Module - 4&5

Introduction to Digital Signal processors (DSp)

-By

Kavita Guddad

Objectives:

- Introduction to basics of digital signal processors.
- Processor architectures and hardware units
- Investigation on fixed-point and floating-point formats
- Illustrate the implementation of digital filters in real time.

- Unlike microprocessors and microcontrollers, digital signal (DS) processors have special features that require operations such as fast **Fourier transform (FFT), filtering, convolution and correlation, and real-time sample-based and block-based proc**essing.
- Hence DS processors use a different dedicated hardware architecture.
- By comparing the architecture of the general microprocessor with that of the DS processor we can understand how DS processors are different from general microprocessors

Architecture of DS Processors

- The design of general microprocessors and microcontrollers is based on the Von Neumann architecture, which was developed from a research paper written by John von Neumann and others in 1946.
- Von Neumann suggested that computer instructions, as we shall discuss, be numerical codes instead of special wiring.
- Figure shows the Von Neumann architecture.
- As shown in Figure 9.1, a Von Neumann processor contains a single, shared memory for programs and data, a single bus for memory access, an arithmetic unit, and a program control unit.
- The processor proceeds in a serial fashion in terms of fetching and execution cycles.
- This means that the central processing unit (CPU) fetches an instruction from memory and decodes it to figure out what operation to do and then executes the instruction.

Von-Neumann Architecture

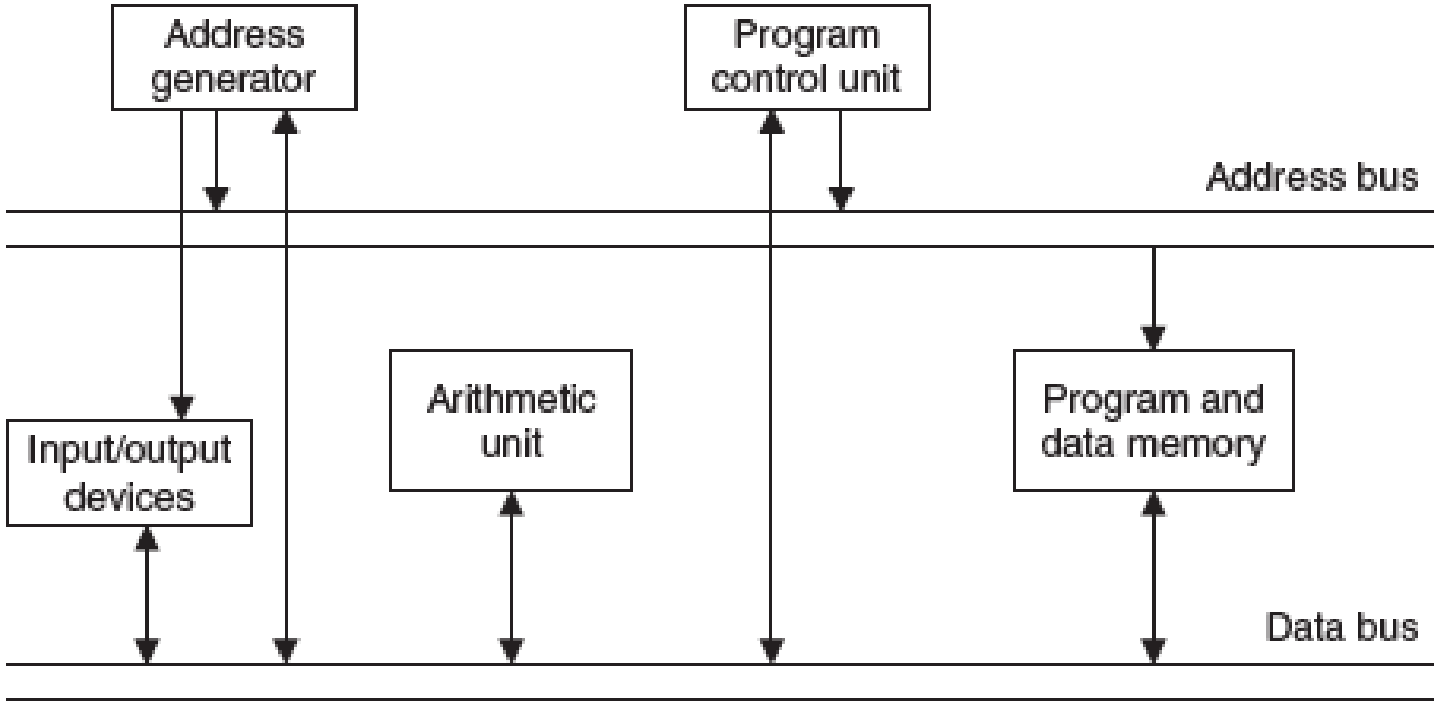


FIGURE 9.1 General microprocessor based on Von Neumann architecture.

- The instruction (in machine code) has two parts:

  the opcode and the operand.
- The opcode specifies what the operation is, that is, tells the CPU what to do.
- The operand informs the CPU what data to operate on.
- These instructions will modify memory, or input and output (I/O).
- After an instruction is completed, the cycles will resume for the next instruction.
- One instruction or piece of data can be retrieved at a time.
- Since the processor proceeds in a serial fashion, it causes most units to stay in a wait state.

- Hence the Von Neumann architecture operates the cycles of fetching and execution by fetching an instruction from memory, decoding it via the program control unit, and finally executing the instruction.
- When execution requires data movement—that is, data to be read from or written to memory—the next instruction will be fetched after the current instruction is completed.
- The Von Neumann–based processor has this bottleneck mainly due to the use of a single, shared memory for both program instructions and data. Increasing the speed of the bus, memory, and computational units can improve speed, but not significantly.
- To accelerate the execution speed of digital signal processing, DS processors are designed based on the Harvard architecture, which originated from the Mark 1 relay-based computers built by IBM in 1944 at Harvard University.
- This computer stored its instructions on punched tape and data using relay latches. Figure 9.2 shows today's Harvard architecture.

Harvard Architecture

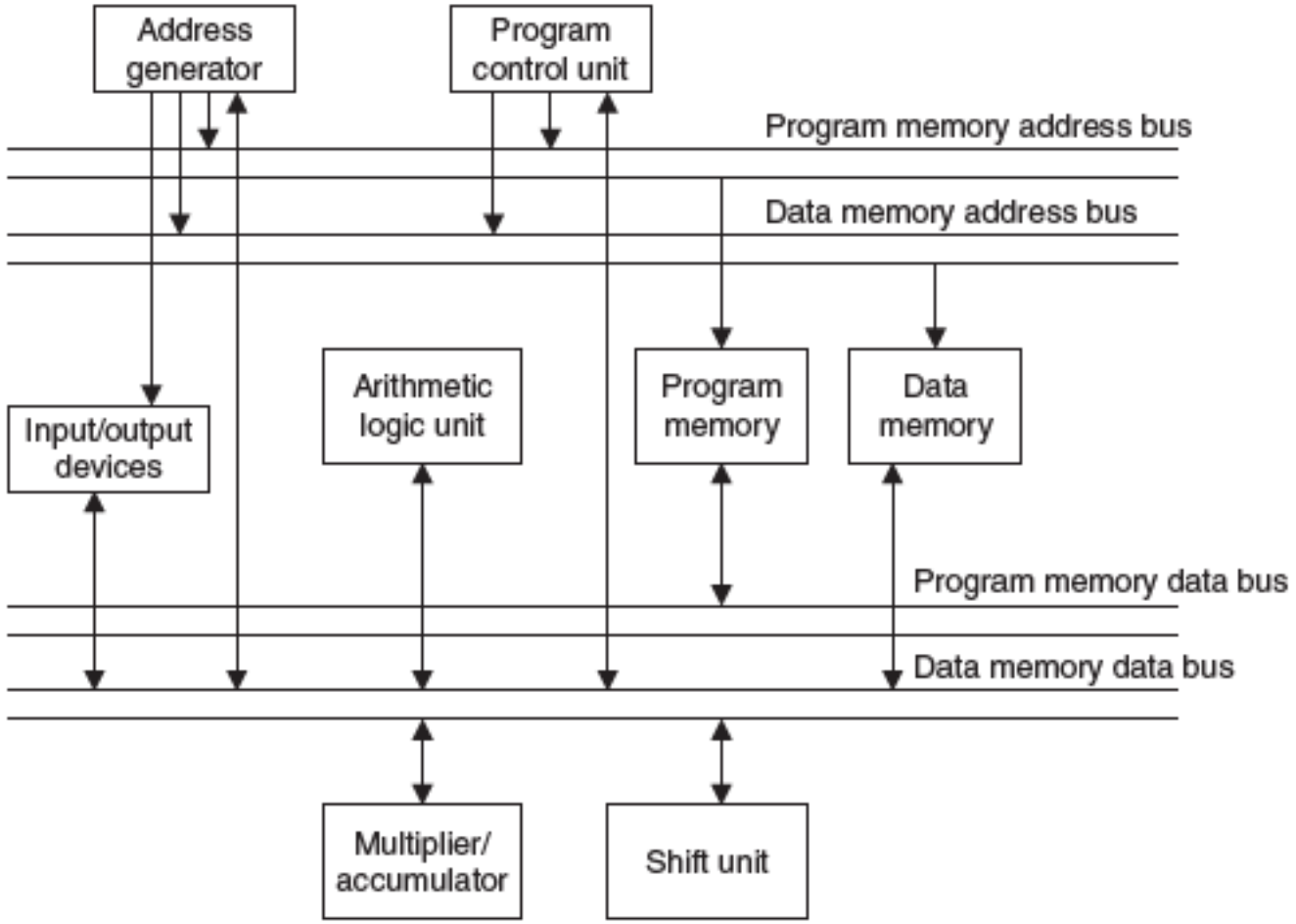


FIGURE 9.2 **Digital signal processors based on the Harvard architecture.**

- As shown in fig , the DS processor has two separate memory spaces. One is dedicated to the program code, while the other is employed for data.
- Hence, to accommodate two memory spaces, two corresponding address buses and two data buses are used. In this way, the program memory and data memory have their own connections to the program memory bus and data memory bus respectively.
- The Harvard processor can fetch the program instruction via the program memory bus and data via the data memory bus, in parallel at the same time
- There is an additional unit called a multiplier and accumulator (MAC), which is the dedicated hardware used for the digital filtering operation.
- The last additional unit, the shift unit, is used for the scaling operation for fixed-point implementation when the processor performs digital filtering.

Comparison of executions of the two architectures

- To compare the executions of the two architectures let us consider the execution cycles of The Von Neumann architecture and Harvard architecture
- The Von Neumann architecture generally has the execution cycles as shown below.

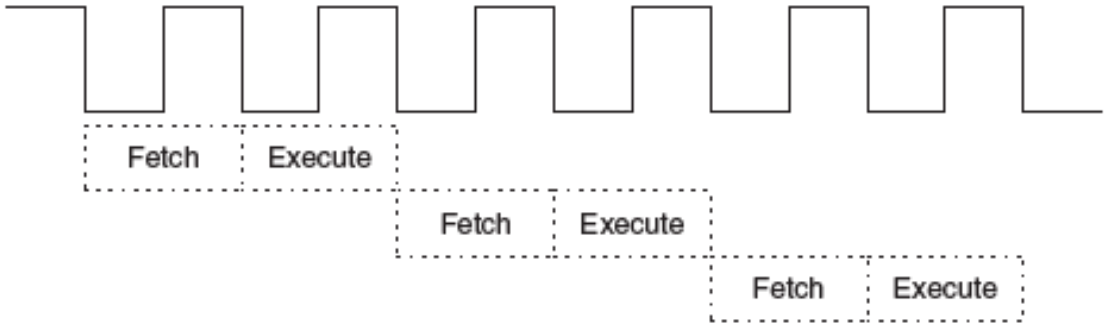


FIGURE 9.3 **Execution cycle based on the Von Neumann architecture.**

- The fetch cycle obtains the opcode from the memory, and the control unit will decode the instruction to determine the operation. Next is the execute cycle.
- Based on the decoded information, execution will modify the content of the register or the memory. Once this is completed, the process will fetch the next instruction and continue.
- The processor operates one instruction at a time in a serial fashion.

To improve the speed of the processor operation

- the Harvard architecture takes advantage of a common DS processor, in which one register holds the filter coefficient while the other register holds the data to be processed, as depicted in Figure 9.4.

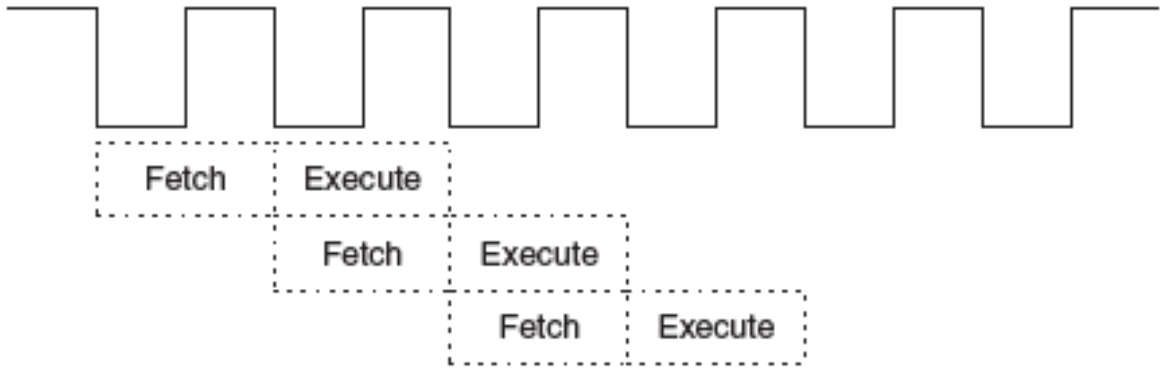


FIGURE 9.4 **Execution cycle based on the Harvard architecture.**

- As shown in Figure above the execute and fetch cycles are overlapped.
- This process of overlapping Fetch and Execute operations is known as the pipelining.
- The DS processor performs execution in one cycle while also fetching the next instruction to be executed. Hence, the processing speed is dramatically increased.

- The Harvard architecture is preferred for all DS processors due to the requirements of most DSP algorithms, such as filtering, convolution, and FFT, which need repetitive arithmetic operations, including multiplications, additions, memory access, and heavy data flow through the CPU.
- For other applications, dependent on simple microcontrollers with less of a timing requirement, the Von Neumann architecture may be a better choice, since it offers much less silica area and is thus less expansive.

Main manufacturers of DSp's are companies such as Texas Instruments, Motorola and Analog Devices .

Code Composer Studio (CCS) is software tool (provided by TI) used to work with C67x processors. It allows the user to build and debug programs from a user-friendly graphical user interface (GUI) and extends the capabilities of code development tools to include real-time analysis.

(Installation, tutorial, coding, and debugging can be found in the CCS Getting Started Guide (Texas Instruments, 2001) and in Kehtarnavaz and Simsek (2000).)

- 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.
- 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 can be a maximum of two cross-path source reads per cycle.
- There are 32 general purpose registers, but some of them are reserved for specific addressing or are used for conditional instructions.

For figure of architecture and memory mapping refer Text Number format Refer Text

C67x Architectural details(by Li Tan-This and next slide contain same info in short)

Architecture:

The system uses Texas Instruments Veloci2 architecture, which is an enhancement of the VLIW (very long instruction word architecture)

CPU:

The CPU has eight functional units divided into two sides A and B, each consisting of units .D, .M, .L, and .S. for each side.

.M unit is used for multiplication operations, .L unit is used for logical and arithmetic operations, .D unit is used for loading/storing and arithmetic operations.

Each side of the C67x CPU has sixteen 32-bit registers that the CPU must go through for interface.

Memory and internal buses:

Memory space is divided into internal program memory, internal data memory, and internal peripheral and external memory space.

The internal buses include a 32-bit program address bus, a 256-bit program data bus to carry out eight 32-bit instructions (VLIW), two 32-bit data address buses, two 64-bit load data buses, two 64-bit store data buses, two 32-bit DMA buses, and two 32-bit DMA address buses responsible for reading and writing.

There also exist a 22-bit address bus and a 32-bit data bus for accessing off-chip or external memory.

Peripherals:

a. EMIF, which provides the required timing for accessing external memory
b. DMA, which moves data from one memory location to another without interfering with the CPU operations
c. Multichannel buffered serial port (McBSP) with a high-speed multichannel serial communication link
d. HPI, which lets a host to access internal memory
e. Boot loader for loading code from off-chip memory or the HPI to internal memory
f. Timers (two 32-bit counters)
g . Power-down units for saving power for periods when the CPU is inactive.

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:

```
        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;
- EP3 contains the three parallel instructions F, G, and H.

- The FP would be as shown in Figure
- 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.

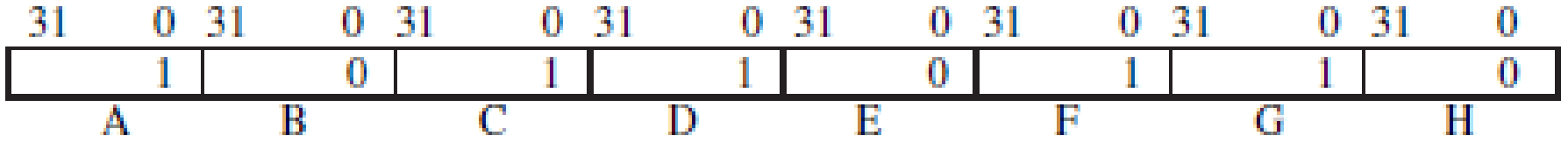


FIGURE 3.3. One FP with three EPs showing the “p” bit of each instruction.

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 a Fetch Packet(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).
- It is assumed that each Fetch packet contains one Execution packet.

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.”

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	DC

- 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.

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)(i.e the data of size >32 bits will be stored in two registers taken in order Odd register no : Even register no. Ex: A3:A2, B5:B4 etc…).

  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.

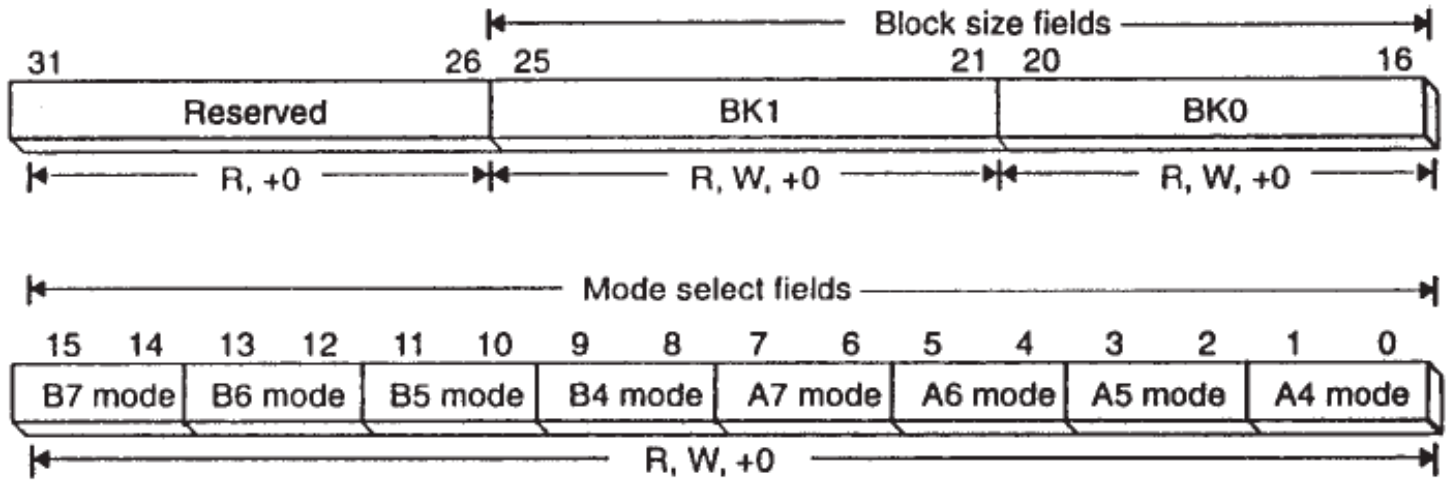


FIGURE B.1. Address mode register (AMR). (Courtesy of Texas Instruments)

(Appendix B)
Registers for Circular Addressing and Interrupts

A number of special-purpose registers available on the C6x processor are shown in Figures B.1 to B.8 [1].

1. Figure B.1 shows the address mode register (AMR) that is used for the circular mode of addressing. It is used to select one of eight register pointers (A4 through A7, B4 through B7) and two blocks of memories (BK0, BK1) that can be used as circular buffers.

2. Figure B.2 shows the control status register (CSR) with bit 0 for the global interrupt enable (GIE) bit.

3. Figure B.3 shows the interrupt enable register (IER).

4. Figure B.4 shows the interrupt flag register (IFR).

5. Figure B.5 shows the interrupt set register (ISR).

6. Figure B.6 shows the interrupt clear register (ICR).

7. Figure B.7 shows the interrupt service table pointer (ISTP).

8. Figure B.8 shows the serial port control register (SPCR).

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.

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.
- 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.

*R → Register R contains the address of a memory location where a data value is stored.

*R++(d) → Register R contains the memory address (location).

After the memory address is used, R is post incremented (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.

*R--(d) →A double minus (--) instead of a double plus would update or post decrement the address to R - d.

*++R(d) → The address is pre incremented or offset by d, such that the current address is R + d.

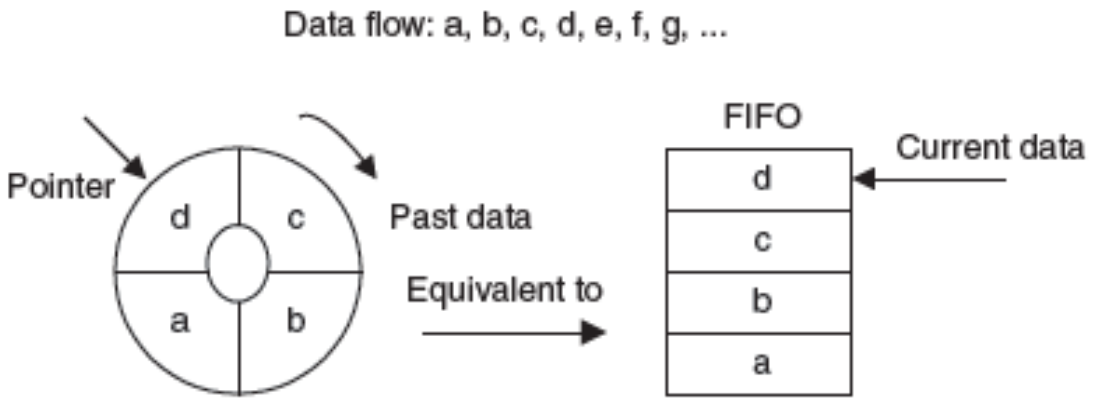
*--R(d) → A double minus would pre decrement the memory address so that the current address is R - d.

*+R(d) → The address is pre incremented by d, such that the current address R + d (as with the preceding case). However, in this case, R pre increments without modification. Unlike the previous case, R is not updated or modified.

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.

- 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

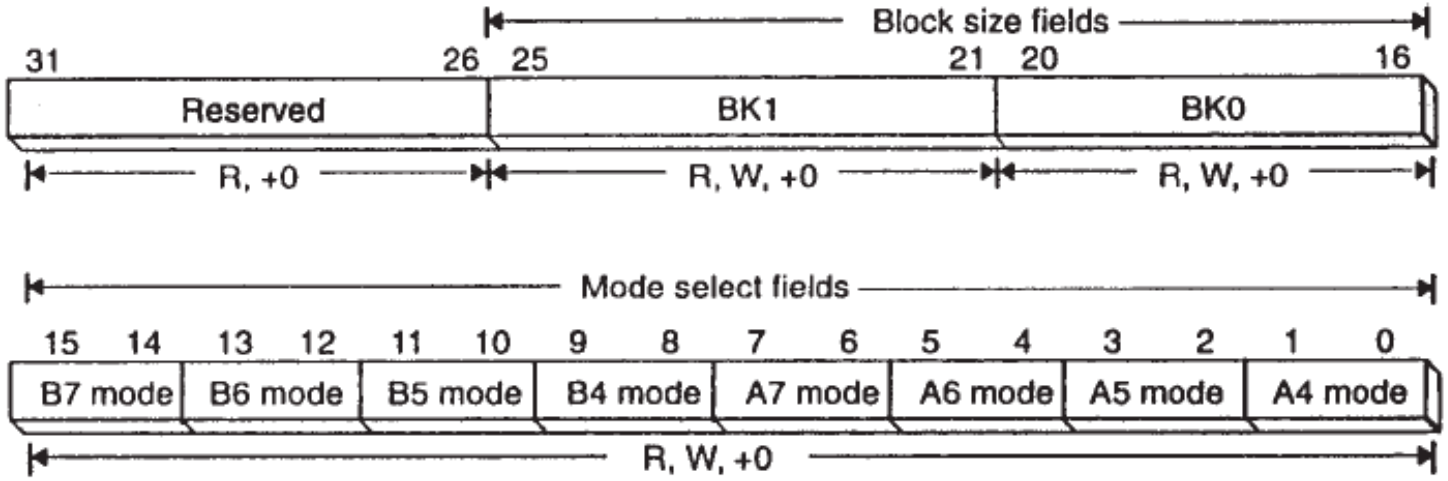


FIGURE B.1. Address mode register (AMR). (Courtesy of Texas Instruments)

```
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 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
- 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 in AMR register.
- The value 0x0005 = (0101)b into the 16MSBs 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.(2^(N+1)=128 means N=6)
- 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 below shows the modes associated with registers A4 through A7 and B4 through B7.

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

For Example to use registers A4(using Bko) and B5(using Bk1) for circular buffers of size 64 and 1024 respectively we need to load AMR with the pattern

- 00000000101011001 → move 0x0169 in upper 16 bits of B2
- 00001000000000001 → move 0x2049 in lower 16 bits of B2

i.e The code segment

```
MVKL  0x2049,B2
MVKH  0x0169,B2
MVC   B2,AMR
```

Will do the required job

Similarly Try :

Write code segment to set AMR to use registers A5(using Bk1) and B6(using Bko) for circular buffers of size 512 and 128 respectively we need to load AMR with the pattern

TMS320C6x INSTRUCTION SET

Assembly Code Format

An assembly code format is represented by the field

Label || [ ] Instruction Unit Operands ;comments

DSP instuctions have format

Mnemonic source, destination

where as GPPS have instruction in the *form*

Mnemonic destination, source

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 used 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.

example,

- [A2], specifies that the associated instruction executes if A2 is not zero.
- [!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.
Unit field → can be one of the eight CPU units, is optional(.L,.D,.S,.M).
It is optional to specify the eight functional units, although this can be useful during debugging and for code efficiency and optimization.

- The instruction (mnemonic or assembler directive) cannot start in column 1.
- The 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.

- The code for the fixed-point processors C62x is compatible with the code for the floating-point C67x.

Types of Instructions

Add/Subtract/Multiply

(a)ADD instruction

Format : *ADD .L1 A3, A7, A7* *; add A3 + A7 →A7 (accumulate in A7)*

Operation: 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) SUB instruction

Format: SUB .S1 A1, 1, A1 *;subtract 1 from A1 →A1*

Operation: Subtracts1 from A1 to decrement it using the .S unit.

(c) MPY instruction

MPY .M2 A7, B7, B6 ;multiply 16 LSBs of A7, B7 →B6

Operation: multiplies contents of A7 with Contents of B7 and stores the result in B6 using the .M2 unit.

(d)The parallel instructions

MPY .M2 A7,B7,B6 ;multiply 16 LSBs of A7, B7 →B6
|| MPYH .M1 A7,B7,A6 ;multiply 16MSBs of A7, B7 →A6

- It is a parallel instruction
- First instruction multiplies the lower or least significant 16 bits (LSBs) of both A7 and B7 and places the product in B6.
- Second instruction is executed in parallel (concurrently within the same execution packet) with a first and it multiplies the higher or most significant 16 bits (MSBs) of A7 and B7 and places the result in A6.
- Thus two MAC operations can be executed within a single instruction cycle.

- Thus parallel instructions 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.i.e the instruction is written as

MPY .M2 A7, B7, B6 || MPYH .M1 A7,B7,A6

2. Load/Store Instructions

(a) Load Instruction

The instruction

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

- It is a(post increment) indirect addressing mode instruction to load the half-word (16 bits) whose address in memory is specified/ pointed to by B2 into B7.
- The register B2 is incremented (post incremented) after loading the value 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.
- The A2 is incremented to point at the next higher memory address after loading.

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 instruction

LDDW

- Load the floating-point double-word on the C6713 ,can simultaneously load two 32-bit registers into side A and two 32-bit registers into side B.

(b) Store instruction

*STW .D2 A1, *+A4[20]* ;store A1→ address in A4 offset by 20

- It is a instruction for store operation using pre increment indirect addressing with displacement
- It 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 Instructions

- The branch instruction deviates the normal execution path based on a condition .

For Example the code segment

```
Loop    MVKL  .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*.
- The instructions MVKL/MVKH in order to get a 32-bit constant into a register.

- Here register A1 is used as a loop counter and is decremented each time(with the SUB instruction)
- A1 is tested for a conditional branch and 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.

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,

assembler directive *.sect "my_buffer"* → defines a section of code or data named *my_buffer*.

assembler directives *.text* and *.data* → indicate a section for text and data, respectively.

assembler directives *.ref* and *.def*, → used for undefined and defined symbols, respectively.

- The assembler creates several sections indicated by directives

  Such as *.text* → for code

  .bss → for global and static variables.

- Some commonly used assembler directives are:

.short → to initialize a 16-bit integer.

.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.

.float → to initialize a 32-bit IEEE single-precision constant.

.double →to initialize a 64-bit IEEE double-precision constant.

.byte, .short, or .int→ Initialize values

.usect → for uninitialized variables, it creates an uninitialized section (like the .bss section),

.sect → creates an initialized section.

For example,

.usect “variable”, 128 → designates an uninitialized section named variable with a section size of 128 in bytes.

INTERRUPTS

- An interrupt is a response by the processor to an event that needs attention from the processor.
- As the name implies, an interrupt causes the processor to halt whatever it is processing in order to execute an ISR (interrupt service routine).
- An interrupt can be issued externally or internally.
- RESET is the highest priority interrupt. It halts the CPU and initializes all the registers to their default values. (Table shows all interrupts of DSp with their priority.)
- When an interrupt occurs
  - The program flow is redirected to an ISR (Interrupt Set Register).
  - The conditions of the current process must be saved so that they can be restored after the interrupt task is performed.
  - Registers are saved and are restored after interrupt task is completed.
- There are 16 interrupt sources which include two timer interrupts, four external interrupts, four McBSP interrupts, and four DMA interrupts.
- Twelve CPU interrupts (INT4–INT15) are available which can be selected by an interrupt selector.

TABLE 3.5 Interrupt Service Table

Interrupt	Offset
RESET	000h
NMI	020h
Reserved	040h
Reserved	060h
INT4	080h
INT5	0A0h
INT6	0C0h
INT7	0E0h
INT8	100h
INT9	120h
INT10	140h
INT11	160h
INT12	180h
INT13	1A0h
INT14	1C0h
INT15	1E0h

Interrupt Control Registers

The interrupt control registers are as follows:

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 the 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 (Non maskable interrupt Return Pointer)

- Interrupts are prioritized
  - Reset having the highest priority .It is an active-low signal used to halt the CPU.
  - Nonmaskable interrupt (NMI) second highest priority.
- The RESET and NMI are external pins.
- 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).
- The IFR shows the status of different flags as shown in fig below.

  1-if an interrupt is set

  0- if reset

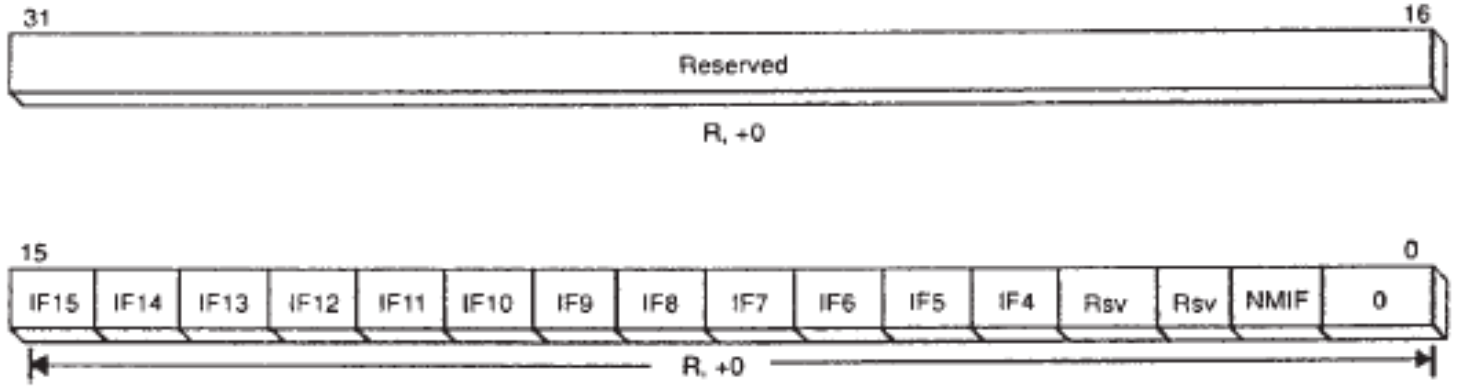


FIGURE B.4. Interrupt flag register (IFR). (Courtesy of Texas Instruments)

NMI

- NMI is nonmaskable Interrupt.
- NMI can be masked (disabled) by clearing the nonmaskable interrupt enable (NMIE) bit within IER(Interrupt Enable Register).
- 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.
- NMI signal alerts the CPU to a potential hardware problem.
- For an NMI to occur, the 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 interrupt may occur.

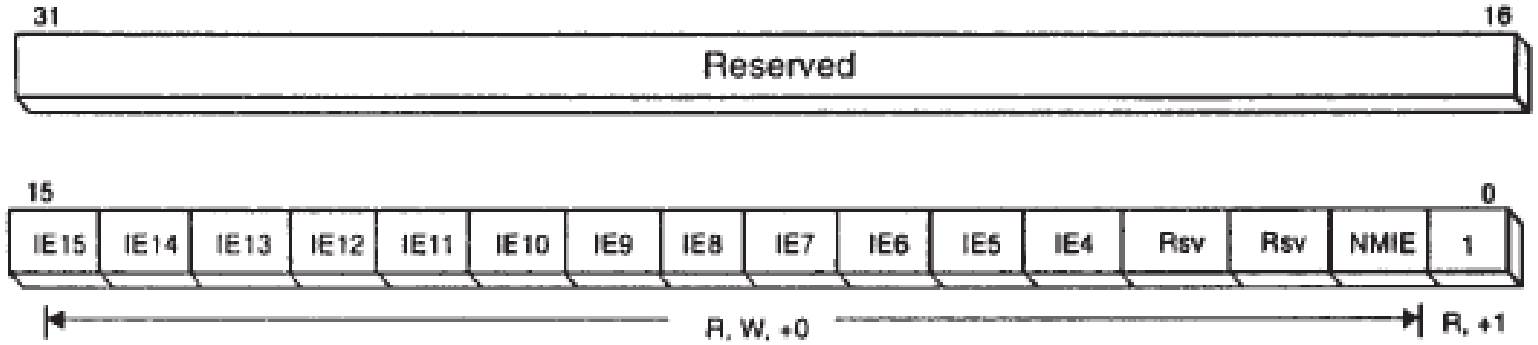


FIGURE B.3. Interrupt enable register (IER). (Courtesy of Texas Instruments)

Maskable interrupts

- INT4 through INT15 are twelve maskable CPU interrupts with lower priorities,
- The priorities of these interrupts are: INT4, INT5, . . . , INT15, with INT4 having the highest priority and INT15 the lowest priority.
- To process a maskable interrupt, the GIE (Global interrupt Enable) bit within the control status register (CSR) and the NMIE bit within IER are set to 1.

**Note that CSR can be ANDed with -2 (using 2's complement, the LSB is 0, while all other bits are 1's) to set the GIE bit to 0 and disable maskable interrupts globally.

- The interrupt enable (IE) bit corresponding to the desirable maskable interrupt is also set to 1 in IER.
- When the interrupt occurs, the corresponding 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.

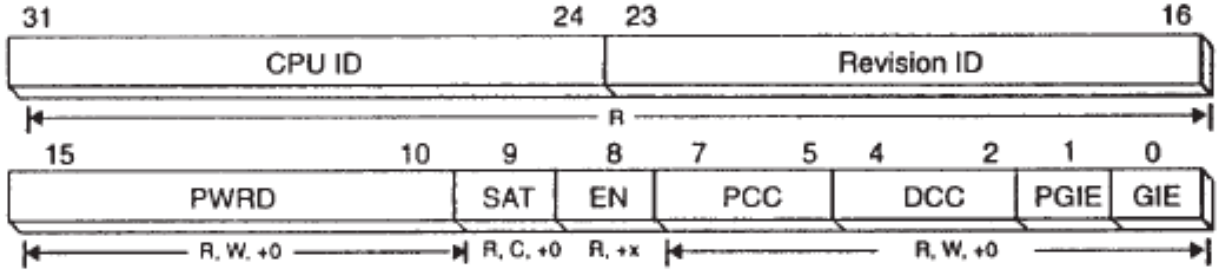


FIGURE B.2. Control status register (CSR). (Courtesy of Texas Instruments)

- 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 an FP associated with each interrupt.
- The table contains 16FPs, 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 FP contains eight 32-bit instructions (256 bits) or 32 bytes, each offset address in the table is incremented by 20 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.

Interrupt Acknowledgment

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

  For example, INUM3 = 1 (MSB), INUM2 = 0, INUM1 = 1, INUM0 = 1 (LSB) correspond to (1011)b = 11,

  indicating that INT11 is being processed.
- The IE11 bit is set to 1 to enable INT11. The 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 non-interruptible.
- Any pending interrupt will be processed as long as there are no pending branches to be completed.