11.2 Special Features of Digital Signal Processors

In the digital domain, a signal is represented by an array of N-numbers and as the value of N is large, the accuracy of representation will be high. The digital processing of signals involve operations like convolution and correlation, which in-turn involve multiplication and addition. Therefore, the real time processing of digital signals require very fast accessing of large volumes of data and very fast computation. The hardware requirement of digital signal processors to perform the above task are listed in table 11.2.

Table 11.2 : Special Requirements of Digital Signal Processors

Processing requirement	Hardware implementation to satisfy the processing requirement
Fast data access	• High-bandwidth memory architecture • Specialized addressing mode • Direct Memory Access (DMA)
Fast computation	• Multiply and accumulate unit • Pipelining of instruction execution • VLIW (Very Large Instruction Word) architecture • Multiprocessor architecture
Numerical fidelity	• Wide accumulator registers • Guard bits
Fast execution control	• Hardware-assisted zero-overhead loop • Shadow registers

11.2.1 Fast Data Access

In digital signal processors, the fast data access is achieved by employing high-bandwidth memory architecture, specialized addressing modes and Direct Memory Access (DMA). The high-bandwidth memory architecture are modified versions of Harvard architecture for simultaneous access of one or more data along with instruction code in a single clock cycle. The specialized addressing modes are provided for easy implementation of signal processing algorithms like FFT, convolution and correlation. The DMA help to transfer data from/to the external and internal memory without involving CPU, so that CPU is relieved for other task to run in parallel.

High-bandwidth Memory Architectures

The general purpose microprocessors are based on the ***Von Neumann architecture*** shown in fig 11.1, which consists of a single memory block to store both program and data and a single bus to transfer data and instruction from/to the CPU. The disadvantage in this architecture is that only one memory access per instruction cycle is possible.

The digital signal processors are based on either Harvard architecture shown in fig 11.2 or modified Harvard architecture shown in fig11.3. In ***Harvard architecture,*** there are separate memory blocks for program and data, and separate buses for transfer data and instruction from/to the CPU. The Harvard architecture facilitates the simultaneous access of instruction and data in a single cycle, i.e., two memory accesses in one cycle.

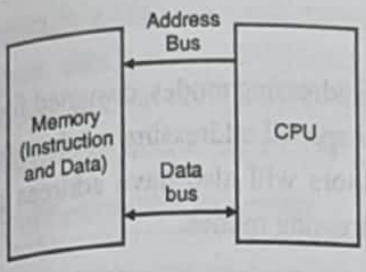


Fig 11.1 : Von Neumann architecture.

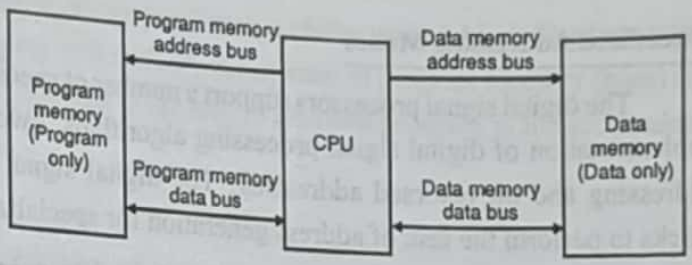


Fig 11.2 : Harvard architecture.

In ***modified Harvard architecture,*** one memory block is dedicated for storing data alone and another memory block for storing both instruction and data. This architecture will also have separate buses to access instruction and data simultaneously in one cycle.

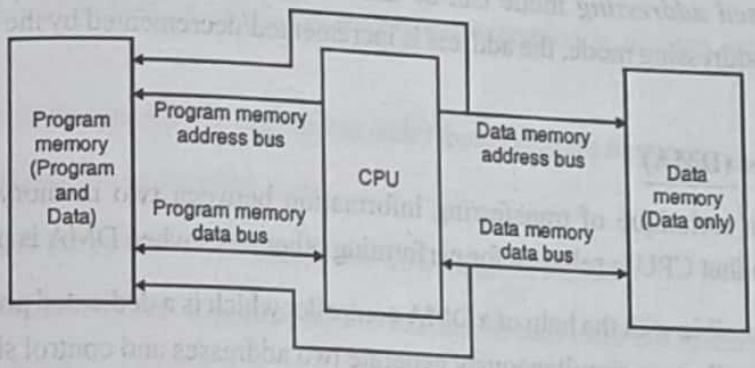


Fig 11.3 : Modified Harvard architecture.

The advanced digital signal processors employ two or more internal memory blocks connected to CPU via separate buses. It also has an external memory block common to program and data connected to CPU via a single external bus, as shown in fig 11.4.

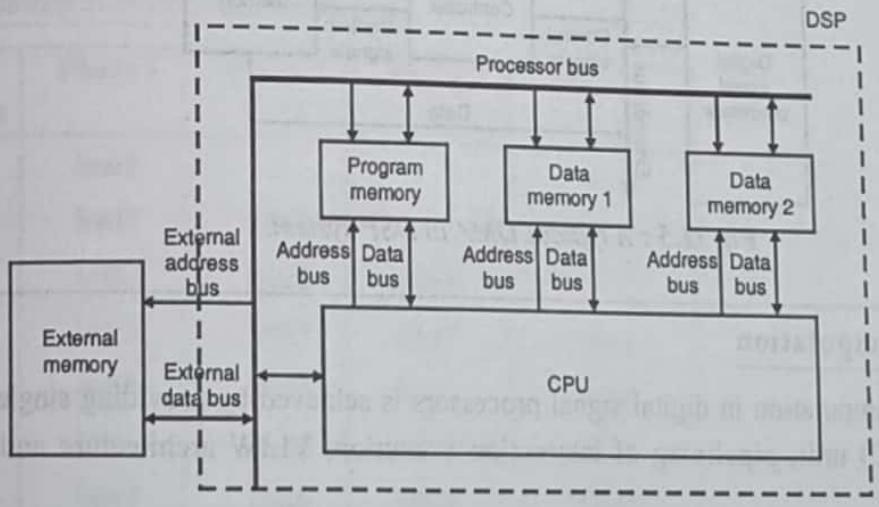


Fig 11.4 : Architecture of advanced digital signal processors.

In these processors, the program and data will be transferred from external memory to internal memory and execution starts from internal memory. Whenever the internal memory locations are free, the program and data will be copied from external memory to internal memory. In this way the execution speed is enhanced. The internal memory is also known as cache memory.

Specialized Addressing Modes

The digital signal processors support a number of specialized addressing modes, designed for efficient implementation of digital signal processing algorithms. Two of the special addressing modes are circular addressing and bit reversed addressing. The digital signal processors will also have address generator blocks to perform the task of address generation for specialized addressing modes.

The ***circular addressing*** mode can be used to access memory declared as circular buffer. The programmer can define a ***circular buffer*** by specifying a start address and end address. In this addressing, when the address pointer is incremented, the address will be checked with the end address of a circular buffer, and if the end address is reached then the pointer is loaded with the start address.

The ***bit reversed addressing*** mode can be used to access data in the bit reversed order for FFT computation. In this addressing mode, the address is incremented/decremented by the number represented in the bit reversed form.

Direct Memory Access (DMA)

The ***DMA*** is a technique of transferring information between two memory blocks/areas without involving the CPU, so that CPU is relieved for performing other tasks when DMA is performed.

The DMA is possible with the help of a DMA controller which is a dedicated processor for performing DMA. The DMA controller can simultaneously generate two addresses and control signals for data transfer between two memories or between memory and IO. Usually the digital signal processors employ DMA for bulk data transfer from the external memory to internal memory or vice versa.

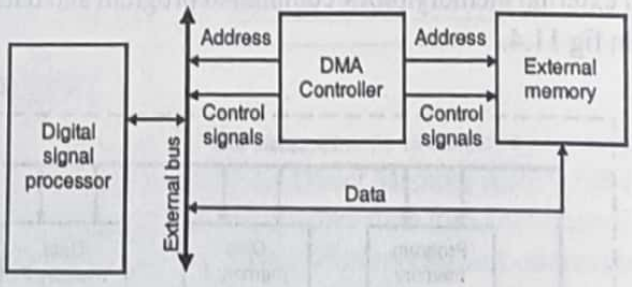


Fig 11.5 : A typical DMA in DSP system.

11.2.2 Fast Computation

The fast computation in digital signal processors is achieved by providing single cycle multiply/accumulate (MAC) unit, pipelining of instruction execution, VLIW architecture and multiprocessor architectures.

MAC (Multiply/Accumulate) Unit

The popular computations in digital signal processing are FFT, convolution and correlation. These operations involve multiplication and summation of lengthy numerical arrays. The ***MAC unit*** in the CPU of digital signal processors is capable of computing one multiplication and addition in a single clock cycle.

Typically a MAC unit will have a multiplier, a set of registers, a shifter and an ALU. The instruction "MACD pgm, dma", of the TMS320C5x processor will multiply the content of program memory (pgm) and data memory (dma) specified by the instruction and add to the sum of previous products in the accumulator with appropriate shift in a single clock cycle.

Pipelining of Instruction Execution

In processors without pipelining, the execution of instruction is performed one by one, i.e., after complete execution of an instruction the next instruction is fetched from memory. In processors with ***pipelining***, the instruction execution is divided into various phases/stages and execution of different phases of two or more instructions are performed in parallel. The number of instructions that can be executed in parallel is called ***depth*** or ***level of pipelining***.

Let us consider a processor in which the instruction execution is divided into the following four phases.

Phase 1 : Fetch the opcode (or instruction code) from program memory.

Phase 2 : Decode the instruction code.

Phase 3 : Read the operands (or data) from data/program memory.

Phase 4 : Execute the task specified by the instruction and store the result.

Let Inst1, Inst2, Inst3, ... be the instructions to be executed sequentially. The execution of the four phases of the instructions for subsequent clock cycles are listed in table 11.3.

In this pipelining when the phase 4 of 1st instruction is executed, the phase 3 of 2nd instruction, the phase 2 of 3rd instruction and the phase 1 of 4th instruction are also executed simultaneously.

Table 11.3 : Pipelining of Instruction Execution

Number of Clock Cycles	Phase 1	Phase 2	Phase 3	Phase 4
1	Inst1	–	–	–
2	Inst2	Inst1	–	–
3	Inst3	Inst2	Inst1	–
4	Inst4	Inst3	Inst2	Inst1
5	Inst5	Inst4	Inst3	Inst2
6	Inst6	Inst5	Inst4	Inst3
7	Inst7	Inst6	Inst5	Inst4
8	Inst8	Inst7	Inst6	Inst5
9	Inst9	Inst8	Inst7	Inst6
.	.	.	.	.
.	.	.	.	.

VLIW (Very Long Instruction Word) Architecture

The *VLIW architecture* has an enhanced parallelism in the instruction execution. In this processor, many instructions are fetched at the same time and issued to multiple execution units to be executed in parallel.

Consider an example of VLIW architecture shown in fig 11.6. In this example, the instructions are packed such that 8 numbers of 32-bit instructions are packet as a single 256-bit wide instruction and stored in on-chip program memory. When a packed instruction is fetched, it is simultaneously issued to 8 execution units to be executed in parallel. Here instruction packing is performed by the compiler during compile-time so as to ensure proper execution of the algorithm.

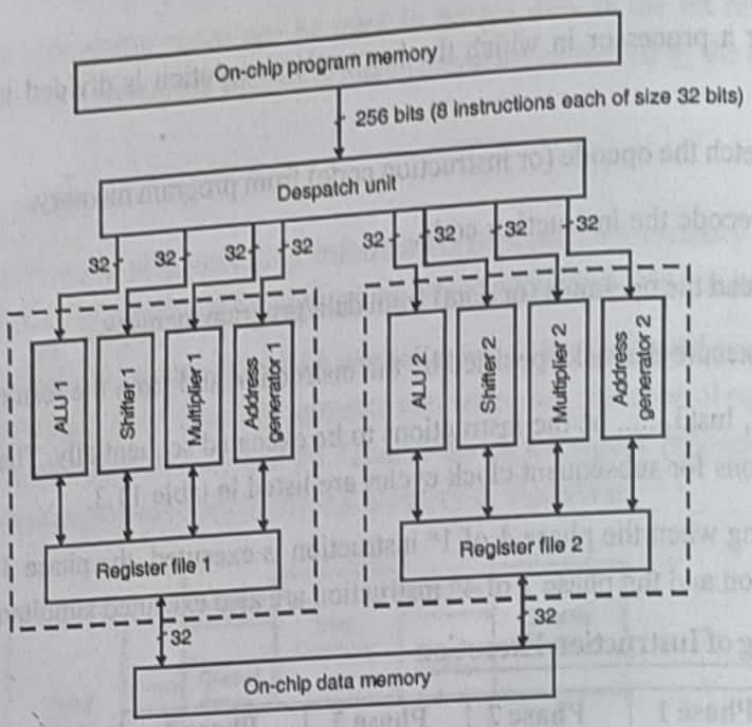


Fig 11.6 : VLIW architecture.

Multiprocessor Architecture

The *multiprocessor architecture* will consist of a number of digital signal processors running in parallel. For example, the TMS320C8x family of processors will have 4 parallel processors and one master processor. The master processor is a 32-bit RISC (Reduced Instruction Set Computer) processor with an internal floating point unit. Each parallel processor is an advanced 32-bit digital signal processors. The communication between the master processor and parallel processors takes place via a high-speed crossbar network which provides simultaneous access to multiple on-chip memory banks.

11.2.3 Numerical Fidelity

The *numerical fidelity* refers to the faithfulness of the digital signal processor to perform any mathematical operation without errors like underflow and overflow. These errors occur when the size of the result of any computation exceeds the size of the register used to store the result. In order to avoid such errors in computations, the digital signal processors will have larger size accumulators and CPU registers. Most of the 16-bit digital signal processors will have 32-bit accumulators and CPU registers, to store the result of any computations.

The TMS320C54x family of processors have 40-bit accumulator and ALU, in which the upper 8 bits are called ***guard bits***. The guard bits are used as headmargin for computations like convolution/correlation to avoid overflow.

In fixed point processors, fractional numbers will be used for computations. The range of fractional numbers is -1.0_{10} to $+1.0_{10}$. In binary, the fixed point fraction numbers are represented in 1.X format where the uppermost bit is used to represent the sign and X is the number of bits used to represent the magnitude. The 32-bit fixed point binary fraction number is represented in 1.31 format, where 31 bits is used to represent the magnitude. Now the smallest value that can be represented in this format is $1/2^{31}$ and the largest value is 1. The ***dynamic range*** in dB for a 32-bit fixed point number is defined as,

$$\text{Dynamic range}_{dB} = 20\log\left(\frac{\text{Largest value}}{\text{Smallest value}}\right) = 20\log\left(\frac{1}{1/2^{31}}\right) = -186.6_{dB}$$

11.2.4 Fast Execution Control

Some of the features of digital signal processors that help to achieve fast execution control are zero-overhead hardware loop and very fast interrupt handling by employing shadow registers.

The zero-overhead hardware loop allows the programmers to initialize loops by setting a counter and defining the loop bounds, without spending any software overhead to update and test loop counters or branching back to the beginning of the loop.

The interrupts are usually given higher priority. When an interrupt occurs while executing a program, the processor will stop the current execution and execute an Interrupt Service Routine (ISR) to service the interrupt. After executing the ISR, the processor will resume the program execution. When the program execution is stopped, the processor has to store the vital informations in the CPU registers and load them back to CPU registers when the program execution is resumed. In digital signal processors, a portion of internal/stack memory locations are dedicated for this purpose and they are called ***shadow registers***, which ensure very fast storage and retrial of information.

11.3 TMS320C5x Family of Digital Signal Processors

The TMS320C5x family of processors are fifth-generation digital signal processors from Texas Instruments, USA. They are 16-bit fixed point processors fabricated using high performance static CMOS technology. These processors have advanced Harvard architecture, with a variety of on-chip peripherals and memory and highly specialized instructions. They can execute 50 Million Instructions Per Second(MIPS).

Some of the features of TMS320C5x family of digital signal processors are,

- 16-bit CPU
- 20 to 50 ns single cycle instruction execution time
- Single cycle 16 × 16-bit MAC (Multiply/Accumulate) unit
- 64k × 16-bit external program memory address space
- 64k × 16-bit external data memory address space
- 64k × 16-bit external IO address space

- 32k × 16-bit external global memory address space
- 2k to 32k × 16-bit single-access On-chip PROM
- 1k to 9k × 16-bit single-access On-chip program/data RAM
- 1k × 16-bit dual-access On-chip program/data RAM
- Synchronous, TDM and buffered serial ports
- Programmable timer and PLL (Phase Locked Loops)
- IEEE standard JTAG ports
- 5 V/3 V operation with low power dissipation and power down modes
- DMA interface
- 100/128/132/144 pins in plastic QFP and TQFP

The various processors of TMS320C5x family and their characteristic features are listed in table 11.4.

Table 11.4 : Characteristics of TMS320C5x Family of Processors

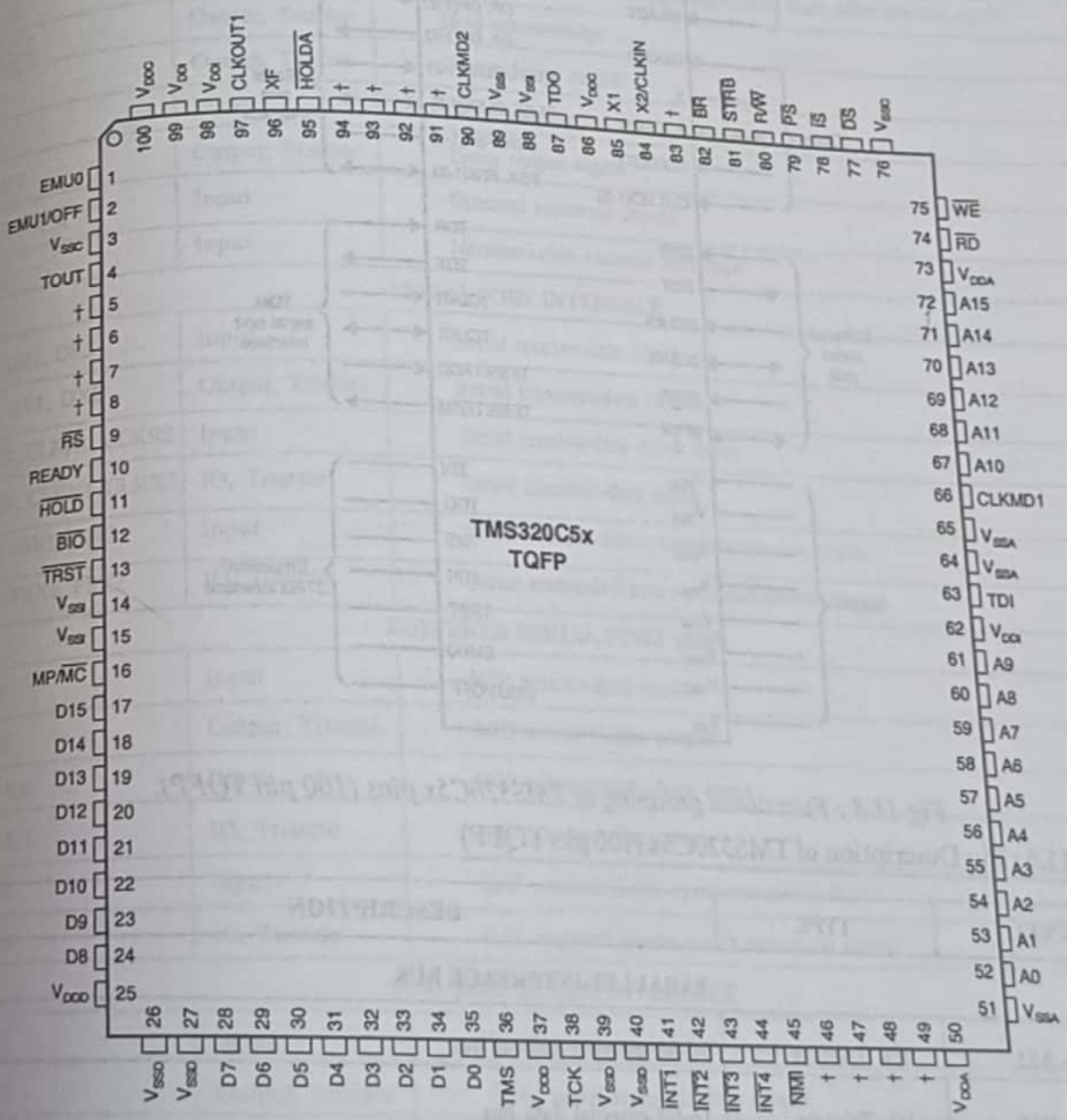
PROCESSOR	ON-CHIP MEMORY (16-BIT WORDS)				IO PORTS		POWER SUPPLY (V)	CYCLE TIME (ns)	NUMBER OF PINS
	DARAM		SARAM	ROM					
	DATA	DATA+ PROG	DATA+ PROG	PROG	SERIAL	PARALLEL			
TMS320C50	544	512	9k	2k	2	64k	5	50/35/25	132 pin
TMS320LC50	544	512	9k	2k	2	64k	3.3	50/40/25	132 pin
TMS320C51	544	512	1k	8k	2	64k	5	50/35/25/20	100/132 pin
TMS320LC51	544	512	1k	8k	2	64k	3.3	50/40/25	100/132 pin
TMS320C52	544	512	–	4k	1	64k	5	50/35/25/20	100 pin
TMS320LC52	544	512	–	4k	1	64k	3.3	50/40/25	100 pin
TMS320C53	544	512	3k	16k	2	64k	5	50/35/25	132 pin
TMS320LC53	544	512	3k	16k	2	64k	3.3	50/40/25	132 pin
TMS320C53S	544	512	3k	16k	2	64k	5	50/35/25	100 pin
TMS320LC53S	544	512	3k	16k	2	64k	3.3	50/40/25	100 pin
TMS320LC56	544	512	6k	32k	2	64k	3.3	35/25	100 pin
TMS320LC57	544	512	6k	32k	2	64k+HPI	3.3	35/25	128 pin
TMS320C57S	544	512	6k	2k	2	64k+HPI	5	50/35/25	144 pin
TMS320LC57S	544	512	6k	2k	2	64k+HPI	3.3	50/35	144 pin

11.3.1 Pin Diagram of TMS320C5x Processors

The TMS320C5x family of processors are available in the following plastic packages.

- 100/132 pins QFP (Quad Flat Package)
- 100/128/144 pins TQFP (Thin Quad Flat Package)

The pin configuration of TMS320C5x processors with 100 pins in TQFP is shown in fig 11.7. The functional groupings of the pins are shown in fig 11.8. The names or functions of various pins are listed in table 11.5.



Note : † See Table 11.6 for device-specific pinouts.

Fig 11.7 : Pin diagram of TMS320C5x (TQFP).

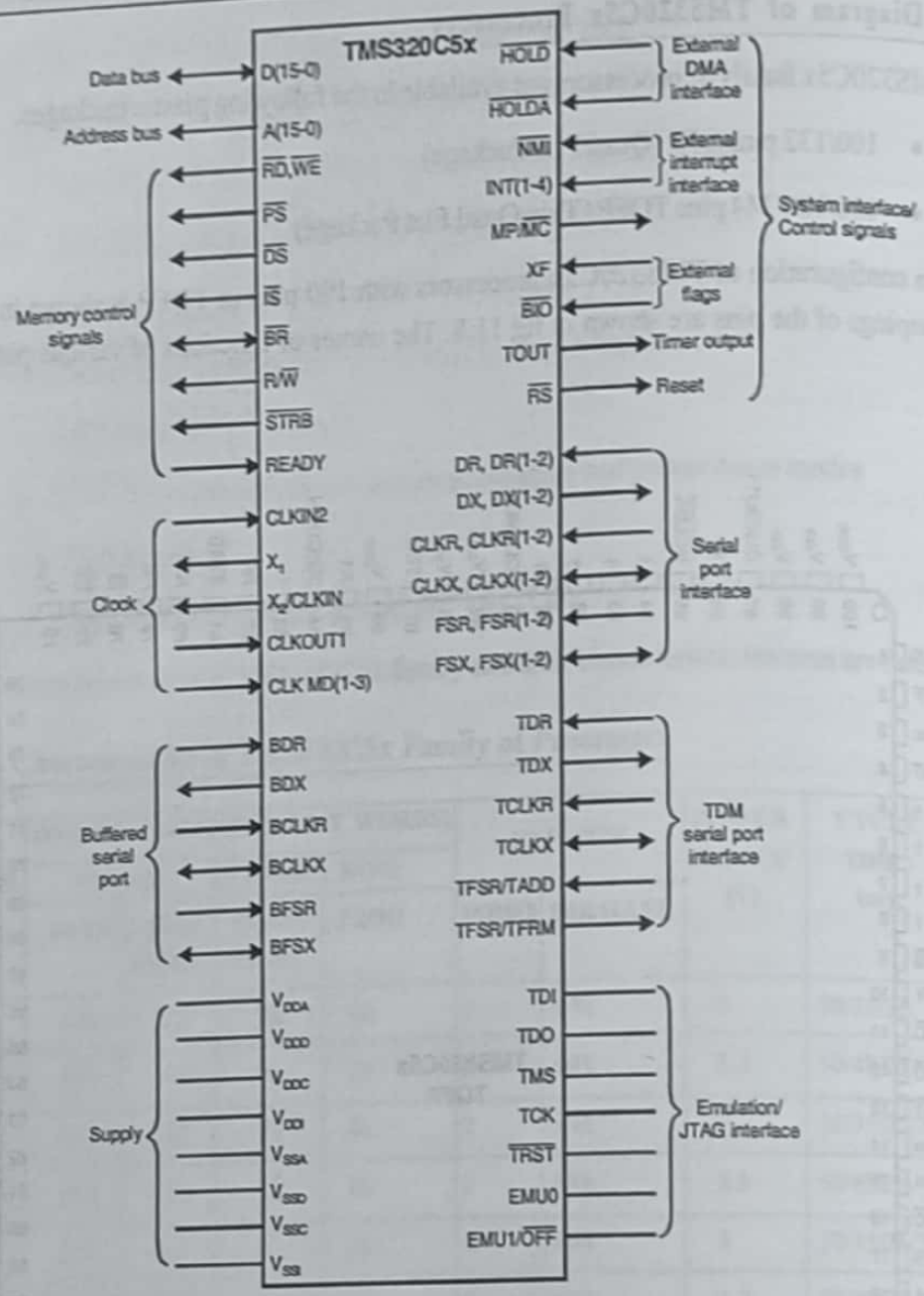


Fig 11.8 : Functional grouping of TMS320C5x pins (100 pin TQFP).

Table 11.5 : Pin Description of TMS320C5x (100 pin TQFP)

SIGNAL	TYPE	DESCRIPTION
	PARALLEL INTERFACE BUS	
A0 – A15	IO, Tristate	16-bit external address bus
D0 – D15	IO, Tristate	16-bit external data bus
$\overline{PS}$, $\overline{DS}$, $\overline{IS}$	Output, Tristate	Program, data and IO space select outputs, respectively
$\overline{STRB}$	IO, Tristate	Timing strobe for external cycles and external DMA
R / $\overline{W}$	IO, Tristate	Read/write select for external cycles and external DMA
$\overline{RD}$, $\overline{WE}$	Output, Tristate	Read and write strobes, respectively, for external cycles

Table 11.5 : Continued...

SIGNAL	TYPE	DESCRIPTION
READY	Input	External bus ready/wait-state control input
$\overline{BR}$	IO, Tristate	Bus request
	SYSTEM INTERFACE/CONTROL SIGNALS	
$\overline{RS}$	Input	Reset
MP/$\overline{MC}$	Input	Microprocessor/microcomputer mode select
$\overline{HOLD}$	Input	Put parallel I/F bus in high-impedance state after current cycle
$\overline{HOLDA}$	Output, Tristate	Hold acknowledge
XF	Output, Tristate	External flag output
$\overline{BIO}$	Input	IO branch control input
TOUT	Output, Tristate	Timer output signal
$\overline{INT1}$ – $\overline{INT4}$	Input	External interrupt inputs
$\overline{NMI}$	Input	Nonmaskable external interrupt
	SERIAL PORT INTERFACE	
DR, DR1, DR2	Input	Serial receive-data input
DX, DX1, DX2	Output, Tristate	Serial transmit-data output
CLKR, CLKR1, CLKR2	Input	Serial receive-data clock input
CLKX, CLKX1, CLKX2	IO, Tristate	Serial transmit-data clock
FSR, FSR1, FSR2	Input	Serial receive-frame-synchronization input
FSX, FSX1, FSX2	IO, Tristate	Serial transmit-frame-synchronization signal
	BUFFERED SERIAL PORT (BSP)	
BDR	Input	BSP receive-data input
BDX	Output, Tristate	BSP transmit-data output
BCLKR	Input	BSP receive-data clock input
BCLKX	IO, Tristate	BSP transmit-data clock
BFSR	Input	BSP receive frame-synchronization input
BFSX	IO, Tristate	BSP transmit frame-synchronization signal
	TDM SERIAL PORT INTERFACE	
TDR	Input	TDM serial receive-data input
TDX	Output, Tristate	TDM serial transmit-data output
TCLKR	Input	TDM serial receive-data clock input
TCLKX	IO, Tristate	TDM serial transmit-data clock
TFSR/TADD	IO, Tristate	TDM serial receive-frame-synchronization input
TFSX/TFRM	Input	TDM serial transmit-frame-synchronization signal

Table 11.5 : Continued...

SIGNAL	TYPE	DESCRIPTION
		EMULATION/JTAG INTERFACE
TDI	Input	JTAG-test-port scan data input
TDO	Output, Tristate	JTAG-test-port scan data output
TMS	Input	JTAG-test-port mode select input
TCK	Input	JTAG-port clock input
$\overline{TRST}$	Input	JTAG-port reset (with pull-down resistor)
EMU0	IO, Tristate	Emulation control 0
EMU1/$\overline{OFF}$	IO, Tristate	Emulation control 1
		CLOCK GENERATION AND CONTROL
X1	Output	Oscillator output
X2/CLKIN, CLKIN2	Input	Clock/oscillator input
CLKMD1, CLKMD2	Input	Clock-mode select inputs
CLKOUT1	Output, Tristate	Device system-clock output
		POWER SUPPLY CONNECTIONS
VDDA	Supply	Supply connection, address-bus output
VDDD	Supply	Supply connection, data-bus output
VDDC	Supply	Supply connection, control output
VDDI	Supply	Supply connection, internal logic
VSSA	Supply	Supply connection, address-bus output
VSSD	Supply	Supply connection, data-bus output
VSSC	Supply	Supply connection, control output
VSSI	Supply	Supply connection, internal logic

Table 11.6 : Device-Specific Pinouts for the 100 pin TQFP

PIN	TMS320C51 TMS320LC51	TMS320C52 TMS320LC52	TMS320C53S TMS320LC53S	TMS320LC56
5	TCLKX	V_{SS}	CLKX2	BCLKX
6	CLKX	CLKX	CLKX1	CLKX
7	TFSR/TADD	V_{SS}	FSR2	BFSR
8	TCLKR	V_{SS}	CLKR2	BCLKR
46	DR	DR	DR1	DR
47	TDR	V_{SS}	DR2	BDR
48	FSR	FSR	FSR1	FSR
49	CLKR	CLKR	CLKR1	CLKR

Table 11.6 : Continued...

PIN	TMS320C51 TMS320LC51	TMS320C52 TMS320LC52	TMS320C53S TMS320LC53S	TMS320LC56
83	CLKIN2	CLKIN2	CLKIN2	CLKMD3
91	FSX	FSX	FSX1	FSX
92	TFSX/TFRM	V_{SS}	FSX2	BFSX
93	DX	DX	DX1	DX
94	TDX	NC	DX2	BDX

11.3.2 Architecture of TMS320C5x Processors

The TMS320C5x processors have an advanced version of Harvard architecture, with separate buses for program and data, which facilitate simultaneous access of program and data. The program bus has separate lines to transmit data and address. Similarly the data bus has separate lines to transmit data and address.

The internal architecture of TMS320C5x processor is shown in fig 11.9. The architecture of TMS320C5x processors can be broadly divided into three major areas. They are, CPU (Central Processing Unit), memory and peripherals.

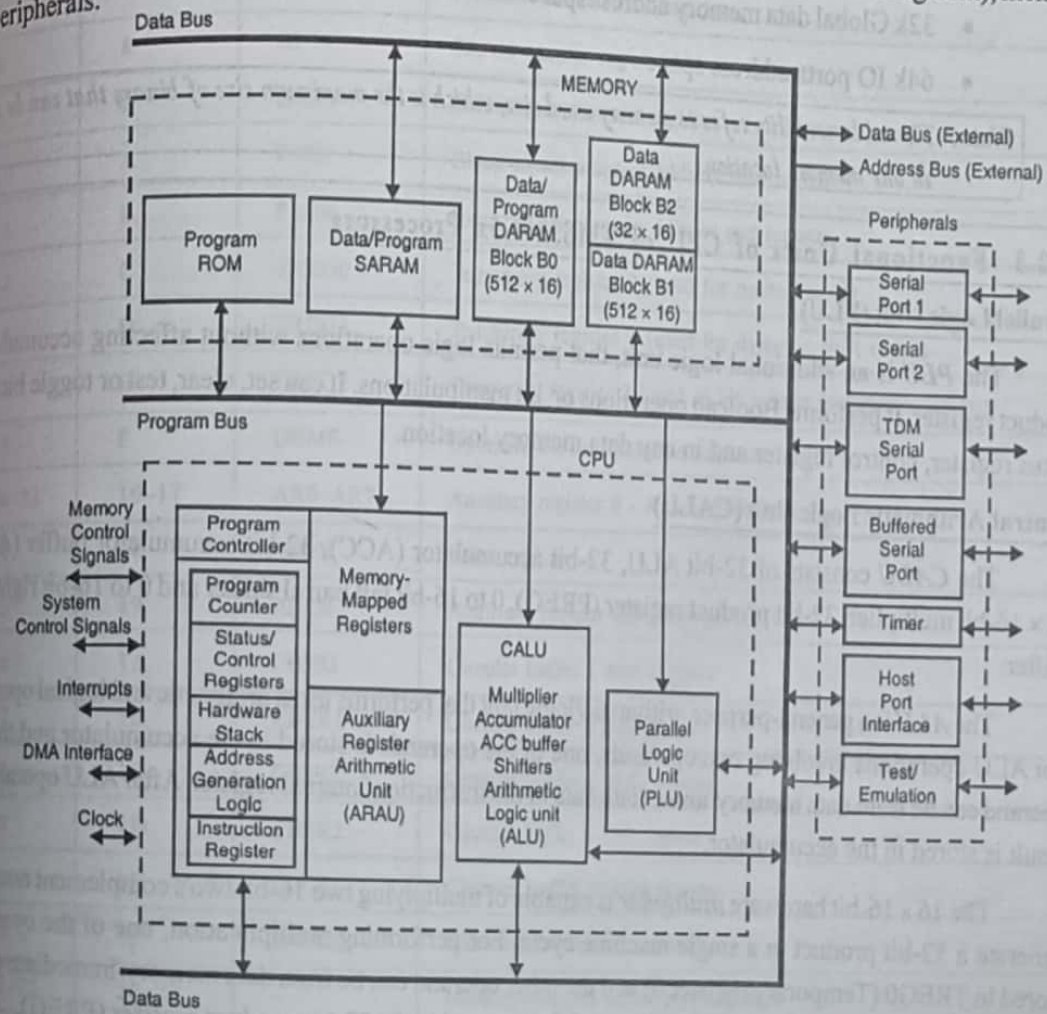


Fig 11.9 : Internal architecture of TMS320C5x.

The functional units of CPU are Parallel Logic Unit (PLU), central ALU, memory mapped registers, Auxiliary Register Arithmetic Unit (ARAU) and program controller.

The TMS320C5x processors has the following internal (or on-chip) memory.

- Program ROM (2k to 32k words)
- Data/Program Dual Access RAM (DARAM) (1024 + 32 = 1056 words)
- Data/Program Single Access RAM/(SARAM) (1k to 9k words)

The various on-chip or internal peripherals of TMS320C5x processors are clock generator, hardware timer, software programmable wait state generators, parallel IO ports, Host Port Interface (HPI), serial port, Buffered Serial Port (BSP), Time Division Multiplexed (TDM) serial port and user maskable interrupts.

The TMS320C5x processors have a total memory address space of 224k (including on-chip memory) with addressability of 16 bits. This address space is divided into four individually selectable address spaces as follows.

- 64k Program memory address space
- 64k Local data memory address space
- 32k Global data memory address space
- 64k IO ports address space

Note : The addressability refer to memory word size, which is the maximum size of binary that can be stored in one memory location.

11.3.3 Functional Units of CPU of TMS320C5x Processors

Parallel Logic Unit (PLU)

The ***PLU*** is an additional logic unit, that permits logic operations without affecting accumulator or product register. It performs Boolean operations or bit manipulations. It can set, clear, test or toggle bits in the status register, control register and in any data memory location.

Central Arithmetic Logic Unit (CALU)

The ***CALU*** consists of 32-bit ALU, 32-bit accumulator (ACC), 32-bit accumulator buffer (ACCB), 16 × 16-bit multiplier, 32-bit product register (PREG), 0 to 16-bit left barrel shifter and 0 to 16-bit right barrel shifter.

The ***ALU*** is a general-purpose arithmetic/logic unit that performs usual arithmetic and logical operations. For ALU operations involving two operands, one of the operands is stored in the accumulator and the other operand can be from data memory/immediate data in the instruction/internal register. After ALU operation, the result is stored in the accumulator.

The 16 × 16-bit hardware ***multiplier*** is capable of multiplying two 16-bit two's complement numbers to generate a 32-bit product in a single machine cycle. For performing multiplication, one of the operands is stored in TREG0 (Temporary register 0) and the other operand can be from data memory/immediate operand in the instruction. After multiplication, the product is stored in the 32-bit ***product register*** (PREG).

The 0 to 16-bit left and right shifter permit the content of memory to be shifted before loading into ALU and vice versa. The content of accumulator (ACC) and product register (PREG) can also be shifted using these shifters.

Memory-Mapped Registers

The TMS320C5x has 96 numbers of 16-bit memory-mapped registers, and they are mapped into page-0 of data memory space. The memory-mapped registers includes various control and status registers for CPU, serial port, timer and software wait-state generators. Also they include 16 memory-mapped IO ports. The memory-mapped registers along with their memory address are listed in table 11.7.

Table 11.7 : Memory-Mapped Registers of TMS320C5x Processors

ADDRESS		NAME	DESCRIPTION
DEC	HEX		
0–3	0–3	–	Reserved
4	4	IMR	Interrupt mask register
5	5	GREG	Global memory allocation register
6	6	IFR	Interrupt flag register
7	7	PMST	Processor mode status register
8	8	RPTC	Repeat counter register
9	9	BRCR	Block repeat counter register
10	A	PASR	Block repeat program address start register
11	B	PAER	Block repeat program address end register
12	C	TREG0	Temporary register 0 (used for multiplicand)
13	D	TREG1	Temporary register 1 (used for dynamic shift count)
14	E	TREG2	Temporary register 2 (used as bit pointer in dynamic bit test)
15	F	DBMR	Dynamic bit manipulation register
16–23	10–17	AR0–AR7	Auxiliary register 0 - Auxiliary register 7
24	18	INDX	Index register
25	19	ARCR	Auxiliary register compare register
26	1A	CBSR1	Circular buffer 1 start register
27	1B	CBER1	Circular buffer 1 end register
28	1C	CBSR2	Circular buffer 2 start register
29	1D	CBER2	Circular buffer 2 end register
30	1E	CBCR	Circular buffer control register
31	1F	BMAR	Block move address register
32–35	20–23	–	Memory-mapped serial port registers

Table 11.7: Continued...

ADDRESS		NAME	DESCRIPTION
DEC	HEX		
36–42	24–2A	–	Memory-mapped peripheral registers
43–47	2B–2F	–	Reserved for test/emulation
48–55	30–37	–	Memory-mapped serial port registers
56–79	38–4F	–	Reserved
80–95	50–5F	–	Memory-mapped IO ports

Auxiliary Register Arithmetic Unit (ARAU)

The *ARAU* contains eight 16-bit auxiliary registers AR0-AR7, a 3-bit Auxiliary Register Pointer (ARP), a 16-bit index register (INDX) and a 16-bit Auxiliary Register Compare Register (ARCR). An unsigned 16-bit arithmetic unit in the ARAU is used to calculate indirect addresses, using the contents of ARP, INDX and ARCR registers. Therefore, the CALU is relieved from task of address manipulation and so it is free for other operations in parallel.

The auxiliary registers can also be used as general-purpose registers for holding the operands for arithmetic and logical operation in CALU.

Program Controller

The ***program controller*** contains logic circuits that decodes the instructions, manages the CPU pipeline, stores the status of CPU operations and decodes the conditional operations. Due to parallelism in architecture, the program controller can perform three concurrent or simultaneous memory operations in any given machine cycle. They are fetch an instruction, read an operand and write an operand.

The program controller unit consists of a 16-bit Program Counter (PC), 16-bit status registers ST0 and ST1, Processor Mode Status register (PMST) and Circular Buffer Control Register (CBCR), a 16 × 16-bit hardware stack, address generation logic, instruction register, interrupt flag register and interrupt mask register.

Status Registers (ST0 and ST1)

The TMS320C5x processors have two 16-bit ***status registers*** (ST0 and ST1) which holds the status of ALU result, pointers for indirect addressing and various bits for interrupt control, hold mode and product shift mode. The format of status registers are shown in fig 11.10 and 11.11. The functions of various bits of status registers are listed in Table 11.8 and 11.9.

The status registers can be stored into data memory and loaded from data memory, thereby allowing the processor status to be saved and restored for subroutines. The LST instruction writes to ST0 and ST1, and the SST instruction reads from them, except that the ARP bits and INTM bit are not affected by the LST #0 instruction.

The ST0 and ST1 each have an associated 1-level deep shadow register stack for automatic context-saving when an interrupt occurs. The INTM and OVM bits in ST0 and the C, CNF, HM, SXM, TC and XF bits in ST1 can be individually set using the SETC instruction or individually cleared using the CLRC instruction.

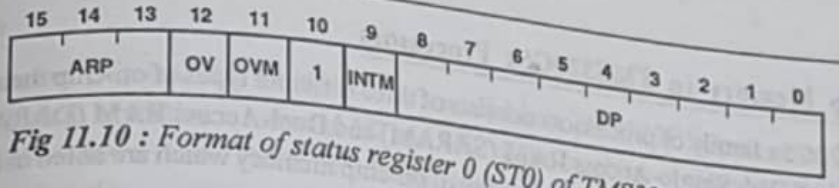


Fig 11.10 : Format of status register 0 (ST0) of TMS320C5x processors.

Table 11.8 : Functions of Various Bits of ST0 of TMS320C5x Processors

BIT	NAME	RESET VALUE	FUNCTION
15-13	ARP	X	Auxiliary register pointer to select AR for indirect addressing
12	OV	0	Overflow flag bit. Indicates an overflow in ALU operation
11	OVM	X	Overflow mode bit. Enables/disables the saturation mode in ALU
10	–	1	Always 1
9	INTM	1	Interrupt mode bit. Globally masks or enables all interrupts except NMI
8-0	DP	X	Data memory page pointer to specify the current data memory page

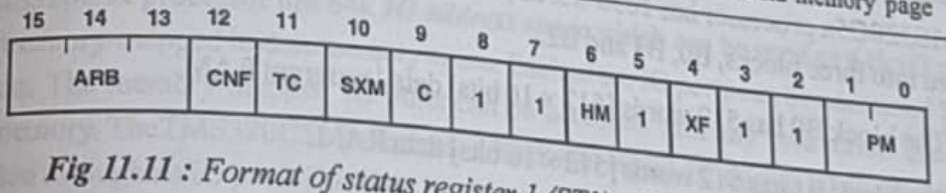


Fig 11.11 : Format of status register 1 (ST1) of TMS320C5x processors.

Table 11.9 : Function of Various Bits of ST1 of TMS320C5x Processors

BIT	NAME	RESET VALUE	FUNCTION
15-13	ARB	XXX	Auxiliary register buffer which holds the previous value in ARP of ST0
12	CNF	0	Configuration control bit to map Block 0 of DARAM either as data or program memory
11	TC	X	Test/control flag bit which stores the results of ALU or PLU test bit operations
10	SXM	1	Sign-extension mode bit. Enables/disables sign extension of an arithmetic operation
9	C	1	Carry bit which indicates a carry or borrow in ALU operation
8-7	–	11	Always 1
6	HM	1	Hold mode bit
5	–	1	Always 1
4	XF	1	Status of external flag pin
3-2	–	11	Always 1
1-0	PM	00	Product shift mode bits

11.3.4 On-Chip Memory in TMS320C5x Processors

The TMS320C5x family of processors consists of three different types of on-chip memory and they are mask-programmable ROM, Single-Access **RAM** (SARAM) and Dual-Access **RAM** (DARAM). The various members of TMS320C5x will have different capacity of on-chip memory which are listed in table 11.4.

Program ROM

The various models of TMS320C5x processors have internal maskable-Program **ROM** (PROM) of size 2k to 32k words. The processor has an option for including or excluding the on-chip PROM addresses in the processor program memory address space.

The main purpose of PROM is to permanently store the program code for a specific application during manufacturing of the chip itself. The processor has an option of boot loading the content of PROM to internal/external RAM during power-ON reset. The content of the PROM can be protected so that any external device cannot have access to the program code. This feature provides security for proprietary algorithms.

Data / Program Dual-Access RAM (DARAM)

The TMS320C5x processor has 1056 words [1056 × 16 bits] on-chip Dual-Access RAM (DARAM), which is divided into three blocks, B0, B1 and B2.

- The block B0 has 512 words [512 × 16 bits] data / program RAM.
- The block B1 has 512 words [512 × 16 bits] data RAM.
- The block B2 has 32 words [32 × 16 bits] data RAM.

Data / Program Single-Access RAM (SARAM)

The various models or TMS320C5x processor has 1k words to 9k words of SARAM.

The internal SARAM can be configured as data memory, program memory and combination of data and program memory

The SARAM can be divided into block of 1k/2k words with continuous address. The processor CPU can access one block for reading while writing in another block.

11.3.5 On-Chip Peripherals of TMS320C5x Processors

The various on-chip peripherals of TMS320C5x processors are clock generator, hardware timer, software-programmable wait-state generators, parallel IO ports, Host Port Interface (HPI) and serial ports.

Clock Generator

The ***clock generator*** of the TMS320C5x processor consists of an internal oscillator and a Phase Locked Loop (PLL) circuit. The clock generator can be driven by an external crystal resonator circuit or supplied by an external clock source. The ***PLL*** circuit can generate an internal CPU clock by multiplying the clock source by a specified factor, so that CPU is driven by high frequency clock and clock source can be used as source for other peripherals which runs at low frequency clock.

Hardware Timer

A 16-bit hardware timer with a 4-bit prescaler is available in TMS320C5x processor. This programmable timer generates clock at a rate that is between 1/2 and 1/32 of the machine cycle rate (CLKOUT1), depending upon the timer divide-down ratio. The timer can be stopped, restarted, reset or disabled by specific status bits. The processor has three registers to control and operate the timer and they are Timer Control Register (TCR), timer counter register (TIM) and timer period register (PRD). The timer counter register gives the current count of the timer. The timer period register defines the period for the timer. The 16-bit timer control register controls the operations of the timer.

Software Programmable Wait - State Generators

The TMS320C5x processor has software-programmable ***wait-state generators***, which can insert/ generate wait-states in external bus cycles for interfacing with slow speed external memory and IO devices. The processor consists of multiple wait-state generating circuits,and each circuit is user-programmable to insert different number of wait states for external memory accesses. These wait-state generators can extend the external bus cycles up to seven machine cycles.

Parallel IO Ports

The TMS320C5x processor has 64k ***IO address space*** which can be used as 64k IO ports and 16 of these ports are memory-mapped in data memory space. Each of the IO ports can be addressed by the IN or the OUT instruction. The memory-mapped IO ports can be accessed with any instruction that reads from or writes to data memory. TheTMS320C5x genearates a hardware signal $\overline{IS}$ during IO access to indicate a read or write operation through an IO port. The TMS320C5x can easily interface with external IO devices through the IO ports with minimal external address decoding circuits.

Host Port interface (HPI)

The HPI is available on the TMS320C57S and TMS320LC57 processors. The ***HPI*** is an 8-bit parallel IO port that provides an interface to a host processor for information exchange between the Digital Signal Processor (DSP) and the host processor. The DSP has 2k word on-chip memory that is accessible to both the host processor and the DSP. The HPI is connected to this memory through a dedicated bus, so that the CPU can work uninterrupted while the host processor accesses the memory through host port.

Note : A host processor is an independent microprocessor/micorcontroller that is designed to carry out some specific tasks and deliver the results to digital signal processor.

Serial Ports

Three different kinds of serial ports are available in TMS320C5x processors and they are general-purpose serial port, Time-Division Multiplexed (TDM) serial port and Buffered Serial Port (BSP). Every TMS320C5x processor contains at least one general-purpose, high-speed synchronous, full-duplexed serial port which can be used to provide direct communication with serial devices such as codec, serial analog-to digital (A/D) converters and other serial systems. The serial port is capable of operating at a clock rate up to one-fourth the machine cycle rate (CLKOUT1). The serial port transmitter and receiver are double-buffered and indiviually controlled by maskable external interrupt signals. For serial communication the data is framed either as bytes or as words.

The Buffered Serial Port (BSP) consists of a full-duplex double-buffered Serial Port Interface (SPI) and an Auto-Buffering Unit (ABU). The ABU allows the SPI to read/write directly to processor internal memory using a dedicated bus which enhances the speed of serial communication.

The Time-Division Multiplexed (TDM) serial ports can be used for serial communication between multiple processors. A maximum of eight processors having TDM ports can be connected via a pair of data lines and a pair of address lines for serial communication.

11.3.6 Addressing Modes of TMS320C5x Processors

The addressing mode is the method of specifying the data to be operated by an instruction. The TMS320C5x family of processors supports the following six addressing modes.

1. Direct addressing
2. Memory-mapped register addressing
3. Indirect addressing
4. Immediate addressing
5. Dedicated-register addressing
6. Circular addressing

Direct Addressing

In direct addressing, the lower 7 bits of data memory address are specified directly in the instruction itself. The upper 9 bits of the address will be the content of data memory page pointer (DP) in status register-0 (ST0).

Example :

```
ADDC 2Ch  ; Add the content of data memory whose page offset address
          ; (2Ch) is specified in the instruction and the carry to accumulator
          ; with sign extension suppressed.
```

Note : The TMS320C5x data memory is organized as 512 (or 2^9) pages with each page having 128 (or 2^7) locations ($2^9 \times 2^7 = 2^{16} = 64k$). The DP holds the 9-bit current page address and the 7-bit address in the instruction is the page offset address.

Memory-Mapped Register Addressing

In memory-mapped register addressing, the address of the memory-mapped register can be specified as direct address in the instruction. The memory-mapped register addressing is a special case of direct addressing in which only page offset address is used to access the memory and the default page address is 000h. Therefore, the data pointer need not be loaded with page address for this addressing mode.

Example :

```
LAMM 16h  ; Load accumulator with the content of memory-mapped
          ; register mapped to address 0016h.
SAMM 16h  ; Store the content of accumulator in memory-mapped
          ; register mapped to address 0016h.
```

Note : The memory-mapped registers of TMS320C5x are mapped to page-0 of data memory address space.

Indirect Addressing

In indirect addressing mode, the data memory address is specified by the content of one of the eight auxiliary registers, AR0 - AR7. The AR (Auxiliary Register) currently used for accessing data is denoted by ARP (Auxiliary Register Pointer).

In indirect addressing mode, the content of AR can be updated automatically either after or before the operand is fetched. The syntax used in the operand field of instruction for modifying the content AR are listed in table 11.10.

Example :

```
LACC *,0     ; Load the content of data memory addressed by AR, (which in turn pointed by ARP) to
             ; accumulator without left shift. AR is not altered.
LACC *+,0    ; Same as above, but AR is incremented by one.
LACC *-,0    ; Same as above, but AR is decremented by one.
LACC *0+,0   ; Same as above, but AR is incremented by the value in index register.
LACC *0-,0   ; Same as above, but AR is decremented by the value in index register.
```

Table 11.10 : Syntax Used in Indirect Address for Modifying AR

Syntax	Modification of AR
*	AR unaltered
*+	AR incremented by 1
*-	AR decremented by 1
*0+	AR incremented by the content of index register
*0-	AR decremented by the content of index register
*BR0+	AR incremented for bit reversed addressing using the content of index register
*BR0-	AR decremented for bit reversed addressing using the content of index register

Bit Reversed Addressing

In bit reversed addressing, the data memory address is specified by AR like indirect addressing, but the content of AR is incremented/decremented in order to generate the data memory address in the bit reversed order, using the content of index register. (The bit reversed addressing is a special case of indirect addressing).

Example :

```
MAC 14F0h, *BR0+ ; The content of program memory is multiplied by the data memory and the product is
                 ; added to the accumulator. The address of the program memory is 14F0h (which is
                 ; specified in the instruction). The address of data memory is the content of AR currently
                 ; pointed by ARP. The AR is incremented to generate the bit reversed address of data
                 ; memory operand.
```

Immediate Addressing

In immediate addressing, the data is specified as a part of the instruction. In this addressing, the instruction will carry an 8-bit/9-bit/13-bit/16-bit constant, which is the data to be operated by the instruction. The immediate constant is specified with # symbol.

Example :

```
ADD # 4Ah     ; Add the immediate, 8-bit data 4Ah given in the instruction to accumulator.
ADD # 8C4Ah   ; Add the immediate 16-bit data 8C4Ah given in the instruction to accumulator.
```

Dedicated Register Addressing

In dedicated register addressing mode, the address of one of the operands is specified by a dedicated CPU register BMAR (**B**lock **M**ove **A**ddress **R**egister). In this addressing mode, the address of the memory block to be accessed can be changed during execution of the program.

In another case of dedicated register addressing, one of the operands is the content of a dedicated CPU register DBMR (**D**ynamic **B**it **M**anupulation **R**egister).

Example :

```
BLDD BMAR, 6Fh ; The source address is the content of BMAR. The lower 7 bits of destination address is
               ; 6Fh, (which is directly given in the instruction) and the upper 9 bits of destination
               ; address is the content of DP. This instruction will copy the content of source address to
               ; destination address.

OPL 6Fh        ; The content of DBMR is logically ORed with the content of data memory specified by the
               ; instruction and result is stored back in data memory. The lower 7 bits of data memory
               ; is 6Fh (which is directly given in the instruction) and the upper 9 bits is the
               ; content of DP.
```

Circular Addressing

The circular addressing is similar to indirect addressing. This addressing mode allows the specified memory buffer to be accessed sequentially with a pointer that automatically wraps around to the beginning of the buffer when the last location is accessed.

In circular addressing mode, when the address pointer is incremented, the address in AR will be checked with the end address of the circular buffer, and if it exceeds the end address then the address is made equal to start address of the circular buffer.

In order to hold the start and end addresses of the circular buffer, the TMS320C5x has four circular buffer registers, namely,

CBSR1 : **C**ircular **B**uffer-1 **S**tart address **R**egister

CBSR2 : **C**ircular **B**uffer-2 **S**tart address **R**egister

CBER1 : **C**ircular **B**uffer-1 **E**nd address **R**egister

CBER2 : **C**ircular **B**uffer-2 **E**nd address **R**egister

With the help of the above registers, at any one time, two circular buffers can be defined. A Circular **B**uffer **C**ontrol **R**egister (CBCR) is used to enable/disable the circular buffers.

11.3.7 Instruction Pipelining in TMS320C5x Processors

The execution of TMS320C5x processor instructions involve four levels/phases of pipelining. The four phases of pipelining are fetch, decode, read and execute. The functions performed in the four phases are given below.

Fetch : The opcode (instruction code) is fetched from program memory and PC is updated.

Decode : The opcode is decoded, the data address is generated and data address generation registers are updated.

Read : The operand is read from data memory.

Execute : Perform the task specified by the instruction.

The TMS320C5x employs a four-phase clock so that one phase of four consecutive instructions can be processed simultaneously. When the first instruction is in execute phase, the second instruction will be in read phase, the third instruction will be in decode phase and the fourth instruction will be in fetch phase. Therefore, the average execution time of one word instruction will be one clock cycle. While executing two word instructions and branch instructions, some of the phases will be idle for some time and so average execution time will be more than one clock cycle.

11.3.8 Instructions of TMS320C5x Processors

The TMS320C5x processors instruction set consists of instructions that supports both numeric-intensive signal processing operations and general-purpose applications. The instructions can be classified into following groups.

1. Arithmetic instructions
2. Logical instructions
3. Branch/control instructions
4. Load/store instructions
5. Block move instructions

The instructions of TMS320C5x processors are classified into the above groups, arranged in alphabetical order and listed in table 11.11.

The size of TMS320C5x instructions is either 1 word or 2 words. When all the instructions and data reside in internal memory, most of the instructions are executed in one or two clock cycles. The execution time for some of the data transfer, branch and MAC instructions is 3 to 4 clock cycles.

Table 11.11 : Instruction Set Summary for TMS320C5x

ARITHMETIC INSTRUCTIONS	
INSTRUCTION	**DESCRIPTION**
ABS	Absolute value of ACC
ADCB	Add ACCB to ACC with carry
ADD *dma [,shift]*	Add memory to ACC with shift (Direct addressing)
ADD *{ind} [,shift [,next ARP]]*	Add memory to ACC with shift (Indirect addressing)
ADD # *k*	Add short immediate to low ACC
ADD # *lk [,shift2]*	Add long immediate to ACC with shift
ADDB	Add ACCB to ACC

Table 11.11 : Continued ....

ARITHMETIC INSTRUCTIONS	
INSTRUCTION	**DESCRIPTION**
ADDC *dma*	Add memory to ACC with carry (Direct addressing)
ADDC *{ind} [,next ARP]*	Add memory to ACC with carry (Indirect addressing)
ADDH *dma*	Add memory to higher 16 bits of accumulator (Direct addressing)
ADDH *{ind} [,next ARP]*	Add memory to higher 16 bits of accumulator (Indirect addressing)
ADDS *dma*	Add memory to ACC with sign extension suppressed (Direct addressing)
ADDS *{ind} [,next ARP]*	Add memory to ACC with sign extension suppressed (Indirect addressing)
ADDT *dma*	Add memory to ACC with shift specified by T-register (Direct addressing)
ADDT *{ind} [,next ARP]*	Add memory to ACC with shift specified by T-register (Indirect addressing)
ADRK *# k*	Add short immediate to AR
APAC	Add PREG to ACC
MAC *pma, dma*	Multiply and accumulate (Direct addressing)
MAC *pma, {ind} [,next ARP]*	Multiply and accumulate (Indirect addressing)
MACD *pma, dma*	Multiply and accumulate with data move (Direct addressing)
MACD *pma, {ind} [,next ARP]*	Multiply and accumulate with data move (Indirect addressing)
MADD *dma*	Multiply and ACC with data move, pma in BMAR (Direct addressing)
MADD *{ind} [, next ARP]*	Multiply and ACC with data move, pma in BMAR (Indirect addressing)
MADS *dma*	Multiply and ACC, pma in BMAR (Direct addressing)
MADS *{ind} [,next ARP]*	Multiply and ACC, pma in BMAR (Indirect addressing)
MPY *dma*	Multiply data in memory with TREG0 (Direct addressing)
MPY *{ind} [,next ARP]*	Multiply data in memory with TREG0 (Indirect addressing)
MPY *# k*	Multiply TREG0 by short immediate
MPY *# lk*	Multiply TREG0 by long immediate
MPYA *dma*	Multiply TREG0 by data, add previous product (Direct addressing)
MPYA *{ind} [,next ARP]*	Multiply TREG0 by data, add previous product (Indirect addressing)
MPYS *dma*	Multiply TREG0 by data, subtract previous product (Direct addressing)
MPYS *{ind} [,next ARP]*	Multiply TREG0 by data, subtract previous product (Indirect addressing)
MPYU *dma*	Unsigned multiplication of memory with TREG0 (Direct addressing)
MPYU *{ind} [,next ARP]*	Unsigned multiplication of memory with TREG0 (Indirect addressing)
NORM	Normalize ACC
SBB	Subtract ACCB from ACC

Table 11.11 : Continued...

INSTRUCTION	DESCRIPTION
SBBB	Subtract ACCB from ACC with borrow
SBRK # k	Subtract short immediate from AR
SPAC	Subtract P-register from ACC
SQRA dma	Square and accumulate previous product (Direct addressing)
SQRA {ind} [,next ARP]	Square and accumulate previous product (Indirect addressing)
SQRS dma	Square and subtract previous product (Direct addressing)
SQRS {ind} [,next ARP]	Square and subtract previous product (Indirect addressing)
SUB dma [,shift]	Subtract memory from ACC with shift (Direct addressing)
SUB {ind} [,shift [,next ARP]]	Subtract memory from ACC with shift (Indirect addressing)
SUB # k	Subtract short immediate from ACC
SUB # lk [,shift 2]	Subtract long immediate from ACC with shift
SUBB dma	Subtract memory from ACC with borrow (Direct addressing)
SUBB {ind} [,next ARP]	Subtract memory from ACC with borrow (Indirect addressing)
SUBC dma	Conditional subtract (Direct addressing)
SUBC {ind} [,next ARP]	Conditional subtract (Indirect addressing)
SUBH dma	Subtract memory from high ACC (Direct addressing)
SUBH {ind} [,next ARP]	Subtract memory from high ACC (Indirect addressing)
SUBS dma	Subtract memory from low ACC with sign extension suppressed (Direct addressing)
SUBS {ind} [,next ARP]	Subtract memory from low ACC with sign extension suppressed (Indirect addressing)
SUBT dma	Subtract memory from ACC with shift specified by TREG1 (Direct addressing)
SUBT {ind} [,next ARP]	Subtract memory from ACC with shift specified by TREG1 (Indirect addressing)
	LOGICAL INSTRUCTIONS
AND dma	AND memory with ACC (Direct addressing)
AND {ind} [,next ARP]	AND memory with ACC (Indirect addressing)
AND # lk [,shift]	AND long immediate with ACC with shift
ANDB	AND ACCB with ACC
APL [# lk], dma	AND memory with DBMR or long constant (Direct addressing)
APL [# lk,] {ind} [,next ARP]	AND memory with DBMR or long constant (Indirect addressing)
BSAR [shift]	Barrel shift ACC right
CMPL	Complement ACC
CMPR CM	Compare AR with AR0 as specified by CM, if true set TC else clear TC

Table 11.11 : Continued...

INSTRUCTION	DESCRIPTION
CPL *[, # lk] dma*	Compare DBMR or long immediate with memory (Direct addressing)
CPL *[, # lk] {ind} [,next ARP]*	Compare DBMR or long immediate with memory (Indirect addressing)
CRGT	Compare ACC and ACCB, store largest in both, if ACC > ACCB set carry
CRLT	Compare ACC and ACCB, store smallest in both, if ACC < ACCB clear carry
NEG	Negate ACC
OPL *[# lk,] dma*	OR memory with DBMR or long constant (Direct addressing)
OPL *[# lk,] {ind} [,next ARP]*	OR memory with DBMR or long constant (Indirect addressing)
OR *dma*	OR memory with ACC (Direct addressing)
OR *{ind} [,next ARP]*	OR memory with ACC (Indirect addressing)
OR *# lk [,shift]*	OR long immediate with ACC with shift
ORB	OR ACCB with ACC
ROL	Rotate ACC left by one bit
ROLB	Rotate ACCB and ACC left by one bit
ROR	Rotate ACC right by one bit
RORB	Rotate ACCB and ACC right by one bit
SATH	Shift ACC 16 bits right if TREG1[4] = 1
SATL	Shift low ACC right as specfied by TREG1 [3-0]
SFL	Shift ACC left by one bit
SFLB	Shift ACCB and ACC left by one bit
SFR	Shift ACC right by one bit
SFRB	Shift ACCB and ACC right by one bit
XOR *dma*	XOR memory with ACC (Direct addressing)
XOR *{ind} [,next ARP]*	XOR memory with ACC (Indirect addressing)
XOR *# lk [,shift]*	XOR long immediate with ACC with shift
XORB	XOR ACCB with ACC
XPL *[# lk] dma*	XOR memory with DBMR or long constant (Direct adressing)
XPL *[# lk,] {ind} [,next ARP]*	XOR memory with DBMR or long constant (Indirect adressing)
BRANCH / CONTROL INSTRUCTIONS	
B[D] *pma [,{ind} [,next ARP]]*	Branch unconditionally to specified pma with AR update and optional delay
BACC[D]	Branch to address specified by ACC with optional delay
BANZ[D] *pma [,{ind} [,next ARP]]*	Branch to pma if content of AR ≠ 0 with AR update and with optional delay

Table 11.11 : Continued...

INSTRUCTION	DESCRIPTION
BBNZ *pma [,{ind} [,next ARP]]*	Branch to specified pma if TC bit of ST1 is not zero with AR update
BBZ *pma*	Branch to specified pma if TC bit of ST1 is zero
BBZ *pma [,{ind} [,next ARP]]*	Branch to specified pma if TC bit of ST1 is zero with AR update
BC *pma*	Branch to specified pma if C bit of ST1 is one
BC *pma [,{ind} [,next ARP]]*	Branch to specified pma if C bit of ST1 is one with AR update
BCND[D] *pma [cond1] [,cond2] [...]*	Branch to pma if specified conditions are true with optional delay
BGEZ *pma*	Branch to specified pma if ACC ≥ 0
BGEZ *pma [, {ind} [, next ARP]]*	Branch to specified pma if ACC ≥ 0 with AR update
BGZ *pma*	Branch to specified pma if ACC > 0
BGZ *pma [, {ind} [, next ARP]]*	Branch to specified pma if ACC > 0 with AR update
BIOZ *pma*	Branch to specified pma if $\overline{BIO} = 0$
BIOZ *pma [, {ind} [, next ARP]]*	Branch to specified pma if $\overline{BIO} = 0$ with AR update
BIT *dma, bit code*	Copy the specified bit of memory to TC bit in ST1 (Direct addressing)
BIT *{ind}, bit code [,next ARP]*	Copy the specified bit of memory to TC bit in ST1 (Indirect addressing)
BITT *dma*	Copy the bit of memory specified by T-register to TC bit in ST1 (Direct addressing)
BITT *{ind} [,next ARP]*	Copy the bit of memory specified by T-register to TC bit in ST1 (Indirect addressing)
BLEZ *pma*	Branch to specified pma if ACC ≤ 0
BLEZ *pma [, {ind} [, next ARP]]*	Branch to specified pma if ACC ≤ 0 with AR update
BLZ *pma*	Branch to specified pma if ACC < 0
BLZ *pma [, {ind} [, next ARP]]*	Branch to specified pma if ACC < 0 with AR update
BNC *pma [, {ind} [, next ARP]]*	Branch to specified pma if C bit in ST1 is zero with AR update
BNV *pma [, {ind} [, next ARP]]*	Branch to specified pma if OV bit in ST0 is zero with AR update
BNZ *pma [, {ind} [, next ARP]]*	Branch to specified pma if ACC ≠ 0 with AR update
BV *pma [, {ind} [, next ARP]]*	Branch to specified pma if OV bit in ST0 is one with AR update
BZ *pma*	Branch to specified pma if ACC = 0
CALA[D]	Call subroutine addressed by ACC with optional delay
CALL[D] *pma [,{ind} [,next ARP]]*	Call subroutine unconditionally with AR update and with optional delay
CC[D] *pma [cond1] [,cond2] [...]*	Call subroutine if specified conditions are true and with optional delay
CLRC *control bit*	Clear the specified control bit
CNFB	Configure on-chip block0 RAM as program memory
CNFD	Configure on-chip block0 RAM as data memory

Table 11.11 : Continued...

INSTRUCTION	DESCRIPTION
DINT	Disable interrupts
EINT	Enable interrupts
IDLE	Idle until interrupt
IDLE2	Idle until interrupt and drive the processor to low-power mode
INTR *k*	Software interrupt with vector number k
NMI	Nonmaskable interrupt (Vector number 15) (Vector address 24h)
NOP	No operation
POP	Pop top of stack to low ACC
POPD *dma*	Pop top of stack to data memory (Direct addressing)
POPD *{ind} [,next ARP]*	Pop top of stack to data memory (Indirect addressing)
PSHD *dma*	Push the content of dma to stack (Direct addressing)
PSHD *{ind} [,next ARP]*	Push the content of dma to stack (Indirect addressing)
PUSH	Push low ACC to stack
RC	Reset carry
RET [D]	Return from subroutine with optional delay
RETC[D] *[cond1] [,cond2] [...]*	Return from subroutine if specified conditions are true with optional delay
RETE	Return from interrupt with interrupt enable
RETI	Return from interrupt
RHM	Reset hold mode
ROVM	Reset overflow mode
RPT *dma*	Repeat next instruction the number times specified by memory (Direct addressing)
RPT *{ind} [,next ARP]*	Repeat next instruction the number times specified by memory (Indirect addressing)
RPT *# k*	Repeat next instruction the number of times specified by short immediate
RPT *#lk*	Repeat next instruction the number of times specified by long immediate
RPTB *pma*	Repeat a block of instructions the number of times specified by BRCR
RPTZ *# lk*	Clear ACC and PREG, repeat next instruction, count specified by long immediate
RSXM	Reset sign-extension mode
RTC	Reset test/control flag
RXF	Reset external flag
SC	Set carry bit
SETC *control bit*	Set the specified control bit

Table 11.11 : Continued...

INSTRUCTION	DESCRIPTION
SHM	Set hold mode
SOVM	Set overflow mode
SPM *2-bit constant*	Set PREG shift mode by copying the specified 2-bit constant to pm field of ST1
SSXM	Set sign-extension mode
STC	Set test/control flag
SXF	Set external flag
TRAP	Software interrupt with vector number 14_{10} (Vector address 22h)
XC *n [,cond1] [,cond2] [...]*	Execute next one or two (n = 1 or 2) instructions if the specified conditions are true
ZAC	The content of ACC is made zero (i.e., cleared)
ZAP	The content of ACC and product register are made zero (i.e., cleared)
ZPR	The content of product register is made zero (i.e., cleared)
	INPUT/OUTPUT INSTRUCTIONS
IN *dma, PA*	Input data from port to the memory specified by dma
IN *{ind}, PA [, next ARP]*	Input data from port to memory specified by indirect addressing and update AR
OUT *dma, PA*	Output data from memory specified by dma to port
OUT *{ind}, PA [, next ARP]*	Output data from memory specified by indirect addressing to port and update AR
TBLR *dma*	Table read (Direct addressing)
TBLR *{ind} [, next ARP]*	Table read (Indirect addressing)
TBLW *dma*	Table write (Direct addressing)
TBLW *{ind} [, next ARP]*	Table write (Indirect addressing)
	LOAD/STORE INSTRUCTIONS
EXAR	Exchange ACCB with ACC
LACB	Load ACCB to ACC
LACC *dma [,shift]*	Load memory to ACC with shift (Direct addressing)
LACC *{ind} [,shift [,next ARP]]*	Load memory to ACC with shift and with AR update (Indirect addressing)
LACC # *lk [,shift2]*	Load long immediate with shift to ACC
LACK *8-bit constant*	Load immediate short constant to accumulator
LACL *dma*	Load from memory to low ACC and clear other bits (Direct addressing)
LACL *{ind} [,next ARP]*	Load from memory to low ACC and clear other bits (Indirect addressing)
LACL # *k*	Load short immediate to low ACC and clear other bits

Table 11.11 : Continued...

INSTRUCTION	DESCRIPTION
LACT *dma*	Load memory to ACC with shift specified by TREG1[3 - 0] (Direct addressing)
LACT *{ind} [,next ARP]*	Load memory to ACC with shift specified by TREG1[3 - 0] (Indirect addressing)
LAMM *dma*	Load the content of memory-mapped register to low ACC (Direct addressing)
LAMM *{ind} [,next ARP]*	Load the content of memory-mapped register to low ACC (Indirect addressing)
LAR *AR, dma*	Load memory to AR (Direct addressing)
LAR *AR, {ind} [,next ARP]*	Load memory to AR (Indirect addressing)
LAR *AR, # k*	Load specified short immediate to AR
LAR *AR, # lk*	Load specified long immediate to AR
LDP *dma*	Load memory to data page pointer register (Direct addressing)
LDP *{ind} [,next ARP]*	Load memory to data page pointer register (Indirect addressing)
LDP *# k*	Load specified short immediate to data page pointer register
LMMR *dma, # lk*	Load MMR addressed by dma to memory specified by long immediate
LMMR *{ind} [,next ARP]*	Load indirectly addressed MMR to memory specified by long immediate
LPH *dma*	Load data of directly addressed memory to high PREG
LPH *{ind} [,next ARP]*	Load data of indirectly addressed memory to high PREG
LST *#n dma*	Load memory to status register n (Direct addressing)
LST *#n, {ind} [,next ARP]*	Load memory to status register n (Indirect addressing)
LST1 *dma*	Load data of directly addressed memory to ST1
LST1 *{ind} [,next ARP]*	Load data of indirectly addressed memory to ST1
LT *dma*	Load data of directly addressed memory to TREG0
LT *{ind} [,next ARP]*	Load data of indirectly addressed memory to TREG0
LTA *dma*	Load memory to TREG0, accumulate previous product (Direct addressing)
LTA *{ind} [,next ARP]*	Load memory to TREG0, accumulate previous product (Indirect addressing)
LTD *dma*	Load TREG0, accumulate previous product and move data (Direct addressing)
LTD *{ind} [,next ARP]*	Load TREG0, accumulate previous product and move data (Indirect addressing)
LTP *dma*	Load memory to TREG0 and load PREG to ACC (Direct addressing)
LTP *{ind} [,next ARP]*	Load memory to TREG0 and load PREG to ACC (Indirect addressing)
LTS *dma*	Load memory to TREG0, subtract PREG from ACC (Direct addressing)

Table 11.11 : Continued...

INSTRUCTION	DESCRIPTION
LTS *{ind} [,next ARP]*	Load memory to TREG0, subtract PREG from ACC (Indirect addressing)
MAR *dma*	Modify auxiliary register (Direct addressing)
MAR *{ind} [,next ARP]*	Modify auxiliary register (Indirect addressing)
PAC	Load product register to ACC
SACB	Store ACC in ACCB
SACH *dma [,shift]*	Store high ACC with shift in memory (Direct addressing)
SACH *{ind} [,shift [,next ARP]]*	Store high ACC with shift in memory (Indirect addressing)
SACL *dma [,shift]*	Store low ACC with shift in memory (Direct addressing)
SACL *{ind} [,shift [,next ARP]]*	Store low ACC with shift in memory (Indirect addressing)
SAMM *dma*	Store ACCL in memory-mapped register (Direct addressing)
SAMM *{ind} [,next ARP]*	Store ACCL in memory-mapped register (Indirect addressing)
SAR *AR, dma*	Store AR in memory (Direct addressing)
SAR *AR, {ind} [,next ARP]*	Store AR in memory (Indirect addressing)
SMMR *dma, # lk*	Store MMR addressed by dma in memory specified by long immediate
SMMR *{ind}, # lk [,next ARP]*	Store indirectly addressed MMR in memory specified by long immediate
SPLK *# lk, dma*	Store long immediate in memory (Direct addressing)
SPLK *# lk, {ind} [,next ARP]*	Store long immediate in memory (Indirect addressing)
SPH *dma*	Store high product register in memory (Direct addressing)
SPH *{ind} [,next ARP]*	Store high product register in memory (Indirect addressing)
SPL *dma*	Store low product register in memory (Direct addressing)
SPL *{ind} [,next ARP]*	Store low product register in memory (Indirect addressing)
SST *# n dma*	Store status register n in memory (Direct addressing)
SST *# n {ind} [,next ARP]*	Store status register n in memory (Indirect addressing)
SST1 *dma*	Store status register 1 in memory (Direct addressing)
SST1 *{ind} [,next ARP]*	Store status register 1 in memory (Indirect addressing)
ZALH *dma*	Zero low ACC and load high ACC from memory (Direct addressing)
ZALH *{ind} [,next ARP]*	Zero low ACC and load high ACC from memory (Indirect addressing)
ZALR *dma*	Zero low ACC, load memory to high ACC with rounding (Direct addressing)
ZALR *{ind} [,next ARP]*	Zero low ACC, load memory to high ACC with rounding (Indirect addressing)

Table 11.11 : Continued...

INSTRUCTION	DESCRIPTION
ZALS *dma*	Zero ACC, load memory to low ACC without sign extension (Direct addressing)
ZALS *{ind} [,next ARP]*	Zero ACC, load memory to low ACC without sign extension (Indirect addressing)
	BLOCK MOVE INSTRUCTIONS
BLDD *# lk, dma*	Block move from directly addressed memory to immediate addressed memory
BLDD *# lk, {ind} [,next ARP]*	Block move from indirectly addressed memory to immediate addressed memory
BLDD *dma, # lk*	Block move from immediate addressed memory to directly addressed memory
BLDD *{ind}, # lk [,next ARP]*	Block move from immediate addressed memory to indirectly addressed memory
BLDD BMAR, DMA	Block move from directly addressed memory to memory addressed by BMAR
BLDD BMAR, *{ind} [,next ARP]*	Block move from indirectly addressed memory to memory addressed by BMAR
BLDD *dma BMAR*	Block move from memory addressed by BMAR to directly addressed memory
BLDD *{ind}, BMAR [,next ARP]*	Block move from memory addressed by BMAR to indirectly addressed memory
BLDP *dma*	Block move from data memory to program memory (Direct addressing)
BLDP *{ind} [,next ARP]*	Block move from data memory to program memory (Indirect addressing)
BLPD *# pma, dma*	Block move from program to data memory (Direct addressing)
BLPD *# pma, {ind} [,next ARP]*	Block move from program to data memory (Indirect addressing)
BLPD BMAR, *{ind} [,next ARP]*	Block move from program to data memory, pma in BMAR (Direct addressing)
BLPD BMAR, *dma*	Block move from program to data memory, pma in BMAR (Indirect addressing)
DMOV *dma*	Move the content of memory to next higher location (Direct addressing)
DMOV *{ind} [,next ARP]*	Move the content of memory to next higher location (Indirect addressing)

Table 11.12 : Symbols and Acronyms Used in the Instruction Set Summary

SYMBOL	DESCRIPTION
ACC	Accumulator
ACCB	Accumulator buffer
AR	Auxiliary register
ARCR	Auxiliary register compare
ARP	Auxiliary register pointer
BMAR	Block move address register
bit code	Bit to test
2-bit constant	2-bit short immediate value

SYMBOL	DESCRIPTION
8-bit constant	8-bit short immediate value
BRCR	Block repeat count register
C	Carry bit
CM	Compare mode (00/01/10/11)
control bit	Bit to be cleared/set
[D]	Optional delay
DBMR	Dynamic bit manipulation register
dma	Data memory address

Table 11.12 : Continued ....

SYMBOL	DESCRIPTION
DP	Data memory page pointer
{ind}	Indirect address *, *+, *–, *0+, *0–, *BR0+, *BR0– (Refer table 11.8)
k	Interrupt vector number
# k	8-bit immediate value (0 to FFh)
# lk	16-bit immediate value (0 to FFFFh)
# n	Register number (0 or 1)
[,next ARP]	Update ARP for next instruction
OV	Overflow bit
PA	Port address

SYMBOL	DESCRIPTION
PC	Program counter
PC	Program counter
pma	Program memory address (0 to FFFFh)
PREG	Product register
shift	Shift of 0-16 bits
shift 2	Shift of 0-15 bits
ST	Status register
T	Temporary register
TC	Test/control bit
TREGn	Temporary register (0-2)
TRFGn	Temporary register (0-2)

Table 11.13 : Conditions for Branch, Call and Return Instructions

Code for Cond1 Cond2, etc.,	Condition	Description
BIO	$\overline{BIO} = 0$	$\overline{BIO}$ signal is low
C	C = 1	Carry bit set
EQ	ACC = 0	Accumulator equal to 0
GEQ	ACC ≥ 0	Accumulator greater than or equal to 0
GT	ACC > 0	Accumulator greater than 0
LEQ	ACC ≤ 0	Accumulator less than or equal to 0
LT	ACC < 0	Accumulator less than 0
NC	C = 0	Carry bit cleared
NEQ	ACC ≠ 0	Accumulator not equal to 0
NOV	OV = 0	No accumulator overflow detected
NTC	TC = 0	Test/control flag cleared
OV	OV = 1	Accumulator overflow detected
TC	TC = 1	Test/control flag set
UNC	None	Unconditional operation

A valid decimal (BCD) constant/number is framed using numeric characters 0 to 9, and no alphabet is placed at the end.

A valid hexadecimal constant/number is framed using numeric characters 0 to 9 and alphabets a to f, and the alphabet h is placed at the end. A zero should be placed/inserted at the beginning of a hexadecimal number if the first digit is an alphabet character from a to f, otherwise the assembler will consider the constant starting with a to f as a symbol.

Examples of valid constant

1011 - Decimal (BCD) constant

1101b - Binary constant

92ach - Hexadecimal constant

0e2h - Hexadecimal constant

Symbols

The ***symbols*** are variables (or terms) used in assembly language program statements in order to represent the variable data and address. While running a program, a value has to be attached to each symbol in the program. The advantage of using a symbol is that the value of the symbol can be dynamically varied while running the program.

Usually a symbol name is constructed such that it reflects the meaning of the value it holds. A variable name selected to represent the temperature of a device can be TEMP, a variable name selected to represent the speed of a motor can be M_SPEED, etc.

The guidelines for framing the symbols are given below:

- The symbol name can be constructed using A to Z, a to z, 0 to 9, $, _ (underscore).
- A number cannot be the first character in the symbol name.
- The maximum length of a symbol name is 32 characters.
- The symbol name is case sensitive.

Assembler Directives

The ***assembler directives*** are the instructions to the assembler regarding the program being assembled. They are also called ***pseudo instructions***. The assembler directives are used to specify start and end of a program, attach value to variables, allocate storage locations to input/output data, to define start and end of segments, procedures, macros, etc.

The assembler directives control the generation of machine code and organization of the program. But no machine codes are generated for assembler directives. Some of the assembler directives that can be used for TMS320C5x assembly language program development are listed in table 11.14.

11.3.9 Assembly Language Programs in TMS320C5x

The assembly language programs for TMS320C5x processors are written using the mnemonics listed in table 11.11. The processor can execute only machine language programs and the conversion from assembly language to machine language is performed by a software tool called assembler.

Texas Instruments has released a number of assembly language program-development tools for their digital signal processors. Some of the tools are assembler, linker, absolute lister, hex-converter and loader.

The ***assembler*** is a software tool that can run on any standard PC (Personal Computer) and permits to type, edit and convert the assembly language program to machine language object file called COFF (Common Object File Format) file.The process of conversion from assembly language program to machine language program is called ***assembling***. The assembly language program can be developed in modules and each module can be assembled separately to generate COFF files and they can be combined to a single executable object file using a linker.

The ***absolute lister*** will map the executable object file of the program to a specific memory location of the system. The ***loader*** is used to download the executable object file into the processor RAM for execution by the processor. When the program has to be permanently stored in ROM/EPROM, the object files have to be converted to standard hex files. The ***hex-converter*** can be used to convert an executable object file to standard hex-file. Using any standard EPROM programmer, the hex-file can be loaded in EPROM for execution by the processor.

Assembly Language Program Statement Format

The assembly program for TMS320C5x processor consists of statements that contain labels, assembler directives, instructions, macro directives and comments. Each statement can have a maximum of 200 characters. A statement may have four fields, namely label, mnemonic, operands and comment. The general format of the statement is shown below.

[label] [:] mnemonic [operands] [;comment]

The guidelines for writing statements are given below.

- A statement must begin with a label, a blank, an asterisk or a semicolon.
- A label should begin in the first column and the label is optional.
- A comment should begin with a semicolon and the comment is optional.
- Each field should be separated by one or more blanks.

Constants

The decimal, binary or hexadecimal numbers used to represent the data or address in assembly language program statement are called ***constants*** or numerical constants. When constants are used to represent the address/data then their values are fixed and cannot be changed while running a program. The binary and hexadecimal constants can be differentiated by placing a specific alphabet at the end of the constant.

A valid binary constant/number is framed using the numeric characters 0 and 1, and the alphabet b is placed at the end.

Table 11.14 : Assembler Directives Summary

ASSEMBLER DIRECTIVE	DESCRIPTION
.align	Align the SPC (Section Program Counter) on a page boundary
.data [value]	Assemble into default or specified data memory
.ds xxxx	Assemble into data memory address xxxx
.end	End program
.endloop	End .loop code block
.entry	Initialize the starting of PC
.even	Align the SPC (Section Program Counter) on an even word boundary
.equ	Equate a value with a symbol
.int *value1 [, ... , value n]*	Initialize one or more 16-bit integers
.loop *[well-defined expression]*	Begin repeatable assembly of a code block
.mmregs	Enter memory-mapped registers into symbol table
.ps xxxx	Assemble into program memory address xxxx
.set	Equate a value with a symbol
.text [value]	Assemble into default or specified program memory
.word *value1 [, ... , value n]*	Initialize one or more 16-bit binary

Program 11.1

Write an assembly language program using instructions of TMS320C5x processors to add two numbers of 64-bit data. Assume that the two data are available in memory. Store the sum in memory.

Problem Analysis

The memory word size of the TMS320C5x processor is 16 bits and so each 64-bit data is stored as 4 words ($4 \times 16 = 64$). Let us use direct addressing. Let the address of 4 words of data-1 be named as AD1W1, AD1W2, AD1W3 and AD1W4. Let the address of 4 words of data-2 be named as AD2W1, AD2W2, AD2W3 and AD2W4. Let the address of 4 words of sum be named as ASW1, ASW2, ASW3 and ASW4.

Let us load the lower two words of data-1 in the 32-bit accumulator and then the word-1 of data-2 is added to the low accumulator, and the word-2 of data-2 is added to the high accumulator. The 32-bit sum in the accumulator is stored in the memory. The addition of the next two words are performed in a similar manner by considering the carry in the previous addition.

Assembly language program

```
;PROGRAM TO ADD TWO NUMBERS OF 64-BIT DATA
AD1W1   .set 00h          ;Offset address for data-1.
AD1W2   .set 01h
AD1W3   .set 02h
AD1W4   .set 03h
AD2W1   .set 04h          ;Offset address for data-2.
```

```
AD2w2   .set 05h
AD2w3   .set 06h
AD2w4   .set 07h        ;offset address for sum.
ASW1    .set 08h
ASW2    .set 09h
ASW3    .set 0ah
ASW4    .set 0bh        ;Include memory-mapped registers.
        .mmregs         ;Origin of program address is 0b00h.
        .ps   0b00h     ;Initialize program counter with starting address of program.
        .entry          ;Clear sign extension mode bit.
AD64:   CLRC SXM        ;Load word-2 of data-1 in high accumulator.
        LACC AD1W2,16   ;Load word-1 of data-1 with sign extension suppressed in low accumulator.
        ADDS AD1W1      ;Add word-1 of data-2 to low accumulator.
        ADDS AD2W1      ;Add word-2 of data-2 to high accumulator.
        ADD  AD2W2,16   ;Store word-1 of sum in memory.
        SACL ASW1       ;Store word-2 of sum in memory.
        SACH ASW2       ;Load word-4 of data-1 in high accumulator.
        LACC AD1W4,16   ;Add word-3 of data-1 and carry to accumulator.
        ADDC AD1W3      ;Add word-3 of data-2 with sign extension suppressed to low accumulator.
        ADDS AD2W3      ;Add word-4 of data-2 to high accumulator.
        ADD  AD2W4,16   ;Store word-3 of sum in memory.
        SACL ASW3       ;Store word-4 of sum in memory.
        SACH ASW4       ;Program end.
        RET             ;Assembly end.
        .end
```

Program 11.2

Write an assembly language program using instructions of TMS320C5x processors to subtract two numbers of 64-bit data. Assume that the two data are available in memory. Store the result in memory.

Problem Analysis

The memory word size of the TMS320C5x processor is 16 bits and so each 64-bit data is stored as 4 words ($4 \times 16 = 64$). Let 4 words of data-1 be stored in memory at address 1101h to 1104h. Let 4 words of data-2 be stored in memory at address 1111h to 1114h. Let the 4 words of result be stored in memory at address 1121h to 1124h. Let us use indirect address using auxiliary registers.

Let us load the lower two words of data-1 in 32-bit accumulator and then the word-1 of data-2 is subtracted from the low accumulator, and the word-2 of data-2 is subtracted from the high accumulator. The 32-bit result in the accumulator is stored in memory. The subtraction of next two words are performed in a similar manner by considering the borrow in the previous subtraction.

Assembly language program

```
;PROGRAM TO SUBTRACT TWO NUMBERS OF 64-BIT DATA
        .mmregs             ;Include memory-mapped registers.
        .ps   0C00h         ;Origin of program address is 0C00h.
        .entry              ;Initialize program counter with starting address of program.
Ini_ARs: LAR  AR1,#1101h    ;Load starting address of data-1 in AR1.
        LAR  AR2,#1111h     ;Load starting address of data-2 in AR2.
        LAR  AR3,#1121h     ;Load starting address of sum in AR3.
SUB64:  CLRC SXM            ;Clear sign extension mode bit.
        LACC *+,0,AR1       ;Load word-1 of data-1 in low accumulator.
        ADD  *+,16,AR1      ;Load word-2 of data-1 in high accumulator.
        SUBS *+,0,AR2       ;Subtract word-1 of data-2 with sign extension suppressed from low,
                            ;accumulator.
```

$FPW1 = P1W1$

$FPW2 = P1W2 + P2W1 + P3W1$

$FPW3 = P2W2 + P3W2 + P4W1$

$FPW4 = P4W2$

Assembly language program

```
;PROGRAM TO MULTIPLY TWO NUMBERS OF 32-BIT DATA
AD1W1   .set 10h        ;offset address for data-1.
AD1W2   .set 11h        ;offset address for data-2.
AD2W1   .set 12h
AD2W2   .set 13h        ;offset address for product.
AFPW1   .set 14h
AFPW2   .set 15h
AFPW3   .set 16h
AFPW4   .set 17h
        .mmregs         ;Include memory-mapped registers.
        .ps   od00h     ;Origin of program address is 0d00h.
        .entry          ;Initialize program counter with starting address of program.
        CLRC SXM        ;Clear sign extension mode bit.
MUL32:  LT   AD2W1      ;Load D2W1 in TREG0.
        MPYU AD1W1      ;Multiply D1W1 and D2W1 to get product-1 (P1) in P-register.
        SPL  AFPW1      ;Store FPW1 in memory.
        SPH  AFPW2      ;Store partial FPW2 in memory.
        MPYU AD1W2      ;Multiply D1W2 and D2W1 to get product-2 (P2) in P-register.
        LTP  AD2W2      ;Save P2 in accumulator. Load D2W2 in TREG0.
        MPYU AD1W1      ;Multiply D1W1 and D2W2 to get product-3 (P3) in P-register.
        MPYA AD1W2      ;Get sum of P2 and P3 in accumulator. Multiply D1W2 and D2W2 to get,
                        ;product-4 (P4) in P-register.
        ADDS AFPW2      ;Add partial FPW2 in memory to low accumulator to get FPW2 in low,
                        ;accumulator.
        SACL AFPW2      ;Save FPW2 in memory.
        BSAR 16         ;Shift accumulator right by 16.
        APAC            ;Add P4 in P-register to accumulator to get FPW3 and FPW4 in low and,
                        ;high accumulator respectively.
        SACL AFPW3      ;Save FPW3 in memory.
        SACH AFPW4      ;Save FPW4 in memory.
        RET             ;Program end.
        .end            ;Assembly end.
```

Program 11.4

Write an assembly language program using instructions of TMS320C5x processors to divide a 16-bit data by an 8-bit data. Assume that the data are 2's complement positive integers available in memory. Store the quotient and remainder in memory.

Problem Analysis

The TMS320C5x processors have a special instruction "SUBC dma" to implement division algorithms. For 16-bit by 8-bit division of positive integers, the 16-bit dividend is stored in the low accumulator and the high accumulator is filled with zero. The 8-bit divisor is stored in data memory as lower 8 bits and upper 8 bits are filled with zero. Then the "SUBC dma" instruction is executed 16 times to generate the quotient in the low accumulator and remainder in the high accumulator.

```
SUB  *+,16,AR2   ;Subtract word-2 of data-2 from high accumulator.
SACL *+,0,AR3    ;Store word-1 of result in memory.
SACH *+,0,AR3    ;Store word-2 of result in memory.
LACC *+,0,AR1    ;Load word-3 of data-1 in low accumulator.
SUBB *+,0,AR2    ;Subtract word-3 of data-2 and previous borrow from low accumulator
ADD  *+,16,AR1   ;Load word-4 of data-1 in low accumulator.
SUB  *+,16,AR2   ;Subtract word-4 of data-2 from high accumulator.
SACL *+,0,AR3    ;Store word-3 of result in memory.
SACH *+,0,AR3    ;Store word-4 of result in memory.
RET              ;Program end.
.end             ;Assembly end.
```

Program 11.3

Write an assembly language program using instructions of TMS320C5x processors to multiply two numbers of unsig 32-bit data. Assume that the two data are available in memory. Save the 64-bit product in memory.

Problem Analysis

In the TMS320C5x processor, the 32-bit multiplication can be implemented in terms of 16 multiplication. The given 32-bit data can be divided into two words (16-bit data) as shown below.

Data-1 (D1) → Data-1 word-2 (D1W2), Data-1 word-1 (D1W1)

Data-2 (D2) → Data-2 word-2 (D2W2), Data-2 word-1 (D2W1)

Using the above four words (D1W1, D1W2, D2W1, D2W2), the following four products are comp Each product will generate a 32-bit result and so they are divided into two words (16-bit) as shown bel

Product-1 (P1) : $D1W1 \times D2W1 = P1 \rightarrow P1W2, P1W1$

Product-2 (P2) : $D1W2 \times D2W1 = P2 \rightarrow P2W2, P2W1$

Product-3 (P3) : $D1W1 \times D2W2 = P3 \rightarrow P3W2, P3W1$

Product-4 (P4) : $D1W2 \times D2W2 = P4 \rightarrow P4W2, P4W1$

The results of the above four products can be added to get the final result as shown belov maximum size of Final Product (FP) will be 64 bits and so it is divided into four words, namely FPW1, I FPW3 and FPW4.

			D1W2	D1W1
			× D2W2	D2W1
			P1W2	P1W1
		P2W2	P2W1	
		P3W2	P3W1	
	P4W2	P4W1		
FPW4	FPW3	FPW2	FPW1	

Let us use direct addressing for data and product. The address of data words and final produc are named as AD1W1, AD1W2, AD2W1, AD2W2, AFPW1, AFPW2, AFPW3, AFPW4.

The products are performed one by one by loading one of the data words in T-register-0 (TRE multiplying with a memory data word to get the 32-bit product in P-register. The results of the pro added as shown below to get the final product words.

Note : The SUBC instruction will left shift the divisor by 15 bits and subtract from dividend and store the result in the accumulator. Then if result is positive, the accumulator is multiplied by 2 and 1 is added, else the accumulator is simply multiplied by 2 (and 1 is not added).

Let us use direct addressing. Let the address of dividend and divisor be ADIVD and ADIVS. Let the address of quotient and remainder be AQUO and AREM.

Assembly language program

```
;PROGRAM FOR 16-BIT BY 8-BIT DIVISION OF POSITIVE DATA
ADIVD    .set 30h       ;Offset address of dividend.
ADIVS    .set 31h       ;Offset address of divisor.
AQUO     .set 32h       ;Offset address of quotient.
AREM     .set 33h       ;Offset address of remainder.
         .mmregs        ;Include memory-mapped registers.
         .ps  0e00h     ;Origin of program address is 0e00h.
         .entry         ;Initialize program counter with starting address of program.
DIV16_8: CLRC SXM       ;Clear sign extension mode bit.
         LACC ADIVD     ;Load dividend in low accumulator with high accumulator as zero.
         RPT  #15       ;Repeat the next instruction 16 times. Count is 1 less than number of,
                        ;repeatitions.
         SUBC ADIVS     ;Perform conditional subtraction of division algorithm.
         SACL AQUO      ;Store the quotient in memory.
         SACH AREM      ;Store the remainder in memory.
         RET            ;Program end.
         .end           ;Assembly end.
```

Program 11.5

Write an assembly language program using instructions of TMS320C5x processor to find the sum of an array stored in memory. Assume that the array has 10_{10} data each of size 16 bits and store the sum in memory.

Problem Analysis

The array of 10 data can be stored in memory with starting address named as "Arr_addr". The 32-bit accumulator is cleared to have an initial sum as zero. Then the data of the array can be added one by one to the accumulator. The final sum in the 32-bit accumulator is stored as two words in memory with address named as Asum_W1 and Asum_W2.

Assembly language program

```
;PROGRAM FOR SUM OF AN ARRAY
Arr_addr .set 1200h          ;Data array starting address is 1200h.
Cnt_val  .set 09h            ;Count value is one less than number of data in the array.
Asum_W1  .set 1300h          ;Address of sum.
Asum_W2  .set 1301h
         .mmregs             ;Include memory-mapped registers.
         .ps  0f00h          ;Origin of program address is 0f00h.
         .entry              ;Initialize program counter with starting address of program.
Sum_arr: CLRC SXM            ;Clear sign extension mode bit.
         LAR  AR0,Cnt_val    ;Load count value in AR0.
         LAR  AR1,Arr_addr   ;Load starting address of array in AR1.
```

```
        ZAP             ;Clear accumulator.
        MAR   *,AR1     ;Make ARP to point AR1 as current AR.
        ADDS  *+,0,AR0  ;Add content of AR1 (without shift and sign extension suppressed),
AGAIN:                  ;to accumulator, increment AR1 and make ARP to point AR0 as next AR.
        BANZ AGAIN,AR1  ;If AR0 is not zero, then decrement AR0 by 1, branch to program,
                        ;memory address AGAIN, make ARP to point AR1 as next AR. If AR0 is,
                        ;zero then go to next instruction.
        SACL Asum_W1    ;Store word-1 of sum in memory.
        SCAH Asum_W2    ;Store word-2 of sum in memory.
        RET             ;Program end.
        .end            ;Assembly end.
```

Program 11.6

The input x(n) and impulse response h(n) of an LTI system is given by,

$$x(n) = \{2, 10, 13, 5\}$$

$$h(n) = \{7, 14, 4\}$$

The response y(n) of the LTI system is given by,

$$y(n) = x(n) * h(n) \ ; \text{ where "*" symbol for convolution.}$$

Write an assembly language program using instructions of TMS320C5x processors to determine the response of LTI system via convolution of input and impulse response.

Problem Analysis

The x(n) is a 4-point sequence and h(n) is a 3-point sequence. The convolution of x(n) and h(n) will produce a sequence of size 4 + 3 – 1 = 6-point sequence. Let us append zero to x(n) and h(n) to convert them to 6-point sequences as shown below and perform linear convolution via circular convolution.

$$\therefore x(n) = \{2, 10, 13, 5, 0, 0\}$$

$$h(n) = \{7, 14, 4, 0, 0, 0\}$$

The circular convolution of x(n) and h(n) is defined as,

$$y(n) = \sum_{m=0}^{N-1} x(m)\, h\big((n-m)\big)_N$$

When N = 6,

$$y(n) = \sum_{m=0}^{5} x(m)\, h\big((n-m)\big)_6$$

$$= x(0)\, h\big((n-0)\big)_6 + x(1)\, h\big((n-1)\big)_6 + x(2)\, h\big((n-2)\big)_6 + x(3)\, h\big((n-3)\big)_6 + x(4)\, h\big((n-4)\big)_6 + x(5)\, h\big((n-5)\big)_6$$

When n = 0 ; $y(0) = x(0)\, h(0) + x(1)\, h((-1))_6 + x(2)\, h((-2))_6 + x(3)\, h((-3))_6 + x(4)\, h((-4))_6 + x(5)\, h((-5))_6$

$$\therefore y(0) = x(0)\, h(0) + x(1)\, h(6-1) + x(2)\, h(6-2) + x(3)\, h(6-3) + x(4)\, h(6-4) + x(5)\, h(6-5)$$

$$\therefore y(0) = x(0)\, h(0) + x(1)\, h(5) + x(2)\, h(4) + x(3)\, h(3) + x(4)\, h(2) + x(5)\, h(1)$$

Similarly,

When n = 1 ; $y(1) = x(0)h(1) + x(1)h(0) + x(2)h(5) + x(3)h(4) + x(4)h(3) + x(5)h(2)$

When n = 2 ; $y(2) = x(0)h(2) + x(1)h(1) + x(2)h(0) + x(3)h(5) + x(4)h(4) + x(5)h(3)$

When n = 3 ; $y(3) = x(0)h(3) + x(1)h(2) + x(2)h(1) + x(3)h(0) + x(4)h(5) + x(5)h(4)$

When n = 4 ; $y(4) = x(0)h(4) + x(1)h(3) + x(2)h(2) + x(3)h(1) + x(4)h(0) + x(5)h(5)$

When n = 5 ; $y(5) = x(0)h(5) + x(1)h(4) + x(2)h(3) + x(3)h(2) + x(4)h(1) + x(5)h(0)$

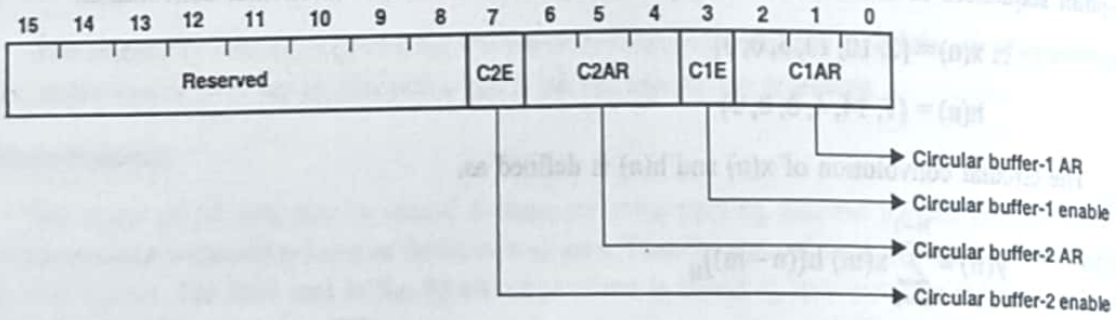
The computation of the above equations involves multiplication and addition, and so can be computed in TMS320C5x processors using RPT and MAC instructions. Let us store input array in program memory and impulse array in data memory. The data memory used to store impulse array can be declared as a circular memory.

The repeat count register is loaded with a count value of 5 for executing MAC instruction 6 times, to compute one data of the output sequence. The output data is stored in memory and the loading of count value and executing MAC instruction are repeated 5 more times to compute the next 5 data of output sequence.

In this computation process, the program memory address has to be incremented and circular data memory address has to be decremented.

In order to enable the circular memory and to specify the auxiliary register to be used for addressing circular memory, an appropriate word (called control word) should be loaded in CBCR (Circular Buffer Control Register).

The format of CBCR is shown below.



Note : 1 = Enable, 0 = Disable.

Let us use circular buffer-1 and use AR0 to address buffer-1, and so the control word to be loaded in CBCR is 08h (0 000 1 000).

Assembly language program

```
;PROGRAM FOR CONVOLUTION
      .mmregs              ;Include memory-mapped registers.
      .ps   0c00h          ;Origin of program address is 0c00h.
      .word 02h, 0Ah, 0dh  ;Store the input array in program.
      .word 05h, 0h, 0h    ;Memory address starting from 0c00h.
      .ds   1100h          ;Origin of data address is 1100h.
```

```
        .word 07h, 0eh, 04h     ;Store the impulse array in data.
        .word 0h, 0h, 0h        ;memory address starting from 1100h.
        .entry                  ;Initialize program counter with starting address of program.
CONV:   SPLK #1100h,CBSR1       ;Load start address of circular buffer-1.
        SPLK #1107h,CBER1       ;Load end address of circular buffer-1.
        SPLK #08h               ;Enable circular buffer-1 .set AR0 as pointer for circular,
                                ;buffer-1.
        LAR  AR0,#1100h         ;Initialize AR0 with start address of impulse array.
        LAR  AR1,#1200h         ;Initialize AR1 with start address of output array.
        LAR  AR2,#05h           ;Initialize AR2 as count for number of data in output array, count,
                                ;is 1 less than number of data.
        ZAP                     ;Clear accumulator and P-register.
LOOP:   MAR  *,AR0              ;Make ARP to point AR0 as current AR.
        RPT  #05h               ;Repeat next instruction 6 times. Count is 1 less than number of,
                                ;repeatitions.
        MAC  0c00h,*-,AR0       ;Add P-register to accumulator, load a word of input array,
                                ;addressed by program memory in T-registers, multiply,
                                ;T-register and a word of impulse array in data memory,
                                ;addressed by AR0, increment program memory address, decrement,
                                ;data memory address.
        MAR  *,AR1              ;Make ARP to point AR1 as current AR.
        SACL *+,AR2             ;Store one word of output array in memory, increment AR1, make ARP,
                                ;to point AR2 as current AR.
        BANZ LOOP               ;If AR2 is not zero, then decrement AR2 by 1, branch to program,
                                ;memory address LOOP. If AR2 is zero, then go to next instruction.
        RET                     ;Program end.
        .end                    ;Assembly end.
```

11.4 TMS320C54x Family of Digital Signal Processors

The TMS320C54x family of processors are advanced versions of TMS320C5x from Texas Instruments, USA. These processors are built with modified Harvard architecture with more internal buses and on-chip peripherals, larger size ALU and very rich instruction set than the TMS320C5x family. The TMS320C54x family of processors can execute 40 to 120 Million Instructions Per-Second (MIPS).

Some of the features of the TMS320C54x family of digital signal processors are,

- 16-bit CPU
- 25 to 8.3 ns single cycle instruction execution time
- Single cycle 17 × 17-bit MAC (Multiply/Accumulate) unit
- 8M × 16-bit virtual program memory address space
- 64k × 16-bit physical program memory address space
- 64k × 16-bit external data memory address space
- 64k × 16-bit external IO address space
- 2k to 48k × 16-bit on-chip program/data ROM
- 5k to 32k × 16-bit on-chip program/data RAM