3

Architecture and Instruction Set of the C6x Processor

• Architecture and instruction set of the TMS320C6x processor
• Addressing modes
• Assembler directives
• Linear assembler
• Programming examples using C, assembly, and linear assembly code

3.1 INTRODUCTION

Texas Instruments introduced the first-generation TMS32010 digital signal processor in 1982, the TMS320C25 in 1986 [1], and the TMS320C50 in 1991. Several versions of each of these processors—C1x, C2x, and C5x—are available with different features, such as faster execution speed. These 16-bit processors are all fixed-point processors and are code-compatible.

In a von Neumann architecture, program instructions and data are stored in a single memory space. A processor with a von Neumann architecture can make a read or a write to memory during each instruction cycle. Typical DSP applications require several accesses to memory within one instruction cycle. The fixed-point processors C1x, C2x, and C5x are based on a modified Harvard architecture with separate memory spaces for data and instructions that allow concurrent accesses.

Quantization error or round-off noise from an ADC is a concern with a fixed-point processor. An ADC uses only a best-estimate digital value to represent an input. For example, consider an ADC with a word length of 8 bits and an input range of ±1.5 V. The steps represented by the ADC are: input range/2^8 = 3/256 = 11.72 mV. This produces errors which can be up to ±(11.72 mV)/2 = ±5.86 mV. Only a best estimate can be used by the ADC to represent input values that are not multiples of
With an 8-bit ADC, 2^8 or 256 different levels can represent the input signal. An ADC with a larger word length such as a 16-bit ADC (currently very common) can reduce the quantization error, yielding a higher resolution. The more bits that an ADC has, the better it can represent an input signal.

The TMS320C30 floating-point processor was introduced in the late 1980s. The C31, C32, and the more recent C33 are all members of the C3x family of floating-point processors [2,3]. The C4x floating-point processors, introduced subsequently, are code-compatible with the C3x processors and are based on the modified Harvard architecture [4].

The TMS320C6201 (C62x), announced in 1997, is the first member of the C6x family of fixed-point digital signal processors. Unlike the previous fixed-point processors, C1x, C2x, and C5x, the C62x is based on a very-long-instruction-word (VLIW) architecture, still using separate memory spaces for instructions and data as with the Harvard architecture. The VLIW architecture has simpler instructions, but more are needed for a task than with a conventional DSP architecture.

The C62x is not code-compatible with the previous generation of fixed-point processors. Subsequently, the TMS320C6701 (C67x) floating-point processor was introduced as another member of the C6x family of processors. The instruction set of the C62x fixed-point processor is a subset of the instruction set of the C67x processor. Appendix A contains a list of instructions available on the C6x processors. A recent addition to the family of the C6x processors is the fixed-point C64x.

An application-specific integrated circuit (ASIC) has a DSP core with customized circuitry for a specific application. A C6x processor can be used as a standard general-purpose digital signal processor programmed for a specific application. Specific-purpose digital signal processors are the modem, echo canceler, and others.

A fixed-point processor is better for devices that use batteries, such as cellular phones, since it uses less power than does an equivalent floating-point processor. The fixed-point processors, C1x, C2x, and C5x are 16-bit processors with limited dynamic range and precision. The C6x fixed-point processor is a 32-bit processor with improved dynamic range and precision. In a fixed-point processor, it is necessary to scale the data. Overflow, which occurs when an operation such as the addition of two numbers produces a result with more bits than can fit within a processor’s register, becomes a concern.

A floating-point processor is generally more expensive since it has more “real estate” or is a larger chip because of additional circuitry necessary to handle integer as well as floating-point arithmetic. Several factors, such as cost, power consumption, and speed, come into play when choosing a specific digital signal processor. The C6x processors are particularly useful for applications requiring intensive computations. Family members of the C6x include both fixed-point (e.g., C62x, C64x) and floating-point processors (e.g., C67x). Other digital signal processors are also available, from companies such as Motorola and Analog Devices [5].

Other architectures include the Super Scalar, which requires special hardware to determine which instructions are executed in parallel. The burden is then on the
The TMS320C6711 onboard the DSK is a floating-point processor based on the VLIW architecture [6–9]. Internal memory includes a two-level cache architecture with 4kB of level 1 program cache (L1P), 4kB of level 1 data cache (L1D), and 64kB of RAM or level 2 cache for data/program allocation (L2). It has a glueless (direct) interface to both synchronous memories (SDRAM and SBSRAM) and asynchronous memories (SRAM and EPROM). Synchronous memory requires clocking but provides a compromise between static SRAM and dynamic SDRAM, with SRAM being faster but more expensive than DRAM.

On-chip peripherals include two multichannel buffered serial ports (McBSPs), two timers, a 16-bit host port interface (HPI), and a 32-bit external memory interface (EMIF). It requires 3.3V for I/O and 1.8V for the core (internal). Internal buses include a 32-bit program address bus, a 256-bit program data bus to accommodate eight 32-bit instructions, two 32-bit data address buses, two 64-bit data buses, and two 64-bit store data buses. With a 32-bit address bus, the total memory space is $2^{32} = 4$ GB, including four external memory spaces: CE0, CE1, CE2, and CE3. Figure 3.1 shows a functional block diagram of the C6711 processor included with CCS.

Independent memory banks on the C6x allow for two memory accesses within one instruction cycle. Two independent memory banks can be accessed using two

![Functional block diagram of TMS320C6x](image-url)
independent buses. Since internal memory is organized into memory banks, two
loads or two stores instructions can be performed in parallel. No conflict results if
the data accessed are in different memory banks. Separate buses for program, data,
and direct memory access (DMA) allow the C6x to perform concurrent program
fetches, data read and write, and DMA operations. With data and instructions
residing in separate memory spaces, concurrent memory accesses are possible. The
C6x has a byte-addressable memory space. Internal memory is organized as sep-
parate program and data memory spaces, with two 32-bit internal ports (two 64-bit
ports with the C64x) to access internal memory.

The C6711 on the DSK includes 72kB of internal memory, which starts at
0x00000000, and 16MB of external SDRAM, mapped through CE0 starting at
0x80000000. The DSK also includes 128kB of Flash memory onboard, starting at
0x90000000. A two-level internal memory block diagram is shown in Figure 3.2,
included with CCS [7]. Table 3.1 shows the memory map. A schematic diagram of
the DSK is included with CCS (C6711dsk_schematics.pdf).

With a clock of 150 MHz onboard the DSK, one can ideally achieve two multi-
plies and accumulates per cycle, for a total of 300 million multiplies and accumu-

FIGURE 3.2. Internal memory block diagram (Courtesy of Texas Instruments).
lates (MACs) per second. With six of the eight functional units in Figure 3.1 (not the .D units described below) capable of handling floating-point operations, it is possible to perform 900 million floating-point operations per second (MFLOPS). Operating at 150 MHz, this translates to 1200 million instructions per second (MIPS) with a 6.67-ns instruction cycle time.

### 3.3 FUNCTIONAL UNITS

The CPU consists of eight independent functional units divided into two data paths A and B, as shown in Figure 3.1. Each path has a unit for multiply operations (.M), for logical and arithmetic operations (.L), for branch, bit manipulation, and arithmetic operations (.S), and for loading/storing and arithmetic operations (.D). The .S and .L units are for arithmetic, logical, and branch instructions. All data transfers make use of the .D units.

The arithmetic operations, such as subtract or add (SUB or ADD), can be performed by all the units except the .M units (one from each data path). The eight functional units consist of four floating/fixed-point ALUs (two .L and two .S), two fixed-point ALUs (.D units), and two floating/fixed-point multipliers (.M units). Each functional unit can read directly from or write directly to the register file.
within its own path. Each path includes a set of sixteen 32-bit registers, A0 through A15 and B0 through B15. Units ending in 1 write to register file A, and units ending in 2 write to register file B.

Two cross-paths (1x and 2x) allow functional units from one data path to access a 32-bit operand from the register file on the opposite side. There can be a maximum of two cross-path source reads per cycle. Each functional unit side can access data from the registers on the opposite side using a cross-path (i.e., the functional units on one side can access the register set from the other side). There are 32 general-purpose registers, but some of them are reserved for specific addressing or are used for conditional instructions.

### 3.4 FETCH AND EXECUTE PACKETS

The architecture VELOCITI, introduced by TI, is derived from the VLIW architecture. An execute packet (EP) consists of a group of instructions that can be executed in parallel within the same cycle time. The number of EPs within a fetch packet (FP) can vary from one (with eight parallel instructions) to eight (with no parallel instructions). The VLIW architecture was modified to allow more than one EP to be included within an EP.

The least significant bit of every 32-bit instruction is used to determine if the next or subsequent instruction belongs in the same EP (if 1) or is part of the next EP (if 0). Consider an FP with three EPs: EP1, with two parallel instructions, and EP2 and EP3, each with three parallel instructions, as follows:

```
Instruction A
|| Instruction B
|| Instruction C
|| Instruction D
|| Instruction E

Instruction F
|| Instruction G
|| Instruction H
```

EP1 contains the two parallel instructions A and B; EP2 contains the three parallel instructions C, D, and E; and EP3 contains the three parallel instructions F, G, and H. The FP would be as shown in Figure 3.3. Bit 0 (LSB) of each 32-bit instruction contains a “p” bit that signals whether it is in parallel with a subsequent instruction. For example, the “p” bit of instruction B is zero, denoting that it is not within the same EP as the subsequent instruction C. Similarly, instruction E is not within the same EP as instruction F.
Pipelining is a key feature in a digital signal processor to get parallel instructions working properly, requiring careful timing. There are three stages of pipelining: program fetch, decode, and execute.

1. The program fetch stage is composed of four phases:
   (a) $PG$: program address generate (in the CPU) to fetch an address
   (b) $PS$: program address send (to memory) to send the address
   (c) $PW$: program address ready wait (memory read) to wait for data
   (d) $PR$: program fetch packet receive (at the CPU) to read opcode from memory

2. The decode stage is composed of two phases:
   (a) $DP$: to dispatch all the instructions within an FP to the appropriate functional units
   (b) $DC$: instruction decode

3. The execute stage is composed of from six phases (with fixed point) to 10 phases (with floating point), due to delays (latencies) associated with the following instructions:
   (a) Multiply instruction, which consists of two phases due to one delay
   (b) Load instruction, which consists of five phases due to four delays
   (c) Branch instruction, which consists of six phases due to five delays

Table 3.2 shows the pipeline phases, and Table 3.3 shows the pipelining effects. The first row in Table 3.3 represents cycle 1, 2, . . . , 12. Each subsequent row represents an FP. The rows represented PG, PS, . . . , illustrate the phases associated with each FP. The program generate (PG) of the first FP starts in cycle 1, and the PG of the second FP starts in cycle 2, and so on. Each FP takes four phases for program fetch and two phases for decoding. However, the execution phase can take from 1 to 10 phases (not all execution phases are shown in Table 3.3). We are assuming that each FP contains one execute packet (EP).

For example, at cycle 7, while the instructions in the first FP are in the first execution phase E1 (which may be the only one), the instructions in the second FP are in the decoding phase, the instructions in the third FP are in the dispatching phase, and so on. All seven instructions are proceeding through the various phases. Therefore, at cycle 7, “the pipeline is full.”
Most instructions have one execute phase. Instructions such as multiply (MPY), load (LDH/LDW), and branch (B) take two, five, and six phases, respectively. Additional execute phases are associated with floating-point and double-precision types of instructions, which can take up to 10 phases. For example, the double-precision multiply operation (MPYDP), available on the C67x, has nine delay slots, so that the execution phase takes a total of 10 phases.

The functional unit latency, which represents the number of cycles that an instruction ties up a functional unit, is 1 for all instructions except double-precision instructions, available with the floating-point C67x. Functional unit latency is different from a delay slot. For example, the instruction MPYDP has four functional unit latencies but nine delay slots. This implies that no other instruction can use the associated multiply functional unit for four cycles. A store has no delay slot but finishes its execution in the third execution phase of the pipeline.

If the outcome of a multiply instruction such as MPY is used by a subsequent instruction, a NOP (no operation) must be inserted after the MPY instruction for the pipelining to operate properly. Four or five NOPs are to be inserted in case an instruction uses the outcome of a load or a branch instruction, respectively.

### 3.6 REGISTERS

Two sets of register files, each set with 16 registers, are available: register file A (A0 through A15) and register file B (B0 through B15). Registers A0, A1, B0, B1, and B2 are used as conditional registers. Registers A4 through A7 and B4 through B7 are used for circular addressing. Registers A0 through A9 and B0 through B9 (except B3) are temporary registers. Any of the registers A10 through A15 and
B10 through B15 used are saved and later restored before returning from a subroutine.

A 40-bit data value can be contained across a register pair. The 32 least significant bits (LSBs) are stored in the even register (e.g., A2) and the remaining 8 bits are stored in the 8 LSBs of the next-upper (odd) register (A3). A similar scheme is used to hold a 64-bit double-precision value within a pair of registers (even and odd).

These 32 registers are considered as general-purpose registers. Several special-purpose registers are also available for control and interrupts: for example, the address mode register (AMR) used for circular addressing and interrupt control registers, as shown in Appendix B.

3.7 LINEAR AND CIRCULAR ADDRESSING MODES

Addressing modes determine how one accesses memory. They specify how data are accessed, such as retrieving an operand indirectly from a memory location. Both linear and circular modes of addressing are supported. The most commonly used mode is the indirect addressing of memory.

3.7.1 Indirect Addressing

Indirect addressing can be used with or without displacement. Register R represents one of the 32 registers A0 through A15 and B0 through B15 that can specify or point to memory addresses. As such, these registers are pointers. Indirect addressing mode uses a “*” in conjunction with one of the 32 registers. To illustrate, consider R as an address register.

1. \( *R \). Register R contains the address of a memory location where a data value is stored.

2. \( *R++(d) \). Register R contains the memory address (location). After the memory address is used, R is postincremented (modified), such that the new address is the current address offset by the displacement value d. If \( d = 1 \) (by default), the new address is \( R + 1 \), or R is incremented to the next-higher address in memory. A double minus (\( -- \)) instead of a double plus would update or postdecrement the address to \( R - d \).

3. \( *++R(d) \). The address is preincremented or offset by d, such that the current address is \( R + d \). A double minus would predecrement the memory address so that the current address is \( R - d \).

4. \( *+R(d) \). The address is preincremented by d, such that the current address is \( R + d \) (as with the preceding case). However, in this case, R preincrements without modification. Unlike the previous case, R is not updated or modified.
3.7.2 Circular Addressing

Circular addressing is used to create a circular buffer. This buffer is created in hardware and is very useful in several DSP algorithms, such as in digital filtering or correlation algorithms where data need to be updated. An example in Chapter 4 illustrates the implementation of a digital filter using a circular buffer to update the “delay” samples.

The C6x has dedicated hardware to allow a circular type of addressing. This addressing mode can be used in conjunction with a circular buffer to update samples by shifting data without the overhead created by shifting data directly. As a pointer reaches the end or “bottom” location of a circular buffer that contains the last element in the buffer, and is then incremented, the pointer is automatically wrapped around or points to the beginning or “top” location of the buffer that contains the first element.

Two independent circular buffers are available using BK0 and BK1 within the address mode register (AMR), as shown in Appendix B. The eight registers A4 through A7 and B4 through B7, in conjunction with the two .D units, can be used as pointers (all registers can be used for linear addressing). The following code segment illustrates the use of a circular buffer using register B2 (only side B can be used) to set the appropriate values within AMR:

```
MVK .S2 0x0004, B2 ;lower 16 bits to B2. Select A5 as pointer
MVKLH .S2 0x0005, B2 ;upper 16 bits to B2. Select B0, set N = 5
MVC .S2 B2, AMR ;move 32 bits of B2 to AMR
```

The two move instructions MKV and MKVLH (using the .S unit) move 0x0004 into the 16 LSBs of register B2 and 0x0005 into the 16 MSBs of B2. The MVC (move constant) instruction is the only instruction that can access the AMR and the other control registers (shown in Appendix B) and executes only on the B side in conjunction with the functional units and registers on the side B. A 32-bit value is created in B2, which is then transferred to AMR with the instruction MVC to access AMR [6].

The value 0x0004 = (0100)₂ into the 16 LSBs of AMR sets bit 2 (third bit) to 1 and all other bits to zero. This sets the mode to 01 and selects register A5 as the pointer to a circular buffer using block BK0.

Table 3.4 shows the modes associated with registers A4 through A7 and B4 through B7. The value 0x0005 = (0101)₂ into the 16 MSBs of AMR sets bits 16 and 18 to 1 (other bits to zero). This corresponds to the value of N used to select the size of the buffer as $2^{N+1} = 64$ bytes using BK0. For example, if a buffer size of 128 is desired using BK0, the upper 16 bits of AMR are set to (0110)₂ = 0x0006.

If assembly code is used for the circular buffer, as execution returns to a calling C function, AMR needs to be reinitialized to the default linear mode. Hence the pointer’s address must be saved.
3.8 TMS320C6x INSTRUCTION SET

3.8.1 Assembly Code Format

An assembly code format is represented by the field

```
Label || [ ] Instruction Unit Operands ;comments
```

A label, if present, represents a specific address or memory location that contains an instruction or data. The label must be in the first column. The parallel bars (||) are there if the instruction is being executed in parallel with the previous instruction. The subsequent field is optional to make the associated instruction conditional. Five of the registers—A1, A2, B0, B1, and B2—are available to use as conditional registers. For example, [A2] specifies that the associated instruction executes if A2 is not zero. On the other hand, with ![A2], the associated instruction executes if A2 is zero. All C6x instructions can be made conditional with the registers A1, A2, B0, B1, and B2 by determining when the conditional register is zero. The instruction field can be either an assembler directive or a mnemonic. An assembler directive is a command for the assembler. For example,

```
.word value
```

reserves 32 bits in memory and fill with the specified value. A mnemonic is an actual instruction that executes at run time. The instruction (mnemonic or assembler directive) cannot start in column 1. The Unit field, which can be one of the eight CPU units, is optional. Comments starting in column 1 can begin with either an asterisk or a semicolon, whereas comments starting in any other columns must begin with a semicolon.

Code for the floating-point processors C3x/C4x is not compatible with code for the fixed-point processors C1x, C2x, and C5x/C54x. However, the code for the fixed-point C62x is compatible with the code for the floating-point C67x. C62x code is actually a subset of C67x code. Additional instructions to handle double-precision and floating-point operations are available only on the C67x processor (some additional instructions are also available on the fixed-point C64x processor).
Several code segments are presented to illustrate the C6x instruction set. Assembly code for the C6x processors is very similar to C3x/C4x code. Single-task types of instructions available for the C62x/C67x make it easier to program than either the previous generation of fixed- or floating-point processors. This contributes to an efficient compiler. Additional instructions available on the C64x (but not on the C62x) resemble the multitask types of instructions for C3x/C4x processors. It is very instructive to read the comments in the programs discussed in this book. Appendix B contains a list of the instructions for the C62x/C67x processors.

### 3.8.2 Types of Instructions

The following illustrates some of the syntax of assembly code. It is optional to specify the eight functional units, although this can be useful during debugging and for code efficiency and optimization, discussed in Chapter 8.

1. **Add/Subtract/Multiply**

   (a) The instruction

   ```
   ADD .L1 A3, A7, A7 ; add A3 + A7 $\rightarrow$ A7 (accum in A7)
   ```

   adds the values in registers A3 and A7 and places the result in register A7. The unit .L1 is optional. If the destination or result is in B7, the unit would be .L2.

   (b) The instruction

   ```
   SUB .S1 A1, 1, A1 ; subtract 1 from A1
   ```

   subtracts 1 from A1 to decrement it, using the .S unit.

   (c) The parallel instructions

   ```
   MPY .M2 A7, B7, B6 ; multiply 16 LSBs of A7, B7 $\rightarrow$ B6
   || MPYH .M1 A7, B7, A6 ; multiply 16 MSBs of A7, B7 $\rightarrow$ A6
   ```

   multiplies the lower or least significant 16 bits (LSBs) of both A7 and B7 and places the product in B6, in parallel (concurrently within the same execution packet) with a second instruction that multiplies the higher or most significant 16 bits (MSBs) of A7 and B7 and places the result in A6. In this fashion, two multiply/accumulate operations can be executed within a single instruction cycle. This can be used to decompose a sum of products into two sets of sum of products: one set using the lower 16 bits to operate on the first, third, fifth, . . . number, and another set using the
higher 16 bits to operate on the second, fourth, sixth, . . . number. Note that the parallel symbol is not in column 1.

2. **Load/Store**
   
   (a) The instruction

   ```
   LDH .D2 *B2++,B7 ;load (B2) → B7, increment B2
   || LDH .D1 *A2++,A7 ;load (A2) → A7, increment A2
   ```

   loads into B7 the half-word (16 bits) whose address in memory is specified/pointed by B2. Then register B2 is incremented (postincremented) to point at the next-higher memory address. In parallel is another indirect addressing mode instruction to load into A7 the content in memory, whose address is specified by A2. Then A2 is incremented to point at the next-higher memory address.

   The instruction `LDW` loads a 32-bit word. Two paths using `.D1` and `.D2` allow for the loading of data from memory to registers A and B using the instruction `LDW`. The double-word load floating-point instruction `LDDW` on the C6711 can simultaneously load two 32-bit registers into side A and two 32-bit registers into side B.

   (b) The instruction

   ```
   ```

   stores the 32-bit word A1 into memory whose address is specified by A4 offset by 20 words (32 bits) or 80 bytes. The address register A4 is preincremented with offset, but it is not modified (two plus signs are used if A4 is to be modified).

3. **Branch/Move.** The following code segment illustrates branching and data transfer.

   ```
   Loop MVK .S1 x,A4 ;move 16 LSBs of x address → A4
   MVKH .S1 x,A4 ;move 16 MSBs of x address → A4
   .
   .
   .
   SUB .S1 A1,1,A1 ;decrement A1
   [A1] B .S2 Loop ;branch to Loop if A1 # 0
   NOP 5 ;five no-operation instructions
   STW .D1 A3,*A7 ;store A3 into (A7)
   ```

   The first instruction moves the lower 16 bits (LSBs) of address x into register A4. The second instruction moves the higher 16 bits (MSBs) of address x into
A4, which now contains the full 32-bit address of x. One must use the instructions MVK/MVKH in order to get a 32-bit constant into a register.

Register A1 is used as a loop counter. After it is decremented with the SUB instruction, it is tested for a conditional branch. Execution branches to the label or address loop if A1 is not zero. If A1 = 0, execution continues and data in register A3 are stored in memory whose address is specified (pointed) by A7.

### 3.9 ASSEMBLER DIRECTIVES

An assembler directive is a message for the assembler (not the compiler) and is not an instruction. It is resolved during the assembling process and does not occupy memory space as an instruction does. It does not produce executable code. Addresses of different sections can be specified with assembler directives. For example, the assembler directive `.sect "my_buffer"` defines a section of code or data named `my_buffer`. The directives `.text` and `.data` indicate a section for text and data, respectively. Other assembler directives, such as `.ref` and `.def`, are used for undefined and defined symbols, respectively. The assembler creates several sections indicated by directives such as `.text` for code and `.bss` for global and static variables.

Other commonly used assembler directives are:

1. `.short`: to initialize a 16-bit integer.
2. `.int`: to initialize a 32-bit integer (also `.word` or `.long`). The compiler treats a long data value as 40 bits, whereas the C6x assembler treats it as 32 bits.
3. `.float`: to initialize a 32-bit IEEE single-precision constant.
4. `.double`: to initialize a 64-bit IEEE double-precision constant.

Initialized values are specified by using the assembler directives `.byte`, `.short`, or `.int`. Uninitialized variables are specified using the directive `.usect`, which creates an uninitialized section (like the `.bss` section), whereas the directive `.sect` creates an initialized section. For example, `.usect "variable", 128,2` designates an uninitialized section named `variable`, the section size in bytes, and the data alignment in bytes, respectively.

### 3.10 LINEAR ASSEMBLY

An alternative to C, or assembly code, is linear assembly. An assembler optimizer (in lieu of a C compiler) is used in conjunction with a linear assembly-coded source program (with extension `.sa`) to create an assembly source program (with extension `.asm`), in much the same way that a C compiler optimizer is used in conjunction with
a C-coded source program. The resulting assembly-coded program produced by the assembler optimizer is typically more efficient than one resulting from the C compiler optimizer. The assembly-coded program resulting from either a C-coded source program or a linear-assembly source program must be assembled to produce an object code.

Linear assembly code programming provides a compromise between coding effort and coding efficiency. The assembler optimizer assigns which functional unit and register to use (optional to be specified by user), finds instructions that can execute in parallel, and performs software pipelining for optimization (discussed in Chapter 8). Two programming examples at the end of this chapter illustrate a C program calling a linear assembly function. Parallel instructions are not valid in a linear assembly program. Specifying the functional unit is optional in a linear assembly program as well as in an assembly program.

Over the last couple of years, the C compiler optimizer has become more and more efficient. Although C code is less efficient (speed performance) than assembly code, it typically involves less coding effort than assembly code, which can be hand-optimized to achieve a 100 percent efficiency but with much greater coding effort.

It may be interesting to note that the C6x assembly code syntax is not as complex as the C2x/C5x or the C3x family of digital signal processors. It is actually simpler to “program” the C6x in assembly. For example, the C3x instruction

\[
\text{DBNZD AR4, LOOP}
\]

decrements (due to the first D) a loop counter AR4, branches (B) conditionally (if AR4 is nonzero) to the address specified by LOOP, with delay (due to the second D). The branch instruction with delay effectively allows the branch instruction to execute in a single cycle (due to pipelining). Such multitask instructions are not available on the C6x (although recently introduced on the C64x processor). In fact, C6x types of instructions are “simpler.” For example, separate instructions are available for decrementing a counter (with a \text{SUB} instruction) and branching. The simpler types of instructions are more amenable for a more efficient C compiler.

However, although it is simpler to program in assembly code to perform a desired task, this does not imply or translate to an efficient assembly-coded program. It can be relatively difficult to hand-optimize a program to yield a totally efficient (and meaningful) assembly-coded program.

Linear assembly code is a cross between assembly and C. It uses the syntax of assembly code instructions such as \text{ADD}, \text{SUB}, and \text{MPY} but with operandsregisters as used in C. In some cases this provides a good compromise between C and assembly.

Linear assembler directives include

\[
\begin{align*}
\text{.cproc} \\
\text{.endproc}
\end{align*}
\]
to specify a C-callable procedure or section of code to be optimized by the assembler optimizer. Another directive, `.reg`, is to declare variables and use descriptive names for values that will be stored in registers. Programming examples with C calling an assembly function or C calling a linear assembly function are illustrated later in this chapter.

### 3.11 ASM STATEMENT WITHIN C

Assembly instructions and directives can be incorporated within a C program using the `asm` statement. The `asm` statement can provide access to hardware features that cannot be obtained using C code only. The syntax is

```c
asm ("assembly code");
```

The assembly line of code within the set of quotes has the same format as a valid assembly statement. Note that if the instruction has a label, the first character of the label must start after the first quote so that it is in column 1. The assembly statement should be valid since the compiler does not check it for syntax error but copies it directly into the compiled output file. If the assembly statement has a syntax error, the assembler would detect it.

Avoid using `asm` statements within a C program, especially within a linear assembly program. This is because the assembler optimizer could rearrange lines of code near the `asm` statements that may cause undesirable results.

### 3.12 C-CALLABLE ASSEMBLY FUNCTION

Two programming examples are included later in this chapter to illustrate a C program calling an assembly function. Register B3 is preserved and is used to contain the return address of the calling function.

An external declaration of an assembly function called within a C program using `extern` is optional. For example,

```c
extern int func();
```

is optional with the assembly function `func` returning an integer value.

### 3.13 TIMERS

Two 32-bit timers can be used to time and count events or to interrupt the CPU. A timer can direct an external ADC to start conversion or the DMA controller to start a data transfer. A timer includes a time period register, which specifies the timer’s frequency; a timer counter register, which contains the value of the incrementing counter; and a timer control register, which monitors the timer’s status.
3.14 INTERRUPTS

An interrupt can be issued internally or externally. An interrupt stops the current CPU process so that it can perform a required task initiated by the interrupt. The program flow is redirected to an interrupt service routine (ISR). The source of the interrupt can be an ADC, a timer, and so on. Upon an interrupt, the conditions of the current process must be saved so that they can be restored after the interrupt task is performed. On interrupt, registers are saved and processing continues to an ISR. Then the registers are restored.

There are 16 interrupt sources. They include two timer interrupts, four external interrupts, four McBSP interrupts, and four DMA interrupts. Twelve CPU interrupts are available. An interrupt selector is used to choose among the 12 interrupts.

3.14.1 Interrupt Control Registers

The interrupt control registers (Appendix B) follow.

1. CSR (control status register): contains the global interrupt enable (GIE) bit and other control/status bits
2. IER (interrupt enable register): enables/disables individual interrupts
3. IFR (interrupt flag register): displays status of interrupts
4. ISR (interrupt set register): sets pending interrupts
5. ICR (interrupt clear register): clears pending interrupts
6. ISTP (interrupt service table pointer): locates an ISR
7. IRP (interrupt return pointer)
8. NRP (nonmaskable interrupt return pointer)

Interrupts are prioritized, with Reset having the highest priority. The reset interrupt and nonmaskable interrupt (NMI) are external pins that have the first and second highest priority, respectively. The interrupt enable register (IER) is used to set a specific interrupt and can check if and which interrupt has occurred from the interrupt flag register (IFR).

NMI is nonmaskable, along with Reset. NMI can be masked (disabled) by clearing the NMIE bit within CSR. It is set to zero only upon reset or upon a nonmaskable interrupt. If NMIE is set to zero, all interrupts INT4 through INT15 are disabled. The interrupt registers are shown in Appendix B.

The reset signal is an active-low signal used to halt the CPU, and the NMI signal alerts the CPU to a potential hardware problem. Twelve CPU interrupts with lower priorities are available, corresponding to the maskable signals INT4 through INT15. The priorities of these interrupts are: INT4, INT5, . . . , INT15, with INT4 having the highest priority and INT15 the lowest priority. For a nonmaskable interrupt to occur, the nonmaskable interrupt enable (NMIE) bit must be 1 (active high). On reset (or
after a previously set NMI), the NMIE bit is cleared to zero so that a reset inter-
rupt may occur.

To process a maskable interrupt, the global interrupt enable (GIE) bit within the
control status register (CSR) and the NMIE bit within the interrupt enable regis-
ter (IER) are set to 1. GIE is set to 1 with bit 0 of CSR set to 1 and NMIE is set to
1 with bit 1 of IER set to 1. Note that CSR can be ANDed with −2 (using 2’s com-
plement, the LSB is zero while all other bits are 1’s) to set the GIE bit to zero and
disable maskable interrupts globally.

The interrupt enable (IE) bit corresponding to the desirable maskable interrupt
is also set to 1. When the interrupt occurs, the corresponding interrupt flag register
(IFR) bit is set to 1 to show the interrupt status. To process a maskable interrupt,
the following apply:

1. The GIE bit is set to 1.
2. The NMIE bit is set to 1.
3. The appropriate IE bit is set to 1.
4. The corresponding IFR bit is set to 1.

For an interrupt to occur, the CPU must not be executing a delay slot associated
with a branch instruction.

The interrupt service table (IST) shown in Table 3.5 is used when an interrupt
begins. Within each location is a fetch packet (FP) associated with each interrupt.
The table contains 16 FPs, each with eight instructions. The addresses on the right
side correspond to an offset associated with each specific interrupt. For example,
the FP for interrupt INT11 is at a base address plus an offset of 160h. Since each

<table>
<thead>
<tr>
<th>Interrupt</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESET</td>
<td>000h</td>
</tr>
<tr>
<td>NMI</td>
<td>020h</td>
</tr>
<tr>
<td>Reserved</td>
<td>040h</td>
</tr>
<tr>
<td>Reserved</td>
<td>060h</td>
</tr>
<tr>
<td>INT4</td>
<td>080h</td>
</tr>
<tr>
<td>INT5</td>
<td>0A0h</td>
</tr>
<tr>
<td>INT6</td>
<td>0C0h</td>
</tr>
<tr>
<td>INT7</td>
<td>0E0h</td>
</tr>
<tr>
<td>INT8</td>
<td>100h</td>
</tr>
<tr>
<td>INT9</td>
<td>120h</td>
</tr>
<tr>
<td>INT10</td>
<td>140h</td>
</tr>
<tr>
<td>INT11</td>
<td>160h</td>
</tr>
<tr>
<td>INT12</td>
<td>180h</td>
</tr>
<tr>
<td>INT13</td>
<td>1A0h</td>
</tr>
<tr>
<td>INT14</td>
<td>1C0h</td>
</tr>
<tr>
<td>INT15</td>
<td>1E0h</td>
</tr>
</tbody>
</table>

*Source:* Courtesy of Texas Instruments.
FP contains eight 32-bit instructions (256 bits) or 32 bytes, each offset address in the table is incremented by \(20\text{h}=32\).

The reset FP must be at address 0. However, the FPs associated with the other interrupts can be relocated. The relocatable address can be specified by writing this address to the interrupt service table base (ISTB) register of the interrupt service table pointer (ISTP) register, shown in Figure B.7. On reset, ISTB is zero. For relocating the vector table, the ISTP is used; the relocatable address is ISTB plus the offset.

Table 3.6 shows the interrupt selector values needed to choose a specific type of interrupt. The interrupt selector value 01000 is also for EDMA_INT, the enhanced DMA interrupt.

The software defined interrupts INT4–INT15 are associated with a physical interrupt signal using the interrupt multiplex registers IML and IMH. The desired interrupt select values in Table 3.5 are stored in the proper IML or IMH fields for INT4–INT15 [7]. See also the support file C6xdskinterrupt.h.

### 3.14.2 Selection of XINT0

In most previous examples, the McBSP0 transmit interrupt was chosen. In the communication file C6xdskinit.c, the function Config_Interrupt_Selector is called, which is within the interrupt header support file C6xinterrupts.h. The corresponding interrupt selector number \((01100) = 0\text{xC}\) is obtained from C6xinterrupts.h (this 5-bit selector value resides within bits 5 through 9 of the IMH register).

**TABLE 3.6 Selection of Interrupts Using Interrupt Selector**

<table>
<thead>
<tr>
<th>Interrupt Selector</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>DSPINT</td>
<td>Host port to DSP interrupt</td>
</tr>
<tr>
<td>00001</td>
<td>TINT0</td>
<td>Timer 0 interrupt</td>
</tr>
<tr>
<td>00010</td>
<td>TINT1</td>
<td>Timer 1 interrupt</td>
</tr>
<tr>
<td>00011</td>
<td>SD_INT</td>
<td>EMIF SDRAM timer interrupt</td>
</tr>
<tr>
<td>00100</td>
<td>EXT_INT4</td>
<td>External interrupt pin 4</td>
</tr>
<tr>
<td>00101</td>
<td>EXT_INT5</td>
<td>External interrupt pin 5</td>
</tr>
<tr>
<td>00110</td>
<td>EXT_INT6</td>
<td>External interrupt pin 6</td>
</tr>
<tr>
<td>00111</td>
<td>EXT_INT7</td>
<td>External interrupt pin 7</td>
</tr>
<tr>
<td>01000</td>
<td>DMA_INT0</td>
<td>DMA channel 0 interrupt</td>
</tr>
<tr>
<td>01001</td>
<td>DMA_INT1</td>
<td>DMA channel 1 interrupt</td>
</tr>
<tr>
<td>01010</td>
<td>DMA_INT2</td>
<td>DMA channel 2 interrupt</td>
</tr>
<tr>
<td>01011</td>
<td>DMA_INT3</td>
<td>DMA channel 3 interrupt</td>
</tr>
<tr>
<td>01100</td>
<td>XINT0</td>
<td>McBSP0 transmit interrupt</td>
</tr>
<tr>
<td>01101</td>
<td>RINT0</td>
<td>McBSP0 receive interrupt</td>
</tr>
<tr>
<td>01110</td>
<td>XINT1</td>
<td>McBSP1 transmit interrupt</td>
</tr>
<tr>
<td>01111</td>
<td>RINT1</td>
<td>McBSP1 receive interrupt</td>
</tr>
</tbody>
</table>

*Source: Courtesy of Texas Instruments.*
3.14.3 Interrupt Acknowledgment

The signals IACK and INUMx (INUM0 through INUM3) are pins on the C6x that acknowledge an interrupt has occurred and is being processed. The four INUMx signals indicate the number of the interrupt being processed. For example,

\[
\text{INUM3} = 1 \text{ (MSB), INUM2} = 0, \text{ INUM1} = 1, \text{ INUM0} = 1 \text{ (LSB)}
\]

corresponds to \((1011)_b = 11\), indicating that INT11 is being processed.

The IE11 bit is set to 1 to enable INT11. The interrupt flag register (IFR) can be read to verify that bit IF11 is set to 1 (INT11 enabled). Writing a 1 to a bit in the interrupt set register (ISR) causes the corresponding interrupt flag to be set in IFR; whereas a 0 to a bit in the interrupt clear register (ICR) causes the corresponding interrupt to be cleared.

All interrupts remain pending while the CPU has a pending branch instruction. Since a branch instruction has five delay slots, a loop smaller than six cycles is noninterruptible. Any pending interrupt will be processed as long as there are no pending branches to be completed. Additional information can be found in Ref. 6.

3.15 MULTICHANNEL BUFFERED SERIAL PORTS

Two multichannels buffered serial ports (McBSPs) are available. They provide an interface to inexpensive (industry standard) external peripherals. McBSPs have features such as full-duplex communication, independent clocking and framing for receiving and transmitting, and direct interface to AC97 and IIS compliant devices. It allows several data sizes between 8 and 32 bits. Clocking and framing associated with the McBSPs for input and output can be found in Ref. 7.

External data communication can occur while data are being moved internally. Figure 3.4 shows an internal block diagram of a McBSP. The data transmit (DX) and the data receive (DR) pins are used for data communication. Control information (clocking and frame synchronization) is through CLKX, CLKR, FSX, and FSR. The CPU or DMA controller reads data from the data receive register (DRR) and writes data to be transmitted to the data transmit register (DXR). The transmit shift register (XSR) shifts these data to DX. The receive shift register (RSR) copies the data received on DR to the receive buffer register (RBR). The data in RBR are then copied to DRR to be read by the CPU or the DMA controller.

Other registers—serial port control register (SPCR), receive/transmit control register (RCR/XCR), receive/transmit channel enable register (RCER/XCER), pin control register (PCR), and sample rate generator register (SRGR)—support further data communication [7].
3.16 DIRECT MEMORY ACCESS

Direct memory access (DMA) allows for the transfer of data to and from internal memory or external devices without intervention from the CPU [7]. Four DMA channels can be configured independently for data transfer. An additional (auxiliary) channel is available for DMA with the host port interface (HPI). DMA can access on-chip memory and the external memory interface (EMIF). Data of different sizes can be transferred: 8-bit bytes, 16-bit half-words, and 32-bit words.

A number of DMA registers are used to configure the DMA: address (source and destination), index, count reload, DMA global data, and control registers. The source and destination addresses can be from internal program memory, internal data memory, external memory interface, and internal peripheral bus. DMA transfers can be triggered by interrupts from internal peripherals as well as from external pins.

For each resource, each DMA channel can be programmed for priorities with the CPU. Between the four DMA channels, channel 0 has the highest priority.
and channel 3 has the lowest priority. Each DMA channel can be made to start initiating block transfer of data independently. A block can contain a number of frames. Within each frame can be many elements. Each element is a single data value. The DMA count reload register contains the value to specify the frame count (16 MSBs) and the element count (16 LSBs). An enhanced DMA (EDMA) is also available with 16 independently programmable channels.

3.17 MEMORY CONSIDERATIONS

3.17.1 Data Allocation

Blocks of code and data can be allocated in memory within sections specified in the linker command file. These sections can be either initialized or uninitialized. Initialized or uninitialized sections, except .text, cannot be allocated into internal program memory. The initialized sections are:

1. .cinit: for global and static variables
2. .const: for global and static constant variables
3. .switch: contains jump tables for large switch statements
4. .text: for executable code and constants

The uninitialized sections are:

1. .bss: for global and static variables
2. .far: for global and static variables declared far
3. .stack: allocates memory for the system stack
4. .sysmem: reserves space for dynamic memory allocation used by the malloc, calloc, and realloc functions

The linker can be used to place sections, such as, text in fast internal memory for most efficient operation.

3.17.2 Data Alignment

The C6x always accesses aligned data which allows it to address bytes, half-words, and words (32 bits). The data format consists of four byte boundaries, two half-word boundaries, and one word boundary. For example, to assign a 32-bit load with LDW, the address must be aligned with a word boundary so that the lower 2 bits of the address are zero. Otherwise, incorrect data can be loaded. A double-word (64 bits) also can be accessed. Both .S1 and .S2 can be used to execute the double-word instruction LDDW to load two 64-bit double words, for a total of 128 bits per cycle.
3.17.3 Pragma Directives

The pragma directives tell the compiler to consider certain functions. Pragmas include `DATA_ALIGN`, `DATA_SECTION`, and so on. The `DATA_ALIGN` pragma has the syntax

```c
#pragma DATA_ALIGN (symbol, constant);
```

which aligns `symbol` to a boundary. The constant is a power of 2. This pragma directive is used later in conjunction with FFT examples to align data in memory.

The `DATA_SECTION` pragma has the following syntax:

```c
#pragma DATA_SECTION (symbol, "my_section");
```

which allocates space for `symbol` in the section named `my_section`.

Another useful pragma directive,

```c
#pragma MUST_ITERATE (20, 20)
```

tells the compiler that the loop following will execute 20 times (minimum and maximum of 20 times).

3.17.4 Memory Models

The compiler generates a small memory model code by default. Every data object is handled as if declared `near` unless it is specifically declared `far`. If the `DATA_SECTION` pragma is used, the object is specified as a `far` variable.

How run-time support functions are called can be controlled by the option `–mr0` with the run-time support data and calls `near`, or by the option `–mr1` with the run-time support data and calls `far`. Using the `far` method to call functions does not imply that those functions must reside in off-chip memory.

Large-memory models can be generated with the linker options `–mlx` (x = 0 to 4). If no level is specified, data and functions default to `near`. These models can be used if calling a function that is more than 1 M word away.

3.18 FIXED- AND FLOATING-POINT FORMAT

Some fixed-point considerations are reviewed in Appendix C.

3.18.1 Data Types

Some data types are:

1. `short`: of size 16 bits represented as 2’s complement with a range from $-2^{15}$ to $(2^{15} - 1)$
2. *int* or *signed int*: of size 32 bits represented as 2’s complement with a range from \(-2^{31}\) to \((2^{31} - 1)\)

3. *float*: of size 32 bits represented as IEEE 32-bit with a range from \(2^{-126} = 1.175494 \times 10^{-38}\) to \(2^{128} = 3.40282346 \times 10^{38}\)

4. *double*: of size 64 bits represented as IEEE 64-bit with a range from \(2^{-1022} = 2.22507385 \times 10^{-308}\) to \(2^{1024} = 1.79769313 \times 10^{308}\)

Data types such as *short* for fixed-point multiplication can be more efficient (fewer cycles) than using *int*. Use of *const* can also increase code performance.

### 3.18.2 Floating-Point Format

With a much wider dynamic range in a floating-point processor, scaling is not an issue. A floating-point number can be represented using single precision (SP) with 32 bits or double precision (DP) with 64 bits, as shown in Figure 3.5. In single-precision format, bit 31 represents the sign bit, bits 23 through 30 represent the exponent bits, and bits 0 through 22 represent the fractional bits, as shown in Figure 3.5a. Numbers as small as \(10^{-38}\) and as large as \(10^{+38}\) can be represented. In double-precision format, more exponent and fractional bits are available, as shown in Figure 3.5b. Since 64 bits are represented, a pair of registers is used. Bits 0 through 31 of the first register pair represent the fractional bits. Bits 0 through 19 of the second register pair also represent the fractional bits, with bits 20 through 30 representing the exponent bits, and bit 31 the sign bit. As a result, numbers as small as \(10^{-308}\) and as large as \(10^{+308}\) can be represented.

Instructions ending in either *SP* or *DP* represent single and double precision, respectively. Some of the floating-point instructions (available on the C67x floating-point processor) have more latencies than do fixed-point instructions. For example, the fixed-point multiplication *MPY* requires one delay or *NOP*, whereas the single-precision *MPYSP* requires three delays, and the double-precision instruction *MPYDP* requires nine delays.

The single-precision floating-point instructions *ADDSP* and *MPYSP* have three delay slots and take four cycles to complete execution. The double-precision instruc-

![Figure 3.5](image-url)
tions ADDDP and MPYDP have six and nine delay slots, respectively. However, the floating-point double-word load instruction LDDW (with four delay slots as with the fixed-point LDW) can load 64 bits. Two LDDW instructions can execute in parallel through both units S1 and S2 to load a total of 128 bits per cycle.

A single-precision floating-point value can be loaded into a single register, whereas a double-precision floating-point value is a 64-bit value that can be loaded into a register pair such as A1:A0, A3:A2, . . . , B1:B0, B3:B2, . . . The least significant 32 bits are loaded into the even register pair, and the most significant 32 bits are loaded into the odd register pair.

One may need to weigh the pros and cons of dynamic range and accuracy with possible degradation in speed when using floating-point types of instructions.

3.18.3 Division

The floating-point C6711 processor has a single-precision reciprocal instruction RCPSP. A division operation can be performed by taking the reciprocal of the denominator and multiplying the result by the numerator [6]. There are no fixed-point instructions for division. Code is available to perform a division operation by using the fixed-point processor to implement a Newton–Raphson equation.

3.19 CODE IMPROVEMENT

Several code optimization schemes are discussed in Chapter 8 using both fixed- and floating-point implementations and ASM code.

3.19.1 Intrinsics

C code can be optimized further by using many of the intrinsics available from the run-time library support file. Intrinsic functions are similar to run-time support library functions. Intrinsics are available to multiply, to add, to find the reciprocal of a square root, and so on. For example, in lieu of using the asterisk operator to multiply, the intrinsic _mpy can be used. Intrinsics are special functions that map directly to inline C6x instructions. For example,

```c
int _mpy()
```

is equivalent to the assembly instruction MPY, to multiply the 16 LSBs of two numbers. The intrinsic function

```c
int _mpyh()
```

is equivalent to the assembly instruction MPYH to multiply the 16 MSBs of two numbers.
3.19.2 Trip Directive for Loop Count

The linear assembly directive \texttt{.trip} is used to specify the number of times a loop iterates. If the exact number is known and used, the linear assembler optimizer can produce pipelined code (discussed in Chapter 8) and redundant loops are not generated. This can improve both code size and execution time. A \texttt{.trip} count specification, even if it is not the exact value, may improve performance: for example, when the actual number of iterations is a multiple of the specified value. The intrinsic function \texttt{_nassert()} can be used in a C program in lieu of \texttt{.trip}. Example 3.1 illustrates the use of \texttt{_nassert()} in the dot product example.

3.19.3 Cross-Paths

Data and address cross-path instructions are used to increase code efficiency. The instruction

\begin{verbatim}
MPY .M1x A2,B2,A4
\end{verbatim}

illustrates a data cross-path that multiplies the two sources A2 and B2 from two different sides, A and B, with the result in A4. If the result is in the B register file, a 2x cross-path is used with the instruction

\begin{verbatim}
MPY .M2x A2,B2,B4
\end{verbatim}

with the result in B4. The instruction

\begin{verbatim}
LDW .D1T2 *A2,B2
\end{verbatim}

illustrates an address cross-path. It loads the content in register A2 (from a register file A) into register B2 (register file B). Only two cross-paths are available on the C6x, so no more than two instructions using cross-paths are allowed within a cycle.

3.19.4 Software Pipelining

Software pipelining uses available resources to obtain efficient pipelining code. The aim is to use all eight functional units within one cycle. However, substantial coding effort is required using the software pipelining technique. There are three stages to a pipelined code:

1. Prolog
2. Loop kernel (or loop cycle)
3. Epilog
The first stage, prolog, contains instructions to build the second-stage loop cycle, and the epilog stage (last stage) contains instructions to finish all loop iterations. Software pipelining is used by the compiler when optimization option level \(-o2\) or \(-o3\) is invoked. The most efficient software pipelined code has loop trip counters that count down: for example,

```plaintext
for (i = N; i != 0; i--) 
```

A dot product example with word-wide hand-coded pipelined code results in \((N/2) + 8\) cycles to obtain the sum of two arrays, with \(N\) numbers in each array. This translates to 108 cycles to find the sum of products of 200 numbers, as illustrated in Chapter 8. This efficiency is obtained using instructions such as \texttt{LDW} to load a 32-bit word, and multiplying the lower and higher 16-bit numbers separately with the two instructions \texttt{mpy} and \texttt{mpyh}, respectively.

Removing the epilog section can also reduce the code size. The available options \(-msn\) \((n = 0, 1, 2)\) directs the compiler to favor code size reduction over performance. Producing a hand-coded software pipelined code can be obtained by first drawing a dependency graph and setting up a scheduling table [8]. In Chapter 8 we discuss software pipelining in conjunction with code efficiency.

### 3.20 CONSTRAINTS

#### 3.20.1 Memory Constraints

Internal memory is arranged through various banks of memory so that loads and stores can occur simultaneously. Since each bank of memory is single-ported, only one access to each bank is performed per cycle. Two memory accesses per cycle can be performed if they do not access the same bank of memory. If multiple accesses are performed to the same bank of memory (within the same space), the pipeline will stall. This causes additional cycles for execution to complete.

#### 3.20.2 Cross-Paths Constraints

Since there is one cross-path in each side of the two data paths, there can be at most two instructions per cycle using cross-paths. The following code segment is valid since both available cross-paths are utilized:

```plaintext
ADD .L1x A1,B1,A0
|| MPY .M2x A2,B2,B3
```

whereas the following is not valid since one cross-path is used for both instructions:
ADD .L1x A1,B1,A0
|| MPY .M1x A2,B2,A3

The \( \times \) associated with the functional unit designates a cross-path.

### 3.20.3 Load/Store Constraints

The address register to be used must be on the same side as the \( .D \) unit. The following code segment is valid:

\[
\text{LDW} \ .D1 \ *A1,A2 \\
|| \text{LDW} \ .D2 \ *B1,B2
\]

whereas the following is not valid:

\[
\text{LDW} \ .D1 \ *A1,A2 \\
|| \text{LDW} \ .D2 \ *A3,B2
\]

Furthermore, loading and storing cannot be from the same register file. A load (or store) using one register file in parallel with another load (or store) must use a different register file. For example, the following code segment is valid:

\[
\text{LDW} \ .D1 \ *A0,B1 \\
|| \text{STW} \ .D2 \ A1,*B2
\]

The following is also valid:

\[
\text{LDW} \ .D1 \ *A0,B1 \\
|| \text{LDW} \ .D2 \ *B2,A1
\]

However, the following is not valid:

\[
\text{LDW} \ .D1 \ *A0,A1 \\
|| \text{STW} \ .D2 \ A2,*B2
\]

### 3.20.4 Pipelining Effects with More Than One EP within an FP

Table 3.3 shows a previous pipeline operation representing eight instructions in parallel within one fetch packet (FP). Table 3.7 shows the pipeline operation when there are more than one execute packet (EP) within an FP.

Consider the operation of six fetch packets (FP1 through FP6) through the pipeline. FP1 contains three execute packets, and FP2, FP3, \ldots, FP6 each contains
one execute packet. In cycles 2 through 5, FP2 through FP5, each starts its program fetch phase. When the CPU detects that FP1 contains more than one EP, it forces the pipeline to stall so that EP2 and EP3, within FP1, can each start its dispatching phase in cycles 6 and 7, respectively. Each instruction within an FP has a “p” bit to specify whether that instruction is in parallel with a subsequent instruction (if a 1, as shown in Figure 3.3).

During clock cycles 1 through 4, a program fetch phase occurs. The three EPs within the same fetch packet cause a stall in the pipeline. This allows the DP phase to start at cycle 6 (not at cycle 5) for EP2 and at cycle 7 for EP3. The subsequent fetch packet (FP2) with only one EP (with all eight instructions in parallel) is stalled so that each of the three EPs in the previous FP (FP1) can go through the DP phase. As a result, while the fetch phase for FP2 starts at cycle 2, its DP phase does not start until cycle 8. The third fetch packet (FP3), also with only one EP, starts its fetch stage at cycle 3, but its DP phase does not start until cycle 9, due to the pipeline stall.

The pipeline then stalls in cycles 6 and 7, as indicated with an “X”. Once EP3 (within FP1) continues onto its decoding phase in cycle 8, the pipeline is released. FP2 can now continue to its dispatching phase in cycle 8. Since FP3 through FP6 also were stalled, each can now resume its program fetch phase in cycle 8.

Hence, with the three EPs within one FP, the pipeline stalls for two cycles. Table 3.7 illustrates the stalling pipeline effects. A pipeline stall would also take place in the event that the first FP had four EPs, each with two parallel instructions.

### 3.21 TMS320C64x PROCESSOR

Another member of the C6000 family of processors is the C64x, which can operate at a much higher clock rate, reaching the gigahertz range. Operating at 750 MHz with eight instructions per cycle, this translates to 6000 million instructions per second (MIPS).
The C64x is based on the architecture VELOCITI.2, which is an extension of VELOCITI [8]. Some of its features include a larger memory and twice as many registers, for a total of sixty-four 32-bit registers. The extra registers allow for packed data types to support four 8-bit or two 16-bit operations associated with one 32-bit register; hence increasing parallelism. For example, the instruction MPYU4 performs four 8-bit multiplications within a single instruction cycle time. Several special-purpose instructions have also been added to handle many operations encountered in wireless and digital imaging applications, where 8-bit data processing is common. In addition, the .M unit (for multiply operations) can also handle shift and rotate operations. Similarly, the .D unit (for data manipulation) can also handle logical operations.

The C64x is a fixed-point processor. Existing instructions are available to more units. Double-word load (LDDW) and store (STDW) instructions can access 64 bits of data, with up to two double-word load or store instructions per cycle (read or write 128 bits per cycle).

A few instructions have been added for the C64x processor. For example, the instruction

BDEC LOOP, B0

decrements a counter B0 and performs a conditional (based on B0) branch to LOOP. The branch decision is before the decrement; with the branch decision based on a negative number (not on whether the number is zero). This multitask instruction resembles the syntax used in the C3x and C4x family of processors.

Furthermore, with the intrinsic C function _dotp2, it can perform two $16 \times 16$ multiplies and adds the products together to further reduce the number of cycles. This intrinsic function in C has the corresponding assembly function DOTP2. With two multiplier units, four $16 \times 16$ multiplies per cycle can be performed, double the rate of the C62x or C67x. At 750 MHz, this corresponds to 3 billion multiply operations per second; or 6 billion $8 \times 8$ multiplies per second.

### 3.22 Programming Examples Using C, Assembly, and Linear Assembly

Six programming examples are discussed in this section. The first example illustrates use of the intrinsic function _nassert to increase the efficiency of the dot product example. The other five examples illustrate both assembly code and linear assembly code implementation: a C program calling an assembly function, a C program calling a linear assembly function, and an assembly-coded program calling an assembly-coded function. The focus here is on illustrating the syntax of both assembly and linear assembly code, not necessarily to produce optimized code. We discuss further optimization techniques in Chapter 8 in conjunction with code efficiency and software pipelining.
**Example 3.1: Efficient Dot Product (dotpopt)**

This example uses the intrinsic function `__nassert` in the dot product example introduced in Chapter 1. Figure 3.6 shows a listing of the program `dotpopt.c`, which calls the C function `dotpfunc.c` listed in Figure 3.7. This function produces more efficient code, with `__nassert` used for the alignment of the incoming pointers as constant pointers. This provides additional information to the compiler about the loop.

Verify that using compiler options `-g` and `-o3`, the number of cycles associated with profiling the function `dotpfunc.c` is reduced from 100 (without the intrin-

```c
//dotpopt.c Optimized dot product of two arrays
#include <stdio.h>
#include "dotp4.h"
#define count 4
short x[count] = {x_array}; //declare 1st array
short y[count] = {y_array}; //declare 2nd array
volatile int result = 0; //result

main()
{
    result = dotpfunc(x, y, count); //call optimized function
    printf("result = %d decimal \n", result); //print result
}

FIGURE 3.6. Dot product program calling function with __nassert intrinsic (dotpopt.c).
```

```c
//dotpfunc.c Optimized dot product function
int dotpfunc(const short *a, const short *b, int ncount)
{
    int sum = 0;
    int i;

    __nassert((int)(a)%4 == 0);
    __nassert((int)(b)%4 == 0);
    __nassert((int)(ncount)%4 == 0);

    for ( i = 0; i < ncount; i++)
    {
        sum += (a[i] * b[i]); //sum of products
    }
    return (sum); //return sum as result
}

FIGURE 3.7. C-called function for a dot product using __nassert (dotpfunc.c).
```
sics functions) to 71 (with intrinsics). Using the options \texttt{-g}, \texttt{-pm}, and \texttt{-o3}, the number of cycles is further reduced to 30. The \texttt{-pm} option uses program level optimization, with the source files compiled into one intermediate file. The results with this option can be compared to the results obtained with the function \texttt{dotp} in Example 1.3.

In Chapter 8 we use optimization techniques associated with the dot product example, using two arrays each with \( N \) numbers. We show that the number of cycles can be reduced to \( 7 + (N/2) + 1 \) with a fixed-point implementation, or 108 cycles using 200 numbers in each array. For a floating-point implementation, we obtain 124 cycles (see Table 8.4).

**Example 3.2: Sum of \( n + (n - 1) + (n - 2) + \ldots + 1 \) Using C Calling Assembly Function (\texttt{sum})**

This example illustrates a C program calling an assembly function. The C source program \texttt{sum.c} (Figure 3.8) calls the assembly-coded function \texttt{sumfunc.asm}

```
//Sum.c Finds \( n+(n-1)+\ldots+1 \). Calls assembly function sumfunc.asm

#include <stdio.h>

main()
{
    short n=6;  //set value
    short result; //result from asm function

    result = sumfunc(n);  //call assembly function sumfunc
    printf("sum = %d", result); //print result from asm function
}
```

**FIGURE 3.8.** C program that calls an ASM function to find \( n + (n - 1) + (n - 2) + \ldots + 1 \) (\texttt{sum.c}).

```
;Sumfunc.asm Assembly function to find \( n+(n-1)+\ldots+1 \)

.def _sumfunc ;function called from C
_sumfunc: MV .L1 A4,A1  ;setup n as loop counter
    SUB .S1 A1,1,A1   ;decrement n

LOOP:  ADD .L1 A4,A1,A4  ;accumulate in A4
    SUB .S1 A1,1,A1   ;decrement loop counter
[A1]  B .S2 LOOP     ;branch to LOOP if A1#0
    NOP 5           ;five NOPs for delay slots
    B .S2 B3       ;return to calling routine
    NOP 5           ;five NOPs for delay slots
.end
```

**FIGURE 3.9.** ASM function called from C in the project \texttt{sum (sumfunc.asm)}.
(Figure 3.9). It implements the sum of \( n + (n - 1) + (n - 2) + \ldots + 1 \). The value of \( n \) is set in the main C program. It is passed through register A4 (by convention). For example, the address of more than one value can be passed to the assembly function through A4, B4, A6, \ldots. The resulting sum from the assembly function is returned to \textit{result} in the C program, which then prints this resulting sum.

The assembly function’s name is preceded by an underscore (by convention). The value \( n \) in register A4 in the \texttt{asm} function is moved to register A1 to set A1 as a loop counter. A1 is then decremented. A loop section of code starts with the label or address \texttt{LOOP} and ends with the first branch statement B. The first addition adds \( n + (n - 1) \) with the result in A4. A1 is again decremented to \( (n - 2) \). The branch statement is conditional based on register A1 (only A1, A2, B0, B1, and B2 can be used as conditional registers), and since A1 is not zero, branching takes place and execution returns to the instruction at the address \texttt{LOOP}, where \( A4 = n + (n - 1) \) is added to \( A1 = (n - 2) \). This process continues until register A1 = 0.

The second branch instruction is to the returning address B3 (by convention) of the C calling program. The resulting sum is contained or accumulated in A4, which is passed to \textit{result} in the C program. The five \texttt{NOP} (no operation) are to account for the five delay slots associated with a branch instruction.

The functional units .S and .L selected are shown but are not required in the program. They can be useful for debugging and analyzing which of the functional units are used in order to improve on the efficiency of the program. Similarly, the two colons after the label \texttt{LOOP} and the function name are not required.

Build and run this project as \texttt{sum}. With a value of \( n \) set to 6 in the C program, verify that \texttt{sum} and its value of 21 are printed.

\section*{Example 3.3: Factorial of a Number Using C Program Calling Assembly Function (\texttt{factorial})}

This example finds the factorial of a number \( n \leq 7 \) with \( n! = n(n - 1)(n - 2) \ldots (1) \). It further illustrates the syntax of assembly code. It is very similar to Example 3.2. The value of \( n \) is set in the C source program \texttt{factorial.c}, shown in Figure 3.10, which calls the assembly function \texttt{factfunc.asm}, shown in Figure 3.11. It is instructive to read the comments.

Register A1 is again set as a loop counter. Within the loop section of code starting with at the address \texttt{LOOP}, the first multiply is \( n(n - 1) \) and accumulates in register A4. The initial value of \( n \) is passed to the \texttt{asm} function through A4. The MPY instruction has one delay slot, hence the \texttt{NOP} following it. Processing continues within the loop section of code until A1 = 0. Note that the functional units are not specified in this program. The resulting factorial is returned to the calling C program through A4.

Build and run this project as \texttt{factorial}. Verify that \texttt{factorial} and its value of 5040 (7!) are printed. Note that the maximum value of \( n \) is 7, since 8! is greater than \( 2^{15} \).
Example 3.4: Dot Product Using Assembly Program Calling Assembly Function (dotp4a)

This example takes the sum of products of two arrays, each array with four numbers. See also Example 1.3, which implements it using only C code, and Examples 3.2 and 3.3, which introduced the syntax of assembly code. Figure 3.12 shows a listing of the assembly program dotp4a_init.asm, which initializes the two arrays of numbers and calls the assembly function dotp4afunc.asm (Figure 3.13) which takes the sum of products of the two arrays. It also sets a return address through register B3 and the result address to A0. The addresses of the two arrays and the size of the array are passed to the function dotp4afunc.asm through registers A4, A6, and B4, respectively. The result from the called function is “sent back” through A4. The resulting sum of product is stored in memory whose address is
Programming Examples Using C, Assembly, and Linear Assembly

;Dotp4a_init.asm  ASM program to init variables. Calls dotp4afunc.asm

\text{.def} init ;starting address
\text{.ref} dotp4afunc ;called ASM function
\text{.text} ;section for code

x_addr .short 1,2,3,4 ;numbers in x array
y_addr .short 0,2,4,6 ;numbers in y array
result_addr .short 0 ;initialize sum of products

init MVK result_addr,A4 ;A4 = lower 16-bit addr -->A4
MVKH result_addr,A4 ;A4 = higher 16-bit addr-->A4
MVK 0,A3 ;A3 = 0
STH A3,\ast A4 ;init result to 0
MVK x_addr,A4 ;A4 = 16 MSBs address of x
MVK y_addr,B4 ;B4 = 16 LSBs address of y
MVKH y_addr,B4 ;B4 = 16 MSBs address of y
MVK 4,A6 ;A6 = size of array
B dotp4afunc ;branch to function dotp4afunc
MVK ret_addr,b3 ;B3 = return addr from dotp4a
MVKH ret_addr,b3 ;B3 = return addr from dotp4a
NOP 3 ;3 more delay slots(branch)

ret_addr MVK result_addr,A0 ;A0 = 16 LSBs result_addr
MVKH result_addr,A0 ;A0 = 16 MSBs result_addr
STW A4,\ast A0 ;store result

wait B wait ;wait here
NOP 5 ;delay slots for branch

\text{FIGURE 3.12.} ASM program calling ASM function to find the sum of products (dotp4a_init.asm).

result_addr. The instruction \text{STW} stores the resulting sum of products (in A4) in memory pointed by A0. Register A0 serves as a pointer with the address result_addr.

The starting address of the calling ASM program is defined as \text{init}. The vector file \text{vectors_dotp4a.asm} (Figure 3.14) specifies a branch to that entry address. The called ASM function \text{dotp4afunc.asm} calculates the sum of products. The loop count value was moved to A1 since A6 cannot be used as a conditional register (only A1, A2, B0, B1, B2 can be used). The two \text{LDH} instructions load (half-word of 16 bits) the addresses of the two arrays starting at x_addr and y_addr into registers A2 and B2, respectively. For example, the instruction

\text{LDH} *B4++,B2

loads the content in memory (the first value in the second array starting at y_address) pointed by B4 (the address of the second array) into B2. Then
register B4, used as a pointer, is postincremented to the next-higher address in memory that contains the second value in the second array. Register A7 is used to accumulate and move the sum of products to register A4, since the result is passed to the calling function through A4.

Support files for this project include (no library file is necessary):

```

;Dotp4afunc.asm Multiply two arrays. Called from dotp4a_init.asm
;A4=x address,B4=y address,A6=count(size of array),B3=return address

.def    dotp4afunc ;dot product function
.text    ;text section
dotp4afunc  MV   A6,A1   ;move loop count -->A1
            ZERO  A7   ;init A7 for accumulation

loop     LDH   *A4++,A2   ;A2=(x. A4 as address pointer
           LDH   *B4++,B2   ;B2=(y). B4 as address pointer
           NOP    4       ;4 delay slots for LDH
           MPY    A2,B2,A3 ;A3 = x * y
           NOP    1       ;1 delay slot for MPY
           ADD    A3,A7,A7 ;sum of products in A7
           SUB    A1,1,A1 ;decrement loop counter
           [A1]  B     loop    ;branch back to loop till A1=0
           NOP    5       ;5 delay slots for branch

           MV    A7,A4   ;A4=result A4=return register
           B     B3     ;return from func to addr in B3
           NOP    5       ;5 delay slots for branch

FIGURE 3.13. ASM function called from an ASM program to find the sum of products (dotp4afunc.asm).

;vectors_dotp4a.asm Vector file for dotp4a project

    .ref    init       ;starting addr in init file
    .sect   "vectors" ;in section vectors
rst:    mvkl  .s2 init,b0 ;init addr 16 LSB -->B0
        mvkh  .s2 init,b0 ;init addr 16 MSB -->B0
        b     b0       ;branch to addr init
        nop
        nop
        nop
        nop

FIGURE 3.14. Vector file that specifies the entry address in the calling ASM program for the sum of products (vectors_dotp4a.asm).
```
1. dotp4a_init.asm
2. dotp4afunc.asm
3. vectors_dotp4a.asm

Build and run this project as dotp4a. Modify the Linker Option (Project → Options) to select “No Autoinitialization.” Otherwise, the warning “entry point symbol _c_int00 undefined” is displayed when this project is built (it can be ignored). This is because the “conventional” entry point is not used in this project with no main function in C.

Set a breakpoint at the first branch instruction in the program dotp4a_init.asm:

B dotp4afunc

Select View → Memory and set address to result_addr and use 16-bit signed integer. Right-click on the memory window and deselect “Float in Main Window.” This allows you to have a better display of the Memory window while viewing the source file dotp4a_init.asm.

Select Run. Execution stops at the set breakpoint. The content in memory at the address result_addr is zero (the called function dotp4afunc.asm is not yet executed). Run again, then halt (since execution is within the infinite wait loop instruction):

wait B wait ; wait here

Verify that the resulting sum of products is now 40. Note that A0 contains the result address (result_addr). View → CPU Registers → Core Registers and verify this address (in hex). Figure 3.15 shows a CCS display of this project. Note from the disassembly file that execution was halted at the infinite wait loop.

Example 3.5: Dot Product Using C Function Calling Linear Assembly Function (dotp4clasm)

Figure 3.16 shows a listing of the C program dotp4clasm.c, which calls the linear assembly function dotp4clasmfunc.sa (Figure 3.17). Example 1.3 introduced the dot product implementation using C code only. The previous three examples introduced the syntax of assembly-coded programs.

The section of code invoked by the linear assembler optimizer starts and ends with the linear assembler directives .cproc and .endproc, respectively. The name of the linear assembly function called is preceded by an underscore since the calling function is in C. The directive .ref (or .def) references (defines) the function.

Functional units are optional as in an assembly-coded program. Registers a, b, prod and sum are defined by the linear assembler directive .reg. The addresses
FIGURE 3.15. CCS windows for the sum of products in the project dotp4a.

//Dotp4clasm.c Multiplies two arrays using C calling linear ASM func

short dotp4clasmfunc(short *a, short *b, short ncount); //prototype
#include <stdio.h> //for printing statement
#include "dotp4.h" //arrays of data values
#define count 4 //number of data values
short x[count] = {x_array}; //declare 1st array
short y[count] = {y_array}; //declare 2nd array
volatile int result = 0; //result

main()
{
    result = dotp4clasmfunc(x,y,count); //call linear ASM func
    printf("result = %d decimal \n", result); //print result
}

FIGURE 3.16. C program calling a linear ASM function to find the sum of products (dotp4clasm.c).
of the two arrays $x$ and $y$ and the size of the array (count) are passed to the linear assembly function through the registers $ap$, $bp$, and $count$. Both $ap$ and $bp$ are registers used as pointers, as in C code. The instruction field is seen to be as in an assembly-coded program and the subsequent field uses a syntax as in C programming. For example, the instruction

```assembly
loop:   ldh  *ap++,a
        ldh  *bp++,b
        mpy  a,b,prod
        add  prod,sum,sum
        sub  count,1,count
        [count] b loop
```

Build and run this project as `dotp4clasm`. Verify that the following is printed:

```
result = 40
```

You may wish to profile the linear assembly code function and compare its execution time with the C-coded version in Example 1.3.

**Example 3.6: Factorial Using C Calling a Linear Assembly Function (`factclasm`)**

Figure 3.18 shows a listing of the C program `factclasm.c`, which calls the linear ASM function `factclasmfunc.sa` (Figure 3.19) to calculate the factorial of a number less than 8. See also Example 3.3, which finds the factorial of a number using a C program that calls an ASM function. Example 3.5 illustrates a C
program calling a linear ASM function to find the sum of products and is instructive for this project. Examples 3.3 and 3.5 cover the essential background for this project.

Support files for this project include factclasm.c, factclasmfunc.sa, vectors, rts6701.lib, and C6xdsk.cmd. Build and run this project as factclasm. Verify that the result of 7! is printed, or factorial = 5040.

REFERENCES


6. TMS320C6000 CPU and Instruction Set, SPRU189F, Texas Instruments, Dallas, TX, 2000.

7. TMS320C6000 Peripherals, SPRU190D, Texas Instruments, Dallas, TX, 2001.


