THE DSP32 DIGITAL SIGNAL PROCESSOR AND ITS APPLICATION DEVELOPMENT TOOLS

James R. Boddie, Renato N. Gadenz, Robert N. Kershaw, W. Patrick Hays, and James Tow

James R. Boddie *is head of the Signal Processing and Integrated Circuit Design Department at AT&T Bell Laboratories in Holmdel, New Jersey.* ***W. Patrick Hays*** *is a supervisor, and* ***Renato N. Gadenz*** *and* ***James Tow*** *are members of technical staff in that department.* ***Robert N. Kershaw*** *is a supervisor in the Digital IC Design Department at AT&T Bell Laboratories in Allentown, Pennsylvania. Mr. Boddie's department designs analog and digital signal processing integrated circuits and support systems. Mr. Boddie joined AT&T in 1977 and has a B.S.E.E. from Auburn University, an S.M. and E.E. from Massachusetts Institute of Technology (MIT), and a Ph.D. from Auburn University, all in electrical engineering. Mr. Gadenz, who designs and tests new VLSI circuits for digital signal processing, joined* (continued on page 104)

The WE® DSP32 digital signal processor is a high-speed, programmable, VLSI circuit with 32-bit floating-point arithmetic. The device can be used cost-effectively in a wide variety of complex digital signal processing applications, such as speech recognition, high-speed modems, low bit-rate voice coders, multichannel signaling systems, and signal processing workstations. In this paper, we review the architecture of the DSP32 and present its instruction set, including some examples. We also discuss the software and hardware support tools that are available for developing DSP32 applications.

An Expanding Family

The WE® DSP32 digital signal processor[1,2] is the latest member of a family of single chip, programmable digital signal processors developed at AT&T Bell Laboratories.

The first family member appeared in 1979 and was called simply DSP.[3] It was one of the first single-chip, programmable digital signal processors ever made.

As very-large-scale-integration (VLSI) technology advanced, the demand for the DSP justified its redesign. The new device—the WE DSP20—ran twice as fast, used less power, and cost less than the original DSP, yet was pin, architecture, and instruction set compatible.

DSP32 Features

Both the DSP and DSP20 are 20-bit, fixed-point processors and can execute instructions at the rate of 1.25 and 2.5 million instructions per second, respectively.

The DSP32, on the other hand, can execute 4 million instructions per second, operating on 32-bit, floating-point numbers. It also has a complete set of microprocessor instructions that operate on 16-bit, fixed-point numbers for ease of use in logic and control operations. In addition, the DSP32 offers both serial and parallel input/output (I/O) with direct-memory access (DMA) capability. (DMA allows data trans-

fers from an input buffer to memory or from memory to an output buffer, without program intervention.)

Like the previous DSPs, the DSP32 has an on-chip, random-access memory (RAM) for storing variable data, and an on-chip, mask-programmable read-only memory (ROM) for storing instructions and fixed data. Thus, the device can be customized for a wide variety of signal processing applications.

Although there are many different types of applications, most require real-time execution of repetitive multiplications and additions. Like its predecessors, the DSP32 is optimized for this type of computation. But it offers a new architecture that allows users to develop more complex applications requiring floating-point arithmetic.

The DSP32 can be cost-effective for a wide variety of digital signal processing applications, such as speech recognition, high-speed modems, low bit-rate voice coders, multichannel signaling systems, and signal processing workstations. Algorithms that are suited for floating point include matrix inversion; spectral analysis; high-quality, low bit-rate speech; graphics; and image processing.

Our goal in designing the DSP32 has been to balance high performance with ease of use. By high performance, we mean fast execution of signal processing algorithms using a high-quality arithmetic. A major contributor to this goal is the use of a 32-bit, floating-point-data arithmetic unit.

An 8-bit exponent in the 32-bit, floating-point word yields a large dynamic range that makes overflow unlikely, while a 24-bit normalized mantissa gives high precision, independent of magnitude. Large dynamic range and high precision are often essential in advanced algorithms.

Floating point also contributes to ease of use, because it frees programmers from concern about intermediate scaling to avoid overflow or loss of precision. Further, algorithms that are initially developed on mainframe computers or array processors and use floating-point arithmetic can be easily adapted to run on the DSP32.

Ease of use applies not only to programming or interfacing to the DSP32, but to the support tools that are available for developing DSP32 applications. We will discuss the architecture, instruction set, and support tools later.

The DSP32 can do the following, all with 32-bit, floating-point precision and dynamic range:

- a 1024-point, complex, Fast Fourier Transform (FFT) in 19.2 ms (including bit reversal)
- a finite impulse response filter in 250 ns per tap
- a second-order section (four multiply) for a recursive, infinite-impulse-response (IIR) filter in 1 μs.

The device is available in two packages: a 40-pin dual in-line package, and a 100-pin pin array that can use external memory to expand the on-chip memory. The chip is manufactured in 1.5-μm effective channel length, n-type metal-oxide semiconductor (NMOS) technology, and has about 155,000 transistors in an 81-mm² area (12.70 mm by 6.35 mm).

The DSP32 operates with a 16.384-MHz clock and a single 5V power supply. For the 40-pin device, typical power dissipation is 1.8W, while worst-case power dissipation is 2.3W. For the 100-pin package, typical power dissipation is 2.0W and worst case is 2.6W. The device is now being manufactured in high volumes.

Recently, the DSP32 has been manufactured with a 10-percent photolithographic shrink. Samples will soon be available from two different processes; one leads to higher speed devices and the other to lower power devices.

Its Architecture

The DSP32's architecture consists of several specialized sections (Figure 1) that work in parallel to achieve a high throughput. A 32-bit bus interconnects these sections, and control is distributed among them.

The device has two execution units—the *data arithmetic unit* (DAU) and the *control arithmetic unit* (CAU)—and its on-chip memory includes 2048 bytes of ROM and 4096 bytes of RAM. The memory can be addressed as 8-, 16-, or 32-bit words and is organized to

access 32-bit data at the same speed as 8-bit data.

A flexible serial I/O (SIO) unit allows direct interface to a codec, a time-division multiplex line, or another DSP32, while an 8-bit parallel I/O (PIO) unit allows bidirectional communication with a microprocessor.

Data Arithmetic Unit. The DAU is the primary execution unit for signal processing algorithms. As Figure 1 shows, it contains a 32-bit, floating-point multiplier; a 40-bit, floating-point adder; and four 40-bit accumulator registers (a0 through a3) that provide temporary storage, thus reducing memory accesses.

The DAU has a multiply-add structure; that is, the multiplier output is directly connected to the adder input. It does 4 million instructions per second of the form, `A = B + C * D`. In this instruction, we have both a floating-point multiplication and a floating-point addition, which yields a throughput of 8 million floating-point operations per second.

The DAU multiplier inputs are 32-bit, floating-point numbers with a 24-bit mantissa and 8-bit exponent.

Inputs to the multiplier and adder can come from memory, I/O registers, or an accumulator (a0–a3). The adder inputs from memory or I/O are 8, 16, or 32 bits wide, while those from the multiplier or an accumulator are 40-bits wide (an 8-bit exponent and a 32-bit mantissa that includes eight guard bits).

These 40 bits of precision are maintained in the adder and the accumulators; the eight guard bits of the mantissa allow extra precision in intermediate accumulations. However, the 40-bit result in an accumulator is truncated to 32-bits when written to memory or I/O, or provided as an input to the multiplier.

The DAU converts floating-point data to and from 16-bit integer data. It also converts floating-point data to and from μ-law and A-law companded data formats.

Control Arithmetic Unit. The CAU—a 16-bit, fixed-point unit—has a dual function. It generates and post modifies addresses for accessing operands in memory, and it executes microprocessor instructions on 16-bit data for logic and control operations. (Post modify means increment the address *after* completing the operation.)

As Figure 1 shows, the CAU has 21 16-bit general purpose registers (r1 through r21); a 16-bit program counter (pc); and a full-function, arithmetic logic unit (ALU). All CAU registers are static and do not require refreshing.

For instructions that are primarily executed in the DAU, registers r1 through r14 are used as memory pointers and registers r15 though r19 hold address increments. A *pointer register* contains the address of an operand in memory that is to be either read or written, and an *increment register* contains a number that is added to the pointer to alter it. This addition is done in the ALU.

Register r20, also called PIN (pointer-in), is the pointer for serial DMA input, and register r21, also called POUT (pointer-out), is the pointer for serial DMA output. These registers can also be used as general-purpose registers, but their effect on DMA operations must be considered.

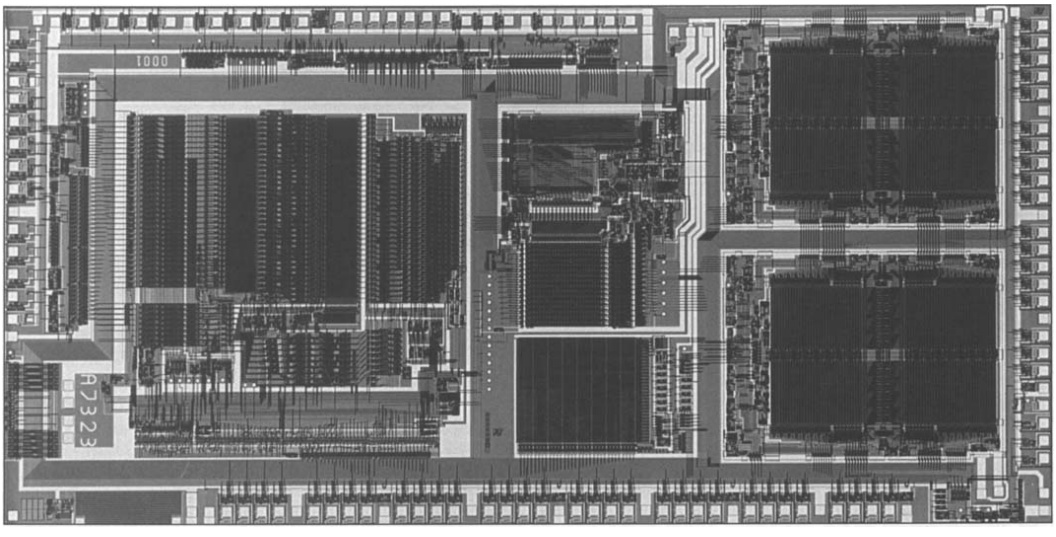
As stated above, the CAU also executes its own set of instructions. They implement two's complement, fixed-point arithmetic operations; logic functions; control operations like conditional branching; and data moves between memory, I/O, and the CAU registers. Executing logic and control operations in the CAU not only makes programming the device easier but also enhances the DSP32 performance.

Memory. The DSP32 provides on-chip memory (Figure 1) that includes a 512 by 32-bit ROM and a 1024 by 32-bit RAM that is divided into two equal 512 by 32-bit segments.

Data can be 8, 16, or 32 bits wide, and memory is uniformly byte addressable. This means that the four individual bytes and the two 16-bit words in each 32-bit word can be addressed independently.

Byte addressability is important because, besides using 32-bit instructions and floating-point data, the DSP32 uses 16-bit fixed-point integers and 8-bit companded (μ-law and A-law) data.

The RAM is dynamic and is refreshed either auto-



matically once every 32 instructions or under program control. The ROM is masked programmed.

For the 100-pin pin array package, an additional 56 kbytes of memory can be accessed externally. If standard byte-wide memory chips are used for expanding memory, no additional interfacing devices are needed. Also, if this external memory is fast enough (80-ns access time), there is no speed penalty when accessing it.

The DSP32 memory space (ROM, RAM, and external memory) is logically divided into two banks: *Lower bank 0*, which can be expanded with external memory, and *Upper bank 1*. Memory space can be configured in four different ways using two of the DSP32 pins.

To achieve maximum throughput, memory accesses must alternate between the two memory banks. As one memory bank is accessed, the other memory bank is being addressed. However, if the user chooses not to interleave, the DSP32 automatically inserts a wait state when two consecutive accesses occur to the same memory bank. This flexible memory accessing makes the DSP32 easier to program.

Instructions can be stored anywhere in the address space (ROM, RAM or external memory) and can be executed from any memory without a speed penalty, if interleaving between the two memory banks is used. Each instruction cycle consists of four machine states, and one read or write operation can occur during each state.

Serial I/O. The SIO is used for serial-to-parallel conversion of input data and parallel-to-serial conversion of output data. Serial input and output operations are independent and asynchronous with respect to each other and to program execution. The SIO control signals allow direct interface to a codec, a time-division multiplexed line, or another DSP32.

Input to the SIO (Figure 1) is loaded into the input shift register (`isr`), and then into the input buffer (`ibuf`). SIO outputs are loaded into the output buffer (`obuf`), and then put into the output shift register (`osr`). This double buffering permits a second serial transmission to begin before the first transmission has been processed.

Data widths can be 8, 16, or 32 bits. The I/O control register `ioc` in the SIO is used to select various I/O conditions, bit lengths, internal or external clocks, and internal or external synchronization signal, thus allowing flexibility when interfacing to external hardware.

For example, in the active mode, the DSP32 generates the I/O clocks and the synchronization signal for external hardware. In the passive mode, the DSP32 acts as a slave and the external hardware generates its clocks and synchronization signal.

SIO transfers could occur under program control or using direct memory access. To enable DMA, the user

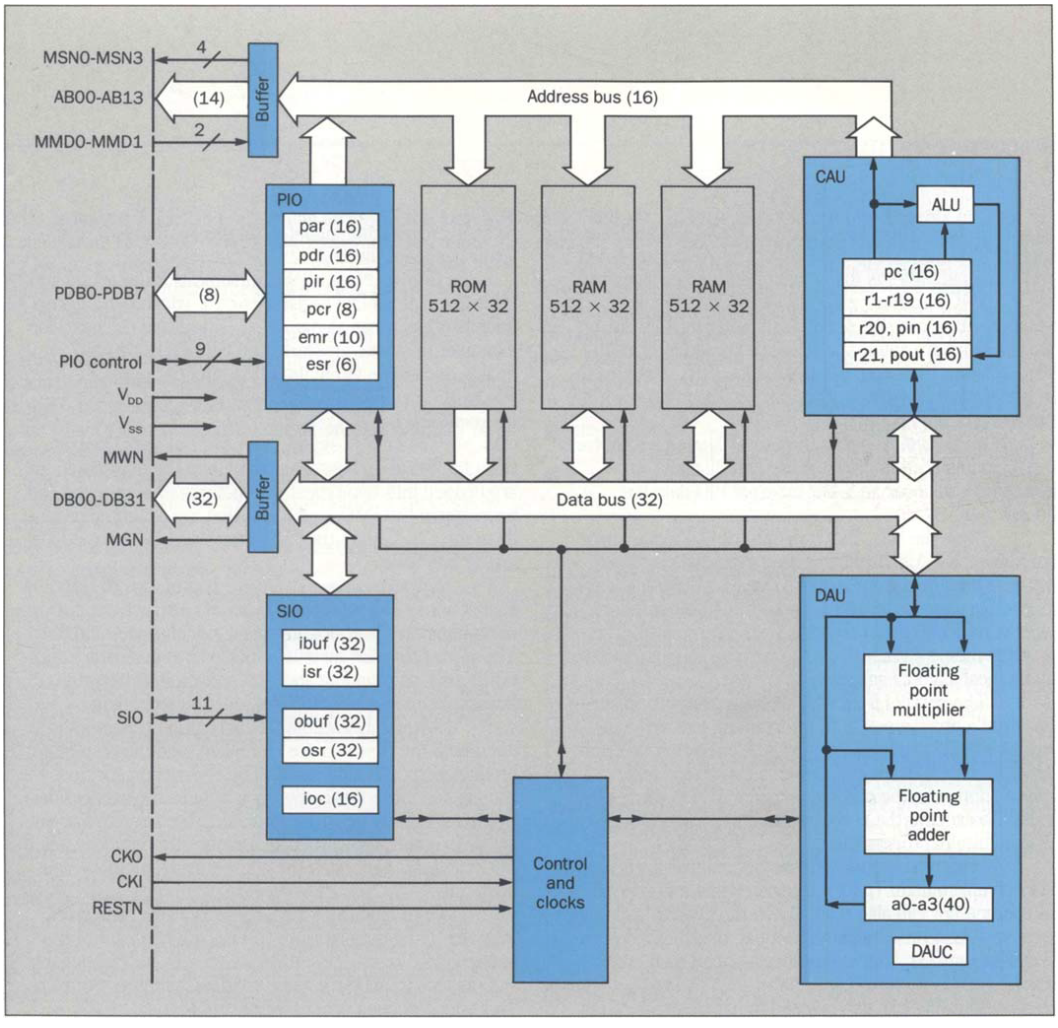


Figure 1. Block diagram of the DSP32 digital signal processor. The numbers in parentheses designate the size of a particular register or bus. The symbols at the left represent pins on the device package.

sets the ioc register appropriately. If DMA is enabled, an implicit DMA request to the DSP32 control occurs whenever the input buffer is full or the output buffer is empty.

CAU registers r20 (PIN) and r21 (POUT), which the user can set, serve as pointers for the DMA transfers and are automatically incremented after each DMA access. The serial I/O DMA allows data samples to be transferred in and out of memory without interrupting execution of the DSP32 program.

Parallel I/O. The PIO (Figure 1) is used for bidirectional communication between the DSP32 and a microprocessor over an 8-bit, external PIO data bus (PDB).

As in the SIO, PIO transfers can be made under program or DMA control. In either case, data is transferred in both directions through the 16-bit, parallel data register (pdr). In the DMA mode, the 16-bit parallel address register (par) contains the DMA pointer. The microprocessor initializes par, which can be incremented automatically after each memory access.

The parallel I/O DMA allows a microprocessor to download a program without interrupting execution of another DSP32 program. (Each DMA operation *steals* one instruction cycle.) The microprocessor can then alter a *branch* address in the original program, which causes the DSP32 to execute the new program. Thus, the DSP32 can be dynamically programmed.

Program outputs can also occur through the parallel interrupt register (pir), a register that an external microprocessor can also read. When the DSP32 loads this register, it also raises an interrupt flag that the microprocessor recognizes. The microprocessor can then take action.

The PIO can also be used to handle error conditions, such as floating-point overflow or underflow in the DAU, or loss of external synchronization signal. When an error occurs, a bit is set in the 6-bit, error source register (esr) that is readable by an external microprocessor. The 10-bit, error mask register (emr), which is written by the microprocessor, conditions the DSP32 to ignore the error or send an interrupt signal to the microprocessor. If the latter happens, the DSP32 could also halt itself.

Control of parallel communications is set up in the 8-bit, parallel control register (pcr), which the external microprocessor can also access. This allows the user, for example, to start or restart the DSP32 (after a halt), or enable or disable automatic refreshing of the RAM.

Instruction Set

The DSP32 supports a powerful set of instructions for signal processing algorithms. These instructions are divided into two types: instructions that are executed primarily in the DAU, and those that are executed primarily in the CAU. After they are assembled, all instructions are 32 bits wide.

DAU Instructions. There are two groups of DAU instructions. The *multiply/accumulate* instructions perform operations on 32-bit floating-point signal processing data. The *special function* instructions perform nonlinear operations, such as rounding and data conversions between floating-point and μ-law, A-law, or integer formats.

A multiply/accumulate instruction requires three operands; the first two are multiplied, and the resulting product is added to the third. The instruction specifies the sources for these three operands (where the data resides). One source must be an accumulator; the other two operands can come from either an accumulator, memory, or I/O. The CAU generates the addresses for memory operands.

Operands from memory or I/O are transferred over the 32-bit data bus into the DAU. The result of the multiply/add operation is always stored in an accumulator and, as an option, can also be written to memory or I/O. This transfer again occurs over the 32-bit data bus.

The formats for DAU multiply/accumulate instructions are

```
[Z =] aN = [−]aM {+,−} Y * X

aN = [−]aM {+,−} (Z = Y) * X
```

In these instructions, aN and aM can be any of the four accumulators (a0–a3). The X and Y operands are memory locations, serial I/O input buffer, or an accumulator. The Z operand is either a memory location or the serial I/O output buffer.

Also, the [ ] and { } are **not** part of the assembly language syntax. Values enclosed in brackets [ ] are optional, and one of the values enclosed in braces { } must be used. For example, "Z =" is in brackets because the result of a multiply/accumulate operation can optionally be written to memory.

The second format is similar to the first, except the Y operand is written to the location that Z specifies, in addition to being operated on by the DAU. This is especially useful for memory-to-memory moves for tap update in finite-impulse-response filters.

The format for special function instructions is

```
[Z =] aN = function(Y)
```

Here, the specified function operates on the value Y, and the result is placed in the accumulator aN. As an option, the result can then be written to the location that Z specifies.

We will give examples of these formats shortly.

The DSP32 instruction syntax is similar to the C programming language and makes the programmer's code almost self-documenting. An asterisk indicates multiplication, but an asterisk preceding a pointer (e.g., *r7) means *memory location pointed to* by the pointer. An equal sign is assignment, a plus sign means addition, and the *plus plus* (++) that follows a pointer indicates post increment (increment the pointer after reading from or writing to memory). Here, pointers are always CAU registers.

Consider the DAU multiply/accumulate instruction

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

that reads from right to left as follows. Multiply the contents of the memory locations pointed to by CAU registers r10 and r7, and add the result to the contents of accumulator a0. Then, store this result in accumulator a1 and the memory location pointed to by CAU register r5. Also, post increment the contents of register r10 using the contents of register r17, and post increment the contents of register r5 by one. (In reality, this is an increment by *one* 32-bit word address, which is equivalent to an increment by *four* byte addresses.)

Clearly, much computation is represented in a single instruction.

An example of a DAU special function instruction is

```
a0 = float(ibuf)
```

In this example, the contents of the input buffer ibuf (assume that the serial input word is 16 bits wide) are converted to DSP32 32-bit, floating-point format, and the result is stored in accumulator a0.

CAU Instructions. There are three groups of CAU instructions: *arithmetic and logic*, *data move*, and *control*. The CAU uses two's-complement, 16-bit fixed-point arithmetic.

Except for a register load from memory, execution of a CAU instruction is completed before execution of the next instruction begins. This feature greatly simplifies using the CAU for logic and control operations.

Here are the formats for the three groups of CAU instructions:

```
rD = rD op {rS,N}

{MEM, I/O} = rD

        or

rD = {MEM, I/O}

if (COND) goto rH, N, {rH+N, rH-N}
```

The first format represents the arithmetic and logic group. These instructions do 16-bit fixed-point integer arithmetic, such as addition and subtraction, or logic operations, such as `AND, OR, XOR`, and `shift`. The values of `rD` and `rS` are the contents of any of the 21 16-bit CAU registers (`r0`–`r20`), and `N` is a 16-bit integer.

The second format represents the data move group. These instructions transfer data between the CAU registers and memory, or between the registers and serial or parallel I/O. In this form of instruction, `MEM` is an 8- or 16-bit memory location that is specified by a direct address or a CAU register. Data can also be moved to or from the serial or parallel I/O.

The third format represents the control group, whose instructions alter the program counter to change the sequence of program execution. The DSP32 can branch on many conditions (`if ... goto`), including CAU, DAU, and I/O conditions. The control instructions also include an unconditional branch (`goto`), and a call to and return from a subroutine.

The values of `rD, rS`, and `rH` are the contents of any of the 21 16-bit CAU registers. However, `rH` can also be the contents of the program counter (`pc`). As before, `N` is a 16-bit integer.

Examples of CAU instructions are:

```
r10 = r10 + r16

*r6++ = r09

if (eq) goto r02 + 12
```

The first instruction belongs to the arithmetic and logic group. It does a logical `AND` of the contents of CAU registers `r16` and `r10` and places the result back into `r10`.

The second instruction belongs to the data move group. It takes the contents of CAU register `r09` and writes it to the 16-bit memory location pointed to by CAU register `r6`. Then, register `r6` is post incremented so that it points to the next 16-bit integer location in memory (an increment of one 16-bit word, or two bytes).

The last instruction, an example of the control group, illustrates a conditional branch. If the result of the last CAU operation equaled zero, then go to the memory location specified by the contents of CAU register `r02` plus 12.

Appendixes A and B present two examples of DSP32 programs.

Development Support Tools

As we mentioned, ease of use requires the availability of a good set of tools that allow a user to translate algorithms into working applications.

The DSP32 software library, DSP32-SL, is written in the C programming language and resides in a host computer that runs the UNIX® operating system. Software tools[4] include an assembler, linker/loader, and simulator. The hardware development system, DSP32-DS,[4] allows real-time program debugging and in-circuit emulation.

Software Tools. The *assembler* translates the user's assembly language program into the binary code that DSP32 uses for instructions. A notable feature of this assembler that distinguishes it from typical microprocessor and other DSP assemblers is its use of the high-level syntax just presented.

The assembler generates relocatable code that another software tool, the *linker/loader*, can easily alter. The relocatable code can reside anywhere in the addressable space and can be combined with code that was assembled separately.

The *simulator* is a program that simulates the operations of the DSP32 program in a nonreal-time environment. For full program debugging, the simulator allows access to all registers and memories. It also provides an interface to the DSP32 hardware development system.

The simulator provides precise simulation down to the timing of synchronous I/O. When running on the AT&T 3B2/300 computer, it executes about 750 instructions per second.

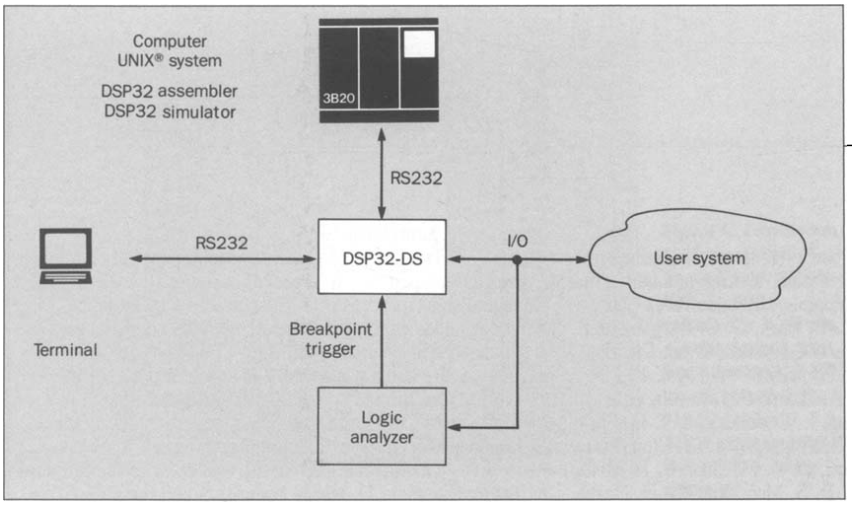


Figure 2. The DSP32 development environment. At the center is the hardware development system, DSP32-DS. The assembler and simulator are part of the software library, DSP32-SL.

In the simulator, the user can freeze processing on many conditions, such as the execution of a specified instruction, access of a specified register or memory location, or occurrence of a specified number of I/O events. Input data can be supplied by a file, while output data can be captured in a file. The user can refer to memory locations by their symbolic names rather than their absolute addresses. In addition, the user can define complex command sequences and display formats, and invoke them with a simple command.

Hardware Tools. The DSP32 hardware development system consists of a small circuit board that contains a DSP32, 14k words of external memory, a control microprocessor, and an RS232C interface. It can be used standalone or as an in-circuit emulator to the user's system.

The DSP32-DS can run DSP32 programs in an environment where its input and output are connected to the user's DSP32 system. The development system also retains many of the simulator debugging features.

Figure 2 depicts a typical DSP32 software and hardware development environment. Normally, the DSP32-DS is connected to a standard computer terminal and a host computer that runs the DSP32 simulator program.

With a special simulator command, the user's application program can be loaded and run using the development system, instead of the simulator. The user can switch back and forth between the simulator and the development system to compare results.

For multiple DSP32 applications, up to seven DSP32-DSs can be connected to a single computer and terminal system. The user can selectively communicate with each system or the simulator program. One application for the multiple DSP32-DS configuration is the use of one or more DSP32-DSs as digital signal generators or signal analyzers for testing a program in another DSP32-DS.

With the appropriate application software, a DSP32-DS can become a powerful digital signal processing workstation that consists of a flexible set of programmable function generators, filters, power meters, detectors, and spectrum analyzers.

Conclusion

We have presented the DSP32—a 32-bit, floating-point, programmable, digital signal processor—and its support tools for developing applications.

The high performance of the device, coupled with its ease of use, make it suitable for a variety of advanced signal processing applications in telecommunications, speech, imaging, and graphics. The device can also be used as a 16-bit fixed-point microcomputer.

The flexible on-chip memory and I/O—the latter with DMA capability on both serial and parallel interfaces—provide the ability to program the DSP32 dynamically and interact easily with the user's system.

Acknowledgments

We gratefully acknowledge the contributions of the following people to the design, testing, and fabrication of the device and the development of its software and hardware support tools: L. E. Bays, G. G. Burgstresser, R. J. Canniff, J. D. Cuthbert, C-F. Chen, D. B. Cuttriss, J. C. Desko, B. J. Dutt, D. J. Emrich, C. J. Faust, E. M. Fields, R. L. Freyman, J. A. Grant, C. J. Garen, G. E. Hall, D. J. Hart, J. Hartung, J. W. Hendricks, T. D. Hopmann, W. C. Ingram, P. W. Kempsey, J. J. Klinikowski, D. F. Koehler, R. D. Lippincott, D. A. McGillis, C. R. Miller, K. Mondal, H. O. Morris, H. S. Moscovitz, H. N. Nham, G. D. O'Donnell, A. A. Pignone, Y. Rotblum, M. A. Steele, P. A. Stiling, P. Subramaniam, W. A. Stocker, T. G. Szymanski, L. V. Tran, C. J. Van Wyk, A. J. White, D. Willauer, W. Witscher, Jr., D. S. Yaney and E. J. Yurek.

References

98

1. R. N. Kershaw, et al., "A Programmable Digital Signal Processor with 32b Floating Point Arithmetic," *Digest of Technical Papers*, 1985 ISSCC, February 13-15, 1985, pp. 92-93.
2. W. P. Hays, et al., "A 32-bit VLSI Digital Signal Processor," *IEEE Journal of Solid-State Circuits*, Vol. SC-20, No. 5, October 1985, pp. 998-1004.
3. Digital Signal Processor, *The Bell System Technical Journal*, Vol. 60, No. 7, Part 2, September 1981.
4. *WE® DSP32 Digital Signal Processor—Information Manual*, AT&T Technologies, Inc., 1986.
5. L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, N. J., 1975, p. 367.
6. G. Szentirmai, "FILSYN: A General Purpose Filter Synthesis Program," *Proceedings of the IEEE*, Vol. 65, October 1977, pp. 1443-1458.

Appendix A. IIR Filter Program

This example shows how to implement an infinite-impulse-response (IIR) filter.[5] It assumes that the DSP32 digital signal processor is connected to a 16-bit linear codec and uses four-coefficient, second-order, IIR sections connected in cascade. With the DSP32 floating-point capability, there usually is no need to consider pole-zero pairing, ordering of sections, or gain distribution associated with a fixed-point arithmetic implementation (which would also require five-multiply sections).

The line numbers in the text refer to the program listing (Example 1), where comments and the `#define` feature of the C preprocessor (lines 1 to 8) are used to enhance program readability. Line 9, an assembler directive, defines global variables that the DSP32 simulator and development system can reference as symbols.

Line 10 programs the serial input/output for 16-bit transfers and uses the DSP32 on-chip clocks to drive an external 8-kHz sampling rate, analog to digital and digital to analog converter chip.

Line 11 shows a wait for the arrival of the serial input sample, indicated by a full input buffer. If `ibe` (input buffer empty) is true, the program will stay in the loop, lines 11 and 12. (All DSP32 branching instructions are *delayed branches*; i.e., the instruction that follows any branch instruction will always be executed before the branch occurs.) Line 12 sets up register `r10` as a loop counter for use in line 21.

Reading of `ibuf`, line 13, will empty the input buffer. Lines 18, 19, 20, and 22 do the four multiply and accumulate operations for the direct-form implementation of a second-order IIR section (Figure A).

Execution of a DSP32 instruction proceeds from right to left. For example, line 19 first forms the product of `Ai2` (the coefficient of the denominator z^{-2} term) with the appropriate state variable (S_{i2} in Figure A) and decrements the state variable pointer. Next, it forms a sum in `a0`, stores the sum as an updated state variable (pointed to by `r13`), and then increments `r13`.

The updated state variable value will be used in

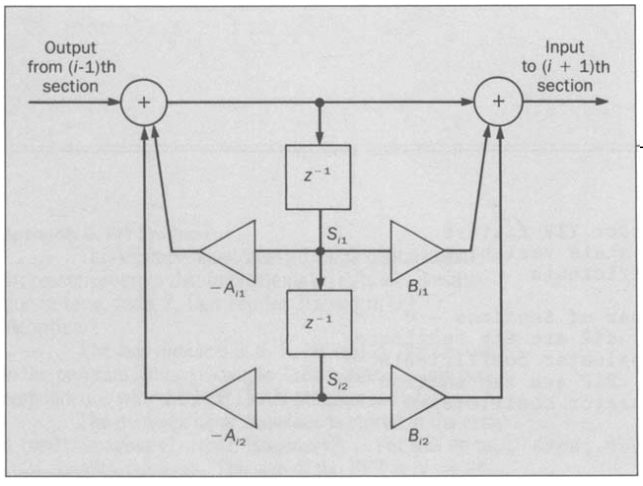


Figure A. Direct-form implementation of a second-order IIR section. A_{in} and B_{in} (where n = 1, 2) are section denominator and numerator coefficients, respectively. The S_{in} (where n = 1, 2) are state variables, and z^{-1} is a delay element.

the next sample, because DAU memory write operations become effective only after a four-instruction delay. By contrast, accumulator updates can be accessed immediately by the adder.

On line 21, the CAU conditional branch instruction ensures that the repeat-loop goes through each section once per sample (here, nine sections). The branch occurs if the content of register `r10` is greater than or equal to zero. Then, the content of the register is automatically decremented by one.

Line 23 instructs the DSP32 to go back to process the next sample value. Line 24 converts the 32-bit, floating-point value back to a 16-bit integer for output via the output buffer (`obuf`). Because of the delayed branching, line 24 will always be executed after the execution of line 23.

We obtained the IIR filter coefficients (lines 25 to 34) directly from a computer program based on Reference 6. The filter design corresponds to a 0.01-dB ripple in the passband (950 to 1050 Hz) and 100-dB stopband loss below 900 Hz and above 1100 Hz. We can easily meet these stringent specifications with the DSP32.

```
 1 /*  Example 1 -- Cascade 2nd order IIR filters
 2       r11 & r13 point to data (state variables)
 3       r12 points to filter coefficients             */

 4 #define Nminus2 7       /* Number of Sections - 2   */
 5 #define Ai1     *r12++  /* Ai1, Ai2 are ith section */
 6 #define Ai2     *r12++  /* denominator coefficients */
 7 #define Bi1     *r12++  /* Bi1, Bi2 are ith section */
 8 #define Bi2     *r12++  /* numerator coefficients   */

 9 .global sample, repeat, end, coef, data

10          ioc=0x0987
11 sample:  if (ibe) goto sample
12          r10 = Nminus2  /* nine-section IIR filter  */

13          a0 = float(ibuf)     /* convert to 32-bit  */
14          r11 = data          /* initialize pointers  */
15          r13 = data
16          r12 = coef
100 17    a0 = a0 * *r12++  /* (input) x (constant multi.) */

18 repeat:  a0 = a0 - *r11++ * Ai1
19          *r13++ =   a0 = a0 - *r11-- * Ai2
20          a0 = a0 + (*r13++ = *r11++) * Bi1
21          if (r10-- >=0) goto repeat
22          a0 = a0 + *r11++ * Bi2

23       goto sample
24 end:  obuf = a0 = int(a0)  /*convert back to 16-bit */

25 coef: float  9.5124534e-7    /* constant multiplier */
26       float  -1.3898448, 0.96402235, -1.5164367, 1.0
27       float  -1.3689732, 0.96678633, -1.2360012, 1.0
28       float  -1.4146495, 0.96782737, -1.5582712, 1.0
29       float  -1.3555562, 0.97476591, -1.2972484, 1.0
30       float  -1.4379917, 0.97618034, -1.2820803, 1.0
31       float  -1.3498068, 0.98484090, -1.6516234, 1.0
32       float  -1.4561744, 0.98593301, -1.5273664, 1.0
33       float  -1.3505858, 0.99500381, -1.0599482, 1.0
34       float  -1.4683514, 0.99540254, 0., -1.0

35 data: 30*float 0.0      /* initialize data to zero  */
```

Appendix B. FFT Program

This example illustrates a DSP32 digital signal processor program that implements an in-place, decimation-in-time, radix 2, Fast Fourier Transform (FFT) algorithm.[5]

The *loop numbers* 5, 6, 7, 10, and 20 that we use in the program listing (Example 2) have a one-to-one correspondence with the FORTRAN program in Reference 5.

The complex input sequence is stored in the array `A` (real1, imaginary1, real2, imaginary2, ...) of size up to 1024 complex numbers. The size of the FFT is $N = 2^M$, where N and M are declared by the user at assembly time (`$N, $M`). Here, N is 256.

If we exclude storage for the array `A`, the program requires 328 bytes of memory. The computation time required for a complex FFT with 128, 256, 512, and 1024 points (including bit reversal) is 2.0, 4.3, 9.0, and 19.2 ms, respectively.


```
/* Example 2 -- In place radix 2 FFT */

#define i r1
#define j r2
#define cnt r5
#define k r3
#define t r4
#define w r4
#define jcnt r6
#define lcnt r7
#define le1 r8
#define leo r9
#define lx r10
#define iz r11
#define kz r12
#define lem r15
#define ur a2
#define ui a3
#define ONE *r13++
#define ZERO *r13--
#define TWO a1
#define ap r14
```

```
        $N = 256        /* assembly-time variable */
        $M = 8          /* assembly-time variable */
        $NM3 = $N - 3
        $N4 = $N*4
        $MM2 = $M-2

/* In place bit reversal ----------------------------- */

        j=A
        i=A
        cnt=$NM3

loop0:  i-j
        if (ge) goto loop5
        nop

        *i++ = a0 = *j++     /* Begin complex exchange */
        *i-- = a0 = *j--     /*    T = A(J)   */
        nop
        *j++ = a0 = *i++     /*  A(J) = A(I)  */
        *j-- = a0 = *i--     /*  A(I) = A(J)  */

loop5:  k=$N4
loop6:  t=k+(A-8)
        t-j
        if (ge) goto loop7
        nop

        j=j-k
        goto loop6
        k=k/2

loop7:  j=j+k
        if (cnt-- >= 0) goto loop0
        i=i+8

/* Begin FFT calculation ----------------------------- */

        lcnt=$MM2
        lx=$N
        leo=4
        le1=1
        w=W
        r13=two
        TWO = *r13++
```

102

```
loop20: i=j
        iz=i
        k=i
        k=k+leo
        kz=k
        cnt=lx+(-2)

/* Butterfly ---------------------------------------- */

loop10: a0 = *i  + ur * *k++        /* T=A(i)r+Ur*A(k)r */
        *iz++=a0=a0-ui * *k--     /* A(i)r=T=T-Ui*A(k)i */
        *kz++=a0=-a0+TWO * *i++     /* A(k)r=-T+2*A(i)r */
        a0 = *i  + ui * *k++        /* T=A(i)i+Ui*A(k)r */
   *iz++lem=a0=a0+ur* *k++lem     /* A(i)i=T=T+Ur*A(k)i */
        if (cnt-- >= 0) goto loop10
   *kz++lem=a0=-a0+TWO* *i++lem     /* A(k)i=-T+2*A(i)i */

        a0 = ur * *w++           /* Compute new twiddle */
        ur = a0 - ui * *w          /*   U = U * W       */
        a0 = ur * *w--
        ui = a0 + ui * *w
        if (jcnt-- >= 0) goto loop20
        j=j+8

        if (lcnt-- >= 0) goto loop30
        w=w+8

/* Constant table ----------------------------------- */

two:    float 2.0, 1.0, 0.0

/* Twiddle table ------------------------------------ */

W:float -1.0000000, 0.0000000     /* cos(pi/1),-sin(pi/1) */
  float  0.0000000, -1.0000000    /* cos(pi/2),-sin(pi/2) */
  float  0.7071068, -0.7071068    /* cos(pi/4),-sin(pi/4) */
  float  0.9238795, -0.3826834    /* cos(pi/8),-sin(pi/8) */
  float  0.9807853, -0.1950903    /*cos(pi/16),-sin(pi/16)*/
  float  0.9951847, -0.0980171    /*cos(pi/32),-sin(pi/32)*/
  float 0.9987955, -4.9067674e-2   /* cos and -sin(pi/64) */
  float 0.9996988, -2.4541228e-2 /* cos and -sin(pi/128) */
  float 0.9999247, -1.2271538e-2 /* cos and -sin(pi/256) */
  float 0.9999812, -6.1358846e-3 /* cos and -sin(pi/512) */

A:                  /* Complex input array stores here */
```

Biographies (continued)

AT&T in 1973. He has an Ingeniero Civil Electricista degree from Universidad de Chile in Santiago, Chile, an M.S.E.E. from the University of Pittsburgh (Pennsylvania), and a Ph.D. in electrical engineering from the University of California at Los Angeles. Mr. Hays was head architect and supervisor responsible for the DSP32 and is now technical manager of a new venture called Array Processor Systems that he launched to design circuit boards and subsystems based on DSP32. He joined AT&T in 1980 and has an A.B. from Harvard College and a Ph.D. from MIT, both in physics. Mr. Kershaw, who joined AT&T in 1963, is responsible for a group that designs integrated circuits. He has a B.S.E.E. from Lafayette College and an M.S.E.E. from Lehigh University. Mr. Tow, who joined AT&T in 1966, is responsible for applications of digital signal processing. He has a B.S.E.E., M.S.E.E., and Ph.D. in electrical engineering from the University of California at Berkeley.

(Manuscript received May 13, 1986)