Digital Signal Processing: A Computer Science Perspective
Jonathan Y. Stein
Copyright © 2000 John Wiley & Sons, Inc.
Print ISBN 0-471-29546-9 Online ISBN 0-471-20059-X

17

Digital Signal Processors

Until now we have assumed that all the computation necessary for DSP applications could be performed either using pencil and paper or by a general-purpose computer. Obviously, those that can be handled by human calculation are either very simplistic or at least very low rate. It might surprise the uninitiated that general-purpose computers suffer from the same limitations. Being 'general-purpose', a conventional central processing unit (CPU) is not optimized for DSP-style 'number crunching', since much of its time is devoted to branching, disk access, string manipulation, etc. In addition, even if a computer *is* fast enough to perform all the required computation in time, it may not be able to *guarantee* doing so.

In the late 1970s, special-purpose processors optimized for DSP applications were first developed, and such processors are still multiplying today (pun definitely intended). Although correctly termed '**D**igital **S**ignal **P**rocessors', we will somewhat redundantly call them 'DSP processors', or simply DSPs. There are small, low-power, inexpensive, relatively weak DSPs targeted at mass-produced consumer goods such as toys and cars. More capable fixed point processors are required for cellular phones, digital answering machines, and modems. The strongest, often floating point, DSPs are used for image and video processing, and server applications.

DSP processors are characterized by having at least some of the following special features: DSP-specific instructions (most notably the MAC), special address registers, zero-overhead loops, multiple memory buses and banks, instruction pipelines, fast interrupt servicing (fast context switch), specialized ports for input and output, and special addressing modes (e.g., bit reversal).

There are also many non-DSP processors of interest to the DSP implementor. There are convolution processors and FFT processors devoted to these tasks alone. There are systolic arrays, vector and superscalar processors, RISC processors for embedded applications, general-purpose processors with multimedia extensions, CORDIC processors, and many more varieties.

DSP 'cores' are available that can be integrated on a single chip with other elements such as CPUs, communications processors, and IO devices. Although beyond the scope of our present treatment the reader would be well advised to learn the basic principles of these alternative architectures.

In this chapter we will study the DSP processor and how it is optimized for DSP applications. We will discuss general principles, without considering any specific DSP processor, family of processors, or manufacturer. The first subject is the MAC operation, and how DSPs can perform it in a single clock cycle. In order to understand this feat we need to study memory architectures and pipelines. We then consider interrupts, ports, and the issue of numerical representation. Finally, we present a simple, yet typical example of a DSP program. The last two sections deal with the practicalities of industrial DSP programming.

17.1 Multiply-and-Accumulate (MAC)

DSP algorithms tend to be number-crunching intensive, with computational demands that may exceed the capabilities of a general-purpose CPU. DSP processors can be much faster for specific tasks, due to arithmetic instruction sets specifically tailored to DSP needs. The most important special-purpose construct is the MAC instruction; accelerating this instruction significantly reduces the time required for computations common in DSP.

Convolutions, vector inner products, correlations, difference equations, Fourier transforms, and many other computations prevalent in DSP all share the basic repeated MAC computation.

```
loop
    update $j$, update $k$
    $a \leftarrow a + x_j y_k$
```

For inner products, correlations, and symmetric or coefficient-reversed FIR filters the updating of indices j and k both involve incrementation; for convolutions one index is incremented while the other is decremented.

First consider the outside of the loop. When a general-purpose CPU executes a fixed-length loop such as

```
for $i \leftarrow 1$ to $N$
    statements
```

there is a lot of overhead involved. First a register must be provided to store the loop index i, and it must be properly initialized. After each execution of the calculation the loop index register must be incremented, and checked for termination. Of course if there are not enough registers the loop index must be retrieved from memory, incremented and checked, and then stored back to memory. Except for the last iteration, a 'branch' or 'jump' instruction must be performed to return execution to the top of the loop.

DSP processors provide a zero-overhead hardware mechanism (often called `repeat` or `do`) that can repeat an instruction or number of instructions a prespecified number of times. Due to hardware support for this repeat instruction no clocks are wasted on branching or incrementing and checking the loop index. The maximum number of iterations is always limited (64K is common, although some processors have low limits such as 128) and many processors limit the number of instructions in the loop (1, 16), but these limitations fall into the envelope of common DSP operations. Some processors allow loop nesting (since the FFT requires 3 loops, this is a common limit), while for others only the innermost loop can be zero overhead.

Now let's concentrate on the computations inside the loop. How would a general-purpose CPU carry out the desired computation? We assume that x and y are stored as arrays in memory, so that x_j is stored j locations after x_0, and similarly for y_k. Furthermore, we assume that the CPU has at least two pointer registers (that we call `j` and `k`) that can be directly updated (incremented or decremented) and used to retrieve data from memory. Finally, we assume the CPU has at least two arithmetic (floating point or fixed point) registers (`x` and `y`) that can be used as operands of arithmetic operations, a double-length register (`z`) that can receive a product, and an accumulator (`a`) for summing up values.

Assuming that the loop has been set up (i.e., the counter loaded, the base pointers for x_j and y_k set, and the automatic updating of these pointers programmed in), the sequence of operations for computation of the contents of the loop on a general-purpose CPU will look something like this.

```
update pointer to $x_j$
update pointer to $y_k$
load $x_j$ into register x
load $y_k$ into register y
fetch operation (multiply)
decode operation (multiply)
multiply x by y storing the result in register z
fetch operation (add)
decode operation (add)
add register z to accumulator a
```

We see that even assuming each of the above lines takes the same amount of time (which is dubious for the multiplication), the computation requires about 10 instruction times to complete. Of course different CPUs will have slightly different instruction sets and complements of registers, but similar principles hold for all CPUs.

A major distinction between a general-purpose CPU and a DSP is that the latter can perform a MAC in a single instruction time. Indeed this feature is of such importance that many use it as the definition of a DSP. The main purpose of this chapter is explain how this miracle is accomplished. In particular it is not enough to simply add a MAC instruction to the set of opcodes; such an 'MAC-augmented CPU' would still have to perform the following steps

```
update pointer to $x_j$
update pointer to $y_k$
load $x_j$ into register x
load $y_k$ into register y
fetch operation (MAC)
decode operation (MAC)
MAC a $\leftarrow$ x * y
```

for a total of seven instruction times. We have managed to save a few clocks but are still far from our goal. Were the simple addition of a MAC instruction all a DSP processor had to offer, it would probably not be worth devoting precious silicon real-estate to the special MAC hardware. In order to build a DSP we need more imagination than this.

The first step in building a true DSP is to note that the pointers to x_j and y_k are independent and thus their updating can be performed in parallel. To implement this we need new hardware; we need to add two address updating units to the hardware complement of our hypothetical DSP processor. Using the symbol || to signify two operations that are performed in parallel, the MAC now looks like this:

```
update pointer to $x_j$ || update pointer to $y_k$
load $x_j$ into register x
load $y_k$ into register y
fetch operation (MAC)
decode operation (MAC)
MAC a $\leftarrow$ x * y
```

We have obviously saved at least the time of one instruction, since the x_j and y_k pointers are now updated simultaneously, but even though we no longer require use of the CPU's own adder it does not seem possible to further exploit this in order to reduce overall execution time. It is obvious that we cannot proceed to load values into the x and y registers until the pointers are ready, and we cannot perform the MAC until the registers are loaded. The next steps in optimizing our DSP call for more radical change.

EXERCISES

17.1.1 For the CPU it would be clearer to have j and k stored in fixed point registers and to retrieve x_j by adding j to the address of x_0. Why didn't we do this?

17.1.2 Explain in more detail why it is difficult for two buses to access the same memory circuits.

17.1.3 Many DSP processors have on-chip ROM or RAM memory. Why?

17.1.4 Many CPU architectures use *memory caching* to keep critical data quickly accessible. Discuss the advantages and disadvantages for DSP processors.

17.1.5 A processor used in personal computers has a set of instructions widely advertised as being designed for multimedia applications. What instructions are included in this set? Can this processor be considered a DSP?

17.1.6 Why does the zero-overhead loop only support loops with a prespecified number of iterations (`for` loops)? What about `while` (*condition*) loops?

17.2 Memory Architecture

A useful addition to the list of capabilities of our DSP processor would be to allow x_j and y_k to be simultaneously read from memory into the appropriate registers. Since x_j and y_k are completely independent there is no fundamental impediment to their concurrent transfer; the problem is that while one value is being sent over the 'data bus' the other must wait. The solution is to provide two data buses, enabling the two values to be read from memory simultaneously. This leaves us with a small technical hitch; it is problematic for two buses to connect to the same memory circuits. The difficulty is most obvious when one bus wishes to write and the other to read from precisely the same memory location, but even accessing nearby locations can be technically demanding. This problem can be solved by using so-called 'dual port memories', but these are expensive and slow.

The solution here is to leave the usual model of a single linear memory, and to define multiple *memory banks*. Different buses service different memory banks, and placing the x_j and y_k arrays in separate banks allows their simultaneous transfer to the appropriate registers. The existence of more than one memory area for data is a radical departure from the memory architecture of a standard CPU.

```
update pointer to $x_j$ || update pointer to $y_k$
load $x_j$ into register x || load $y_k$ into register y
fetch operation (MAC)
decode operation (MAC)
MAC a $\leftarrow$ x * y
```

The next step in improving our DSP is to take care of the `fetch` and `decode` steps. Before explaining how to economize on these instructions we should first explain more fully what these steps do. In modern CPUs and DSPs instructions are stored sequentially in memory as opcodes, which are binary entities that uniquely define the operation the processor is to perform. These opcodes typically contain a group of bits that define the operation itself (e.g., multiply or branch), individual bit parameters that modify the meaning of the instruction (multiply *immediate* or branch *relative*), and possibly bits representing numeric fields (multiply immediate by *2* or branch relative forward by *2*). Before the requested function can be performed these opcodes must first be retrieved from memory and decoded, operations that typically take a clock cycle each.

We see that a nonnegligible portion of the time it takes to execute an instruction is actually devoted to retrieving and decoding it. In order to reduce the time spent on each instruction we must find a way of reducing this overhead. Standard CPUs use 'program caches' for this purpose. A program cache is high speed memory inside the CPU into which program instructions are automatically placed. When a program instruction is required that has already been fetched and decoded, it can be taken from the program cache rather than refetched and redecoded. This tends to significantly speed up the execution of loops. Program caches are typically rather small and can only remember the last few instructions; so loops containing a large number of instructions may not benefit from this tactic. Similarly CPUs may have 'data caches' where the last few memory locations referenced are mirrored, and redundant data loads avoided.

Caches are usually avoided in DSPs because caching complicates the calculation of the time required for a program to execute. In a CPU with caching a set of instructions requires different amounts of run-time depending on the state of the caches when it commences. DSPs are designed for real-time use where the prediction of exact timing may be critical. So DSPs must use a different trick to save time on instruction fetches.

Why can't we perform a fetch one step before it is needed (in our case during the two register loads)? Once again the fundamental restriction is that we can't fetch instructions from memory at the same time that data is being transferred to or from memory; and the solution is, once again, to use separate buses and memory banks. These memory banks are called *program memory* and *data memory* respectively.

Standard computers use the same memory space for program code and data; in fact there is no clear distinction between the two. In principle the same memory location may be used as an instruction and later as a piece of data. There may even be self-modifying code that writes data to memory and later executes it as code. This architecture originated in the team that built one of the first digital computers, the 18,000-vacuum-tube ENIAC (Electronic Numerical Integrator and Computer) designed in the early forties at the University of Pennsylvania. The main designers of this machine were J.W. Mauchly and J. Presper Eckert Jr. and they relied on earlier work by J.V. Atanasoff. However, the concept of a single memory for program and data is named after John von Neumann, the Hungarian-born German-American mathematician-physicist, due to his 1945 memo and 1946 report summarizing the findings of the ENIAC team regarding storing instructions in binary form. The single memory idea intrigued von Neumann because of his interest in artificial intelligence and self-modifying learning programs.

Slightly before the ENIAC, the Mark I computer was built by a Harvard team headed by Howard Aiken. This machine was electromechanical and was programmed via paper tape, but the later Mark II and Mark III machines were purely electrical and used magnetic memory. Grace Hopper coined the term 'bug' when a moth entered one of the Harvard computers and caused an unexpected failure. In these machines the program memory was completely separate from data memory. Most DSPs today abide by this *Harvard architecture* in order to be able to overlap instruction fetches with data transfers. Although von Neumann's name is justly linked with major contributions in many areas of mathematics, physics, and the development of computers, crediting him with inventing the 'von Neumann architecture' is not truly warranted, and it would be better to call it the 'Pennsylvania architecture'. Aiken, whose name is largely forgotten, is justly the father of the two-bus architecture that posterity named after his institution. No one said that posterity is fair.

In the Harvard architecture, program and data occupy different address spaces, so that address A in program memory is completely distinct from address A in data memory. These two memory spaces are connected to the processor using separate buses, and may even have different access speeds and bit widths. With separate buses we can perform the fetch in parallel with data transfers, and no longer need to waste a clock. We will explain the precise mechanism for overlapping these operations in the next section, for now we will simply ignore the instruction-related operations. Our MAC now requires only three instruction times.

```
update pointer to $x_j$ || update pointer to $y_k$
load $x_j$ into register x || load $y_k$ into register y
MAC a $\leftarrow$ x * y
```

We seem to be stuck once again. We still can't load x_j and y_k before the pointers are updated, or perform the MAC before these loads complete. In the next section we take the step that finally enables the single clock MAC.

EXERCISES

17.2.1 A pure Harvard architecture does not allow any direct connection between program and data memories, while the *modified Harvard* architecture contains copy commands between the memories. Why are these commands useful? Does the existence of these commands have any drawbacks?

17.2.2 DSPs often have many different types of memory, including ROM, on-chip RAM, several banks of data RAM, and program memory. Explain the function of each of these and demonstrate how these would be used in a real-time FIR filter program.

17.2.3 FIR and IIR filters require a fast MAC instruction, while the FFT needs the butterfly

$$\begin{aligned} x &\leftarrow x + Wy \\ y &\leftarrow x - Wy \end{aligned}$$

where x and y are complex numbers and W a complex root of unity. Should we add the butterfly as a basic operation similar to the MAC?

17.2.4 There are two styles of DSP assembly language syntax. The opcode-mnemonic style uses commands such as `MPY A0, A1, A2`, while the programming style looks more like a conventional high-level language $A0 = A1 * A2$. Research how the MAC instruction with parallel retrieval and address update is coded in both these styles. Which notation is better? Take into account both algorithmic transparency and the need to assist the programmer in understanding the hardware and its limitations.

17.3 Pipelines

In the previous sections we saw that the secret to a DSP processor's speed is not only special instructions, but the exploitation of parallelism. Address registers are updated in parallel, memory retrievals are performed in parallel, and program instructions are fetched in parallel with execution of previous instructions. The natural extension is to allow parallel execution of any operations that logically *can* be performed in parallel.

update 1	update 2	update 3	update 4	update 5		
	load 1	load 2	load 3	load 4	load 5	
		MAC 1	MAC 2	MAC 3	MAC 4	MAC 5

Figure 17.1: The pipelining of a MAC calculation. Time runs from left to right, while height corresponds to distinct hardware units, 'update' meaning the updating of the x_j and y_k pointers, 'load' the loading into x and y, and 'MAC' the actual computation. At the left there are three cycles during which the pipeline is filling, while at the right there are a further three cycles while the pipeline is emptying. The result is available seven cycles after the first update.

The three steps of the three-clock MAC we obtained in the previous section use different processor capabilities, and so should be allowed to operate simultaneously. The problem is the dependence of each step on the completion of the previous one, but this can be sidestepped by using a *pipeline* to overlap these operations. The operation of the pipeline is clarified in Figure 17.1. In this figure 'update 1' refers to the first updating of the pointers to x_j and y_k; 'load 1' to the first loading of x_j and y_k into registers x and y; and 'MAC 1' means the first multiplication. As can be seen, the first load takes place only after the first update is complete, and the MAC only after the loads. However, we do not wait for the MAC to complete before updating the pointers; rather we immediately start the second update after the first pointers are handed over to the loading process. Similarly, the second load takes place in parallel with the first MAC, so that the second MAC can commence as soon as the first is completed. In this way the MACs are performed one after the other without waiting, and once the pipeline is filled each MAC requires only one instruction cycle. Of course there *is* overhead due to the pipeline having to fill up at the beginning of the process and empty out at the end, but for large enough loops this overhead is negligible. Thus the pipeline allows a DSP to perform one MAC per instruction clock *on the average.*

Pipelines can be exploited for other purposes as well. The simplest general-purpose CPU must wait for one basic operation (e.g., fetch, decode, register arithmetic) to complete before embarking on the next; DSPs exploit parallelism even at the subinstruction level. How can the different primitive operations that make up a single instruction be performed in parallel? They can't; but the primitive operations that comprise successive instructions *can*.

Until now we have been counting 'instructions' and have not clarified the connection between 'instruction times' and 'clock cycles'. All processors are fed a clock signal that determines their speed of operation. Many processors are available in several versions differing only in the maximum clock speed at which they are guaranteed to function. While a CPU processor is always specified by its clock frequency (e.g., a 400 MHz CPU), DSP processors are usually designated by clock interval (e.g., a 25 nanosecond DSP).

Even when writing low-level assembly language that translates directly to native opcodes, a line of code does not directly correspond to a clock interval, because the processor has to carry out many operations other than the arithmetic functions themselves. To see how a CPU really works at the level of individual clock cycles, consider an instruction that adds a value in memory to a register, leaving the result in the same register. At the level of individual clock cycles the following operations might take place.

```
fetch instruction
decode instruction
retrieve value from memory
perform addition
```

We see that a total of four clock cycles is required for this single addition, and our 'instruction time' is actually four 'clock cycles'. There might be additional subinstruction operations as well, for instance, transfer of a value from the register to memory. Fixed point DSP processors may include an optional postarithmetic scaling (shift) operation, while for floating point there is usually a postarithmetic normalization stage that ensures the number is properly represented.

Using a subinstruction pipeline we needn't count four clock cycles per instruction. While we are performing the arithmetic portion of an instruction, we can already be decoding the next instruction, and fetching the one after that! The number of overlapable operations of which an instruction is comprised is known as the *depth* of the pipeline. The minimum depth is three (fetch, decode, execute), typical values are four or five, but by dividing the arithmetic operation into stages the maximum depth may be larger. Recent DSP processors have pipeline depths as high as 11.

fetch 1	fetch 2	fetch 3	fetch 4	fetch 5			
	decode 1	decode 2	decode 3	decode 4	decode 5		
		get 1	get 2	get 3	get 4	get 5	
			add 1	add 2	add 3	add 4	add 5

Figure 17.2: The operation of a depth-four pipeline. Time runs from left to right, while height corresponds to distinct hardware units. At the left there are three cycles during which the pipeline is filling, while at the right there are three cycles while the pipeline is emptying. The complete sum is available eight cycles after the first fetch.

As an example, consider a depth-four pipeline that consists of fetch, decode, load data from memory, and an arithmetic operation, e.g., an addition. Figure 17.2 depicts the state of a depth-four pipeline during all the stages of a loop adding five numbers. Without pipelining the summation would take $5 * 4 = 20$ cycles, while here it requires only eight cycles. Of course the pipeline is only full for two cycles, and were we to sum 100 values the pipelined version would take only 103 cycles. Asymptotically we require only a single cycle per instruction.

The preceding discussion was based on the assumption that we know what the next instruction will be. When a branch instruction is encountered, the processor only realizes that a branch is required after the decode operation, at which point the next instruction is already being fetched. Even more problematic are conditional branches, for which we only know which instruction is next after a computation has been performed. Meanwhile the pipeline is being filled with erroneous data. Thus pipelining is useful mainly when there are few (if any) branches. This is the case for many DSP algorithms, while possibly unjustified for most general-purpose programming.

As discussed above, many processor instructions only return results after a number of clocks. Attempting to retrieve a result before it is ready is a common mistake in DSP programming, and is handled differently by different processors. Some DSPs assist the programmer by locking until the result is ready, automatically inserting wait states. Others provide no locking and it is entirely the programmer's responsibility to wait the correct number of cycles. In such cases the NOP (no operation) opcode is often inserted to simply waste time until the required value is ready. In either case part of the art of DSP programming is the rearranging of operations in order to perform useful computation rather than waiting with a NOP.

EXERCISES

17.3.1 Why do many processors limit the number of instructions in a repeat loop?

17.3.2 What happens to the pipeline at the end of a loop? When a branch is taken?

17.3.3 There are two styles of DSP assembly language syntax regarding the pipeline. One emphasizes time by listing on one line all operations to be carried out simultaneously, while the other stresses data that is logical related. Consider a statement of the first type

$$A1 = A1 + A2; A2 = A3 * A4; A3 = *R1++; A4 = *R2++$$

where $A1, A2, A3, A4$ are accumulators and $R1, R2$ pointer registers. Explain the relationship between the contents of the indicated registers. Next consider a statement of the second type

$$A0 = A0 + (*R1++ * *R2++)$$

and explain when the operations are carried out.

17.3.4 It is often said that when the pipeline is not kept filled, a DSP is slower than a conventional processor, due to having to fill up and empty out the pipeline. Is this a fair statement?

17.3.5 Your DSP processor has 8 registers `R1, R2, R3, R4, R5, R6, R7, R8`, and the following operations

- load register from memory: `Rn ← location`
- store register to memory: `location ← Rn`
- single cycle no operation: `NOP`
- negate: `Rn ← − Rn` [1 cycle latency]
- add: `Rn ← Ra + Rb` [2 cycle latency]
- subtract: `Rn ← Ra − Rb` [2 cycle latency]
- multiply: `Rn ← Ra · Rb` [3 cycle latency]
- MAC: `Rn ← Rn + Ra · Rb` [4 cycle latency]

where the latencies disclose the number of cycles until the result is ready to be stored to memory. For example,

```
    R1  ←  R1 + R2 · R3
answer  ←  R1
```

does not have the desired effect of saving the MAC in answer, unless four NOP operations are interposed. Show how to efficiently multiply two complex numbers. (Hint: First code operations with enough `NOP` operations, and then interchange order to reduce the number of `NOP`s.)

17.4 Interrupts, Ports

When a processor stops what it has been doing and starts doing something else, we have a *context switch*. The name arises from the need to change the run-time context (e.g, the pointer to the next instruction, the contents of the registers). For example, the operating system of a time-sharing computer system must continually force the processor to jump between different tasks, performing numerous context switches per second. Context switches can be initiated by outside events as well (e.g., keyboard presses, mouse clicks, arrival of signals). In any case the processor must be able to later return to the original task and continue as if nothing had happened.

Were the software responsible for initiating all externally driven context switches, it would need to incessantly poll all the possible sources of such requests to see whether servicing is required. This would certainly be a waste of resources. All processors provide a hardware mechanism called the *interrupt*. An interrupt forces a context switch to a predefined routine called the *interrupt handler* for the event in question. The concept of an interrupt is so useful that many processors provide a 'software interrupt' (sometimes called a trap) by which the software itself can instigate a context switch.

One of the major differences between DSPs and other types of CPUs is the speed of the context switch. A CPU may have a latency of dozens of cycles to perform a context switch, while DSPs always have the ability to perform a low-latency (perhaps even zero-overhead) interrupt.

Why does a DSP need a fast context switch? The most important reason is the need to capture interrupts from incoming signal values, either immediately processing them or at least storing them in a buffer for later processing. For the latter case this signal value capture often occurs at a high rate and should only minimally interfere with the processing. For the former case delay in retrieving an incoming signal may be totally unacceptable.

Why do CPU context switches take so many clock cycles? Upon restoration of context the processor is required to be in precisely the same state it would have been had the context switch not occurred. For this to happen many state variables and registers need to be stored for the context being switched out, and restored for the context being switched in. The DSP fast interrupt is usually accomplished by saving only a small portion of the context, and having hardware assistance for this procedure. Thus if the context switch is for the sole purpose of storing an incoming sample to memory, the interrupt handler can either not modify unstored registers, or can be coded to manually restore them to their previous state.

All that is left is to explain how signal values are input to and output from the DSP. This is done by *ports*, of which there are several varieties. Serial ports are typically used for low-rate signals. The input signal's bits are delivered to the DSP one at a time and deposited in an internal shift register, and outputs are similarly shifted out of the DSP one bit per clock. Thus when a 16-bit A/D is connected to a serial port it will send the sample as 16 bits, along with a bit clock signal telling the DSP when each bit is ready. The bits may be sent MSB first or LSB first depending on the A/D and DSP involved. These bits are transferred to the DSP's internal serial port shift register. Each time the A/D signals that a bit is ready, the DSP serial port shift register shifts over one bit and receives the new one. Once all 16 bits are input the A/D will assert an interrupt requesting the DSP to store the sample presently in the shift register to memory.

Parallel ports are faster than serial ports but require more pins on the DSP chip itself. Parallel ports typically transfer eight or sixteen bits at a time. In order to further speed up data transfer **D**irect **M**emory **A**ccess (DMA) channels are provided that can transfer whole blocks of data to or from the DSP memory without interfering with the processing. Typically once a DMA transfer is initiated, only a single interrupt is required at the end to signal that the transfer is complete.

Finally, communications ports are provided on those DSPs that may be interconnected with other DSPs. By constructing arrays of DSPs processing tasks may be divided up between processors and such platforms may attain processing power far exceeding that available from a single processor.

EXERCISES

17.4.1 How does the CPU know which interrupt handler to call?

17.4.2 Some DSPs have 'internal peripherals' that can generate interrupts. What can these be used for?

17.4.3 What happens when an interrupt interrupts an interrupt?

17.4.4 When a DSP is on a processing board inside a host computer there may be a method of input and output other than ports—shared memory. Discuss the pros and cons of shared memory vs. ports.

17.5 Fixed and Floating Point

The first generation of DSP processors were integer-only devices, and even today such fixed point DSPs flourish due to their low cost. This seems paradoxical considering that DSP tasks are number-crunching intensive. You probably wouldn't consider doing serious numeric tasks on a conventional CPU that is not equipped with floating point hardware. Yet the realities of speed, size, power consumption, and price have compelled these inconvenient devices on the DSP community, which has had to develop rather intricate numeric methods in order to use them. Today there are floating point DSPs, but these still tend to be much more expensive, more power hungry, and physically larger than their fixed point counterparts. Thus applications requiring embedding a DSP into a small package, or where power is limited, or price considerations paramount, still typically utilize fixed point DSP devices. Fixed point DSPs are also a good match for A/D and D/A devices, which are typically unsigned or two's-complement integer devices.

The price to be paid for the use of fixed point DSPs is extended development time. After the required algorithms have been simulated on computers with floating point capabilities, floating point operations must then be carefully converted to integer ones. This involves much more than simple rounding. Due to the limited dynamic range of fixed point numbers, rescaling must be performed at various points, and special underflow and overflow handling must be provided. The exact placement of the rescalings must be carefully chosen in order to ensure the maximal retention of signal vs. quantization noise, and often extensive simulation is required to determine the optimal placement. In addition, the precise details of the processor's arithmetic may need to be taken into account, especially when interoperability with other systems is required. For example, standard speech compression algorithms are tested by providing specified input and comparing the output bit stream to that specified in the standard. The output must be exact to the bit, even though the processor may compute using any number of bits. Such bit-exact implementations may utilize a large fraction of the processor's MIPS just to coerce the fixed point arithmetic to conform to that of the standard.

The most common fixed point representation is 16-bit two's complement, although longer registers (e.g., 24- or 32-bit) also exist. In fixed point DSPs this structure must accommodate both integers and real numbers; to represent the latter we multiply by some large number and round. For example, if we are only interested in real numbers between -1.0 and $+1.0$ we multiply by 2^{15} and think of the two's-complement number as a binary fraction.

When two 16-bit integers are added, the sum can require 17 bits; when multiplied, the product can require 32 bits. Floating point hardware takes care of this bit growth by automatically discarding the least significant bits, but in fixed point arithmetic we must explicitly handle the increase in precision. CPUs handle addition by assuming that the resultant usually does fit into 16 bits; if there is an overflow a flag is set or an exception is triggered. Products are conventionally stored in two registers, and the user must decide what to do next based on the values in the registers. These strategies are not optimal for DSP since they require extra operations for testing flags or discarding bits, operations that would break the pipeline.

Fixed point DSPs use one of several strategies for handling the growth of bits without wasting cycles. The best strategy is for the adder of the MAC instruction to use an accumulator that is longer than the largest possible product. For example, if the largest product is 32 bits the accumulator could have 40 bits, the extra bits allowing eight MACs to be performed without any possibility of overflow. At the end of the loop a single check and possible discard can be performed. The second strategy is to provide an optional scaling operation as part of the MAC instruction itself. This is basically a right shift of the product before the addition, and is built into the pipeline. The least satisfactory way out of the problem, but still better than nothing, is the use of 'saturation arithmetic'. In this case a hard limiter is used whenever an overflow occurs, the result being replaced by the largest representable number of the appropriate sign. Although this is definitely incorrect, the error introduced is smaller than that caused by straight overflow.

Other than these surmountable arithmetic problems, there are other possible complications that must be taken into account when using a fixed point processor. As discussed in Section 15.5, after designing a digital filter its coefficients should not simply be rounded; rather the best integer coefficients should be determined using an optimization procedure. Stable IIR filters may become unstable after quantization, due to poles too close to the unit circle. Adaptive filters are especially sensitive to quantization. When bits are discarded, overflows occur, or limiting takes place, the signal processing system ceases to be linear, and therefore cycles and chaotic behavior become possible (see Section 5.5).

Floating point DSPs avoid many of the above problems. Floating point numbers consist of a mantissa and an exponent, both of which are signed integers. A recognized standard details both sizes for the mantissa and exponent and rules for the arithmetic, including how exceptions are to be handled. Not all floating point DSPs conform to this standard, but some that don't provide opcodes for conversion to the standard format.

Unlike the computing environments to which one is accustomed in off-line processing, even the newer floating point DSP processors do not usually have instructions for division, powers, square root, trigonometric functions, etc. The software libraries that accompany such processors *do* include such functions, but these general-purpose functions may be unsuitable for the applications at hand. The techniques of Chapter 16 can be used in such cases.

EXERCISES

17.5.1 Real numbers are represented as integers by multiplying by a large number and rounding. Assuming there is no overflow, how is the integer product related to the real product? How is a fixed point multiply operation from two b-bit registers to a b-bit register implemented?

17.5.2 Simulate the simple IIR filter $y_n = \alpha y_{n-1} + x_n$ $(0 \le \alpha \le 1)$ in floating point and plot the impulse response for various α. Now repeat the simulation using 8-bit integer arithmetic (1 becomes 256, $0 \le \alpha \le 256$). How do you properly simulate 8-bit arithmetic on a 32-bit processor?

17.5.3 Design a narrow band-pass FIR filter and plot its empirical frequency response. Quantize the coefficients to 16 bits, 8 bits, 4 bits, 2 bits, and finally a single bit (the coefficient's sign); for each case replot the frequency response.

17.5.4 Repeat the previous exercise for an IIR filter.

17.6 A Real-Time Filter

In this section we present a simple example of a DSP program that FIR filters input data in real-time. We assume that the filter coefficients are given, and that the number of coefficients is L.

Since this is a real-time task, every t_s seconds a new input sample x_n appears at the DSP's input port. The DSP must then compute

$$y_n = \sum_{l=0}^{L-1} h_l x_{n-l}$$

and output y_n in less than t_s seconds, before the next sample arrives. This should take only somewhat more than L processor cycles, the extra cycles being unavoidable overhead.

On a general purpose CPU the computation might look like this:

```
for l ← 1 to (L-1)
    x[l-1] ← x[l]
x[L-1] ← input

y ← 0
for l ← 0 to (L-1)
    y ← y + h[l] * x[L-1-l]
output ← y
```

We first made room for the new input and placed it in `x[L-1]`. We then computed the convolution and output the result.

There are two main problems with this computation. First we wasted a lot of time in moving the static data in order to make room for the new input. We needn't physically move data if we use a *circular buffer*, but then the indexation in the convolution loop would be more complex. Second, the use of explicit indexation is wasteful. Each time we have need `x[L-1-l]` we have to compute `L-1-l`, find the memory location, and finally retrieve the desired data. A similar set of operations has to be performed for `h[l]` before we are at last ready to multiply. A more efficient implementation uses 'pointers'; assuming we initialize `h` and `x` to point to h_0 and x_{L-1-l} respectively, we have the following simpler loop:

```
y ← 0
repeat L times
    y ← y + (*h) * (*x)
    h ← h + 1
    x ← x - 1
```

Here `*h` means the contents of the memory location to which the pointer `h` points. We can further improve this a little by initializing `y` to $h_0 x_n$ and performing one less pass through the loop.

How much time does this CPU-based program take? In the loop there is one multiplication, two additions and one subtraction, in addition to assignment statements; and the loop itself requires an additional implicit decrement and comparison operation.

Now we are ready to try doing the same filtering operation on a DSP. Figure 17.3 is a program in assembly language of an imaginary DSP. The words starting with dots (such as `.table`) are 'directives'; they direct the assembler to place the following data or code in specific memory banks. In

```
.table
H:         $h_0$
           $h_1$
             ⋮
HLAST:     $h_{L-1}$

.data
X:         (L-1) * 0
XNEW:      0

.program
START:
  if (WAIT) goto START
  *XNEW ← INPUT
  h ← HLAST
  x ← X

  y ← (*h)*(*x) || h ← h-1 || x ← x+1
  repeat (L-1) times
       y ← y + (*h)*(*x) || *(x-1) ← *x || h ← h-1 || x ← x+1

  NOP
  OUTPUT ← y
  goto START
```

Figure 17.3: A simple program to FIR filter input data in real-time.

this case the L filter coefficients $h_0 \ldots h_{L-1}$ are placed in 'table memory'; the static buffer of length L is initialized to all zeros and placed in 'data memory'; and the code resides in 'program memory'. These placements ensure that the MAC instructions will be executable in a single cycle. The names followed by colons (such as `HLAST:`) are 'labels', and are used to reference specific memory locations.

The filter coefficients are stored in the following order $h_0, h_1, \ldots h_{L-1}$ with h_0 bearing the label H and h_{L-1} labeled HLAST. The static buffer is in reversed order $x_{n-(L-1)}, \ldots x_{n-1}, x_n$ with the oldest value bearing the label X and the present input labeled XNEW.

The program code starts with the label START, and each nonempty line thereafter corresponds to a single processor cycle. The first line causes the processor to loop endlessly until a new input arrives. In a real program such a tight loop would usually be avoided, but slightly looser do-nothing loops are commonly used.

Once an input is ready it is immediately copied into the location pointed to by `XNEW`, which is the end of the static buffer. Then pointer register `h` is set to point to the end of the filter buffer (h_{L-1}) and pointer `x` is set to point to the beginning of that buffer (the oldest stored input).

Accumulator `y` is initialized to $h_{L-1}x_0$, the last term in the convolution. Note that `y` is a numeric value, *not* a pointer like `x`. The || notation refers to operations that are performed in parallel. In this line the filter buffer pointer is decremented and the static buffer pointer is incremented. These operations are carried out before they are next required.

The next line contains a 'zero-overhead loop'. This loop is only executed `L-1` times, since the last term of the convolution is already in the accumulator. The last iteration multiplies the h_0 coefficient by the new input. However, something else is happening here as well. The `*(x-1)` $\leftarrow$ `*x` being executed in parallel is a data-move that shifts the input data that has just been used one place down; by the time the entire loop has been executed the static buffer has all been shifted and is ready for the next iteration.

Once the entire convolution has been carried out we are ready to output the result. However, in some DSP processors this output operation can only take place once the pipeline has been emptied; for this reason we placed a `NOP` (no-operation) command before copying the accumulator into the output register. Finally, we jump back to the start of the program and wait for the next input to arrive.

EXERCISES

17.6.1 Taking a specific number of coefficients (e.g., `L=5`), walk through the program in Figure 17.3, noting at each line the values of the pointers, the state of the static buffer, and the algebraic value in the accumulator.

17.6.2 Code a real-time FIR filter for a DSP that does not support data-move in parallel with the MAC, but has hardware support for a circular buffer.

17.6.3 Code a real-time IIR routine similar to the FIR one given in the text. The filter should be a cascade of N second order sections, and the main loop should contain four lines and be executed N times.

17.6.4 Write a filtering program for a real DSP and run it in real-time.

17.7 DSP Programming Projects

DSP programming is just like any other programming, only more so. As in any other type of programming attention to detail is essential, but for DSP processors this may extend beyond syntax issues. For example, some processors require the programmer to ensure that the requisite number of cycles have passed before a result is used; forgetting a NOP in such a situation creates a hard-to-locate bug. For many types of programming intimate knowledge of the hardware capabilities isn't crucial, but for DSP programming exploitation of special-purpose low-level features may mean the difference between success and failure. As in any other type of programming, familiarity with the software development tools is indispensable, but for DSP processors emulation, debugging and profiling may be much more difficult and critical tasks.

In this section we present a model that you may find useful to consider when embarking on a new DSP programming project. However, whether or not you adhere to all the details of this model, remember that you must always obey the *golden rule of DSP programming*:

Always program for correctness first, efficiency second.

All too often we are driven by the need to make our algorithms faster and faster, and are tempted to do so at the expense of system stability or thorough testing. These temptations are to be avoided at all costs.

Once this is understood I suggest that the task of implementing a new system is CHILD's play. Here the word CHILD is a mnemonic for:
Collect requirements and decide on architecture
High-level design
Intermediate level, simulation and porting to platform
Low-level coding and efficiency improvement
Deliver and document
We shall devote a paragraph or two to each of these stages.

The collection stage is a critical one, all too often incompletely executed. The implementor must collect all the requirements, including the expected range of inputs, the exact output(s) required, the overall development schedule and budget, the desired end user cost, interface specifications, etc. Sometimes someone else has done the preliminary work for you and you receive a **H**ardware **R**equirements **S**pecification (HRS) and a **S**oftware **R**equirements Specification (SRS). Remember that anything missed during the collection stage will be difficult or impossible to reintroduce later on. One of the things to be decided at this stage is how the final product is to be tested, and the exact criteria for success. You should make sure the end users (or technical marketing personnel) 'sign off' on the requirement specifications and acceptance procedures.

Between the end of the collection stage and the beginning of the high-level design stage it is highly recommended to go on vacation.

The technical output of the high-level design stage will usually be a pair of documents, the **H**ardware **D**esign **D**ocument (HDD) and the **S**oftware **D**esign **D**ocument (SDD). There will also be project management literature, including various charts detailing precisely what each team member should be doing at every time, dates by which critical tasks should be completed, milestones, etc. We will focus on the SDD. The SDD explains the signal processing system, first in generality, and then increasingly in detail. The function of each subsystem is explained and its major algorithms noted. There are two ways to write an SDD. The first (and most commonly encountered) is to have done something extremely similar in the past. In this case one starts by cutting and pasting and then deleting, inserting, and modifying until the present SRS is met. The more interesting case is when something truly new is to be built. In this case a correct SDD cannot be written and the project management literature should be considered science fiction. Remember that the 'R' and 'D' in R&D are two quite different tasks, and that a true research task cannot be guaranteed to terminate on a certain date and in a certain way (the research would be unnecessary if it could).

Often simulations must be performed during the high-level design stage. For these simulations efficiency is of no concern; but development speed, ease of use and visualization ability are of the utmost importance. For this reason special development environments with graphics and possibly visual programming are commonly used. The output of these simulations, both block diagrams and performance graphs, can be pasted into the design documents. The amount of memory and processing power required for each subsystem can now be better estimated. It is best to plan on using only 50% to 75% of the available processing power (it will always turn out to require a lot more than you anticipate).

At the end of the high-level design a **D**esign **R**eview (DR) should be carried out. Here the HDD and SDD are explained and comments solicited. Invite as many relevant people as possible to the DR. Remember that mistakes in the high-level design are extremely costly to repair later on.

The intermediate-stage may be bypassed only for very small projects. Here the block diagrams developed in the high-level stage are fleshed out and a complete program is written. This is often done first in a high-level language, liberally using floating point numbers and library functions. While the high-level software design stage is often carried out by a single person, the intermediate stage is usually handed over to the full development team. Once the team starts to work, the project should be placed under revision control. Once completed and integrated the full program can be tested with test inputs and outputs to ensure passing the final acceptance procedures. Next the high-level language program is rewritten in a real-time style, using the proper block lengths, converting to fixed point if required, etc. After each major step the program behavior can be compared to that of the original to ensure correctness. Mechanisms for debugging, exception handling, maintainability, and extensibility should be built into the code.

The low-level programming commences as a straightforward port of the intermediate level program to the final platform. Once again the first code should be written to maintain correctness at the expense of efficiency. After the first port, decisions must be made as to what can remain in a high-level language and what must be coded in assembly language; where major improvements in efficiency are required; where memory usage is excessive; etc. Efficiency is increased incrementally by concentrating on areas of code where the program spends most of its time. Various debugging tools, such as simulators, emulators, debug ports, and real-time monitoring are used. Eventually a correct version that is fast enough is generated.

Delivery of a version is something that no one likes doing, but the project is not complete without it. The final version must be cleaned up and acceptance tests thoroughly run (preferably with the end user or disinterested parties present). User and programmer documentation must be completed. The former is usually written by professional publications personnel. The latter include internal documentation (the final code should be at least 25% comments), an updated SDD, and a **V**ersion **D**escription **D**ocument (VDD) that describes all limitations, unimplemented features, changes, and outstanding problems.

After delivery the boss takes the development team out to lunch, or lets everyone take an extended weekend. The following week the whole process starts over again.

17.8 DSP Development Teams

'Congratulations, you've got the job!' You heard those words just last week, but you are already reporting for work as a junior DSP engineer in the ASP (Advanced Signal Processing) division of III (Infinity Integrators Inc.). In your previous jobs you worked by yourself or with one or two other people on some really impressive DSP projects. You've programmed DSP boards in PCs, completing whole applications in a week, often writing over 500 lines of assembly language in a day. You are quite sure that this has prepared you for this new job. Of course in a big company like III with hundreds of engineers and programmers, working on multimillion dollar projects, there will be more overhead, but DSP programming is DSP programming. There is only one thing that your new boss said during your interview that you don't quite understand. Why do the programmers here only write an average of ten to twenty lines of code in a day? Are they lazy or just incompetent?

Well, you'll soon find out. Your first assignment is to understand the system you will be working on. This system is a newer version of an older one that has been operational for over five years. Your boss has given you five days to come up to speed. Sounds easy.

In your cubicle you find a stack of heavy documents. The first thing you have to learn is what a TLA is. TLA is a self referential term for **T**hree **L**etter **A**cronym, and the system you are going to work on is full of them. There are several FEUs (front end units), a BEU (back end unit), and a MPC (main processing computer) with a HIC (human interface console). You learn all this by reading parts of two documents titled HRS and SRS, that are followed by even larger ones marked HDD and SDD. These documents are so highly structured that you soon understand that it would take a full five days just to read them and follow up the cross references. There are also countless other documents that you have been told are not as important to you (yet), like the PMP (program management plan) that even specifies how much time you are allotted to learn the system (5 days).

Your second day is spent trying to figure out where DSP fits in to all this and what you are expected to do. As a shortcut you look up your tasks in the PMP. After an hour or so you have jotted down the cross references of tasks to which you have been assigned, and start looking up what they are. A lot is written about when they start and end, what resources they need, and what they hold up if not finished on time. The only explanation of exactly what you are expected to do seems to be one-line descriptions that are entirely incomprehensible. In fact, the only thing you think you fully understand is the 'handbook for the new employee' that human resources placed in your cubicle. Even that seems to have been written in the same style, but at least it is one self-contained document (except the reference to the manual for using the voicemail system).

On your third day on the job you attend your first weekly TRM (team review meeting). The main subject seems to be when the PDR (preliminary design review) and CDR (critical design review) will take place. Luckily the TRM will only last about a half hour since the boss has to go to the TLM (team leader meeting) that has to take place before the GLM (group leader meeting) headed by the director of ASP. Any questions? Somewhat timidly you speak up—why is it called the *FEU* and not the *DSP*? In all the jobs you worked on up to now the module with the DSPs was simply called the *DSP*. The boss explains patiently (while several coworkers smile) that the name has been chosen to be accurate from a system point of view. *DSP* stands for digital signal processing. Here the fact that the processing is *digital* is irrelevant to the rest of the system; indeed in an earlier version a lot of the processing was analog. The words *signal* and *processing* turn out to be incorrect from a system point of view. The purpose of this unit is acquisition of the data needed by the rest of the system, even if this data already requires a great deal of processing from raw input.

After the meeting you spend a few hours in the library. The SDD references a lot of international standards and each of these in turn references still other standards. The standards documents seem even less comprehensible than the SDD itself. You spend the next few hours by the copying machine.

On the fourth day you are invited to attend a discussion between the hardware and software guys. Not yet having learned about the subsystem in question you can't quite make out what is going on, other than the hardware guys saying it's an obvious software bug and the software guys saying its a hardware failure. You speak up asking why a simple test program can't be used to test the hardware. The hardware people explain that they had written such a program and that is precisely how they know its a software problem. The software people reply that the hardware test program was unrealistically simplistic and didn't really test this aspect of the design. They, however, have written a simulation of the hardware, and their software runs perfectly on it.

You glance at your watch. Although it's after six o'clock you think you'll spend a few more hours reading the SDD. Just this morning you had at last found out what the DSP processing elements were, and were beginning to feel more confident that it was, in principle, possible to extract information from these documents. How will you ever finish reading all this by tomorrow?

The fifth day starts auspiciously—your development system arrives. You are given a table in the lab; its one floor down from your cubicle and you learn to use the stairs rather than wait for the elevator. The entire morning is spent on unpacking, reading the minimal amount of documentation, hooking up all the cables, and configuring the software. A coworker helps you out, mostly in order to see the improvements in the new version you have received. You go out to lunch with your co-workers, and the entire time is spent talking about work. Once back you return to your cubicle only to find that someone has 'borrowed' your copy of the SDD. You return to the lab only to find that several power cables are missing as well. It's after three and you start to panic. Are your coworkers playing some kind of initiation prank or does this kind of thing happen all the time?

Bibliographical Notes

Jonathan Allen from MIT gave two early, but still relevant, overviews of the basic architecture of digital signal processors [3, 4]. More modern reviews are [142, 143, 141]. In particular [61] can be used as a crash course in DSP processors: what defines a DSP, how DSPs differ from CPUs, and how they differ one from another.

Manuals supplied by the various processor manufacturers are the best source for information on DSP architecture and how to best exploit it. Usually each processor has a *Processor User's Guide* that details its architecture and instruction set; an *Assembly Language Reference* with explanations of its programming environment; and an *Applications Library Manual* with sample code and library routines for FFTs, FIR and IIR filters, etc. For popular processors many useful DSP functions and full applications will be available for licensing or in the public domain. The annual EDN DSP directory [148] is a treasure-trove of information regarding all extant DSPs.

[140] is devoted entirely to fundamentals of DSP processors, and its authors also publish an in-depth study and comparison of available DSPs. Readers considering implementing DSP functions in VLSI should consult [154].