CHAPTER 3

DSP MICROPROCESSORS IN EMBEDDED SYSTEMS

The term embedded system is often used to refer to a processor and associated circuits required to perform a particular function that is not the sole purpose of the overall system. For example, a keyboard controller on a computer system may be an embedded system if it has a processor that handles the keyboard activity for the computer system. In a similar fashion, digital signal processors are often embedded in larger systems to perform specialized DSP operations to allow the overall system to handle general purpose tasks. A special purpose processor used for voice processing, including analog-to-digital (A/D) and digital-to-analog (D/A) converters, is an embedded DSP system when it is part of a personal computer system. Often this type of DSP runs only one application (perhaps speech synthesis or recognition) and is not programmed by the end user. The fact that the processor is embedded in the computer system may be unknown to the end user.

A DSP's data format, either fixed-point or floating-point, determines its ability to handle signals of differing precision, dynamic range, and signal-to-noise ratios. Also, ease-of-use and software development time are often equally important when deciding between fixed-point and floating-point processors. Floating-point processors are often more expensive than similar fixed-point processors but can execute more instructions per second. Each instruction in a floating-point processor may also be more complicated, leading to fewer cycles per DSP function. DSP microprocessors can be classified as fixed-point processors if they can only perform fixed-point multiplies and adds, or as floating-point processors if they can perform floating-point operations.

The precision of a particular class of A/D and D/A converters (classified in terms of cost or maximum sampling rate) has been slowly increasing at a rate of about one bit every two years. At the same time the speed (or maximum sampling rate) has also been increasing. The dynamic range of many algorithms is higher at the output than at the input and intermediate results are often not constrained to any particular dynamic range. This requires that intermediate results be scaled using a shift operator when a fixed-point DSP is used. This will require more cycles for a particular algorithm in fixed-point than on an equal floating-point processor. Thus, as the A/D and D/A requirements for a particular application require higher speeds and more bits, a fixed-point DSP may need to be replaced with a faster processor with more bits. Also, the fixed-point program may require extensive modification to accommodate the greater precision.

In general, floating-point DSPs are easier to use and allow a quicker time-to-market than processors that do not support floating-point formats. The extent to which this is true depends on the architecture of the floating-point processor. High-level language programmability, large address spaces, and wide dynamic range associated with floating-point processors allow system development time to be spent on algorithms and signal processing problems rather than assembly coding, code partitioning, quantization error, and scaling. In the remainder of this chapter, floating-point digital signal processors and the software required to develop DSP algorithms are considered in more detail.

3.1 TYPICAL FLOATING-POINT DIGITAL SIGNAL PROCESSORS

This section describes the general properties of the following three floating-point DSP processor families: AT&T DSP32C and DSP3210, Analog Devices ADSP-21020 and ADSP-21060, and Texas Instruments TMS320C30 and TMS320C40. The information was obtained from the manufacturers' data books and manuals and is believed to be an accurate summary of the features of each processor and the development tools available. Detailed information should be obtained directly from manufacturers, as new features are constantly being added to their DSP products. The features of the three processors are summarized in sections 3.1.1, 3.1.2, and 3.1.3.

The execution speed of a DSP algorithm is also important when selecting a processor. Various basic building block DSP algorithms are carefully optimized in assembly language by the processor's manufacturer. The time to complete a particular algorithm is often called a benchmark. Benchmark code is always in assembly language (sometimes without the ability to be called by a C function) and can be used to give a general measure of the maximum signal processing performance that can be obtained for a particular processor. Typical benchmarks for the three floating-point processor families are shown in the following table. Times are in microseconds based the on highest speed processor available at publication time.

		DSP32C DSP3210	ADSP21020 ADSP21060	TMS320C30	TMS320C40
Maximum Instruction Cycle Speed (MIPs)		20	40	20	30
1024 Complex FFT with bitreverse	cycles	161311*	19245	40457	38629
	time	2016.4	481.13	2022.85	1287.63
FIR Filter (35 Taps)	cycles	187*	44	45	42
	time	2.34	1.1	2.25	1.4
IIR Filter (2 Biquads)	cycles	85*	14	23	21
	time	1.06	0.35	1.15	0.7
4x4 * 4x1 Matrix Multiply	cycles	80*	24	58	37
	time	1.0	0.6	2.9	1.23

*Cycle counts for DSP32C and DSP3210 are clock cycles including wait states (1 instruction = 4 clock cycles).

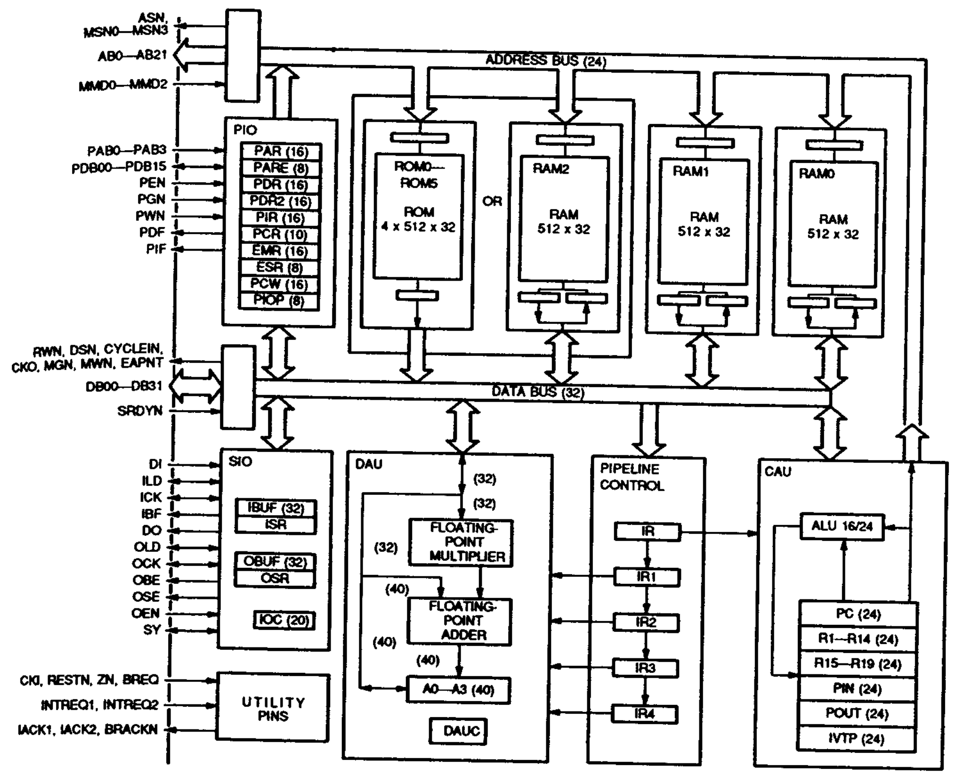
3.1.1 AT&T DSP32C and DSP3210

Figure 3.1 shows a block diagram of the DSP32C microprocessor manufactured by AT&T (Allentown, PA). The following is a brief description of this processor provided by AT&T.

The DSP32C's two execution units, the control arithmetic unit (CAU) and the data arithmetic unit (DAU), are used to achieve the high throughput of 20 million instructions per second (at the maximum clock speed of 80 MHz). The CAU performs 16- or 24-bit integer arithmetic and logical operations, and provides data move and control capabilities. This unit includes 22 general purpose registers. The DAU performs 32-bit floating-point arithmetic for signal processing functions. It includes a 32-bit floating-point multiplier, a 40-bit floating-point adder, and four 40-bit accumulators. The multiplier and the adder work in parallel to perform 25 million floating-point computations per second. The DAU also incorporates special-purpose hardware for data-type conversions.

On-chip memory includes 1536 words of RAM. Up to 16 Mbytes of external memory can be directly addressed by the external memory interface that supports wait states and bus arbitration. All memory can be addressed as 8-, 16- or 32-bit data, with the 32-bit data being accessed at the same speed as 8- or 16-bit data.

The DSP32C has three I/O ports: an external memory port, a serial port and a 16-bit parallel port. In addition to providing access to commercially available memory, the external memory interface can be used for memory mapped I/O. The serial port can interface to a time division multiplexed (TDM) stream, a codec, or another DSP32C. The parallel port provides an interface to an external microprocessor. In summary, some of the key features of the DSP32C follow.



LEGEND*:

A0—A3	Accumulators 0—3	ISR	Input shift register	PDR2	PIO data register 2
ALU	Arithmetic logic unit	IVTP	Interrupt vector table pointer	PIN	Serial DMA input pointer
CAU	Control arithmetic unit	OBUF	Output buffer	PIO	Parallel I/O unit
DAU	Data arithmetic unit	OSR	Output shift register	PIOP	Parallel I/O port register
DAUC	DAU control register	PAR	PIO address register	PIR	PIO interrupt register
EMR	Error mask register	PARE	PIO address register extended	POUT	Serial DMA output pointer
ESR	Error source register	PC	Program counter	R1—R19	Registers 1—19
IBUF	Input buffer	PCR	PIO control register	RAM	Read/write memory
IOC	Input/output control register	PCW	Processor control word	ROM	Read-only memory
IR	Instruction register	PDR	PIO data register	SIO	Serial I/O unit
IR1—IR4	Instruction register pipeline				

* For a detailed description, see Architecture.

FIGURE 3.1 Block diagram of DSP32C processor (Courtesy AT&T).

KEY FEATURES

- 63 instructions, 3-stage pipeline
- Full 32-bit floating-point architecture for increased precision and dynamic range
- Faster application evaluation and development yielding increased market leverage
- Single cycle instructions, eliminating multicycle branches
- Four memory accesses/instruction for exceptional memory bandwidth
- Easy-to-learn and read C-like assembly language
- Serial and parallel ports with DMA for clean external interfacing
- Single-instruction data conversion for IEEE P754 floating-point, 8-, 16-, and 24-bit integers, μ-law, A-Law and linear numbers
- Fully vectored interrupts with hardware context saving up to 2 million interrupts per second
- Byte-addressable memory efficiently storing 8- and 16-bit data
- Wait-states of 1/4 instruction increments and two independent external memory speed partitions for flexible, cost-efficient memory configurations

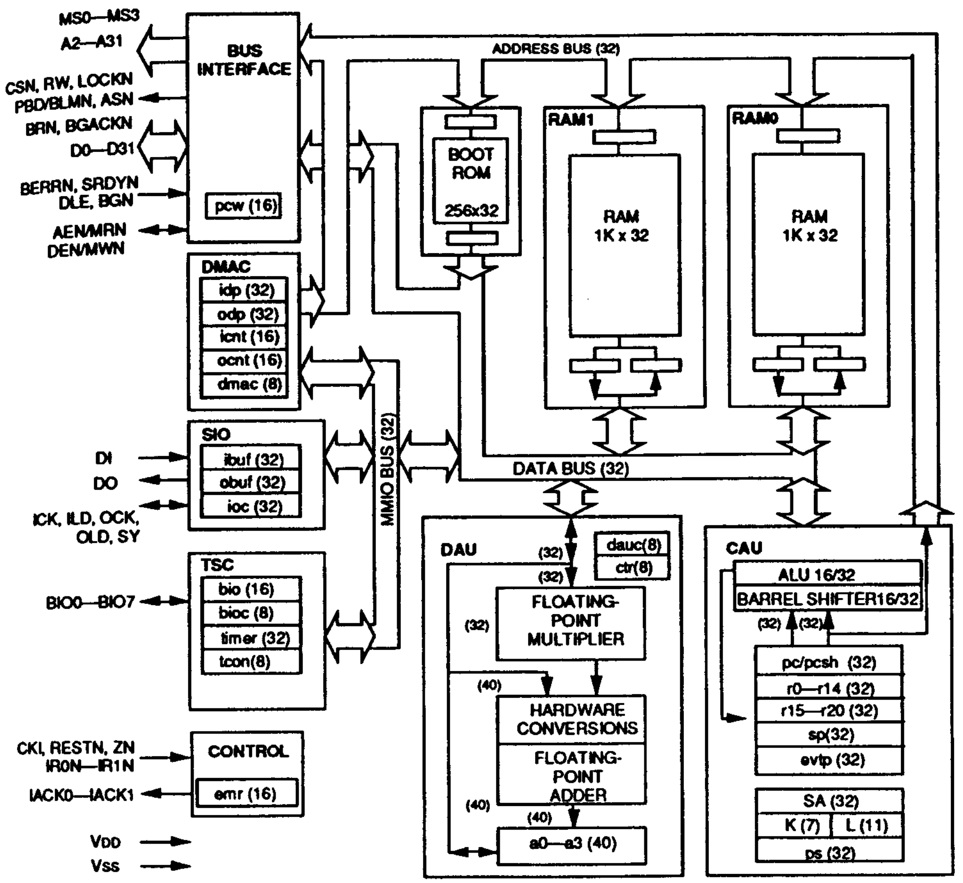
DESCRIPTION OF AT&T HARDWARE DEVELOPMENT HARDWARE

- Full-speed operation of DSP32C (80 MHz)
- Serial I/O through an AT&T T7520 High Precision Codec with mini-BNC connectors for analog input and output
- Provision for external serial I/O through a 34-pin connector
- Upper 8-bits of DSP32C parallel I/O port access via connector
- DSP32C Simulator providing an interactive interface to the development system.
- Simulator-controlled software breakpoints; examining and changing memory contents or device registers

Figure 3.2 shows a block diagram of the DSP3210 microprocessor manufactured by AT&T. The following is a brief description of this processor provided by AT&T.

AT&T DSP3210 DIGITAL SIGNAL PROCESSOR FEATURES

- 16.6 million instructions per second (66 MHz clock)
- 63 instructions, 3-stage pipeline
- Two 4-kbyte RAM blocks, 1-kbyte boot ROM
- 32-bit barrel shifter and 32-bit timer
- Serial I/O port, 2 DMA channels, 2 external interrupts
- Full 32-bit floating-point arithmetic for fast, efficient software development
- C-like assembly language for ease of programming
- All single-cycle instructions; four memory accesses/instruction cycle
- Hardware context save and single-cycle PC relative addressing



LEGEND:

CAU	Control Arithmetic Unit	SIO	Serial Input/Output
DAU	Data Arithmetic Unit	TSC	Timer/Status/Control
RAM	Random Access Memory	DMAC	DMA Controller
ROM	Read Only Memory	MMIO	Memory Mapped I/O

FIGURE 3.2 Block diagram of DSP3210 processor (Courtesy AT&T).

- Microprocessor bus compatibility (The DSP3210 is designed for efficient bus master designs. This allows the DSP3210 to be easily incorporated into microprocessor-based designs
- 32-bit, byte-addressable address space allowing the DSP3210 and a microprocessor to share the same address space and to share pointer values.
- Retry, relinquish/retry, and bus error support
- Page mode DRAM support
- Direct support for both Motorola and Intel signaling

AT&T DSP3210 FAMILY HARDWARE DEVELOPMENT SYSTEM DESCRIPTION

The MP3210 implements one or two AT&T DSP3210 32-bit floating-point DSPs with a comprehensive mix of memory, digital I/O and professional audio signal I/O. The MP3210 holds up to 2 Mbytes of dynamic RAM (DRAM). The DT-Connect interface enables real-time video I/O. MP3210 systems include: the processor card; C Host drivers with source code; demos, examples and utilities, with source code; User's Manual; and the AT&T DSP3210 Information Manual. DSP3210 is the low-cost Multimedia Processor of Choice. New features added to the DSP3210 are briefly outlined below.

DSP3210 FEATURES	USER BENEFITS
• Speeds up to 33 MFLOPS	The high performance and large on-chip memory space enable
• 2k × 32 on-chip RAM	fast, efficient processing of complex algorithms.
• Full, 32-bit, floating-point	Ease of programming/higher performance.
• All instructions are single cycle	
• Four memory accesses per instruction cycle	Higher performance.
• Microprocessor bus compatibility • 32-bit byte-addressable designs. • Retry, relinquish/retry • error support • Boot ROM • Page mode DRAM support • Directly supports 680X0 and 80X86 signaling	Designed for efficient bus master. This allows the DSP3210 address space to easily be incorporated into µP bus-based designs. The 32-bit, byte-addressable space allows the DSP3210 and a µP to share the same address space and to share pointer values as well.

3.1.2 Analog Devices ADSP-210XX

Figure 3.3 shows a block diagram of the ADSP-21020 microprocessor manufactured by Analog Devices (Norwood, MA). The ADSP-21060 core processor is similar to the ADSP-21020. The ADSP-21060 (Figure 3.4) also includes a large on-chip memory, a DMA controller, serial ports, link ports, a host interface and multiprocessing features

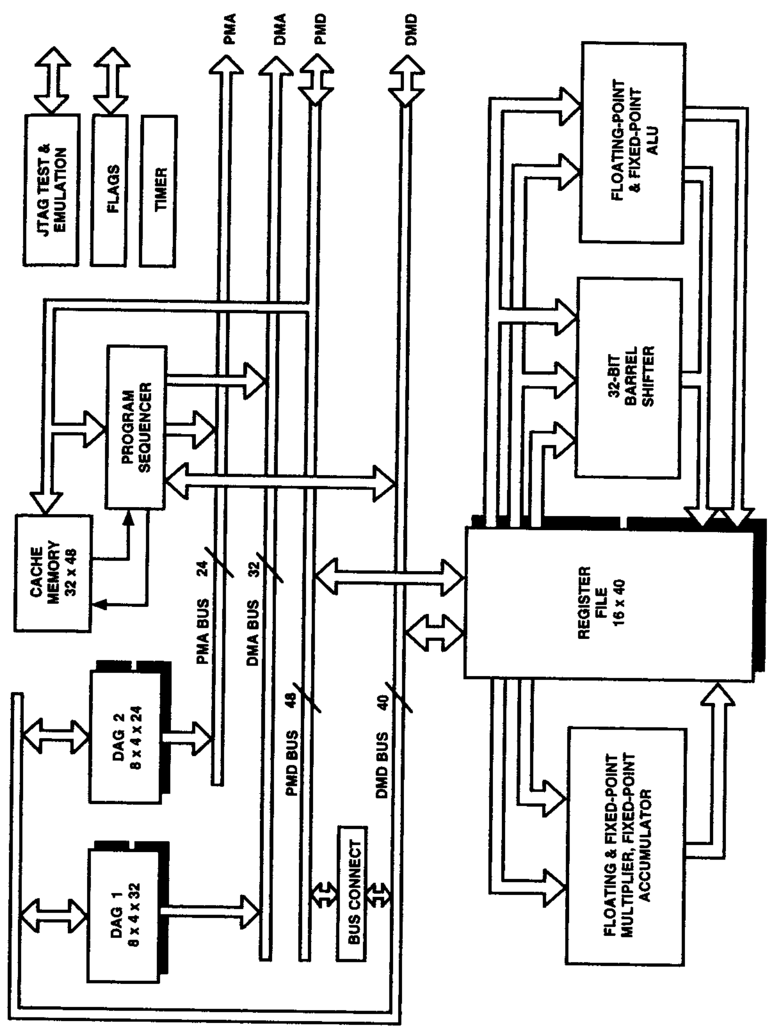


FIGURE 3.3 Block diagram of ADSP-21020 processor (Courtesy Analog Devices.)

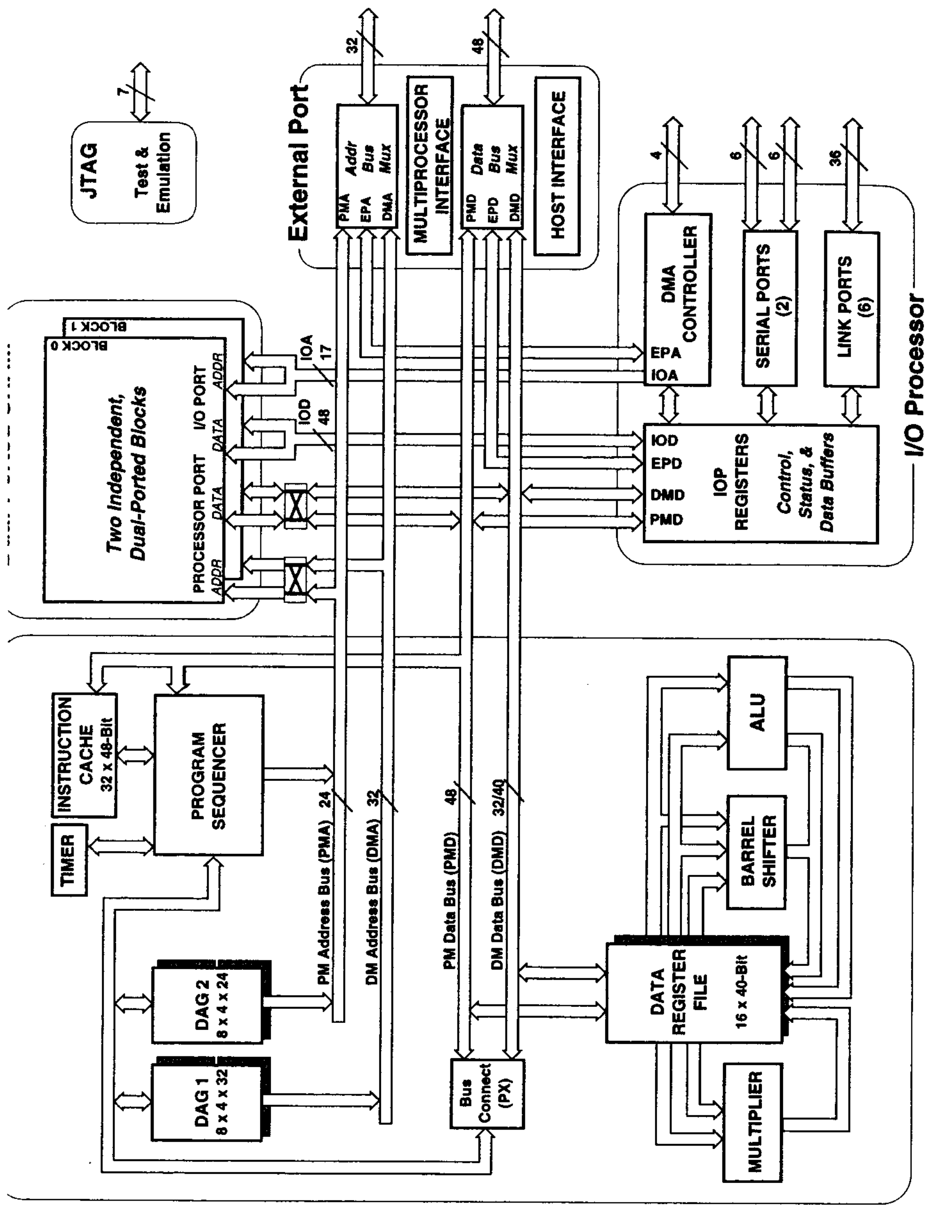


FIGURE 3.4 Block diagram of ADSP-21060 processor (Courtesy Analog Devices.)

with the core processor. The following is a brief description of these processors provided by Analog Devices.

The ADSP-210XX processors provide fast, flexible arithmetic computation units, unconstrained data flow to and from the computation units, extended precision and dynamic range in the computation units, dual address generators, and efficient program sequencing. All instructions execute in a single cycle. It provides one of the fastest cycle times available and the most complete set of arithmetic operations, including seed 1/x, min, max, clip, shift and rotate, in addition to the traditional multiplication, addition, subtraction, and combined addition/subtraction. It is IEEE floating-point compatible and allows interrupts to be generated by arithmetic exceptions or latched status exception handling.

The ADSP-210XX has a modified Harvard architecture combined with a 10-port data register file. In every cycle two operands can be read or written to or from the register file, two operands can be supplied to the ALU, two operands can be supplied to the multiplier, and two results can be received from the ALU and multiplier. The processor's 48-bit orthogonal instruction word supports fully parallel data transfer and arithmetic operations in the same instruction.

The processor handles 32-bit IEEE floating-point format as well as 32-bit integer and fractional formats. It also handles extended precision 40-bit IEEE floating-point formats and carries extended precision throughout its computation units, limiting data truncation errors.

The processor has two data address generators (DAGs) that provide immediate or indirect (pre- and post-modify) addressing. Modulus and bit-reverse addressing operations are supported with no constraints on circular data buffer placement. In addition to zero-overhead loops, the ADSP-210XX supports single-cycle setup and exit for loops. Loops are both nestable (six levels in hardware) and interruptable. The processor supports both delayed and nondelayed branches. In summary, some of the key features of the ADSP-210XX core processor follow:

- 48-bit instruction, 32/40-bit data words
- 80-bit MAC accumulator
- 3-stage pipeline, 63 instruction types
- 32 × 48-bit instruction cache
- 10-port, 32 × 40-bit register file (16 registers per set, 2 sets)
- 6-level loop stack
- 24-bit program, 32-bit data address spaces, memory buses
- 1 instruction/cycle (pipelined)
- 1-cycle multiply (32-bit or 40-bit floating-point or 32-bit fixed-point)
- 6-cycle divide (32-bit or 40-bit floating-point)
- 2-cycle branch delay
- Zero overhead loops
- Barrel shifter

- Algebraic syntax assembly language
- Multifunction instructions with 4 operations per cycle
- Dual address generators
- 4-cycle maximum interrupt latency

3.1.3 Texas Instruments TMS320C3X and TMS320C40

Figure 3.5 shows a block diagram of the TMS320C30 microprocessor and Figure 3.6 shows the TMS320C40, both manufactured by Texas Instruments (Houston, TX). The TMS320C30 and TMS320C40, processors are similar in architecture except that the TMS320C40 provides hardware support for multiprocessor configurations. The following is a brief description of the TMS320C30 processor as provided by Texas Instruments.

The TMS320C30 can perform parallel multiply and ALU operations on integer or floating-point data in a single cycle. The processor also possesses a general-purpose register file, program cache, dedicated auxiliary register arithmetic units (ARAU), internal dual-access memories, one DMA channel supporting concurrent I/O, and a short machine-cycle time. High performance and ease of use are products of these features.

General-purpose applications are greatly enhanced by the large address space, multiprocessor interface, internally and externally generated wait states, two external interface ports, two timers, two serial ports, and multiple interrupt structure. High-level language is more easily implemented through a register-based architecture, large address space, powerful addressing modes, flexible instruction set, and well-supported floating-point arithmetic. Some key features of the TMS320C30 are listed below.

- 4 stage pipeline, 113 instructions
- One 4K × 32-bit single-cycle dual access on-chip ROM block
- Two 1K × 32-bit single-cycle dual access on-chip RAM blocks
- 64 × 32-bit instruction cache
- 32-bit instruction and data words, 24-bit addresses
- 40/32-bit floating-point/integer multiplier and ALU
- 32-bit barrel shifter
- Multiport register file: Eight 40-bit extended precision registers (accumulators)
- Two address generators with eight auxiliary registers and two arithmetic units
- On-chip direct memory access (DMA) controller for concurrent I/O
- Integer, floating-point, and logical operations
- Two- and three-operand instructions
- Parallel ALU and multiplier instructions in a single cycle
- Block repeat capability
- Zero-overhead loops with single-cycle branches
- Conditional calls and returns

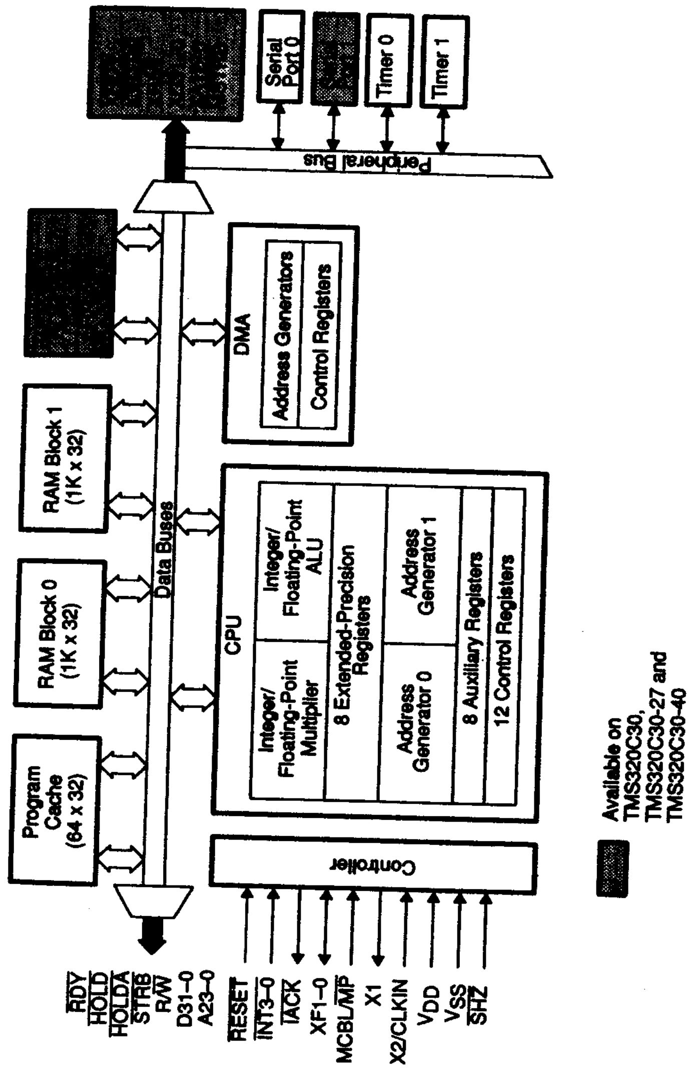


FIGURE 3.5 Block diagram of TMS320C30 and TMS32C31 processor (Courtesy Texas Instruments).

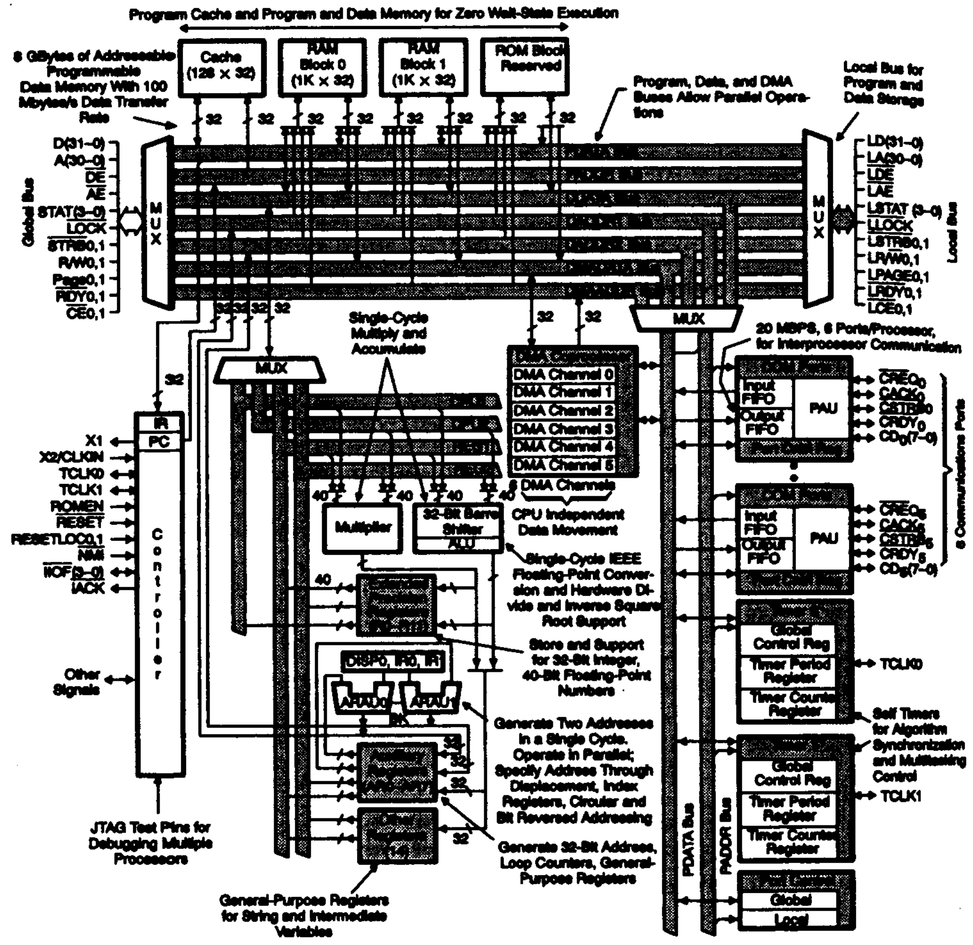

Program Cache and Program and Data Memory for Zero Wait-State Execution
8 GBytes of Addressable Programmable Data Memory With 100 Mbytes/s Data Transfer Rate
Cache (128 × 32)
RAM Block 0 (1K × 32)
RAM Block 1 (1K × 32)
ROM Block Reserved
Program, Data, and DMA Buses Allow Parallel Operations
Local Bus for Program and Data Storage
Global Bus
Local Bus
MUX
Single-Cycle Multiply and Accumulate
DMA Channel 0
DMA Channel 1
DMA Channel 2
DMA Channel 3
DMA Channel 4
DMA Channel 5
6 DMA Channels
CPU Independent Data Movement
20 MBPS, 6 Ports/Processor, for Interprocessor Communication
Input FIFO
Output FIFO
PAU
6 Communications Ports
Multiplier
32-Bit Barrel Shifter ALU
Single-Cycle IEEE Floating-Point Conversion and Hardware Divide and Inverse Square Root Support
Store and Support for 32-Bit Integer, 40-Bit Floating-Point Numbers
DISP0, IR0, IR1
ARAU0
ARAU1
Generate Two Addresses in a Single Cycle. Operate in Parallel; Specify Address Through Displacement, Index Registers, Circular and Bit Reversed Addressing
Generate 32-Bit Address, Loop Counters, General-Purpose Registers
General-Purpose Registers for String and Intermediate Variables
Global Control Reg
Timer Period Register
Timer Counter Register
TCLK0
TCLK1
Self Timers for Algorithm Synchronization and Multitasking Control
Global
Local
PDATA Bus
PADDR Bus
IR
PC
Controller
JTAG Test Pins for Debugging Multiple Processors
Other Signals
X1
X2/CLKIN
TCLK0
TCLK1
ROMEN
RESET
RESETLOC0,1
NMI
IIOF(3-0)
IACK
D(31-0)
A(30-0)
DE
AE
STAT(3-0)
LOCK
STRB0,1
R/W0,1
Page0,1
RDY0,1
CE0,1
LD(31-0)
LA(30-0)
LDE
LAE
LSTAT(3-0)
LLOCK
LSTRB0,1
LR/W0,1
LPAGE0,1
LRDY0,1
LCE0,1
CREQ0
CACK0
CSTRB0
CRDY0
CD0(7-0)
CREQ5
CACK5
CSTRB5
CRDY5
CD5(7-0)

FIGURE 3.6 Block diagram of TMS320C40 processor (Courtesy Texas Instruments).

- Interlocked instructions for multiprocessing support
- Two 32-bit data buses (24- and 13-bit address)
- Two serial ports
- DMA controller
- Two 32-bit timers

3.2 TYPICAL PROGRAMMING TOOLS FOR DSP

The manufacturers of DSP microprocessors typically provide a set of software tools designed to enable the user to develop efficient DSP algorithms for their particular processors. The basic software tools provided include an assembler, linker, C compiler, and simulator. The simulator can be used to determine the detailed timing of an algorithm and then optimize the memory and register accesses. The C compilers for DSP processors will usually generate assembly source code so that the user can see what instructions are generated by the compiler for each line of C source code. The assembly code can then be optimized by the user and then fed into the assembler and linker.

Most DSP C compilers provide a method to add in-line assembly language routines to C programs (see section 3.3.2). This allows the programmer to write highly efficient assembly code for time-critical sections of a program. For example, the autocorrelation function of a sequence may be calculated using a function similar to a FIR filter where the coefficients and the data are the input sequence. Each multiply-accumulate in this algorithm can often be calculated in one cycle on a DSP microprocessor. The same C algorithm may take 4 or more cycles per multiple-accumulate. If the autocorrelation calculation requires 90 percent of the time in a C program, then the speed of the program can be improved by a factor of about 3 if the autocorrelation portion is coded in assembly language and interfaced to the C program (this assumes that the assembly code is 4 times faster than the C source code). The amount of effort required by the programmer to create efficient assembly code for just the autocorrelation function is much less than the effort required to write the entire program in assembly language.

Many DSP software tools come with a library of DSP functions that provide highly optimized assembly code for typical DSP functions such as FFTs and DFTs, FIR and IIR filters, matrix operations, correlations, and adaptive filters. In addition, third parties may provide additional functions not provided by the manufacturer. Much of the DSP library code can be used directly or with small modifications in C programs.

3.2.1 Basic C Compiler Tools

AT&T DSP32C software development tools. The DSP32C's C Compiler provides a programmer with a readable, quick, and portable code generation tool combined with the efficiency provided by the assembly interface and extensive set of library routines. The package provides for compilation of C source code into DSP32 and

DSP32C assembly code, an assembler, a simulator, and a number of other useful utilities for source and object code management. The three forms of provided libraries are:

- libc A subset of the Standard C Library
- libm Math Library
- libap Application Software Library, complete C-callable set of DSP routines.

DSP32C support software library. This package provides assembly-level programming. Primary tools are the assembler, linker/loader, a make utility that provides better control over the assembly and link/load task, and a simulator for program debugging. Other utilities are: library archiver, mask ROM formatter, object file dumper, symbol table lister, object file code size reporter, and EPROM programmer formatter. The SL package is necessary for interface control of AT&T DSP32C Development Systems.

The Application Library has over seven dozen subroutines for arithmetic, matrix, filter, adaptive filter, FFT, and graphics/imaging applications. All files are assembly source and each subroutine has an example test program. Version 2.2.1 adds four routines for sample rate conversion.

AT&T DSP3210 software development tools. This package includes a C language compiler, libraries of standard C functions, math functions, and digital signal processing application functions. A C code usage example is provided for each of the math and application library routines. The C Compiler also includes all of the assembler, simulator, and utility programs found in the DSP3210 ST package. Since the C libraries are only distributed as assembled and then archived ".a" files, a customer may also find the DSP3210-AL package useful as a collection of commented assembly code examples.

DSP3210 support software library. The ST package provides assembly level programming. The primary tools of the package are the assembler, linker/loader, and a simulator for program development, testing, and debugging. A 32C to 3210 assembly code translator assists developers who are migrating from the DSP32C device. Additional utilities are library archiver, mask ROM formatter, object code disassembler, object file dumper, symbol table lister, and object code size reporter. The AT&T Application Software Library includes over ninety subroutines of typical operations for arithmetic, matrix, filter, adaptive filter, FFT, and graphics/imaging applications. All files are assembly source and each subroutine has an example test program.

Analog devices ADSP-210XX C tools. The C tools for the ADSP-21000 family let system developers program ADSP-210XX digital signal processors in ANSI C. Included are the following tools: the G21K C compiler, a runtime library of C functions, and the CBUG C Source-Level Debugger. G21K is Analog Devices' port of GCC, the GNU C compiler from the Free Software Foundation, for the ADSP-21000 family of digital signal processors. G21K includes Numerical C, Analog Devices' numerical processing extensions to the C language based on the work of the ANSI Numerical C Extensions Group (NCEG) subcommittee.

The C runtime library functions perform floating-point mathematics, digital signal processing, and standard C operations. The functions are hand-coded in assembly language for optimum runtime efficiency. The C tools augment the ADSP-21000 family assembler tools, which include the assembler, linker, librarian, simulator, and PROM splitter.

Texas Instruments TMS320C30 C tools. The TMS320 floating-point C compiler is a full-featured optimizing compiler that translates standard ANSI C programs into TMS320C3x/C4x assembly language source. The compiler uses a sophisticated optimization pass that employs several advanced techniques for generating efficient, compact code from C source. General optimizations can be applied to any C code, and target-specific optimizations take advantage of the particular features of the TMS320C3x/C4x architecture. The compiler package comes with two complete runtime libraries plus the source library. The compiler supports two memory models. The small memory model enables the compiler to efficiently access memory by restricting the global data space to a single 64K-word data page. The big memory model allows unlimited space.

The compiler has straightforward calling conventions, allowing the programmer to easily write assembly and C functions that call each other. The C preprocessor is integrated with the parser, allowing for faster compilation. The Common Object File Format (COFF) allows the programmer to define the system's memory map at link time. This maximizes performance by enabling the programmer to link C code and data objects into specific memory areas. COFF also provides rich support for source-level debugging. The compiler package includes a utility that interlists original C source statements into the assembly language output of the compiler. This utility provides an easy method for inspecting the assembly code generated for each C statement.

All data sizes (**`char`**, **`short`**, **`int`**, **`long`**, **`float`**, and **`double`**) are 32 bits. This allows all types of data to take full advantage of the TMS320Cx/C4x's 32-bit integer and floating-point capabilities. For stand-alone embedded applications, the compiler enables linking all code and initialization data into ROM, allowing C code to run from reset.

3.2.2 Memory Map and Memory Bandwidth Considerations

Most DSPs use a Harvard architecture with three memory buses (program and two data memory paths) or a modified Harvard architecture with two memory buses (one bus is shared between program and data) in order to make filter and FFT algorithms execute much faster than standard von Neumann microprocessors. Two separate data and address busses allow access to filter coefficients and input data in the same cycle. In addition, most DSPs perform multiply and addition operations in the same cycle. Thus, DSPs execute FIR filter algorithms at least four times faster than a typical microprocessor with the same MIPS rating.

The use of a Harvard architecture in DSPs causes some difficulties in writing C programs that utilize the full potential of the multiply-accumulate structure and the multiple memory busses. All three manufacturers of DSPs described here provide a method to assign separate physical memory blocks to different C variable types. For example, auto variables that are stored on the heap can be moved from internal memory to external memory by assigning a different address range to the heap memory segment. In the assembly language generated by the compiler the segment name for a particular C variable or array can be changed to locate it in internal memory for faster access or to allow it to be accessed at the same time as the other operands for the multiply or accumulate operation. Memory maps and segment names are used by the C compilers to separate different types of data and improve the memory bus utilization. Internal memory is often used for coefficients (because there are usually fewer coefficients) and external memory is used for large data arrays.

The ADSP-210XX C compiler also supports special keywords so that any C variable or array can be placed in program memory or data memory. The program memory is used to store the program instructions and can also store floating-point or integer data. When the processor executes instructions in a loop, an instruction cache is used to allow the data in program memory (PM) and data in the data memory (DM) to flow into the ALU at full speed. The **pm** keyword places the variable or array in program memory, and the **dm** keyword places the variable or array in data memory. The default for static or global variables is to place them in data memory.

3.2.3 Assembly Language Simulators and Emulators

Simulators for a particular DSP allow the user to determine the performance of a DSP algorithm on a specific target processor before purchasing any hardware or making a major investment in software for a particular system design. Most DSP simulator software is available for the IBM-PC, making it easy and inexpensive to evaluate and compare the performance of several different processors. In fact, it is possible to write all the DSP application software for a particular processor before designing or purchasing any hardware. Simulators often provide profiling capabilities that allow the user to determine the amount of time spent in one portion of a program relative to another. One way of doing this is for the simulator to keep a count of how many times the instruction at each address in a program is executed.

Emulators allow breakpoints to be set at a particular point in a program to examine registers and memory locations, to determine the results from real-time inputs. Before a breakpoint is reached, the DSP algorithm is running at full speed as if the emulator were not present. An in-circuit emulator (ICE) allows the final hardware to be tested at full speed by connecting to the user's processor in the user's real-time environment. Cycle counts can be determined between breakpoints and the hardware and software timing of a system can be examined.

Emulators speed up the development process by allowing the DSP algorithm to run at full speed in a real-time environment. Because simulators typically execute DSP programs several hundred times slower than in real-time, the wait for a program to reach a particular breakpoint in a simulator can be a long one. Real world signals from A/D converters can only be recorded and then later fed into a simulator as test data. Although the test data may test the algorithm performance (if enough test data is available), the timing

of the algorithm under all possible input conditions cannot be tested using a simulator. Thus, in many real-time environments an emulator is required.

The AT&T DSP32C simulator is a line-oriented simulator that allows the user to examine all of the registers and pipelines in the processor at any cycle so that small programs can be optimized before real-time constraints are imposed. A typical computer dialog (user input is shown in bold) using the DSP32C simulator is shown below (courtesy of AT&T):

```
$im: SHOWRW=1
$im: b end
bp set at addr 0x44
$im: run
12 | r000004* ________* ________* ________* |0000: r11 = 0x7f(127)
16 | r000008* ________* ________* w00007c* |0004: * r2 = r111
20 | r00000c**________* ________* r00007c* |0008: a3 = *r2
25 | r000010**________* r5a5a5a* ________* |000c: r101 = * r1
30 | r000014* ________* ________* ________* |0010: NOP
34 | r000018* ________* ________* ________* |0014: r10 = r10 + 0xff81(-127)
38 | r00001c* ________* ________* w000080* |0018: * r3 = r10
42 | r000020**________* ________* r000080* |001c: *r3 = a0 = float(*r3)
47 | r000024**________* ________* ________* |0020: a0 = a3 * a3
52 | r000028* ________* r000074* r000070**|0024: a1 = *r4- + a3 * *r4-
57 | r00002c**________* r000068* r000064**|0028: a2 = *r5- + a3 * *r5-
63 | r000030**w000080**________* ________* |002c: a0 = a0 * a3
69 | r000034* ________* ________* r00006c* |0030: a1 = *r4 + a1 * a3
73 | r000038**________* r00005c* r000058**|0034: a3 = *r6- + a3 * *r6-
79 | r00003c**________* r00007c* r000060**|0038: a2 = *r5 + a2 * *r2
85 | r000040**________* ________* ________* |003c: a1 = a1 + a0
90 | r000044* ________* r000080* ________* |0040: *r7 = a0 = a1 + *r3
breakpoint at end{0x000044} decode:*r7 = a0 = a1 + *r3
$im: r7.f
r7 = 16.000000
$im: nwait.d
nwait = 16
```

In the above dialog the flow of data in the four different phases of the DSP32C instruction cycle are shown along with the assembly language operation being performed. The cycle count is shown on the left side. Register r7 is displayed in floating-point after the breakpoint is reached and the number of wait states is also displayed. Memory reads are indicated with an **r** and memory writes with a **w**. Wait states occurring when the same memory is used in two consecutive cycles are shown with ******.

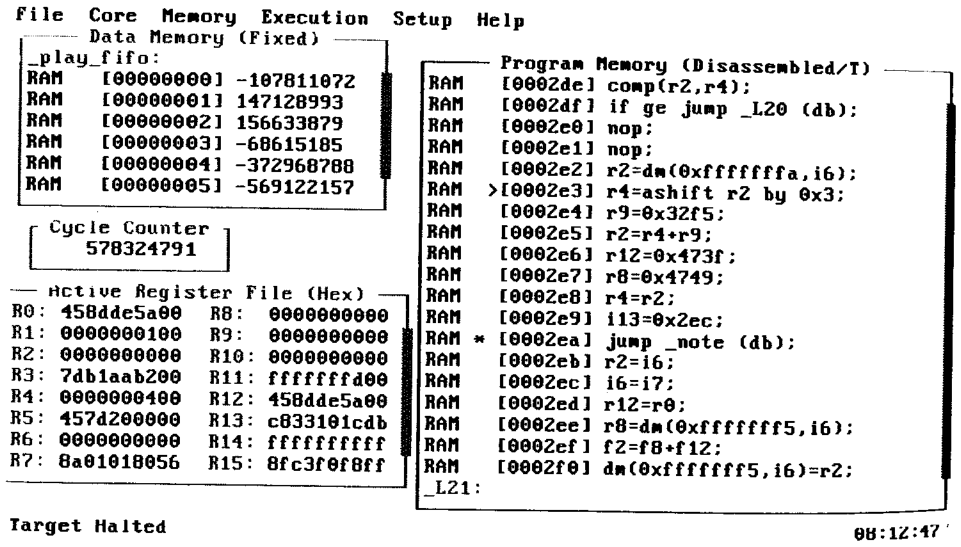
AT&T DSP32C EMULATION SYSTEM DESCRIPTION

- Real-time application development
- Interactive interface to the ICE card(s) provided by the DSP32C simulator through the high-speed parallel interface on the PC Bus Interface Card

- Simulator-controlled software breakpoints; examine and change memory contents or device registers
- For multi-ICE configuration selection of which ICE card is active by the simulator

Figure 3.7 shows a typical screen from the ADSP-21020 screen-oriented simulator. This simulator allows the timing and debug of assembly language routines. The user interface of the simulator lets you completely control program execution and easily change the contents of registers and memory. Breakpoints can also be set to stop execution of the program at a particular point. The ADSP-21020 simulator creates a representation of the device and memory spaces as defined in a system architecture file. It can also simulate input and output from I/O devices using simulated data files. Program execution can be observed on a cycle-by-cycle basis and changes can be made on-line to correct errors. The same screen based interface is used for both the simulator and the in-circuit emulator. A C-source level debugger is also available with this same type of interface (see section 3.3.1).

Figure 3.8 shows a typical screen from the TMS320C30 screen-oriented simulator. This simulator allows the timing of assembly language routines as well as C source code, because it shows the assembly language and C source code at the same time. All of the registers can be displayed after each step of the processor. Breakpoints can also be set to stop execution of the program at a particular point.


```
File  Core  Memory  Execution  Setup  Help
 ---- Data Memory (Fixed) ----
_play_fifo:
RAM   [00000000] -107811072
RAM   [00000001] 147128993
RAM   [00000002] 156633879
RAM   [00000003] -68615185
RAM   [00000004] -372968788
RAM   [00000005] -569122157

 Cycle Counter
    578324791

-- Active Register File (Hex) --
R0: 458dde5a00   R8:  0000000000
R1: 0000000100   R9:  0000000000
R2: 0000000000   R10: 0000000000
R3: 7db1aab200   R11: ffffffd00
R4: 0000000400   R12: 458dde5a00
R5: 457d200000   R13: c833101cdb
R6: 0000000000   R14: ffffffffff
R7: 8a01018056   R15: 8fc3f0f8ff

 ---- Program Memory (Disassembled/T) ----
RAM   [0002de] comp(r2,r4);
RAM   [0002df] if ge jump _L20 (db);
RAM   [0002e0] nop;
RAM   [0002e1] nop;
RAM   [0002e2] r2=dm(0xfffffffa,i6);
RAM  >[0002e3] r4=ashift r2 by 0x3;
RAM   [0002e4] r9=0x32f5;
RAM   [0002e5] r2=r4+r9;
RAM   [0002e6] r12=0x473f;
RAM   [0002e7] r8=0x4749;
RAM   [0002e8] r4=r2;
RAM   [0002e9] i13=0x2ec;
RAM * [0002ea] jump _note (db);
RAM   [0002eb] r2=i6;
RAM   [0002ec] i6=i7;
RAM   [0002ed] r12=r0;
RAM   [0002ee] r8=dm(0xfffffff5,i6);
RAM   [0002ef] f2=f8+f12;
RAM   [0002f0] dm(0xfffffff5,i6)=r2;
_L21:

Target Halted                                            08:12:47
```

FIGURE 3.7 ADSP-21020 simulator displaying assembly language and processor registers (Courtesy Analog Devices.)

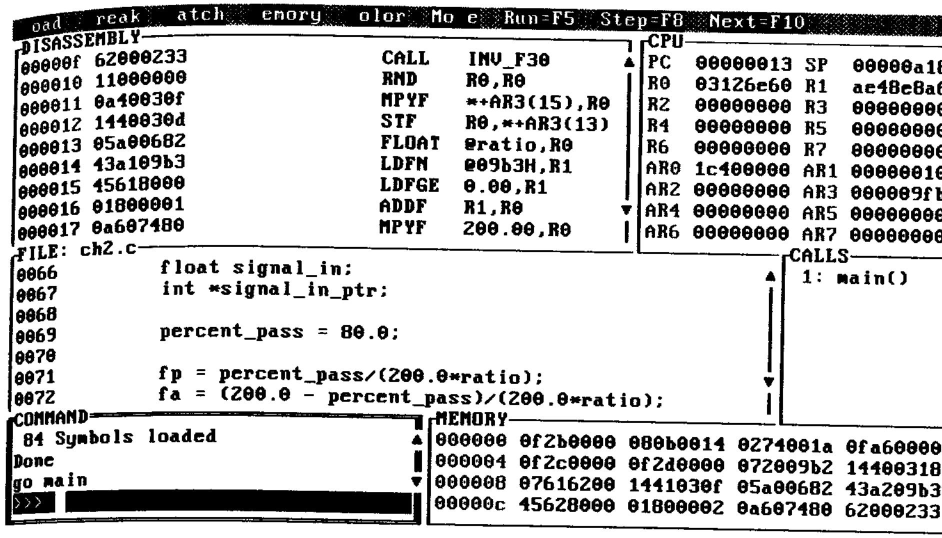

```
 oad  reak  atch  emory  olor  Mo e  Run=F5  Step=F8  Next=F10
DISASSEMBLY                                              CPU
00000f 62000233          CALL    INV_F30        PC  00000013 SP  00000a18
000010 11000000          RND     R0,R0          R0  03126e60 R1  ae48e8a6
000011 0a40030f          MPYF    *+AR3(15),R0   R2  00000000 R3  00000000
000012 1440030d          STF     R0,*+AR3(13)   R4  00000000 R5  00000000
000013 05a00682          FLOAT   @ratio,R0      R6  00000000 R7  00000000
000014 43a109b3          LDFN    @09b3H,R1      AR0 1c400000 AR1 00000010
000015 45618000          LDFGE   0.00,R1        AR2 00000000 AR3 000009fb
000016 01800001          ADDF    R1,R0          AR4 00000000 AR5 00000000
000017 0a607480          MPYF    200.00,R0      AR6 00000000 AR7 00000000
FILE: ch2.c                                           CALLS
0066       float signal_in;                             1: main()
0067       int *signal_in_ptr;
0068
0069       percent_pass = 80.0;
0070
0071       fp = percent_pass/(200.0*ratio);
0072       fa = (200.0 - percent_pass)/(200.0*ratio);
COMMAND                   MEMORY
 84 Symbols loaded        000000 0f2b0000 080b0014 0274001a 0fa60000
Done                      000004 0f2c0000 0f2d0000 072009b2 14400318
go main                   000008 07616200 1441030f 05a00682 43a209b3
>>>                       00000c 45628000 01800002 0a607480 62000233
```

FIGURE 3.8 TMS320C30 simulator in mixed mode assembly language mode (Courtesy Texas Instruments.)

3.3 ADVANCED C SOFTWARE TOOLS FOR DSP

This section describes some of the more advanced software tools available for floating-point DSP microprocessors. Source-level debugging of C source code is described in the next section. Section 3.3.2 describes several assembly language interfaces often used in DSP programming to accelerate key portions of a DSP algorithm. Section 3.3.3 illustrates the numeric C extensions to the C language using DSP algorithms as examples (see Section 2.10 for a description of numeric C).

3.3.1 Source Level Debuggers

Communication Automation & Control (CAC), Inc. (Allentown, PA) offers a debugger for DSP32C assembly language with C-source debugging capability. Both versions are compatible with the following vendors' DSP32C board for the AT computer under MS-DOS: all CAC boards, Ariel, AT&T DSP32C-DS and ICE, Burr-Brown ZPB34, Data Translation, Loughborough Sound Images, and Surrey Medical Imaging Systems. C-source code of the drivers is provided to enable the user to port either debugger to an unsupported DSP32C based board.

Both D3EMU (assembly language only) and D3BUG (C-source and mixed assembly code) are screen-oriented user-friendly symbolic debuggers and have the following features:

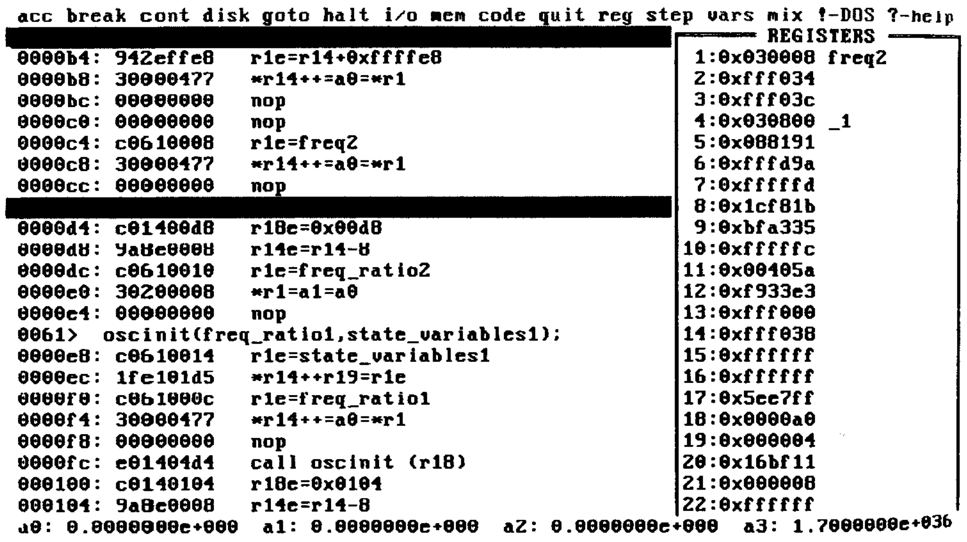
- Single-step, multiple breakpoints, run at full speed
- Accumulator/register/mem displays updated automatically after each step
- Global variables with scrolling
- Stand-alone operation or callable by host application

D3BUG ONLY

- C-source and assembly language mixed listing display modes.
- Local and watch variables.
- Stack trace display.
- Multiple source files/directories with time stamp check.

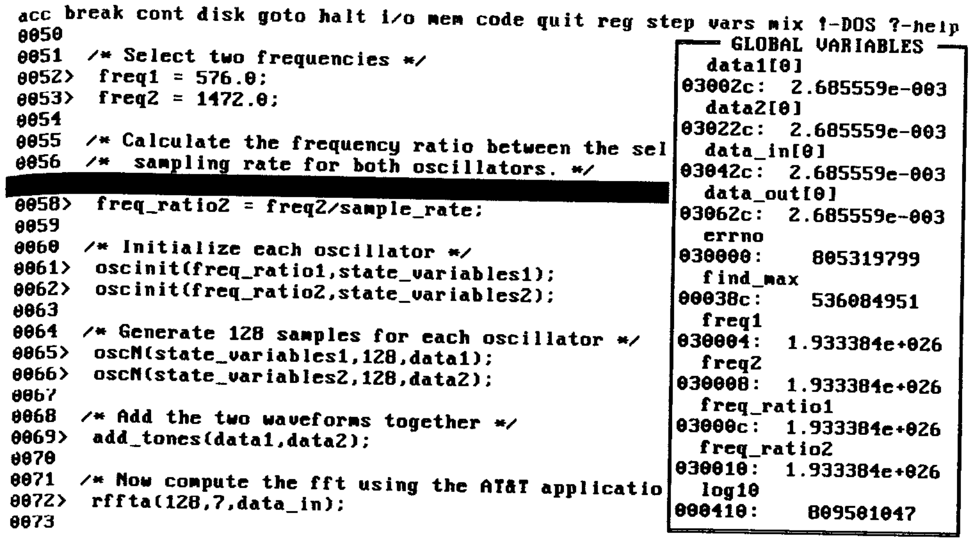
Figure 3.9 shows a typical screen generated using the D3BUG source level debugger with the DSP32C hardware executing the program. Figure 3.9 shows the mixed assembly-C source mode of operation with the DSP32C registers displayed. Figure 3.10 shows the C source mode with the global memory location displayed as the entire C program is executed one C source line at a time in this mode.

Figure 3.11 shows a typical screen from the ADSP-21020 simulator when C source level debugging is being performed using CBUG. C language variables can be displayed and the entire C program can be executed one C source line at a time in this mode. This same type of C source level debug can also be performed using the in-circuit emulator.



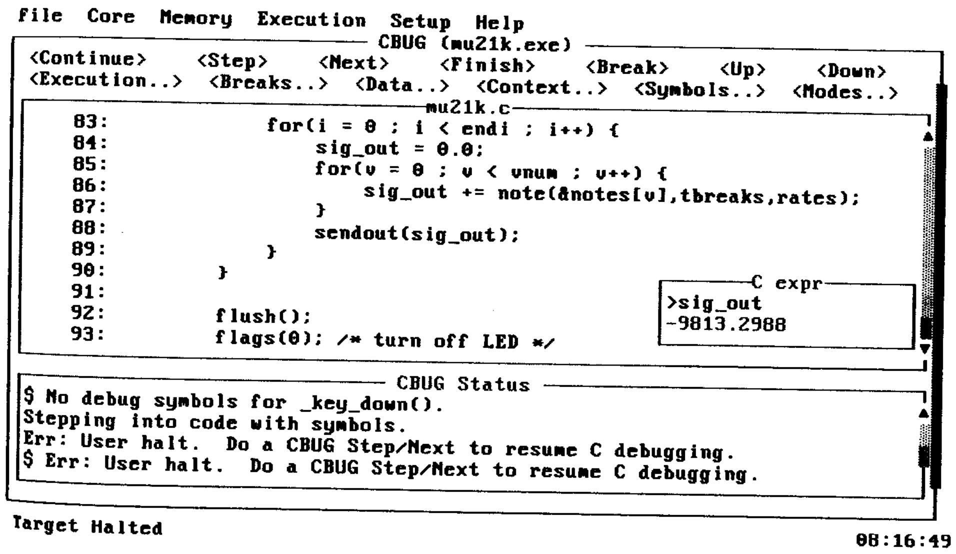
```
acc break cont disk goto halt i/o mem code quit reg step vars mix !-DOS ?-help
                                                          ────── REGISTERS ──────
0000b4: 942effe8   r1e=r14+0xffffe8                        1:0x030008 freq2
0000b8: 30000477   *r14++=a0=*r1                           2:0xfff034
0000bc: 00000000   nop                                     3:0xfff03c
0000c0: 00000000   nop                                     4:0x030800 _1
0000c4: c0610008   r1e=freq2                               5:0x008191
0000c8: 30000477   *r14++=a0=*r1                           6:0xfffd9a
0000cc: 00000000   nop                                     7:0xfffffd
                                                           8:0x1cf81b
0000d4: c01400d8   r18e=0x00d8                             9:0xbfa335
0000d8: 9a8e0008   r14e=r14-8                             10:0xfffffc
0000dc: c0610010   r1e=freq_ratio2                        11:0x00405a
0000e0: 30200008   *r1=a1=a0                              12:0xf933e3
0000e4: 00000000   nop                                    13:0xfff000
0061>  oscinit(freq_ratio1,state_variables1);              14:0xfff038
0000e8: c0610014   r1e=state_variables1                   15:0xffffff
0000ec: 1fe101d5   *r14++r19=r1e                          16:0xffffff
0000f0: c061000c   r1e=freq_ratio1                        17:0x5ee7ff
0000f4: 30000477   *r14++=a0=*r1                          18:0x0000a0
0000f8: 00000000   nop                                    19:0x000004
0000fc: e01404d4   call oscinit (r18)                     20:0x16bf11
000100: c0140104   r18e=0x0104                            21:0x000008
000104: 9a8e0008   r14e=r14-8                             22:0xffffff
a0: 0.0000000e+000  a1: 0.0000000e+000  a2: 0.0000000e+000  a3: 1.7000000e+036
```

FIGURE 3.9 DSP32C debugger D3BUG in mixed assembly C-source mode (Courtesy Communication Automation & Control (CAC), Inc. (Allentown, PA).)



```
acc break cont disk goto halt i/o mem code quit reg step vars mix !-DOS ?-help
0050                                                     ── GLOBAL VARIABLES ──
0051  /* Select two frequencies */                         data1[0]
0052>  freq1 = 576.0;                                    03002c:  2.685559e-003
0053>  freq2 = 1472.0;                                     data2[0]
0054                                                     03022c:  2.685559e-003
0055  /* Calculate the frequency ratio between the sel     data_in[0]
0056  /*  sampling rate for both oscillators. */         03042c:  2.685559e-003
                                                           data_out[0]
0058>  freq_ratio2 = freq2/sample_rate;                  03062c:  2.685559e-003
0059                                                       errno
0060  /* Initialize each oscillator */                   030000:    805319799
0061>  oscinit(freq_ratio1,state_variables1);              find_max
0062>  oscinit(freq_ratio2,state_variables2);            00038c:    536084951
0063                                                       freq1
0064  /* Generate 128 samples for each oscillator */     030004:  1.933384e+026
0065>  oscN(state_variables1,128,data1);                   freq2
0066>  oscN(state_variables2,128,data2);                 030008:  1.933384e+026
0067                                                       freq_ratio1
0068  /* Add the two waveforms together */               03000c:  1.933384e+026
0069>  add_tones(data1,data2);                             freq_ratio2
0070                                                     030010:  1.933384e+026
0071  /* Now compute the fft using the AT&T applicatio     log10
0072>  rffta(128,7,data_in);                             000410:    809501047
0073
```

FIGURE 3.10 DSP32C debugger D3BUG in C-source mode (Courtesy Communication Automation & Control (CAC), Inc. (Allentown, PA).)

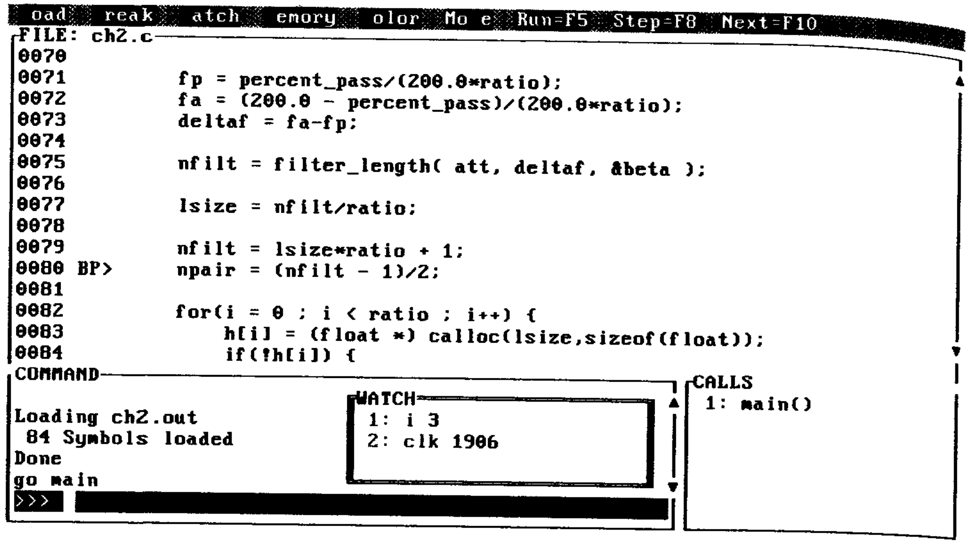


```
File   Core   Memory   Execution   Setup   Help
─────────────────────────── CBUG (mu21k.exe) ───────────────────────────
 <Continue>    <Step>    <Next>    <Finish>    <Break>    <Up>    <Down>
 <Execution..>  <Breaks..>  <Data..>  <Context..>  <Symbols..>  <Modes..>
 ─────────────────────────────── mu21k.c ───────────────────────────────
    83:           for(i = 0 ; i < endi ; i++) {
    84:               sig_out = 0.0;
    85:               for(v = 0 ; v < vnum ; v++) {
    86:                   sig_out += note(&notes[v],tbreaks,rates);
    87:               }
    88:               sendout(sig_out);
    89:           }
    90:       }
    91:                                                    ──── C expr ────
    92:       flush();                                     >sig_out
    93:       flags(0); /* turn off LED */                 -9813.2988

 ────────────────────────────── CBUG Status ─────────────────────────────
 $ No debug symbols for _key_down().
 Stepping into code with symbols.
 Err: User halt.  Do a CBUG Step/Next to resume C debugging.
 $ Err: User halt.  Do a CBUG Step/Next to resume C debugging.

Target Halted
                                                                08:16:49
```

FIGURE 3.11 ADSP-21020 simulator displaying C source code (Courtesy Analog Devices.)



```
FILE: ch2.c

              fp = percent_pass/(200.0*ratio);
              fa = (200.0 - percent_pass)/(200.0*ratio);
              deltaf = fa-fp;

              nfilt = filter_length( att, deltaf, &beta );

              lsize = nfilt/ratio;

              nfilt = lsize*ratio + 1;
     BP>      npair = (nfilt - 1)/2;

              for(i = 0 ; i < ratio ; i++) {
                  h[i] = (float *) calloc(lsize,sizeof(float));
                  if(!h[i]) {
```

```
COMMAND

Loading ch2.out
 84 Symbols loaded
Done
go main
>>>
```

```
WATCH
 1: i 3
 2: clk 1906
```

```
CALLS
 1: main()
```

FIGURE 3.12 TMS320C30 simulator in C-source mode (Courtesy Texas Instruments.)

Figure 3.12 shows a typical screen from the TMS320C30 simulator when C source level debugging is being performed. C language variables can be displayed and the entire C program can be executed one C source line at a time in this mode.

3.3.2 Assembly-C Language Interfaces

The DSP32C/DSP3210 compiler provides a macro capability for in-line assembly language and the ability to link assembly language functions to C programs. In-line assembly is useful to control registers in the processor directly or to improve the efficiency of key portions of a C function. C variables can also be accessed using the optional operands as the following scale and clip macro illustrates:

```
asm void scale(flt_ptr,scale_f,clip)
{
% ureg flt_ptr,scale_f,clip;
a0 = scale_f * *flt_ptr
a1 = -a0 + clip
a0 = ifalt(clip)
*flt_ptr++ = a0 = a0
}
```

Assembly language functions can be easily linked to C programs using several macros supplied with the DSP32C/DSP3210 compiler that define the beginning and the end of the assembly function so that it conforms to the register usage of the C compiler.

The macro **@B** saves the calling function's frame pointer and the return address. The macro **@EO** reads the return address off the stack, performs the stack and frame pointer adjustments, and returns to the calling function. The macros do not save registers used in the assembly language code that may also be used by the C compiler—these must be saved and restored by the assembly code. All parameters are passed to the assembly language routine on the stack and can be read off the stack using the macro **param()**, which gives the address of the parameter being passed.

The ADSP-210XX compiler provides an **asm()** construct for in-line assembly language and the ability to link assembly language functions to C programs. In-line assembly is useful for directly accessing registers in the processor, or for improving the efficiency of key portions of a C function. The assembly language generated by **asm()** is embedded in the assembly language generated by the C compiler. For example, **asm("bit set imask 0x40;")** will enable one of the interrupts in one cycle. C variables can also be accessed using the optional operands as follows:

```
asm("%0=clip %1 by %2;" : "=d" (result) : "d" (x), "d" (y));
```

where **result**, **x** and **y** are C language variables defined in the C function where the macro is used. Note that these variables will be forced to reside in registers for maximum efficiency.

ADSP-210XX assembly language functions can be easily linked to C programs using several macros that define the beginning and end of the assembly function so that it conforms to the register usage of the C compiler. The macro **entry** saves the calling function's frame pointer and the return address. The macro **exit** reads the return address off the stack, performs the stack and frame pointer adjustments, and returns to the calling function. The macros do not save registers that are used in the assembly language code which may also be used by the C compiler—these must be saved and restored by the assembly code. The first three parameters are passed to the assembly language routine in registers r4, r8, and r12 and the remaining parameters can be read off the stack using the macro **reads()**.

The TMS320C30 compiler provides an **asm()** construct for in-line assembly language. In-line assembly is useful to control registers in the processor directly. The assembly language generated by **asm()** is embedded in the assembly language generated by the C compiler. For example, **asm(" LDI @MASK,IE")** will unmask some of the interrupts controlled by the variable **MASK**. The assembly language routine must save the calling function frame pointer and return address and then restore them before returning to the calling program. Six registers are used to pass arguments to the assembly language routine and the remaining parameters can be read off the stack.

3.3.3 Numeric C Compilers

As discussed in section 2.10, numerical C can provide vector, matrix, and complex operations using fewer lines of code than standard ANSI C. In some cases the compiler may be able to perform better optimization for a particular processor. A complex FIR filter can be implemented in ANSI C as follows:

```
typedef struct {
    float real, imag;
} COMPLEX;

COMPLEX float x[1024],w[1024];
COMPLEX *xc,*wc,out;
xc=x;
wc=w;
out.real = 0.0;
out.imag = 0.0;
for(i = 0 ; i < n ; i++) {
    out.real += xc[i].real*wc[i].real - xc[i].imag*wc[i].imag;
    out.imag += xc[i].real*wc[i].imag + xc[i].imag*wc[i].real;
}
```

The following code segment shows the numeric C implementation of the same complex FIR filter:

```
complex float out,x[1024],w[1024];
{
    iter I = n;
    out=sum(x[I]*w[I]);
}
```

The numeric C code is only five lines versus the ten lines required by the standard C implementation. The numeric C code is more efficient, requiring 14 cycles per filter tap versus 17 in the standard C code.

More complicated algorithms are also more compact and readable. The following code segment shows a standard C implementation of a complex FFT without the bit-reversal step (the output data is bit reversed):

```
void fft_c(int n,COMPLEX *x,COMPLEX *w)
{
    COMPLEX u,temp,tm;
    COMPLEX *xi,*xip,*wptr;
    int i,j,le,windex;

    windex = 1;
    for(le=n/2 ; le > 0 ; le/=2) {
        wptr = w;
        for (j = 0 ; j < le ; j++) {
            u = *wptr;
            for (i = j ; i < n ; i = i + 2*le) {
                xi = x + i;
                xip = xi + le;
                temp.real = xi->real + xip->real;
```

```
                temp.imag = xi->imag + xip->imag;
                tm.real = xi->real - xip->real;
                tm.imag = xi->imag - xip->imag;
                xip->real = tm.real*u.real - tm.imag*u.imag;
                xip->imag = tm.real*u.imag + tm.imag*u.real;
                *xi = temp;
            }
            wptr = wptr + windex;
        }
        windex = 2*windex;
    }
}
```

The following code segment shows the numeric C implementation of a complex FFT without the bit-reversal step:

```
void fft_nc(int n, complex float *x, complex float *w)
{
    int size,sect,deg = 1;
    for(size=n/2 ; size > 0 ; size/=2) {
        for(sect=0 ; sect < n ; sect += 2*size) {
            complex float *x1=x+sect;
            complex float *x2=x1+size;
            { iter I=size;
              for(I) {
                complex float temp;
                temp = x1[I] + x2[I];
                x2[I] = (x1[I] - x2[I]) * w[deg*I];
                x1[I] = temp;
              }
            }
        }
        deg *= 2;
    }
}
```

The twiddle factors (**w**) can be initialized using the following numeric C code:

```
void init_w(int n, complex float *w)
{
    iter I = n;
    float a = 2.0*PI/n;
    w[I] = cosf(I*a) + 1i*sinf(I*a);
}
```

Note that the performance of the **init_w** function is almost identical to a standard C implementation, because most of the execution time is spent inside the cosine and sine functions. The numerical C implementation of the FFT also has an almost identical execution time as the standard C version.

4 REAL-TIME SYSTEM DESIGN CONSIDERATIONS

Real-time systems by definition place a hard restriction on the response time to one or more events. In a digital signal processing system the events are usually the arrival of new input samples or the requirement that a new output sample be generated. In this section several real-time DSP design considerations are discussed.

Runtime initialization of constants in a program during the startup phase of a DSP's execution can lead to much faster real-time performance. Consider the pre-calculation of $1/\pi$, which may be used in several places in a particular algorithm. A lengthy divide in each case is replaced by a single multiply if the constant is pre-calculated by the compiler and stored in a static memory area. Filter coefficients can also be calculated during the startup phase of execution and stored for use by the real-time code. The tradeoff that results is between storing the constants in memory which increases the minimum memory size requirements of the embedded system or calculating the constants in real-time. Also, if thousands of coefficients must be calculated, the startup time may become exceeding long and no longer meet the user's expectations. Most DSP software development systems provide a method to generate the code such that the constants can be placed in ROM so that they do not need to be calculated during startup and do not occupy more expensive RAM.

3.4.1 Physical Input/Output (Memory Mapped, Serial, Polled)

Many DSPs provide hardware that supports serial data transfers to and from the processor as well as external memory accesses. In some cases a direct memory access (DMA) controller is also present, which reduces the overhead of the input/output transfer by transferring the data from memory to the slower I/O device in the background of the real-time program. In most cases the processor is required to wait for some number of cycles whenever the processor accesses the same memory where the DMA process is taking place. This is typically a small percentage of the processing time, unless the input or output DMA rate is close to the MIPS rating of the processor.

Serial ports on DSP processors typically run at a maximum of 20 to 30 Mbits/second allowing approximately 2.5 to 4 Mbytes to be transferred each second. If the data input and output are continuous streams of data, this works well with the typical floating-point processor MIPS rating of 12.5 to 40. Only 4 to10 instructions could be executed between each input or output leading to a situation where very little signal processing could be performed.

Parallel memory-mapped data transfers can take place at the MIPs rating of the processor, if the I/O device can accept the data at this rate. This allows for rapid transfers of data in a burst fashion. For example, 1024 input samples could be acquired from a 10 MHz A/D converter at full speed in 100 μsec, and then a FFT power spectrum calculation could be performed for the next 5 msec. Thus, every 5.1 msec the A/D converter's output would be used.

Two different methods are typically used to synchronize the microprocessor with the input or output samples. The first is polling loops and the second is interrupts which are discussed in the next section. Polling loops can be highly efficient when the input and output samples occur at a fixed rate and there are a small number of inputs and outputs. Consider the following example of a single input and single output at the same rate:

```
for(;;) {
    while(*in_status & 1);
    *out = filter(*in)
}
```

It is assumed that the memory addresses of **in**, **out**, and **in_status** have been defined previously as global variables representing the physical addresses of the I/O ports. The data read at **in_status** is bitwise ANDed with 1 to isolate the least significant bit. If this bit is 1, the **while** loop will loop continuously until the bit changes to 0. This bit could be called a "not ready flag" because it indicates that an input sample is not available. As soon as the next line of C code accesses the **in** location, the hardware must set the flag again to indicate that the input sample has been transferred into the processor. After the **filter** function is complete, the returned value is written directly to the output location because the output is assumed to be ready to accept data. If this were not the case, another polling loop could be added to check if the output were ready. The worst case total time involved in the filter function and at least one time through the **while** polling loop must be less than the sampling interval for this program to keep up with the real-time input. While this code is very efficient, it does not allow for any changes in the filter program execution time. If the filter function takes twice as long every 100 samples in order to update its coefficients, the maximum sampling interval will be limited by this larger time. This is unfortunate because the microprocessor will be spending almost half of its time idle in the **while** loop. Interrupt-driven I/O, as discussed in the next section, can be used to better utilize the processor in this case.

3.4.2 Interrupts and Interrupt-Driven I/O

In an interrupt-driven I/O system, the input or output device sends a hardware interrupt to the microprocessor requesting that a sample be provided to the hardware or accepted as input from the hardware. The processor then has a short time to transfer the sample. The interrupt response time is the sum of the interrupt latency of the processor, the time required to save the current context of the program running before the interrupt occurred and the time to perform the input or output operation. These operations are almost always performed in assembly language so that the interrupt response time can be minimized. The advantage of the interrupt-driven method is that the processor is free to perform other tasks, such as processing other interrupts, responding to user requests or slowly changing the parameters associated with the algorithm. The disadvantage of interrupts is the overhead associated with the interrupt latency, context save, and restore associated with the interrupt process.

The following C code example (file INTOUT.C on the enclosed disk) illustrates the functions required to implement one output interrupt driven process that will generate 1000 samples of a sine wave:

```
#include <signal.h>
#include <math.h>
#include "rtdspc.h"

#define SIZE 10

int output_store[SIZE];
int in_inx = 0;
volatile int out_inx = 0;

void sendout(float x);
void output_isr(int ino);

int in_fifo[10000];
int index = 0;

void main()
{
    static float f,a;
    int i,j;
    setup_codec(6);

    for(i = 0 ; i < SIZE-1 ; i++) sendout(i);
    interrupt(SIG_IRQ3, output_isr);

    i = 0;
    j = 1;
    for(;;) {
        for(f=0.0 ; f < 1000.0 ; f += 0.005) {
            sendout(a*sinf(f*PI));
            i += j;
            if(i%25 == 0) {
                a = 100.0*exp(i*5e-5);
                if(a > 30000.0 || i < 0) j = -j;
            }
        }
    }
}
void sendout(float x)
{
```

```
    in_inx++;
    if(in_inx == SIZE) in_inx = 0;
    while(in_inx == out_inx);
    output_store[in_inx] = (int)x;
}
void output_isr(int ino)
{
    volatile int *out = (int *)0x40000002;

    if(index < 10000)
      in_fifo[index++]=16*in_inx+out_inx;

    *out = output_store[out_inx++] << 16;
    if(out_inx == SIZE) out_inx = 0;
}
```

The C function **output_isr** is shown for illustration purposes only (the code is ADSP-210XX specific), and would usually be written in assembly language for greatest efficiency. The functions **sendout** and **output_isr** form a software first-in first-out (FIFO) sample buffer. After each interrupt the output index is incremented with a circular 0-10 index. Each call to **sendout** increments the **in_inx** variable until it is equal to the **out_inx** variable, at which time the output sample buffer is full and the while loop will continue until the interrupt process causes the **out_inx** to advance. Because the above example generates a new **a** value every 25 samples, the FIFO tends to empty during the **exp** function call. The following table, obtained from measurements of the example program at a 48 KHz sampling rate, illustrates the changes in the number of samples in the software FIFO.

Sample Index	in_inx value	out_inx value	Number of Samples in FIFO
0	2	2	10
1	3	3	10
2	4	4	10
3	4	5	9
4	4	6	8
5	4	7	7
6	4	8	6
7	4	9	5
8	7	0	7
9	9	1	8
10	2	2	10

As shown in the table, the number of samples in the FIFO drops from 10 to 5 and then is quickly increased to 10, at which point the FIFO is again full.

3.4.3 Efficiency of Real-Time Compiled Code

The efficiency of compiled C code varies considerably from one compiler to the next. One way to evaluate the efficiency of a compiler is to try different C constructs, such as **case** statements, nested **if** statements, integer versus floating-point data, **while** loops versus **for** loops and so on. It is also important to reformulate any algorithm or expression to eliminate time-consuming function calls such as calls to exponential, square root, or transcendental functions. The following is a brief list of optimization techniques that can improve the performance of compiled C code.

(1) Use of arithmetic identities—multiplies by 0 or 1 should be eliminated whenever possible especially in loops where one iteration has a multiply by 1 or zero. All divides by a constant can also be changed to multiplies.

(2) Common subexpression elimination—repeated calculations of same subexpression should be avoided especially within loops or between different functions.

(3) Use of intrinsic functions—use macros whenever possible to eliminate the function call overhead and convert the operations to in-line code.

(4) Use of register variables—force the compiler to use registers for variables which can be identified as frequently used within a function or inside a loop.

(5) Loop unrolling—duplicate statements executed in a loop in order to reduce the number of loop iterations and hence the loop overhead. In some cases the loop is completely replaced by in-line code.

(6) Loop jamming or loop fusion—combining two similar loops into one, thus reducing loop overhead.

(7) Use post-incremented pointers to access data in arrays rather than subscripted variables (**x=array[i++]** is slow, **x=*ptr++** is faster).

In order to illustrate the efficiency of C code versus optimized assembly code, the following C code for one output from a 35 tap FIR filter will be used:

```
float in[35],coefs[35],y;
main()
{
  register int i;
  register float *x = in, *w = coefs;
  register float out;

  out = *x++ * *w++;
  for(i = 16 ; i-- >= 0; ) {
    out += *x++ * *w++;
    out += *x++ * *w++;
  }
  y=out;
}
```

The FIR C code will execute on the three different processors as follows:

Processor	Optimized C Code Cycles	Optimized Assembly Cycles	Relative Efficiency of C Code (%)
DSP32C	462	187	40.5
ADSP-21020	185	44	23.8
TMS320C30	241	45	18.7

The relative efficiency of the C code is the ratio of the assembly code cycles to the C code cycles. An efficiency of 100 percent would be ideal. Note that this code segment is one of the most efficient loops for the DSP32C compiler but may not be for the other compilers. This is illustrated by the following 35-tap FIR filter code:

```
float in[35],coefs[35],y;
main()
{
  register int i;
  register float *x = in;
  register float *w = coefs;
  register float out;

  out = *x++ * *w++;
  for(i = 0 ; i < 17 ; i++ ) {
    out += *x++ * *w++;
    out += *x++ * *w++;
  }
  y=out;
}
```

This **for**-loop based FIR C code will execute on the three different processors as follows:

Processor	Optimized C Code Cycles	Optimized Assembly Cycles	Relative Efficiency of C Code (%)
DSP32C	530	187	35.3
ADSP-21020	109	44	40.4
TMS320C30	211	45	21.3

Note that the efficiency of the ADSP-21020 processor C code is now almost equal to the efficiency of the DSP32C C code in the previous example.

The complex FFT written in standard C code shown in Section 3.3.3 can be used to