The TMS320C30 Floating-Point Digital Signal Processor

APPLICATION REPORT: SPRA397

Panos Papamichalis
Ray Simar, Jr.
Digital Signal Processor Products
Semiconductor Group
Texas Instruments

Digital Signal Processing Solutions

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain application using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

TRADEMARKS

TI is a trademark of Texas Instruments Incorporated.

Other brands and names are the property of their respective owners.

CONTACT INFORMATION

US TMS320 HOTLINE	(281) 274-2320
US TMS320 FAX	(281) 274-2324
US TMS320 BBS	(281) 274-2323
US TMS320 email	dsph@ti.com

The TMS320C30 Floating-Point Digital Signal Processor

Abstract

This chapter, reprinted from *Proceedings of the IEEE*, describes the origin and development of the TMS320 Family of Digital Signal Processors. The topics covered include:

- An overview of the characteristics of digital signal processing
- A history describing how digital signal processing has evolved over the last several decades
- A description of the three generations of the TMS320 family
- Hardware and software tools used in development and support
- How applications use DSP solutions

Support graphics include:

- An example of the building blocks comprising the TMS320 DSP family
- A graph showing the instruction cycles available for signal processing
- A diagram showing a minimal processing system with external data RAM and PROM/EPROM
- TMS320C10, TMS320C25, TMS320C30 functional block diagrams

The chapter concludes with a summary and a lengthy list of references.

Product Support

World Wide Web

Our World Wide Web site at www.ti.com contains the most up to date product information, revisions, and additions. Users registering with TI&ME can build custom information pages and receive new product updates automatically via email.

Email

For technical issues or clarification on switching products, please send a detailed email to (dsph@ti.com). Questions receive prompt attention and are usually answered within one business day.

The TMS320C30 Floating-Point Digital Signal Processor

Digital signal processors have significantly impacted the way we bring real-time implementations of sophisticated DSP algorithms to life. What was once only a laboratory curiosity that required large computers or specialized, bulky, and expensive hardware is now incorporated into low-cost consumer products. The rapid advancement of programmable DSPs since their commercial introduction in the early 1980s lets us satisfy the needs of very demanding applications. Implementation of basic DSP functions, such as digital filters and fast Fourier transforms, has been integrated into advanced system solutions involving speech algorithms, image processing, and control applications. The variety of the applications increases every day as researchers, developers, and entrepreneurs discover new areas in which DSP devices can be used. At the same time, the design of new devices incorporates features that make such implementations easier.

The Texas Instruments family of TMS320 DSPs[1] evolved with the expanding needs of the DSP applications and currently encompasses over 17 devices. The TMS320 family consists of three generations of devices. The first two generations are 16-bit, fixed-point-arithmetic devices while the third one, represented by the TMS320C30 and explained in detail here, is a 32-bit, floating-point device. Architecturally, the TMS320 family, like most DSP devices, relies on multiple Harvard buses. In the first two generations, we expanded the basic Harvard architecture to permit communication between the program and data spaces. In the third generation, we unified the two spaces to form an organization that encompasses the advantages of both the Harvard and the von Neumann architectures.

Overview of the TMS320C30

The 320C30 is a fast processor (16.7 million instructions per second for an instruction cycle time of 60 nanoseconds) with a large memory space (16 million 32-bit words) and floating-point-arithmetic capabilities. This last feature is a major trend in new DSP devices, which was developed to answer the need for quicker, more accurate solutions to numerical problems. DSP algorithms, being very intensive numerically, cause a designer to worry about overflows and the accuracy of results. The introduction of floating-point capabilities eliminates these difficulties.

Panos Papamichalis
Ray Simar, Jr.

Texas Instruments

Vol. 8, No. 6, pp. 10-28; December 1986

In the 320C30, a chip design with 1-μm geometries produces instruction cycle times lower than those achieved with the fixed-point devices of the first two generations. In addition, the design produces a controlled increase in die size that results more from the extended on-chip memory spaces than from the floating-point capabilities.

The pipelined architecture of the 320C30 permits the higher throughput achieved by the device, as we explain later. Yet, programmers do not have to worry about the pipeline when writing the code. We can describe the design philosophy of the 320C30 (as well as all the other devices in the TMS320 family) as an "interlocked" or "hidden-pipeline" approach. When writing the program, programmers can assume that the result of any instruction will be available for the next instruction. Most of the instructions execute in one machine cycle. If a conflict arises between executing an instruction in one cycle and having the data available for the next instruction, the device automatically inserts the necessary delay to eliminate the conflict. Since this delay could result in loss of performance, we provide development tools that identify where such conflicts occur. With this data, programmers can rearrange and optimize code.

Many applications, such as graphics and image processing, are difficult to implement on the earlier DSP devices because they require a large memory space. To satisfy this need, the 320C30 provides a total memory space of 16 million 32-bit words, memory several orders of magnitude larger than the fixed-point devices. Furthermore, it contains significantly increased on-chip memory: six thousand 32-bit words of RAM and ROM. The desire to have a device capable of offering system-level solutions to the implemented algorithms guided the design decision to increase on-chip memory. In other words, the 320C30 attempts to offer the capability of implementing an algorithm with as little peripheral circuitry as possible.

Along the same lines, the 320C30 contains a peripheral bus on which on-chip peripherals can be attached using a memory-mapped approach. Currently available peripherals include two serial ports, two timers, and a DMA controller. The modularity of the design permits easy change, addition, or deletion of peripherals to accommodate different needs. For instance, if a μ-law-to-linear format converter or a gate array is more important than one of the timers for certain applications, a user can make the change without impacting the core of the device.

As the power of the DSP devices increases, so does the sophistication of the algorithms that are implemented. The implication is that constructing and debugging an algorithm at the assembly-language level becomes a more and more tedious task. To address that problem, we provide the 320C30 development tools, which include a high-level-language compiler and a DSP operating system. The extended memory space, the software stack, and the large on-chip register file also facilitate such a development. We've already introduced a C compiler and announced an Ada compiler. We expect compiler availability to change significantly the way DSP algorithms are ported to DSP devices. With these tools, programmers can develop the algorithms on large computers, requiring at the most only selective optimization when they incorporate the algorithm on the 320C30.

Here, we describe the 320C30 architecture in detail, discussing both the internal organization of the device and the external interfaces. We also explain the pipeline structure, addressing software-related issues and constructs, and examine the development tools and support. Finally, we present examples of applications.

Architecture of the 320C30

Studying the architecture of the device helps in understanding how the different components contribute toward a high-throughput system. The interaction and the efficient use of the parts can contribute to very effective programming. Another very important aspect to consider is the system cost of the application. We designed the device to incorporate on-chip features that minimize the amount and the cost of external logic, thus leading to very compact and cost-effective solutions. These advantages become explicit when looking at the architecture in detail. The internal structure of the 320C30, as shown in Figure 1, consists of the

- on-chip memory and cache,
- CPU with register file,
- peripheral bus and peripherals, and
- interconnecting buses.

See Figure 2 for the die photograph. To interface with the external world, the 320C30 provides pins corresponding to

- two buses (primary and expansion),
- two serial ports and two timers,
- four external interrupt signals,
- two external flags, and
- hold and hold-acknowledge signals.

In addition, other pins exist for address and data strobs, power, and so on.

The overall architecture of the device is a Harvard type in the sense that internally and externally it has multiple buses to access program instructions, data, or perform DMA transfers. However, it also has a von Neumann flavor since the memory space is unified, and there is no separation of program and data spaces. As a result, the user can choose to locate programs and data at any desired location.

Some of the major features of the 320C30 are:

- a 60-ns cycle time that results in execution of over 16 million instructions per second (MIPS) and over 33 million floating-point operations per second (Mflops);
- 32-bit data buses and 24-bit address buses for a 16M-word overall memory space;
- dual-access, 4K × 32-bit on-chip ROM and 2K × 32-bit on-chip RAM;

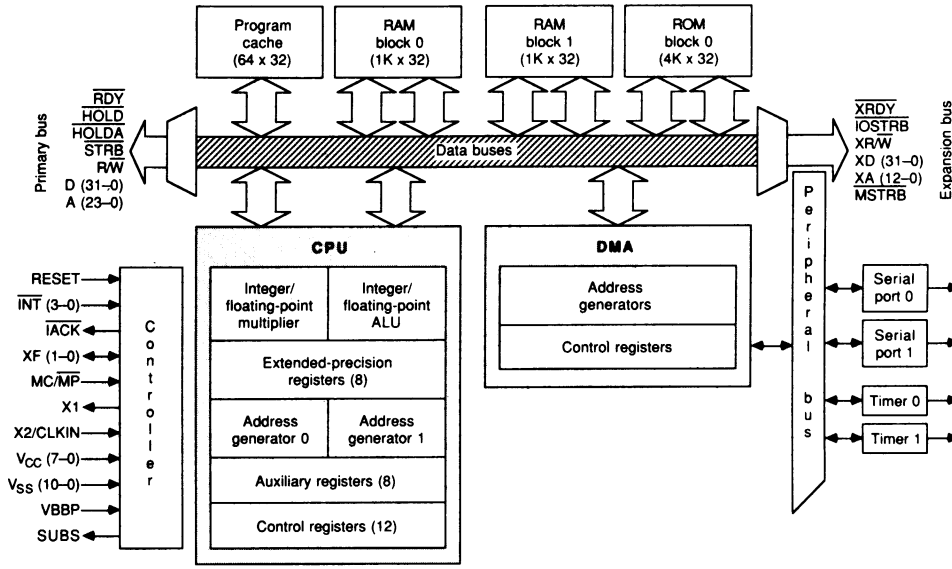

Figure 1. Block diagram of the TMS320C30 architecture.

- a 64 × 32-bit program cache;
- a 32-bit integer/40-bit floating-point multiplier and ALU;
- eight extended-precision registers, eight auxiliary registers, and 12 control and status registers;
- generally single-cycle instructions;
- integer, floating-point, and logical operations;
- two- and three-operand instructions;
- an on-chip DMA controller; and
- fabrication in 1-μm CMOS technology and packaging in a 180-pin package.

Memory organization. The 320C30 provides 4K 32-bit words of on-chip ROM, and 2K 32-bit words of on-chip RAM. The on-chip ROM is mapped into the first 4K of the overall memory map; it is accessed when the processor operates in the microcomputer mode. Location 0 of the memory map holds the reset vector, and adjacent locations hold other interrupt vectors. In microprocessor mode, the reset vector resides in external memory, and on-chip ROM is not accessed. The 2K on-chip RAM consists physically of two segments of 1K words each. These two segments of RAM are mapped into adjacent sections of the memory. Figure 3 on the next page shows the arrangement of the on-chip memory, as well as the cache, buses, and two external interfaces/buses, which we examine later.

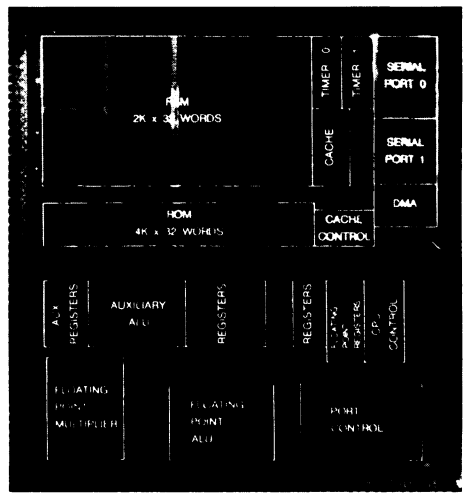

Figure 2. Die photograph of the 320C30.

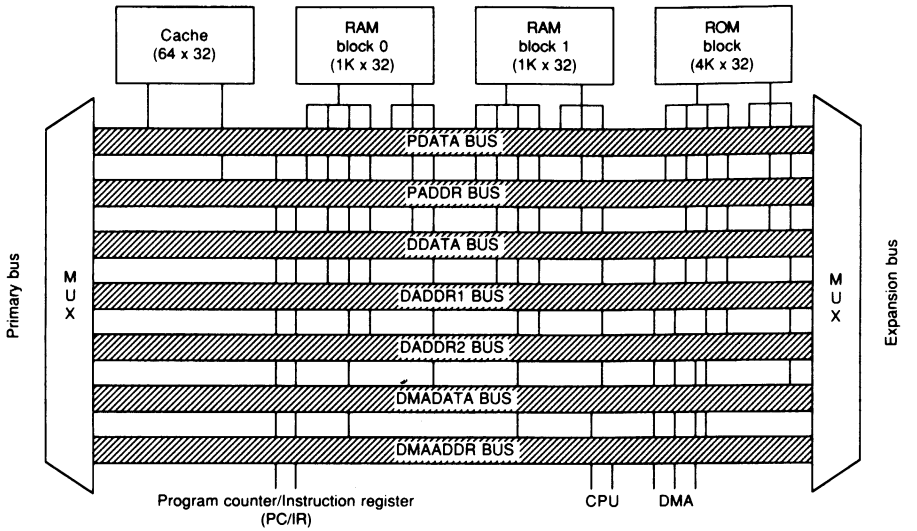

Figure 3. On-chip memory, cache, and buses.

The internal memory (both ROM and RAM) supports two accesses for reads and/or writes in one cycle. This key feature permits high throughput and ease of programming, since it makes possible three-operand instructions with two operands residing in the memory. Notice that, to support this feature, we include two buses dedicated to data addresses (DADDR1, DADDR2) and one bus to carry the data (DDATA). There are also separate program buses, PDATA and PADDR.

The address buses are 24 bits wide, indicating that the overall memory space is 16 million (32-bit) words. We believe this large space will facilitate implementation of algorithms in image processing applications that often require large amounts of memory. The unified memory space offers flexibility in placing program and data. But it also permits optimal use of the memory space as a trade-off between program and data.

An important addition to the architecture is the 64-word instruction cache. To reduce the overall system cost of applications, system designers often use slower (and cheaper) external memories, a tactic that could slow down the processor and degrade the performance. The instruction cache addresses this problem by storing on-chip instructions that have been fetched previously. Its main advantage becomes obvious when loops must be executed. In this case, the first time the instructions are fetched, they are also stored in the cache. Any subsequent execution of the loop does not access external memory but fetches instructions from the cache, resulting in higher speed and making the external buses available for data transfers.

The cache is segmented into two sections of 32 words each that are transparent to users. A user can, however, control the operation of the cache by manipulating three control bits that are contained in the status register of the CPU. Each control bit is dedicated to a specific operation: cache enable/disable, cache freeze, and cache clear. When a cache miss occurs, that is, when the next instruction is not included in the cache, the instruction is brought in and also stored in the cache. The two cache sections are updated on a least recently used basis.

CPU organization. The CPU consists of the ALU (arithmetic logic unit), the hardware multiplier, and the register file. These units are shown in Figure 4.

The register file consists of

- eight 40-bit-wide, extended-precision registers R0 through R7,
- eight 32-bit auxiliary registers AR0 through AR7, and
- twelve 32-bit control registers.

The extended-precision registers function as accumulators and can handle both floating-point and integer numbers. When they are used for floating-point numbers, the top eight bits represent the exponent and the bottom 32 bits the mantissa of the number. In their integer format, registers R0 through R7 use only their bottom 32 bits, keeping the top 8 bits unchanged in any integer or logical operation.

The eight auxiliary registers AR0 through AR7 can function as memory pointers in indirect addressing, as loop counters, or as general-purpose registers in integer arithmetic or logical operations. Associated with these registers are two auxiliary register arithmetic units (ARAU) that generate two memory addresses in parallel for the instructions that need them. The flexibility of indirect addressing increases even further when two index registers are used in conjunction with the auxiliary registers, as we discuss later.

The register file contains 12 control registers designated for specific functions. If the control registers are not used for these functions, they can be treated as general-purpose registers in integer arithmetic and logical operations. Examples of such control registers are the

- status register,
- index registers,
- stack pointer,
- interrupt mask and interrupt flag registers, and
- repeat-block registers.

In particular, the stack-pointer register points to the software stack. The user has the flexibility of designating where the stack resides, and even of changing its location during the program execution. This feature also makes the stack of essentially unlimited depth and permits its usage not only for storing the program counter during subroutine calls but also for passing arguments to subroutines. Such an arrangement is particularly convenient in the development of compilers, and we have used it extensively in the 320C30's optimizing C compiler.

The ALU performs floating-point, integer, and logical operations. The ALU always stores the result in the register file, but the input can come either from the register file or from memory, or it can be an immediate value.

In the case of floating-point arithmetic, the input to the ALU can originate from either a 40-bit extended-precision register or a 32-bit memory datum. Registers R0 through R7 store the 40-bit-word result. On the other hand, in integer arithmetic, both input and output are 32-bit numbers, and the output can move to either the lower 32 bits of the R0 through R7 registers or to any other register in the register file.

The single-cycle hardware multiplier has been an integral part of DSPs because any real-time application relies on the fast execution of multiplies. Following the same distinction as in the previous paragraph on the ALU, the multiplier performs both floating-point and integer multiplications. The 32-bit inputs to a floating-point multiplication yield a 40-bit-wide result for storage in one of the extended-precision registers.

In both the ALU and the multiplier the results of the operations are automatically normalized, thus handling any overflows of the mantissa. If there is an exponent overflow, the result is saturated in the direction of overflow and the overflow flag is set. Underflows are handled by setting the result to zero and setting an underflow flag.

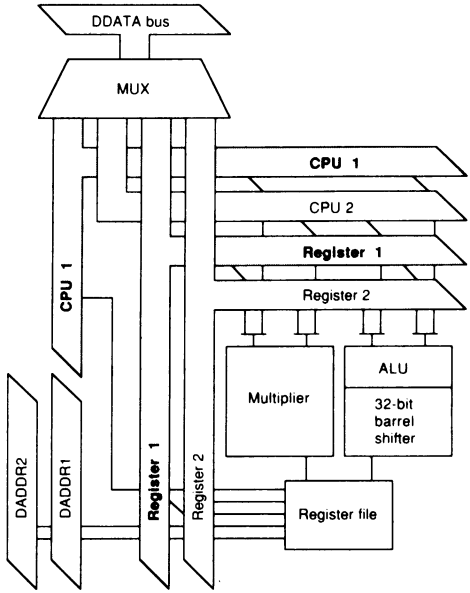

Figure 4. The 320C30 central processing unit.

Buses and peripherals. Figure 3 shows that multiple on-chip buses handle program, data, and DMA operations in parallel. The device contains separate address and data buses for these three operations, with the data having two address buses to accommodate the access of multiple operands from the memory in one cycle. Also, separate buses lead to the register file. The rule to remember is that, in one cycle, up to two data memory accesses are permitted for any on-chip memory block. This multiplicity of buses eliminates bottlenecks. The user can maximize the throughput of the device by a judicious combination of the on-chip memory with the two external buses (the primary bus and the expansion bus).

The primary bus contains a 24-bit address bus and a 32-bit data bus. Its true space, though, is 16M words minus the on-chip memory and the expansion bus. The primary bus can be placed in high impedance when the device is put on hold. To facilitate its interfacing with slow memories, the 320C30 offers programmable wait states (up to seven) as well as an external ready signal.

The expansion bus contains a 13-bit address bus and a 32-bit data bus. It has two strobes, one for memory and one for I/O accesses. In other words, the memory space of the

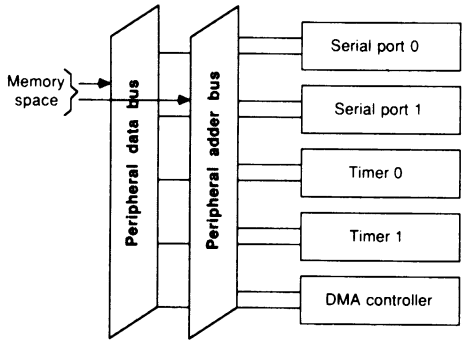

Figure 5. Peripheral bus and peripherals.

expansion bus is two segments of 8K words each, one segment mapped as regular memory and the other one mapped as I/O. Like the primary bus, the expansion bus has up to seven software-programmable wait states.

A major innovation in the 320C30—to support system-level solutions and to help in adapting the device to changing needs—is the peripheral bus shown in Figures 1 and 5. The peripheral bus supplies a way of expanding or varying the interface with the outside world without changing the core of the device. All of the peripherals attached to this bus are mapped to memory, and they can be replaced by others with a minimal effort if certain applications have different demands.

Currently, we have implemented a DMA controller, two serial ports, and two timers as peripherals. The DMA controller performs reads from and writes to any location in the 320C30 memory map without interfering with the operation of the CPU. The DMA controller contains its own address generators, source and destination address registers, and transfer counter. The two modular and totally independent serial ports are identical with a complementary set of control registers. Each serial port can be configured to transfer 8, 16, 24, or 32 bits of data per word, with each port clock originating either internally or externally. The pins of the serial ports are configurable as general-purpose I/O pins, while the serial ports can also be configured and used as timers.

The two 320C30 timer modules function as general-purpose timer/event counters; each have two signaling modes and internal or external clocking. Available to each timer is an I/O pin for use as an input clock to the timer, as an output signal driven by the timer, or as a general-purpose pin.

Software

The software features of a programmable DSP are probably the most important features because they determine the effectiveness of the implementation. Typically, the user first develops an application on a large computer using a high-level language and, once it is working satisfactorily, ports it to a DSP device. The software features of the 320C30 that we discuss include the integer and floating-point number representations, addressing modes, pipeline effects, and different types of instructions and constructs.

Integer and floating-point formats. A 32-bit, twos-complement notation represents the integers. In addition to this single-precision format, we have a short format, consisting of 16-bit, twos-complement numbers used only for immediate operands. Every instruction of the 320C30 consists of one 32-bit word.

We use three formats for floating-point numbers: short, single precision, and extended precision. The single-precision, 32-bit-wide format assigns 24 bits to the mantissa and 8 bits to the exponent. The exponent occupies the 8 most significant bits, and it is represented in twos-complement notation, taking values between −128 and 127. The exponent value −128 is the result reserved to represent zero.

The mantissa, placed at the 24 least significant bits of a 32-bit number, is normalized to a number with an absolute value between 1.0 and 2.0. Since the mantissa is represented in a normalized, twos-complement notation, the leftmost bit, which corresponds to the sign, and its adjacent bit will always be the complement of each other. As a result, only the sign bit is represented, with the most significant bit suppressed. In other words, the mantissa contains 24 significant bits plus the sign bit, with the most significant bit implied.

Addressing modes. The 320C30 supports several addressing modes that allow the user to access data from memory, registers, and the instruction word. The basic addressing modes are

- register,
- direct,
- indirect,
- short immediate,
- long immediate, and
- PC relative.

In register mode the operand is placed into a CPU register that is explicitly specified in an instruction. In direct mode the data memory address is formed by preceding the 16 least significant bits of the instruction word with the 8 least significant bits of the data page pointer. To keep all instructions one word long, we store only the 16 least significant bits from the address in the instruction word; the rest become the data page pointer. This restriction implies that in direct addressing the memory space is segmented into 256 pages of 64K words each.

Table 1.
Addressing modes of the 320C30.

Mode	Example	Operation	Description
Register	ADDF R0,R1		Operand in R0
Direct	ADDF @MEM, R1	Addr = MEM	Operand in MEM
Short immediate	ADDF 3.14,R1		Operand = 3.14
Long immediate	BR LABEL		Branch to LABEL
PC relative	BGE LABEL		Branch to LABEL
Indirect	ADDF *+AR0(di),R1	Addr = AR0 + di	Predisplacement add without modification
Indirect	ADDF *−AR0(di),R1	Addr = AR0 − di	Predisplacement subtract without modification
Indirect	ADDF *++AR0(di),R1	Addr = AR0 + di AR0 = AR0 + di	Predisplacement add and modify
Indirect	ADDF *−−AR0(di),R1	Addr = AR0 − di AR0 = AR0 − di	Predisplacement subtract and modify
Indirect	ADDF *AR0++(di),R1	Addr = AR0 AR0 = AR0 + di	Postdisplacement add and modify
Indirect	ADDF *AR0−−(di),R1	Addr = AR0 AR0 = AR0 − di	Postdisplacement subtract and modify
Indirect	ADDF *AR0++(di)%,R1	Addr = AR0 AR0 = circ(AR0 + di)	Postdisplacement add and circular modify
Indirect	ADDF *AR0−−(di)%,R1	Addr = AR0 AR0 = circ(AR0−di)	Postdisplacement subtract and circular modify
Indirect	ADDF *AR0++(IR0)B,R1	Addr = AR0 AR0 = B(AR0 + IR0)	Postindex (IR0) add and bit-reversed modify

di is an integer between 0 and 255 or one of the index registers IR0 and IR1.

Indirect addressing, the most versatile of all the modes, specifies the address of an operand in memory through the contents of an auxiliary register. As an option, the contents of the register can be modified by constant displacements or by the contents of the index registers. Table 1 lists all of the addressing modes, with particular emphasis on indirect addressing modes.

An instruction explicitly specifies the auxiliary register used for indirect addressing. The user can modify it by a constant displacement taking values 0 to 255 or by the contents of one of the two index registers IR0 or IR1. The modification can take place before or after accessing the memory. In the case of premodification, the user has the option to change the contents of the auxiliary register either permanently or temporarily. The notation used for such modifications is reminiscent of the C-language syntax.

Two special forms of indirect addressing that are particularly useful are bit-reversed and circular addressing. Bit-reversed addressing is used with the fast Fourier transform to compensate for the fact that normally ordered data at the input of the transform are scrambled at output (bit-reversed order). To avoid moving the data around to place them in the proper order, bit-reversed addressing accesses the data in scrambled order for any subsequent operation.

Circular addressing implements circular buffers. Such buffers are very convenient for use in digital-filtering operations. In circular addressing, BK, one of the control registers, specifies the size of the block. Then, when the user modifies the contents of an auxiliary register (pointing within that block) in a circular fashion, the final value is tested to determine if it is still within the block. If it is not, it is wrapped around using modulo arithmetic.

The short-immediate mode encodes immediate, 16-bit-long operands of arithmetic operations. The long-immediate mode encodes program control instructions (branch instructions) for which it is useful to have a 24-bit absolute address contained in the instruction word. Finally, the PC-relative addressing also applies to program control instructions and uses the difference from the present location of the PC counter rather than an absolute address. The last two

modes are transparent to the user. The user specifies the branching label wanted, and the assembler assigns the appropriate addressing mode.

Pipeline. To achieve the high throughput of the device, the 320C30 uses a four-phase pipeline with five major functional units operating in parallel. These five units are

- instruction fetching,
- instruction decoding and address generation,
- operand reads,
- instruction execution, and
- DMA transfer.

Figure 6 shows diagrammatically how the pipeline operates on successive instructions. When the pipeline is full, an instruction completes the execution phase every 60-ns machine cycle.

Occasionally conflicts may arise, as in the case of a loaded auxiliary register that needs to be used for indirect addressing in the next instruction. To handle such cases, we established a priority between the different units, giving DMA the lowest priority. Among the others, an Execute instruction has the highest and a Fetch instruction the lowest priority.

In programming the device, the user does not have to worry about the pipeline conflicts, which do not occur that often anyway. When a conflict does occur, the device automatically inserts the necessary extra cycle(s) to make the instructions behave as expected. In most cases, this arrangement will be sufficient for successful operation. For time-critical operations, though, it may be necessary to remove the extra cycles caused by pipeline conflicts. The user can make this correction by rearranging the instructions of the program. To do so, the user must determine how to identify the locations where insertions occur. For that purpose, the development tools (simulator, emulators) contain a tracing feature that can display the pipeline. In this trace, any conflicts are immediately identified, and then the user can take steps to correct the problem.

Instruction set features. The instruction set of the 320C30 supports both two- and three-operand instructions. In all arithmetic instructions (except Store), the destination is a register in the register file. The source operands can come from memory or from a register or, in the case of two-operand instructions, can be part of the instruction word.

A unique feature of the 320C30 is the set of instructions in which operations execute in parallel. This construct permits a high degree of concurrency and execution of any arithmetic or logical instruction in parallel with a Store instruction. It also supports parallel multiplies and adds, as well as parallel loading and storing of two registers. Parallel multiply and adds lead to the peak performance of 33 Mflops. Executing the Store instruction at the same time with another arithmetic operation essentially permits this kind of data movement without a penalty. As an example, the following instruction adds the contents of memory pointed to by AR1 (indicated by *AR1) to register R0 (treating them as floating-point numbers) and places the result in register R1. In parallel with that process, the original contents of R1 are stored in the memory location indicated by AR3.

```
        ADDF    *AR1,R0,R1
||      STF     R1,*AR3
```

When executing a branch instruction, the pipeline must be flushed since the path followed after the branch is data dependent. As a result, a regular branch instruction is more costly than other instructions, taking four cycles to complete. This overhead may be unacceptable in some time-critical applications. To alleviate this problem and to offer more flexibility to the programmer, the 320C30 contains a set of delayed branches that complement the set of standard branches. In a delayed branch, the three instructions following the branch instruction execute whether the branch is taken or not taken. As a result, the delayed branch ends up taking only one cycle to execute. The same approach can be used even when there are less than three such instructions, by adding NOPs (no operations). The branch will still take less than four cycles.

The greatest cost of branching occurs during the execution of loops. In looping, a counter is decremented and compared to zero at the end of the loop. If it is not zero, a branch is taken to the beginning of the loop. The 320C30 offers a special arrangement that implements loops with no

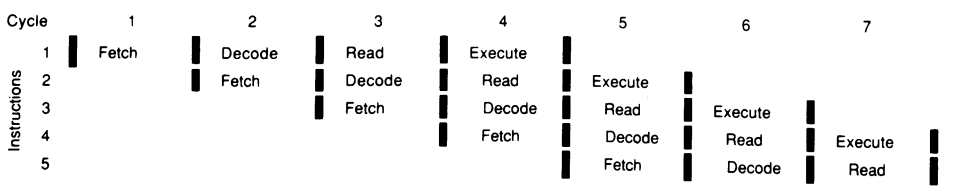

Figure 6. Pipeline of 320C30 instructions.

User-friendly development tools offer extra support: an optimizing C compiler and a DSP operating system.

overhead. The two instructions RPTB (repeat block) and RPTS (repeat single) realize this arrangement. The format of the RPTB instruction is:

```
RPTB        LABEL

(put instructions here)

LABEL (last instruction)
```

Associated with the repeat-block construct are three of the 12 control registers in the register file. One register indicates the beginning of the block, the second indicates the end of the block, and the third acts as the repeat counter. The assembler automatically assigns values to the first two registers. They contain the address of the instruction immediately below RPTB, and the address of LABEL respectively. Users should initialize the repeat counter before entering the loop. In terms of execution time, this arrangement behaves as if the loop were implemented with straight-line code.

The instruction RPTS has the format

```
RPTS    count
```

and it repeats the following instruction "count" times. It differs from RPTB in that it

- applies to only one instruction;
- does not refetch the instruction for every execution, but keeps it in the instruction register thus freeing the buses for data transfers, and
- is not interruptible.

Table 2 on the next page is a sample of the instructions available on the 320C30. Although we included a rich set of instructions for both DSP and general-purpose processing, the perceived size of the instruction set is much smaller. The reason is that a symmetry exists between integer and floating-point instructions, between instructions with two or three operands, and between single and parallel instructions. For instance, addition is represented by ADDI, ADDF, or ADDC in the case of adding integers, floating-point numbers, or adding with a carry. The three-operand instructions have the same form, with a 3 appended at the end (ADDF3). All of the multiplier and ALU operations can be performed in parallel with a Store instruction, and such instructions take the form of the following example:

```
        ADDF3    *AR0,R1,R2
||      STF      R0,*AR1
```

Furthermore, two loads or two stores can execute in parallel, as is also the case with a multiply and an add or a multiply and a subtract. The design of the instruction set has been guided by a desire to ease programming efforts. The execution results of an instruction are always available for use in the instruction that follows.

Besides the regular arithmetic and logical instructions, the 320C30 includes instructions to handle the software stack, internal and external interrupts, and branches and subroutine calls. Conditional loads and calls make the programming more compact and efficient, while special instructions (called interlocked instructions) can be used in multiprocessor environments.

Development tools and support

The newer DSP devices offer increased processing power that permits the implementation of more complicated and demanding algorithms. However, as the complexity of the algorithm increases, the task of debugging the implementation becomes more difficult. The 320C30 addresses this problem by providing user-friendly development tools and offering extra support in the form of an optimizing C compiler and a DSP operating system.

The assembler translates assembly-language source files into machine-language object files. Source files can contain instructions, assembler directives, and macro directives. Assembler directives control various aspects of the assembly process such as the source-listing format, symbol definition, and method of placing the source code into sections. Macro directives permit a concise representation of groups of instructions that occur frequently.

The linker combines object files into one executable object module. As it creates the executable module, the linker performs relocation operations and resolves external references. The linker accepts relocatable COFF (Common Object File Format) object files, created by the assembler, as input. It can also accept archive library members and output modules created by a previous linker run. Linker directives allow the user to combine object-file sections, bind sections or symbols to specific addresses or within specific portions of 320C30 memory, and define or redefine global symbols. An associated archiver can create macro or object-file libraries.

The software simulator is a very important tool for debugging 320C30 programs. Its interface consists of a screen broken into windows that display the internal registers, the reverse-assembled program, and a versatile window where memory, breakpoints, and a wealth of other information can be displayed. The same interface (modified to accommodate some special features) is also used with the hardware emulator. The major features of the simulator include:

- Simulation of the entire 320C30 instruction set and the

Table 2.
Instructions for the 320C30.

Instruction	Description	Instruction	Description
Load and store instructions			
LDE	Load floating-point exponent	POP	Pop integer from stack
LDF	Load floating-point value	POPF	Pop floating-point value from stack
LDF*cond*	Load floating-point value conditionally	PUSH	Push integer on stack
LDI	Load integer	PUSHF	Push floating-point value on stack
LDI*cond*	Load integer conditionally	STF	Store floating-point value
LDM	Load floating-point mantissa	STI	Store integer
Two-operand instructions			
ABSF	Absolute value of a floating-point number	NORM	Normalize floating-point value
ABSI	Absolute value of an integer	NOT	Bitwise logical-complement
ADDC †	Add integers with carry	OR †	Bitwise logical-OR
ADDF †	Add floating-point values	RND	Round floating-point value
ADDI †	Add integers	ROL	Rotate left
AND †	Bitwise logical-AND	ROLC	Rotate left through carry
ANDN †	Bitwise logical-AND with complement	ROR	Rotate right
ASH †	Arithmetic shift	RORC	Rotate right through carry
CMPF †	Compare floating-point values	SUBB †	Subtract integers with borrow
CMPI †	Compare integers	SUBC	Subtract integers conditionally
FIX	Convert floating-point value to integer	SUBF	Subtract floating-point values
FLOAT	Convert integer to floating-point value	SUBI	Subtract integer
LSH †	Logical shift	SUBRB	Subtract reverse integer with borrow
MPYF †	Multiply floating-point values	SUBRF	Subtract reverse floating-point value
MPYI †	Multiply integers	SUBRI	Subtract reverse integer
NEGB	Negate integer with borrow	TSTB †	Test bit fields
NEGF	Negate floating-point value	XOR †	Bitwise exclusive-OR
NEGI	Negate integer		
Program control instructions			
B*cond*	Branch conditionally (standard)	IDLE	Idle until interrupt
B*cond*D	Branch conditionally (delayed)	NOP	No operation
BR	Branch unconditionally (standard)	RETI*cond*	Return from interrupt conditionally
BRD	Branch unconditionally (delayed)	RETS*cond*	Return from subroutine conditionally
CALL	Call subroutine	RPTB	Repeat block of instructions
CALL*cond*	Call subroutine conditionally	RPTS	Repeat single instruction
DB*cond*	Decrement and branch conditionally (standard)	SWI	Software interrupt
DB*cond*D	Decrement and branch conditionally (delayed)	TRAP*cond*	Trap conditionally

† Two- and three-operand versions

key peripheral features;

- Command entry from either menu-driven keystrokes (menu mode) or from line commands (line mode);
- Help menus for all screen modes;
- Quick storage and retrieval of simulation parameters from files to facilitate preparation for individual sessions;
- Reverse assembly allowing editing and reassembly of source statements;
- Multiple execution modes;
- Trace expressions that are easy to define;
- Trace execution that can display designated expression values, cache memory, and the instruction pipeline; and
- Breakpoints that can occur on address read, write, or both, on address execute, and on expression valid.

Perhaps the most important trend with the newer DSPs is the availability of high-level-language compilers. The presence of C and Ada compilers in the 320C30 is not an accident since the 320C30 was designed with a compiler in mind. We expect this path to a high-level language to make the porting of application programs from large computers much easier. The algorithm can be developed almost entirely on a large computer and then converted to the 320C30 assembly language by compilation.

The C compiler for the 320C30 has exceptional efficiency,[2] which makes a good C program almost as effective as the assembly-language program. The C compiler will be sufficient for most applications. The exception is time-critical applications. In such cases one can use the fact that most DSP algorithms spend the vast majority of the execution time on a small section of the code. (Researchers often mention the 90/10 rule: 90 percent of the time is spent on 10 percent of the code.) Under these circumstances, the user can optimize execution by creating very fast assembly-language routines that implement the time-critical sections, and call them from C as regular C functions. To achieve this, we define the C function interface very precisely so that users can create their own routines. The C-compiler package comes with a library of general-purpose mathematical, interface, and I/O functions.

Besides this method of optimizing the performance of the C language, two more methods can be used. The first one is based on the fact that the output of the compiler is an assembly-language program. The user can edit this program and optimize it by rearranging the instructions. The second method is to use the "asm" directive supported by the C compiler. The arguments of this directive are passed to the output of the compilation without any alteration so that the user can insert assembly-language instructions into the middle of the C program.

A key part of the 320C30 development environment is Spox, the first real-time operating-system for a single-chip DSP. Spox, developed by Spectron Microsystems, extends the core C language with a library of standard I/O routines and, most importantly, a DSP math package. One of Spox's unique features is that it provides users with software objects that are especially suited for DSP. Some of these objects are vectors, matrices, filters, and streams. The math package and these software objects are carefully designed to take full advantage of the capabilities of the 320C30. Spox also supports multitasking, thus allowing the user to easily implement the more complex control structures that are becoming essential for DSP systems.

Perhaps the most important trend with the newer DSPs is the availability of high-level-language compilers.

By providing a complete software development environment that includes compilers and operating systems along with the more-traditional tools such as assemblers and linkers, we allow the user to move from system conception to system implementation in the shortest possible time.

The next level of development tools includes the hardware emulators for debugging target hardware or determining the performance of an algorithm on the 320C30 device itself. The XDS1000 is a real-time, in-circuit emulator/software development tool based on the 320C30. Besides these tools from Texas Instruments, other companies offer related support, such as the PC-based development board by Atlanta Signal Processors and the development platform of Spectron Microsystems for PCs and Sun workstations.

Applications

Certain features of the 320C30 such as its high speed, versatile architecture, and rich instruction set, make it easy to implement very demanding algorithms. The large memory space makes the device suitable for application areas such as image processing in which memory addressing is one of the prime considerations. And the C compiler makes it easy to construct algorithms with complicated logic.

General DSP algorithms. Almost every DSP application needs to perform some kind of filtering, the first application considered for a DSP device. Digital filters are categorized as FIR (finite-length impulse response) and IIR (infinite impulse response) filters,[3,4] or, equivalently, as filters that have only zeros or both poles and zeros. Each of these categories can have either fixed or adaptive coefficients.

The 320C30 implements FIR filters very efficiently. For instance, let an FIR filter have an impulse response $h[0]$, $h[1], \ldots, h[N \times 1]$, and let $x[n]$ represent the input of the filter at time n. Then, the following equation gives the output $y[n]$ with the equation:

$$y[n] = h[0] \times x[n] + h[1] \times x[n-1] + \ldots + h[N-1] \times x[n-N+1]$$

```
;
; Typical Calling Sequence:
;
;       load    AR0
;       load    AR1
;       load    RC
;       load    BK
;       CALL    FIR
;
;
; Data Memory Organization:
;
;                Impulse                    Initial          Final
;               response                  input samples   input samples
;    Low     +------------+  Oldest  +------------+  +------------+
; address !    h(N-1)     !  input   ! x(n-(N-1)) !  !    x(n)    !---+
;         +------------+           +------------+  +------------+   !
;         !    h(N-2)     !        ! x(n-(N-2)) !  ! x(n-(N-1)) !   !
;         +------------+           +------------+  +------------+   !
;               .                        .              .        Circular
;               .                        .              .         queue
;         +------------+           +------------+  +------------+   !
;         !    h(1)     !        !    x(n-1)    !  !    x(n-2)    !   !
;  High   +------------+  Newest  +------------+  +------------+   !
; address !    h(0)     !  input   !     x(n)     !  !    x(n-1)    !---+
;         +------------+           +------------+  +------------+
;
; The physical address for the start of the input samples must be on
; a boundary with the LSBs set to zero according to the length of the
; buffer. The pointer to the input sequence (x) is incremented and
; assumed to be moving from an older input to a newer input. At the
; end of the subroutine AR1 will be pointing to the position for the
; next input sample.
;
; Argument Assignments:
;
;   Argument  | Function
;   ----------+-----------------------
;   AR0       | Address of h(N-1)
;   AR1       | Address of x(N-1)
;   RC        | Length of filter - 2 (N-2)
;   BK        | Length of filter (N)
;
; Registers used as input: AR0, AR1, RC, BK
; Registers modified: R0, R2, AR0, AR1, RC
; Register containing result: R0
;
;
; Program size: 6 words
;
; Execution cycles: 11 + (N-1)
;
;=============================================================================
;
        .global     FIR
;                                           ; initialize R0:
FIR     MPYF3   *AR0++(1),*AR1++(1)%,R0     ; h(N-1) * x(n-(N-1)) -> R0
        LDF     0.0,R2                      ; initialize R2.
;
; filter ( 1 <= i < N)
;
        RPTS    RC                          ; setup the repeat single.
        MPYF3   *AR0++(1),*AR1++(1)%,R0     ; h(N-1-i) * x(n-(N-1-i)) -> R0
||      ADDF3   R0,R2,R2                    ; multiply and add operation
;
        ADDF    R0,R2,R0                    ; add last product
;
; return sequence
;
        RETS                                ; return
;
; end
;
   .end
```

Figure 7. FIR filter implementation on the 320C30.

```
;
; Typical Calling Sequence:
;
;       load    R2
;       load    AR0
;       load    AR1
;       load    IR0
;       load    IR1
;       load    BK
;       load    RC
;       CALL    IIR2
;
; Data Memory Organization:
;
;                Filter                    Initial delay     Final delay
;             coefficients                  node values       node values
;    Low    +-------------+  Newest  +-------------+   +-------------+
; address   |    a2(0)    |  delay   |    d(0,n)   |   |   d(0,n-1)  |<----+
;           +-------------+          +-------------+   +-------------+     |
;           |    b2(0)    |          |   d(0,n-1)  |   |   d(0,n-2)  | circular
;                                                                        queue
;           +-------------+  oldest  +-------------+   +-------------+     |
;           |    a1(0)    |  delay   |   d(0,n-2)  |   |    d(0,n)   |<----+
;           +-------------+          +-------------+   +-------------+
;           |    b1(0)    |          |    Empty    |   |    Empty    |
;           +-------------+          +-------------+   +-------------+
;           |    b0(0)    |                 |
;           +-------------+          +-------------+   +-------------+
;                 .                  |  d(N-1,n)   |   | d(N-1,n-1)  |<----+
;                 .                  +-------------+   +-------------+     |
;                 .                  | d(N-1,n-1)  |   | d(N-1,n-2)  | circular
;                                                                        queue
;           +-------------+          +-------------+   +-------------+     |
;           |   a2(N-1)   |          | d(N-1,n-2)  |   |   d(N-1,n)  |<----+
;           +-------------+          +-------------+   +-------------+
;           |   b2(N-1)   |          |    Empty    |   |    Empty    |
;           +-------------+          +-------------+   +-------------+
;           |   a1(N-1)   |
;           +-------------+
;           |   b1(N-1)   |
;  High     +-------------+
; address   |   b0(N-1)   |
;           +-------------+
;
; The physical address for the start of each circular queue of delay node
; values must be on a boundary with the LSBs set to zero according to the
; length of the buffer. The BK (block size) register must contain the
```

(Continued on page 26)

Figure 8. Implementation of N biquads on the 320C30.

Two features of the 320C30 facilitate the implementation of the FIR filters: parallel multiply/add operations and circular addressing. The first feature permits a multiplication and an addition to execute in one machine cycle, while the second makes a finite buffer of length N sufficient for the data $x[n]$. Figure 7 shows the arrangement of the data and the assembly code for an FIR filter. Note that the filter takes one cycle of execution per tap.

The transfer function of the IIR filters contains both poles and zeros, and its output depends on both the input and the past output. As a rule, these filters need less computation than a FIR filter of similar frequency response, but they have the drawback of being sensitive to coefficient quantization. Most often, the IIR filters are implemented as a cascade of second-order sections, called biquads. To implement an IIR filter consisting of N biquads, let $a1[i]$, $a2[i]$ be the numerator coefficients of the ith biquad and $b0[i]$, $b1[i]$, $b2[i]$ the denominator coefficients of the same biquad. Also, let $x[n]$ be the input and $y[n]$ be the output of the IIR filter. In canonic form, the following C code implements the N biquads:

```
y[0,n] = x[n];
for (i=0; i<N; i++){
 d[i,n] = a2[i]*d[i,n-2] + a1[i]*d[i,n-1] + y[i-1,n];
 y[i,n] = b2[i]*d[i,n-2] + b1[i]*d[i,n-1] +
          b0[i]*d[i,n];
}
y[n] = y[N-1,n];
```

Figure 8 shows the memory arrangement and the code for this implementation on the 320C30.

In addition to the fixed-coefficient filters, the 320C30 can also implement very effectively adaptive filters (with three cycles per updated tap).

Fourier transforms are another important tool often used in DSP systems. The purpose of the transform is to convert information from the time domain to the frequency do-

```
; value 3. The result y(n) is placed in R0.  At the end of the program,
; AR1 points to the new d(0,n-2) so that it is set when the new sample
; comes in.
;
; Argument Assignments:
;   Argument  : Function
;   ----------+----------------------
;   R2        : Input sample x(n)
;   AR0       : Address of filter coefficients (a2(0))
;   AR1       : Address of delay node values (d(0,n-2))
;   BK        : BK = 3
;   IR0       : IR0 = 4
;   IR1       : IR1 = 4*N-4
;   RC        : Number of biquads (N) - 2
;
; Registers used as input: R2, AR0, AR1, IR0, IR1, BK, RC
; Registers modified: R0, R1, R2, AR0, AR1, RC
; Register containing result: R0
;
; Program size: 17 words
;
; Execution cycles: 23 + 6N
;
;*****************************************************************************
;
        .global IIR2
;
IIR2    MPYF3   *AR0, *AR1, R0                  ; a2(0) * d(0,n-2) -> R0
        MPYF3   *++AR0(1), *AR1--(1)%, R1       ; b2(0) * d(0,n-2) -> R1
;
        MPYF3   *++AR0(1), *AR1, R0             ; a1(0) * d(0,n-1) -> R0
||      ADDF3   R0, R2, R2                      ; first sum term of d(0,n).
;
        MPYF3   *++AR0(1), *AR1--(1)%, R0       ; b1(0) * d(0,n-1) -> R0
||      ADDF3   R0, R2, R2                      ; second sum term of d(0,n).
;
        MPYF3   *++AR0(1), R2, R2               ; b0(0) * d(0,n) -> R2
||      STF     R2, *AR1--(1)%                  ; store d(0,n)| point to
;                                                 d(0,n-2).
;
;
    RPTB    LOOP                                ; loop for 1 <= i < N
;
        MPYF3   *++AR0(1), *++AR1(IR0), R0      ; a2(i) * d(i,n-2) -> R0
||      ADDF3   R0,R2,R2                        ; first sum term of y(i-1,n)
;
        MPYF3   *++AR0(1), *AR1--(1)%, R1       ; b2(i) * d(i,n-2) -> R1
||      ADDF3   R1,R2,R2                        ; second sum term of y(i-1,n)
;
        MPYF3   *++AR0(1), *AR1, R0             ; a1(i) * d(i,n-1) -> R0
||      ADDF3   R0, R2, R2                      ; first sum term of d(i,n).
;
        MPYF3   *++AR0(1), *AR1--(1)%, R0       ; b1(i) * d(i,n-1) -> R0
||      ADDF3   R0, R2, R2                      ; second sum term of d(i,n).
;
        STF     R2, *AR1--(1)%                  ; store d(i,n)| point to
;                                                 d(i,n-2).
LOOP    MPYF3   *++AR0(1), R2, R2               ; b0(i) * d(i,n) -> R2
;
; final summation
;
        ADDF    R0,R2                           ; first sum term of y(N-1,n)
        ADDF3   R1,R2,R0                        ; second sum term of y(N-1,n)
;
        NOP     *AR1--(IR1)                     ; return to first biquad
        NOP     *AR1--(1)%                      ; point to d(0,n-1)
;
; return sequence
;
        RETS                                    ; return
;
; end
;
    .end
```

Figure 8 (cont'd.)

main. Computationally efficient implementation of Fourier transforms are known as the fast Fourier transform (FFT).[3-5] Table 3 shows the timing for different FFTs on the 320C30. The code for these FFTs, as well as the routines listed in Table 4, appear in the *TMS320C30 User's Guide*.[6]

The 320C30 has many features that make it well suited for FFTs, such as the high speed of the device, the floating-point capability, the block-repeat construct, and the bit-reversed addressing mode. For instance, the FFT shown in Figure 9 on the next page can be implemented in code that can be entirely contained in the 64-word cache of the 320C30.[7]

Telecommunications and speech. Telecommunications and speech applications have many requirements in common with other DSP applications, but they also have some special needs. For instance, telecommunications applications interfacing to T1 carriers sometimes need to convert between a linear signal and one compressed by μ-law or A-law formats. Such a conversion can be realized with hardware by adding a peripheral to the DSP peripheral bus. This is the approach taken in some members of the TMS320 first generation of devices. An alternative way is to do the same function with software.

In speech applications, digital filters are often implemented in lattice form. Depending on the application, both FIR and IIR filters are realized this way, although sometimes the terminology lattice filter and inverse lattice filter is used respectively.

Graphics and image processing. In graphics and image processing applications DSPs perform operations on two-dimensional signals, and matrix arithmetic takes on particular significance. In the 320C30 matrix arithmetic can be decomposed into a series of dot products, which can be very effectively implemented using constructs similar to the FIR filter implementation discussed earlier. Additionally, the large memory space of the 320C30 allows processing of large segments of data at a time.

Benchmarks. We have implemented several general-purpose and applications-oriented routines for the 320C30 and include these in the *User's Guide*.[6] Table 4 lists some of these routines with the necessary cycles and the memory requirements for the program.

The last five years have seen a tremendous growth in the utility of digital signal processors. This growth has been fueled, at least in part, by the ever-increasing level of performance and ease of use of general-purpose DSPs. The TMS320C30 represents the newest generation of DSPs. But, the end of this trend is not yet in sight. Rather, we expect the trend of higher levels of performance and greater ease of use to continue. For DSPs, the next five years look bright indeed.

Table 3.
Timing of an FFT on the 320C30.

Number of points	Radix-2 (complex)	Radix-4 (complex)	Radix-2 (real)
FFT timing (ms)			
64	0.167	0.123	0.075
128	0.367	—	0.162
256	0.801	0.624	0.354
512	1.740	—	0.771
1,024	3.750	3.040	1.670
Code size (Words)	55	176	86

The code size does not include the sine/cosine tables. The timing does not include bit reversal or data I/O.

Table 4.
Program memory and timing requirements for 320C30 routines.

Application	Words	Cycles (best case/ worst case)
Inverse of a floating-point number	31	31
Integer division	27	27/58
Double-precision integer multiplication	24	20/24
Square root	32	35
Dot product of two vectors	10	$8 + (N - 1)$
Matrix times vector operation	10	$2 + R(C + 9)$
FIR filter	5	$7 + (N - 1)$
IIR filter (one biquad)	7	7
IIR filter ($N > 1$ biquads)	16	$19 + 6N$
LMS adaptive filter	9	$8 + 3(N - 1)$
LPC lattice filter	11	$9 + 5(P - 1)$
Inverse LPC lattice filter	9	$9 + 3(P - 1)$
μ-law compression	16	16
μ-law expansion	13	11/16
A-law compression	18	18
A-law expansion	15	14/21

N = length of appropriate vector
P = length of lattice filter
R = number of rows of a matrix
C = number of columns of a matrix

```
;
;   GENERIC PROGRAM TO DO A LOOPED-CODE RADIX-2 FFT COMPUTATION IN 320C30.
;
; THE PROGRAM IS ADAPTED FROM THE FORTRAN PROGRAM IN PAGE 111 OF
;     REFERENCE [5]
;
; AUTHOR: PANOS E. PAPAMICHALIS
;         TEXAS INSTRUMENTS                          JULY 16, 1987
;
        .GLOBL  N               ; FFT SIZE
        .GLOBL  M               ; LOG2(N)
        .GLOBL  SINE            ; ADDRESS OF SINE TABLE
        .BSS    INP,1024        ; MEMORY WITH INPUT/OUTPUT DATA

        .TEXT

;   INITIALIZE
        .WORD   FFT             ; STARTING LOCATION OF THE PROGRAM
        .SPACE  100             ; RESERVE 100 WORDS FOR VECTORS, ETC.
FFTSIZ  .WORD   N
LOGFFT  .WORD   M
SINTAB  .WORD   SINE
INPUT   .WORD   INP

FFT:    LDP     FFTSIZ          ; COMMAND TO LOAD DATA PAGE POINTER
        LDI     @FFTSIZ,IR1
        LSH     -2,IR1          ; IR1=N/4, POINTER FOR SIN/COS TABLE
        LDI     0,AR6           ; AR6 HOLDS THE CURRENT STAGE NUMBER
        LDI     @FFTSIZ,IR0
        LSH     1,IR0           ; IR0=2*N1 (BECAUSE OF REAL/IMAG)
        LDI     @FFTSIZ,R7      ; R7=N2
        LDI     1,AR7           ; INITIALIZE REPEAT COUNTER OF FIRST LOOP
        LDI     1,AR5           ; INITIALIZE IE INDEX (AR5=IE)

;   OUTER LOOP
LOOP:   NOP     *++AR6(1)       ; CURRENT FFT STAGE
        LDI     @INPUT,AR0      ; AR0 POINTS TO X(I)
        ADDI    R7,AR0,AR2      ; AR2 POINTS TO X(L)
        LDI     AR7,RC
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #

; BUTTERFLY WITHOUT TWIDDLE FACTORS
        RPTB    BLK1
        ADDF    *AR0,*AR2,R0            ; R0=X(I)+X(L)
        SUBF    *AR2++,*AR0++,R1        ; R1=X(I)-X(L)
        ADDF    *AR2,*AR0,R2            ; R2=Y(I)+Y(L)
        SUBF    *AR2,*AR0,R3            ; R3=Y(I)-Y(L)
        STF     R2,*AR0--               ; Y(I)=R2  AND...
||      STF     R3,*AR2--               ; Y(L)=R3
BLK1    STF     R0,*AR0++(IR0)          ; X(I)=R0  AND...
||      STF     R1,*AR2++(IR0)          ; X(L)=R1 AND AR0,2 = AR0,2 + 2*N1
; IF THIS IS THE LAST STAGE, YOU ARE DONE
        CMPI    @LOGFFT,AR6
        BZD     END

; MAIN INNER LOOP
        LDI     2,AR1                   ; INIT LOOP COUNTER FOR INNER LOOP
        LDI     @SINTAB,AR4             ; INITIALIZE IA INDEX (AR4=IA)
INLOP:  ADDI    AR5,AR4                 ; IA=IA+IE; AR4 POINTS TO COSINE
        LDI     AR1,AR0
        ADDI    2,AR1                   ; INCREMENT INNER LOOP COUNTER
        ADDI    @INPUT,AR0              ; (X(I),Y(I)) POINTER
        ADDI    R7,AR0,AR2              ; (X(L),Y(L)) POINTER
        LDI     AR7,RC
        SUBI    1,RC            ; RC SHOULD BE ONE LESS THAN DESIRED #
        LDF     *AR4,R6                 ; R6=SIN

; GENERAL  BUTTERFLY
        RPTB    BLK2
        SUBF    *AR2,*AR0,R2            ; R2=X(I)-X(L)
        SUBF    *++AR2,*++AR0,R1        ; R1=Y(I)-Y(L)
        MPYF    R2,R6,R0                ; R0=R2*SIN AND...
||      ADDF    *++AR2,*++AR0,R3        ; R3=Y(I)+Y(L)
        MPYF    R1,*++AR4(IR1),R3       ; R3=R1*COS AND...
```

Figure 9. Example of a radix-2, decimation-in-frequency FFT.

```
||      STF     R3,*++AR0               ; Y(I)=Y(I)+Y(L)
        SUBF    R0,R3,R4                ; R4=R1*COS-R2*SIN
        MPYF    R1,R6,R0                ; R0=R1*SIN AND...
||      ADDF    *AR2,*AR0,R3            ; R3=X(I)+X(L)
        MPYF    R2,*++AR4(IR1),R3       ; R3=R2*COS AND...
||      STF     R3,*AR0++(IR0)          ; X(I)=X(I)+X(L) AND AR0=AR0+2*N1
        ADDF    R0,R3,R5                ; R5=R2*COS+R1*SIN
BLK2    STF     R5,*AR2++(IR0)          ; X(L)=R2*COS+R1*SIN, INCR AR2
AND...
||      STF     R4,*++AR2               ; Y(L)=R1*COS-R2*SIN

        CMPI    R7,AR1
        BNE     INLOP                     ; LOOP BACK TO THE INNER LOOP

        LSH     1,AR7                   ; INCREMENT LOOP COUNTER FOR NEXT TIME
        LSH     1,AR5                     ; IE=2*IE
        LDI     R7,IR0                    ; N1=N2
        LSH     -1,R7                     ; N2=N2/2
        BR      LOOP                      ; NEXT FFT STAGE
END     NOP
        .END
```

Figure 9 (cont'd.)

References

1. K.-S. Lin, G.A. Frantz, and R. Simar, "The TMS320 Family of Digital Signal Processors," *Proc. IEEE*, Vol. 75, No. 9, Sept. 1987, pp. 1143-1159.
2. R. Simar and A. Davis, "The Application of High-Level Languages to Single-Chip Digital Signal Processors," *Proc. 1988 Int'l. Conf. Acoustics, Speech, and Signal Processing*, Apr. 1988, pp. 1678-1681.
3. A. Oppenheim and R. Schafer, *Digital Signal Processing*, Prentice Hall, Englewood Cliffs, N.J., 1975, 585 pp.
4. L. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, Prentice Hall, 1975, 762 pp.
5. C.S. Burrus and T.W. Parks, *DFT/FFT and Convolution Algorithms*, John Wiley & Sons, New York, 1985, 232 pp.
6. *TMS320C30 User's Guide*, Texas Instruments, Dallas, Tex., 1988.
7. P. Papamichalis, "FFT Implementation on the TMS320C30," *Proc. 1988 Int'l. Conf. on Acoustics, Speech, and Signal Processing*, Apr. 1988, pp. 1399-1402.

Panos Papamichalis is a senior member of the technical staff and a section manager in the Texas Instruments DSP Applications Group. He is also an adjunct professor for the Electrical and Computer Engineering Department at Rice University in Houston. Author of *Practical Approaches to Speech Coding*, his interests include digital signal processing with applications to speech processing and telecommunications.

Papamichalis received his engineering degree from the School of Mechanical and Electrical Engineering, National Technical University of Athens. His MS and PhD degrees in electrical engineering come from the Georgia Institute of Technology in Atlanta. He is a member of the Institute of Electrical and Electronics Engineers and Sigma Xi.

Ray Simar, Jr. is a group member of the TI Semiconductor technical staff and the principal architect and program manager of the TMS320C30. He has supported the TMS320 family of digital signal processors.

Simar holds a BS degree in bioengineering from Texas A&M University, College Station, and an MSEE from Rice University. He is a member of Tau Beta Pi, Phi Eta Sigma, and Phi Kappa Phi.

Questions concerning this article can be directed to Panos Papamichalis, Texas Instruments, Inc., PO Box 1443, M/S 701, Houston, TX 77251-1443.