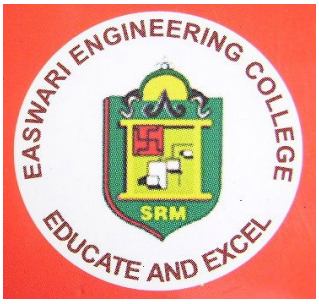
# EASWARI ENGINEERING COLLEGE
## Bharathi salai, Ramapuram
## Chennai - 89




## DEPARTMENT OF ECE

## EI 2355 – DIGITAL SIGNAL PROCESSING LAB

**NAME** :_____

**ROLL NO** :_____

**YEAR/SEM** :_____

**BRANCH :**_____

# EASWARI ENGINEERING COLLEGE
# RAMAPURAM CHENNAI-89
# DEPARTMENT OF E.C.E

## III- YEAR V SEM- ECE

## EC1306 –DSP LAB MANUAL

1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.

| Exp. No.: | STUDY OF TMS320C5416 DSP PROCESSOR |
|---|---|
| Date: | |

**AIM:**

To Study the architecture, memory configuration and instruction set of the TMS320VC5416 DSP processor and also the components of the TMS320VC5416 DSP Starter kit development board.

**APPARATUS REQUIRED:**

-

| S.NO | ITEM | Q.TY |
|---|---|---|
| 1 | TMS320VC3416 DSK Development Board | 1 |
| 2. | PC with Code Composer Studio IDE | 1 |
| 3 | USB Cable | 1 |
| 4 | +5V Universal Power Supply | 1 |
| 5 | AC Power cord | 1 |

**THEORY:**

**OVERVIEW OF TMS320VC5416 DSK DEVELOPMENT BOARD**

The 5416 DSP Starter Kit (DSK) is a low-cost platform, which lets enables customers to evaluate and develop applications for the TI C54X DSP family.
The primary features of the DSK are:

- 160 MHz TMS320VC5416 DSP
- PCM3002 Stereo Codec
- Four Position User DIP Switch and Four User LEDs
- On-board Flash and SRA

The TMS320VC5416 DSP is the heart of the system. It is a core member of Texas Instruments' C54X line of fixed point DSP's whose distinguishing features are 128Kwords of fast internal memory, 3 multi-channel buffered serial ports (McBSPs), an on-board timer and a 6 channel direct memory access (DMA) controller. Since members of the C54X family share common features, it is easy to use the 5416 DSK a development platform for other members of the C54X family, which have a subset of the 5416's resources (e.g. the 5402).

The 5416 have a significant amount of internal memory so typical applications will have all code and data on-chip. But when external accesses are necessary, the 5416 use a 16-bit wide external memory interface (EMIF) optimized for asynchronous memories. The DSK includes an external non-volatile Flash chip to store boot code and an external SRAM to serve

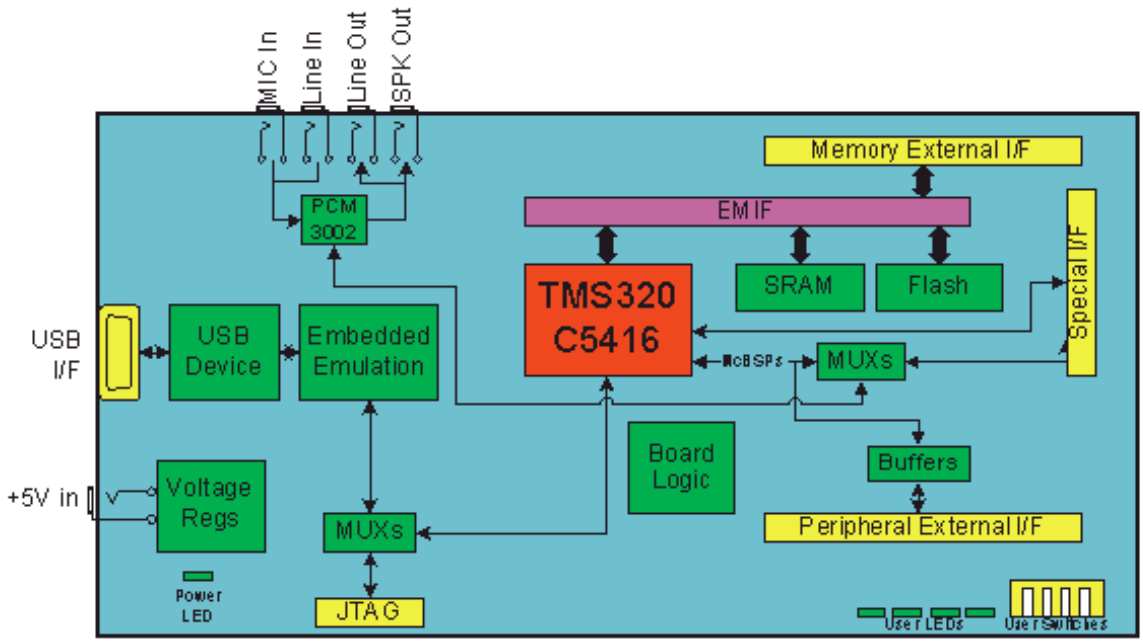
as an example of how to include external memories in your own system. The EMIF and other signals are brought out to standard TI expansion bus connectors so more features can be added by plugging in daughter-card modules.

The 5416 DSK implements the logic necessary to tie the board components together in a programmable logic device called a CPLD. In addition to random glue logic, the CPLD implements a set of 8 software programmable registers that can be used to configure various board parameters. These registers are key in using the 5416 DSK to its full potential. DSP's are frequently used in audio processing applications so the DSK includes an on-board codec called the PCM3002. Codec stands for coder/decoder, the job of the PCM3002 is to code analog input samples into a digital format for the DSP to process, then decode data coming out of the DSP to generate the processed analog output. On the DSK McBSP2 is used to send and receive the digital data to and from the codec.

Finally, the 5416 has 4 light emitting diodes (LEDs) and 4 DIP switches that allow users to interact with programs through simple LED displays and user input on the switches

**DSK BOARD FEATURES**

| Feature | Details |
|---|---|
| TMS320VC5416 DSP | 160MHz, fixed point, 128Kwords internal RAM |
| CPLD | Programmable "glue" logic |
| External SRAM | 64Kwords, 16-bit interface |
| External Flash | 256Kwords, 16-bit interface |
| PCM3002 Codec | Stereo, 6KHz –48KHz sample rate, 16 or 20 bit samples, mic, line-in, line-out and speaker jacks |
| 4 User LEDs | Writable through CPLD |
| 4 User DIP Switches | Readable through CPLD |
| 4 Jumpers | Selects power-on configuration and boot modes |
| Daughter card Expansion Interface | Allows user to enhance functionality with add-on daughter cards |
| HPI Expansion Interface | Allows high speed communication with another DSP |
| Embedded JTAG Emulator | Provides high speed JTAG debug through widely accepted USB host interface |




TMS320VC5416 DSK Overview Block Diagram

## DSK HARDWARE INSTALLATION

- Shut down and power off the PC
- Connect the supplied USB port cable to the board
- Connect the other end of the cable to the USB port of PC

***Note:*** *If you plan to install a Microphone, speaker, or Signal generator/CRO these must be plugged in properly before you connect power to the DSK*

- Plug the power cable into the board
- Plug the other end of the power cable into a power outlet
- The user LEDs should flash several times to indicate board is operational
- When you connect your DSK through USB for the first time on a Windows loaded PC the new hardware found wizard will come up. **So, Install the drivers** (The CCS CD contains the require drivers for C5416 DSK).

## OVERVIEW OF TMS320VC5416 PROCESSOR

### KEY FEATURES:

- 16 bit fixed point, high performance lower power Digital signal processor.

- Advanced multi-bus architecture with three separate 16-bit data memory buses and one program memory bus
- 40-bit arithmetic logic unit (ALU), including a 40-bit barrel shifter and two independent 40-bit accumulators
- 17- × 17-bit parallel multiplier coupled to a 40-bit dedicated adder for non-pipelined single-cycle multiply/accumulate (MAC) operation
- Compare, select, and store unit (CSSU) for the add/compare selection of the Viterbi operator
- Exponent encoder to compute an exponent value of a 40-bit accumulator value in a single cycle
- Two address generators with eight auxiliary registers and two auxiliary register arithmetic units (ARAUs)
- Data buses with a bus holder feature
- Extended addressing mode for up to 8M × 16-bit maximum addressable external program space
- 128Kwords memory – high-speed internal memory for maximum performance.
- On-chip peripherals
    - Software-programmable wait-state generator and programmable bank-switching
    - On-chip PLL – generates processor clock rate from slower external clock reference.
    - Timer – generates periodic timer events as a function of the processor clock. Used by DSP/BIOS to create time slices for multitasking.
    - DMA Controller – 6 channel direct memory access controller for high speed data transfers without intervention from the DSP.
    - 3 McBSPs – Multi-channel buffered serial ports. Each McBSP can be used for high-speed serial data transmission with external devices or reprogrammed as general purpose I/Os.
    - Time-division multiplexed (TDM) serial port
    - 16-bit timer with 4-bit pre-scalar
- On-chip scan-based emulation capability
- IEEE 1149.1 † (JTAG) boundary scan test capability
- 5.0-V power supply devices with speeds up to 40 million instructions per second (MIPS) (25-ns instruction cycle time)

## ARCHITECTURE

The ’54x DSP’s use an advanced, modified Harvard architecture that maximizes processing power by maintaining one program memory bus and three data memory buses.

These processors also provide an arithmetic logic unit (ALU) that has a high degree of parallelism, application-specific hardware logic, on-chip memory, and additional on-chip peripherals. These DSP families also provide a highly specialized instruction set, which is the basis of the operational flexibility and speed of these DSPs.

Separate program and data spaces allow simultaneous access to program instructions and data, providing the high degree of parallelism. Two reads and one write operation can be performed in a single cycle.

### CENTRAL PROCESSING UNIT (CPU)

The CPU of the ’54x devices contains:

- A 40-bit arithmetic logic unit (ALU)
- Two 40-bit accumulators
- A barrel shifter
- A 17 X 17-bit multiplier/adder
- A compare, select, and store unit (CSSU)

### Arithmetic Logic Unit (ALU)

The ’54x devices perform 2s-complement arithmetic using a 40-bit ALU and two 40-bit accumulators (ACCA and ACCB). The ALU also can perform Boolean operations.The ALU can function as two 16-bit ALUs and perform two 16-bit operations simultaneously when the C16 bit in status register 1 (ST1) is set.

### Accumulators

The accumulators, ACCA and ACCB, store the output from the ALU or the multiplier / adder block; the accumulators can also provide a second input to the ALU or the multiplier / adder. The bits in each accumulator is grouped as follows:

Guard bits (bits 32–39)

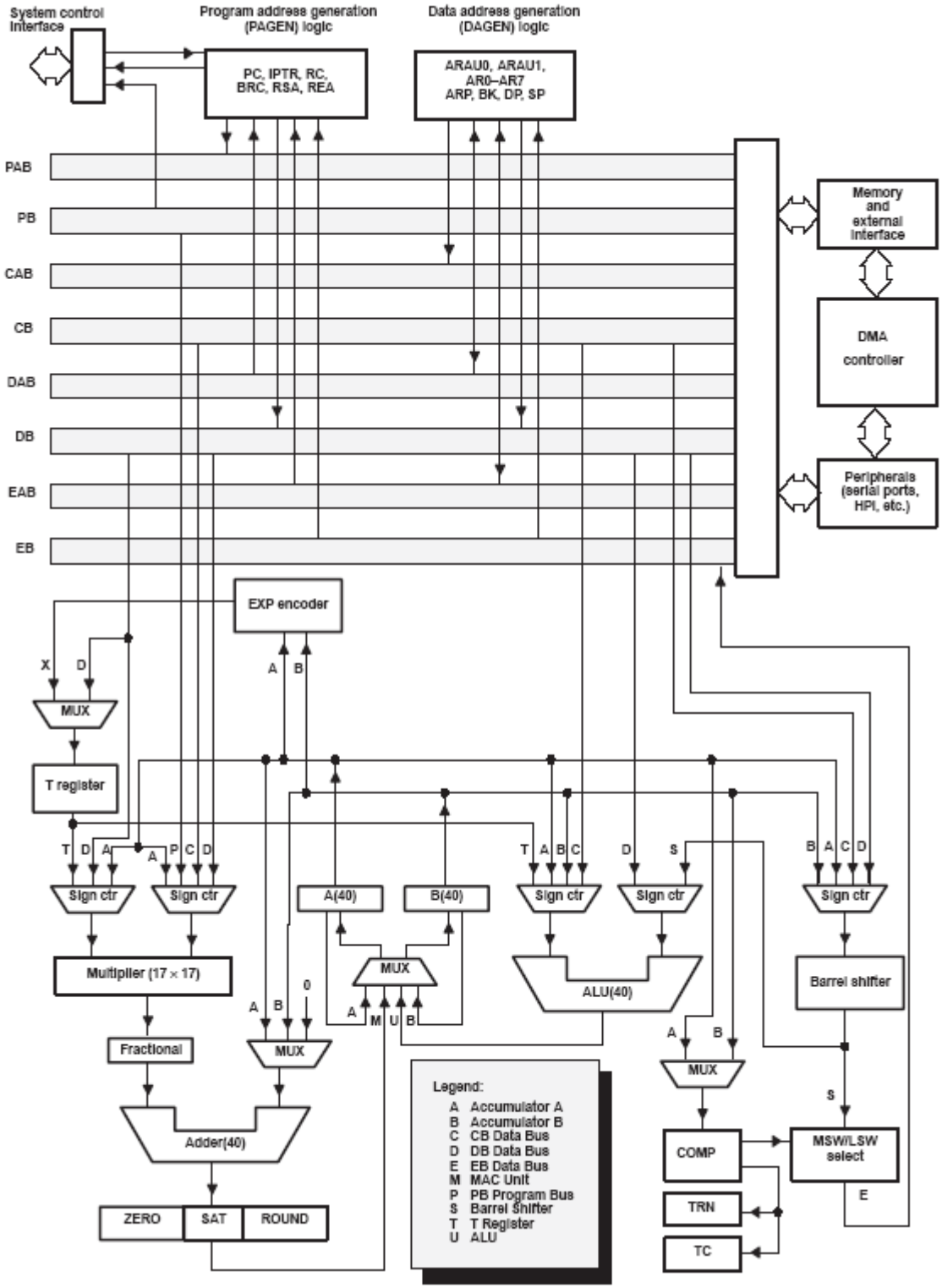
A high-order word (bits 16–31)

A low-order word (bits 0–15)

### Multiplier/ Adder

The multiplier / adder performs 17 X 17-bit 2s-complement multiplication with a 40-bit accumulation in a single instruction cycle. The multiplier has two inputs: one input is selected from the TREG, a data-memory operand, or an accumulator; the other is selected from the program memory, the data memory, an accumulator, or an immediate value.

## FUNCTIONAL BLOCK DIAGRAM




## Barrel Shifter

The ’54x’s barrel shifter has a 40-bit input connected to the accumulator or data memory (CB, DB) and a 40-bit output connected to the ALU or data memory (EB). The

barrel shifter produces a left shift of 0 to 31 bits and a right shift of 0 to 16 bits on the input data. The shift requirements are defined in the shift-count field (ASM) of ST1 or defined in the temporary register (TREG), which is designated as a shift-count register. This shifter and the exponent detector normalize the values in an accumulator in a single cycle.

**Compare, Select, and Store Unit (CSSU)**

The compare, select, and store unit (CSSU) performs maximum comparisons between the accumulator's high and low words, allows the test/ control (TC) flag bit of status register 0 (ST0) and the transition (TRN) register to keep their M transition histories, and selects the larger word in the accumulator to be stored in data memory. The CSSU also accelerates Viterbi-type butterfly computation with optimized on-chip hardware.

**Auxiliary Registers (AR0–AR7)**

The eight 16-bit auxiliary registers (AR0–AR7) can be accessed by the central airthmetic logic unit (CALU) and modified by the auxiliary register arithmetic units (ARAUs). The primary function of the auxiliary registers is generating 16-bit addresses for data space. However, these registers also can act as general-purpose registers or counters.

**Temporary Register (TREG)**

The TREG is used to hold one of the multiplicands for multiply and multiply/ accumulate instructions. It can hold a dynamic (execution-time programmable) shift count for instructions with a shift operation such as ADD, LD, and SUB. It also can hold a dynamic bit address for the BITT instruction.

The EXP instruction stores the exponent value computed into the TREG, while the NORM instruction uses the TREG value to normalize the number. For ACS operation of Viterbi decoding, TREG holds branch metrics used by the DADST and DSADT instructions.

**Circular-Buffer-Size Register (BK)**

The 16-bit BK is used by the ARAUs in circular addressing to specify the data block size. Block-Repeat Registers (BRC, RSA, REA) The block-repeat counter (BRC) is a 16-bit register used to specify the number of times a block of code is to be repeated when performing a block repeat. The block-repeat start address (RSA) is a 16-bit register containing the starting address of the block of program memory to be repeated when operating in the repeat mode. The 16-bit block-repeat end address (REA) contains the ending address if the block of program memory is to be repeated when operating in the repeat mode.

**Bus Structure**

The '54x device architecture is built around eight major 16-bit buses:

- One program-read bus (PB) which carries the instruction code and immediate operands from program memory

- Two data-read buses (CB, DB) and one data-write bus (EB), which interconnect to various elements, such as the CPU, data-address generation logic (DAGEN), program-address generation logic (PAGEN), on-chip peripherals, and data memory
- The CB and DB carry the operands read from data memory.
- The EB carries the data to be written to memory.
- Four address buses (PAB, CAB, DAB, and EAB), which carry the
- addresses needed for instruction execution

**MEMORY**

The minimum memory address range for the ’54x devices is 192K words composed of 64K words in program space, 64K words in data space, and 64K words in I/O space. Selected devices also provide extended program memory space of up to 8M words.

**On-Chip ROM**

The ’54x devices include on-chip maskable ROM that can be mapped into program memory or data memory depending on the device. On-chip ROM is mapped into program space by the microprocessor/microcontroller (MP/MC) mode control pin. On-chip ROM that can be mapped into data space is controlled by the DROM bit in the processor mode status register (PMST). This allows an instruction to use data stored in the ROM as an operand.

**On-Chip Dual-Access RAM (DARAM)**

Dual-access RAM blocks can be accessed twice per machine cycle. This memory is intended primarily to store data values; however, it can be used to store program as well. At reset, the DARAM is mapped into data memory space. DARAM can be mapped into program/data memory space by setting the OVLY bit in the PMST register.

**On-Chip Single-Access RAM (SARAM)**

Each of the SARAM blocks is a single-access memory. This memory is intended primarily to store data values; however, it can be used to store program as well. SARAM can be mapped into program/data memory space by setting the OVLY bit in the PMST register.

**Program Memory**

- The standard external program memory space on the ’54x devices addresses up to 64K 16-bit words. Software can configure their memory cells to reside inside or outside of the program address map.

**Data Memory**

The data memory space on the ’54x device addresses 64K of 16-bit words. The device automatically accesses the on-chip RAM when addressing within its bounds. When an address is generated outside the RAM bounds, the device automatically generates an external access.

## INSTRUCTION SET:

| Mnemonic Syntax | Description | Words/ Cycles† | Opcode MSB | | | LSB |
|---|---|---|---|---|---|---|
| **Arithmetic Instructions** | | | | | | |
| **ABDST** *Xmem, Ymem* | Absolute distance | 1/1 | 1110 | 0011 | XXXX | YYYY |
| **ABS** *src* [, *dst* ] | Absolute value of ACC | 1/1 | 1111 | 01SD | 1000 | 0101 |
| **ADD** *Smem, src* | Add operand to ACC | 1/1 | 0000 | 000S | IAAA | AAAA |
| **ADD** *Smem*, **TS**, *src* | Add (shifted by TREG[5:0]) operand to ACC | 1/1 | 0000 | 010S | IAAA | AAAA |
| **ADD** *Smem*, **16**, *src* [, *dst* ] | Add (shifted by 16 bits) operand to ACC | 1/1 | 0011 | 11SD | IAAA | AAAA |
| **ADD** *Smem* [, *SHIFT*], *src* [, *dst* ] | Add shifted operand to ACC (2-word opcode) | 2/2 | 0110<br>0000 | 1111<br>11SD | IAAA<br>000S | AAAA<br>HIFT |
| **ADD** *Xmem, SHFT, src* | Add shifted operand to ACC | 1/1 | 1001 | 000S | XXXX | SHFT |
| **ADD** *Xmem, Ymem, dst* | Add dual operands, shift result by 16 | 1/1 | 1010 | 000D | XXXX | YYYY |
| **ADD** *#lk* [, *SHFT*], *src* [, *dst* ] | Add shifted long-immediate value to ACC | 2/2 | 1111 | 00SD | 0000 | SHFT |
| **ADD** *#lk*, **16**, *src* [, *dst* ] | Add (shifted by 16 bits) long-immediate to ACC | 2/2 | 1111 | 00SD | 0110 | 0000 |
| **ADD** *src* [, *SHIFT*], [, *dst* ] | Add ACC(s) (A/B), then shift result | 1/1 | 1111 | 01SD | 000S | HIFT |
| **ADD** *src*, **ASM** [, *dst* ] | Add ACC(s) (A/B), then shift result by ASM value | 1/1 | 1111 | 01SD | 1000 | 0000 |
| **ADDC** *Smem, src* | Add to accumulator with carry | 1/1 | 0000 | 011S | IAAA | AAAA |
| **ADDM** *#lk, Smem* | Add long-immediate value to memory | 2/2 | 0110 | 1011 | IAAA | AAAA |
| **ADDS** *Smem, src* | Add to ACC with sign-extension suppressed | 1/1 | 0000 | 001S | IAAA | AAAA |
| **DADD** *Lmem, src* [, *dst* ] | Double/dual add to accumulator | 1/1 | 0101 | 00SD | IAAA | AAAA |
| **DADST** *Lmem, dst* | Double/dual add/subtract of T, long operand | 1/1 | 0101 | 101D | IAAA | AAAA |
| **DELAY** *Smem* | Memory delay | 1/1 | 0100 | 1101 | IAAA | AAAA |
| **DRSUB** *Lmem, src* | Double/dual 16-bit subtract from long word | 1/1 | 0101 | 100S | IAAA | AAAA |
| **DSADT** *Lmem, dst* | Double/dual, subtract/add of T, long operand | 1/1 | 0101 | 111D | IAAA | AAAA |
| **DSUB** *Lmem, src* | Double-precision/dual 16-bit subtract from ACC | 1/1 | 0101 | 010S | IAAA | AAAA |

† Values for words and cycles assume the use of DARAM for data. Add one word and one cycle when using long-offset indirect addressing or absolute addressing with a single data-memory operand.
‡ Delayed Instruction
§ Condition true
¶ Condition false

| Mnemonic Syntax | Description | Words/ Cycles† | Opcode MSB | | | LSB |
|---|---|---|---|---|---|---|
| **Arithmetic Instructions (Continued)** | | | | | | |
| **DSUBT** *Lmem, dst* | Double/dual, subtract/subtract of T, long operand | 1/1 | 0101 | 110D | IAAA | AAAA |
| **EXP** *src* | Accumulator exponent | 1/1 | 1111 | 010S | 1000 | 1110 |
| **FIRS** *Xmem, Ymem, pmad* | Symmetrical finite impulse response filter | 2/3 | 1110 | 0000 | XXXX | YYYY |
| **LMS** *Xmem, Ymem* | Least mean square | 1/1 | 1110 | 0001 | XXXX | YYYY |
| **MAC[R]** *Smem, src* | Multiply by TREG, add to ACC, round if specified | 1/1 | 0010 | 10RS | IAAA | AAAA |
| **MAC[R]** *Xmem, Ymem, src* [*, dst* ] | Multiply dual, add to ACC, round if specified | 1/1 | 1011 | 0RSD | XXXX | YYYY |
| **MAC** *#lk, src* [*, dst* ] | Multiply TREG by long-immediate, add to ACC | 2/2 | 1111 | 00SD | 0110 | 0111 |
| **MAC** *Smem, #lk, src* [*, dst* ] | Multiply by long-immediate value, add to ACC | 2/2 | 0110 | 01SD | IAAA | AAAA |
| **MACA[R]** *Smem* [*, B* ] | Multiply by ACCA, add to ACCB [round] | 1/1 | 0011 | 01R1 | IAAA | AAAA |
| **MACA[R]** **T**, *src* [*, dst* ] | Multiply TREG by ACCA, add to ACC [round] | 1/1 | 1111 | 01SD | 1000 | 100R |
| **MACD** *Smem, pmad, src* | Multiply by program memory, accumulate/delay | 2/3 | 0111 | 101S | IAAA | AAAA |
| **MACP** *Smem, pmad, src* | Multiply by program memory, then accumulate | 2/3 | 0111 | 100S | IAAA | AAAA |
| **MACSU** *Xmem, Ymem, src* | Multiply signed by unsigned, then accumulate | 1/1 | 1010 | 011S | XXXX | YYYY |
| **MAS[R]** *Smem, src* | Multiply by T, subtract from ACC [round] | 1/1 | 0010 | 11RS | IAAA | AAAA |
| **MAS[R]** *Xmem, Ymem, src* [*, dst* ] | Multiply dual, subtract from ACC [round] | 1/1 | 1011 | 1RSD | XXXX | YYYY |
| **MASA** *Smem* [*, B* ] | Multiply operand by ACCA, subtract from ACCB | 1/1 | 0011 | 0011 | IAAA | AAAA |
| **MASA[R]** **T**, *src* [*, dst* ] | Multiply ACCA by T, subtract from ACC [round] | 1/1 | 1111 | 01SD | 1000 | 101R |
| **MAX** *dst* | Accumulator maximum | 1/1 | 1111 | 010D | 1000 | 0110 |
| **MIN** *dst* | Accumulator minimum | 1/1 | 1111 | 010D | 1000 | 0111 |

| Mnemonic Syntax | Description | Words/ Cycles† | Opcode MSB | | | LSB |
|---|---|---|---|---|---|---|
| **Arithmetic Instructions (Continued)** | | | | | | |
| **MPY[R]** *Smem, dst* | Multiply TREG by operand, round if specified | 1/1 | 0010 | 00RD | IAAA | AAAA |
| **MPY** *Xmem, Ymem, dst* | Multiply dual data-memory operands | 1/1 | 1010 | 010D | XXXX | YYYY |
| **MPY** *Smem, #lk, dst* | Multiply operand by long-immediate operand | 2/2 | 0110 | 001D | IAAA | AAAA |
| **MPY** *#lk, dst* | Multiply TREG value by long-immediate operand | 2/2 | 1111 | 000D | 0110 | 0110 |
| **MPYA** *Smem* | Multiply single data-memory operand by ACCA | 1/1 | 0011 | 0001 | IAAA | AAAA |
| **MPYA** *dst* | Multiply TREG value by ACCA | 1/1 | 1111 | 010D | 1000 | 1100 |
| **MPYU** *Smem, dst* | Multiply unsigned | 1/1 | 0010 | 010D | IAAA | AAAA |
| **NEG** *src* [, *dst* ] | Negate accumulator | 1/1 | 1111 | 01SD | 1000 | 0100 |
| **NORM** *src* [, *dst* ] | Normalize | 1/1 | 1111 | 01SD | 1000 | 1111 |
| **POLY** *Smem* | Evaluate polynomial | 1/1 | 0011 | 0110 | IAAA | AAAA |
| **RND** *src* [, *dst* ] | Round accumulator | 1/1 | 1111 | 01SD | 1001 | 1111 |
| **SAT** *src* | Saturate accumulator | 1/1 | 1111 | 010S | 1000 | 0011 |
| **SQDST** *Xmem, Ymem* | Square distance | 1/1 | 1110 | 0010 | XXXX | YYYY |
| **SQUR** *Smem, dst* | Square single data-memory operand | 1/1 | 0010 | 011D | IAAA | AAAA |
| **SQUR A**, *dst* | Square ACCA high | 1/1 | 1111 | 010D | 1000 | 1101 |
| **SQURA** *Smem, src* | Square and accumulate | 1/1 | 0011 | 100S | IAAA | AAAA |
| **SQURS** *Smem, src* | Square and subtract | 1/1 | 0011 | 101S | IAAA | AAAA |
| **SUB** *Smem, src* | Subtract operand from accumulator | 1/1 | 0000 | 100S | IAAA | AAAA |
| **SUB** *Smem*, **TS**, *src* | Shift by TREG[5:0], then subtract from ACC | 1/1 | 0000 | 110S | IAAA | AAAA |
| **SUB** *Smem*, 16, *src* [, *dst* ] | Shift operand 16 bits, then subtract from ACC | 1/1 | 0100 | 00SD | IAAA | AAAA |
| **SUB** *Smem* [, *SHIFT* ], *src* [, *dst* ] | Shift operand, then subtract from ACC (2-word opcode) | 2/2 | 0110<br>0000 | 1111<br>11SD | IAAA<br>001S | AAAA<br>HIFT |
| **SUB** *Xmem, SHFT, src* | Shift operand, then subtract from ACC | 1/1 | 1001 | 001S | XXXX | SHFT |
| **SUB** *Xmem, Ymem, dst* | Shift dual operands by 16, then subtract | 1/1 | 1010 | 001D | XXXX | YYYY |

| Mnemonic Syntax | Description | Words/ Cycles† | Opcode MSB | | | LSB |
|---|---|---|---|---|---|---|
| **Arithmetic Instructions (Continued)** | | | | | | |
| **SUB** *#lk* [, *SHFT* ], *src* [, *dst* ] | Shift long-immediate, then subtract from ACC | 2/2 | 1111 | 00SD | 0001 | SHFT |
| **SUB** *#lk*, 16, *src* [, *dst* ] | Shift long-immediate 16 bits, subtract from ACC | 2/2 | 1111 | 00SD | 0110 | 0001 |
| **SUB** *src* [, *SHIFT* ], [, *dst* ] | Subtract shifted ACC from ACC | 1/1 | 1111 | 01SD | 001S | HIFT |
| **SUB** *src*, **ASM** [, *dst* ] | Subtract ACC shifted by ASM from ACC | 1/1 | 1111 | 01SD | 1000 | 0001 |
| **SUBB** *Smem, src* | Subtract from accumulator with borrow | 1/1 | 0000 | 111D | IAAA | AAAA |
| **SUBC** *Smem, src* | Subtract conditionally | 1/1 | 0001 | 111S | IAAA | AAAA |
| **SUBS** *Smem, src* | Subtract from ACC, sign-extension suppressed | 1/1 | 0000 | 101S | IAAA | AAAA |
| **Control Instructions** | | | | | | |
| B[D] *pmad* | Branch unconditionally with optional delay | 2/4,2‡ | 1111 | 00Z0 | 0111 | 0011 |
| BACC[D] *src* | Branch to address in ACC, optional delay | 1/6,4‡ | 1111 | 01ZS | 1110 | 0010 |
| BANZ[D] *pmad, Sind* | Branch on AR(ARP) not zero, optional delay | 2/4§,2¶,2‡ | 0110 | 11Z0 | IAAA | AAAA |
| BC[D] *pmad, cond* [, *cond* [, *cond* ]] | Branch conditionally, optional delay | 2/5§,3¶,3‡ | 1111 | 10Z0 | CCCC | CCCC |
| CALA[D] *src* | Call subroutine at address in ACC, optional delay | 1/6,4‡ | 1111 | 01ZS | 1110 | 0011 |
| CALL[D] *pmad* | Call unconditionally, optional delay | 2/4,2¶ | 1111 | 00Z0 | 0111 | 0100 |
| CC[D] *pmad, cond* [, *cond* [, *cond* ]] | Call conditionally, optional delay | 2/5§,3¶,3‡ | 1111 | 10Z1 | CCCC | CCCC |
| FB[D] *extpmad* | Far branch unconditionally (optional delay) | 2/4,2‡ | 1111 | 10Z0 | 1KKK | KKKK |
| FBACC[D] *src* | Far branch to address in ACC, optional delay | 1/6,4‡ | 1111 | 01ZS | 1110 | 0110 |
| FCALA[D] *src* | Far call to address in ACC, optional delay | 1/6,4‡ | 1111 | 01ZS | 1110 | 0111 |
| FCALL[D] *extpmad* | Far call unconditionally, optional delay | 2/4,2‡ | 1111 | 10Z1 | 1KKK | KKKK |
| FRAME *K* | Stack pointer immediate offset | 1/1 | 1110 | 1110 | KKKK | KKKK |
| FRET[D] | Far return (FRETD is for delayed return) | 1/6,4‡ | 1111 | 01Z0 | 1110 | 0100 |

| Mnemonic Syntax | Description | Words/ Cycles† | Opcode MSB | | | LSB |
|---|---|---|---|---|---|---|
| Control Instructions (Continued) | | | | | | |
| **FRETE[D]** | Far return, enable interrupts, optional delay | 1/6,4‡ | 1111 | 01Z0 | 1110 | 0101 |
| **IDLE** *K* | Idle until interrupt | 1/4 | 1111 | 01NN | 1110 | 0001 |
| **INTR** *K* | Software interrupt | 1/3 | 1111 | 0111 | 110K | KKKK |
| **MAR** *Smem* | Modify auxiliary register | 1/1 | 0110 | 1101 | IAAA | AAAA |
| **NOP** | No operation | 1/1 | 1111 | 0100 | 1001 | 0101 |
| **POPD** *Smem* | Pop top of stack to data memory | 1/1 | 1000 | 1011 | IAAA | AAAA |
| **POPM** *MMR* | Pop top of stack to memory-mapped register | 1/1 | 1000 | 1010 | IAAA | AAAA |
| **PSHD** *Smem* | Push data-memory value onto stack | 1/1 | 0100 | 1011 | IAAA | AAAA |
| **PSHM** *MMR* | Push memory-mapped register onto stack | 1/1 | 0100 | 1010 | IAAA | AAAA |
| **RC[D]** *cond* [, *cond* [, *cond* ]] | Return conditionally, optional delay | 1/5§,3¶,3‡ | 1111 | 11Z0 | CCCC | CCCC |
| **RESET** | Software reset | 1/3 | 1111 | 0111 | 1110 | 0000 |
| **RET[D]** | Return, optional delay | 1/5,3‡ | 1111 | 11Z0 | 0000 | 0000 |
| **RETE[D]** | Return and enable interrupts, optional delay | 1/5,3‡ | 1111 | 01Z0 | 1110 | 1011 |
| **RETF[D]** | Return fast and enable interrupts, optional delay | 1/3,1‡ | 1111 | 01Z0 | 1001 | 1011 |
| **RPT** *Smem* | Repeat next instruction, count is in operand | 1/1 | 0100 | 0111 | IAAA | AAAA |
| **RPT** *#K* | Repeat next instruction, count is short immediate | 1/1 | 1110 | 1100 | KKKK | KKKK |
| **RPT** *#lk* | Repeat next instruction, count is long immediate | 2/2 | 1111 | 0000 | 0111 | 0000 |
| **RPTB[D]** *pmad* | Block repeat, optional delay | 2/4,2‡ | 1111 | 00Z0 | 0111 | 0010 |
| **RPTZ** *dst, #lk* | Repeat next instruction and clear accumulator | 2/2 | 1111 | 000D | 0111 | 0001 |
| **RSBX** *N, SBIT* | Reset status-register bit | 1/1 | 1111 | 01N0 | 1011 | SBIT |
| **SSBX** *N, SBIT* | Set status-register bit | 1/1 | 1111 | 01N1 | 1011 | SBIT |
| **TRAP** *K* | Software interrupt | 1/3 | 1111 | 0100 | 110K | KKKK |

| Mnemonic Syntax | Description | Words/ Cycles† | Opcode MSB | | | LSB |
|---|---|---|---|---|---|---|
| **Control Instructions (Continued)** | | | | | | |
| **XC** *n, cond* [*, cond* [*, cond* ] ] | Execute conditionally | 1/1 | 1111 | 11N1 | CCCC | CCCC |
| **I/O Instructions** | | | | | | |
| **PORTR** *PA, Smem* | Read data from port | 2/2 | 0111 | 0100 | IAAA | AAAA |
| **PORTW** *Smem, PA* | Write data to port | 2/2 | 0111 | 0101 | IAAA | AAAA |
| **Load/Store Instructions** | | | | | | |
| **CMPS** *src, Smem* | Compare, select and store maximum | 1/1 | 1000 | 111S | IAAA | AAAA |
| **DLD** *Lmem, dst* | Long-word load to accumulator | 1/1 | 0101 | 011D | IAAA | AAAA |
| **DST** *src, Lmem* | Store accumulator in long word | 1/2 | 0100 | 111S | IAAA | AAAA |
| **LD** *Smem, dst* | Load accumulator with operand | 1/1 | 0001 | 000D | IAAA | AAAA |
| **LD** *Smem*, **TS**, *dst* | Shift operand by TREG[5:0], then load into ACC | 1/1 | 0001 | 010D | IAAA | AAAA |
| **LD** *Smem*, **16**, *dst* | Shift operand by 16 bits, then load into ACC | 1/1 | 0100 | 010D | IAAA | AAAA |
| **LD** *Smem* [*, SHIFT* ], *dst* | Shift operand, then load into ACC (2-word opcode) | 2/2 | 0110<br>0000 | 1111<br>110D | IAAA<br>010S | AAAA<br>HIFT |
| **LD** *Xmem, SHFT, dst* | Shift operand, then load into ACC | 1/1 | 1001 | 010D | XXXX | SHFT |
| **LD** *#K, dst* | Load ACC with short-immediate operand | 1/1 | 1110 | 100D | KKKK | KKKK |
| **LD** *#lk* [*, SHFT* ], *dst* | Shift long-immediate, then load into ACC | 2/2 | 1111 | 000D | 0010 | SHFT |
| **LD** *#lk*, **16**, *dst* | Shift long-immediate 16 bits, load into ACC | 2/2 | 1111 | 000D | 0110 | 0010 |
| **LD** *src*, **ASM** [*, dst* ] | Shift ACC by value in ASM register | 1/1 | 1111 | 01SD | 1000 | 0010 |
| **LD** *src* [*, SHIFT* ] [*, dst* ] | Shift accumulator | 1/1 | 1111 | 01SD | 010S | HIFT |
| **LD** *Smem*, **T** | Load TREG with single data-memory operand | 1/1 | 0011 | 0000 | IAAA | AAAA |
| **LD** *Smem*, **DP** | Load DP with single data-memory operand | 1/3 | 0100 | 0110 | IAAA | AAAA |
| **LD** *#k9*, **DP** | Load DP with 9-bit operand | 1/1 | 1110 | 101K | KKKK | KKKK |
| **LD** *#k5*, **ASM** | Load ACC shift-mode register with 5-bit operand | 1/1 | 1110 | 1101 | 000K | KKKK |

| Mnemonic Syntax | Description | Words/ Cycles† | Opcode MSB | | | LSB |
|---|---|---|---|---|---|---|
| Load/Store Instructions (Continued) | | | | | | |
| **LD** *#k3*, **ARP** | Load ARP with 3-bit operand | 1/1 | 1111 | 0100 | 1010 | 0KKK |
| **LD** *Smem*, **ASM** | Load operand bits 4–0 into ASM register | 1/1 | 0011 | 0010 | IAAA | AAAA |
| **LD** *Xmem, dst*<br>**\|\| MAC[R]** *Ymem* [, *dst_* ] | Parallel load, multiply/accumulate [round] | 1/1 | 1010 | 10RD | XXXX | YYYY |
| **LD** *Xmem, dst*<br>**\|\| MAS[R]** *Ymem* [, *dst_* ] | Parallel load, multiply/subtract [round] | 1/1 | 1010 | 11RD | XXXX | YYYY |
| **LDM** *MMR, dst* | Load memory-mapped register | 1/1 | 0100 | 100D | IAAA | AAAA |
| **LDR** *Smem, dst* | Load memory value in ACC high with rounding | 1/1 | 0001 | 011D | IAAA | AAAA |
| **LDU** *Smem, dst* | Load unsigned memory value | 1/1 | 0001 | 001D | IAAA | AAAA |
| **LTD** *Smem* | Load TREG and insert delay | 1/1 | 0100 | 1100 | IAAA | AAAA |
| **SACCD** *src, Xmem, cond* | Store accumulator conditionally | 1/1 | 1001 | 111S | XXXX | COND |
| **SRCCD** *Xmem, cond* | Store block-repeat counter conditionally | 1/1 | 1001 | 1101 | XXXX | COND |
| **ST T**, *Smem* | Store TREG | 1/1 | 1000 | 1100 | IAAA | AAAA |
| **ST TRN**, *Smem* | Store TRN | 1/1 | 1000 | 1101 | IAAA | AAAA |
| **ST** *#lk*, *Smem* | Store long-immediate operand | 2/2 | 0111 | 0110 | IAAA | AAAA |
| **STH** *src*, *Smem* | Store accumulator high to data memory | 1/1 | 1000 | 001S | IAAA | AAAA |
| **STH** *src*, **ASM**, *Smem* | Shift ACC high by ASM, store to data memory | 1/1 | 1000 | 011S | IAAA | AAAA |
| **STH** *src, SHFT, Xmem* | Shift ACC high, then store to data memory | 1/1 | 1001 | 101S | XXXX | SHFT |
| **STH** *src* [, *SHIFT* ], *Smem* | Shift ACC high, then store to data memory (2-word opcode) | 2/2 | 0110<br>0000 | 1111<br>110S | IAAA<br>011S | AAAA<br>HIFT |
| **ST** *src, Ymem*<br>**\|\| ADD** *Xmem, dst* | Store ACC with parallel add | 1/1 | 1100 | 00SD | XXXX | YYYY |
| **ST** *src, Ymem*<br>**\|\| LD** *Xmem, dst* | Store ACC with parallel load into accumulator | 1/1 | 1100 | 10SD | XXXX | YYYY |
| **ST** *src, Ymem*<br>**\|\| LD** *Xmem*, **T** | Store ACC with parallel load into TREG | 1/1 | 1110 | 01S0 | XXXX | YYYY |
| **ST** *src, Ymem*<br>**\|\| MAC[R]** *Xmem, dst* | Parallel store and multiply ACC [round] | 1/1 | 1101 | 0RSD | XXXX | YYYY |

| Mnemonic Syntax | Description | Words/ Cycles† | Opcode MSB | | | LSB |
|---|---|---|---|---|---|---|
| **Load/Store Instructions (Continued)** | | | | | | |
| **ST** *src*, *Ymem*<br>**\|\| MAS[R]** *Xmem*, *dst* | Parallel store, multiply, and subtract | 1/1 | 1101 | 1RSD | XXXX | YYYY |
| **ST** *src*, *Ymem*<br>**\|\| MPY** *Xmem*, *dst* | Parallel store and multiply | 1/1 | 1100 | 11SD | XXXX | YYYY |
| **ST** *src*, *Ymem*<br>**\|\| SUB** *Xmem*, *dst* | Parallel store and subtract | 1/1 | 1100 | 01SD | XXXX | YYYY |
| **STL** *src*, *Smem* | Store ACC low to data memory | 1/1 | 1000 | 000S | IAAA | AAAA |
| **STL** *src*, **ASM**, *Smem* | Shift ACC low by ASM, store to data memory | 1/1 | 1000 | 010S | IAAA | AAAA |
| **STL** *src*, *SHFT*, *Xmem* | Shift ACC low, then store to data memory | 1/1 | 1001 | 100S | XXXX | SHFT |
| **STL** *src* [, *SHIFT*], *Smem* | Shift ACC low, then store to data memory (2-word opcode) | 2/2 | 0110<br>0000 | 1111<br>110S | IAAA<br>100S | AAAA<br>HIFT |
| **STLM** *src*, *MMR* | Store accumulator low to memory | 1/1 | 1000 | 100S | IAAA | AAAA |
| **STM** *#lk*, *MMR* | Store ACC low into memory-mapped register | 2/2 | 0111 | 0111 | IAAA | AAAA |
| **STRCD** *Xmem*, *cond* | Store TREG conditionally | 1/1 | 1001 | 1100 | XXXX | COND |
| **Logical Instructions** | | | | | | |
| **AND** *Smem*, *src* | AND single data-memory operand with ACC | 1/1 | 0001 | 100S | IAAA | AAAA |
| **AND** *#lk* [, *SHFT* ], *src* [, *dst* ] | Shift long-immediate operand, AND with ACC | 2/2 | 1111 | 00SD | 0011 | SHFT |
| **AND** *#lk*, **16**, *src* [, *dst* ] | Shift long-immediate 16 bits, AND with ACC | 2/2 | 1111 | 00SD | 0110 | 0011 |
| **AND** *src* [, *SHIFT* ], [, *dst* ] | AND accumulator(s), then shift result | 1/1 | 1111 | 00SD | 100S | HIFT |
| **ANDM** *#lk*, *Smem* | AND memory with long-immediate operand | 2/2 | 0110 | 1000 | IAAA | AAAA |
| **BIT** *Xmem*, *BITC* | Test bit | 1/1 | 1001 | 0110 | XXXX | BITC |
| **BITF** *Smem*, *#lk* | Test bit field specified by immediate value | 2/2 | 0110 | 0001 | IAAA | AAAA |
| **BITT** *Smem* | Test bit specified by TREG | 1/1 | 0011 | 0100 | IAAA | AAAA |
| **CMPL** *src* [, *dst* ] | Complement accumulator | 1/1 | 1111 | 01SD | 1001 | 0011 |

| Mnemonic Syntax | Description | Words/ Cycles† | Opcode MSB | | | LSB |
|---|---|---|---|---|---|---|
| **Logical Instructions (Continued)** | | | | | | |
| **CMPM** *Smem, #lk* | Compare memory with long-immediate operand | 2/2 | 0110 | 0000 | IAAA | AAAA |
| **CMPR** *CC, ARx* | Compare auxiliary register with AR0 | 1/1 | 1111 | 01CC | 1010 | 1ARX |
| **OR** *Smem, src* | OR single data-memory operand with ACC | 1/1 | 0001 | 101S | IAAA | AAAA |
| **OR** *#lk* [, *SHFT* ], *src* [, *dst* ] | Shift long-immediate operand, then OR with ACC | 2/2 | 1111 | 00SD | 0100 | SHFT |
| **OR** *#lk*, 16, *src* [, *dst* ] | Shift long-immediate 16 bits, then OR with ACC | 2/2 | 1111 | 00SD | 0110 | 0100 |
| **OR** *src* [, *SHIFT* ], [, *dst* ] | OR accumulator(s), then shift result | 1/1 | 1111 | 00SD | 101S | HIFT |
| **ORM** *#lk, Smem* | OR memory with constant | 2/2 | 0110 | 1001 | IAAA | AAAA |
| **ROL** *src* | Rotate accumulator left | 1/1 | 1111 | 010S | 1001 | 0001 |
| **ROLTC** *src* | Rotate accumulator left using TC | 1/1 | 1111 | 010S | 1001 | 0010 |
| **ROR** *src* | Rotate accumulator right | 1/1 | 1111 | 010S | 1001 | 0000 |
| **SFTA** *src, SHIFT* [, *dst* ] | Shift accumulator arithmetically | 1/1 | 1111 | 01SD | 011S | HIFT |
| **SFTC** *src* | Shift accumulator conditionally | 1/1 | 1111 | 010S | 1001 | 0100 |
| **SFTL** *src, SHIFT* [, *dst* ] | Shift accumulator logically | 1/1 | 1111 | 00SD | 111S | HIFT |
| **XOR** *Smem, src* | XOR operand with ACC | 1/1 | 0001 | 110S | IAAA | AAAA |
| **XOR** *#lk* [, *SHFT*], *src* [, *dst* ] | Shift long-immediate, then XOR with ACC | 2/2 | 1111 | 00SD | 0101 | SHFT |
| **XOR** *#lk*, 16, *src* [, *dst* ] | Shift long-immediate 16 bits, then XOR with ACC | 2/2 | 1111 | 00SD | 0110 | 0101 |
| **XOR** *src* [, *SHIFT*] [, *dst* ] | XOR accumulator(s), then shift result | 1/1 | 1111 | 00SD | 110S | HIFT |
| **XORM** *#lk, Smem* | XOR memory with constant | 2/2 | 0110 | 1010 | IAAA | AAAA |
| **Move Instructions** | | | | | | |
| **MVDD** *Xmem, Ymem* | Move within data memory, X/Y addressing | 1/1 | 1110 | 0101 | XXXX | YYYY |
| **MVDK** *Smem, dmad* | Move data, destination addressing | 2/2 | 0111 | 0001 | IAAA | AAAA |
| **MVDM** *dmad, MMR* | Move data to memory-mapped register | 2/2 | 0111 | 0010 | IAAA | AAAA |
| **MVDP** *Smem, pmad* | Move data to program memory | 2/4 | 0111 | 1101 | IAAA | AAAA |

| Mnemonic Syntax | Description | Words/ Cycles† | Opcode MSB | | | LSB |
|---|---|---|---|---|---|---|
| Move Instructions (Continued) | | | | | | |
| **MVKD** *dmad, Smem* | Move data with source addressing | 2/2 | 0111 | 0000 | IAAA | AAAA |
| **MVMD** *MMR, dmad* | Move memory-mapped register to data | 2/2 | 0111 | 0011 | IAAA | AAAA |
| **MVMM** *MMRx, MMRy* | Move between memory-mapped registers | 1/1 | 1110 | 0111 | MMRX | MMRY |
| **MVPD** *pmad, Smem* | Move program memory to data memory | 2/3 | 0111 | 1100 | IAAA | AAAA |
| **READA** *Smem* | Read data memory addressed by ACCA | 1/5 | 0111 | 1110 | IAAA | AAAA |
| **WRITA** *Smem* | Write data memory addressed by ACCA | 1/5 | 0111 | 1111 | IAAA | AAAA |

**RESULT:**

Thus the architecture, instruction set and memory configuration of the TMS320VC5416 DSP processor and also the components of the TMS320VC5416 DSP Starter kit development board were studied.

| Exp. No.: | CODE COMPOSER STUDIO TUTORIAL |
|---|---|
| Date: | |

**AIM:**

To Study the usage of Code Composer Studio development environment, to build and to debug embedded real-time software applications using TMS320C5416 DSP processor.

**APPARATUS REQUIRED:**

-

| S.NO | ITEM | Q.TY |
|---|---|---|
| 1 | TMS320VC5416 DSK Development Board | 1 |
| 2. | PC with Code Composer Studio IDE | 1 |
| 3 | USB Cable | 1 |
| 4 | +5V Universal Power Supply | 1 |
| 5 | AC Power cord | 1 |

**THEORY**

Texas Instruments' Code Composer Studio development tools are bundled with the 5416DSK providing the user with an industrial-strength integrated development environment for C and assembly programming. Code Composer Studio communicates with the DSP using an on-board JTAG emulator through a USB interface**.**

**DEVELOPMENT ENVIRONMENT**

Code Composer Studio is TI's flagship development tool. It consists of an assembler, a C compiler, an integrated development environment (IDE, the graphical interface to the tools) and numerous support utilities like a hex format conversion tool.

The Code Composer IDE is the piece you see when you run Code Composer. It consists of

- Editor for creating source code in assembly and C language.
- Project manager to identify the source files.
- Integrated source level debugger that lets you examine the behavior of your program while it is running.

The IDE is responsible for calling other components such as the compiler and assembler so developers don't have to deal with the hassle of running each tool manually.

The 5416 DSK includes a special device called a JTAG emulator on-board that can directly access the register and memory state of the 5416 chip through a standardized JTAG interface port. When a user wants to monitor the progress of his program, Code Composer sends commands to the emulator through its USB host interface to check on any data the user is interested in.

When you recompile a program in Code Composer on your PC you must specifically load it onto the 5416 onto the DSK. Other things to be aware of are:

When you tell Code Composer to run, it simply starts executing at the current program counter. If you want to restart the program, you must reset the program counter by using Debug à Restart or re-loading the program, which sets the program counter implicitly.

After you start a program running it continues running on the DSP indefinitely. To stop it you need to halt it with Debug à Halt.

## GETTING STARTED WITH CODE COMPOSER STUDIO

To access the Code Composer Studio Tutorial, perform the following steps:

1) Start Code Composer Studio by double-clicking on the "CCStudio" icon located on the desktop.

2) From the Code Composer Studio Help menu, select Tutorial->Code Composer Studio Tutorial.

## USING CODE COMPOSER STUDIO WINDOWS AND TOOLBARS

All windows (except Edit windows) and all toolbars are dockable within the Code Composer Studio environment. This means you can move and align a window or toolbar to any portion of the Code Composer Studio main window. You can also move dockable windows and toolbars out of the Code Composer Studio main window and place them anywhere on the desktop. To move a toolbar, simply click-and-drag the toolbar to its new location.

### To Move a Window Out of the Main Window

1) Right-click in the window and select Allow Docking from the context menu.

2) Left-click in the window's title bar and drag the window to any location on your desktop.

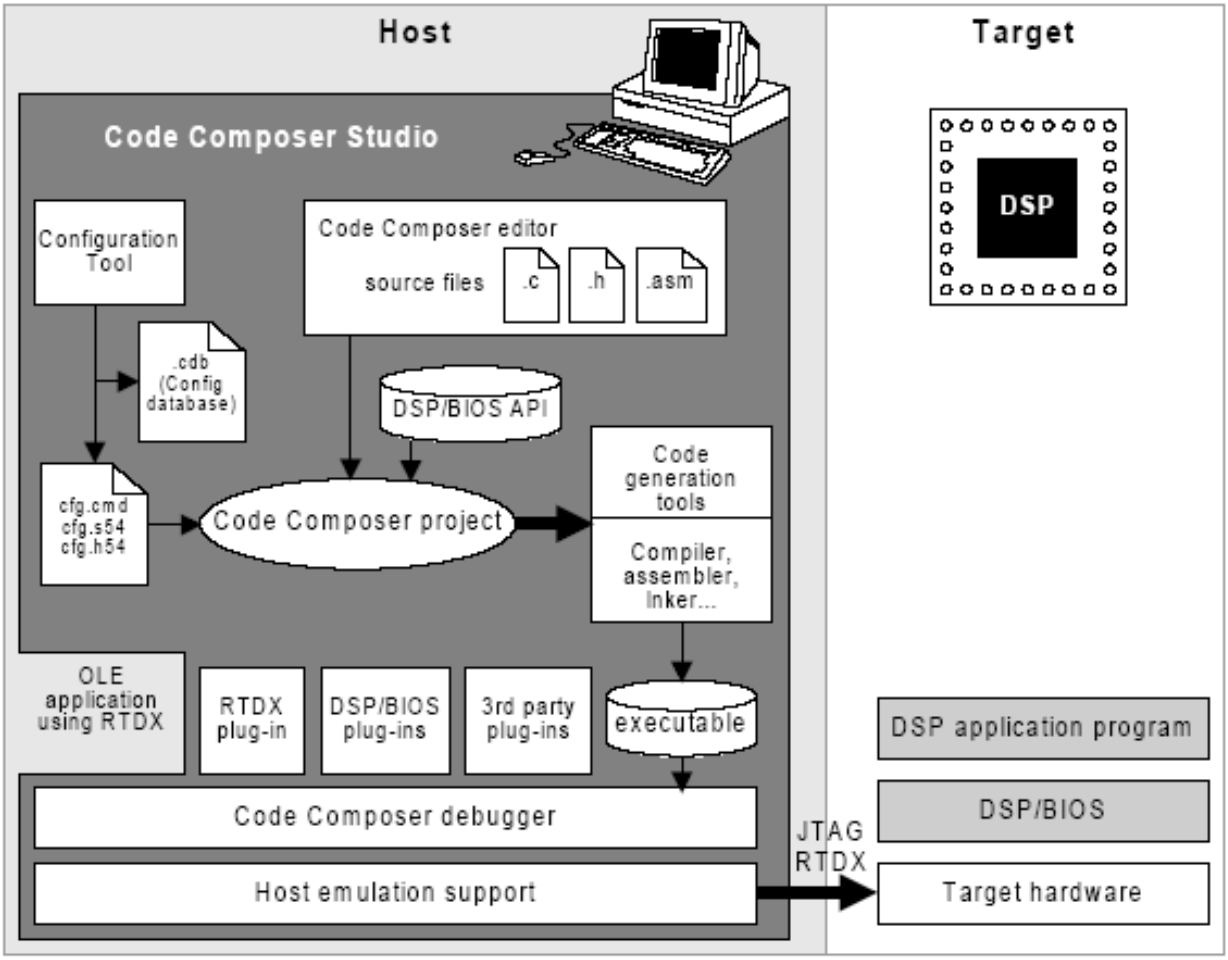
All dockable windows contain a context menu that provides three options for controlling window alignment. Allow Docking Toggles window docking on and off. Hide,

Hides the active window beneath all other windows. Float in the Main Window Turns off docking and allows the active window to float in the main window.

Code Composer Studio includes the following components:

- TMS320C54x code generation tools.
- Code Composer Studio Integrated Development Environment (IDE).
- DSP/BIOS plug-ins and API.
- RTDX plug-in, host interface, and API.

These components work together as shown here:




**CODE COMPOSER FEATURES INCLUDE:**

- IDE
- Debug IDE
- Advanced watch windows
- Integrated editor
- File I/O, Probe Points, and graphical algorithm scope probes
- Advanced graphical signal analysis

- Interactive profiling
- Automated testing and customization via scripting
- Visual project management system
- Compile in the background while editing and debugging
- Multi-processor debugging
- Help on the target DSP

## PROCEDURE TO WORK ON CODE COMPOSER STUDIO

**To create the New Project**

Project → New (File Name. pjt , Eg: **Vectors.pjt**)

**To Create a Source file**

File → New → Type the code (Save & enter file name, Eg: **sum.c or sum.asm if you have written the code in assembly**).

**To Add Source files to Project**

Project → Add files to Project → **sum.c or sum.asm( for assembly code).**

**To Add rts.lib file & hello.cmd:**

Project → Add files to Project →rts_ext.lib

Library files: rts_ext.lib(Path: c:\ ti\c5400\cgtools\lib\rts_ext.lib)

Note: Select Object & Library in(*.o,*.l) in Type of files

Project → Add files to Project →hello.cmd

CMD file – Which is common for all non real time programs.

(Path: c:\ti\tutorial\dsk5416\hello1\hello.cmd)

Note: Select Linker Command file(*.cmd) in Type of files

**To Enable –mf option:**

Project→ Build Options → Advanced (in Category) →

–Use Far Calls (- mf) (C548 and higher).

**Compile:**

**To Compile:** Project → Compile

**To Rebuild:** Project → rebuild,
Which will create the final .out executable file.(Eg. Vectors.out).

**Procedure to Load and Run program:**

Load the program to DSK: File → Load program → Vectors. out

To Execute project: Debug → Run.

**FILE EXTENSIONS**

While using Code Composer Studio, you work with files that have the following file-naming conventions:

- **project.mak**. Project file used by Code Composer Studio to define a
- project and build a program
- **program.c**. C program source file(s)
- **program.asm**. Assembly program source file(s)
- **filename.h**. Header files for C programs, including header files for
- DSP/BIOS API modules
- **filename.lib**. Library files
- **project.cmd**. Linker command files
- **program.obj**. Object files compiled or assembled from your source files
- **program.out**. An executable program for the target (fully compiled,
- assembled, and linked). You can load and run this program with Code
- Composer Studio.
- **project.wks**. Workspace file used by Code Composer Studio to store
- information about your environment settings
- **program.cdb**. Configuration database file created within Code Composer
- Studio. This file is required for applications that use the DSP/BIOS API,
- and is optional for other applications. The following files are also generated when you save a configuration file:
    - **programcfg.cmd**. Linker command file
    - **programcfg.h54**. Header file
    - **programcfg.s54**. Assembly source file

**SAMPLE PROGRAMS TO WORK ON CODE COMPOSER STUDIO.**

**sum.c**

```
# include<stdio.h>
main()
{
int i=0;
i++;
printf("%d",i);
}
```

Create a new project and type the above program in the editor , save the file as sum.c and add the source file to the project, compile and build the project . Load the .out file to the target processor and click run to see the result in the output window.

## ASSEMBLY LANGUAGE PROGRAMMING IN C54X

The Assembly language programming is done using the following assembler directives

**.text section**

It consists of the assembly program, which has to be translated in object code by the assembler, and it is loaded into the program memory for execution.

**.data section.**

It consists of the constants and variables which are intialized and are loaded into the data memory area. The origin of the data memory is given in the .cmd file.

**.bss section.**

It is used to reserve a block of memory which is uninitalized.

**.mmregs**

It permits to use the memory mapped registers to be referred using the names such as AR0, SP etc.

**.include "XX"**

It informs the assembler to insert a list of instructions in the file "XX" while assembling.

**.end**

To end the assembly language program.

**.equ**

To equate a symbol with a constant value.

**.word x,y, ….z**

It reserves 16 bit locations and initialises them with values x,y,….z

## sum.asm

```
                .include "5416_IV.asm"
                .word 0003h,0004h
                .data
                .text
begin           STM #1000h,AR1 {memory location of P}
                STM #1001h,AR2  {memory location of Q}
                STM #1500h,AR1{Result is stored in the memory}
                LD *AR1,A          {This accumulator loads that accumulator A
                                      with the data in the memory 1000h}
                LD *AR2,B          {{This accumulator loads that accumulator A
                                      with the data in the memory 1001h}
                ADD A,0,B           {accumulator A is added eith accumulator B
                                     and output will be in accumulator B}
                STL B,*AR3          (output will be stored in the memory location
```

1500h}

.end

**ADDING A PROBE POINT FOR FILE I/O**

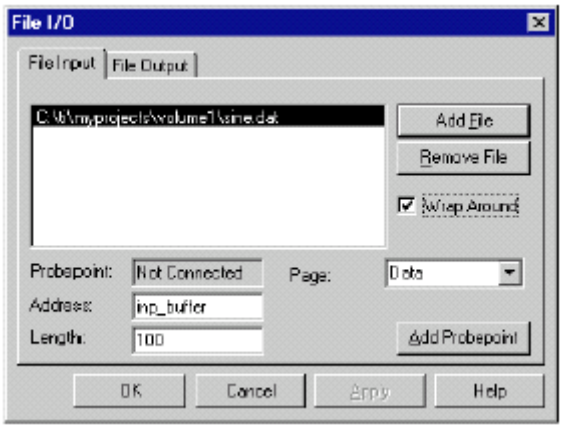
In this section, you add a Probe Point, which reads data from a file on your PC. Probe Points are a useful tool for algorithm development. You can use them in the following ways:

- To transfer input data from a file on the host PC to a buffer on the target for use by the algorithm
- To transfer output data from a buffer on the target to a file on the host PC for analysis
- To update a window, such as a graph, with data
- Probe Points are similar to breakpoints in that they both halt the target to perform their action. However, Probe Points differ from breakpoints in the following ways:
- Probe Points halt the target momentarily, perform a single action, and resume target execution.
- Breakpoints halt the CPU until execution is manually resumed and cause all open windows to be updated.
- Probe Points permit automatic file input or output to be performed; breakpoints do not.

This chapter shows how to use a Probe Point to transfer the contents of a PC file to the target for use as test data. It also uses a breakpoint to update all the open windows when the Probe Point is reached. These windows include graphs of the input and output data.

1) Put your cursor in the line of the main function, say: dataIO(); The dataIO function acts as a placeholder.
2) Click the  (Toggle Probe Point) toolbar button. The line is highlighted in blue.
3) Choose File→File I/O. The File I/O dialog appears so that you can select input and output files.
4) In the File Input tab, click Add File.
5) Choose the "xx".dat file. Notice that you can select the format of the data in the Files of Type box. The "xx".dat file contains hex values for a sine waveform.
6) Click Open to add this file to the list in the File I/O dialog. A control window for the sine.dat file appears. (It may be covered by the Code Composer Studio window.) Later, when you run the program, you can use this window to start, stop, rewind, or fast forward within the data file.


7) In the File I/O dialog, change the Address to inp_buffer and the Length to 100. Also, put a check mark in the Wrap Around box.




The Address field specifies where the data from the file is to be placed.

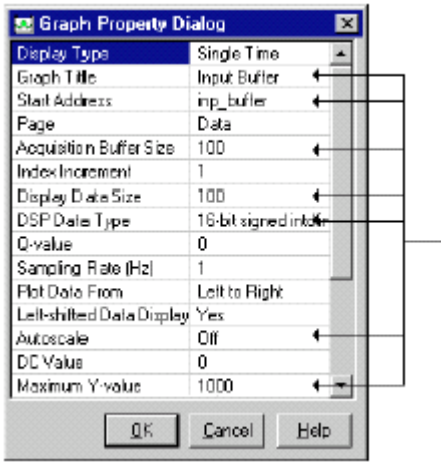
The Length field specifies how many samples from the data file are read each time the Probe Point is reached. You use 100 because that is the value set for the BUFSIZE constant in volume.h (0x64).

The Wrap Around option causes Code Composer Studio to start reading from the beginning of the file when it reaches the end of the file. This allows the data file to be treated as a continuous stream of data even though it contains only 1000 values and 100 values are read each time the Probe Point is reached.

**DISPLAYING GRAPHS**

Code Composer Studio provides a variety of ways to graph data processed by your program. In this example, you view a signal plotted against time. You open the graphs in this section and run the program in the next section.

1) Choose View→Graph→Time/Frequency.
2) In the Graph Property Dialog, change the Graph Title, Start Address, Acquisition Buffer Size, Display Data Size, DSP Data Type, Autoscale, and Maximum Y-value properties to the values shown here. Scroll down or resize the dialog box to see all the properties.




3) Click OK. A graph window for the Input Buffer appears.

4) Right-click on the Input Buffer window and choose Clear Display from the pop-up menu.

5) Choose View→Graph→Time/Frequency again.

6) This time, change the Graph Title to Output Buffer and the Start Address to out_buffer. All the other settings are correct.

7) Click OK to display the graph window for the Output Buffer. Right-click on the graph window and choose Clear Display from the pop-up menu.

## ANIMATING THE PROGRAM AND GRAPHS

So far, you have placed a Probe Point, which temporarily halts the target, transfers data from the host PC to the target, and resumes execution of the target application. However, the Probe Point does not cause the graphs to be updated. In this section, you create a breakpoint that causes the graphs to be updated and use the Animate command to resume execution automatically after the breakpoint is reached.

1) In the Volume.c window, put your cursor in the line that calls dataIO.

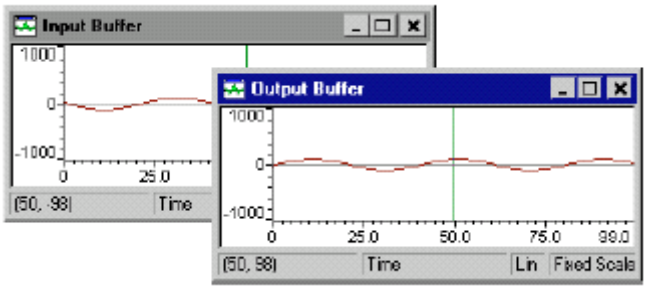
2) Click the (Toggle Breakpoint) toolbar button or press F9. The line is highlighted in both magenta and blue (unless you changed either color using Option→Color) to indicate that both a breakpoint and a Probe Point are set on this line. You put the breakpoint on the same line as the Probe Point so that the target is halted only once to perform both operations—transferring the data and updating the graphs.

3) Arrange the windows so that you can see both graphs.

3) Click the (Animate) toolbar button or press F12 to run the program. The Animate command is similar to the Run command. It causes the target application to run until it reaches a breakpoint. The target is then halted and the windows are updated. However, unlike the Run command, the Animate command then resumes execution until it reaches

another breakpoint. This process continues until the target is manually halted. Think of the Animate command as a run-break-continue process.

4) Notice that each graph contains 2.5 sine waves and the signs are reversed in these graphs. Each time the Probe Point is reached, Code Composer Studio gets 100 values from the sine.dat file and writes them to the inp_buffer address. The signs are reversed because the input buffer contains the values just read from sine.dat, while the output buffer contains the last set of values processed by the processing function.

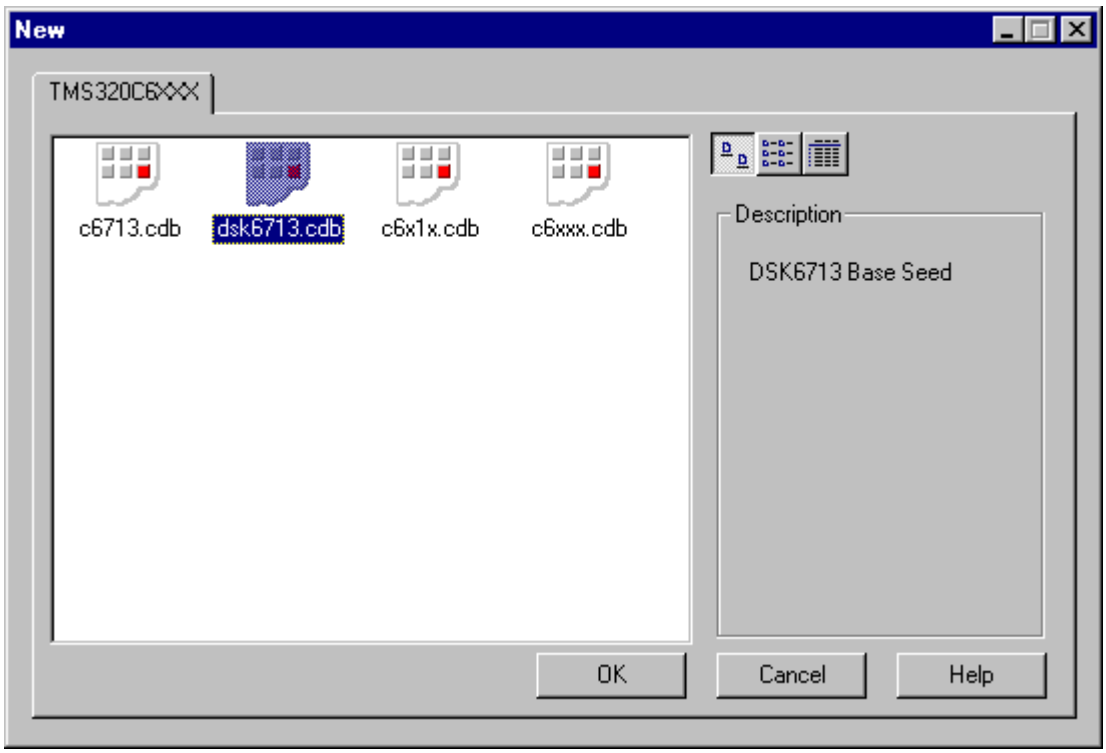



**Note: Target Halts at Probe Points**

Code Composer Studio briefly halts the target whenever it reaches a Probe Point. Therefore, the target application may not meet real-time deadlines if you are using Probe Points. At this stage of development, you are testing the algorithm. Later, you analyze real-time behavior using RTDX and DSP/BIOS.

**PROCEDURE FOR REAL TIME PROGRAMS:**

1. Connect CRO to the Socket Provided for **LINE OUT**.
2. Connect a Signal Generator to the **LINE IN** Socket.
3. Switch on the Signal Generator with a sine wave of frequency 500 Hz. and Vp-p=1.5v
4. Now Switch on the DSK and Bring Up Code Composer Studio on the PC.
5. Create a new project with name **codec.pjt.( c:\ccstudio_v3.1\myprojects\codec)**
6. From the **File** Menu ➔ **new** ➔ **DSP/BIOS Configuration** ➔select "**dsk6713.cdb**" and **open.**
7. Save it as "**xyz.cdb**"




7. Add "**xyz.cdb**" to the current project.
8. Add the given "**codec.c**" and "**FIR.c (or IIR.c)"** file to the current project

   **(copy the source files to the codec.pjt folder)**
9. Add the library file "**dsk6713bsl.lib**" to the current project

   **Path** → "**C:\CCStudio\C6000\dsk6713\lib\dsk6713bsl.lib**"
10. Build, Load and Run the program.
11. You can notice the input signal of 500 Hz. appearing on the CRO verifying the codec configuration.
12. You can also pass an audio input and hear the output signal through the speakers.
13. You can also vary the sampling frequency using the **DSK6713_AIC23_setFreq** Function in the "**codec.c**" file and repeat the above steps.

**RESULT:**

Thus the usage of Code Composer studio was studied and the sample programs were run and the results were seen.

| Exp. No.: | ADDITION, SUBTRACTION, MULTIPLICATION AND DIVISION USING VARIOUS ADDRESSING MODES ON TMS320C5416 |
|---|---|
| Date: | |

**AIM**

To write assembly language programs in TMS320C54XX DSP processor to add, subtract, multiply and divide two numbers using the processor instruction and using various addressing modes to get the data from memory.

**APPARATUS REQUIRED**

| S.NO | ITEM | Q.TY |
|---|---|---|
| 1 | TMS320VC5416 DSK Development Board | 1 |
| 2. | PC with Code Composer Studio IDE | 1 |
| 3 | USB Cable | 1 |
| 4 | +5V Universal Power Supply | 1 |
| 5 | AC Power cord | 1 |

**(i) ADDITION USING INDIRECT ADDRESSING**

In Indirect addressing any location in the 64 K word data space can be accessed via a 16-bit address. Indirect addressing uses the auxiliary registers (AR's) to point out a location in data or program memory. Using the pointer location the data can be read or write using LD or ST instructions.

**ALGORITHM:**

**Step 1:** Store any auxiliary register say AR1 with any address location for example 1000h
**Step 2:** Store another auxiliary register say AR2 with another address location for example ` 1001h.
**Step 3:** Store another auxiliary register say AR3 with address location for example 1500h
**Step 4:** Load the first data from the first address location to accumulator A using indirect addressing.
**Step 5:** Load the second data from the second address location to accumulator B using indirect addressing.
**Step 6:**Add the two data in the accumulators A and B.
**Step 7:** Store the output in the accumulator to the memory location addressed by the auxillary register.
**Step 8:** End the program.

**PROGRAM FOR ADDITION USING INDIRECT ADDRESSING:**

| LABEL | MNEMONICS | COMMENTS |
|---|---|---|
| | .include<br>"5416_IV.asm"<br><br>.word<br>0003h, 0004h<br>.data<br>.text | |
| Begin: | | **Begin the program** |
| | STM #1000h,AR1 | Memory location of P |
| | STM #1001h,AR2 | Memory location of Q |
| | STM #1500h,AR3 | Memory location of the result. |
| | LD *AR1, A | The accumulator A is loaded with the data in the memory location pointed by AR1. |
| | LD *AR2, B | The accumulator B is loaded with the data in the memory location pointed byAR2. |
| | ADDC A, 0, B | Accumulator A is added with accumulator B and the output will be in accumulator B |
| | STL B, *AR3 | The result in accumulator B is stored to the memory location pointed by AR3. |
| | . end | **End the program** |

**(ii) SUBTRACTION USING IMMEDIATE ADDRESSING:**

Immediate addressing uses the instruction to encode a fixed value. The operand required for the instruction is specified by the instruction word itself. The number of bits of the operand may be 3,4,8 or 9 in short addressing mode or 16-bit number in case of long addressing mode. LD instructions can be used in immediate addressing

**ALGORITHM:**

**Step 1**: Load the first operand in the A register using immediate addressing.
**Step 2:** Load the second operand in the B register.
**Step 3:** Subtract the values in A and B register.
**Step 4:** Store the memory address in Auxiliary register using immediate addressing.
**Step 5:** Move the result stored in B register to the location pointed by the Auxiliary register.
**Step 6:** End the program.

**PROGRAM FOR SUBTRACTION USING IMMEDIATE ADDRESSING:**

| LABEL | MNEMONICS | COMMENTS |
|---|---|---|
| | .include<br>"5416_IV.asm"<br><br>.data<br>.text | |

| | | |
|---|---|---|
| Begin: | STM #1500h,AR3 | **Begin the program**<br>Memory location of the result. |
| | LD A, #0010h | Loads the accumulator A with the immediate value 0010h. |
| | LD B, #0004h | Loads the accumulator A with the immediate value 0004h. |
| | SUB A,0,B | Accumulator A is subtracted with accumulator B and the output will be in accumulator B |
| | STL B, *AR3 | The result in accumulator B is stored to the memory location pointed by AR3. |
| | . end | **End the program** |

**(iii) MULTIPLICATION USING DIRECT ADDRESSING:**

In direct addressing the instruction carries the 7-bit data memory address offset in the instruction itself. The offset address is added with the DP (Data pointer) or SP (Stack Pointer) to get the 16-bit data memory address. The offset address can be added with either DP or SP depending on the CPL (Compiler mode bit) in the Status register ST1.When CPL=0, the offset is concatenated with 9 bit DP to generate 16 bit Data memory address. When CPL=1 the offset is concatenated with SP to generate the 16 bit Data memory address.

**ALGORITHM:**

**Step 1:** Reset the Compiler mode bit to zero to use the DP register to generate data memory address.
**Step 2:** Initialise the DP register with page number.
**Step 3:** Load the value from the location pointed by DP and offset in the Accumulator A.
**Step 4:** Load the second value from the location pointed by DP and offset in the Accumulator B.
**Step 5:** Multiply both the values in A & B.
**Step 6:** Store the value in the accumulator to the memory location pointed by the offset.
**Step 7:** End the program.

**PROGRAM FOR MULTIPLICATION USING DIRECT ADDRESSING:**

| LABEL | MNEMONICS | COMMENTS |
|---|---|---|
| | .include<br>"5416_IV.asm"<br><br>.word<br>0003h, 0004h<br>.data<br>.text | |
| Begin: | | **Begin the program** |

| | | |
|---|---|---|
| | RSBX CPL | Compiler mode bit CPL is reset to zero. |
| | LD #20h, DP | Loads the DP register with the value 20h that corresponds to the page address 1000h |
| | LD 01h,A | Loads the value from address location 1001h to the accumulator A |
| | LD 02h, 2, B | Loads the second value from address location 1002h to the accumulator B left shifted by 2 bits. |
| | MPY A, 0, B | Multiplies accumulator A & B. |
| | STL B, 15h | The result in accumulator B is stored to the memory location pointed by AR3. |
| | . end | End the program. |

**(iv) DIVISTION USING MEMORY MAPPED REGISTER ADDRESSING:**

In memory mapped register addressing the current value of data page pointer (DP) or Stack pointer (SP) are not modified. It works for both direct as well as indirect addressing. The memory mapped registers are used for addressing.

**ALGORITHM:**

**Step 1:** Load the memory mapped register with the data address of the first operand.
**Step 2:** Load the value pointed by the memory mapped register to the accumulator.
**Step 3:** Increment the memory mapped register and point to the second operand.
**Step 4:** Subtract the second operand from the first operand until it becomes negative.
**Step 5:** Increment the

**PROGRAM FOR DIVISION USING MEMORY MAPPED REGISTER ADDRESSING:**

| LABEL | MNEMONICS | COMMENTS |
|---|---|---|
| | .include "5416_IV.asm"<br><br>.word 0003h, 0004h<br>.data<br>.text | |
| Begin: | | **Begin the program** |
| | STM #1000h,AR1 | Memory location of first operand. |
| | STM #1001h,AR2 | Memory location of second operand. |
| | LD *AR1, A | The accumulator A is loaded with the data in the memory location pointed by AR1. |
| | RPT #15 | The next instruction is repeated for 16 times . |

| | | |
|---|---|---|
| | SUBC *AR2,A | Subtract conditionally and the instruction is repeated for 16 times. |
| | STL A, *AR1+<br>STL A, *AR1 | The result in accumulator A is stored to the memory location pointed by AR1. |
| | . end | **End the program** |

**(v) FINDING MAXIMUM OF A SET OF NUMBERS USING CIRCULAR ADDRESSING:**

**ALGORITHM:**

**Step 1:** Initialise a group of values in the data memory.
**Step 2:** Load the first value in the accumulator.
**Step 3:** Set the buffer size register to the maximum number of values.
**Step 4:** Compare the value in the accumulator with the second value in memory using circular addressing.
**Step 5:** If the value is lesser than the first value move the value to the previous location.
**Step 6:** Repeat the steps 4 and 5 until all the data get sorted.
**Step 7:** Store the maximum value to the memory.
**Step 8:** End the program.

**PROGRAM FOR FINDING MAXIMUM:**

| LABEL | MNEMONICS | COMMENTS |
|---|---|---|
| | .include<br>"5416_IV.asm"<br><br>.word<br>0003h, 0004h, 0006h,<br>0001h, 0005h.<br>.data<br>.text | |
| Begin: | | **Begin the program** |
| | STM #1000h,AR1 | Memory location of first value. |
| | STM #1500h,AR3 | Memory location of the result. |
| | STM #5h,BK | Set the circular buffer size register to the number of values in the memory. |
| | LD *AR1+%, A | The accumulator A is loaded with the data in the memory location pointed by AR1.AR1 is incremented by 1. |
| | LD *AR2, B | The accumulator B is loaded with the data in the memory location pointed byAR2. |
| | ADDC A, 0, B | Accumulator A is added with accumulator B and the output will be in accumulator B |
| | STL B, *AR3 | The result in accumulator B is stored to the memory location pointed by AR3. |
| | . end | |

| | | |
|---|---|---|
| | | **End the program** |

**MULTIPLICTION**

```
                .include "5416_IV.asm"
                .word 0003h,0004h
                .data
                .text
begin           STM #1000h,AR1 {memory location of P}
                STM #1001h,AR2  {memory location of Q}
                STM #1500h,AR1{Result is stored in the memory}
                MPY *AR1,*AR2,B   {accumulator A is multiplied with
                                   accumulator B and output will be in
                                   accumulator B}
                STL B,*AR3        (output will be stored in the memory location
                                  1500h}
.end
```

**DIVISION**

```
                .include "5416_IV.asm"
                .word 0003h,0004h
                .data
                .text
begin           STM #1000h,AR1 {memory location of P}
                STM #1001h,AR2  {memory location of Q}
                LD *AR1+,A          {This accumulator loads that accumulator A
                                     with the data in the memory 1000h}
                RPT#15
                SUBC *AR0,A
                STL A,*AR1+(output will be stored in the memory location
                                  1500h}
                STL A,*AR1          {Store remainder}
.end
```

**RESULT:**

Thus the addition, multiplication and division operation is performed using TMS320C5416

**EXP NO:**

## FIR FILER USING TMS320C5416

**AIM**

To design a FIR Low Pass filter (using Kaiser Window) with cutoff frequency 1 K Hz

**APPARATUS REQUIRED**

TMS320C5416 Processor with a PC

**Algorithm:**

Step 1 declare an array called input buffer
Step 2 declare the coefficients in data memory
Step 3 declare an array called delay buffer
Step 4 declare an array called output buffer
Step 5 point the auxiliary register to corresponding arrays.
Step 6 get the value from the input buffer and transfer it to the first location
Of the delay buffer and increment the input buffer by 1
Step 7 make the delay register to point it to the last location by using MAR
Step 8 multiply the delay buffer with coefficient and accumulate it with previous
Output.
Step 9 make the pointer point to the starting location of delay buffer
Step 10 store the output in the buffer
Step 11 repeat Step6 to Step10 128 times.

**PROGRAM:**

Starting address :1000h
;Input address : 1600h
;Output address : 1700h

```
                ;FIR application Program
                ;Filter order 9
                ;Cutoff Frequency  1KHz
;---------------------------------------------------------------------------
                .include "5416_IV.asm"
                .data
COEFF           .word   086eh,0b9eh,0e5fh,1064h,1176h,1064h,0e5fh,0b9eh,086eh
;Filter Co-efficients in data
```

```
;memory
            .text

start       LD    #COEFF,DP              ;Variable Declaration
            RSBX  INTM
            LD    #022Bh,0,A
            STLM  A, PMST
            SSBX  SXM
            RSBX  FRCT
            RSBX  OVM

            STM   #150,BK              ;Circular Buffer for Input and Output
            STM   #1600h,AR5           ;Input Buffer Starts at 1600h
            STM   #1700h,AR6           ;Output Buffer Starts at 1700h

            LD    #0h,A
            STM   #1900h,AR3           ;Temporary Buffer Initialization
            RPT   #10
            STL   A,*AR3+
            STM   #1900h,AR3


            MACD  *AR3-,COEFF,A

            SFTA  A,-15                ;Shifting the output to Lower order
            STLM  A,McBSP0_DXR1        ;o/p for R Channel
            STLM  A,McBSP0_DXR2        ;o/p for L Channel

            STL   A,0,*AR6+%           ;Output is stored at 1700h
            MAR   *AR5+%               ;Modify the Input Buffer
            RETE
```

**RESULT:**

Thus the FIR LPF filter with cut off frequency 1 kHz is generated using TMS320C5416

**EXP NO: 13**

## FAST FOURIER TRANSFORM USING TMS320C5416

**Aim**

To find the FFT for the signal this contains 128 samples

**Algorithm:**

Step 1 Declare four buffers for real input, real exponent, imaginary exponent And imaginary input
Step 2 Scale the input to ovoid the over flow during manipulation.
Step 3 Declare three Counters for stages, Groups and butterflies.
Step 4 implement the Fast fourier transform formula on the input signal
Step 5 store the output in the output buffer
Step 6 decrement the butterfly counter
Step 7 if it is not zero repeat the step 4

Step 8 If the counter is zero modify the exponent value and decrement the group Counter
Steps 9 if the group counter is zero repeat the step 4
Step 10 if it zero then multiply the butterfly counter by 2 and divide the group Counter by 2
Step 11 Decrement the stage counter. if it is not zero repeat step 4
Step 12 if counter is zero stop the execution.

**PROGRAM:**

Starting address : 0700h
;Input address :1600h
;Output address :1700h

```
                .include "5416_IV.asm"
                .def start
                .data
bpole           .word  97e3h,154fh              ;IIR Filter Co-efficients
azero           .word  0b4ch,1698h,0b4ch
xin             .word   0,0
xout            .word   0
yin             .word   0
S1              .word   00
E               .word  45h
```

```
                .text

start           LD      #bpole,DP               ;Variable Declaration
                RSBX    INTM
                LD      #022Bh,0,A
                STLM    A,PMST
        RSBX    INTM

                LD      #02Fh,0,A
                STLM    A,IMR

                STM     #0h,McBSP0_DXR1
                STM     #0h,McBSP0_DXR2

                STM     #0007h,GPIOCR
                STM     #0003h,GPIOSR

                STM     #SPCR2,McBSP2_SPSA
                STM     #00E1h,McBSP2_SPSD      ;Mclk

                NOP
                STM     #0007h,GPIOSR

                STM     #SPCR2,McBSP0_SPSA
                STM     #00E1h,McBSP0_SPSD      ;Sclk & Fs
                SSBX    SXM
                RSBX    FRCT
                RSBX    OVM

                STM   #128,BK               ;Circular Buffer for Input and Output
                STM   #1600h,AR4            ;Input Buffer Starts at 1600h
                STM   #1700h,AR1            ;Output Buffer Starts at 1700h
                STM   #1400h,AR5            ;IIR Filter Output
                STM   #1500h,AR6             ;Zero Output Buffer

WAIT             NOP
                NOP
                NOP
                NOP
                NOP
                B       WAIT
_XINT0_ISR

              LDM     McBSP0_DRR1,A         ;R Channel (Input Sample From CODEC)
              LDM     McBSP0_DRR2,A         ;R Channel (Input Sample From CODEC)
               STM     #1800h,AR3           ;Pole Temporary Buffer
               STM     #1300h,AR7            ;Pole Output Buffer
               STL     A,0,xin
               NOP
               NOP
               LD              xin,A
               STL      A,0,*AR4+%
```

```
NOP
NOP
STM    xout,AR2
RPT     #02h                    ;Multiplication of Input with zeros
MACD   *AR2-,azero,A
SFTA    A,-15
STL     A,0,*AR6                ;Zero Output

MVDD   *AR5,*AR3+               ;Transfer IIR Output to Temp Buffer
LD      #bpole,DP
RPT     #01h
MACD   *AR3-,bpole,A            ;Multiplication of Output with Poles

SFTA    A,-15
STL     A,0,*AR7                ;Pole Output
NOP
NOP
LD      *AR6,A
LD      *AR7,B
SUB     B,0,A                   ;Zero Output - Pole Output
STL     A,0,*AR5

STL     A,0,*AR1+%
STLM    A,McBSP0_DXR1           ;o/p for R Channel
STLM    A,McBSP0_DXR2           ;o/p for L Channel


   RETE
```

**RESULT:**

Thus the FFT for the signal of 128 samples is obtained using TMS320C5416

**EXP NO: 14**

## SAMPLING USING TMS320C5416

**AIM:**

To sample the signal at different time intervals

**PROGRAM**

```
Sampling program
;Starting address : 1000h
;DSPIK output : 1600h

                .include "5416_IV.asm"
                .data
                .text

.start          RSBX    INTM
                LD      #022Bh,0,A
                STLM    A,PMST
                RSBX    INTM
                LD      #02Fh,0,A
                STLM    A,IMR

                STM     #0h,McBSP0_DXR1
                STM     #0h,McBSP0_DXR2

                STM     #0007h,GPIOCR
                STM     #0003h,GPIOSR

                STM     #SPCR2,McBSP2_SPSA
                STM     #00E1h,McBSP2_SPSD      ;Mclk
                NOP
                STM     #0007h,GPIOSR

                STM     #SPCR2,McBSP0_SPSA
                STM     #00E1h,McBSP0_SPSD      ;Sclk & Fs

;---------------------------------------------------------------------------
                STM     #128,BK
```

```
                STM    #1600h,AR1
                SSBX   SXM
WAIT            NOP
                NOP
                NOP
                B      WAIT

_XINT0_ISR

                LDM    McBSP0_DRR1,A        ;R Channel
;               LDM    McBSP0_DRR2,A        ;L Channel

                STL    A,0,*AR1+%

                STLM   A,McBSP0_DXR1        ;o/p for R Channel ;current(Y)
                STLM   A,McBSP0_DXR2        ;o/p for L Channel ;voltage(R)

                RETE
```

**RESULT:**

Thus the signal is sampled at various time intervals using TMS320C5416

## II-CYCLE EXPERIMENTS

| Exp. No.: | GENERATION OF SEQUENCES |
|---|---|
| Date: | |

**AIM:**

To generate the Unit Impulse signal, Unit Step signal, Exponential signal, Sinusoidal and cosine sequence using Mat lab.

**APPARATUS REQUIRED:**

A PC with Mat lab version 6.5.

**ALGORITHM:**

**Unit Impulse signal:**

1) Enter the program in the Mat lab editor or debugger window
2) Write a program with Mat lab functions for generating a wave of
   $\delta(n) = 1, n=0$
   $= 0, n\neq 0$
3) Enter the value of n
4) Display the output using plot or stem function
5) Give the title for the program
6) Save & run the program
7) Give the input details in command window

**Unit Step signal:**

1) Enter the program in the Mat lab editor or debugger window
2) Write a program with Mat lab functions for generating a wave of
   $U(n) = 1, n\geq 0$
   $= 0, n<0$
3) Enter the value of n
4) Display the output using plot or stem function
5) Give the title for the program
6) Save & run the program
7) Give the input details in command window

**Exponential signal:**

1) Enter the program in the Mat lab editor or debugger window
2) Write a program with Mat lab functions for generating a wave of
   $y = 2\exp(a*t)$
3) Enter the value of n
4) Enter the value of a
5) Display the output using plot or stem function
6) Give the title for the program
7) Save & run the program

**Sinusoidal sequence:**

1) Enter the program in the Mat lab editor or debugger window
2) Write a program with Mat lab functions for generating a wave of
   y(n) = a *sin(2*pi*f*n)
3) Enter the value of a
4) Enter the value of f
5) Display the output using plot or stem function
6) Give the title for the program
7) Save & run the program

**Cosine sequence:**

1) Enter the program in the Mat lab editor or debugger window
2) Write a program with Mat lab functions for generating a wave of
   y(n) = a *sin(2*pi*f*n)
3) Enter the value of a
4) Enter the value of f
5) Display the output using plot or stem function
6) Give the title for the program
7) Save & run the program

**PROGRAM:**

```
clear all;
%----------------Impulse Response-----------------------------%
N=input('Enter the length of the sequence');
n=-N:N;
x=[zeros(1,N),1,zeros(1,N)];
subplot(3,2,3);
stem(n,x);
title('Unit Impulse');
xlabel('Title');
ylabel('Amplitude');
%--------------Unit Step Function-----------------------------%
e=input('Enter the length of the sequence');
f=ones(1,e);
m=0:e-1;
subplot(3,2,4);
stem(m,f);
xlabel('e');
ylabel('u(e)');
title('Unit Step FUnction');
%----------------Exponential Wave-------------------------%
b=input('Enter the length of sequence');
c=0:1:b;
g=input('Enter the value');
h=0.8*exp(g*c);
subplot(3,2,5);
stem(c,h);
ylabel('Amplitude');
xlabel('b');
title('Exponential Sequence');
```

```
%-----------------Sine wave-----------------------------%

a=input('enter the value of amplitude');
f=input('enter the value of frequency');
t=0:0.001:0.01;
z=a*sin(2*pi*f*t);
subplot(3,2,1);
plot(t,z);
xlabel('Timeperiod');
ylabel('Amplitude');
title('Sine wave');
%-----------------cos wave-----------------------------%
d=a*cos(2*pi*f*t);
subplot(3,2,2);
plot(t,d);
xlabel('Time perios');
ylabel('Amplitude');
title('cos wave');
```

**COMMANDS:**

1) **INPUT** Prompt for user input.
   R = INPUT('How many apples') gives the user the prompt in the text string and then waits for input from the keyboard.
   The input can be any MATLAB expression, which is evaluated, using the variables in the current workspace, and the result returned in R. If the user presses the return key without entering anything, INPUT returns an empty matrix.
   R = INPUT('What is your name','s') gives the prompt in the text string and waits for character string input. The typed input is not evaluated; the characters are simply returned as a MATLAB string.

2) **PLOT** Linear plot.
   PLOT(X,Y) plots vector Y versus vector X. If X or Y is a matrix, then the vector is plotted versus the rows or columns of the matrix, whichever line up. If X is a scalar and Y is a vector, length(Y) disconnected points are plotted.
   **PLOT(Y)** plots the columns of Y versus their index.
   If Y is complex, PLOT(Y) is equivalent to PLOT(real(Y),imag(Y)).
   In all other uses of PLOT, the imaginary part is ignored.

3) **STEM** Discrete sequence or "stem" plot.
   STEM(Y) plots the data sequence Y as stems from the x axis terminated with circles for the data value.
   STEM(X,Y) plots the data sequence Y at the values specified in X.

4) **SUBPLOT** Create axes in tiled positions.

H = SUBPLOT(m,n,p), or SUBPLOT(mnp), breaks the Figure window into an m-by-n matrix of small axes, selects the p-th axes for for the current plot, and returns the axis handle. The axes are counted along the top row of the Figure window, then the second row, etc.

5) **TITLE** Graph title.
   TITLE('text') adds text at the top of the current axis.

6) **XLABEL** X-axis label.
   XLABEL('text') adds text beside the X-axis on the current axis.

7) **YLABEL** Y-axis label.
   YLABEL('text') adds text beside the Y-axis on the current axis.

8) **ZEROS** Zeros array.
   ZEROS(N) is an N-by-N matrix of zeros.
   ZEROS(M,N) or ZEROS([M,N]) is an M-by-N matrix of zeros.

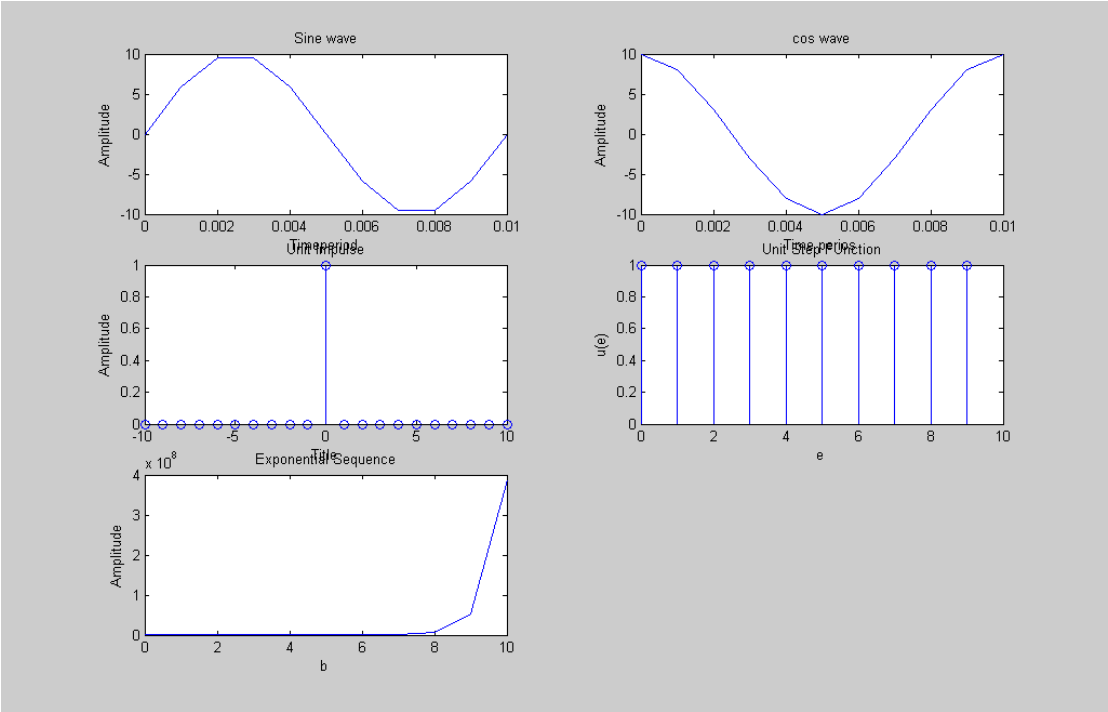
9) **EXP** Exponential.
   EXP(X) is the exponential of the elements of X, e to the X.
   For complex Z=X+i*Y, EXP(Z) = EXP(X)*(COS(Y)+i*SIN(Y)).

10) **SIN** Sine.
    SIN(X) is the sine of the elements of X.

11) **COS** Cosine.
    COS(X) is the cosine of the elements of X.

**OUTPUT:**

**Enter the length of the sequence10**
**Enter the length of the sequence10**
**Enter the length of sequence10**
**Enter the value2**
**enter the value of amplitude10**
**enter the value of frequency100**




**RESULT:**

The Unit Impulse signal, Unit Step signal, Exponential signal, Sinusoidal and cosine sequence were generated (above the input values) using Mat lab.

## EXP NO: 2

## LINEAR AND CIRCULAR CONVOLUTION

## AIM:

To write a program to compute the circular and linear convolution of the following sequence.

$x_1(n) = [2,1,2,1]$
$x_2(n) = [1,1,-1,1]$

## APPARATUS REQUIRED:

A PC with Mat lab version 6.5.

## ALGORITHM:

### LINEAR CONVOLUTION:

1) Enter the 1st and 2nd sequence of convolution.
2) Find the linear convolution using MATLAB function.
3) Enter the length of sequence.
4) Get the intervals of 'n'.
5) Display the o/p using stem function.

### CIRCULAR CONVOLUTION:

1) Enter the 1st and 2nd sequence of convolution.
2) Find the circular convolution using MATLAB function.
3) Enter the length of sequence.
4) Get intervals of 'n'.
5) Display o/p using stem function.

## PROGRAM:

```
%----------Linear Convolution----------%
clear all;
a=input('Enter the first sequence');
b=input('Enter the second sequence');
c=conv(a,b);
m=length(c)-1;
n=0:1:m;
disp(c);
stem(n,c);
xlabel('Convolution ');
ylabel('Sequence');
title('Linear Convolution');

%----------Circular Convolution----------%
title('Circular convolution');
x1=[2,1,2,1];
x2=[1,1,-1,1];
a=fft(x1);
b=fft(x2);
```

```
z=ifft(a.*b);
disp(z);
figure(1);
subplot(2,2,1);
stem(x1);
title('FFT OF x1');
xlabel('sequence');
ylabel('Convolution');
subplot(2,2,2);
stem(x2);
title('FFT OF x2');
xlabel('sequence');
ylabel('Convolution');
subplot(2,2,3);
stem(z);
title('FFT OF Sequence 1 and 2');
xlabel('Sequence');
ylabel('Convolution');
```

**COMMANDS:**

1) **FFT** Discrete Fourier transform.
   FFT(X) is the discrete Fourier transform (DFT) of vector X. For matrices, the FFT operation is applied to each column. For N-D arrays, the FFT operation operates on the first non-singleton dimension.FFT(X,N) is the N-point FFT, padded with zeros if X has less than N points and truncated if it has more.

2) **CONV** Convolution and polynomial multiplication.
   C = CONV(A, B) convolves vectors A and B. The resulting vector is length LENGTH(A)+LENGTH(B)-1.
   If A and B are vectors of polynomial coefficients, convolving them is equivalent to multiplying the two polynomials.
3) **DISP** Display array.
   DISP(X) displays the array, without printing the array name. In all other ways it's the same as leaving the semicolon off an expression except that empty arrays don't display.
   If X is a string, the text is displayed.

4) **IFFT** Inverse discrete Fourier transform.
   IFFT(X) is the inverse discrete Fourier transform of X.
   IFFT(X,N) is the N-point inverse transform.
   IFFT(X,[],DIM) or IFFT(X,N,DIM) is the inverse discrete Fourier transform of X across the dimension DIM.

**OUTPUT:**

**Enter the first sequence : [2, 1, 2, 1]**

**Enter the second sequence: [1, 1,-1, 1]**

**Linear convolution:**

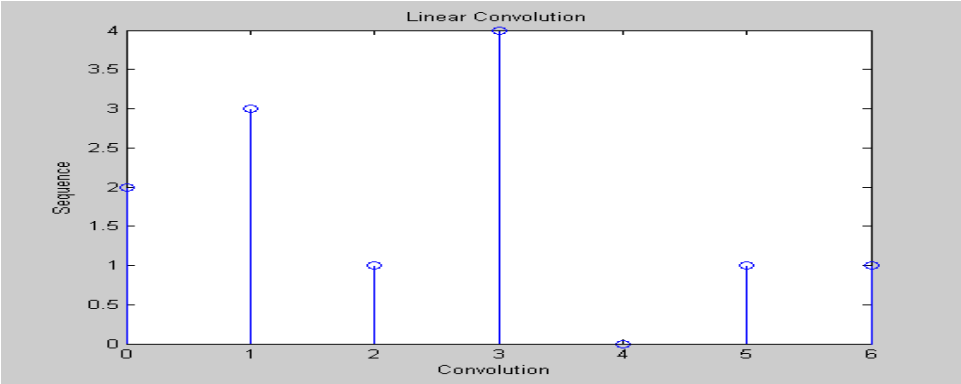
Output = [2 3 1 4 0 1 1]

**Circular convolution:**

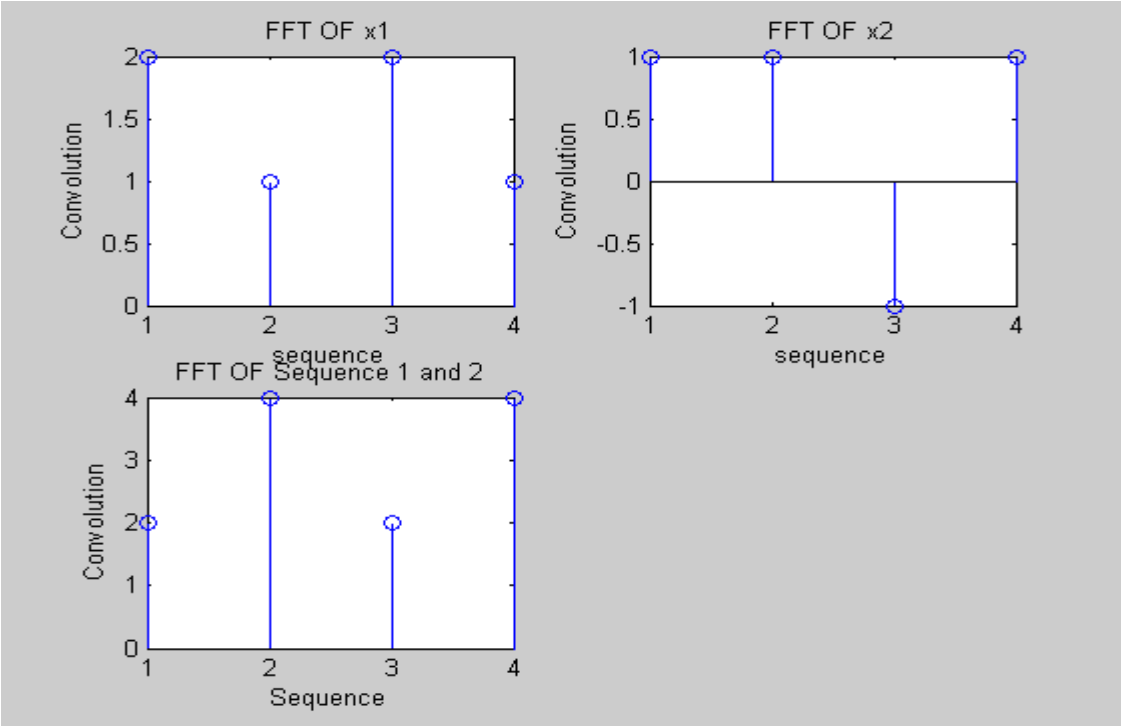
Output= [2 4 2 4]

## MODEL GRAPH:

**LINEAR CONVOLUTION:**




**CIRCULAR CONVOLUTION:**




**RESULT:**

Thus the linear & circular convolution are calculated using MAT LAB function.

**EXP NO:3**

**IMPLIMENTATION OF FFT**

**AIM:**

To perform the FFT of signal x(n) using Mat lab.

**APPARATUS REQUIRED:**

A PC with Mat lab version 6.5.

**ALGORITHM:**

1) Get the input sequence
2) Number of DFT point(m) is 8
3) Find out the FFT function using MATLAB function.
4) Display the input & outputs sequence using stem function

## PROGRAM:

```
clear all;
N=8;
m=8;
a=input('Enter the input sequence');
n=0:1:N-1;
subplot(2,2,1);
stem(n,a);
xlabel('Time Index n');
ylabel('Amplitude');
title('Sequence');
x=fft(a,m);
k=0:1:N-1;
subplot(2,2,2);
stem(k,abs(x));
ylabel('magnitude');
xlabel('Frequency Index K');
title('Magnitude of the DFT sample');
subplot(2,2,3);
stem(k,angle(x));
xlabel('Frequency Index K');
ylabel('Phase');
title('Phase of DFT sample');ylabel('Convolution');
```

**COMMANDS:**

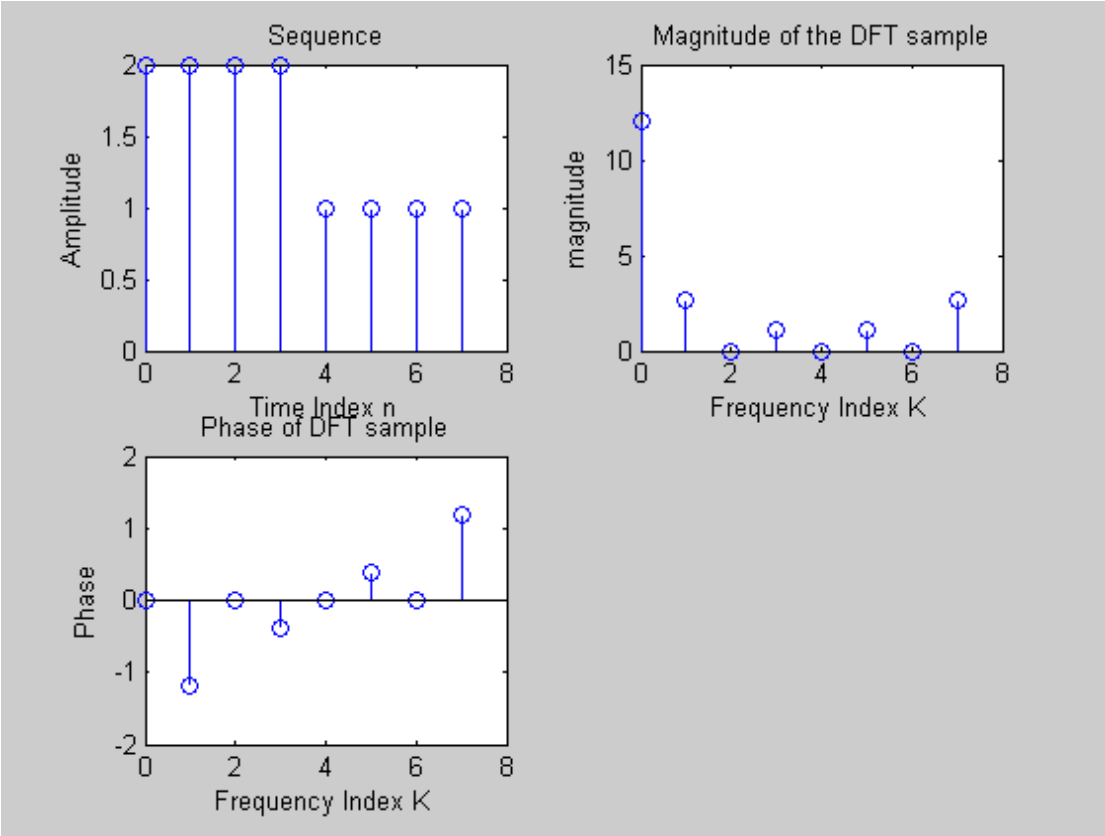
1) **FFT** Discrete Fourier transform.
   FFT(X) is the discrete Fourier transform (DFT) of vector X. For matrices, the FFT operation is applied to each column. For N-D arrays, the FFT operation operates on the first non-singleton dimension.
   FFT(X,N) is the N-point FFT, padded with zeros if X has less than N points and truncated if it has more.

2) **ABS** Absolute value.
   ABS(X) is the absolute value of the elements of X. When X is complex, ABS(X) is the complex modulus (magnitude) of the elements of X.

3) **ANGLE** Phase angle.
ANGLE(H) returns the phase angles, in radians, of a matrix with complex elements.

## OUTPUT:

**Enter the sequence [1,1,1,1,0,0,0,0]**




**RESULT:**

Thus the linear & circular convolution are calculated using MAT LAB function.

**EXP NO: 4**

**DESIGN OF IIR FILTER (HPF & LPF)**

**AIM:**

To design an analog Butterworth low pass filter and high pass filter using Mat lab for the given specification

Pass band ripple $\alpha_p = 0.4$dB

Stop band ripple $\alpha_s = 30$dB

Pass band edge frequency $f_p = 400$Hz

Stop band edge frequency $f_s = 800$Hz

Sampling frequency $f = 2000$Hz

**APPARATUS REQUIRED:**

A PC with Mat lab version 6.5.

**ALGORITHM:**

1) Get the Pass band ripple & Stop band ripples
2) Get the Pass band edge frequency & Stop band edge frequency
3) Get the Sampling frequency
4) Calculate the order of filter
5) Draw the magnitude and phase representation

## PROGRAM:

### %LPF%

```
clear all;
alphap=input('Enter the Pass band ripple ');
alphas=input('Enter the stop band ripple ');
fp=input('Enter the Pass edge frequency  ');
fs=input('Enter the stop edge frequency  ');
f=input('Enter the Sampling freq');
omp=2*(fp/f);
oms=2*(fs/f);
[n,wn]=buttord(omp,oms,alphap,alphas);
[b,a]=butter(n,wn);
w=0:0.01:pi;
[n,om]=freqz(b,a,w,'whole');
m=20*log10(abs(n));
an=angle(n);
subplot(2,1,1);
plot(om/pi,m);
grid;
ylabel('gain in db');
xlabel('Normalized freq');
subplot(2,1,2);
plot(om/pi,an);
grid;
xlabel('Normalized freq');
ylabel('Phase in radius');
title('Butterworth LPF');
```

**%HIGH PASS FILTER**

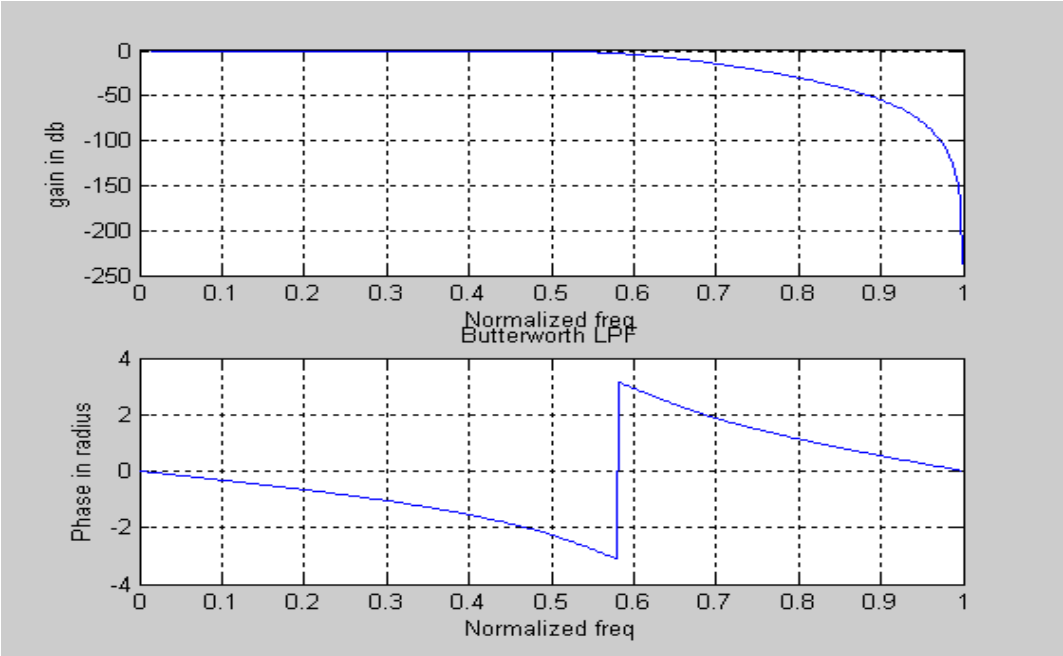
```
clear all;
alphap=input('Enter the Pass band ripple ');
```

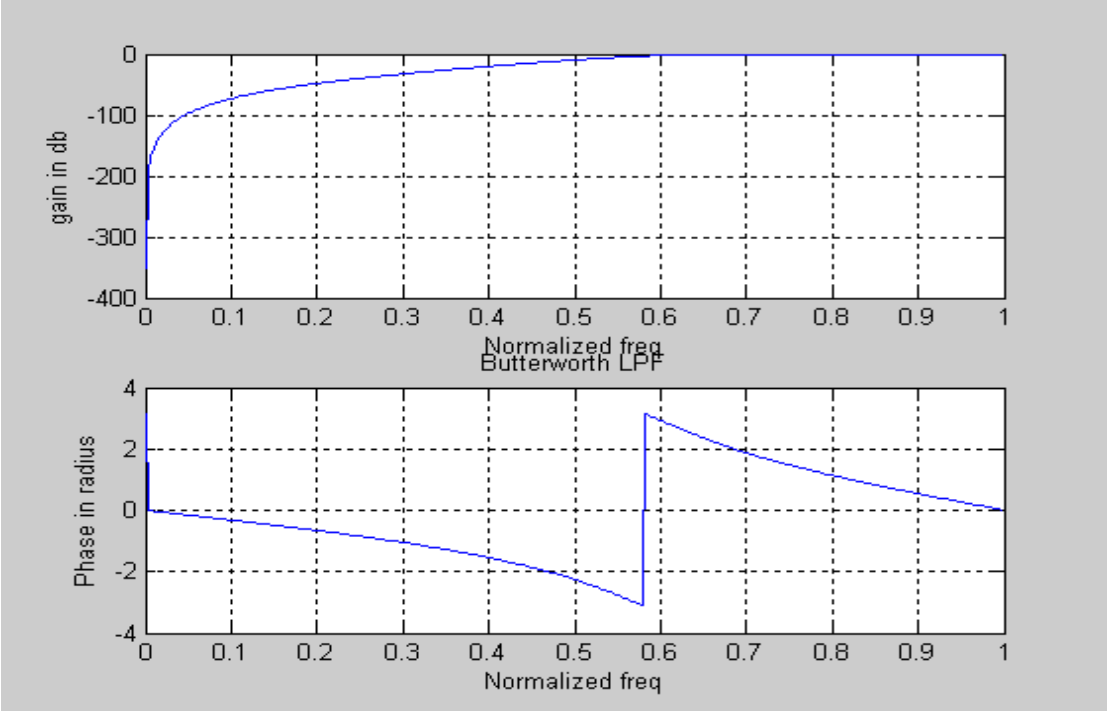
```
alphas=input('Enter the stop band ripple ');
fp=input('Enter the Pass edge frequency  ');
fs=input('Enter the stop edge frequency  ');
f=input('Enter the Sampling freq');
omp=2*(fp/f);
oms=2*(fs/f);
[n,wn]=buttord (omp,oms,alphap,alphas);
[b,a]=butter(n,wn,'HIgh');
w=0:0.01:pi;
[n,om]=freqz(b,a,w);
m=20*log10(abs(n));
an=angle(n);
subplot(2,1,1);
plot(om/pi,m);
grid;
ylabel('gain in db');
xlabel('Normalized freq');
subplot(2,1,2);
plot(om/pi,an);
grid;
xlabel('Normalized freq');
ylabel('Phase in radius');
title('Butterworth LPF');
```

**COMMANDS:**

1) [N, Wn] = BUTTORD(Wp, Ws, Rp, Rs) returns the order N of the lowest order digital Butterworth filter that loses no more than Rp dB in the passband and has at least Rs dB of attenuation in the stopband. Wp and Ws are the passband and stopband edge frequencies, normalized from 0 to 1 (where 1 corresponds to pi radians/sample). For example,
   Lowpass:  Wp = .1,  Ws = .2
   Highpass:  Wp = .2,  Ws = .1
   Bandpass:  Wp = [.2 .7], Ws = [.1 .8]
   Bandstop:  Wp = [.1 .8], Ws = [.2 .7]
   BUTTORD also returns Wn, the Butterworth natural frequency (or, the "3 dB frequency") to use with BUTTER to achieve the specifications.

## MODEL GRAPH:




**RESULT:**

An analog Butterworth low pass filter and high pass filter are design for the given specification using Mat lab.

**EXP NO: 5**

## DESIGN OF IIR FILTER (BPF & BSF)

**AIM:**

To design an Chebyshev Band Pass filter and Band Stop filter using Mat lab for the given specification

Pass band ripple $\alpha_p$ = 23B

Stop band ripple $\alpha_s$ = 46dB

Pass band edge frequency $w_p$ = 1300Hz

Stop band lower edge frequency $w_s$ = 1550Hz

Sampling frequency $f_s$ = 7800Hz

**APPARATUS REQUIRED:**

A PC with Mat lab version 6.5.

**ALGORITHM:**

1) Get the Pass band ripple & Stop band ripples
2) Get the Pass band edge frequency & Stop band edge frequency
3) Calculate the order of filter
4) Find the magnitude and phase of the BPF & BSF
5) Draw the magnitude and phase representation

## PROGRAM:

```
%--------------BPF-----------------------------------%
clear all;
format long
alphap=input('Enter the Pass band ripple ');
alphas=input('Enter the stop band ripple ');
wp=input('Enter the Pass edge frequency  ');
ws=input('Enter the stop edge frequency  ');
fs=input('Enter the Sampling frequency  ');
w1=2*wp/fs;
w2=2*ws/fs;
[n]=cheb1ord(w1,w2,alphap,alphas,'s');
wn=[w1 w2];
[b,a]=cheby1(n,alphap,wn,'bandpass','s');
w=0:0.01:pi;
[h,om]=freqs(b,a,w);
m=20*log10(abs(h));
an=angle(h);
subplot(2,1,1);
plot(om/pi,m);
ylabel('gain in db');
xlabel('Normalized freq');
subplot(2,1,2);
subplot(2,1,2);
plot(om/pi,an);
ylabel('Phase in radians');
xlabel('Normalized freq');
```

```
%--------------------------BSF-----------------------%
```

```
clear all;
format long
alphap=input('Enter the Pass band ripple ');
alphas=input('Enter the stop band ripple ');
wp=input('Enter the Pass edge frequency  ');
ws=input('Enter the stop edge frequency  ');
fs=input('Enter the Sampling frequency  ');
w1=2*wp/fs;
w2=2*ws/fs;
[n]=cheb1ord(w1,w2,alphap,alphas,'s');
wn=[w1 w2];
[b,a]=cheby1(n,alphap,wn,'stop','s');
w=0:0.01:pi;
[h,om]=freqs(b,a,w);
m=20*log10(abs(h));
an=angle(h);
subplot(2,1,1);
plot(om/pi,m);
ylabel('gain in db');
xlabel('Normalized freq');
subplot(2,1,2);
subplot(2,1,2);
plot(om/pi,an);
ylabel('Phase in radians');
xlabel('Normalized freq');
```

**COMMANDS:**

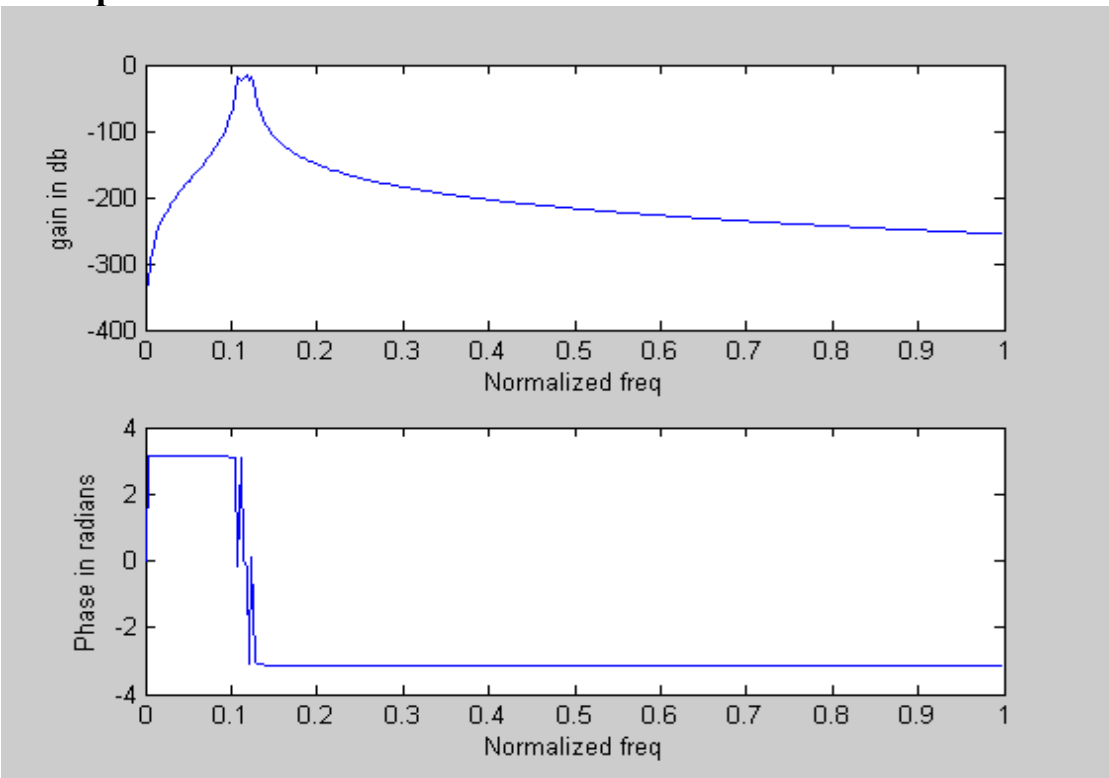
1) CHEB2ORD Chebyshev Type II filter order selection.
   [N, Wn] = CHEB2ORD(Wp, Ws, Rp, Rs) returns the order N of the lowest order digital Chebyshev Type II filter that loses no more than Rp dB in the passband and has at least Rs dB of attenuation in the stopband. Wp and Ws are the passband and stopband edge frequencies, normalized from 0 to 1 (where 1 corresponds to pi radians/sample). For example,
   Lowpass: Wp = .1, Ws = .2
   Highpass: Wp = .2, Ws = .1
   Bandpass: Wp = [.2 .7], Ws = [.1 .8]
   Bandstop: Wp = [.1 .8], Ws = [.2 .7]
   CHEB2ORD also returns Wn, the Chebyshev natural frequency to use with CHEBY2 to achieve the specifications.
   [N, Wn] = CHEB2ORD(Wp, Ws, Rp, Rs, 's') does the computation for an analog filter, in which case Wp and Ws are in radians/second.

**OUTPUT:**

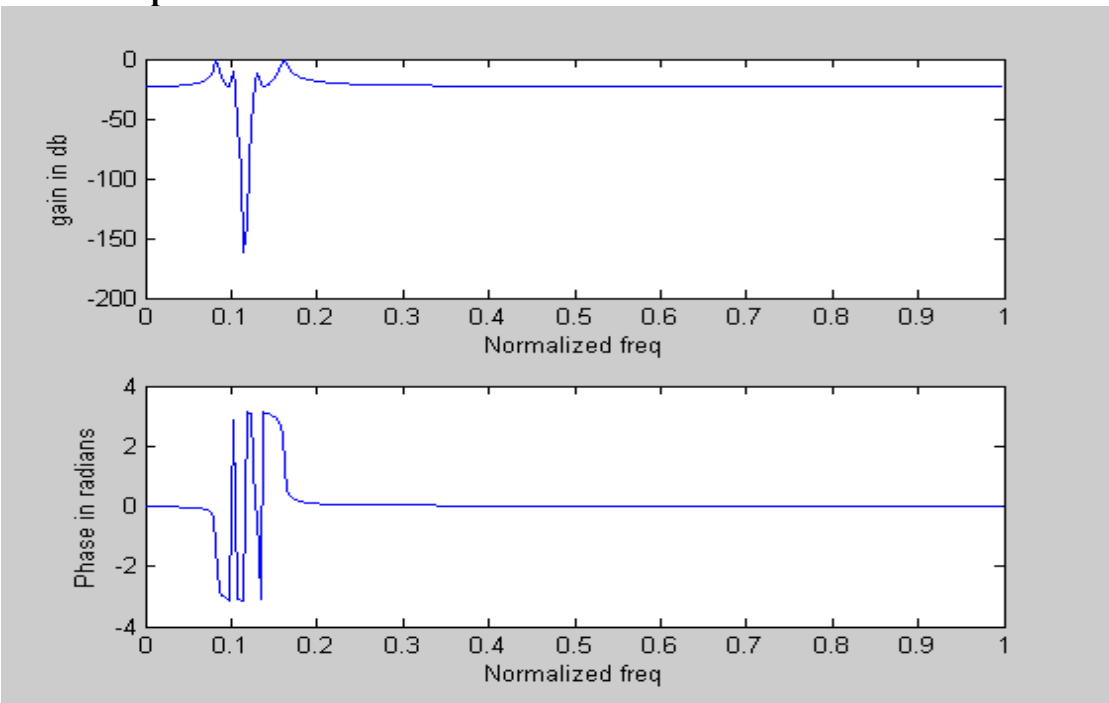
```
Enter the Pass band ripple 23
Enter the stop band ripple 46
Enter the Pass edge frequency  1300
Enter the stop edge frequency  1550
Enter the Sampling frequency  7800
```

## MODEL GRAPH:

**Band pass filter**




**Band Stop filter**




**RESULT:**

A Chebyshev Band Pass filter and Band Stop filter were designed an for the given specification using Mat lab.

**EXP NO: 6**

## DESIGN OF FIR FILTER

**AIM:**

To design a FIR LPF,HPF,BPF &BSF filter using following windowing techniques

i. Rectangular window
ii. Kaiser window
iii. Hamming window

**APPARATUS REQUIRED:**

A PC with Mat lab version 6.5.

**ALGORITHM:**

1) Get the order of the filter
2) Find the magnitude and phase of the filter
3) Draw the magnitude and phase representation

## PROGRAM:

**Rectangular window:**

```
clear all;
%----------------------LPF---------------------%
n=input('Enter the order of filter ');
wn=1/3;
b=fir1(n,wn,boxcar(n+1));
[h,o]=freqz(b,1);
m=20*log10(abs(h));
an=angle(h);
subplot(4,1,1);
plot(o/pi,m);
title('LPF Rectangular window');
ylabel('gain in db');
xlabel('Normalized freq');
subplot(4,1,2);
plot(o/pi,an);
ylabel('Phase in radians');
xlabel('Normalized freq');
%---------------HPF-------------------------------%
b1=fir1(n,wn,'high');
[h1,o1]=freqz(b1,1);
m1=20*log10(abs(h1));
an1=angle(h1);
subplot(4,1,3);
plot(o1/pi,m1);
title('HPF Rectangular window');
ylabel('gain in db');
xlabel('Normalized freq');
subplot(4,1,4);
plot(o1/pi,an1);
ylabel('Phase in radians');
xlabel('Normalized freq');
% %--------------------BPF-----------------------%
wn1=[1/6 1/3];
```

```
b2=fir1(n,wn1,'bandpass');
[h2,o2]=freqz(b2,1);
m2=20*log10(abs(h2));
an2=angle(h2);
subplot(4,1,1);
plot(o2/pi,m2);
title('BPF Rectangular window');
ylabel('gain in db');
xlabel('Normalized freq');
subplot(4,1,2);
plot(o2/pi,an2);
ylabel('Phase in radians');
xlabel('Normalized freq');
%--------------------BSF----------------------%
b3=fir1(n,wn1,'stop');
[h3,o3]=freqz(b3,1);
m3=20*log10(abs(h3));
an3=angle(h3);
subplot(4,1,3);
plot(o3/pi,m3);
title('BSF Rectangular window');
ylabel('gain in db');
xlabel('Normalized freq');
subplot(4,1,4);
plot(o3/pi,an3);
ylabel('Phase in radians');
xlabel('Normalized freq');
```

**Kaiser Window:**

```
clear all;
%---------------------LPF---------------------%
n=input('Enter the order of filter ');
wn=1/3;
b=fir1(n,wn,kaiser(n+1,4));
[h,o]=freqz(b,1);
m=20*log10(abs(h));
an=angle(h);
subplot(4,1,1);
plot(o/pi,m);
title('LPF kaiser window');
ylabel('gain in db');
xlabel('Normalized freq');
subplot(4,1,2);
plot(o/pi,an);
ylabel('Phase in radians');
xlabel('Normalized freq');
%--------------HPF-------------------------------%
b1=fir1(n,wn,'high',kaiser(n+1,4));
[h1,o1]=freqz(b1,1);
m1=20*log10(abs(h1));
an1=angle(h1);
subplot(4,1,3);
plot(o1/pi,m1);
```

```
title('HPF kaiser window');
ylabel('gain in db');
xlabel('Normalized freq');
subplot(4,1,4);
plot(o1/pi,an1);
ylabel('Phase in radians');
xlabel('Normalized freq');
%--------------------BPF-----------------------%
wn1=[1/6 1/3];
b2=fir1(n,wn1,kaiser(n+1,4));
[h2,o2]=freqz(b2,1);
m2=20*log10(abs(h2));
an2=angle(h2);
subplot(4,1,1);
plot(o2/pi,m2);
title('BPF kaiser window');
ylabel('gain in db');
xlabel('Normalized freq');
subplot(4,1,2);
plot(o2/pi,an2);
ylabel('Phase in radians');
xlabel('Normalized freq');
%--------------------BSF-----------------------%
b3=fir1(n,wn1,'stop',kaiser(n+1,4));
[h3,o3]=freqz(b3,1);
m3=20*log10(abs(h3));
an3=angle(h3);
subplot(4,1,3);
plot(o3/pi,m3);
title('BSF kaiser window');
ylabel('gain in db');
xlabel('Normalized freq');
subplot(4,1,4);
plot(o3/pi,an3);
ylabel('Phase in radians');
xlabel('Normalized freq');
```

**Hamming window:**

```
clear all;
%----------------------LPF---------------------%
n=input('Enter the order of filter ');
wn=1/3;
b=fir1(n,wn,hamming(n+1));
[h,o]=freqz(b,1);
m=20*log10(abs(h));
an=angle(h);
subplot(4,1,1);
plot(o/pi,m);
title('LPF Hamming window');
ylabel('gain in db');
xlabel('Normalized freq');
subplot(4,1,2);
plot(o/pi,an);
```

```
ylabel('Phase in radians');
xlabel('Normalized freq');
%---------------HPF-------------------------------%
b1=fir1(n,wn,'high',hamming(n+1));
[h1,o1]=freqz(b1,1);
m1=20*log10(abs(h1));
an1=angle(h1);
subplot(4,1,3);
plot(o1/pi,m1);
title('HPF Hamming window');
ylabel('gain in db');
xlabel('Normalized freq');
subplot(4,1,4);
plot(o1/pi,an1);
ylabel('Phase in radians');
xlabel('Normalized freq');
%--------------------BPF-----------------------%
wn1=[1/6 1/3];
b2=fir1(n,wn1,hamming(n+1));
[h2,o2]=freqz(b2,1);
m2=20*log10(abs(h2));
an2=angle(h2);
subplot(4,1,1);
plot(o2/pi,m2);
title('BPF Hamming window');
ylabel('gain in db');
xlabel('Normalized freq');
subplot(4,1,2);
plot(o2/pi,an2);
ylabel('Phase in radians');
xlabel('Normalized freq');
%--------------------BSF-----------------------%
b3=fir1(n,wn1,'stop',hamming(n+1));
[h3,o3]=freqz(b3,1);
m3=20*log10(abs(h3));
an3=angle(h3);
subplot(4,1,3);
plot(o3/pi,m3);
title('BSF Hamming window');
ylabel('gain in db');
xlabel('Normalized freq');
subplot(4,1,4);
plot(o3/pi,an3);
ylabel('Phase in radians');
xlabel('Normalized freq');
```

**COMMANDS:**

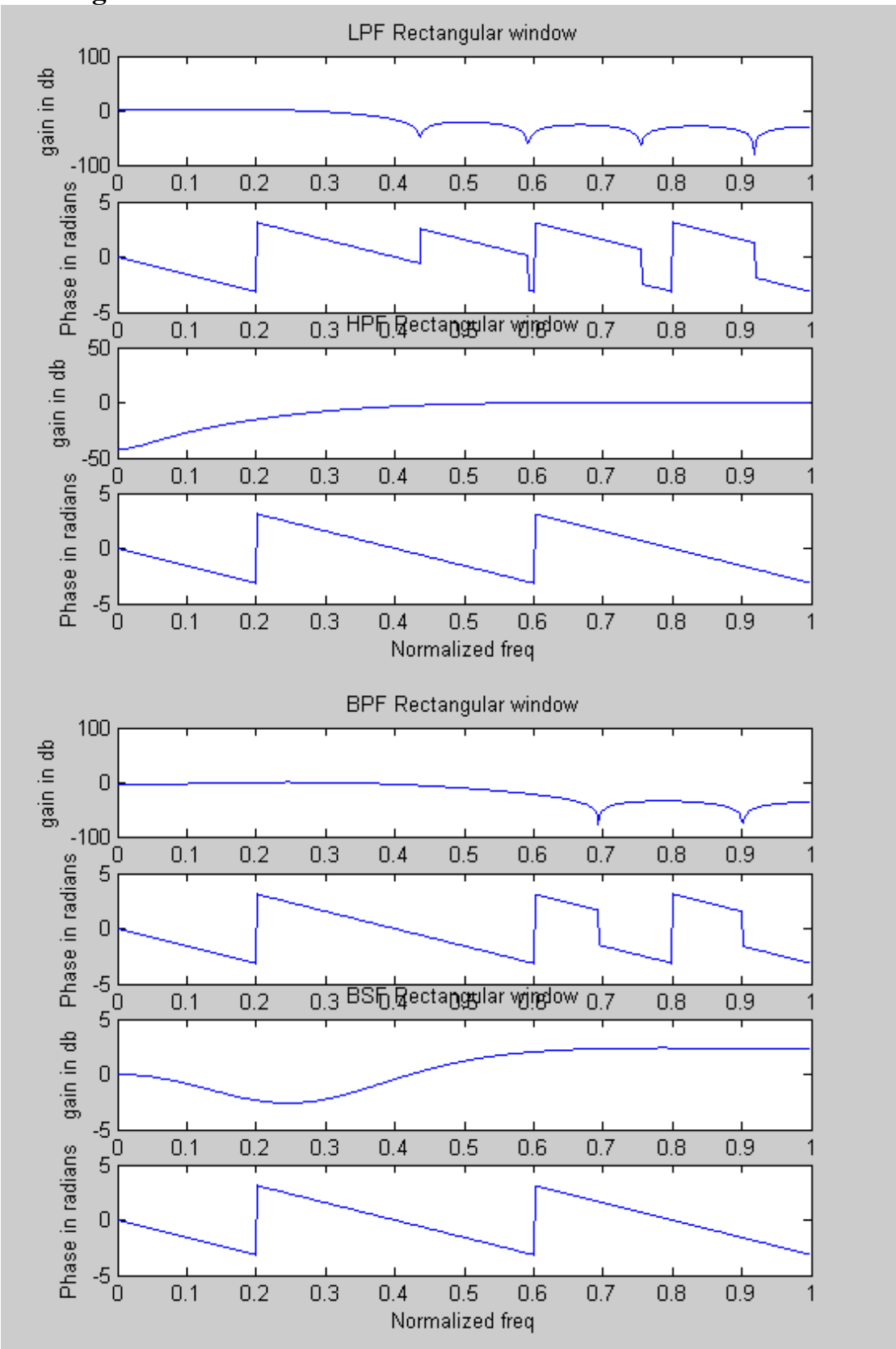
1) **FIR1** FIR filter design using the window method.
   B = FIR1(N,Wn) designs an N'th order lowpass FIR digital filter
   and returns the filter coefficients in length N+1 vector B.
   The cut-off frequency Wn must be between 0 < Wn < 1.0, with 1.0
   corresponding to half the sample rate. The filter B is real and

has linear phase. The normalized gain of the filter at Wn is
-6 dB.
B = **FIR1**(N,Wn,'high') designs an N'th order highpass filter.
You can also use B = FIR1(N,Wn,'low') to design a lowpass filter.
If Wn is a two-element vector, Wn = [W1 W2], FIR1 returns an
order N bandpass filter with passband W1 < W < W2. You can
also specify B = FIR1(N,Wn,'bandpass'). If Wn = [W1 W2],
**B = FIR1(N,Wn,'stop')** will design a bandstop filter.
If Wn is a multi-element vector,
Wn = [W1 W2 W3 W4 W5 ... WN],
FIR1 returns an order N multiband filter with bands
0 < W < W1, W1 < W < W2, ..., WN < W < 1.
B = FIR1(N,Wn,'DC-1') makes the first band a passband.
B = FIR1(N,Wn,'DC-0') makes the first band a stopband.
B = **FIR1**(N,Wn,WIN) designs an N-th order FIR filter using
the N+1 length vector WIN to window the impulse response.
If empty or omitted, FIR1 uses a Hamming window of length N+1.
For a complete list of available windows, see the help for the
WINDOW function. KAISER and CHEBWIN can be specified with an
optional trailing argument. For example, B = FIR1(N,Wn,kaiser(N+1,4))
uses a Kaiser window with beta=4. B = FIR1(N,Wn,'high',chebwin(N+1,R))
uses a Chebyshev window with R decibels of relative sidelobe
attenuation.
For filters with a gain other than zero at Fs/2, e.g., highpass
and bandstop filters, N must be even. Otherwise, N will be
incremented by one. In this case the window length should be
specified as N+2.
By default, the filter is scaled so the center of the first pass band
has magnitude exactly one after windowing. Use a trailing 'noscale'
argument to prevent this scaling, e.g. B = FIR1(N,Wn,'noscale'),
B = FIR1(N,Wn,'high','noscale'), B = FIR1(N,Wn,wind,'noscale'). You
can also specify the scaling explicitly, e.g. FIR1(N,Wn,'scale'), etc.

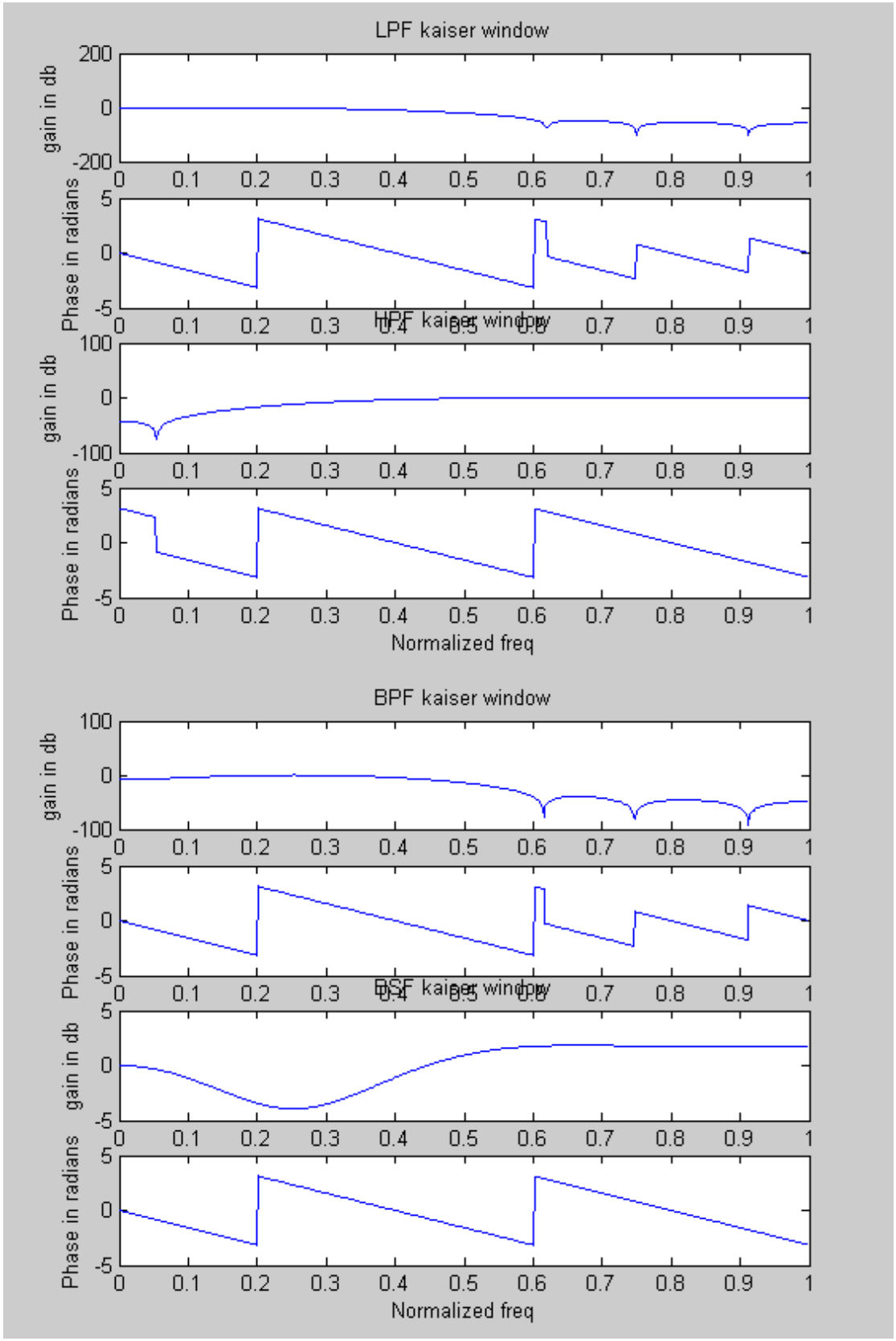
2) **FREQZ** Digital filter frequency response.
[H,W] = FREQZ(B,A,N) returns the N-point complex frequency response
vector H and the N-point frequency vector W in radians/sample of
the filter:

## MODEL GRAPH:

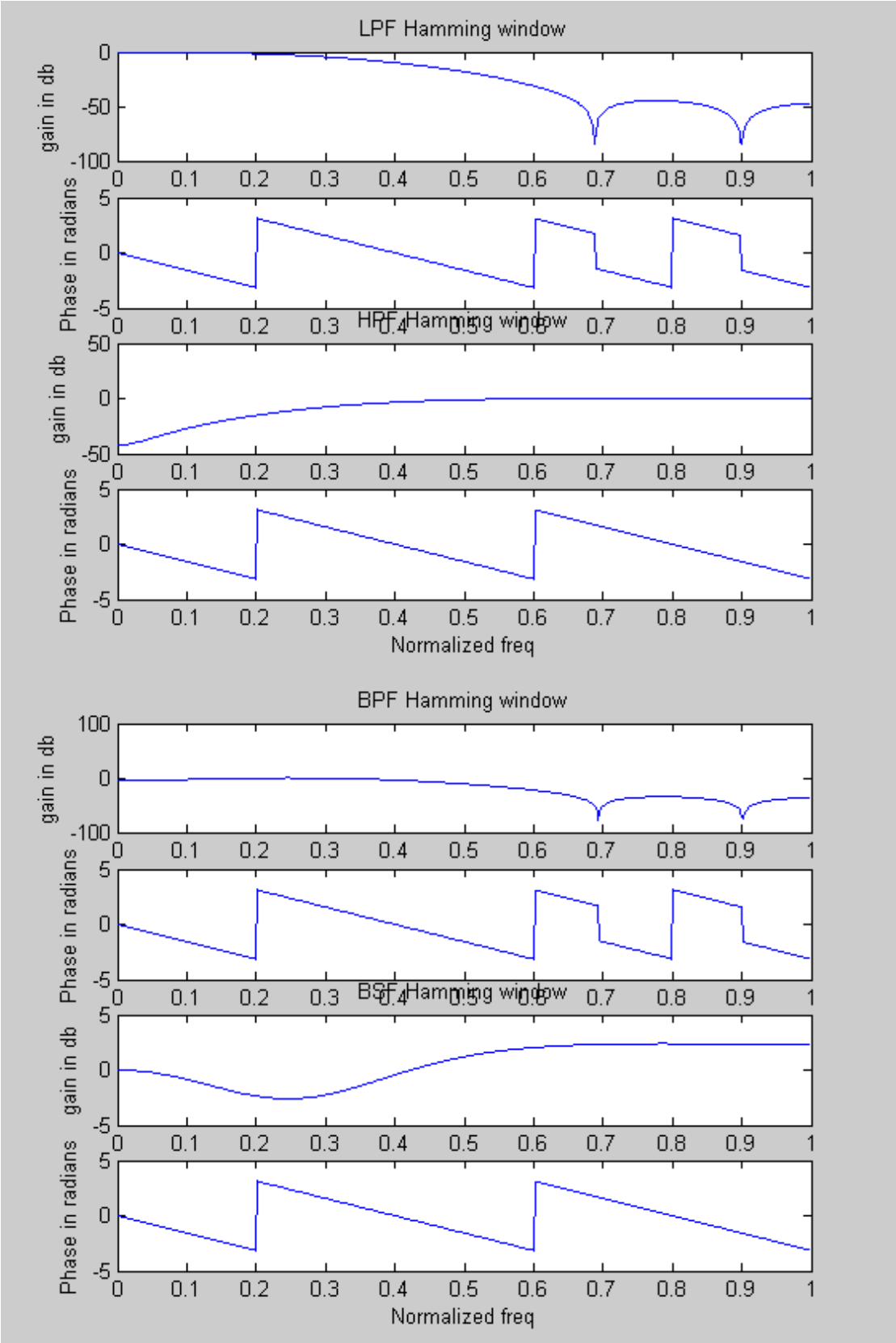
## Rectangular window




**Kaiser Window**




**Hamming Window**




**RESULT:**

The FIR filter has been designed by using different windowing techniques.

**EXP NO: 7**

## DECIMATION AND INTERPOLATION

**AIM:**

To generate the two sinusoidal sequences with frequencies f1 and f2 and decimate and interpolate by the factor M and L respectively using Mat lab.

**APPARATUS REQUIRED:**

A PC with Mat lab version 6.5.

**ALGORITHM:**

1) Enter the program in the Mat lab editor or debugger window
2) Get the frequencies f1 & f2
3) Get the decimation factor(M) & interpolation factor(L)
4) Add the two sinesiodal wave
5) Find decimated and interpolated waves
6) Display the output using stem function
7) Give the title for the program
8) Save & run the program
9) Give the input details in command window

**PROGRAM:**

```
clear all;
%----------------------up sampling & down sampling----------------------%
N=input('Enter the length of input sequence =');
L=input('Enter the upsampling factor   =');
M=input('Enter the downsampling factor   =');
f1=input('Enter the frequency of the Ist sinesiod=');
f2=input('Enter the frequency of the IInd sinesiod=');
n=0:N-1
x=sin(2*pi*f1*n)+sin(2*pi*f2*n);
y=decimate(x,M,'fir');
subplot(3,1,1);
stem(n,x(1:N));
title('input sequence');
ylabel('time index n');
xlabel('Amplidute');
subplot(3,1,2);
m=0:N/M-1
stem(m,y(1:N/M));
title('output sequence');
ylabel('time index n');
xlabel('Amplidute');
y1=interp(x,L);
subplot(3,1,3);
m=0:N*L-1
stem(m,y1(1:N*L));
title('output sequence');
ylabel('time index n');
xlabel('Amplidute');
```

1) **SIN** Sine.
   SIN(X) is the sine of the elements of X.
2) **DECIMATE** Resample data at a lower rate after lowpass filtering.
   Y = DECIMATE(X,R) resamples the sequence in vector X at 1/R times the original sample rate. The resulting resampled vector Y is R times shorter, LENGTH(Y) = LENGTH(X)/R.
   DECIMATE filters the data with an eighth order Chebyshev Type I lowpass filter with cutoff frequency .8*(Fs/2)/R, before resampling.

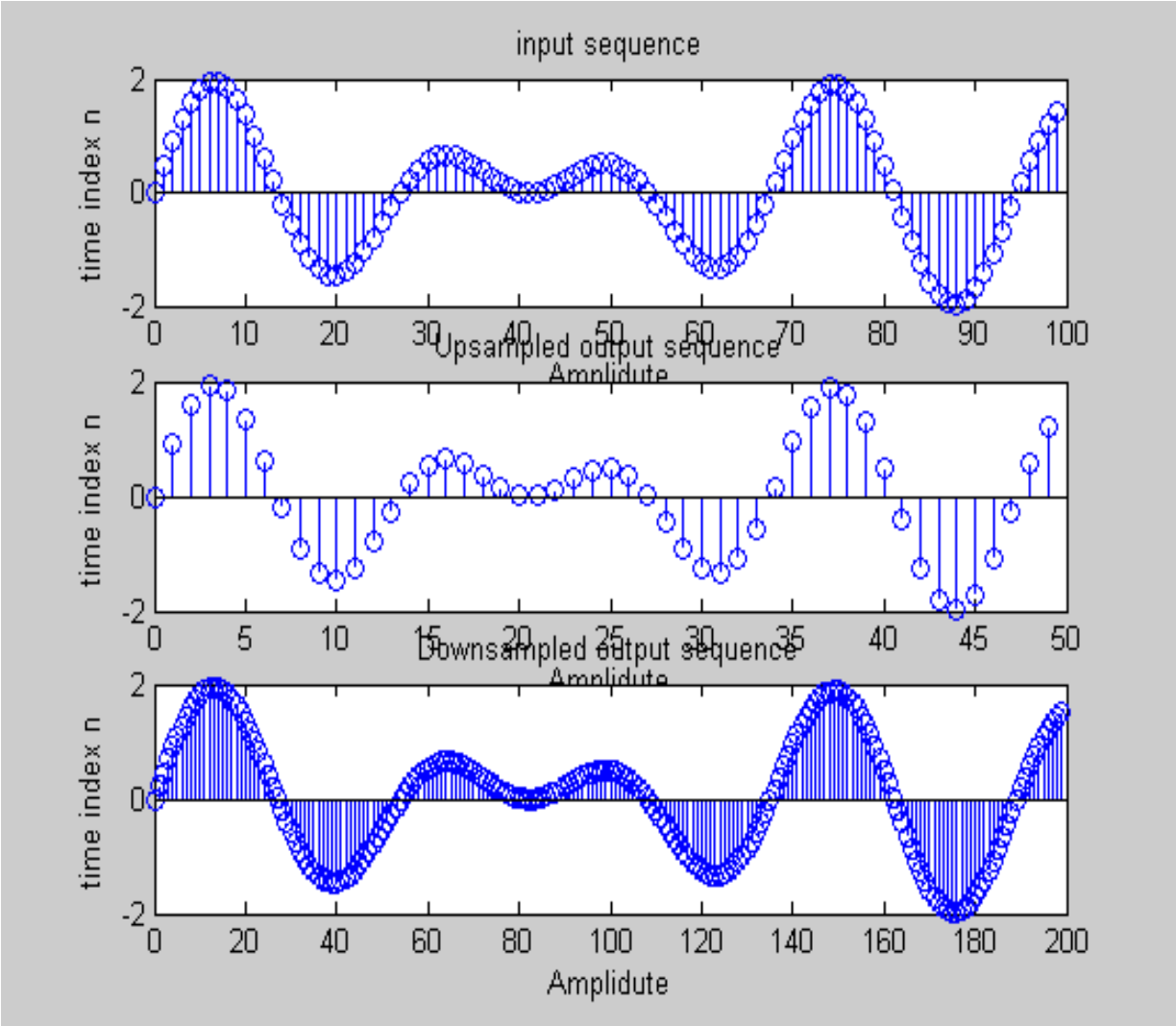
Y = DECIMATE(X,R,N) uses an N'th order Chebyshev filter.
Y = DECIMATE(X,R,'FIR') uses the 30 point FIR filter generated by FIR1(30,1/R) to filter the data.
Y = DECIMATE(X,R,N,'FIR') uses the N-point FIR filter.

3) **INTERP** Resample data at a higher rate using lowpass interpolation.
Y = INTERP(X,R) resamples the sequence in vector X at R times the original sample rate. The resulting resampled vector Y is R times longer, LENGTH(Y) = R*LENGTH(X).

**OUTPUT**

Enter the length of input sequence =100
Enter the upsampling factor =2
Enter the downsampling factor =2
Enter the frequency of the Ist sinesiod=0.043
Enter the frequency of the IInd sinesiod=0.03

**GRAPH:**




**RESULT:**

Thus the decimated and interpolated waves are plotted for the given specification using Mat lab.

Enter the length of input sequence =100
Enter the upsampling factor =2
Enter the downsampling factor =2
Enter the frequency of the Ist sinesiod=0.043
Enter the frequency of the IInd sinesiod=0.031

**EXP NO: 8**

## ANALOG TO DIGITAL CONVERSION

**AIM:**

To convert Analog to Digital filter using Bilinear transformation & Impulse Invariant transformation using Mat lab.

$$H(s) = 2/(s^2+3s+2)$$

**APPARATUS REQUIRED:**

A PC with Mat lab version 6.5.

**ALGORITHM:**

1) Enter the program in the Mat lab editor or debugger window
2) Enter the co-efficient of transfer function $H(s)=2/(s^2+3s+2)$
3) Analog to digital filter conversion of Bilinear transformation is done by using Mat lab function
4) Analog to digital filter conversion of impulse invariant transformation is done by using Mat lab function
5) Display the Bilinear transformation output & impulse invariant transformation output

**PROGRAM:**

```
clear all;
%--------ANALOG TO DIGITAL CONVERSION----------%
b=[2];
a=[1,3,2];
f=1;
%----------Bilinear transformation ------------%
[bz,az]=bilinear(b,a,f)
disp(bz);
disp(az);
%---------Impulse Invariant transformation ----%
[bz1,az1]= impinvar(b,a,f)
disp(bz);
disp(az);
```

**COMMANDS:**

1) **BILINEAR** Bilinear transformation with optional frequency prewarping.
   [Zd,Pd,Kd] = BILINEAR(Z,P,K,Fs) converts the s-domain transfer function specified by Z, P, and K to a z-transform discrete equivalent obtained from the bilinear transformation:
   H(z) = H(s) |
   | s = 2*Fs*(z-1)/(z+1)
   where column vectors Z and P specify the zeros and poles, scalar K specifies the gain, and Fs is the sample frequency in Hz.
2) **IMPINVAR** Impulse invariance method for analog to digital filter conversion.
   [BZ,AZ] = IMPINVAR(B,A,Fs) creates a digital filter with numerator and denominator coefficients BZ and AZ respectively whose impulse response is equal to the impulse response of the analog filter with coefficients B and A sampled at a frequency of Fs Hertz. The B and A coefficients will be scaled by 1/Fs.

**OUTPUT:**

**Bilinear transformation**

bz = 0.1667 0.3333 0.1667

az = 1.0000 -0.3333 0.0000
0.1667 0.3333 0.1667
1.0000 -0.3333 0.0000

**Impulse Invariant transformation**
bz1 = 0 0.4651

az1 = 1.0000 -0.5032 0.0498
0.1667 0.3333 0.1667
1.0000 -0.3333 0.0000

**RESULT:**

$$H(s)=2/(s^2+3s+2)$$

Thus the analog transfer function was converted by using Bilinear transformation & Impulse Invariant transformation in Mat lab function.

**EXP NO: 9**

**IMPULSE AND STEP RESPONSE**

**AIM:**

To determine the impulse and step response for the following causal system and to plot the pole-zero pattern and to sketch the stability by using Mat lab.

i. y(n) = 0.7y(n-1)-0.1y(n-2)+2x(n)-x(n-2)
ii. y(n) =(3/4)y(n-1)-(1/8)y(n-2)+x(n)

**APPARATUS REQUIRED:**

A PC with Mat lab version 6.5.

**ALGORITHM:**

1) Enter the program in the Mat lab editor or debugger window
2) Enter the co-efficient of numerator and denominator as ‘a’ & ‘b’
3) Find the impulse response using Z-transform
4) Find the step response using the convolution function
5) Check whether the system is stable or not
6) Plot the response

**PROGRAM:**

```
clear all;
%--------Impulse response & Step response-----------%
a=input('Enter the Co-efficient values of a');
b=input('Enter the Co-efficient values of b');
h=impz(a,b);
% disp(h);
subplot(3,2,1);
stem(h);
title('Impulse response');
ylabel('Amplitude');
xlabel('Time');
u=[ones(1,length(h))];
s=conv(u,h);
disp(s);
subplot(3,2,2);
stem(s);
title('Step response');
ylabel('Amplitude');
xlabel('Time');
[z,p,k]=tf2zp(a,b)
subplot(3,2,3);
zplane(z,p);
title('Pole-Zero plot');
if i<i
   disp('The system is stable');
else
   disp('The system is not stable');
end
```

**COMMANDS:**

1) **IMPZ** Impulse response of digital filter
   [H,T] = IMPZ(B,A) computes the impulse response of the filter B/A choosing the number of samples for you, and returns the response in column vector H and a vector of times (or sample intervals) in T (T = [0 1 2 ...]').
   [H,T] = IMPZ(B,A,N) computes N samples of the impulse response. If N is a vector of integers, the impulse response is computed only at those integer values (0 is the origin).
   [H,T] = IMPZ(B,A,N,Fs) computes N samples and scales T so that samples are spaced 1/Fs units apart. Fs is 1 by default.
   [H,T] = IMPZ(B,A,[],Fs) chooses the number of samples for you and scales

T so that samples are spaced 1/Fs units apart.

2) **TF2ZP** Transfer function to zero-pole conversion.
[Z,P,K] = TF2ZP(NUM,DEN) finds the zeros, poles, and gains:

$$H(s) = K\frac{(s-z1)(s-z2)...(s-zn)}{(s-p1)(s-p2)...(s-pn)}$$

from a SIMO transfer function in polynomial form:

$$H(s) = \frac{NUM(s)}{DEN(s)}$$

Vector DEN specifies the coefficients of the denominator in descending powers of s. Matrix NUM indicates the numerator coefficients with as many rows as there are outputs. The zero locations are returned in the columns of matrix Z, with as many columns as there are rows in NUM. The pole locations are returned in column vector P, and the gains for each numerator transfer function in vector K.

For discrete-time transfer functions, it is highly recommended to make the length of the numerator and denominator equal to ensure correct results. You can do this using the function EQTFLENGTH in the Signal Processing Toolbox. However, this function only handles single-input single-output systems.

**OUTPUT:**

y(n) = 0.7y(n-1)-0.1y(n-2)+2x(n)-x(n-2)

```
z  =  0.7071
     -0.7071

p  =  0.5000
      0.2000
k  =  2
```

The system is not stable

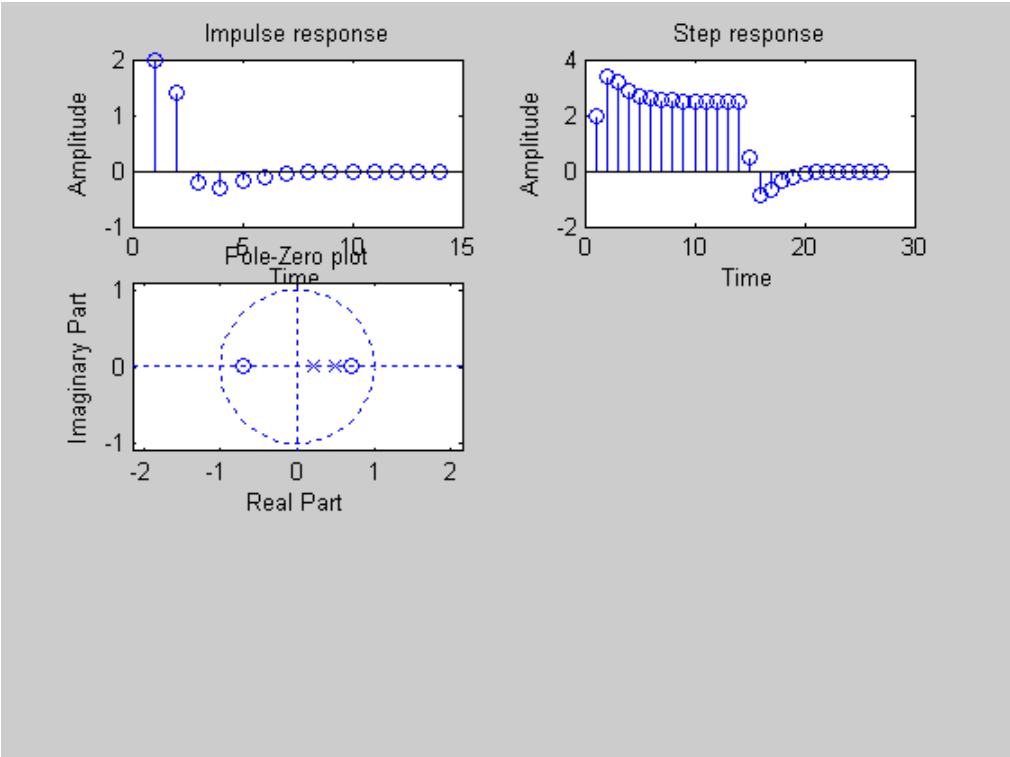
i. y(n) =(3/4)y(n-1)-(1/8)y(n-2)+x(n)

z = Empty matrix: 0-by-1

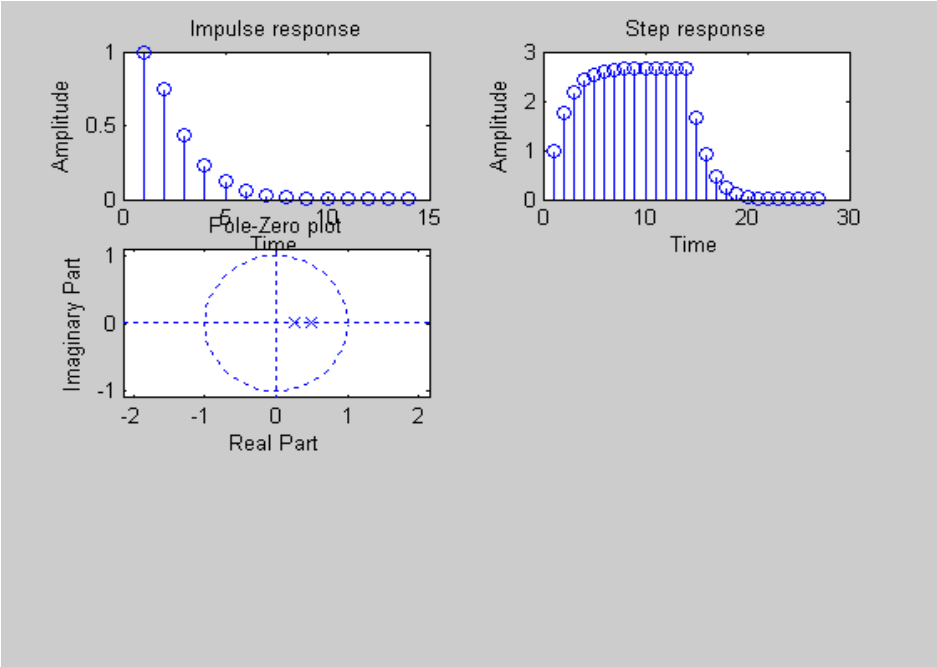
p = 0.5000
0.2500
k = 1

**MODEL GRAPH:**

i. y(n) = 0.7y(n-1)-0.1y(n-2)+2x(n)-x(n-2)




ii. y(n) =(3/4)y(n-1)-(1/8)y(n-2)+x(n)




**RESULT:**

The impulse and step response is obtained for the given causal system and also pole-zero pattern was plotted.

**EXP NO: 10**

## SAMPLING AND ALIASING EFFECT

**AIM:**

To write a Mat lab code for a continuous time signal sample that various frequency illustrating the problem of aliasing caused by sampling at low of a frequency.

**APPARATUS REQUIRED:**

A PC with Mat lab version 6.5.

**ALGORITHM:**

1) Enter the program in the Mat lab editor or debugger window
2) The figure-size, $T_{S1}$, $T_{S2}$, $T_{S3}$ and $w_{s1}$, $w_{s2}$, $w_{s3}$ values are initialized
3) To calculate the original continuous time signal using the sum of two cosines
4) Sampling the continuous time signal is done at various $T_S$= 0.05s, $T_S$ =0.1s, $T_S$ = 0.25 respectively
5) The Nyquist sampling theorem is violated and x(t) could not be recovered from the samples using LPF. Aliasing is then occurred
6) Few frequencies of sampled signals are computed

**PROGRAM:**

```
clear all;
%--------Sampling and Aliasing effect----------%
% ws1=125.6637, ws2=62.8319, ws3= 31.4159
fig_size=[232 84 774 624];
Ts1=0.05;Ts2=0.1;Ts3=0.2;
ws1=2*pi/Ts1;ws2=2*pi/Ts2;ws3=2*pi/Ts3;
%---------frequencies for the CT signal and time vector-%
w1=7;w2=23;
t=[0:0.005:2];
x=cos(w1*t)+cos(w2*t);
figure(1),clf,plot(t,x),grid,xlabel('Time (s)'),ylabel('Amplitude'),...
title('continuous time signal ');%x(t)=cos(7t)+cos(23t),...
set(gcf,'Position',fig_size)
%---Sampling the continuous time signal with a sampling period Ts=0.05s---%
t1=[0:Ts1:2];
xs1=cos(w1*t1)+cos(w2*t1);
figure(2),clf,stem(t1,xs1);grid,hold on,plot(t,x,'r:'),hold off,...
xlabel('Time (s)'),ylabel('Amplitude'),...
title('Sampled s version of x(t) with Ts=0.05s'),...
set(gcf,'Position',fig_size)
%---Sampling the continuous time signal with a sampling period Ts=0.1s---%
t2=[0:Ts2:2];
xs2=cos(w1*t2)+cos(w2*t2);
figure(3),clf,stem(t2,xs2);grid,hold on,plot(t,x,'r:'),hold off,...
xlabel('Time (s)'),ylabel('Amplitude'),...
title('Sampled s version of x(t) with Ts=0.1s'),...
set(gcf,'Position',fig_size)
%---Sampling the continuous time signal with a sampling period Ts=0.2s---%
t3=[0:Ts3:2];
xs3=cos(w1*t3)+cos(w2*t3);
figure(4),clf,stem(t3,xs3);grid,hold on,plot(t,x,'r:'),hold off,...
xlabel('Time (s)'),ylabel('Amplitude'),...
title('Sampled s version of x(t) with Ts=0.2s'),...
```

```
set(gcf,'Position',fig_size)
%--Since ws3<2*w2, the Nyquist sampling theorem is violated, and x(t)could
%not be recovered from the samples obtained with Ts3 using an ideal low
%pass filter. Aliasing has occurred
w2s3=w2-ws3;
x1=cos(w1*t1)+cos(w2s3*t1);
figure(5),clf,stem(t3,xs3);grid,hold on,plot(t,x,'b:'),...
hold off,xlabel('Time (s)'),ylabel('Amplitude'),...
title('Sampling x(t)and x_1(t) with Ts=0.2s'),...
set(gcf,'Position',fig_size),...
text(1.13,1.2,'x(t)'),text(0.1,1.6,'x_1(t)')
%----Computing the first few frequencies in the sampled signals---------%
n=[-1 0 1];wx=[-w2 -w1 w1 w2];
wx1=[]; wx2=[];wx3=[];
for i=1:length(n);
   wx1=[wx1 (wx+n(i)*ws1)];
   wx2=[wx2 (wx+n(i)*ws2)];
   wx1=[wx3 (wx+n(i)*ws3)];
end
wx1=sort(wx1); wx2=sort(wx2); wx3=sort(wx3);
clear i
```

**COMMANDS:**

1) **GRID** Grid lines.

   GRID ON adds major grid lines to the current axes.
   GRID OFF removes major and minor grid lines from the current axes.
   GRID MINOR toggles the minor grid lines of the current axes.
   GRID, by itself, toggles the major grid lines of the current axes.
   GRID(AX,...) uses axes AX instead of the current axes.
   GRID sets the XGrid, YGrid, and ZGrid properties of
   the current axes.
   set(AX,'XMinorGrid','on') turns on the minor grid.

2) **PLOT** Linear plot.

   PLOT(X,Y) plots vector Y versus vector X. If X or Y is a matrix,
   then the vector is plotted versus the rows or columns of the matrix,
   whichever line up. If X is a scalar and Y is a vector, length(Y)
   disconnected points are plotted.
   PLOT(Y) plots the columns of Y versus their index.
   If Y is complex, PLOT(Y) is equivalent to PLOT(real(Y),imag(Y)).
   In all other uses of PLOT, the imaginary part is ignored.
   Various line types, plot symbols and colors may be obtained with
   PLOT(X,Y,S) where S is a character string made from one element
   from any or all the following 3 columns:

| | | | | | |
|---|---|---|---|---|---|
| b | blue | . | point | - | solid |
| g | green | o | circle | : | dotted |
| r | red | x | x-mark | -. | dashdot |
| c | cyan | + | plus | -- | dashed |
| m | magenta | * | star | | |
| y | yellow | s | square | | |
| k | black | d | diamond | | |
| | | v | triangle (down) | | |
| | | ^ | triangle (up) | | |
| | | < | triangle (left) | | |

> triangle (right)
p pentagram
h hexagram

For example, PLOT(X,Y,'c+:') plots a cyan dotted line with a plus at each data point; PLOT(X,Y,'bd') plots blue diamond at each data point but does not draw any line.

PLOT(X1,Y1,S1,X2,Y2,S2,X3,Y3,S3,...) combines the plots defined by the (X,Y,S) triples, where the X's and Y's are vectors or matrices and the S's are strings.

For example, PLOT(X,Y,'y-',X,Y,'go') plots the data twice, with a solid yellow line interpolating green circles at the data points.

The PLOT command, if no color is specified, makes automatic use of the colors specified by the axes ColorOrder property. The default ColorOrder is listed in the table above for color systems where the default is blue for one line, and for multiple lines, to cycle through the first six colors in the table. For monochrome systems, PLOT cycles over the axes LineStyleOrder property.

PLOT returns a column vector of handles to LINE objects, one handle per line.

The X,Y pairs, or X,Y,S triples, can be followed by parameter/value pairs to specify additional properties of the lines.

3) **HOLD** Hold current graph.

HOLD ON holds the current plot and all axis properties so that subsequent graphing commands add to the existing graph.

HOLD OFF returns to the default mode whereby PLOT commands erase the previous plots and reset all axis properties before drawing new plots.

HOLD, by itself, toggles the hold state.

HOLD does not affect axis autoranging properties.

Algorithm note:

HOLD ON sets the NextPlot property of the current figure and axes to "add".

HOLD OFF sets the NextPlot property of the current axes to "replace".CLF Clear current figure.

CLF deletes all children of the current figure with visible handles.

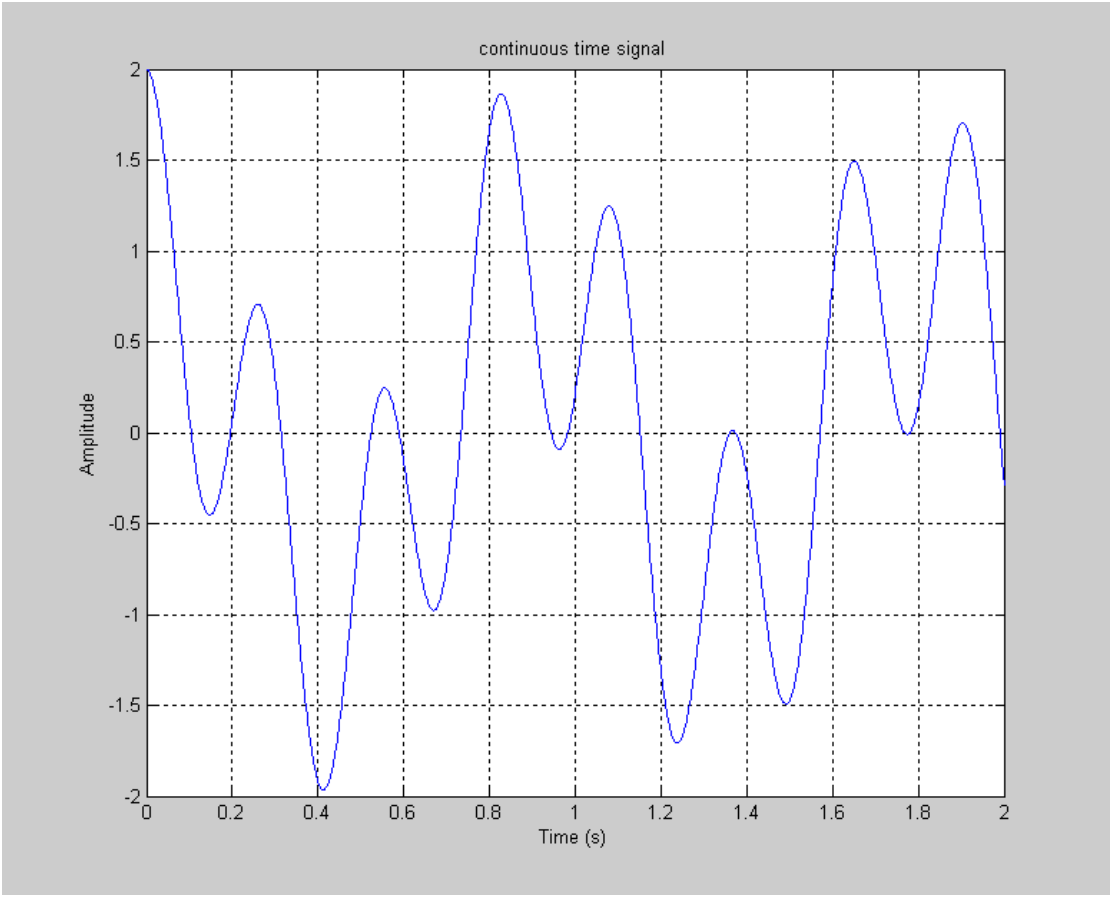
CLF RESET deletes all children (including ones with hidden handles) and also resets all figure properties, except Position and Units, to their default values.

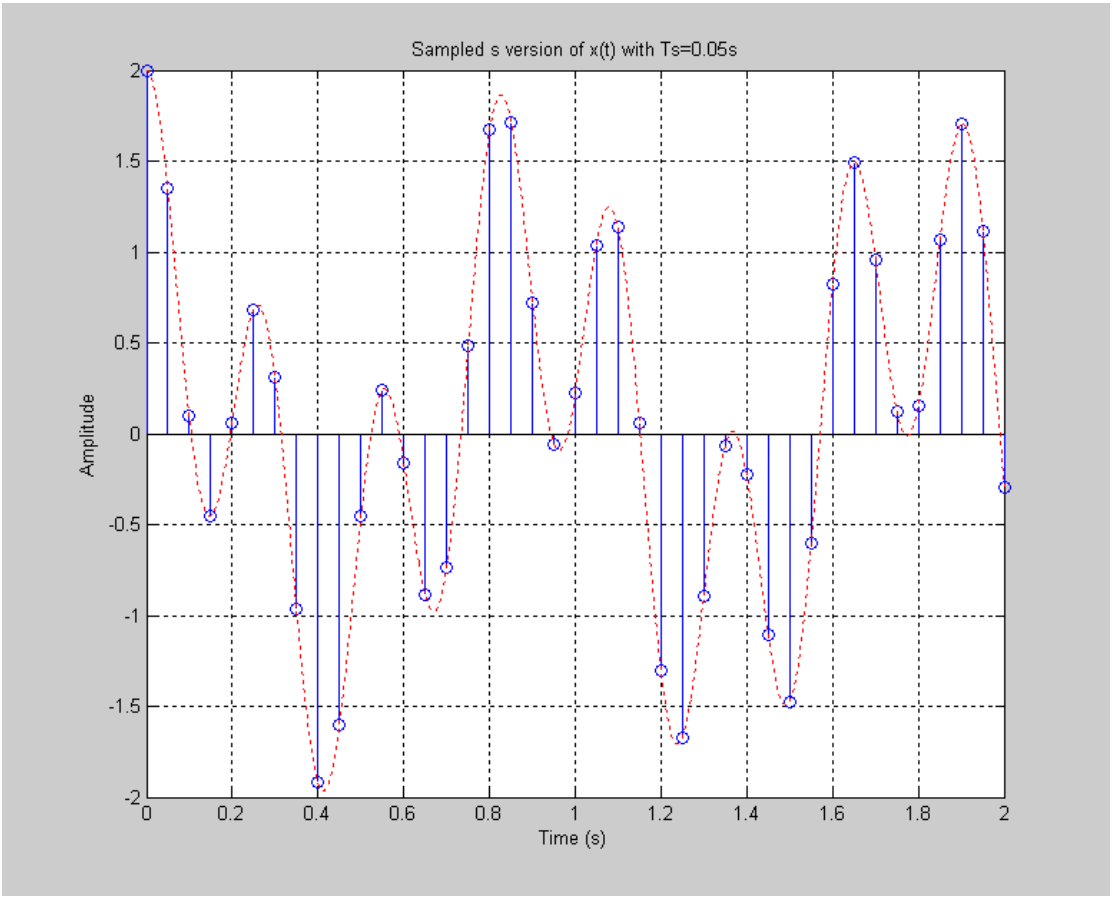
4) SET Set object properties.

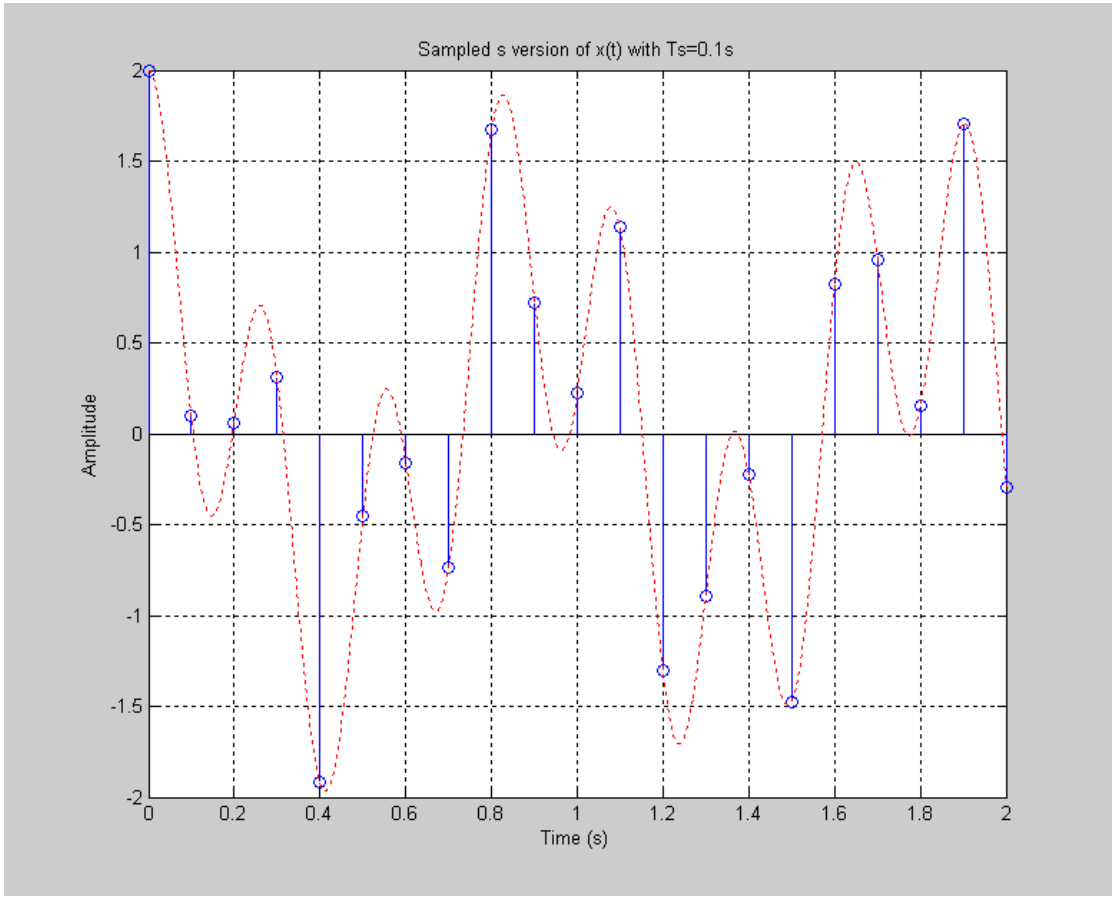
SET(H,'PropertyName',PropertyValue) sets the value of the specified property for the graphics object with handle H.

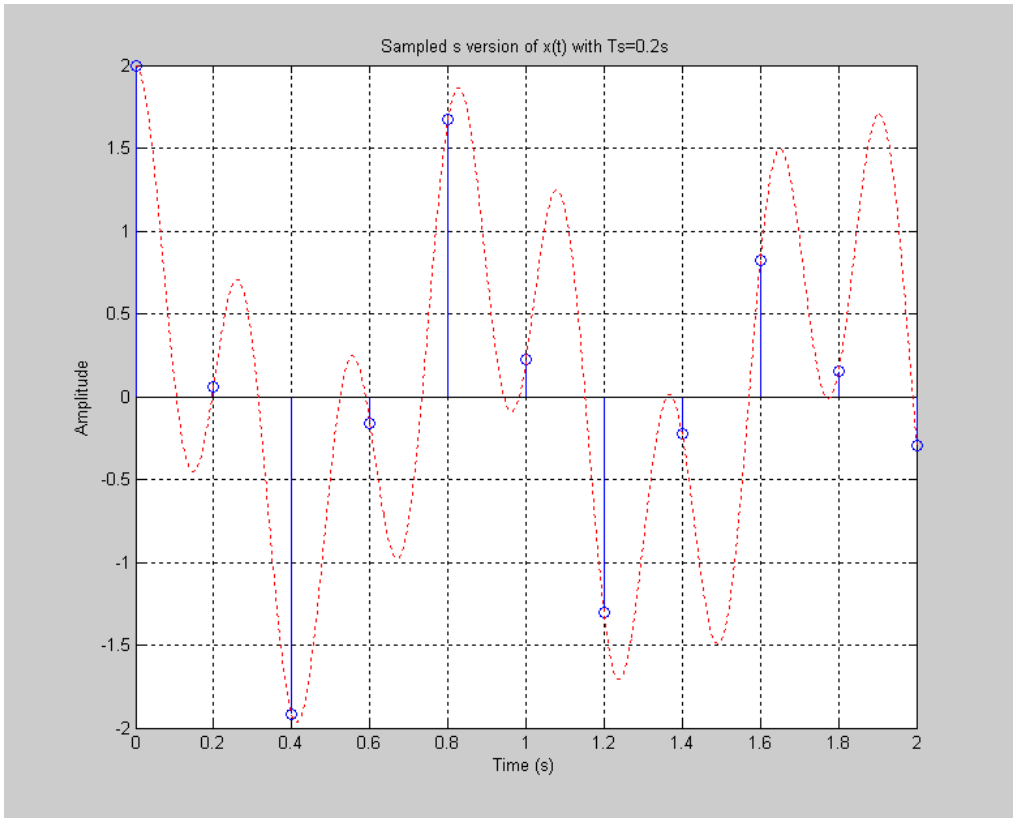
H can be a vector of handles, in which case SET sets the properties' values for all the objects.

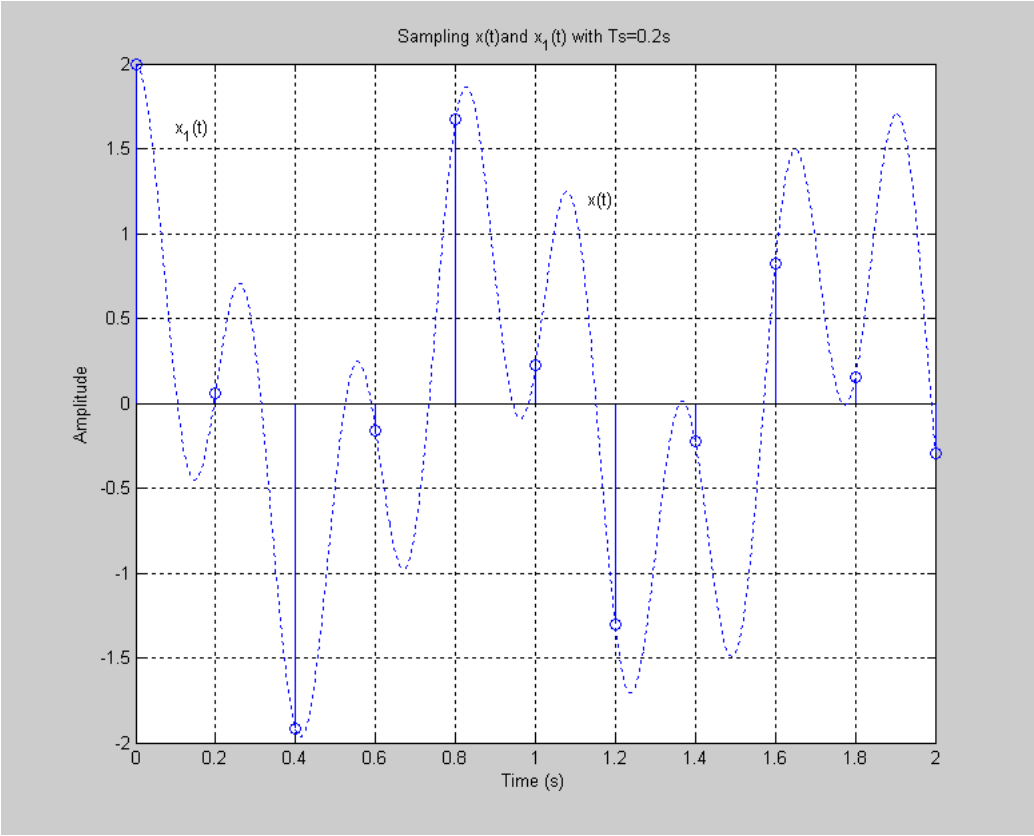
**MODEL GRAPH:**


continuous time signal
Amplitude
Time (s)
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2


Sampled s version of x(t) with Ts=0.05s
Amplitude
Time (s)
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2


Sampled s version of x(t) with Ts=0.1s
Amplitude
Time (s)
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2


Sampled s version of x(t) with Ts=0.2s
Amplitude
Time (s)
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2




**RESULT:**

Thus continuous time signal is sampled at various frequency and aliasing effect is analysed

## LIST OF EXPERIMENTS

| | | |
|---|---|---|

| | |
|---|---|
| **Exp. No.:** | |
| **Date:** | |

**AIM:**

**APPARATUS REQUIRED:**

**THEORY:**

**RESULT:**