

# APPLICATION DECIDES OPTIMAL DSP PROCESSOR ARCHITECTURE

## ADSP-2100 family best suited for real-world signal processing Instruction set, arithmetic, data flow, program sequencing are keys

by Bob Fine and David Fair

Selecting an appropriate digital signal processor (DSP) involves many tradeoffs. First, there are traditional engineering factors like cost, memory, speed, programming effort, power consumption. But unlike the selection of an op amp or a/d converter, a DSP's suitability for a given job also involves an aspect more difficult to characterize: the appropriate processor *architecture*. It makes a processor easier to program optimally—a major consideration—and able to accomplish the complete digital signal-processing (DSP) algorithm in less time while requiring less memory and other support.

Real-world DSP applications include digital filters, such as the finite impulse-response (FIR) system modeled in Figure 1, and the decimation-in-time fast Fourier transform, shown schematically in Figure 2; the latter's algorithm flow chart appears in Figure 3. Both applications involve a repetitive pattern of data

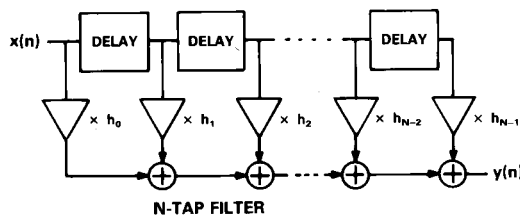


Figure 1. FIR filter principle.

handling and unique symmetries in data structure. To be effective, DSP architectures for processing real-world signals in real time (or at least fast enough to be truly useful) must provide:

- a. *fast and flexible arithmetic*, including single-cycle multiplication (often with accumulation), shifting, and logic operations.
- b. *extended dynamic range on multiply-and-add*—needed for this commonly used DSP function—to protect against overflow, resulting from many successive accumulations.

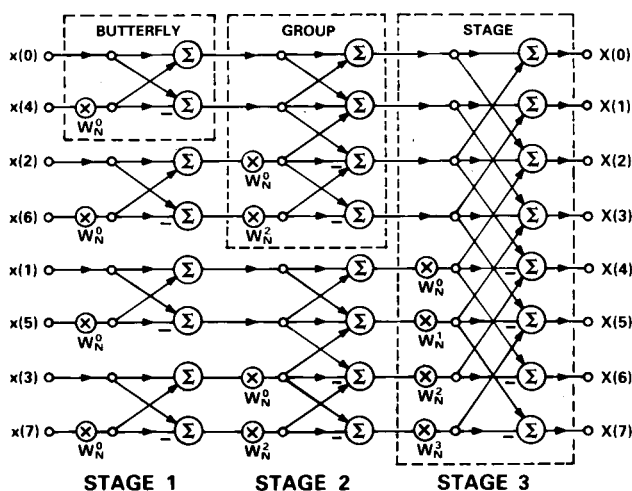


Figure 2. Eight-point decimation-in-time FFT.

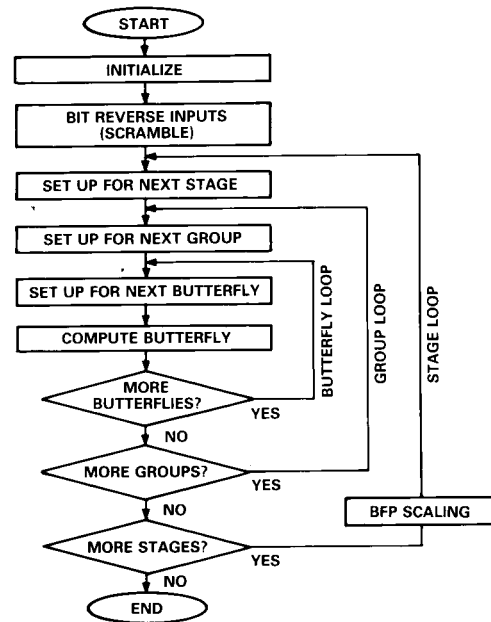


Figure 3. Flow chart for decimation-in-time FFT.

c. *single-cycle fetch of two operands*, as data and coefficient for repeated multiply-and-add calculations.

d. *circular buffering in hardware*, for efficient execution and minimal software burden in address generation for applications such as digital filtering.

e. *looping and branching with zero overhead*: frequently employed DSP algorithms are often inherently repetitive; commonly implemented as program loops, they should execute without undergoing the penalty of extra cycles for checking the end of the loop or for conditional branching out of the loop.

Real-time signal-processing applications are especially critical to three aspects of DSP architecture: *arithmetic processing* (a and b), *data addressing* (c and d), and *program sequencing* (e). The ADSP-2100 family\* of DSP processors is designed with these factors especially optimized for timely signal processing, in contrast to general processing and off-line calculations.

### ARITHMETIC PROCESSING

The arithmetic section of the ADSP-2100 has three computational units (Figure 4)—linked, but independent of one another. The arithmetic/logic unit (ALU), multiplier/accumulator (MAC), and barrel shifter are connected via the R ("results") bus, so that the output of any unit can be used as an input for itself or any other unit on the next cycle. Operands for the ALU and MAC can come from program- or data memory—or specified on-chip registers.

ALU operations can be done on any X-Y pair of the two X and two Y input registers, which in turn can be loaded with data words from any combination of program or data memory buses, or other data registers within the processor. Here's an example of a multifunction ALU instruction combining addition and two

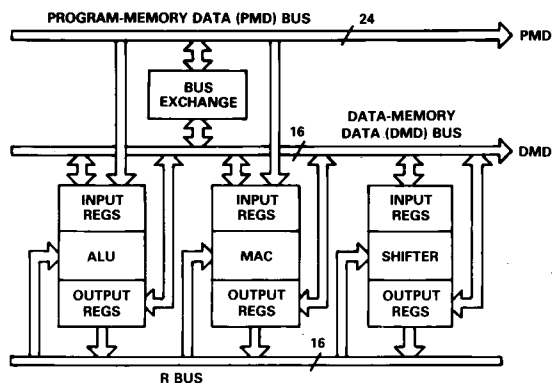


Figure 4. Simplified block diagram of ADSP-2100 arithmetic section.

memory reads (AR is the ALU output register, AX0 and AY1 are two of the ALU input registers):

$$AR = AX0 + AY1, AX0 = DM(I0,M3), AY1 = PM(I4,M7)$$

The first part (up to the first comma) is the addition, the second part loads one X input register with a new operand from data memory, and the third part loads a Y register with a new operand from program memory. The terms in parentheses specify the memory locations to be computed by a pair of address generators. The entire operation is completed in a single cycle (80 ns for a 12.5-MHz-clocked ADSP-2100A).

In contrast, some DSP architectures require one operand to come from the accumulator while the other comes from either the multiplier or from the data bus (via a shifter). When these other architectures are used to add two numbers, the accumulator is first loaded with one data number, then the second number is added to the accumulator—a two-cycle operation. In addition, for this result to be used as an input for anything but another ALU operation, the data must first be transferred from the accumulator to data memory. These restrictions result in a severe arithmetic throughput penalty.

The MAC in the ADSP-2100 performs the complete multiply-and-add operation in one cycle; like the ALU, it has two X and two Y input registers. MAC operands may be loaded from any combination of program memory and data memory, or other processor registers; the MAC feedback and result registers can serve as operands for any MAC operation. Two new operands can be loaded into the input registers during a computation cycle; thus a new MAC operation, with new operands, can start with every cycle (even when accessing off-chip memory).

By comparison, some DSP architectures do not have a multiplier/accumulator as a single entity; the multiplier is separate from the ALU, and the ALU is used for MAC accumulations. A complete MAC operation thus requires two cycles: one to multiply and one to accumulate. The interdependency of ALU and multiplier means that MAC operations cannot be intermingled easily with ALU operations; the order of calculations for the final algorithm may have to be changed to avoid conflicts. In addition to requiring more system time, the instructions for the two-step process require more program memory, in contrast to the multifunction instruction of the ADSP-2100.

The barrel shifter of the ADSP-2100 accepts as input any result register in the processor, including its own result (or its own input register). The shifter can place a 16-bit input value anywhere

within a 32-bit field in a single cycle, and shift any number of input bits from off-scale right to off-scale left. Functions such as exponent detection, normalization and denormalization, and block floating-point manipulation can be realized via this shifter. All shifts, regardless of number of bits to be shifted, are performed in a single cycle.

## DATA ADDRESSING

Fast arithmetic is of course wasted if the required data cannot be fetched at a commensurate speed, regardless of source. To fully utilize the separate data and program memories of the “Harvard” architecture used in most DSPs (in contrast to the single interleaved program/data memory of the Von Neumann computer architecture), the data addressing must support simultaneous dual-operand fetches. The circular buffers often found in DSP algorithms are supported in some DSP processors via built-in address-pointer wraparound; processors lacking this wraparound are at a serious disadvantage in application effectiveness.

The ADSP-2100 has two separate address generators: one typically supplies addresses for data-fetch from program memory (PM) while the other supports data-memory (DM) data fetch. Each address generator’s multiple registers store pointers (addresses), address modifiers, and buffer lengths—for circular (modulo) addressing. For efficient FFT execution, the address generator can reverse (with zero overhead) the order of bits in an address as it is being sent out.

Both indirect and direct addressing are available. In indirect mode, the address *index* register is updated by the contents of a *modify* register, while being put on the bus. The pairing of the various base-address and modify registers is up to the programmer, useful for two-dimensional array addressing or for pointer increment/decrement. The 24-bit-wide instruction word (Figure 5) includes 4 two-bit fields to point to specific PM and DM address registers, plus their respective modify registers.

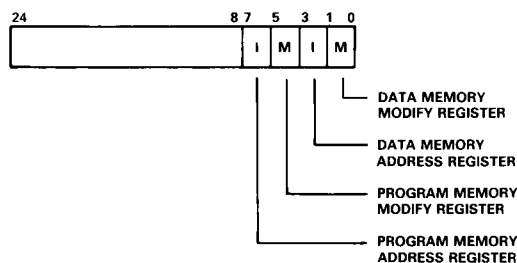


Figure 5. Indirect addressing portion of instruction word

A special *length* register is used for circular addressing. Loading a non-zero buffer length into this register automatically activates the modulus logic. The address and its modulus are maintained transparently by the address generator without explicit calculation by the programmer; and this internal calculation has zero overhead—like many other ADSP-2100 functions.

Some DSPs support both direct and indirect addressing, but with a very limited set of address and modifier registers. This limits flexibility in interleaving several indirect sequences for complex algorithms since the old modify value must be stored and a new one written before the new indirect mode is used. Similar constraints exist for-base-address switching.

## PROGRAM SEQUENCING

An efficient DSP for signal processing wastes little or no overhead in maintaining the desired control of data flow. Loops are funda-

mental to many signal processing algorithms (typified by the ubiquitous multiply-and-add operation). When a DSP program can be expressed in loop form, the coding is simplified and shortened; further, changes require less work (for example, changing the number of taps in the FIR filter). Equally critical, branching specifies conditions under which the program flow is redirected and program execution begins at a new point.

The ADSP-2100's program sequencer (Figure 6) selects the next address for the address bus from either the program counter (for sequential addressing), the instruction word itself (for direct jumps and subroutine calls), the program counter stack (for returns from subroutines and interrupts), or the interrupt logic (to vector to the interrupt routine when an external interrupt is asserted). All address selection and execution occurs in a single cycle; when an interrupt occurs all processor status registers are automatically pushed onto the status stack for later recall.

When address looping is used, it is automatic and transparent. Without any extra checking cycles, the processor determines if a loop should terminate (either because it has run the specified number of cycles or another termination condition is met), and it outputs the next instruction address; if loop iterations continue, the address of the first loop instruction is output. In one cycle the last instruction of the loop is executed, and on the very next cycle the next instruction is executed (either within the loop—or outside, when terminated).

To achieve speed, some other DSP architectures use a three-level pipeline (for instruction prefetch, decode, and actual execution) in the program sequencer. They also require an extra instruction to check for loop count or branching conditions. Any deviation from the sequential flow of instructions—such as for returning to the beginning of a loop or terminating the loop—requires that the pipeline be emptied and then refilled.

Besides making analysis of program flow complex (and bench-

marks hard to calculate), such approaches encourage straight-line, non-looping coding of algorithms. These are inefficient to program, inflexible, and consume more memory than loops (often by a factor of hundreds for larger data sets or matrices). A separate, explicit instruction to check loop count and branching adds one cycle of overhead for each iteration.

### RISC vs. CISC vs. DSP ARCHITECTURES

As central processor architectures matured, their instructions sets became "richer". The complex-instruction-set computer (CISC) includes instructions for basic processor operations, plus single instructions that are highly sophisticated—for example, to evaluate a high-order polynomial. But CISC has a price: many of the instructions execute via microcode in the CPU and require numerous clock cycles—plus silicon real estate for code storage.

In contrast, the reduced-instruction-set computer (RISC) recognizes that, in many applications, basic instructions such as LOAD and STORE—with simple addressing modes—are used much more frequently than the advanced instructions, and should not incur an execution penalty. These simpler instructions are "hard-wired" in the CPU logic to execute in a single clock cycle, reducing execution time and CPU complexity.

**RISC and DSP Applications:** Although the RISC approach offers many advantages in general purpose computing, it is not well-suited to DSP. For example, most RISCs do not support single-instruction multiplication, a very common and repetitive operation in DSP. The DSP is optimized to accomplish its task fast enough to be "real-time" in the context of the application, which requires single-cycle arithmetic operations and accumulations.

DSP algorithms have unique needs not found in general-purpose computing: circular buffering, pointer updating and fast looping with zero overhead, bit reversing, barrel shifting, scaling, and data-dependent execution branching. Each of these should execute within the DSP instruction, and not as a separate time-consuming instruction cycle. The computational unit within the DSP must be run efficiently, with data arriving from at least two separate data memories with no time penalty for data access. CISCs and RISCs support virtually none of these needs.

Software programming also differs. RISCs and CISCs are programmed in high-level languages (HLLs) to minimize software development time and hide the instruction set from the programmer. For real-time DSP applications, however, code optimization (primarily of execution time, but also of memory usage) requires that the software engineer use assembly language to get satisfactory performance. Critical sections of the program are examined and recoded if necessary, to reduce overall execution time, after simulation and run-time histograms.

In theory, *any processor can accomplish any software task, given enough time.* However, DSPs are optimized for the unique computational requirements of real-world signal processing, while CISCs and RISCs are better-suited for general-purpose calculations that can often be performed off-line. ▣

### REFERENCES

"Considerations for Selecting a DSP Processor," by Bob Fine.  
 "Comparison of CISC, RISC, and DSP Architectures," by David Fair.  
 Book, available from Analog Devices:  
*Digital Signal Processing in VLSI*, by Richard J. Higgins. Englewood Cliffs, NJ: Prentice Hall, 1990.

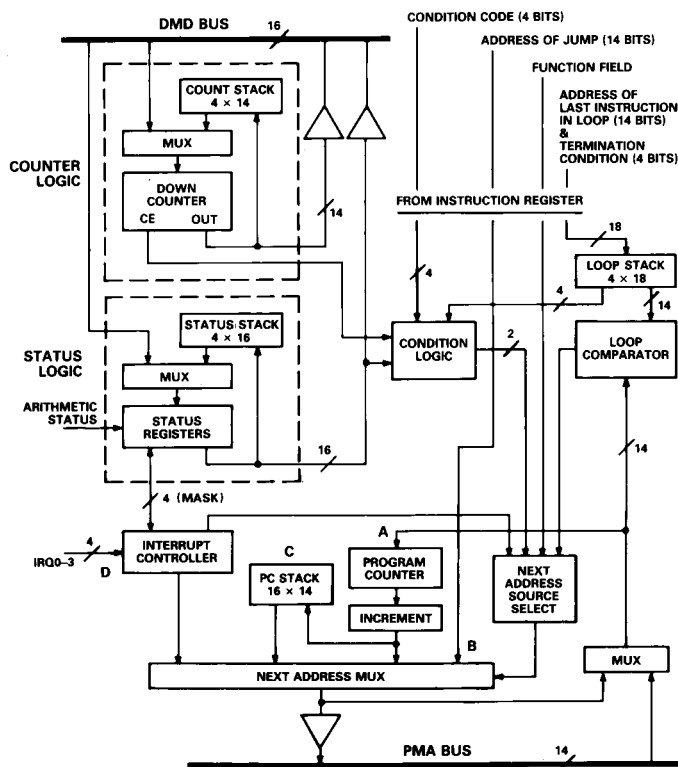


Figure 6. Program-sequencer architecture.