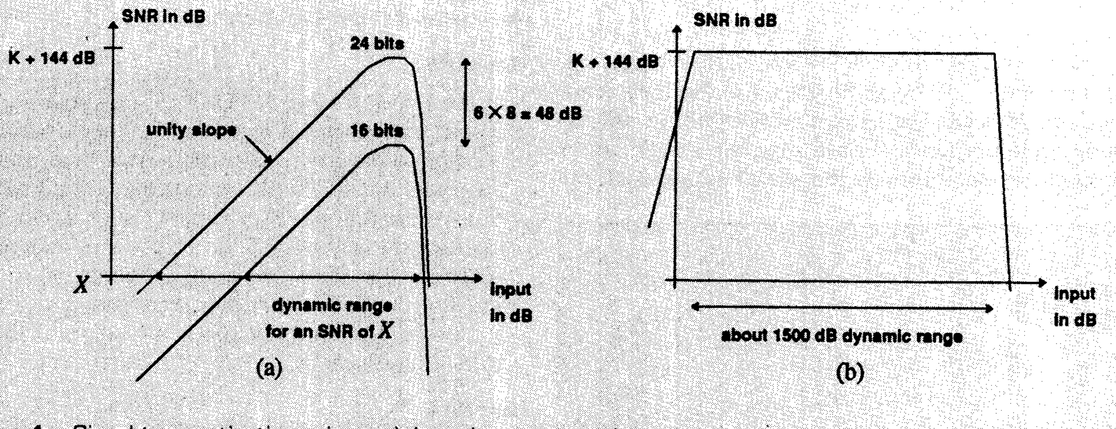


Figure 4. Signal to quantization noise and dynamic range are shown here as a function of signal level. In (a), fixed-point arithmetic with 16 and 24 bits is assumed. In (b), floating-point arithmetic with a 24 bit mantissa and 8 bit exponent is assumed. The horizontal axes are not to scale. The dynamic range of the floating-point format is much greater than that of the fixed-point format.

gether, it is possible to get a result where several of the most significant bits are identical and hence need not be stored. A shift can be performed to dispose of these extra bits and the exponent can be adjusted to keep track of the scaling. In all floating-point DSPs except the NEC μPD77230, the renormalization is done automatically in hardware. In the μPD77230, an extra normalization instruction is required.

The specific format for floating-point numbers unfortunately differs in all DSPs. The subtleties in different formats are beyond the scope of this paper, but one important point is worth mentioning. The DSP96002 and Zoran 35325 are expected to be the first DSPs to conform with the IEEE 754 floating-point standard.

A rough comparison of 32-bit floating-point formats with fixed-point formats is given in Figure 4. As shown, a 24-bit fixed-point format approaches $K + 144$ dB SNR at the point where the probability of overflow begins to dominate. This represents therefore a bound on the SNR achievable with 24 bits. A floating-point number with a 24-bit mantissa, however, essentially accomplishes perfect scaling and consequently achieves the best performance possible with 24-bits. Furthermore, an 8-bit exponent is adequate to maintain this SNR over a huge dynamic range of about 1500 dB. This means the programmer can almost ignore scaling issues.

3. MEMORY

Traditional microprocessor architectures have limited memory bandwidth and achieve high performance with register-to-register instructions. DSPs have much higher memory bandwidth and use more memory-to-memory instructions. For some DSPs, up to six memory fetches can occur in each instruction cycle. Most DSPs achieve the required memory bandwidth using parallel memory banks and small, fast, simple memories (i.e. without virtual memory or familiar cache memories). High bandwidth means that it is possible to access several memory locations in each instruction cycle.

Consider a finite impulse response (FIR) filter shown in Figure 5. Each tap requires: (1) fetching the instruction, (2) fetching two operands from memory, (3) multiplying, (4) accumulating, and (5) shifting data in the delay line. All modern DSPs can implement an FIR filter in one instruction cycle per tap. We have seen the parallel arithmetic hardware that permits the simultaneous multiply

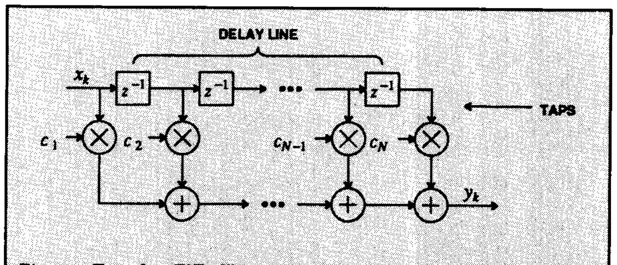


Figure 5. An FIR filter, one of the most basic benchmarks for programmable DSPs.

and accumulate, but how to achieve the simultaneous memory accesses is not as obvious. The solution is a combination of parallel memory banks and rich addressing modes.

3.1. *Parallel Memories.*

In principle, memory bandwidth can be increased with fast busses and memories; each instruction cycle would involve several memory accesses, so the memory cycle time would have to be a fraction of the instruction cycle time. However, fast memories are larger (in VLSI area) and consume more power. The increased cost and reduced memory capacity of this approach would compromise the utility of the DSP. An alternative approach is multi-ported memories, but a more common approach is to use multiple memory banks and only one or two memory cycles per instruction cycle. Six organizations are shown in Figure 6. We will illustrate these possible organizations with a sequence of examples.

Example 4

One of the earliest DSPs, the TI TMS 32010, uses the basic Harvard architecture, as shown in Figure 7. There are two memories and two busses. The fetching of an instruction coincides with the fetching of an operand for the previous instruction. Instructions with one operand from memory can be executed at a rate equal to the memory cycle time, assuming other hardware is fast enough. Contrast this with a single-memory architecture in which the instruction and its operands would have to be fetched sequentially from the same memory. Although it is possible to store data in the program memory, accessing that data (via TBLR and TBLW instructions) is slow.

The first variation on the basic Harvard architecture is to permit data stored in the program memory to be used directly in arithmetic instructions. An instruction cannot be fetched at the same time that data is being fetched, so instructions with two or three operands require two memory cycle times to execute.

Example 5

In the TMS32020 and TMS320C25, operands can be accessed from program memory. The memory cycle time is equal to the basic instruction cycle time, so instructions with two operands from memory required two cycles for execution. An example of such a two-operand instruction is the MAC (multiply and accumulate), which specifies both operands to be multiplied. We will see shortly that further modifications to the Harvard architecture permit this instruction to sometimes execute in one cycle.

Example 6

In the DSP32 and DSP32C there are two memory banks, either of which can have instructions and data. Unlike the TMS32010/20/C25, the memory cycle time is half of a basic instruction cycle time, so two accesses of each

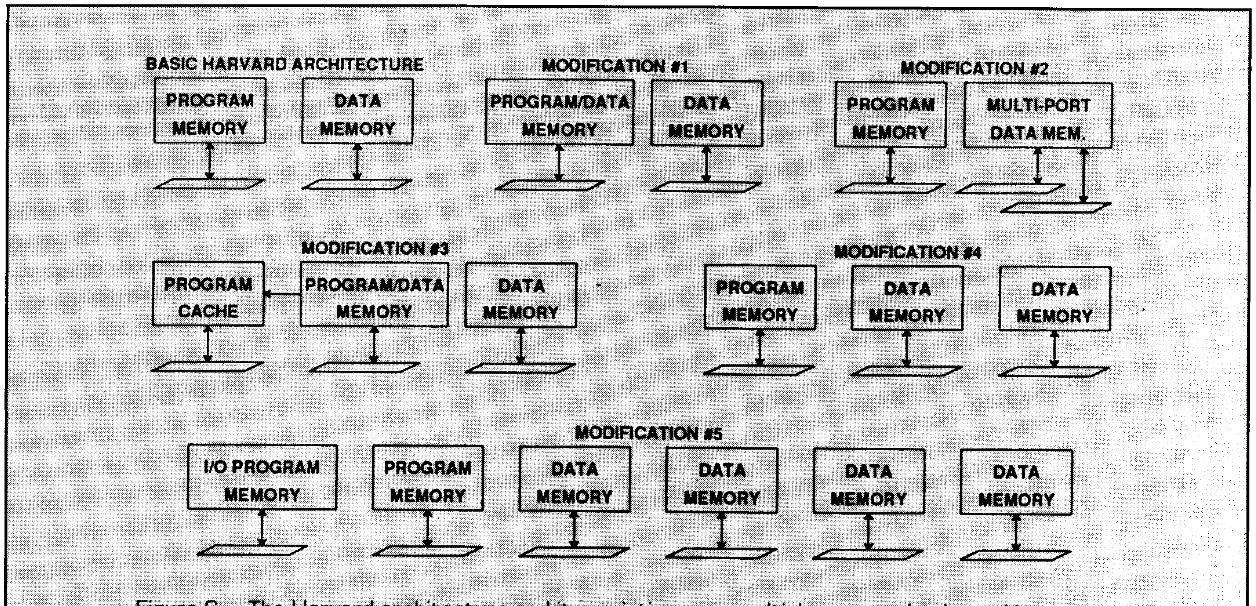


Figure 6. The Harvard architecture and its variations use multiple memory banks and busses so that up to six simultaneous memory accesses can occur.

of the two memories can be done in each instruction cycle. Instructions with up to three operands from memory can therefore be executed in one cycle. An example is

$$(m1 = a0) = a0 + m2 * m3$$

where m*i* represent memory addresses in the appropriate assembly language format (discussed below). Such an instruction does not always execute in one cycle, however. The data has to be appropriately scattered in the two memory banks, and neighboring instructions must not be trying to access the same memory banks at the same time (this will be discussed further in Part II of the paper. Fortunately, the hardware ensures correct operation by delaying instructions if necessary. Consequently, *the programmer can ignore the Harvard architecture* until it becomes necessary to optimize the code.

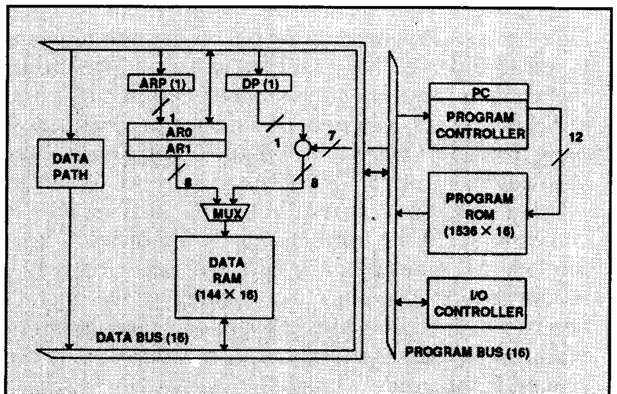


Figure 7. Simplified block diagram of one of the early DSPs, the TI TMS32010. It uses the basic Harvard architecture. The data path is similar to that in Figure 1.

The second modification to the basic Harvard architecture (see Figure 6) is to use a multi-ported data memory. Although this is often perceived as an expensive solution, it has the advantage of permitting multi-operand instructions without worry about separating operands into multiple banks.

Example 7

The Fujitsu MB86232 (see Table 2) has a program memory and a triple-ported data memory (512 words). Simultaneous access to the data memory is accomplished using three busses; taking these busses off-chip would require many pins and the external memory would be non-standard and expensive, so off-chip data memory for the MB86232 is single-ported. Multi-operand instructions using off-chip memory take multiple cycles to execute.

The third modification to the basic Harvard architecture (see Figure 6) supplements program/data memory with an instruction cache. Whenever instructions in the cache are being executed, no instruction-fetch cycle is required of the program/data memory, so a cycle is freed for a data fetch.

Example 8

The TMS32020 and TMS320C25 provide a *repeat* instruction for use in combination with instructions such as the MACD. For example, the following code segment can be used to implement an FIR filter:

```
RPTK    constant
MACD    m1,m2
```

The RPTK instruction causes the following instruction, MACD, to be loaded into a unit-length instruction cache from which it is executed the number of times specified by "constant". After the first execution of MACD, the program/data memory is not needed for instruction fetches and can therefore be used for data fetches with addresses given by $m1$ and $m2$. So the first MACD executes in two cycles, but subsequent MACDs execute in one.

The unit-length instruction cache of Example 8 serves the basic function of economizing on memory cycles, but it also permits *low-overhead loops*. The RPTK instruction provides a mechanism for iterating an instruction a fixed number of times without devoting extra instructions to setting and testing a loop counter and branching. The Hitachi 61810 was the first DSP to support low-overhead looping, although it is a common feature today. An obvious extension to the technique of Example 8 is to use a larger instruction cache.

Example 9

The DSP16 and DSP16A also use the third modification of the Harvard architecture, with an instruction cache that can hold 15 instructions (see Figure 8). Low-overhead loops can have up to 15 instructions.

Example 10

The Analog Devices ADSP-2100 is similar except that the program and data memory are off chip. The program cache (16 instructions) is on chip. The ADSP-2101/2 have some memory on chip.

Low-overhead looping is not always associated with an instruction cache; in fact, the most flexible low-overhead looping capability is found in the DSP56001 and DSP96002, which have no instruction cache. These use a further extension of the Harvard architecture, which is to expand

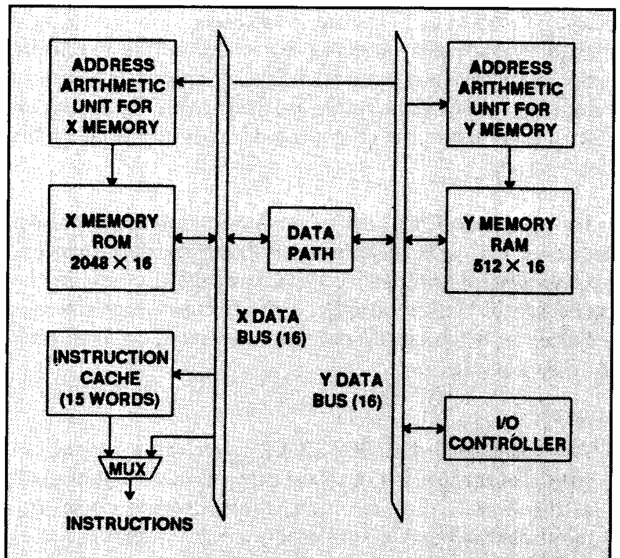


Figure 8. Simplified block diagram of the DSP16 and DSP16A showing the memory organization, which corresponds to modification #3 in Figure 6. The data path detail is shown in Figure 2.

the instruction cache until it becomes a memory bank, getting modification #4 in Figure 6. This permits simultaneous fetches of an instruction and two operands when the memory cycle time is equal to the instruction cycle time.

Example 11

The Motorola DSP56001 and 96002 use three memory banks as shown in Figure 9. If the programmer is careful to divide data so that operations with two operands from memory fetch their data from both the X and Y memories, then such operations consume only one instruction cycle. In the 96002, the memories can be accessed *twice* in each instruction cycle, like the DSP32 and 32C. The second access is used for DMA. Consequently, DMA does not interfere with program execution.

Example 12

The TMS320C30, shown in Figure 10, has three internal memory blocks similar to the banks of the DSP56001 and 96002, two RAMs and a ROM. If the program is in ROM, then instructions with two operands from memory consume only one cycle. Like the 96002, DSP32 and 32C, each memory block can be accessed twice in each instruction cycle. Unlike the 96002, the second access can be used by some multi-operand instructions, so that even if two operands come from the same memory block, the instruction still only consumes one cycle. The extra memory cycles are also available to the DMA unit to transfer external data to the memories without interfering with the executing program.

Even with such high memory bandwidth, it is still possible to have conflicts. A given sequence of instructions may try to access the same memory bank three times in one instruction cycle. In this event, the hardware detects a conflict and delays one of the instructions. Hence, like the DSP32 and 32C, the programmer can ignore the parallel memory architecture until it becomes necessary to optimize the code.

Just as with the DSP56001, 96002, DSP32 and 32C, an instruction cache is not required for instructions to execute at top speed because the memory bandwidth is high enough. However, this is only true if the program is stored on-chip. The TMS320C30 has an instruction cache that is used when the program is stored off-chip. Off-chip accesses can only occur once per instruction cycle instead of twice, and only two busses are available externally instead of four (see Figure 10). When an off-chip program access is initiated, the 320C30 first checks to see if the instruction is in the cache. If it is, the external memory cycle is conserved. Otherwise, the instruction is fetched and loaded into the instruction cache. The detailed cache strategy, reminiscent of that used in many general purpose computers, is described in the user's manual.

The TMS320C30 achieves high memory bandwidth by accessing parallel memories twice in each cycle. This

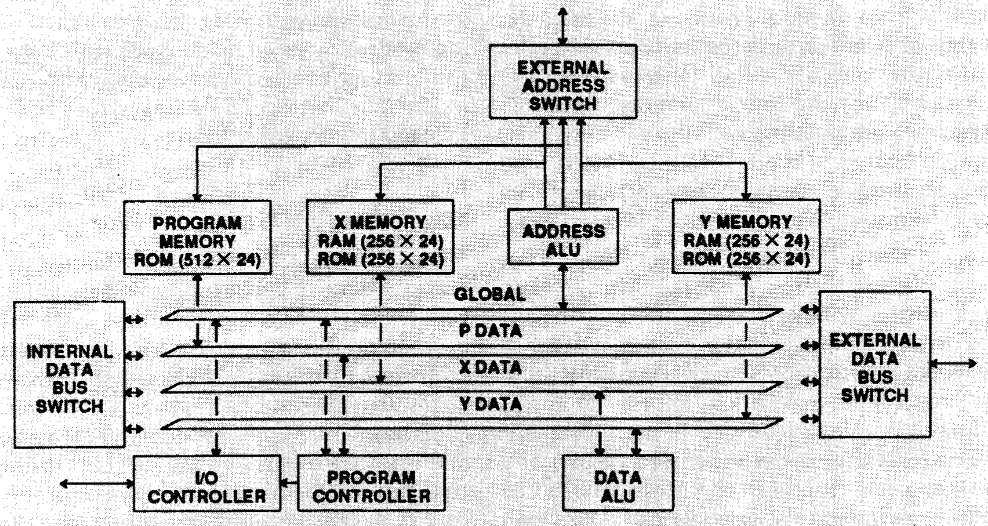


Figure 9. Simplified block diagram of the Motorola DSP56001 showing three separate memory banks. The address ALU is shown in Figure 13. The data ALU is shown in Figure 3.

means an instruction cycle time cannot be smaller than twice the memory cycle time. It is possible to get a similar effect using many more parallel memories.

Example 13

The DSPi from Hitachi uses six memory banks as in modification #5 of Figure 6. One instruction cycle time

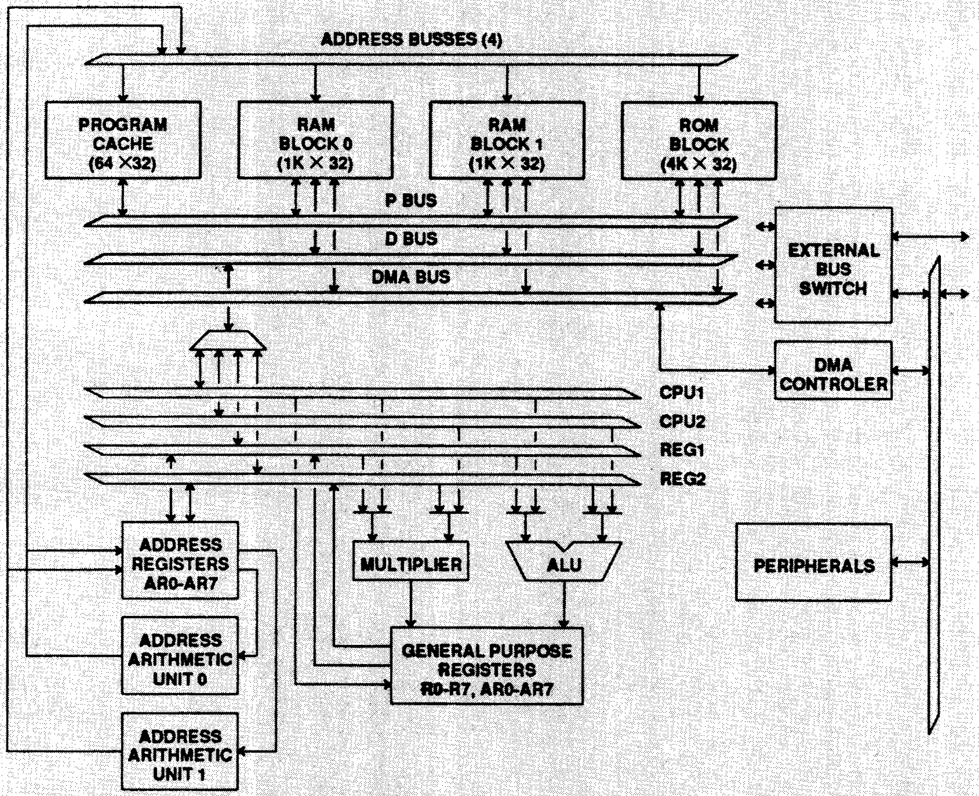


Figure 10. A simplified block diagram of the TMS320C30.

is equal to one memory cycle time. Instructions with three operands from memory (involving, for example, two reads and write) use three of the four data memories and the program memory. Simultaneously, an I/O instruction can access the fourth memory. Although faster instructions are possible with the same memory technology (compared to the DSP32C, DSP96002, and TMS320C30), the programmer must carefully arrange data in memory to take advantage of the multiple memories.

The *demand ratio* is defined to be the total number of memory cycles per instruction cycle [Kog81]. Demand ratios are summarized in Table 4. DSPs with the smallest demand ratio (two) require instruction caches in order to be able to execute instructions with two operands from memory in a single cycle. In addition, in order to implement FIR filters in one instruction cycle per tap, either circular buffer addressing modes (discussed below) or a specialized write cycle (found in the TMS32010/20/C25 and the DSP16) is required (discussed below). DSPs with a demand ratio of three (the DSP56001) do not require an instruction cache, but do require circular buffer addressing modes for delay lines. DSPs with a demand ratio of four or higher need neither circular buffer addressing modes nor instruction caches. Note, however, that the demand ratio is given for on-chip accesses and is usually much lower when memory is off-chip.

3.2. Internal and External Memory.

Traditionally, DSPs have been intended for use in highly cost-competitive applications, and the intent was to provide complete or nearly complete systems-on-a-chip. In order to do this, most DSPs are microcomputers, not microprocessors, which means that memory is on-chip. There are more subtle reasons for having memory on-chip, however. Multiple independent memory banks and multi-ported memories are difficult to implement off-chip because the pin count required for the off-chip bussing is excessive, and the speed penalty of off-chip

TABLE 4. A summary of DSP memory systems. The demand ratio is the total number of memory cycles per instruction cycle. It is specified only for internal memories, and usually differs for external memories. Also, highly specialized memory cycles are not counted; for example, the DSP16 and TMS32010/20/C25 have a specialized write cycle used to move data in a delay line. The size of the instruction cache is indicated, when it is present. The amount of internal memory is indicated (in numbers of words), as well as the size of the total address space.

part	# of banks	demand ratio	instr. cache	internal memory	external space
DSP32	2	4	no	(2)512W data RAM 512W prog ROM	14KW data/prog
DSP32C	3	4	no	(2)512W data RAM 512W prog RAM or 1kW prog ROM	4MW data/prog
DSP16	2	2	15W	512W data RAM 2KW prog/data ROM	64KW program
DSP16A	2	2	15W	2kW data RAM 4KW prog/data ROM	64KW program
DSP56001	3	3	no	(2)256W data RAM (2)256W data ROM 512W prog RAM	(3)64KW prog/data
DSP96002	3	4	no	512W prog RAM (2)512W data RAM (2)512W data ROM	4GW prog (2)4GW data
TMS32010	2	2	no	144W data RAM 1.5KW prog ROM	4KW prog
TMS32020	2	2	1W	288W data RAM 256W prog/data RAM	64KW data 64KW prog
TMS320C25	2	2	1W	288W data RAM 256W prog/data RAM 4KW prog ROM	64KW data 64KW prog
TMS320C30	3	4	64W	(2)1KW prog/data RAM 4KW prog/data ROM	16MW data/prog

bussing increases the system cost by requiring faster memories.

Example 14

The Motorola DSP96001 has a 32-bit word and three memory banks, each with a 32-bit address (for a 4GW word address space per memory bank, the largest of all the DSPs). To implement the memory externally using three address and data busses would require 192 pins for the bussing alone. Multiplexing the busses would be difficult because of the fast cycle times, and it would require costly external demultiplexing hardware. The 96001 therefore brings only one of the busses outside. The user can select the bus using a bus switch like that shown in Figure 9. The 96002 is a 200-pin version of the 96001 that brings two busses outside (the user can again select which busses). The cost is a more expensive package.

In order to converse pins for other uses, most DSPs have enough internal memory in each memory bank for viable operation. Internal busses are then multiplexed to the outside. The system designer may choose to expand only one of the memory banks off-chip, in which case full-speed operation is often possible (if fast enough off-chip memories are used). With the 96002, full-speed operation is possible when two memory banks are off-chip. Sometimes the off-chip memories can be used to store pages of data or program destined for memory banks that are not expanded off-chip, in which case the software must handle the paging.

3.3. *Addressing Modes.*

Parallel memory banks can increase memory bandwidth as discussed above, but one fundamental problem remains. Instructions must specify as many as three memory accesses. For a reasonable address space, the number of bits required to specify each address is large enough that the width of the instruction word gets large. This means more memory required to store the program, wider busses to access the program memory, or more memory cycles to access each instruction (if multi-word instructions are used).

The universal solution is register-indirect addressing modes. Register-indirect addressing modes are typically the workhorse of programmable DSPs because they permit the specification of many memory addresses in a one-word instruction. A set of *address registers* is provided for the purpose. These registers are loaded with an address of one word in a data structure (for example, the first or last sample in the delay line of Figure 5), and then all instructions accessing the data structure specify this register as the one containing the address. Since the register bank is small, few bits are required to specify the register. Furthermore, parallel hardware is provided to update registers containing memory addresses. This minimizes the number of register-load instructions required.

Consider again the FIR filter in Figure 5. A reasonable implementation begins by setting one address register to point to the end of the delay line and another register to point to the last filter coefficient. Then a sequence of instructions (one per tap) *each* do the following: (1) fetch the two operands, using the register addresses, (2) multiply and accumulate, (3) decrement the register contents, and (4) move the data operand to the following location, thus implementing a delay line.

Example 15

In the DSP32 and DSP32C, a five tap FIR filter can be accomplished with the following code:

```
r1=address of last word in delay line
r2=address of last coefficient
r3=address of last word in delay line
a1=new sample (input)
a0=*r1--**r2--
a0=a0+(*r3--=*r1--)**r2--
a0=a0+(*r3--=*r1--)**r2--
a0=a0+(*r3--=*r1--)**r2--
a1=a0+(*r3=a1)**r2
result is in a1
```

The first four instructions initialize the registers, and a0 and a1 are accumulator registers. The symbol "*" is used to indicate multiplication and also to indicate indirection. In other words, *ri refers to the contents of the memory location whose address is given by the register ri. The "--" symbol following the *ri symbols indicates that the register contents should be decremented by one *after* the instruction (but before the next instruction). This is called a *post-auto-decrement.* Notice that the arithmetic instructions accomplish the filtering with one instruction per tap.

Example 16

It is instructive to consider register-indirect addressing with one of the earlier DSPs, the TMS32010, shown in Figure 7. An extra level of indirection is introduced using an address register pointer (ARP) to point to the "current" address register. The TMS32010 has only two address registers, AR0 and AR1, so the ARP has only one bit. Only instructions with a single operand from memory are supported, so the implementation of an FIR filter is as follows:

```
LARK    AR0,address of last coefficient
LARK    AR1,address of last data word
LARP    0
LT      *-,AR1
MPY     *-,AR0
LTD     *-,AR1
MPY     *-,AR0
...
LTD     *-,AR1
MPY     *-,AR0
APAC
ADD     ONE,14
SACH    RESULT,1
```

The first two instructions load the address registers and the third instruction loads ARP. The fourth instruction uses the "current" AR (which is AR0) as its operand, denoted "*", and then sets the ARP to point to AR1. The symbol "*-" can be read: "use the contents of the current AR as the memory address for the operand, then decrement the AR by one." The rest of the program can be understood by referring to Figure 7. The LT instruction loads the T register. The MPY instruction multiplies the contents of the T register by its operand. The LTD instruction is the same as the LT instruction except that the operand is copied into the next higher memory location. This instruction is specifically designed to perform the shifting of a data in a delay line and is not usually useful for anything else. The LTD instruction also performs an addition; adding the P register to the accumulator. Note that two instructions per tap are required to implement an FIR filter in this machine. We saw in Example 8 how a unit-length instruction cache is used to reduce this to one instruction per tap in the TMS32020.

Shifting of data in a delay line is expensive in memory bandwidth. An extra memory write cycle is needed. An efficient alternative in some cases is *modulo-mode* addressing. The delay line is implemented as a circular buffer as shown in Figure 11. $M + 1$ contiguous memory locations are allocated, beginning at location L, and when a register is incremented beyond location $L + M$, it reverts to location L. Similarly, if an address register is decremented below location L, it again wraps around to higher locations. This can be done in some DSPs without extra instructions except those required to initialize some registers.

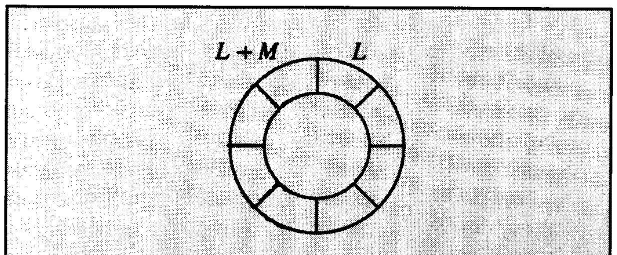


Figure 11. A circular buffer begins at memory location L and ends at memory location $L + M$, but conceptually the next memory location above $L + M$ is L.

Example 17

The Y memory in the WE DSP16 can be accessed using modulo-mode addressing by setting rb to point to the beginning of the buffer and re to point to the end, as shown in Figure 12. A hardware comparator then checks to see if the modified address is out of range and adjusts it if it is.

Example 18

The Motorola DSP56001 and DSP96002 have address registers that come in triplets, Rx, Mx, and Nx, where x

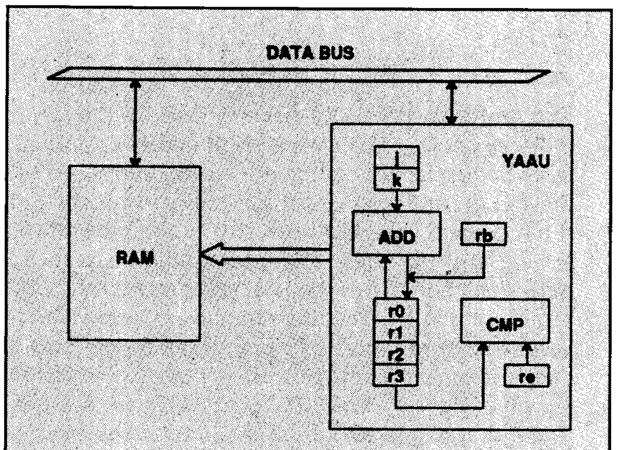


Figure 12. Modulo-mode addressing in the WE DSP16 is accomplished in the Y memory by setting rb to point to the beginning of the buffer and re to point to the end.

varies from 0 to 7, as shown in Figure 13. The address is in Rx, the increment for the post-auto-increment is in Nx, and the length of the circular buffer (if a modulo-mode addressing is being used) is in Mx. These registers are loaded and read through the global data bus (see Figure 9). Auto-increment and modulo arithmetic is performed in the two address arithmetic units, so simultaneous modulo-mode addressing can be performed in two separate memory banks. This leads to the simple FIR filter program shown below:

```
start   move    address of delay line, r0
        move    address of first coefficient, r4
        move    filter_order-1, m0
        move    m0,m4
fir     movep              x:input,(r0)
        clr     a          x:(r0)-,x0  y:(r4)+,y0
        rep     m0
        mac     x0,y0,a    x:(r0)-,x0  y:(r4)+,y0
        macr    x0,y0,a    (r0)+
        movep              a,x:output
        jmp     fir
```

The data delay line is in X memory and the coefficients are in Y memory; circular buffers are used for both. This is accomplished simply by loading the m0 and m4 registers in the third and fourth instructions. The first movep instruction fetches the input; the clr initializes the accumulator and fetches the first pair of operands; the rep indicates that the next instruction should be repeated *filter_order-1* number of times; the mac is the multiply-and-accumulate; the macr is the multiply-and-accumulate with rounding; and the final movep outputs the result.

In order to simplify the address arithmetic units, circular buffers are restricted to begin on power of two boundaries. In other words, if the length of the buffer is less than or equal to 2^k but greater than 2^{k-1}, then the low order k bits of the starting address must be zero. Consequently, the starting address of the circular buffer must fall on a multiple of 2^k. Unlike other DSPs,

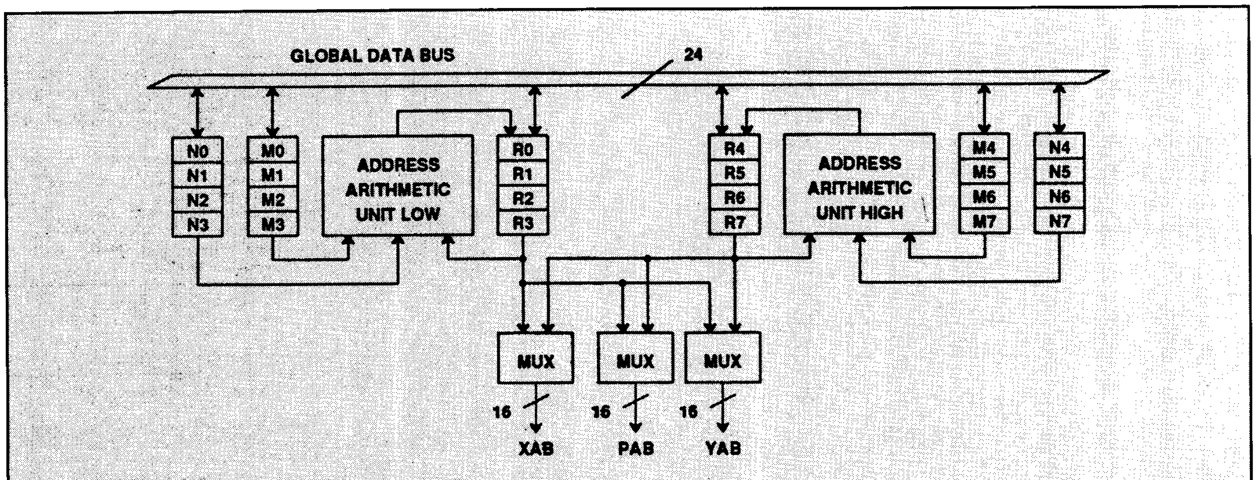


Figure 13. Modulo-mode addressing in the Motorola DSP56001 is accomplished using two registers, M*x* and N*x*, associated with each address register R*x*. M*x* specifies the length of the circular buffer, and N*x* specifies the increment (or decrement). Circular buffers must begin on power-of-two boundaries. The DSP96002 is similar.

the 56001 and 96002 support multiple simultaneous circular buffers that can reside in either memory.

Example 19

The TMS320C30 uses a single register BK to specify the length of a circular buffer. Like the DSP56001 and DSP96002, the circular buffer must begin on a power-of-two boundary. Any address register AR*x* can be used to address the circular buffer using the notation *AR*x*++(IR*y*)%. Translated, this means that the contents of ARx will be post-modified by adding the index register IR*y* in such a way that the resulting address remains inside the circular buffer (indicated by the "%"). The IR*y* can be replaced with an immediate displacement in the range 0–255 and the "++" can be replaced with a "--". The ARx must already contain an address inside the circular buffer, which implies that if the low order *k* bits of the AR*x* register are set to zero, the resulting address will be the beginning of the buffer. An FIR filter computation using circular buffers can be accomplished with the following code:

```
   LDI      filter_order+1, BK
   LDI      last_coef_address, AR0
   LDI      end_of_delay_line, AR1
   LDF      0,R0
   LDF      0,R2
   RPTS     filter_order
   MPYF3    *AR0++(1)%,*AR1++(1)%,R0
|| ADDF3    R0,R2,R2
   ADDF     R0,R2
```

The first five instructions load immediate data (integer and floating point) into registers. The sixth instruction indicates that the seventh instruction should be repeated *filter_order*+1 times. The seventh instruction is a parallel multiply and add, and takes two lines to write. It is important to note that the multiply and add occur in parallel, so although the product is placed in R0, it is not accumulated in R2 until the *next* time the parallel multiply and add instruction is executed (see the discussion on pipelining below). The first addition executed, therefore, simply adds two zeros. The final instruction performs the last add. To build a complete FIR filter from this code segment, I/O instructions must be added and an outer loop must be specified.

Register-indirect and modulo-mode addressing cannot generally be used exclusively. As seen in the above code segments, most DSPs have some form of immediate addressing where the operand is part of the instruction, and direct addressing where the address is part of the instruction. Such addressing modes are required for register load instructions, for example. In many cases, these addressing modes require two-word instructions. Such instructions usually cannot be executed as fast as one-word instructions because two fetches from program memory are required. Some DSPs have "short" immediate and direct address data which can fit in one instruction word and execute in one cycle, but the operations that can be done with such data are always more limited than what can be done with register-indirect addressing.

Other addressing modes that are found are index mode in which the increment is added to the address *before* the fetch rather than after.

Example 20

In the TMS320C30, the assembler notation *+AR*n(disp)* means that the content of address register AR*n* is added to the 8 bit constant *disp* (which is part of the instruction word) *before* the memory fetch. The notation *++AR*n*(*disp*) is similar except that AR*n* is modified. One of two special registers, IR0 and IR1, can be used in place of *disp*. Of course, post-modification is also available, and would be denoted *AR*n*++(*disp*).

TABLE 5. A summary of addressing modes. The first column specifies whether immediate data can be part of an instruction. Most processors require an extra memory location and an extra cycle for instructions with immediate data, although some DSPs support "short" immediate data in one cycle. For the DSP32, only integer instructions can have immediate data. The second column specifies whether a direct address can be part of an instruction. Again, two cycles are often required, although some processors use short addresses in combination with paging to reduce it to one cycle. When paging is used, the size of each page is indicated. The next column specifies the register-indirect addressing capability by indicating the number of address registers. The last three columns indicate modulo-mode addressing (for circular buffers), indexed addressing (for array access), and bit-reversed addressing (for FFTs).

part #	immed.	direct	indirect			bit reverse
			# reg.	modulo	index	
DSP16	yes	yes	5	yes	no	no
DSP16A	yes	yes	5	yes	no	no
DSP32	integer only	integer only	15	no	no	no
DSP32C	integer only	integer only	15	no	no	yes
DSP56001	yes	yes	8	yes	yes	yes
DSP96002	yes	yes	8	yes	yes	yes
TMS32010	yes	paged, 128W	2	no	no	no
TMS32020	yes	paged, 128W	5	no	yes	no
TMS320C25	yes	paged, 128W	8	no	yes	yes
TMS320C30	yes	paged, 64kW	8	yes	yes	yes

Example 21

The DSP56001 and 96002 also have indexed addressing, but instructions that use indexed addressing require an extra cycle to complete.

Indexed addressing is useful for random access of arrays, and can sometimes be used to write position-independent code.

The addressing modes of the DSPs in question are summarized in Table 5.

4. CONCLUSION

Without discussing it explicitly, we have already seen some evidence of pipelining in DSPs. We have seen that an instruction is fetched in parallel with the operands of the previously fetched instruction. Part II of this paper, to appear in the next issue of *ASSP Magazine,* will discuss pipelining in depth. In addition, it will put forth some bold predictions about the future of DSPs.

5. ACKNOWLEDGMENTS

The author gratefully acknowledges the careful reading and thoughtful comments of Jim Boddie, Craig Garen, and John Hartung from AT&T, Philip Goldworth and T. J. Shan of Fujistu, Kenji Kaneko from Hitachi, Bryant Wilder, Kevin Kloker, and Garth Hillman from Motorola, Takao Nishitani of NEC, and Panos Papamichalis and Ray Simar from Texas Instruments. Other helpful suggestions were made by Bob Owen. Any remaining errors are entirely the fault of the author.

REFERENCES

[Abi86] S. Abiko, M. Hashizume, Y. Matsushita, K. Shinozaki, and T. Takamizawa, "Architecture and Applications of a 100ns CMOS VLSI Digital Signal Processor," *Proceedings of ICASSP 86,* pp. 393–396, April 1986 (Describes the TI TMS320C25).

[All85] J. Allen, "Computer Architecture for Digital Signal Processing," *Proceedings of the IEEE,* May, 1985 73(5).

[Bar88] B. Barazesh, J.-C. Michalina, and A. Picco, "A VLSI Signal Processor with Complex Arithmetic Capability," *IEEE Trans. on Circuits and Systems,* Vol. 35, No. 5, May 1988 (Describes the Thomson/Mostek 68931).

[Bod81] J. R. Boddie, G. T. Daryanani, I. I. Eldumiati, R. N. Gadenz, J. S. Thompson, and S. M. Walters, "Digital Signal Processor: Architecture and Performance," *Bell Sys. Tech. J.,* 60 pp. 1449–1462, 1981 (Describes the Bell Labs DSP1).

[Bod88] J. R. Boddie, C. J. Garen, M. L. Fuccio, and J. Tow, "A Floating Point DSP with Optimizing C Compiler," *Proceedings of ICASSP,* pp. 2009–2012, New York, April, 1988. (Describes the Bell Labs DSP32C).

[Hag83] Y. Hagiwara, Y. Kita, T. Miyamoto, Y. Toba, H. Hara, and T. Akazawa, "A Single Chip Digital Signal Processor and its Application to Real-Time Speech Analysis," *IEEE Trans. on ASSP,* ASSP-31(1), 1983 (Describes the Hitachi HD61810 (HSP)).

[Gam87] H. Gambel, T. Ikezawa, N. Kobayashi, T. Tanabe, and S. Unagami, "A 32 Bit Floating Point Digital Signal Processor FDSP-4 and its Application to the Communication Systems," *Proceedings of Globecom '87,* 1987 (Describes the Fujitsu MB86232).

[Kan87] K. Kaneko, T. Nakagawa, A. Kiuchi, Y. Hagiwara, H. Ueda, H. Matsushima, T. Akazawa, and J. Ishida, "A 50ns DSP with Parallel Processing Architecture," *IEEE Int. Solid-State Circuits Conference, Digest of Technical Papers* 1987 (Describes the Hitachi DSPi).

[Kaw86] Y. Kawakami, H. Tanaka, T. Nukiyama, M. Yoshida, T. Nishitani, I. Kuroda, M. Araki, T. Hoshi, "A 32b Floating Point CMOS Digital Signal Processor," *ISSCC 86 Digest of Technical Papers,* 1986, (Describes the NEC μPD77230).

[Ker85] R.N. Kershaw, L.E. Bays, R.L. Freyman, J.J. Klinikowsi, C.R. Miller, K. Mondal, H.S. Moscovitz, W.A. Stocker, and L.V. Tran, "A Programmable Digital Signal Processor with 32b Floating Point Arithmetic," *ISSCC 85 Digest of Technical Papers,* Feb. 13, 1985 (Describes the AT&T Bell Labs DSP32).

[Kik83] H. Kikuchi, T. Inaba, Y. Kubono, H. Hambe, and T. Ikesawa, "A 23 K Gate CMOS DSP with 100 ns Multiplication," *IEEE Int. Solid-State Circuits Conference, Digest of Technical Papers,* pp. 128–129, 1983 (Describes the Fujitsu MD8764).

[Klo86] K. Kloker, "Motorola DSP56000 Digital Signal Processor," *IEEE Micro,* December, 1986.

[Kog81] P.M. Kogge, *The Architecture of Pipelined Computers,* Hemisphere Publishing Co., McGraw, New York, 1981.

[Mag82] S. Magar, E. Caudel, and A. Leigh, "A Microcomputer with Digital Signal Processing Capability," *International Solid-State Circuits Digest of Technical Papers,* Feb., 1982, pp. 32–33 (Describes TI TMS32010).

[Mag85] S. Magar, D. Essig, E. Caudel, S. Marchall, and R. Peters, "An NMOS Digital Signal Processor with Multiprocessing Capability," *International Solid-State Circuits Digest of Technical Papers,* Feb., 1985 pp. 90–91 (Describes the TI TMS32020).

[Nic78] W.E. Nicholson, R.W. Blasco, and K.R. Reddy, "The S2811 Signal Processing Peripheral" *Proceedings of WESCON,* pp. 1–12, 1978 (Describes the AMI S2811).

[Nis81] T. Nishitani, R. Maruta, Y. Kawakami, and H. Goto, "A Single-Chip Digital Signal Processor for Telecommunications Applications" *IEEE J. Solid-State Circuits* SC-16, pp. 372–376, 1981 (Describes the NEC μPD7720).

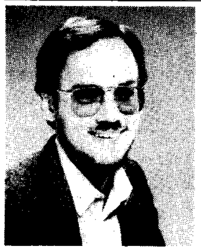
[Nis86] T. Nishitani, "Signal Processor Design Methodology," in *Design Methodologies,* edited by S. Goto, Elsevier Science Publishers B.V. (North-Holland), 1986.

[Owe84] R.E. Owen, "VLSI Architectures for Digital Signal Processing," *VLSI Design,* June, 1984.

[Sim87] R. Simar, T. Leigh, P. Koeppen, J. Leach, J. Potts, and D. Blolock, "A 40 MFLOPS Digital Signal Processor: The First Supercomputer on a Chip," *Proceedings of ICASSP 87,* pp. 535–538, April, 1987 (Describes the TI TMS320C30).

[Tow79] M. Townsend, M.E. Hoff, and R.E. Holm, "An NMOS Microprocessor for Analog Signal Processing," *IEEE J. Solid-State Circuits* SC-15(1) Feb. 1980 (Describes the Intel 2920).

[Ung85] G. Ungerboeck, D. Maiwald, H.P. Kaeser, P.R. Chevillat, and J.P. Beraud, "Architecture of a digital signal processor," *IBM Journal of Research and Development,* March 1985.



Edward A. Lee has been an assistant professor in the Electrical Engineering and Computer Science Department at U.C. Berkeley since July, 1986. His research activities include parallel computation, architecture and software techniques for programmable DSPs, design environments for real-time software development, and digital communication. He has taught short courses on the architecture of programmable DSPs and telecommunications applications of programmable DSPs. He was a recipient of the 1987 NSF Presidential Young Investigator award, an IBM faculty development award, and the 1986 Sakrison prize at U.C. Berkeley for the best thesis in Electrical Engineering. He is co-author of "Digital Communication", with D.G. Messerschmitt, Kluwer Academic Press, 1988. His B.S. degree is from Yale University (1979), his masters (S.M.) from MIT (1981), and his PhD from U.C. Berkeley (1986). From 1979 to 1982 he was a member of technical staff at Bell Labs in Holmdel, New Jersey, in the Advanced Data Communications Laboratory, where he did extensive work with early programmable DSPs, and exploratory work in voiceband data modem techniques and simultaneous voice and data transmission.