Programmable DSP Architectures: Part II

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. Part I, which appeared in the previous issue of *ASSP Magazine*, discussed arithmetic and memory organizations. This part discusses pipelining. Three distinct techniques are used for dealing with pipelining, interlocking, time-stationary coding, and data-stationary coding. These techniques are studied in light of the performance benefit and the impact on the user. As in part I, representative DSPs from AT&T, Motorola, and Texas Instruments are used to illustrate the ideas. It is not the intent of the author to catalog available DSPs nor their features, nor to endorse particular manufacturers. It is the intent to compare different solutions to the same problems. The paper concludes with a discussion of trends and some bold predictions for the future.

1. INTRODUCTION

In Part I of this paper, which appeared in the previous issue of *ASSP Magazine,* we found that programmable DSPs use multiple memory banks in order to get adequate memory bandwidth. Several variations on the basic Harvard architecture were described, but they all have one feature in common; an instruction is fetched at the same time that operands for a previously fetched instruction are being fetched. This can be viewed as a form of pipelining, where the instruction fetch, operand fetch, and instruction execution form a three-stage pipeline. Close examination, however, shows that most DSPs are not so simple. This paper examines the timing of instructions in DSPs, revealing intricacies and subtleties that easily evade the casual observer. In addition, trends are discussed, complete with predictions for the future.

Most of the examples used in this paper come from one of the DSPs in Table 1, reproduced from Part I. Other important DSPs are listed in Table 2 of Part I. Most of the architectural features of the DSPs in Table 2 of Part I are also represented in Table 1, so their explicit inclusion in this paper would be redundant. The choice of DSPs in Table 1 stems primarily from the familiarity of the author with the devices, and should not be construed as an endorsement. 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.

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.

2. PIPELINING

A typical programmable DSP has instructions that will fetch two operands from memory, multiply them, add them to an accumulator, write the result to memory, and post-increment three address registers. It is obvious that if all these operations had to be done sequentially within one instruction cycle, the instruction cycle times would be much longer than they are. Fast execution is accomplished using pipelining.

Pipelining effectively speeds up the computation, but it can have a serious impact on programmability. There are three fundamentally different techniques for dealing with pipelining in a programmable processor: interlocking, time-stationary coding, and data-stationary coding. The TI DSPs primarily use interlocking, the Motorola DSPs and the AT&T DSP16/16A use time-stationary coding, and the AT&T DSP32/32C use data-stationary coding. As with most taxonomies, the boundaries between the categories are not rigid, and most DSPs have some of the flavor of all three.

2.1. *Interlocking.*

One philosophy is that the programmer should not be bothered with the internal timing or parallelism of the architecture. Programs should be written in an assembly language in which the programmer can assume that every action specified in one instruction completes before the next instruction begins. Furthermore, each instruction should completely specify its operand memory locations and the operation performed. The processor may be pipelined, but it should not act as if it were so.

A simple model for the pipelining of a programmable processor divides the instruction execution into instruction fetch, decode, operand fetch, and execute stages, as shown in Figure 1. In the figure, the cross-hatched boxes indicate *latches,* which latch signals once per instruction cycle. The instruction fetch occurs at the same time that the previous instruction is being decoded, and at the same time that the operands for the instruction before that are being fetched. The trick is to overlap instructions in this way and still gave the impression that every instruction finishes before the next instruction begins.

0740-7467/89/0100-0004$1.00 © 1989 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. The numbers marked with "?" are projected. 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 version. 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 or 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	97.5	16/32	
	TMS320C30	88	60	24/32	32/40

Example 1

The TMS320C30 conforms well with the pipeline model of Figure 1. Consider the parallel multiply and add instruction (see Example 18 of Part I for a program using this instruction). Its timing is shown in Figure 2 using a *reservation table*. Hardware resources are listed on the left and time increases to the right. First the instruction is fetched. We assume internal memory is used, in which case only half an instruction cycle is required for the fetch, but time is available for an external access, which would require a full instruction cycle. Then two parallel address arithmetic units are used to compute the operand addresses. The TMS320C30 provides indexed addressing, in which an index must be added to the address *prior* the fetch, so computing operand addresses is non-trivial. After this, the operands are fetched. They may be fetched from two different memories, as shown, or from the same memory. The DDATA bus is used to transfer the operands to the arithmetic units. Finally, the multiply and add proceed in parallel, consuming a complete instruction cycle. A similar instruction can be fetched every cycle without any conflict for resources.

Although the execution of the instruction is scattered over four cycles, it is important the programmer be unaware of this. A store instruction that follows the parallel multiply and add must be able to store either the result of the multiply or the add (or both) without any delay.

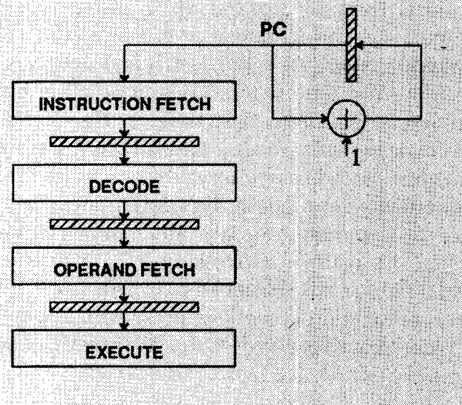


Figure 1. A common model for the pipelining of a programmable processor divides the instruction execution into instruction fetch, decode, operand fetch, and execute stages.

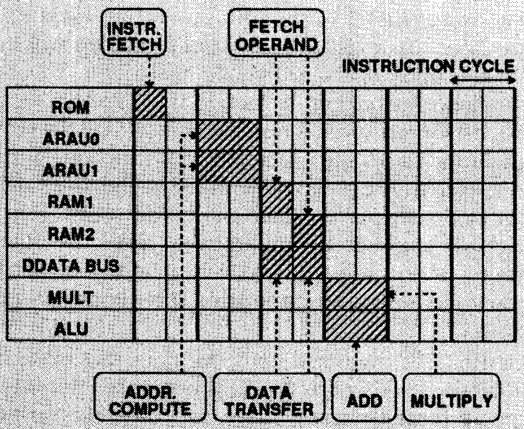


Figure 2. The author's estimate of the timing of a parallel multiply and add instruction on the TMS320C30 is shown using a reservation table. Hardware resources are listed on the left and time increases to the right. All memory accesses are assumed to be internal and suffer no conflicts with neighboring instructions. The instruction cycle time is 60ns.

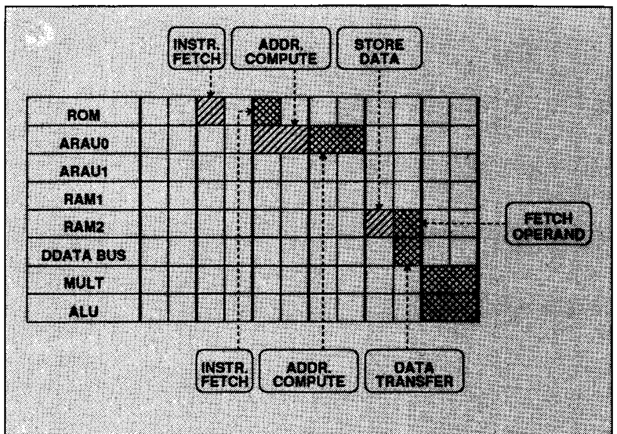


Figure 3. The author's estimate of the timing of a store instruction (cross-hatched, explained on top) and a following arithmetic instruction (doubly cross-hatched, explained on the bottom). It is important that the store be able to store data computed by the instruction of Figure 2, and that the arithmetic instruction (cross-hatched) be able to read the data stored.

The earliest possible time that such a store could occur is in the fifth instruction cycle of Figure 2. In Figure 3, a store to RAM2 is shown occurring at that time. Also shown, cross-hatched, is a third instruction, fetched in the third instruction cycle, that reads from RAM2. In order to hide the pipelining from the programmer, it is essential that this instruction be able to read the data just stored. With the timing shown this occurs.

There are many possible variations on the instructions shown in Figure 2 and Figure 3. Suppose, for example, that the arithmetic instruction in Figure 3 required two operands from RAM2. If the immediately preceding instruction (the store) were not using RAM2, there would be no problem, but as it is, there would be contention for RAM2 in the fifth instruction cycle of Figure 3. In this event, the control hardware will delay the execution of the arithmetic instruction. This delay is called *interlocking*. In fact, in the TMS320C30, contention for resources is not uncommon, and the control hardware delays the execution of instructions in order to resolve it. The programmer need not be aware of this, but obviously performance will be degraded. Interestingly, TI supplies a simulator that gives the detailed timing of any sequence of instructions, so that programmers intent on optimizing their code can do so easily.

Of the DSPs discussed here, the ones that make most use of interlocking are the TI processors. The internal timing of these devices is quite elaborate, and varies depending on whether internal or external memories are being used, and sometimes changes in complicated ways during the execution of a program. Nevertheless, we can gain intuition by considering a few more examples.

Example 2

Careful examination of the TMS32010/20/C25 architectures suggests that the ADD instruction is executed as

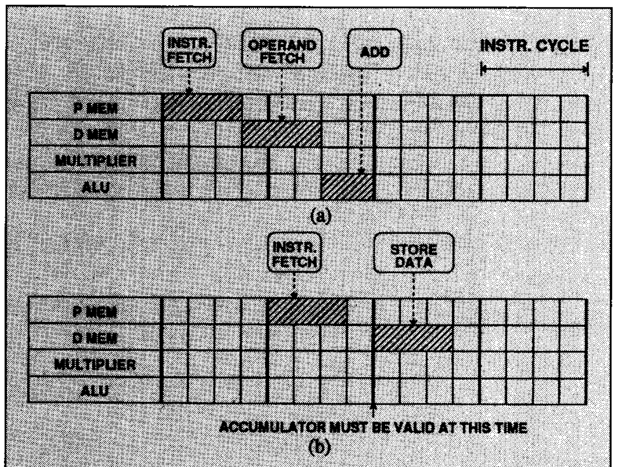


Figure 4. The author's estimate of the reservation table for the ADD (a) and SACH (b) instructions in the TMS32010/20/C25. An instruction cycle is divided into four subcycles. The SACH instruction is shown being fetched immediately after the ADD instruction, in which case the accumulator data must be valid at the time marked.

shown in Figure 4a. The ADD instruction adds an operand from data memory to the accumulator. The instruction cycle is divided into four subcycles, and internal memory accesses are completed in three subcycles. A total of two instruction cycles is required, although it is clear from the reservation table that ADD instructions can be executed at the rate of one per instruction cycle without any resource conflict. Furthermore, an ADD instruction can use the result (stored in the accumulator register) of the immediately preceding ADD instruction. So there is no pipeline hazard.

Notice that the actual addition takes only half an instruction cycle. This must be so because the ADD instruction can be followed by a SACH instruction that stores the result of the ADD (in the accumulator) to memory. The timing of the SACH instruction is shown in Figure 4b. The accumulator must be valid at the time marked in Figure 4b if the SACH instruction is to work properly. The time marked is precisely the end of the addition in Figure 4a. Furthermore, the SACH instruction may be followed by an ADD that uses the data value just stored to memory; this might be foolish, but it is certainly permissible. This determines that the write must be completed no later than shown in Figure 4b, so that the read of a following ADD instruction reads valid data. For this sequence of instructions (arithmetic, store, arithmetic) to work without evident pipelining, it is necessary that a write, the arithmetic, and a read complete within two instruction cycles.

The previous examples illustrate two important concepts. First, the execution of an instruction need not be constrained to one instruction cycle in order to *appear* constrained to one instruction cycle. Second, the internal timing of the DSP architecture can be inferred by carefully considering the requirements of different sequences of instructions.

Example 3

It is instructive to consider the FIR filter code for the TMS32010, reproduced from Example 15 of Part I:

```
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
```

We will later compare the timing of this implementation to the faster and more compact code using the RPTK and MACD instructions. The heart of the code is the alternating LTD and MPY instructions. The LT instruction loads the T register with a value from memory (see Figure 1 of Part I). The LTD instruction does the same thing, but in addition, the value loaded into the T register is copied into the memory location above where it came from (to implement a delay-line shift) and the product register is added to the accumulator. One possible timing for the LTD instruction is shown in Figure 5a. The addition could actually occur earlier, since it does not depend on the operand fetched, but control hardware is probably simpler if it is positioned as shown because of its similarity to the ADD instruction in Figure 4a. One possible instruction timing for the MPY is shown in Figure 5b. The LTD and MPY instructions can alternate as in Example 3 without conflict for resources, and all required data is available on time. Because of the late start of the addition in Figure 5a, the multiplication in Figure 5b has a full instruction cycle to complete its operation. Notice that execution of this instruction actually spills into a third instruction cycle.

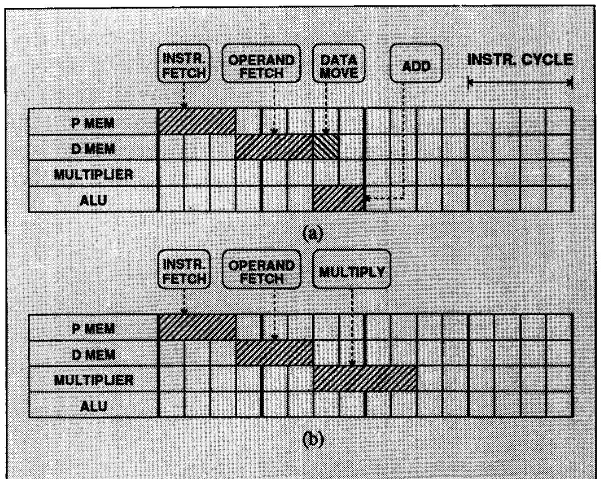


Figure 5. The author's estimate of the reservation table of the LTD (a) and MPY (b) instructions in the TMS32010/20/C25.

Example 4

The TMS32020 and TMS320C25 have a more compact construct for FIR filtering using the RPTK and MACD instructions:

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

One possible timing of the first MACD instruction is shown in Figure 6. In this case, both the multiplication and addition could begin earlier, but as shown their timing coincides with that of the MPY and ADD instructions, so the control hardware is probably simpler this way. As shown, the instruction consumes two instruction cycles before the next instruction can be fetched. If the instruction is fetched from the unit length instruction cache, however, then the doubly cross-hatched instruction fetch in Figure 6 is not required and only one instruction cycle is consumed. This is how these architectures achieve FIR filtering in one instruction cycle per tap.

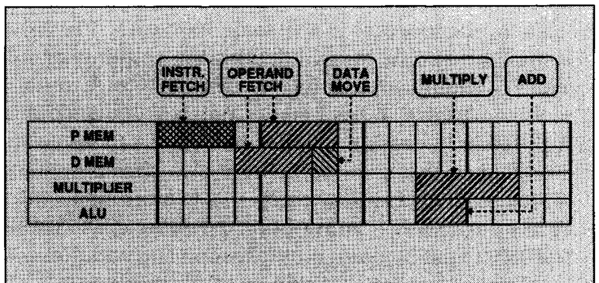


Figure 6. The author's estimate of the reservation table of the MACD instruction in the TMS32020 and 320C25. This instruction replaces the MPY and LTD instructions in an FIR filter implementation.

2.2. Time-Stationary Coding.

Although clearly beneficial for the programmer, interlocking has its costs. Higher performance can often be obtained by giving the programmer more explicit control over the pipeline stages. The most common way to do this is using time-stationary coding, in which an instruction specifies the operations that occur simultaneously in one instruction cycle.

Several DSPs are built around the rough outline of a reservation table shown in Figure 7. An instruction would explicitly specify three (or more) operations to be performed in parallel, two memory fetches and one (or more) arithmetic operations. Referring back to Figure 1, each instruction specifies *simultaneous* operand fetch and execute operations, rather than *successive* operand fetch and execute operations. In essence, the program model is one of *parallelism* rather than pipelining.

Example 5

A multiply and accumulate instruction for the DSP56001

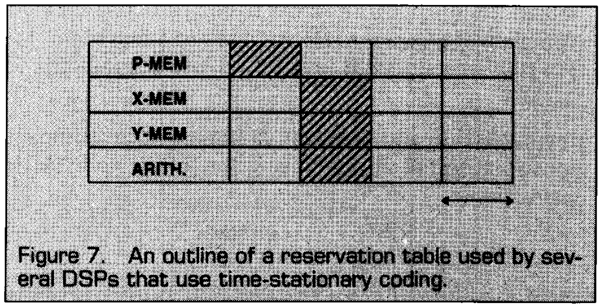


Figure 7. An outline of a reservation table used by several DSPs that use time-stationary coding.

or 96002 is:

```
MAC X0,Y0,A    X:(R0)+,X0    Y:(R4)-,Y0
```

There are three fields in this instruction, one specifying the arithmetic operation, and the other two specifying operand fetches *for the next instruction.* The operands of the arithmetic operation are the contents of the X0 and Y0 registers, which were loaded from memory *in a previous instruction.*

The result of the multiplication is added to the A register. Unlike any other DSP, the DSP56001 has integrated an adder into the multiplier, so that multiplication and accumulation are not two successive operations but actually occur together in the same hardware. The MAC instruction shown here multiplies the contents of X0 and Y0, simultaneously adding the result to A, so that in the next cycle, A has been completely updated. This is possible because multiplier hardware can be easily modified so that as it multiplies it also adds a number to the product. Unfortunately, this is more difficult to accomplish with floating point arithmetic, so in the DSP96002 floating point multiplication and addition can be specified separately using instructions like:

```
FMPY D4,D5,D0  FADD D0,D1   X:(R0)+,D4  Y:(R4)+,D5
```

The contents of D4 and D5 are multiplied and stored in D0. Meanwhile, the *previous* contents of D0 (not the result of the FMPY) are added to D1. In addition, the two data moves occur simultaneously, affecting the values of the D4 and D5 registers for *subsequent* instructions. In effect, the programmer explicitly fashions the pipeline by specifying the activity in each stage of the pipeline.

Compare this instruction with the 320C30 parallel multiply and add, MPYF3 || ADDF3 (see Example 18 of Part I). In the 320C30, the operands are fully specified in the instruction that uses then. In the 96002, the operands are specified (memory addresses given) in an instruction preceding the arithmetic instruction. Nonetheless, by permitting parallel instructions, the 320C30 has introduced an element of time-stationary coding.

Example 6

The NEC 77230 is similar to the DSP96002 in that multiply, add, and move instructions are specified in one instruction. Interestingly, the assembler syntax for such instructions attempts to mimic that of processors with hidden pipelining. An example of a parallel multiply and add instruction is:

```
MOV       LKR0,ROM
ADDF      WR0,M
INCBP0
INCRP;
```

Here, four fields can be specified on four separate lines, and a semicolon groups the fields. The first line specifies a move of two operands from memory (ROM and RAM) into the L and K registers. Meanwhile, the current contents (before the move) of the L and K registers are multiplied. No mnemonic is given for the multiplication because it occurs automatically in every cycle regardless of whether its result is used. Meanwhile, the product register M (from multiplication in the previous instruction) is added to the working register WR0, which acts as an accumulator. The last two lines specify the auto-increment for the pointers to memory (ROM and RAM) that are used in the first line. The four fields can also be specified together on one line.

Example 7.

The AT&T DSP16 and DSP16A use a reservation table outlined in Figure 8 for instructions with two operands from memory. A typical instruction is:

```
a0=a0+p    p=x*y    y=*r0++    x=*pt++
```

The product register p from the previous multiplication is added to a0 at the same time that a new product is formed using the contents of the x and y registers. Meanwhile, the x and y registers are loaded with new values using the address registers r0 and pt. This processor only has a demand ratio of two, so two-operand instructions consume two instruction cycles, as shown in Figure 8. However, if the instruction is fetched from the instruction cache, then the doubly cross-hatched operation (the instruction fetch) is not required, and only one cycle is consumed. Again, unlike the DSP56001, the multiplication and addition are specified as separate parallel operations.

Time-stationary coding has a number of advantages. First, the timing of a program is clearer. With interlocking, it is difficult to determine exactly how many cycles an instruction will consume because it depends on the neighboring instructions. Second, interrupts can be much more efficient. Since the programmer has explicit

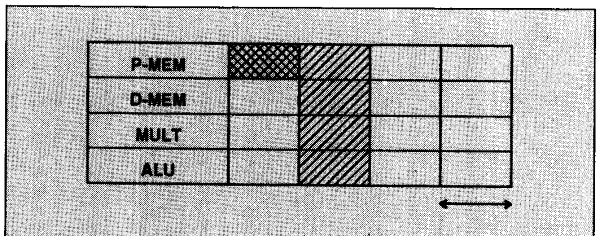


Figure 8. An outline of the reservation table for instructions with two operands from memory in the DSP16. If the instruction is fetched from the instruction cache, then the doubly cross-hatched operation is not required.

control over the pipeline, there is no need to flush the pipe prior to invoking the interrupt. One consequence of this is the possibility of very fast interrupts.

Example 8

The DSP56001 and 96002 have a fast interrupt, which takes exactly two instruction cycles. It can be used to grab input data and put it in a buffer, for example.

2.3. Data-Stationary Coding.

Time-stationary coding resembles microcode in that fields of the instruction specify operations in different parts of the architecture. While it is easy to grow accustomed to it, it is more natural to think of our algorithms in a data-stationary way. In data-stationary coding, a single instruction specifies all of the operations performed on a set of operands from memory. In other words, the instruction specifies what happens to that data, rather than specifying what happens at a particular time in the hardware. A major consequence is that the results of the instruction may not be immediately available in the subsequent instructions.

Example 9.

The most dramatic examples of data-stationary coding are the AT&T DSP32 and 32C. A typical instruction is:

```
r5++=a1=a0+*r7**r10++r17
```

The address registers r7 and r10 specify the two operands for the multiplier. The register r17 specifies the post-auto-increment for r10. The product is added to a0 and result stored in a1 and in the memory location specified by r5. The instruction is easy to read and understand, but it should be obvious that all these operations cannot be finished within one instruction cycle. The timing of the instruction, shown in Figure 9, actually covers six instruction cycles. An instruction of this type can be issued once per instruction cycle, so only a single cycle is consumed. Instructions like this one can be used to implement an FIR filter in one instruction cycle per tap. In fact, careful examination of the reservation table reveals that the hardware resources listed are 100 percent utilized by such a program if the number of filter taps is large enough.

Suppose the result is to be read from memory in a subsequent instruction and used as an operand. An instruction accomplishing this read would have to be fetched in the fourth cycle after the multiply and add instruction. In other words, the three instructions fetched immediately after the multiply and add instruction cannot read the result from memory because it has not yet been written to memory when they fetch their operands. The pertinent restrictions, evident in Figure 9, are summarized as follows:

- When an accumulator a*n* is used as an operand to the multiplier, the value of the accumulator is that established three instructions earlier.
- When a result is written to memory, the updated value of the memory location cannot be accessed until four instructions later.

Although results are not ready in the next instruction, data-stationary coding is no less efficient than time-stationary coding. In time-stationary coding, to specify a multiply, accumulate, and store operation on a pair of operands requires several instructions, and the total time to completion is the same as with data-stationary coding, assuming a similar hardware organization. In time-stationary coding, other operations proceed in parallel, specified in unused fields of the multiply and accumulate instructions. In data-stationary coding, other operations proceed in parallel specified by neighboring instructions.

Fast interrupts are more difficult with data-stationary

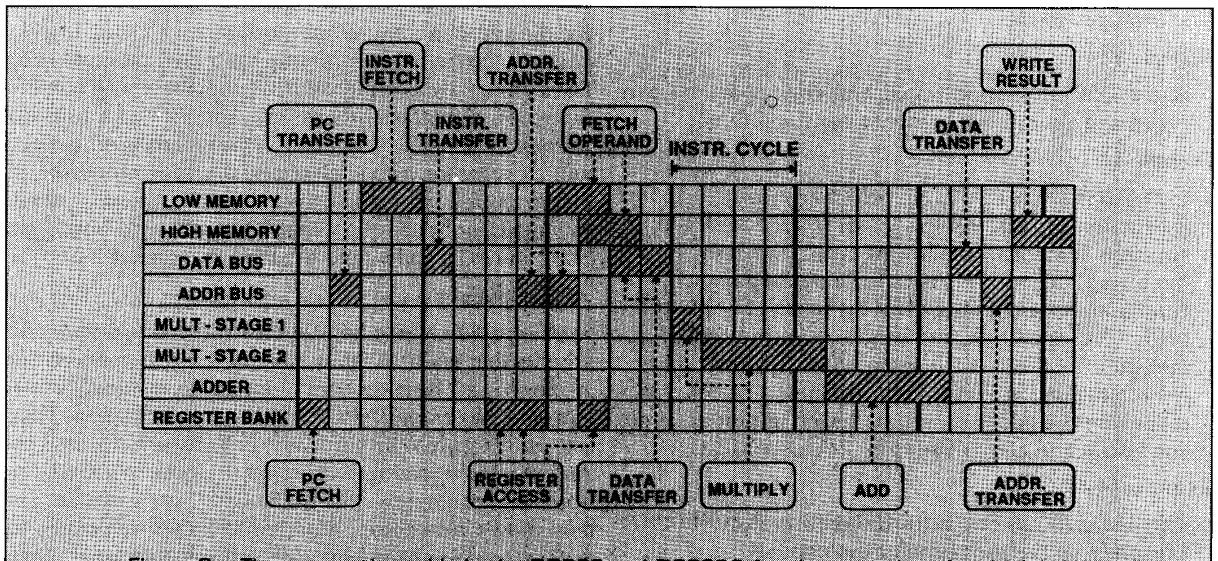


Figure 9. The reservation table in the DSP32 and DSP32C for the execution of multiply/add/store instructions. Notice that six instruction cycles are involved.

than with time-stationary coding, but are nonetheless possible.

Example 10

The DSP32C has a three-cycle quick interrupt. To accomplish this, the chip designers inserted a second set of pipeline registers that "shadow" the main set, storing the processor state when an interrupt occurs. Roughly 400 bits are stored. Nested interrupts are not possible, of course, since there is only one set of shadow registers.

2.4 Branching.

One difficulty with pipelining that we have thus far ignored concerns branching, particularly conditional branching. Several problems conspire to make it difficult to achieve efficient branching.

- There may not be sufficient time between instruction fetches to decode a branch instruction before the next instruction is fetched.
- If the program address space is large, the destination address may not fit in an instruction word, so a second fetch from the instruction memory may be required. Alternatives are paging and PC-relative addressing.
- In the case of conditional branching, the fetch of the next instruction cannot occur before the condition codes in the ALU can be tested.

Example 11

In the TMS32010/20/C25, in order to hide the pipelining from the programmer, branch instructions require several cycles to execute, where the exact number depends on the system configuration. In the case of an unconditional branch, the extra cycle is needed to fetch a destination address from the program memory. In the case of a conditional branch, the ALU condition codes can be tested while the fetch of the destination address proceeds.

Example 12

In the DSP16, unconditional branches consume two cycles and conditional branches consume three.

Example 13

In the DSP32 and DSP32C, when any control group instruction (if, call, return, goto) is executed, the instruction immediately following is also executed before the branch occurs. This is called a *delayed branch.* For conditional branches based on the result of a data arithmetic (DA) operation, the condition tested will be established by the last DA instruction *four instructions prior* to the test. It is evident from Figure 9 that the conditions on the adder cannot be tested in time to affect any instruction earlier than four instructions later.

Example 14

The TMS320C30 has both delayed and multi-cycle branches, so the programmer can choose.

Because of the inefficiencies of multi-cycle and delayed branches, it is important to use the low-overhead looping capability of the processors for tight inner loops, rather than using branch instructions.

Example 15

The 56001 and 96001 have the best developed low-overhead looping capability. Any number of instructions may be included inside a loop, loops are interruptible, and loops may be nested. The assembler syntax is straightforward:

```
          DO          10,END
          loop body
    END
```

Another technique used to avoid the inefficiencies of conditional branches is *conditional instructions.* These are operations other than branches that are conditionally executed. For example, a conditional write instruction can often be used to avoid a conditional branch.

3. THE FUTURE

As with all microelectronics, programmable DSPs have evolved considerably in the last ten years. It is easy to extrapolate the current trends and predict processors with more memory, faster MAC times, more I/O flexibility and bandwidth, etc. But such VLSI-driven improvement is by no means the only visible trend.

3.1. The Market.

The market for programmable DSPs remains limited to specialized products with relatively low volume (with the exception of modems and consumer products like the "Julie" doll by Worlds of Wonder). However, this is likely to change dramatically in the near future. Programmable DSPs are likely to become standard peripherals in personal computers and workstations. The standard microprocessor used now will continue to handle operating system tasks and interactive applications, but the DSP will handle real-time and compute intensive tasks. In principle, the same board with one (or a few) programmable DSPs can be used as a modem, a general purpose number-cruncher, a graphics processor, a speech and music synthesizer, a speech recognizer, a music analyzer, a digital audio processor, and a telephone message processor, including voice store-and-forward. Such a product would obviously enhance the capabilities of today's workstations and PCs, and would broaden the market for DSPs.

3.2. Parallelism.

Many applications have such stringent real-time constraints that multiple DSPs must be used in concert. Surprisingly, very little thought or effort has historically been put into designing DSPs for parallel computation. There are few features, in hardware or software, to ease the task of synchronizing processors or accessing shared resources. Fortunately, this is changing. For example, several newer processors have controllable *wait-states* for external memory accesses. This is invaluable for access to shared memory where the access may have to be delayed due to contention. In addition, most DSPs have extra pins that can be tested in software. These pins can be used to synchronize multiple processors. The TMS320C30 has specialized instructions for doing this; TI calls it a hardware interlock capability. Motorola facilitates the design of multiprocessor systems with the dual expansion ports in the DSP96002.

All of these are small steps, however. An essential capability that is almost totally lacking is software simulators capable of simulating multiple-DSP systems. System designers must build first, test later. A notable exception is Motorola, which supplies a simulator in the form of subroutines, which can be called from user-written code. Each call to such a subroutine emulates the state change of a processor in one clock cycle. A system designer planning to use more than one DSP56001 can write a C program that emulates the interconnection of the DSPs, shared memory, busses, and whatever other hardware is used (assuming the designer is willing to write emulation code for this other hardware). At Berkeley, we are integrating Motorola's callable simulator into a general-purpose hardware simulator from Stanford called Thor [Tho86] in order to get a clean user interface for designing and simulating parallel DSP systems.

A more radical approach to parallel DSPs has been proposed by NEC with the introduction of the μPD7281, a data flow machine for image processing. This chip may be simply ahead of its time, since it has not achieved wide acceptance.

3.3 Software.

One of the main impediments to widespread use of DSPs is that they remain difficult to use compared with other microporocessors. Products take years to develop, and programs take months to write even though the final code can often be stored in less than 1K words of program ROM. There are several reasons for this difficulty:

- Although the performance is impressive, today's DSPs are barely fast enough for many real-time applications. Programs must be tuned by hand to meet speed constraints.
- On-chip memories are small, and expansion beyond the chip boundaries is practically limited to only one (sometimes two) of the memory banks. Furthermore, off-chip memories that do not slow down the processor must be fast, and hence are expensive. Programs must be hand tuned to avoid squandering memory.
- Compounding the above problems, DSPs often compete with custom circuits in fiercely competitive marketplaces, such as in voiceband data modems. Programs must be hand tuned to minimize the overall hardware requirements of the systems.

One possible solution to the above problems is a good optimizing compiler. Some C compilers have appeared for some DSPs, but so far they do not appear to generate efficient enough code to meet the above constraints. Optimizing C compilers for the forthcoming generation of floating-point DSPs looks promising, however [Har88] [Sim88]. Regardless of their efficiency, the C compilers will inevitably be used for large applications of the DSPs, which are not practical to code by hand, such as graphics.

For digital signal processing, it is doubtful that good C compilers alone are the complete solution. Higher level design environments are being constructed to permit rapid prototyping (for algorithm development) and efficient code generation (for deployment in a competitive marketplace). The most promising systems under development are based on block-diagram programming, in which the user graphically constructs a block diagram of the algorithm. The user can use standard blocks from a supplied library, or define new blocks, possibly even in C. Burr-Brown has already demonstrated a preliminary code generator for the AT&T DSP32 that begins with a high-level block-diagram description of the algorithm. It uses the same interface as their DSPlay signal processing simulator. NEC is also known to be developing a system for the 77230. At Berkeley, we are developing a block-diagram programming environment called Gabriel that systematically manages real-time constraints, changes in sample rates, recurrences (feedback), conditionals, and iteration, and is capable of generating code for multiple processors [Lee87b]. A Gabriel screen is shown in Figure 10. This system is intended to be retargettable, and we have demonstrated its basic capabilities for the DSP56001 and DSP32.

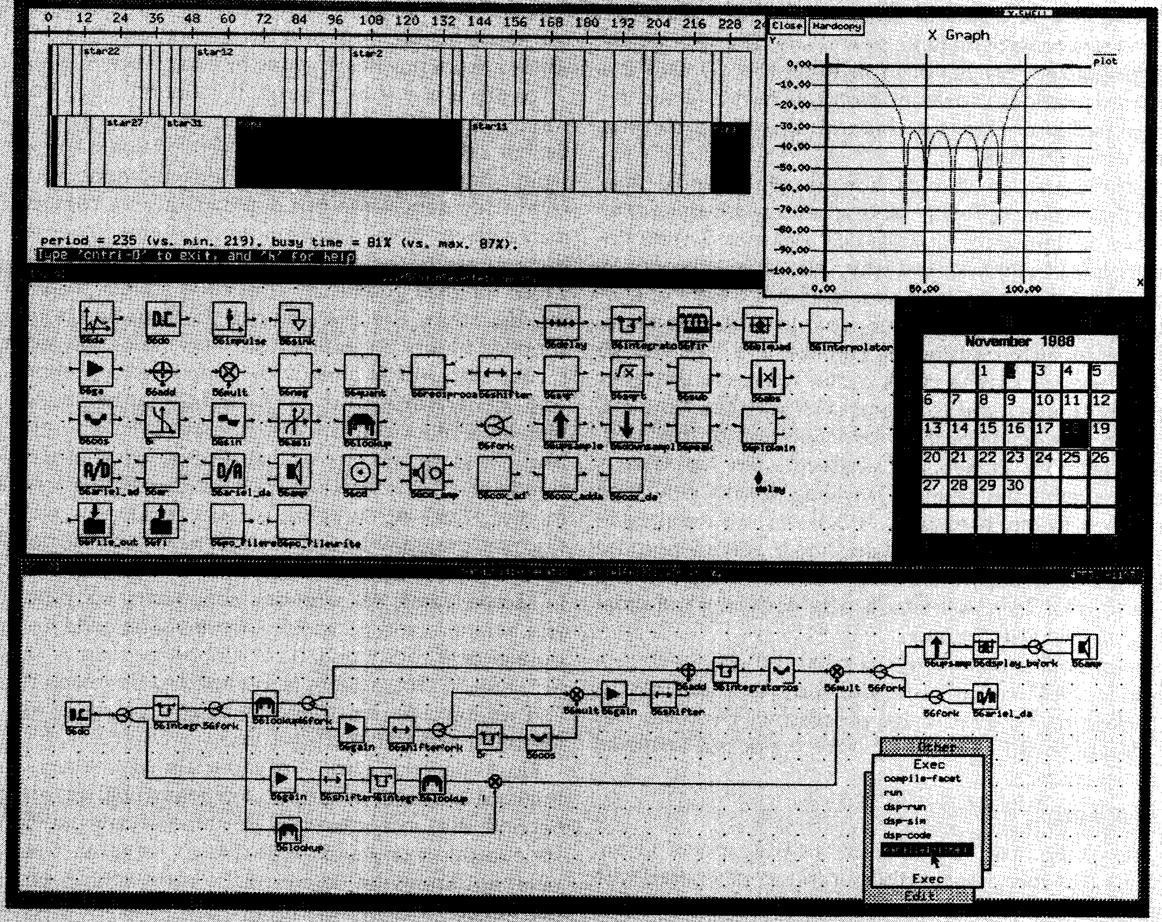
Block-diagrams have two important advantages. First, they are a natural description of many DSP algorithms. Second, they can potentially be automatically partitioned for execution on parallel processors [Lee87a]. The user need not know the details of the architecture, or even the number or type of DSPs. Block-diagram languages fit the *data-flow* model of computation, about which considerable theory has been developed. Generalizing these techniques to get the full expressive power of a programming language, however, is still a challenging research area.

3.4 Simpler Processors.

The dominant trend in DSPs is towards complexity, not simplicity. Every new device has features that the previous ones lacked, such as floating point, DMA, vectored interrupts, bit-reversed addressing, zero-overhead looping, and more extensive I/O. With all these features, DSPs are starting to tread on the turf of microprocessors. Unfortunately, this trend ignores the market that spurred the development of DSPs in the first place, which required arithmetic performance near the limits of what current technology could supply. A market exists for simple and fast DSPs. Although many manufacturers appear to be moving away from this market, some are embracing it. For example, the Hitachi DSPi and AT&T DSP16A are high speed chips with more limited functionality than the current generation of floating-point DSPs.

3.5 Semi-Custom Processors.

Many DSPs can be purchased in two versions, one with



```
period = 235 (vs. min. 219), busy time = 81% (vs. max. 87%).
Type 'cntrl-D' to exit, and 'h' for help
```

Figure 10. This figure shows an example of the screen layout of Gabriel, an experimental block-diagram DSP programming environment under development at Berkeley. The bottom window is the block diagram for an application, in this case an FM-based sound synthesis system. The middle window is a palette of functional blocks from which an application can be built. This particular palette supports the Motorola DSP56001. At the top left is a Gantt chart for a two-processor schedule automatically generated by Gabriel. A Thor-based hardware simulation of a multiprocessor system can be invoked to run the code produced by Gabriel. At the top right is a signal display window.

program RAM, and the other with mask-programmed program ROM. A typical development uses the first version for code development and migrates to the second version when ready for production. A DSP with mask-programmed ROM can be considered an application-specific IC.

Of course, the contents of the program memory may not be the only feature of the DSP that the user wishes to customize. It would be useful, for example, to customize the sizes of the memories. VLSI real-estate could be freed for this purpose by eliminating parts of the DSP that are not used. Possibilities include:

- Trim the arithmetic word width to what is actually needed.
- Remove the multiplier for low-speed applications, or applications that make little use of it, and replace with shift-and-add code.
- Remove I/O facilities when they are not used, such as DMA controllers.
- Customize barrel shifters to only perform the shifts actually used in the program.
- Customize the size of the register file.
- Eliminate bit-reversed or indexed addressing, if it is not used.
- Customize or eliminate the instruction cache, depending on whether it is required to meet real-time constraints.
- Customize the size of the address space, and hence the width of registers and busses and the number of pins.

A user would develop the application using a high-level description such as C, a block diagram language, or some other language, and given a real-time constraint, a compiler would automatically determine the required architecture parameters.

Automated layout programs have been demonstrated that are capable, in principle, of generating layouts that are parametrized in these ways. For example, a system called Lager that has many of these capabilities is under development at Berkeley [Pop85].

The idea of customizing an existing architecture has its limitations. An alternative approach is to automatically synthesize an architecture well suited in every way to the application. This approach appears to be most promising for applications with extremely high performance requirements and relatively low complexity, such as video-rate algorithms. The Cathedral project at KUL (Katholieke Universiteit Leuven) is an example of a research effort aimed in this direction [Cat88].

Although automatic layout has improved dramatically in recent years, there are still many difficult problems that remain to be solved before these techniques are fully practical. But the progress thus far is encouraging.

3.6. Pipeline Interleaving.

As discussed above, pipelining introduces a special set of difficulties that are either born by the architecture designer or by the programmer. Three techniques for dealing with pipelining are described above, hiding it, time-stationary coding, and data-stationary coding. Hiding the pipelining completely requires compromising performance. Time-stationary coding resembles microcode and can be difficult to generate (for either humans or compilers). Data-stationary coding has artifacts (called hazards) such as delayed data validity that again make code difficult to write. A fourth solution that has not yet been implemented in any commercial DSP is pipeline interleaving [Lee87c].

The idea is an old one, dating back to the 1960s. Consider the following strategy for writing code on a processor such as the AT&T DSP32C which uses data-stationary coding and extensive pipelining. Instead of writing a single in-line instruction stream, alternate instructions from three or four *reasonably independent* instruction streams, as illustrated in Figure 11. In other words, begin by identifying operations that can proceed in parallel.

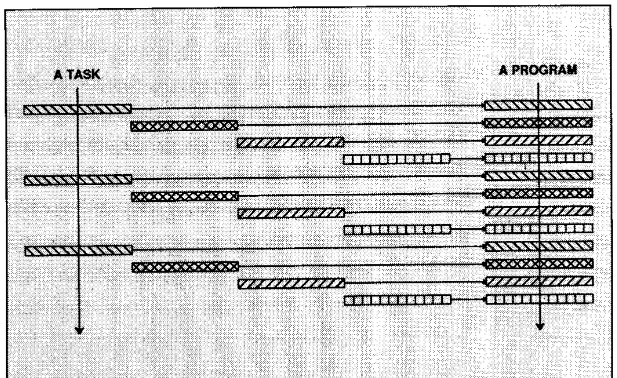


Figure 11. Instructions from independent tasks can be interleaved to overcome pipelining hazards.

Then partition the register set among these applications, and write code for each application, ignoring pipeline hazards. Then interleave the code so that pipeline hazards become irrelevant because sufficient time passes between any two instructions in one stream for results to be valid. The DSP32 conveniently provides a relatively large number of address registers (15) and accumulator registers (4) so that such partitioning is viable.

The main advantage of the above strategy is that the programmer can ignore the pipeline, but the architecture does not suffer the compromises that result from hiding the pipelining. However, a serious problem remains. Suppose that one of the three or four interleaved instruction streams requires a branch. Unfortunately, there is only one program counter in the DSP32, so all instruction streams must branch together. A simple solution is to introduce multiple PCs, one for each instruction stream. This technique is called pipeline interleaving.

Pipeline interleaving transforms a single DSP with data-stationary code and pipeline hazards into multiple processors (called processor slices) that have no pipeline hazards and actually share the same hardware, except registers. A pipeline interleaved architecture that can be built with conservative technology is described in detail in [Lee87c].

Although pipeline interleaving removes pipeline hazards, it introduces new problems. The algorithm must be partitioned for parallel computation. One proposal is to use the data flow properties of block-diagram languages, as described in [Lee87d] to automatically (at compile time) partition and synchronize the task.

4. ACKNOWLEDGEMENTS

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. Most importantly, the editor in chief of the *ASSP Magazine,* Tom Alexander, took a great interest in this paper and was extremely helpful. Any remaining errors are entirely the fault of the author.

REFERENCES

[Cat88] F. Catthor, J. Rabaey, G. Goossens, J. L. Van Meerbergen, R. Jain, H. J. De Man, and J. Vandewalle, "Architectural Strategies for an Application-Specific Synchronous Multiprocessor Environment," *IEEE Trans. ASSP,* February 1988, 36(2).

[Har88] J. Hartung, S. L. Gay, and S. G. Haigh, "A Practical C Language Compiler/Optimizer for Real-Time Implementation on a Family of Floating Point DSPs," *Proceedings of ICASSP,* pp 1674–1677, New York, April, 1988.

[Lee87a] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs For

Digital Signal Processing," *IEEE Trans. on Computers*, January 1987, C-36(2).

[Lee87b] E. A. Lee and D. G. Messerschmitt,"Synchronous Data Flow," *IEEE Proceedings*, September, 1987.

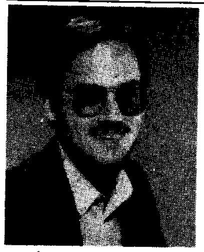
[Lee87c] E. A. Lee and D. G. Messerschmitt, "Pipeline Interleaved Programmable DSPs: Architecture," *IEEE Trans. on ASSP*, September, 1987 ASSP-35(9).

[Lee87d] E. A. Lee and D. G. Messesrschmitt, "Pipeline Interleaved Programmable DSPs: Synchronous Data Flow Programming," *IEEE Trans. on ASSP*, September, 1987 ASSP-35(9).

[Pop85] S. Pope, J. Rabaey, and R. W. Brodersen, "An Integrated Automatic Layout Generation System for DSP Circuits," *IEEE Trans. on Computer-aided Design*, July 1985 CAD-4(3) pp. 285–296.

[Sim88] R. Simar Jr. and A. Davis, "The Application of High-Level Language to Single-Chip Digital Signal Processors," *Proceedings of ICASSP*, pp 1678–1681, New York, April, 1988.

[Tho86] VLSI/CAD Group, "Thor Tutorial," Stanford University, Stanford, CA, 1986.



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.