

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  $\pm 1.5$  V. The steps represented by the ADC are:  $\text{input range}/2^8 = 3/256 = 11.72 \text{ mV}$ . This produces errors that can be up to  $\pm(11.72 \text{ mV})/2 = \pm 5.86 \text{ mV}$ . Only a best estimate can be used by the ADC to represent input values that are not multiples of 11.72 mV. 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 (or larger, currently very common), can reduce the quantization error, yielding a higher resolution. The more bits an ADC has, the better it can represent an input signal.

The TMS320C30 floating-point processor was introduced in the late 1980s. The C31, the 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 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 more recent addition to the family of the C6x fixed-point processors is the C64x. The C64x is introduced in Appendix H.

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 DSP 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 DSP. 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 (e.g., C67x) processors. Other DSP's 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 processor more than on the programmer, as in the VLIW architecture. It does not necessarily execute the same group of instructions, and as a result, it is difficult to time. Thus, it is rarely used in DSP.

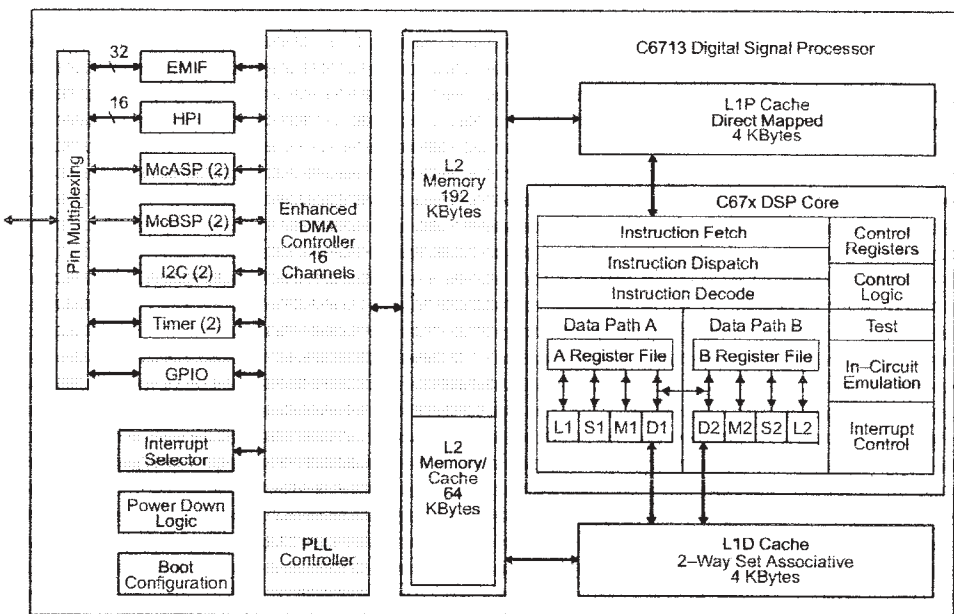
### 3.2 TMS320C6x ARCHITECTURE

The TMS320C6713 onboard the DSK is a floating-point processor based on the VLIW architecture [6–10]. Internal memory includes a two-level cache architecture with 4kB of level 1 program cache (L1P), 4kB of level 1 data cache (L1D), and 256kB of level 2 memory shared between program and data space. 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 DRAM, with SRAM being faster but more expensive than DRAM.

On-chip peripherals include two McBSPs, two timers, a host port interface (HPI), and a 32-bit EMIF. It requires 3.3 V for I/O and 1.26 V 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 C6713 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 independent buses. Since internal memory is organized into memory banks, two loads or two stores of 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 separate program and data memory spaces, with two 32-bit internal ports (two 64-bit ports with the C64x) to access internal memory.

The C6713 on the DSK includes 264kB of internal memory, which starts at  $0 \times 00000000$ , and 16MB of external SDRAM, mapped through CE0 starting at  $0 \times 80000000$ . The DSK also includes 512kB of Flash memory (256kB readily available to the user), mapped through CE1 starting at  $0 \times 90000000$ . Figure 3.2 shows



**FIGURE 3.1.** Functional block diagram of TMS320C6713 (Courtesy of Texas Instruments).

the L2 internal memory configuration, included with CCS [7]. Table 3.1 shows the memory map, also included with CCS [7]. A schematic diagram of the DSK is included with CCS (*6713dsk\_schem.pdf*).

With the DSK operating at 225 MHz, one can ideally achieve two multiplies and accumulates per cycle, for a total of 450 million multiplies and accumulates (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 1350 million floating-point operations per second (MFLOPS). Operating at 225 MHz, this translates into 1800 million instructions per second (MIPS) with a 4.44-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

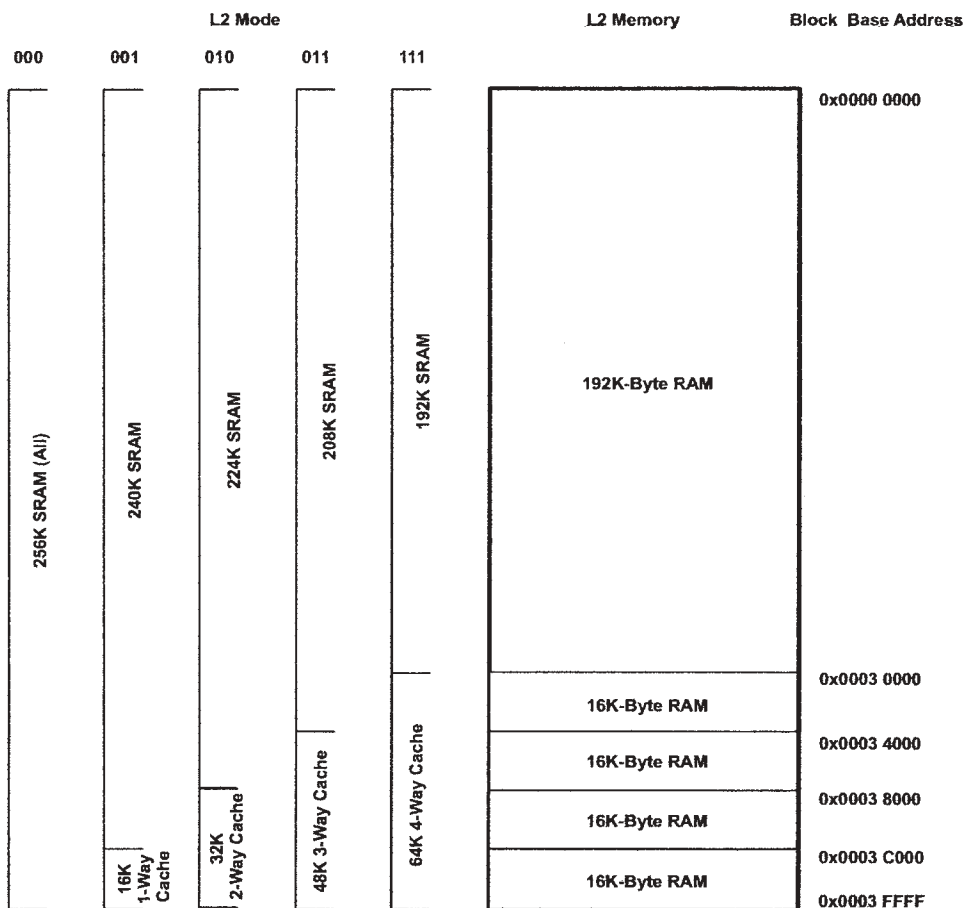


FIGURE 3.2. Internal memory configuration of L2 (Courtesy of Texas Instruments).

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.

**TABLE 3.1 Memory Map**

Memory Block Description	Block Size (Bytes)	Hex Address Range
Internal RAM (L2)	192 K	0000 0000–0002 FFFF
Internal RAM/cache	64 K	0003 0000–0003 FFFF
Reserved	24 M–256 K	0004 0000–017F FFFF
External memory interface (EMIF) registers	256 K	0180 0000–0183 FFFF
L2 registers	128 K	0184 0000–0185 FFFF
Reserved	128 K	0186 0000–0187 FFFF
HPI registers	256 K	0188 0000–018B FFFF
McBSP 0 registers	256 K	018C 0000–018F FFFF
McBSP 1 registers	256 K	0190 0000–0193 FFFF
Timer 0 registers	256 K	0194 0000–0197 FFFF
Timer 1 registers	256 K	0198 0000–019B FFFF
Interrupt selector registers	512	019C 0000–019C 01FF
Device configuration registers	4	019C 0200–019C 0203
Reserved	256 K–516	091C 0204–019F FFFF
EDMA RAM and EDMA registers	256 K	01A0 0000–01A3 FFFF
Reserved	768 K	01A4 0000–01AF FFFF
GPIO registers	16 K	01B0 0000–01B0 3FFF
Reserved	240 K	01B0 4000–01B3 FFFF
I2C0 registers	16 K	01B4 0000–01B4 3FFF
I2C1 registers	16 K	01B4 4000–01B4 7FFF
Reserved	16 K	01B4 8000–01B4 BFFF
McASP0 registers	16 K	01B4 C000–01B4 FFFF
McASP1 registers	16 K	01B5 0000–01B5 3FFF
Reserved	160 K	01B5 4000–01B7 BFFF
PLL registers	8 K	01B7 C000–01B7 DFFF
Reserved	264 K	01B7 E000–01BB FFFF
Emulation registers	256 K	01BC 0000–01BF FFFF
Reserved	4 M	01C0 0000–01FF FFFF
QDMA registers	52	0200 0000–0200 0033
Reserved	16 M–52	0200 0034–02FF FFFF
Reserved	720 M	0300 0000–2FFF FFFF
McBSP0 data port	64 M	3000 0000–33FF FFFF
McBSP1 data port	64 M	3400 0000–37FF FFFF
Reserved	64 M	3800 0000–3BFF FFFF
McASP0 data port	1 M	3C00 0000–3C0F FFFF
McASP1 data port	1 M	3C10 0000–3C1F FFFF
Reserved	1 G + 62 M	3C20 0000–7FFF FFFF
EMIF CE0*	256 M	8000 0000–8FFF FFFF
EMIF CE1*	256 M	9000 0000–9FFF FFFF
EMIF CE2*	256 M	A000 0000–AFFF FFFF
EMIF CE3*	256 M	B000 0000–BFFF FFFF
Reserved	1 G	C000 0000–FFFF FFFF

\* The number of EMIF address pins (EA[21:2]) limits the maximum addressable memory (SDRAM) to 128 MB per CE space.

Source: Courtesy of Texas Instruments.

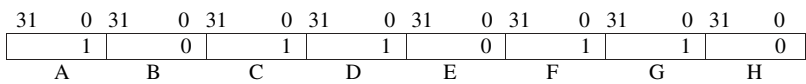
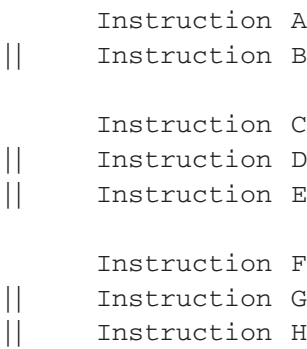


FIGURE 3.3. One FP with three EPs showing the “p” bit of each instruction.

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 FP.

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:



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.

3.5 PIPELINING

Pipelining is a key feature in a DSP 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 6 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

**TABLE 3.2    Pipeline Phases**

Program Fetch				Decode		Execute
PG	PS	PW	PR	DP	DC	E1–E6 (E1–E10 for double precision)

**TABLE 3.3    Pipelining Effects**

Clock Cycle											
1	2	3	4	5	6	7	8	9	10	11	12
PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6
	PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5
		PG	PS	PW	PR	DP	DC	E1	E2	E3	E4
			PG	PS	PW	PR	DP	DC	E1	E2	E3
				PG	PS	PW	PR	DP	DC	E1	E2
					PG	PS	PW	PR	DP	DC	E1
						PG	PS	PW	PR	DP	E1
							PG	PS	PW	PR	DC

to 10 phases (not all execution phases are shown in Table 3.3). We are assuming that each FP contains one 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 8LSBs 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 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 in assembly code using a circular buffer to update the “delay” samples. Implementing a circular buffer using C code is less efficient.

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 AMR. 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:

```
MVKL  .S2  0x0004 ,B2    ;lower 16 bits to B2. Select A5 as pointer
MVKH  .S2  0x0005 ,B2    ;upper 16 bits to B2. Select BK0, set N = 5
MVC   .S2  B2 ,AMR       ;move 32 bits of B2 to AMR
```

The two move instructions MVKL and MVKH (using the .S unit) move 0x0004 into the 16 LSBs of register B2 and 0x0005 into the 16 most significant bits (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 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  $0x0004 = (0100)_b$  into the 16 LSBs of AMR sets bit 2 (the third bit) to 1 and all other bits to 0. This sets the mode to 01 and selects register A5 as the pointer to a circular buffer using block BK0 (see Figure B.1).

Table 3.4 shows the modes associated with registers A4 through A7 and B4 through B7. The value  $0x0005 = (0101)_b$  into the 16 MSBs of AMR sets bits 16 and 18 to 1 (other bits to 0). 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  $(0110)_b = 0x0006$ . 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.

**TABLE 3.4 AMR Mode and Description**

Mode	Description
0 0	For linear addressing (default on reset)
0 1	For circular addressing using BK0
1 0	For circular addressing using BK1
1 1	Reserved

### 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 processors 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. Also, some additional instructions are available only 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 C6x 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 A 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 → 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 16LSBs of A7, B7 → B6
|| MPYH .M1    A7, B7, A6    ;multiply 16MSBs of A7, B7 → 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 MAC 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 to 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 C6713 can simultaneously load two 32-bit registers into side A and two 32-bit registers into side B.

**(b) The instruction**

```
STW   .D2  A1, *+A4[20]    ;store A1→(A4) offset by 20
```

stores the 32-bit word A1 in memory whose address is specified by A4 offset by 20 words (32 bits) or 80 bytes. The address register A4 is pre-incremented 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  MVKL   .S1  x, A4      ;move 16LSBs of x address →A4
      MVKH   .S1  x, A4      ;move 16MSBs 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 `MVKL`/`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` designates an uninitialized section named `variable` with a section size of 128 in bytes.

### 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 the functional unit and register to use (optional to be specified by the 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.

In recent years, the C compiler optimizer has become more and more efficient. Although C code is less efficient (speed performance) than assembly code, it typi-