ANALOG DEVICES

AN-386
APPLICATION NOTE

ONE TECHNOLOGY WAY • P.O. BOX 9106 • NORWOOD, MASSACHUSETTS 02062-9106 • 617/329-4700

Considerations for Selecting a DSP Processor (ADSP2100 Family vs. TMS320C25)

by Bob Fine

INTRODUCTION

Digital signal processing systems demand high performance. But high performance cannot be measured by a processor's multiplication/accumulation speed alone. What distinguishes DSPs from other types of microprocessor and microcontroller architectures is how well they perform in each of the following areas.

1. Fast and flexible arithmetic

   A DSP processor must provide single-cycle computation for multiplication, multiplication with accumulation, arbitrary amounts of shifting, and standard arithmetic and logical operations. In addition, the arithmetic units should allow for any sequence of computation so that a given DSP algorithm can be executed without being reformulated.

2. Extended dynamic range on multiplication/accumulation

   Extended sums-of-products are common in DSP algorithms. Protection against overflow in successive accumulations ensures that no loss of data or range occurs.

3. Single-cycle fetch of two operands (from either on- or off-chip)

   Again, in extended sums-of-products calculations, two operands are always needed to feed the calculation. A processor must be able to sustain two operand data throughput.

4. Hardware circular buffering (both on- and off-chip)

   A large class of DSP algorithms including most filters require circular buffers. Hardware to handle address pointer wraparound reduces overhead (increasing performance) and simplifies implementation.

5. Zero overhead looping and branching

   DSP algorithms are naturally repetitive and can easily be expressed as loops. Program sequencing that supports looped code with zero overhead provides the best performance and the easiest programming implementation. Likewise, overhead penalties for conditional program flow are unacceptable in signal processing applications.

Not all processors currently used for DSP and DSP-like functions meet these architectural and performance requirements equally well. This article examines these considerations for selecting a DSP processor, comparing two leading 16-bit fixed-point processors, the ADSP-2100A from Analog Devices and the TMS320C25 from Texas Instruments.

The three sections that follow discuss the five points above. The arithmetic section discusses items one and two, the addressing capabilities sections discusses items three and four and the program sequencing section discusses item five.

Program examples and benchmarks can be found at the end of this article.

ARITHMETIC CAPABILITIES

The basis of a successful DSP implementation is the ability to perform fast math. Arithmetic capabilities are the foundation of DSP performance.

General Purpose Math

One indicator of good arithmetic architecture is the ability to perform a wide range of arithmetic computation. These computations should be handled in a flexible manner so that the algorithm can be implemented without rearranging the order of the arithmetic operations or operands. If the arithmetic architecture is fixed, too special-purpose or limited and the algorithm must be rearranged, this poses extra work for the DSP designer or programmer and delays getting a system running. Algorithm development frequently turns out to be

much of the work of implementing a DSP system. If an algorithm can be used "as is" with no extra work, the design can be finished sooner and with less chance of error.

Arithmetic Architecture

Figure 1 shows a block diagram of the arithmetic section of the ADSP-2100A while Figure 2 shows that of the TMS320C25. Both of these devices utilize a modified Harvard architecture which can feed data operands from both program memory and data memory to the arithmetic section. The ADSP-2100A extends its Harvard architecture off chip while the TMS320C25 does not. Also, both of these devices work with 16-bit numbers.

The ADSP-2100A has three independent computational units: an ALU, multiplier/accumulator (MAC), and a barrel shifter. They are connected (via the R bus) so that the output of any unit may be used as the input for itself or any other unit on the next cycle. In addition, the ALU and MAC are directly connected to both the program and data memory buses. Operands for ALU and MAC operations can come from both memories or any combination of off-chip memory and other data registers in the processor.

The TMS320C25 contains a multiplier, an ALU, a 16-bit scaling shifter and additional shifters at the outputs of both the accumulator and multiplier. The multiplier has an input register and an output register. The multiplier has direct connection to both the program and data bus while the ALU connects only to the data bus (through the shifter) and to the output of the multiplier. Results are always sent to either the data bus or the accumulator registers. In some cases, the result must first be stored back in data memory before it can be used as an input to another calculation.

ADSP-2100A ALU

The ALU has two X and two Y input registers: AX0, AX1, and AY0, AY1. ALU operations are performed on any X-Y assortment of these input registers. They may be loaded from any combination of program and data memory or other data registers in the processor. The result of the operation appears in the ALU result (AR) or ALU feedback (AF) register. AR and AF can also be used as the X and Y operands (respectively) in any calculation. In addition, the result registers of the MAC and barrel shifter can also be used directly as X inputs to the ALU (and vice versa).

ALU instructions are coded in a register transfer, algebraic syntax. An example of addition is shown below. This example is a multifunction instruction. The first "clause" of the instruction (up to the first comma) is the addition operation. The second clause loads the X input register from data memory ("DM") and the third clause loads the Y input from program memory. An addition (or any other ALU operation) can be executed on a sustained, single-cycle basis. (These operand fetching clauses of the instruction may be omitted, if they are not needed.)

```
AR = AX0 + AY1, AX0=DM(I0,M0), AY1=PM(I4,M4)
```

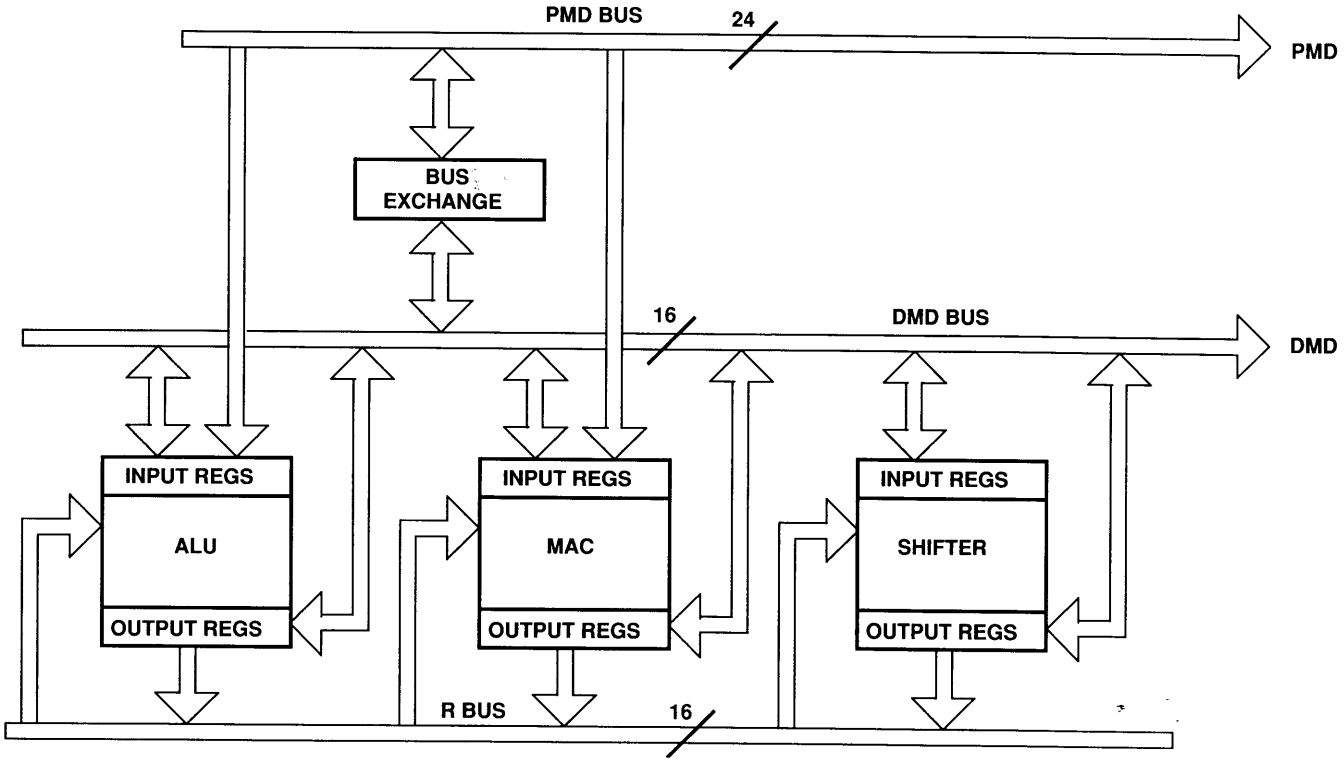

Figure 1. ADSP-2100A Arithmetic Section

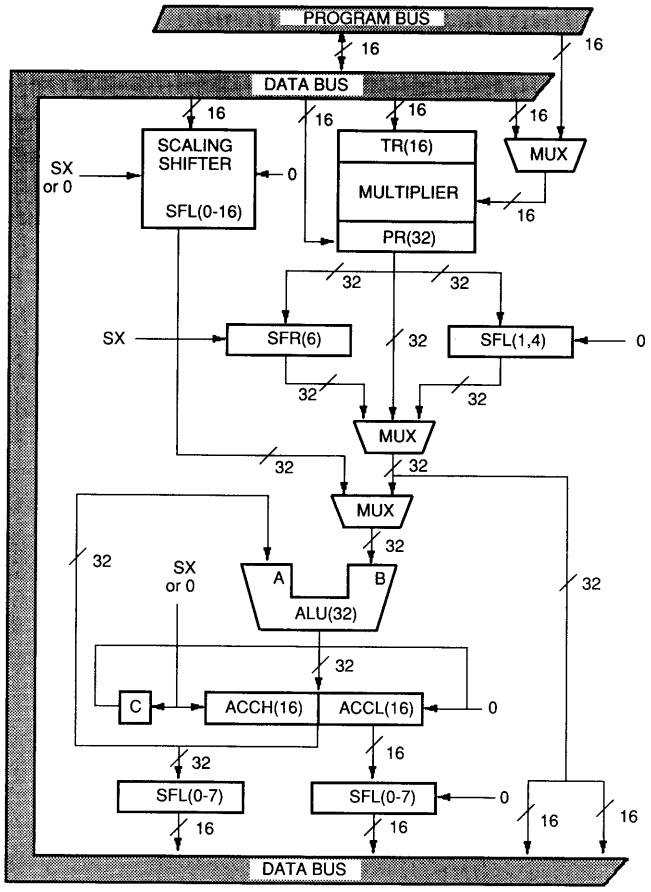

Figure 2. TMS320C25 Arithmetic Section

All ALU operations complete in a single 80ns cycle. (All references to cycles for the ADSP-2100A assume a 12.5MHz device.) The ADSP-2100A runs at full speed even with off-chip memory access.

TMS320C25 ALU

ALU operations require that one operand must come from the accumulator while the other comes from either the multiplier output or from the data bus through a shifter. To add two numbers, the accumulator must be loaded with the first data value. After the accumulator is loaded, a second number can be added to the accumulator. The instructions for the ALU are specified with a mnemonic. The two instructions required to add two numbers are shown below.

```
ZALH    <Data memory address>
ADDH    <Data memory address>
```

For the result to be used as an input value for anything other than another ALU operation, the data must first be stored back into data memory from the accumulator. Not all ALU operations can be performed in a single 80ns cycle; an add as shown above can be accomplished every two cycles. (All references to TMS320C25 cycles assume a 50MHz device with an 80ns cycle time.) Not all ALU instructions can be used with the repeat feature. There is a speed penalty for use of off-chip data memory; depending upon the memory configuration, some ALU operations can take as many as four cycles.

ADSP-2100A MAC

As shown in Figure 1, the ADSP-2100A multiplier/accumulator (MAC) sits next to the ALU. Like the ALU it has two X and two Y input registers, MX0, MX1 and MY0, MY1. The unit performs both multiplications and MACs independent of the ALU. This is a key difference from the architecture of the TMS320C25.

MAC operations are performed on any X-Y assortment of input registers. They may be loaded from any combination of program and data memory or other data registers in the processor. The result of the operation appears in the MAC result register (MR) or the MAC feedback register (MF). Like the ALU, the feedback and result registers can also serve as the X and Y operands for any multiplication or MAC operation. In addition, the result registers of the barrel shifter and ALU can also be used directly as X inputs to the MAC (and vice versa).

The instructions for the MAC are specified in a register transfer, algebraic syntax. An example is shown below. The first line shows multiplication of two signed operands and the second example shows multiplication with accumulation of one signed and one unsigned operand. (Signed and unsigned operands can be mixed in any combination.)

The second example is a multifunction instruction. The first "clause" of the instruction (up to the first comma) is the MAC operation. The second clause loads the X input register from data memory ("DM") and the third clause loads the Y input from program memory. Any MAC operation can be executed on a sustained, single-cycle basis. (These operand fetching clauses of the instruction may be omitted, if they are not needed, as in the first example.)

```
MR=MX0*MY0 (SS)

MR=MR+MX1*MY1 (SU), MX1=DM(I0,M0), MY1=PM(I4,M4)
```

The MR (MAC result) register is actually a 40-bit accumulator. For 16-bit calculations it is divided into two 16-bit pieces (MR0 and MR1) and an 8-bit overflow register (MR2). DSP applications frequently deal with numbers over a large dynamic range. The eight "overflow" bits of MR2 allow for 256 MAC overflows before a loss of data can occur.

All multiplication and MAC operations execute in a single 80ns cycle. Two new operands can be loaded into the input registers in parallel with the computation so that a new MAC operation with new operands can be started every cycle. The ADSP-2100A runs at full speed even with off-chip memory access.

TMS320C25 MAC Operation

There is no dedicated multiplier/accumulator hardware in the TMS320C25. The TMS320C25 requires the use of both the multiplier and the ALU to perform a complete multiplication/accumulation operation. A multiplication is performed by loading the T register with the first operand. Once this data is

loaded, a value from the data bus can be multiplied with the value in the T register. The instructions for the multiplier are specified with a mnemonic. The instructions for a multiplication are shown below.

```
LT  <data memory address>
MPY <data memory address>
```

A product is obtained every two cycles.

A full multiplication/accumulation requires the use of the ALU as well as the multiplier. The instruction required to perform a MAC operation is shown below. This instruction requires two words of program memory storage.

```
MAC <prog. mem. address> <data mem. address>
```

With both operands in on-chip memory, the MAC instruction takes three 80ns cycles in non-repeat mode. In repeat mode, it will require $2 + n$ cycles, where n is the number of repeats.

The TMS320C25 provides one bit of extension in the accumulator (a 33-bit accumulator compared to the 40-bit accumulator of the ADSP-2100A). After more than one overflow, the calculation is corrupted.

ADSP-2100A Shifter

The barrel shifter in the ADSP-2100A has an input register, SI, and accepts as inputs any result registers in the processor (e.g. MR1, AR) including its own result register, SR. Like the MAC result register set, the 32-bit SR is divided into two 16-bit registers, SR0 and SR1. The shifter also has an exponent register, SE, which is set automatically by the exponent adjust instructions and used for normalization instructions.

The shifter can place a 16-bit input value anywhere within a 32-bit field in a single cycle. The input can be shifted any number of bits from off-scale left to off-scale right. Other functions such as exponent detection, normalization, denormalization, block floating point exponent maintenance, and pattern merging can also be performed with this shifter. All shifter operations are performed in a single cycle. Numbers can be normalized, regardless of the number of bits to be shifted, in a single cycle.

TMS320C25 Shifter

The TMS320C25 scaling shifter shifts to the left from 0 to 16 bits. Two other shifters can shift data coming from the multiplier left 1 bit or 4 bits, or right 6 bits, or can shift data coming from the accumulator left from 0 to 7 bits. These two shifters add the advantage of being able to scale data during the data move instead of requiring an additional shifter operation.

DSP Requirement	ADSP-2100A	TMS320C25
Single-cycle ALU operations	✔	no
Single-cycle multiplication	✔	no
Single-cycle MAC operations	✔	✔*
Single-cycle shifting	0–32 bits	0-16 bits left or left or right 0-7 bits left or 1 or 4 bits left or 6 bits right
Accumulator overflow protection	8 bits	1 bit
Signed, unsigned or mixed-mode multiplications	✔	no mixed mode

*Approaches single-cycle efficiency when using repeat mode

Table 1. Arithmetic Capabilities

Arithmetic Summary

Table 1, on the facing page, summarizes the comparison of arithmetic capabilities of these processors.

The side-by-side architecture of the ADSP-2100A results in easier implementation of many DSP algorithms as compared to the fixed sequence, end-to-end architecture of the TMS320C25. Due to the dependency of the ALU on the multiplier for multiplication/accumulations in the TMS320C25, MAC operations can not be easily intermingled with ALU operations. This may require changing the order of calculations in an algorithm so that the interdependency of ALU and multiplier does not cause a problem. The local storage registers found in the ADSP-2100A make data movement for calculations easy. If data is to be used many times, it can reside in a register to eliminate the need of fetching it from memory each time. With local registers and the open architecture, it is easy to perform arithmetic operations in any order and to guarantee that input operands and results remain intact until explicitly overwritten or moved.

DATA ADDRESSING

A digital signal processor's ability to perform fast arithmetic is wasted if the required data cannot be fetched at an equal and sustained speed. Addressing hardware must support the dual operand fetches required to fully utilize the Harvard architecture found in most DSPs. Circular buffers are frequently found in DSP algorithms; hardware support of address pointer wraparound is another feature distinguishing a signal processor from other types of high-performance processors.

Figure 3 shows the address generation circuitry of the ADSP-2100A while Figure 4, on the next page, shows that of the TMS320C25.

ADSP-2100A Addressing

There are two independent address generators in the ADSP-2100A. One typically supplies addresses for program memory data fetches while the other handles data memory, making efficient use of the modified Harvard architecture. Each address generator has four I registers which store pointers (addresses), four M registers for address modifiers, and four L registers storing buffer lengths for modulo addressing of circular buffers.

The address generator can bit-reverse an address as it is sent out to the address bus for zero-overhead bit-reversing for the FFT. The I, M, and L registers can be also used for general purpose data storage.

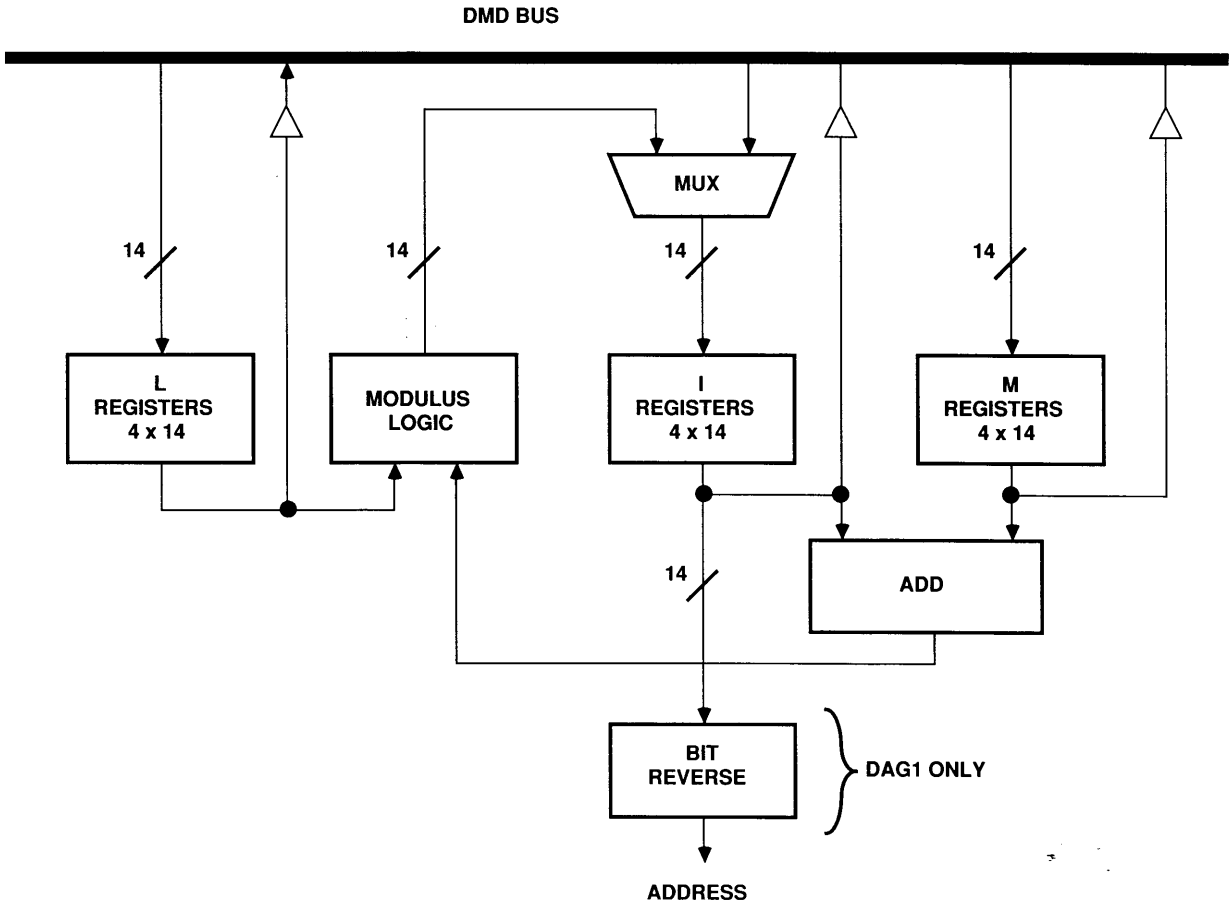

Figure 3. ADSP-2100A Address Generation Logic

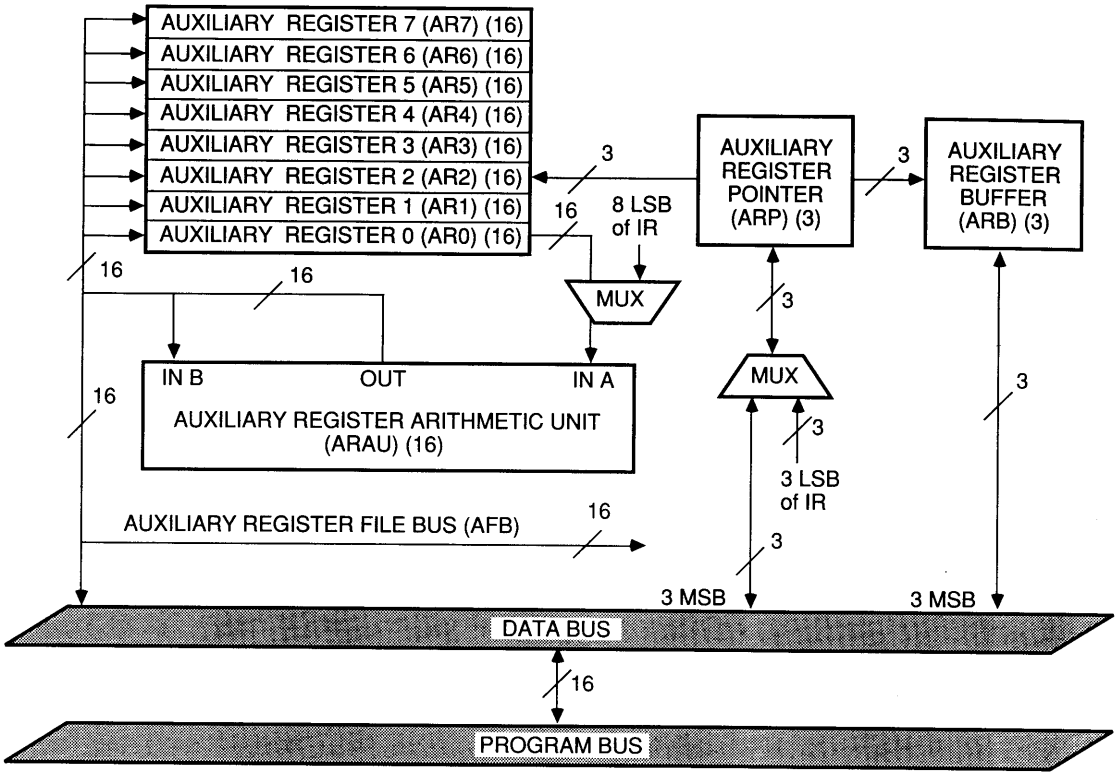

Figure 4. TMS320C25 Address Generation Logic

ADSP-2100A Indirect Addressing

With indirect addressing, the address in an I register drives either the data or program memory address bus. While the memory is being accessed, the address is simultaneously updated with the contents of any of the modify (M) registers, as shown in Figure 5. The specific pairing of I and M registers is up to the programmer. For example, I0 and M3 could be specified in the instruction as in

```
AX0 = DM(I0,M3)      {load AX0 from Data Memory}
```

The ability to mix I registers and M registers is especially useful for two-dimensional addressing or for supporting pointer increment and decrement without constantly reloading a new modify value. This instruction syntax shows explicitly what registers are used to generate the address and where the data is going; nothing has to be inferred.

Loading the length of a circular buffer into the L register activates the modulus logic, guaranteeing that the address is kept inside the buffer in a modulo fashion. This is maintained automatically by the address generator hardware and does not have to be calculated explicitly by the programmer. Circular buffers, such as for the delay lines of digital filters, are both transparent and require zero-overhead.

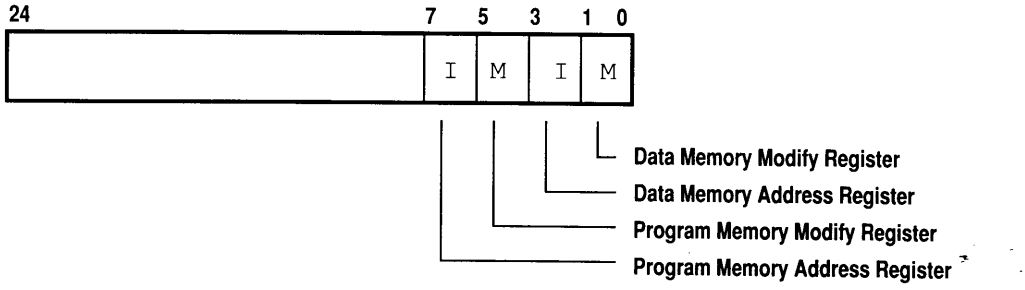

Figure 5. Indirect Addressing In ADSP-2100A

ADSP-2100A Direct Addressing

Due to the 24-bit width of the ADSP-2100A instruction, a full 14-bit address can be specified within a (single-word) instruction for single-cycle access to any data. Figure 6 illustrates this. Below is an example of an instruction using direct addressing to read from data memory.

```
MX0 = DM(some_label)
```

TMS302C25 Addressing

The auxiliary register file of the TMS320C25 is used for storage of addresses and a single modifier. Only one address can be supplied at a time with the auxiliary register file so that two data fetches cannot be achieved.

TMS320C25 Indirect Addressing

The auxiliary register file is connected to an arithmetic unit which will auto-index the contents of the auxiliary register or modify a register by the contents of auxiliary register number 0. The TMS320C25 has a single modify register. This limits the addressing capabilities for indirect addressing. No support is provided for circular modulo addressing; it must be calculated by the programmer as part of the computational load of the program. This diminishes the performance of DSP algorithms using circular buffers.

Also, the only way to read coefficients from program memory using indirect addressing is to use table reads.

TMS320C25 Direct Addressing

The TMS320C25 can directly access data within a 128-word block (compared to a 16K word block with the ADSP-2100A). A data page register is used in conjunction with the direct address to access a larger data space. To access data within a different block requires software overhead to update the 9-bit data page register. The update of the page register poses the requirement on the programmer to detect when the page boundary has been exceeded and when it is necessary to update the page register.

TMS320C25 Addressing Instructions

The instruction mnemonics of the TMS320C25 involve several addressing modes. Since the number of registers are limited, there is not a set of specific instructions to load registers from memory as with the ADSP-2100A. Indirect and direct addressing is specified within arithmetic instructions and, depending upon the memory configuration, can impose several overhead cycles (overhead can be as high as eight cycles with external memory). Some general syntax examples are shown below.

```
ADDH {*|*+|*-|*0+|*0-|*BRO+|*BRO-} [,<next ARP>]
MPY  {*|*+|*-|*0+|*0-|*BRO+|*BRO-} [,<next ARP>]
```

Specific examples of these are shown below:

```
ADDH    *
MPY     *0+
```

The first example uses the contents of an auxiliary register as the address and the second uses the contents of an auxiliary register as the address and adds the contents of auxiliary register 0 as a modifier. This instruction syntax can be hard to decipher because it does not directly name which auxiliary register is being used. That information is stored in the auxiliary register pointer (ARP).

The address generator can bit-reverse an address as it is sent out to the address bus for zero-overhead bit-reversing for the FFT. Auxiliary registers can also be used for general purpose data storage and the Auxiliary ALU can be used for limited math.

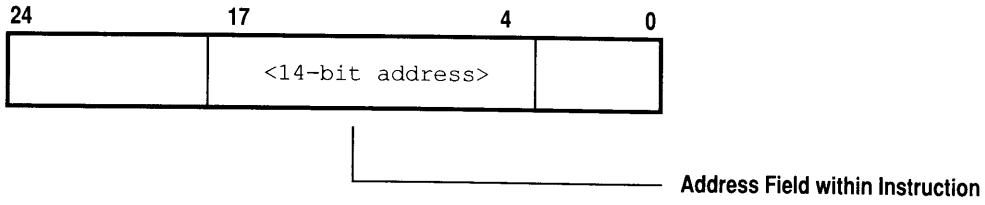

Figure 6. ADSP-2100A Direct Addressing

ADDRESS GENERATION SUMMARY

Sustaining high rates of arithmetic operations demands maximum performance from the data addressing part of a processor's architecture. Table 2 below summarizes the differences between the two processors in terms of their data addressing capabilities.

PROGRAM SEQUENCING

Efficient architectures for signal processing require fast arithmetic capabilities and matching speed in data addressing and fetching capabilities. To fully deliver the performance required for real-world signal processing, a DSP machine must execute its program with little or no overhead spent on maintaining the proper flow of control.

Efficiency in program sequencing has many different aspects; they cannot all be covered in this article. The comparison focusses primarily on two features

- the execution of loops and
- how branching and branching on conditions are handled.

Loops are fundamental to the way DSP algorithms are expressed in their natural mathematical form. Operations such as sums-of-products are repetitive. If the program can be efficiently expressed in a looped form then coding is quite straightforward and changing the program (for example, to increase the number of taps in a filter) requires very little work.

Branching is fundamental to program structure. Branching on conditions (and executing arithmetic on conditions) is a natural way to construct any program which must respond to its environment.

Program Sequencer Architecture

Figure 7 shows the architecture of the program sequencer of the ADSP-2100A and Figure 8, on page 10, shows that of the TMS320C25.

ADSP-2100A Program Sequencer

The program sequencer of the ADSP-2100A contains logic that selects a program memory address source and routes the address to the program memory address bus (PMA). This address selection occurs automatically in response to the

DSP Requirement	ADSP-2100A	TMS320C25
Single-cycle fetch of two operands from off-chip	✔	no
Single-cycle fetch of two operands from on-chip	✔	✔*
Generate new program memory and data memory addresses each cycle	✔	✔**
Modify two addresses by two different modify values on every cycle	✔	no
Bit-reverse data memory addresses for FFT	✔	✔
Automatic pointer wraparound for circular buffers	✔	no

* MAC & MACD instructions only

** Direct addressing mode only and only with MAC and MACD instructions

Table 2. Data Addressing Capabilities

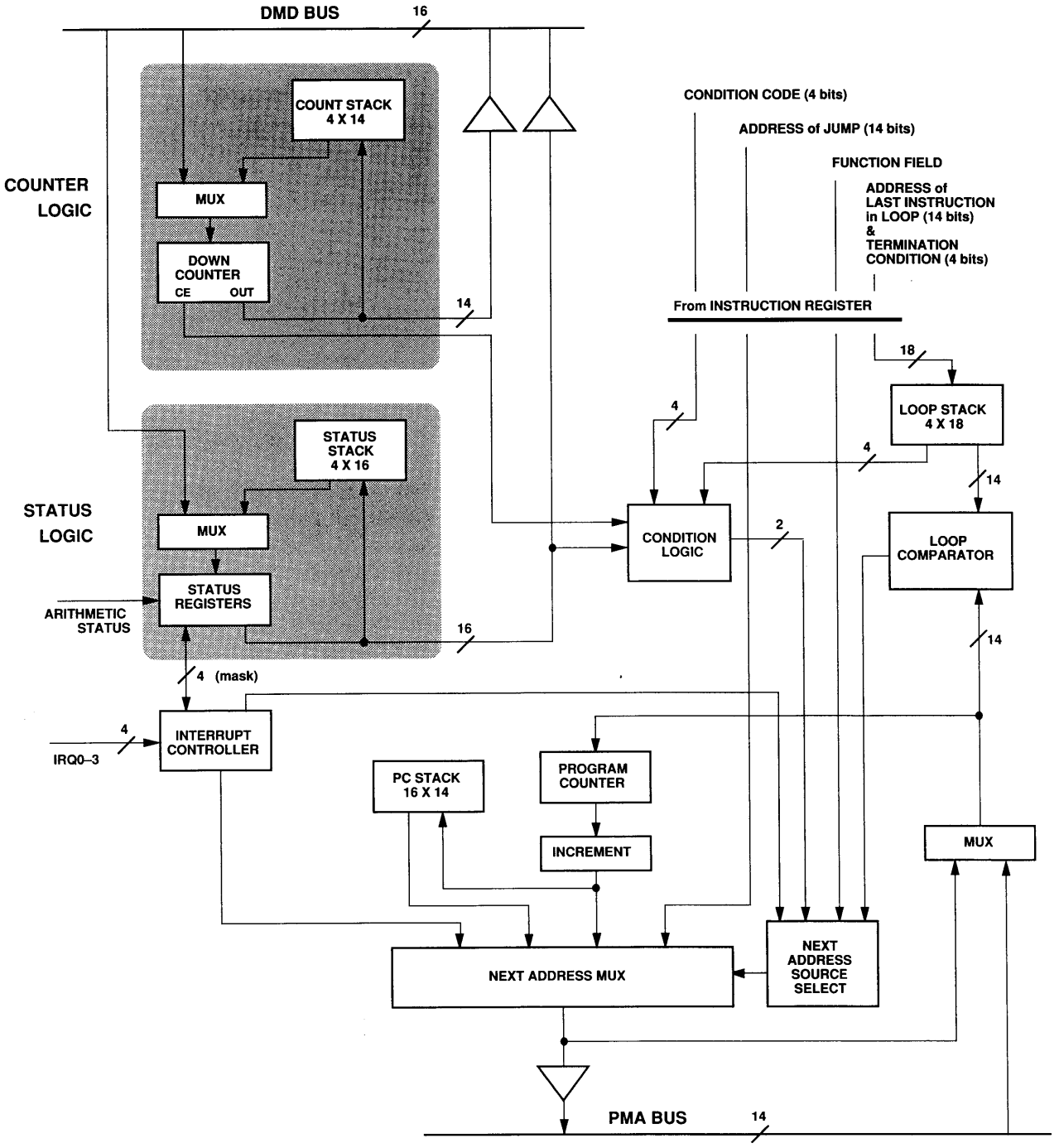

Figure 7. ADSP-2100A Program Sequencer Architecture

current instruction. The address placed on the address bus can come from

- the program counter (for sequential addressing),
- a 14-bit address in the instruction word itself, for direct jumps and subroutine calls,
- the PC stack, for returns from subroutines and interrupts, and
- the interrupt logic, to automatically vector to the interrupt routine upon assertion of any external interrupt.

All instructions execute in a single cycle; this applies equally to jumps, calls and interrupts. No instruction pipelining is required in the ADSP-2100A so that program flow is simple to understand.

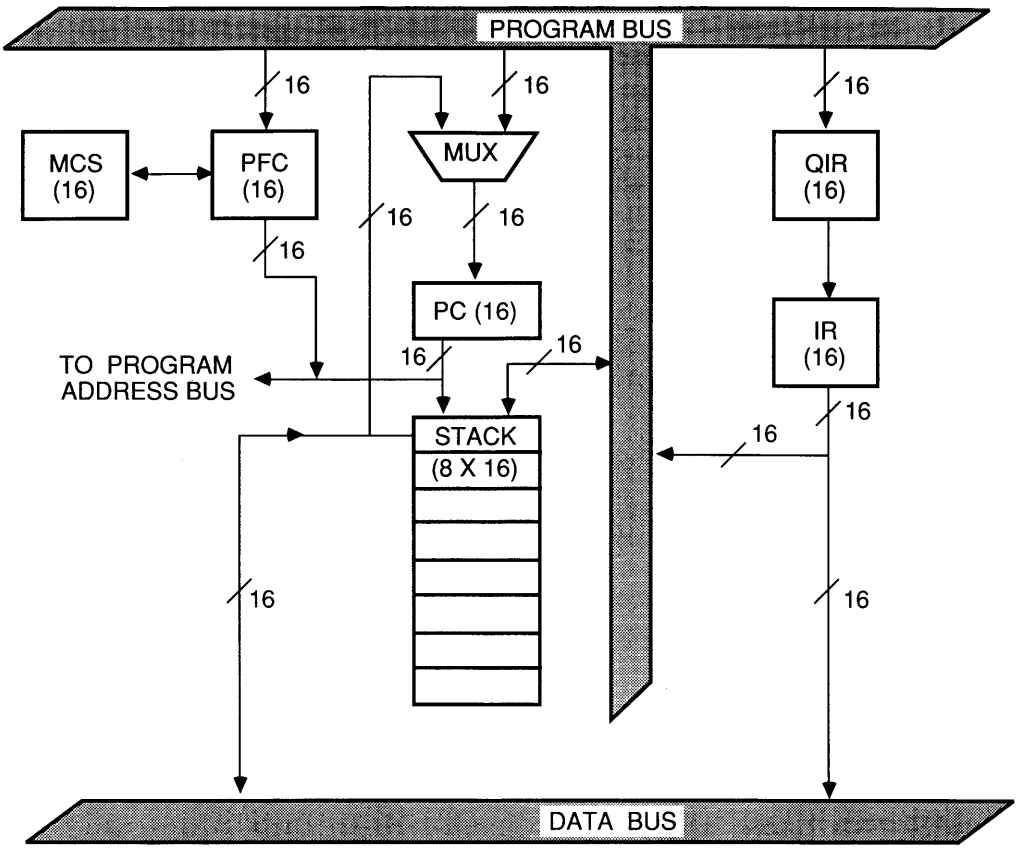

Figure 8. TMS320C25 Program Sequencer Architecture

When an interrupt occurs the complete status of the processor (stack status, mode status, arithmetic status and interrupt mask) is automatically pushed onto the status stack as part of the interrupt vector process.

ADSP-2100A Looping Capabilities

The ADSP-2100A program sequencer supports zero-overhead "DO UNTIL" loops. Using the count stack, loop stack and loop comparator, the processor can determine whether a loop should terminate and the address of the next instruction (either the top of the loop or the instruction after the loop) with no overhead cycle.

A DO UNTIL loop may be as large as program memory size permits. A loop may terminate when a 16-bit counter expires or any when any arithmetic condition occurs. The example below shows a three instruction loop that is to be repeated 100 times.

```
        CNTR = 100
        DO Label UNTIL CE
             First instruction of loop
             Second instruction of loop
Label:       Last instruction of loop
        First instruction outside loop
```

The first instruction loads the counter with 100. The DO UNTIL instruction contains the address of the last instruction in the loop (in this case the address represented by the identifier, *Label*) and also contains the termination condition (in this case the count expiring, CE). The execution of the DO UNTIL instruction causes the address of the first instruction of the loop to be pushed on the PC stack and the address of the last instruction of the loop to be pushed on the loop stack (See Figure 7).

As instruction addresses are output to the program memory address bus and the instruction is fetched, the loop comparator checks to see if the instruction is the last instruction of the loop. If it is, the program sequencer checks the status and condition logic to see if the termination condition is satisfied. The program sequencer then either takes the address from the PC stack (to go back to the top of the loop) or simply increments the PC (to go to the first instruction outside the loop).

The looping mechanism of the ADSP-2100A is automatic and transparent to the user. As long as the DO UNTIL instruction is specified, all stack and counter maintenance and program flow is handled by the sequencer logic with no overhead. This means that in one cycle the last instruction of the loop is being executed and in the very next cycle, the first instruction of the loop is executed or the first instruction outside the loop is executed depending upon whether the loop terminated or not.

ADSP-2100A Program Sequencer Instructions

There are many conditional instructions for the ADSP-2100A. Most arithmetic instructions as well as jumps, subroutine calls, returns from interrupts and returns from subroutines may all be conditional. The program sequencer decides on the fly whether the condition is true and what action to take, requiring zero overhead cycles. The coding of conditional jumps, subroutine calls and returns is straightforward. Some examples of the syntax are shown below.

```
IF condition JUMP label
IF condition JUMP I4
IF condition CALL label
IF condition CALL I4
IF condition RTS
```

In the above examples, I4 references an address generator register for indirect branching. *Condition* refers to any of a set of sixteen arithmetic conditions in the processor and *label* refers to any address or label in the program memory space.

TMS320C25 Program Sequencer

The program sequencer logic of the TMS320C25 controls instruction execution and consists of a program counter and related hardware. Instruction execution for the TMS320C25 utilizes a three-level pipeline consisting of a prefetch, decode, and execution stage. A prefetch counter (PFC) contains the address of the next instruction to be prefetched. The prefetched instruction is loaded into the instruction register (IR), unless the instruction register still contains an instruction currently executing. In this case the prefetched instruction is temporarily stored in the queue instruction register (QIR). The instruction pipeline (which can be either two levels or three levels depending upon the memory configuration) in conjunction with multi-cycle instruction execution can make program flow complex and difficult to understand. Calculating a benchmark for a particular algorithm can also become difficult for the same reason.

The program counter can supply an address for sequential addressing. The single 8-deep PC stack is used for storage of return addresses as well as for providing the ability to push and pop data for the accumulator. An interrupt flag register (IFR) is used for the vectoring to an interrupt routine. Unlike the ADSP-2100A, status is not automatically saved on the TMS320C25 for interrupts so that the programmer must perform any save and restore functions explicitly. Logic is included to repeat an instruction as many as 256 times.

Branch instructions which contain a direct address require multiple program memory locations because both the instruction bits and the address can not fit in the 16-bit instruction width.

Instruction Pipelining in the TMS320C25

Anytime the flow of the program deviates from sequential instruction fetches, the instruction pipeline must be emptied and then refilled based on the destination address of the branch, call or interrupt vector. These types of operations require at least three cycles to execute when fetching the instruction from external memory or from internal program ROM. This type of instruction pipelining is not found in the ADSP-2100A (the fast instruction execution speed is achieved by other design techniques) and no extra overhead is encountered in the ADSP-2100A for jumps, subroutines or interrupts regardless of whether they are conditional or not.

TMS320C25 Program Sequencer Instructions

Arithmetic instructions cannot be conditional. Only branch instructions are conditional. Branch instructions with direct addresses require two program memory words.

```
BACC
BANZ address
BGEZ address
BIOZ address
```

The TMS320C25 must use an explicit instruction to check a loop count and perform conditional branches. This requires one cycle of overhead for each iteration. A repeat instruction is also provided. It allows a single instruction to be repeated up to 256 times. The syntax is shown below.

```
RPT data memory address
RPT {*|*+|*-|*0+|*0-|*BRO+|*BRO-} [,<next ARP>]
```

PROGRAM SEQUENCER SUMMARY

Efficient looping capabilities are very important for DSP algorithms due to their repetitive nature. If zero-overhead looping capabilities are not found in a DSP processor, as with the TMS320C25, straight line coding may be required to avoid the overhead incurred with looping. This type of coding avoids overhead cycles but makes inefficient use of program memory space. In fact, the TMS320C25 can require hundreds of times more program memory than the ADSP-2100A for algorithms such as the FFT because of this characteristic of its architecture.

DSP Requirement	ADSP-2100A	TMS320C25
Zero-overhead looping (1 instruction inside loop)	✔	✔
Zero-overhead looping (2 or more instructions inside loop)	✔	no
Conditional arithmetic instructions	✔	no
Zero-overhead branching	✔	no*
Speed achieved with pipelining	not required	✔
Automatic status saving during interrupt vector	✔	no

*Affects the pipeline; the exact number of cycles of overhead is a function of memory configuration and branch destination

Table 3. Program Sequencing Capabilities

A FOOTNOTE:

The ADSP-2100A Compared to the TMS320C30

The TMS320C30 architecture has some of the same characteristics as the TMS320C25. Two modify registers, IR0 and IR1, were added in recognition of the need for more than a single modify register. Some DSP algorithms, especially applications such as video signal processing, will still require additional modify registers, as in the ADSP-2100A.

The TMS320C30 contains one block size register for circular buffers. If the DSP application deals with more than one channel of filtering, for example, several circular buffers may need to be maintained simultaneously. With the TMS320C30 the block size register and an auxiliary register must be reloaded and the old value must be saved in memory or the register file. This requires extra overhead that will not occur in the ADSP-2100A with the use of the eight length registers.

The TMS320C30 also has an instruction pipeline similar to the TMS320C25. Pipeline conflicts as well as overhead cycles in the program flow will result because of the instruction pipeline. The ADSP-2100A instructions execute in a single cycle with no extra overhead regardless of whether that instruction is a jump, subroutine call, or conditional instruction.

The looping capability of the TMS320C30 is improved over that of the TMS320C25. It still cannot execute some constructs supported by the ADSP-2100A such as terminating a loop upon an arithmetic condition as well as the expiration of a count.

The ADSP-2100A has a true off-chip Harvard architecture with two separate external memory maps. Even though the TMS320C30 has two external memory ports, they reside in the same single memory map.

SUMMARY

The DSP processors available on the market today vary drastically in their ability to meet these requirements. In fact, some DSP-oriented processors, like the TMS320C25, are better high speed microcontrollers than they are DSP processors. Analyzing the requirements of your DSP system and matching them to the capabilities of a DSP architecture will assure efficient operation.

Digital signal processing is a specialized branching of processor design and application. The fundamental requirements of DSP are summarized in Table 4 below.

Due to space limits, this article does not cover many topics in detail. Consult the *ADSP-2100 User's Manual* and the *ADSP-2100 Cross-Software Manual* for a greater depth of information on this processor.

DSP Requirements	ADSP-2100A	TMS320C25
Fast arithmetic	✔	✔
Extended dynamic range on multiplication / accumulation	✔	no
Single-cycle fetch of two operands (from either on- or off-chip)	✔	no
Hardware circular buffering (both on- and off-chip)	✔	no
Zero overhead looping & branching	✔	no

Table 4. Overall DSP Requirements

APPENDIX A: PERFORMANCE BENCHMARKS

Since evaluating every detailed feature of many DSP processors can be time consuming and tedious, performance benchmarks can be frequently used to tell the whole story. If the arithmetic, address generation, and program sequencing architecture is superior it will be reflected in the benchmarks. A list of benchmarks are shown below for the ADSP-2100A and the TMS320C25. TMS320C25 benchmarks could not be located for all the algorithms shown.

Algorithm	ADSP-2100A	TMS320C25
8-pole canonic IIR Filter (5X) [1, 2]	3.28 µs	3.52 µs
Matrix multiply 3x3 times 3x1 [1]	1.6 µs	1.8 µs
1024 point FFT [1, 3]	4.23 ms / 3161 words	9.08 ms / 23636 words
1024 point FFT [4]	2.97 ms	–
1024 point FFT [5]	–	7.1 ms [6]
ADPCM Full Duplex [7]	68 µs	-
Tenth order LPC Analysis [8]	0.36 ms	-
DTMF [9]	12 channels	–

1 These benchmarks are directly from the September 29, 1988 issue of EDN.
2 This is an eight filter made by cascading 2-pole canonic biquad sections using the 5-multiply technique.
3 This performs a complex, 1024-point, radix-2 FFT. Results include bit-reversal.
4 This is a complex, 1024-point, radix-4 FFT, including digit-reversal, fully optimized.
5 Source: Texas Instruments Seminar Materials (1988). This is for a 1024-complex, radix-4, straight line coded FFT
6 This is based on a 100ns cycle time part, scaled for an 80ns part, the time should be 5.6ms
7 Based on CCITT G.721
8 Using a 240-point rectangular window
9 Includes digit validation, speech rejection and software µ-law expansion

Table 5. Benchmark Comparison

APPENDIX B: PROGRAM EXAMPLE

To illustrate some of the issues discussed above, a code example is shown below for the ADSP-2100A and the TMS320C25. To avoid long listings and confusion, a short program which performs a matrix multiply of a 3 x 3 matrix with a 3 x 1 matrix is shown. Both processors perform identical tasks so that no interpretation of the type of algorithm is required. Both code examples do not show any initialization of pointers or the set up of any modes. The examples only focus on the core operation for simplicity.

Because these examples are short, the performance advantage of the ADSP-2100A is not as apparent as in a more realistic example. Nevertheless, the ease of coding and benefits of the looped structure can be seen. Consult the benchmark tables for more definitive performance comparisons.

These program examples are taken from the EDN benchmark study of DSP processors that appeared in EDN magazine September 29, 1988. A diskette containing this code is available from the publisher of EDN.

ADSP 2100A Code Example Description

The code above uses the looping capabilities of the ADSP-2100A and can be expanded for larger matrices by simply changing the number of loops (the value loaded into the counter). The I registers of the address generator are initialized to point to the first element of the input arrays and the output array. Circular buffering is used so that the address will circulate through the array. This allows automatic circulation through the 3 x 1 matrix for each row calculation. Data can reside anywhere in data memory without restrictions and can take up the full 16K data space if necessary.

The routine starts by fetching the first element of the the two matrices from the data memories. One value is fetched from program memory data space while the other is fetched from data memory. The counter is then loaded with the number of rows of the matrix. The DO UNTIL loop is set up and the computations can begin.

Matrix elements are multiplied while the next elements are fetched. Both values are treated as signed numbers with the specification (SS). The products are accumulated with the MAC instructions and the final sum of products is rounded so that it can be sent as a 16-bit result to data memory.

The program memory requirements are not large because of the looping capabilities. The program size does not change for larger arrays, only the loop counter value changes. Also, for multidimensional arrays, the loops can be nested with column loops inside of row loops for very compact code. Total execution time for this example program is 1.6μs.

ADSP-2100A Matrix Multiply Code Example

```
start:      MX0=DM(I0,M0), MY0=PM(I4,M4);
            CNTR=3;
            DO row_loop UNTIL CE;
                MR=MX0*MY0(SS), MX0=DM(I0,M0), MY0=PM(I4,M4);
                MR=MR+MX0*MY0(SS), MX0=DM(I0,M0), MY0=PM(I4,M4);
                MR=MR+MX0*MY0(RND), MX0=DM(I0,M0), MY0=PM(I4,M4);
row_loop:   DM(I1,M0)=MR1;
```

TMS320C25 Code Example Description

The code for the TMS320C25 is straight line since no looping capability is available. Direct addressing is used so that the data must be restricted to a 128 word block. If the matrix is to be expanded, the program must be rewritten because it is not general purpose. Indirect addressing is not used because there is no support for circular buffering and there is no looping capability that allows for more iterations to be specified.

The routine starts by loading the T register with the first matrix element. Next, data of the second matrix is fetched and the two values are multiplied. A new value is loaded into the T register and the first product is saved in the accumulator. Another multiply is performed. The T register is loaded with another value as the previous products are accumulated. Notice that the fetching of operands and accumulate, and the multiply are specified and performed in different cycles as opposed to the ADSP-2100A which allows for everything to be specified and performed in a single cycle. The data result is then written to data memory.

As the matrix gets larger, the code space requirement also gets larger. The lack of looping capability results in the inefficient use of program memory space. If looping is desired, it would need to be done explicitly with extra instructions. These extra instructions would introduce overhead and would hurt the benchmark performance. Almost all benchmarks for the TMS320C25 need to be done with straight line code for good performance. This program executes in 1.8μs. This approach, of course, can use quite a bit of program memory space.

E1306–16–6/89

TMS320C25 Matrix Multiply Code Example

```
LT a1
MPY b1
LTP a2
MPY b2
LTA a3
MPY b3
LTA a4
SACH R1,S
MPY b1
LTP a5
MPY b2
LTA a6
MPY b3
LTA a7
SACH R2,S
MPY b1
LTP a8
MPY b2
LTA a9
MPY b3
APAC
SACH R3,S
```

PRINTED IN U.S.A.