PVP 23

Code: 23EC3602

III B.Tech - II Semester - Regular Examinations – APRIL 2026

MICROPROCESSORS & MICROCONTROLLERS

(ELECTRONICS & COMMUNICATION ENGINEERING)

Duration: 3 hours Max. Marks: 70

Note: 1. This question paper contains two Parts A and B.
2. Part-A contains 10 short answer questions. Each Question carries 2 Marks.
3. Part-B contains 5 essay questions with an internal choice from each unit. Each Question carries 10 marks.
4. All parts of Question paper must be answered in one place.

BL – Blooms Level CO – Course Outcome

PART – A

		BL	CO
1.a)	List different types of interrupts in 8086.	L1	CO1
1.b)	Discuss the role of queue in 8086 architecture.	L2	CO1
1.c)	What is the function of the Debug system in Cortex-M?	L1	CO1
1.d)	Explain register set in Cortex-M architecture.	L2	CO1
1.e)	Explain any two assembler directives in Cortex-M.	L2	CO1
1.f)	List different data processing instructions in Cortex-M.	L1	CO1
1.g)	Discuss the steps to configure a microcontroller pin as input and output.	L2	CO1
1.h)	Define GPIO in a Cortex-M based microcontroller.	L1	CO1
1.i)	Discuss the need for timing interfaces in real-time applications.	L2	CO1
1.j)	Why is frequency pre-scaling required?	L2	CO1

PART – B

			BL	CO	Max. Marks
UNIT-I					
2	a)	Explain assembly language development tools.	L1	CO1	5 M
	b)	Compare minimum mode and maximum mode configuration.	L2	CO1	5 M
OR					
3	a)	Explain various addressing modes with examples.	L3	CO2	5 M
	b)	Develop an assembly language program to add two 16-bit numbers.	L3	CO2	5 M
UNIT-II					
4	a)	Describe the bus system and bus matrix organization in Cortex-M.	L3	CO2	5 M
	b)	Describe interrupt configuration in Cortex-M.	L3	CO2	5 M
OR					
5	a)	Explain exception handling mechanism in Cortex-M.	L3	CO2	5 M
	b)	Differentiate between Cortex-M and 8086 with respect to architectures.	L3	CO2	5 M
UNIT-III					
6	a)	Demonstrate LDR and STR memory access instructions.	L3	CO2	5 M
	b)	Illustrate various Data Processing instructions in Cortex-M.	L3	CO2	5 M
OR					

7	a)	Demonstrate 16 bit instruction encoding in Cortex-M.	L3	CO2	5 M
	b)	Demonstrate the execution flow of a loop using branch instructions.	L3	CO2	5 M
UNIT-IV					
8	a)	Analyze LED and switch interfacing with Cortex-M with neat diagram.	L4	CO3	5 M
	b)	Compare and contrast serial communication and parallel communication in UART.	L4	CO3	5 M
OR					
9	a)	Analyze how to interface a seven segment display and TM4C123 microcontroller.	L4	CO3	5 M
	b)	Analyze how to configure ADC for analog input conversion.	L4	CO3	5 M
UNIT-V					
10	a)	Analyze basic timing interfaces in TM4C123 microcontroller.	L4	CO3	5 M
	b)	Evaluate the generation of pulse width modulation using timer as an output device.	L5	CO4	5 M
OR					
11	a)	Analyze the working of Systick Timer in TM4C123 microcontroller.	L4	CO3	5 M
	b)	Design the steps to generate variable frequency signals using timers.	L5	CO4	5 M

Code: 23EC3602

PVP23

III B.Tech – II Semester – Regular Examinations
April 2026
MICRPROCESSORS AND MICROCONTROLLERS
(ELECTRONICS AND COMMUNICATION ENGINEERING)

Duration : 3 hours **Max. Marks:70**

Scheme of Evaluation

PART-A

1) **a)** Types of Interrupts-1 Mark, Any two interrupts list-1Mark
1) **b)** Purpose and role of queue –2Marks
1) **c)** Purpose and Function of Debug--2Marks
1) **d)** Types-1mark, Explanation—1 Mark
1) **e)** Explanation of any two assembler directives-2 Marks
1) **f)** List of any 4 data processing instructions-2 Marks (each 0.5 Marks)
1) **g)** Any 2 steps--2 Marks (each 1 Marks)
1) **h)** Definition—1Mark, Explanation—1 Mark
1) **i)** Purpose of timing interfaces -2Marks
1) **j)** Explanation of Frequency pre-scaling—2 Marks

PART-B

UNIT-I

2) **a)** List –1 Mark, Explanation-4Marks
2) **b)** Any 5 points—5Marks

(OR)

3) **a)** List –1 Mark, Explanation about any 4 addressing modes-4Marks
3) **b)** Program—5Marks

UNIT-II

4) **a)** Diagram--2 Marks, Explanation—3 Marks
4) **b)** Diagram—1 mark, Explanation—4Marks

(OR)

5) **a)** Explanation—5 Marks
5) **b)** Any 5 points—5Marks

UNIT-III

6) **a)** Demonstration—3 Marks, Syntax---2 Marks
6) **b)** Demonstration about any 5 instructions with syntax—5 Marks

(OR)

7) **a)** Diagram--2 Marks, Explanation—3 Marks
7) **b)** Explanation about for loop -2 1/2 Marks.
Explanation about while loop -2 1/2 Marks

UNIT-IV

8) **a)** Diagram--2 Marks, Explanation—3 Marks
8) **b)** Any 5 points—5Marks

(OR)

9) **a)** Any one Diagram--2 Marks, Explanation—3 Marks
9) **b)** List of configuration steps—2 Marks, Explanation—3 Marks

UNIT-V

10) **a)** Diagram--1 Mark, Explanation—4 Marks
10) **b)** Diagram--2 Marks, Explanation—3 Marks

(OR)

11) **a)** Diagram--1 Mark, Explanation—4 Marks
11) **b)** Table--1 Mark, Explanation—4 Marks

Code: 20EC3602

PVP23

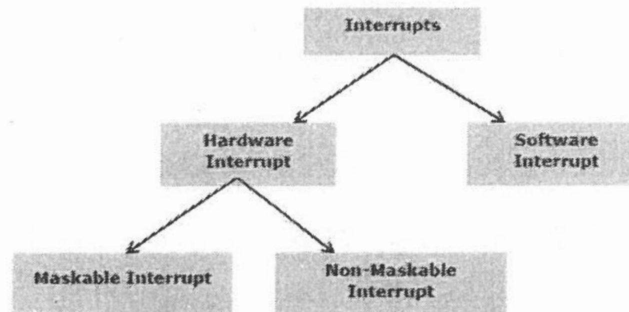
III B.Tech – II Semester – Regular Examinations
April 2026
MICRPROCESSORS AND MICROCONTROLLERS
(ELECTRONICS AND COMMUNICATION ENGINEERING)

Duration : 3 hours **Max. Marks:70**

Scheme of Evaluation

PART-A

1 a) List different types of interrupts in 8086.



Hardware Interrupts: NMI, INTR
Software Interrupts: TYPE 0 to TYPE 255

1b) Discuss the role of queue in 8086 architecture

In the 8086 microprocessor, the **queue** refers to the **instruction queue** used for faster execution. The 8086 microprocessor uses a 6-byte instruction queue to store upcoming instructions. This allows overlapping of instruction fetching and execution (pipelining), improving speed. While one instruction is being executed, the next instructions are fetched and stored in the queue, increasing overall efficiency

1c) What is the function of debug system in Cortex-M?

In ARM Cortex-M, the debug system is used to monitor and control program execution during development. The debug system in Cortex-M provides features like **breakpoints, watch points, and single-step execution**, allowing developers to pause and inspect the program. It helps in **detecting and correcting errors**, analysing register and memory contents, and ensuring proper program operation. This improves reliability and simplifies troubleshooting during embedded system development.

1d) Explain register set in Cortex-M architecture

In ARM Cortex-M, the register set consists of general-purpose and special-purpose registers used for data processing and control. The Cortex-M register set includes **16 registers (R0–R15)**. Registers **R0–R12** are general-purpose registers used for data operations. **R13 (SP)** is the Stack Pointer, **R14 (LR)** is the Link Register used for subroutine return, and **R15 (PC)** is the Program Counter that holds the address of the next instruction. Additionally, special registers like **xPSR** store status and control information.

1e) Explain any Two assembler directives in Cortex-M

(Consider any 2)

In ARM Cortex-M, assembler directives are instructions given to the assembler to control program organization and memory usage (they are not executed by the CPU).

Common assembler directives in Cortex-M include:

- **AREA** – Defines a code or data section
- **ENTRY** – Specifies the program entry point
- **END** – Marks the end of the program
- **EQU** – Defines constants
- **DCB / DCW / DCD** – Define byte, word, and double word data
- **ALIGN** – Aligns data in memory
- **EXPORT / IMPORT** – Used for linking between modules

These directives help in organizing code, defining data, and controlling memory layout.

1f) List data processing instructions in Cortex-M

In ARM Cortex-M, data processing instructions are used to perform arithmetic and logical operations on registers.

Common data processing instructions in Cortex-M include:

- **ADD, ADC** – Addition operations
- **SUB, SBC** – Subtraction operations
- **MUL** – Multiplication
- **AND, ORR, EOR** – Logical operations (AND, OR, XOR)
- **BIC** – Bit clear operation
- **CMP, CMN** – Comparison operations
- **MOV, MVN** – Data transfer and negation
- **LSL, LSR, ASR, ROR** – Shift and rotate operations

These instructions operate directly on data in registers and are essential for computation.

1g) Discuss the steps to configure a microcontroller pin as input and output

To configure a microcontroller, pin as input or output (in **ARM Cortex-M**):

1. **Enable clock** for the GPIO port using the control register.
2. **Select pin function** (GPIO mode) using the pin function register.
3. **Set direction**:
    - Configure the pin as **input** by clearing the direction bit.
    - Configure the pin as **output** by setting the direction bit.
4. **Configure pull-up/pull-down resistors** and initial output state (Optional).

These steps allow the pin to be used for reading (input) or driving signals (output).

1h) Define GPIO in a Cortex-M based micro controller

In a ARM Cortex-M microcontroller, GPIO (General Purpose Input/Output) refers to programmable pins used for digital input and output operations. GPIO are configurable pins that can be set as **input** to read signals (e.g., switches) or as **output** to control external devices (e.g., LEDs). Each GPIO pin can be individually

controlled through registers, making them essential for interfacing the microcontroller with external hardware.

1i) Discuss the need for timing interfaces in real time applications

In real-time applications using ARM Cortex-M, timing interfaces are essential to ensure tasks are executed at the correct time. Timing interfaces (like timers and counters) are needed to generate precise delays, schedule tasks, and synchronize events. They help maintain accurate timing control, which is critical in applications such as communication systems, control systems, and embedded automation where operations must occur within strict time constraints.

1j) Why is frequency pre-scaling required.

Frequency pre-scaling is required in microcontrollers like ARM Cortex-M to reduce the input clock frequency supplied to timers. This allows timers to count more slowly, enabling the generation of longer delays and accurate timing intervals without overflow. It also helps in matching high-speed system clocks with low-speed timing requirements.

The tradeoff between frequency resolution and range can be achieved using a frequency prescaler, which divides the time of the frequency input signal before feeding it to the counter.

PART-B

2a) Explain Assembly language development tools.

In embedded systems like ARM Cortex-M, assembly language development tools are used to write, translate, test, and debug programs.

Assembly language development involves several tools, each performing a specific function:

1. **Editor**
2. **Assembler**
3. **Linker**
4. **Locator**
5. **Debugger**
6. **Emulator**

Editor:

- An editor is a program which allows you to create a file containing the assembly language statements for your program.
- When you have typed in your entire program, you then save the file on a hard disk.
  This file is called source file.
- The next step is to process the source file with an assembler.

- If you are going to use the TASM or MASM assembler, you should give your source file name the extension .ASM.

Assembler:

- An assembler is programming tool which is used to translate the assembly language mnemonics for instructions to the corresponding binary codes.
- The assembler generates two files.
- The first file, called the *object file*, is given the extension .OBJ.
- The object file contains the binary codes for the instructions and information about
- the addresses of the instructions.
- After further processing the contents of this file will be loaded into memory and run.
- The second file generated by the assembler is called the *assembler list file* and is given the extension. LST.

Linker:

- The linker is program used to join several object files into one large object file.
- The linkers which come with the TASM or MASM assemblers produce link files with the .EXE extension.

Locator:

- A locator is a program used to assign the specific addresses of where the segments
- of object code are to be loaded into memory.

Debugger:

- If your program requires no external hardware or requires only hardware accessible directly from your microcomputer, then you can use debugger to run and debug your program.
- A debugger is a program which allows you to load your object code program into system memory, execute the program, and troubleshoot or' debug' it.

Emulator:

- Another way to run your program is with an emulator.
- An emulator is a mixture of hardware and software.
- It is usually used to test and debug the hardware and software of an external system.

2b) Compare minimum mode and maximum mode configuration.

Minimum mode	Maximum mode
In minimum mode there will be only one processor i.e. 8086.	In maximum mode there can be multiple processors with 8086, like 8087 and 8089.
MN/MX is 1 to indicate minimum mode.	MN/MX is 0 to indicate maximum mode.
ALE for the latch is given by 8086 as it is the only processor in the circuit.	ALE for the latch is given by 8288 bus controller as there can be multiple processors in the circuit.
Direct control signals M/IO,RD and WR are given by 8086.	Instead of control signals, each processor generates status signals called S2,S1 and S0

Control signals M/IO,RD and WR are decoded by a 3:8 decoder like 74138.	Status signals S2,S1 and S0 are decoded by a bus controller like 8288 to produce control signals.
INT A is given by 8086 in response to an interrupt on INTR line.	INT A is given by 8288 bus controller in response to an interrupt on INTR line.
HOLD and HLDA signals are used for bus request.	RQ/GT,lines are used for bus requests by other processors like 8087 or 8089.
The circuit is simpler.	The circuit is more complex.
Multiprocessing cannot be performed	Multiprocessing can be performed
Since it doesn't perform Multiprocessing hence performance is lower.	Since it performs Multiprocessing hence performance is very high

3a) Explain various addressing modes with examples.

(Consider any 4)

- Addressing Modes are way of indicating data or operands.
- Depending upon the data types used in the instruction and the memory addressing modes, any instruction may belong to one or more addressing modes, or some instruction may not belong to any of the addressing modes.
- Thus the addressing modes describe the types of operands and the way they are accessed for executing an instruction.

Various Addressing Modes:

1. Immediate addressing mode
2. Direct addressing mode
3. Register addressing mode
4. Register indirect addressing mode
5. Indexed addressing mode
6. Register relative addressing mode
7. Based indexed addressing mode
8. Relative based indexed
9. Intersegment direct
10. Intersegment indirect
11. Intrasegment direct mode
12. Intrasegment indirect mode

1. **IMMEDIATE ADDRESSING MODE**
    - Source data is within the instruction
    - Ex: MOV AX,0002H

      AL=02H, AH=00H
2. **DIRECT ADDRESSING MODE**

   In the direct addressing mode, a 16-bit memory address (offset) directly specified in the instruction as a part of it.

Example: MOV AX, [5000H].

3. **REGISTER ADDRESSING MODE**

   In the register addressing mode, the data is stored in a register and it is referred using the particular register.

   Example: MOV BX, AX

4. **REGISTER INDIRECT ADDRESSING MODE**

   The address of the memory location which contains data or operands is determined in an indirect way, using the offset registers.

   Example: MOV AX, [BX].

5. **Indexed Addressing Mode**

   In indexed addressing mode, the **effective address (EA)** of the operand is obtained by adding an **index register (SI or DI)** to a displacement (optional). It is useful for handling sequential data.

   Example:

   MOV AL, [SI]

   Here, the content of memory location pointed by **SI register** is moved into AL.

6. **Register relative addressing mode:**

   In this addressing mode, the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the register BX, BP, SI & DI.

   Example: MOV AX, 50H [BX]

7. **Based indexed addressing mode:**

   The effective address of data is formed in this addressing mode, by adding content of a base register (any one of BX or BP) to the content of an index register (any one of SI or DI).

   Example: MOV AX, [BX][SI]

8. **Relative based indexed:**

   The effective address is formed by adding an 8 or 16-bit displacement with the sum of contents of any of the base registers (BX or BP) and any one of the index registers, in a default segment.

   Example: MOV AX, 50H [BX] [SI]

9. **Intrasegment indirect mode:**

   Here, the branch address is found as the content of a register or a memory location.

   Example: JMP [BX]; Jump to effective address stored in BX.

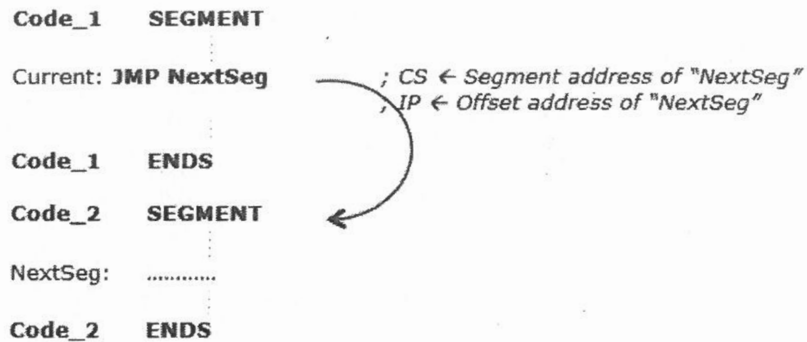
10. **Intersegment indirect:**

    In this mode, the address to which the control is to be transferred lies in a different segment and it is passed to the instruction indirectly.

    Example: JMP [2000H]. Jump to an address in the other segment specified at effective address 2000H.

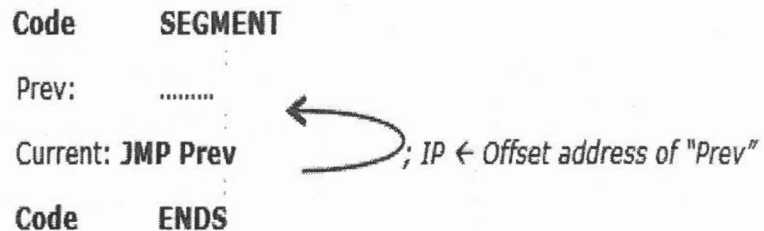
11. **Inter Segment Direct Addressing Mode**

Example:



12.**Intra segment direct:**

xample:



3b) Develop an Assembly language program to add two 16-bit numbers.

```
ASSUME CS: CODE, DS: DATA
DATA  SEGMENT
OPR1 DW 0BCDEH
OPR2 DW 0ABCDH
RES DW 1 DUP (0H)
DATA ENDS
CODE  SEGMENT
START:
MOV AX, DATA
MOV DS, AX
MOV AX, OPR1
MOV BX, OPR2
ADD AX, BX
MOV RES, AX
INT 03H
CODE ENDS
END START
END
```

4a) Describe the Bus system and Bus matrix organization in Cortex-M

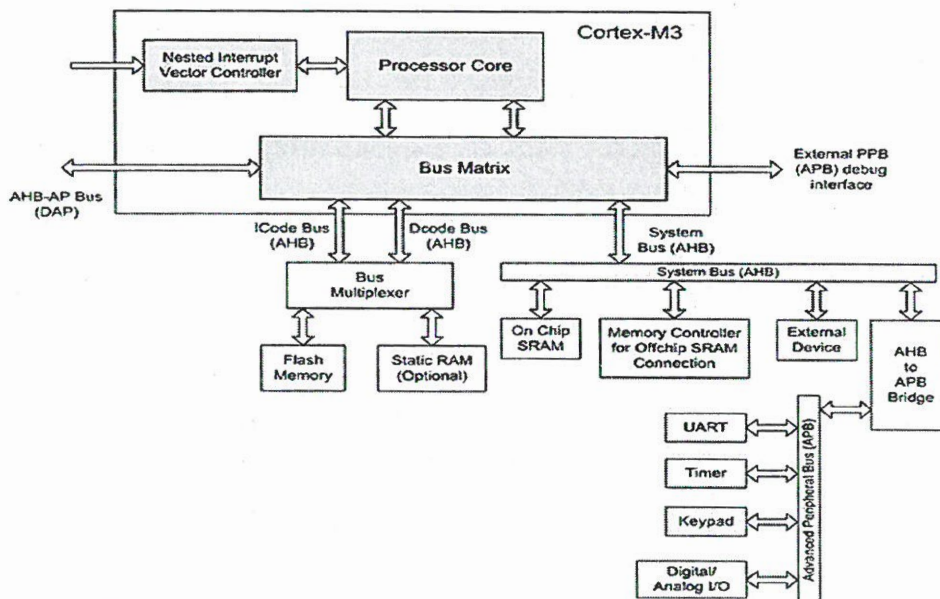


Figure : Bus system for memory, peripherals, and debug interconnection.

In ARM Cortex-M, the bus system and bus matrix are used for data transfer between components.

The **Bus System** is a set of communication lines (address, data, and control buses) that connect the CPU, memory, and peripherals, enabling data transfer.

The **Bus Matrix** is an advanced interconnection system that allows **multiple bus masters (like CPU, DMA)** to access different slaves (memory/peripherals) simultaneously, improving performance by enabling parallel data transfers and reducing bottlenecks.

- Components are connected together through an Advanced High-Performance Bus (AHB) and an Advanced Peripheral Bus (APB).
- An AHB-to-APB bus bridge is used to connect I/O interfaces, timer, keypad, universal asynchronous receiver/transmitter (UART), etc. to the processor core through system bus (AHB) and bus matrix.
- A bus matrix is used at the heart of the Cortex-M3 internal bus system as can be seen in the figure.
- The bus matrix effectively is an AHB interconnection network, allowing data and instruction code transfers to take place on different buses simultaneously unless both buses are trying to access the same memory region.
- Most processor implementations contain multiple external AHB interfaces and one APB interface.
- An internal AHB-to-APB bus bridge is used to connect a number of APB devices, such as debugging components, which follow the private peripheral bus interface.
- Various buses include:
  - ICode Bus
  - DCode Bus
  - System Bus
  - AHB-AP Bus
  - Private Peripheral Bus (PPB)

ICode Bus:

- The ICode bus is a 32-bit bus based on the AHB-Lite bus protocol.
- It allows to perform instruction fetches in memory regions from 0x00000000 to 0x1FFFFFFF.

- Instruction fetches are performed in word size, even for 16bit Thumb instructions.
- Therefore, the CPU core can fetch up to two Thumb instructions at a time.

DCode Bus:

- The DCode bus is a 32-bit bus based on the AHB-Lite bus protocol.
- Data access in memory regions from 0x00000000 to 0x1FFFFFFF can be performed via this bus

System Bus:

- The system bus is a 32-bit bus based on the AHB-Lite bus protocol.
- It is used for instruction fetch and data access in memory regions from 0x20000000 to 0xDFFFFFFF and from 0xE0100000 to 0xFFFFFFFF.
- Similar to the D-Code bus, all the transfers on the system bus are aligned.

AHB-AP Bus:

- The processor contains an AHB-AP (access port) interface for debug accesses.
- An external Debug Port (DP) component accesses this interface.

The Cortex-M3 system supports three possible DP implementations:

- The Serial Wire JTAG Debug Port (SWJ-DP)
- The Serial Wire DP
- No DP present

The Serial Wire JTAG Debug Port (SWJ-DP):

- The SWJ-DP is a standard Core Sight debug port that combines JTAG-DP and Serial Wire Debug Port (SW-DP).

The Serial Wire DP:

- This provides a two-pin interface to the AHB-AP for debugging purposes and is widely used in today's microcontrollers for debugging interface.

No DP present:

- If no debug functionality is present within the processor, a DP is not required.
- "The DP and AP together are referred to as the Debug Access Port (DAP)".

Private Peripheral Bus (PPB):

- The private peripheral bus is a 32-bit bus based on AMBA-based APB protocol.
- This is intended for private peripheral accesses in memory regions 0xE0040000 to 0xE00FFFFF.
- However, since some part of this APB memory is already used for different debug interfaces, the memory region that can be used for attaching extra peripherals on this bus is only 0xE0042000 to 0xE00FF000.
- Transfers on this bus are word aligned.

4b) Describe Interrupt Configuration in Cortex-M

- Interrupt configuration process has multiple components.
- Global interrupt configuration is applicable to all the interrupts, while local interrupt configuration is specific to that particular source of interrupt.
- Both local as well as global interrupt configurations are required for proper functioning of interrupts.
- In addition to interrupt configuration, we need to set up a table of interrupt vectors holding the information related to interrupt service routines.
- This table is called interrupt vector table and is used at the time of occurrence of an interrupt to determine where the corresponding service routine is located in the instruction memory.

Basic Interrupt Configuration

- There are two aspects related to interrupt configuration. The first aspect is the global configuration for the desired interrupt processing behavior.
- The following configurations are normally involved in setting up the interrupt behavior globally.
  - Interrupt/exception masking registers configuration
  - Setting up interrupt vector table
  - Configuring interrupt priority groups
- The second aspect is the device or peripheral specific interrupt configuration of the source of interrupt and can be configured using the associated registers.
- Some of the important local device specific configurations, based on the functionality, are listed below.
  - Enabling and disabling of interrupts locally
  - Interrupt pending control and status
  - Priority level configuration
  - Active status indication

Interrupt Masking

- There are three different interrupt/exception masking special registers in a Cortex-M processor.
- They are PRIMASK, FAULTMASK, and BASEPRI.
- These registers are useful for enabling or disabling interrupts and masking them based on assigned priority levels.
- These registers can only be accessed in privileged access level.
- In unprivileged access level: Write operation is ignored and Read operation returns zero.
- On reset, these registers are cleared to zero resulting in no interrupt masking.

PRIMASK Register:

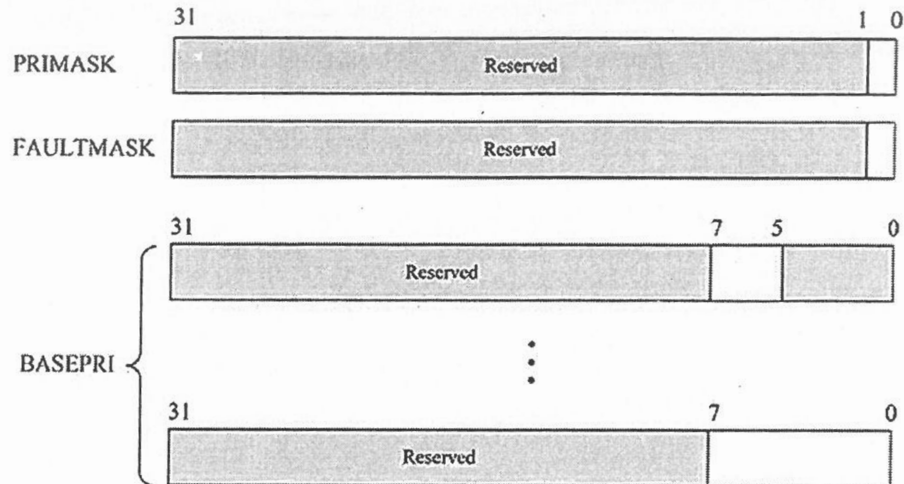
- PRIMASK is a 1-bit register (only least significant bit is used).
- When set, all exceptions/interrupts are blocked except Reset, NMI, and HardFault.
- Setting PRIMASK to 1 is equivalent to raising the priority level to 0.
- Commonly used to disable all interrupts during execution of a critical code section.

FAULTMASK Register:

- FAULTMASK is also a 1-bit register.
- Setting FAULTMASK to 1 allows only Reset and NMI but masks HardFault.
- Equivalent to setting the priority level to –1.
- Helps avoid further triggering of fault interrupts.
- Can be used to suppress bus faults.
- Automatically cleared when returning from an exception.

BASEPRI Register:

- BASEPRI provides flexibility in interrupt masking.
- Number of priority levels in Cortex-M can be between 8 and 256.Requires 3 to 8 bits in BASEPRI register.
- Actual number of bits is determined by the microcontroller manufacturer.
- Most Cortex-M3 and Cortex-M4 have 8 or 16 programmable priority levels.
- When BASEPRI is non-zero: Blocks interrupts of same or lower priority.
- When BASEPRI is zero: Masking is disabled.



5a) Explain exception handling mechanism in Cortex-M

In ARM Cortex-M, exception handling is used to manage interrupts and error conditions efficiently.
The Cortex-M uses a **vector table** to handle exceptions, where each exception has a predefined handler address. When an exception occurs, the processor automatically **saves the current context (registers)** and jumps to the corresponding handler. After execution, it restores the context and resumes normal program execution. This mechanism ensures fast and efficient interrupt handling.

- When an exception or an interrupt occurs, the processor uses handlers for exceptions or service routines for interrupts.
- User application programs are mostly responsible for implementing interrupt service routines.
- An operating system deals with the exception handlers.
- Based on this terminology, the following handlers and service routines are defined.
  - Interrupt Service Routines: External or peripheral interrupts (exceptions 16 to 255) are handled by interrupt service routines.
  - Fault Handlers: Used for Hard Fault, MemManage fault, Usage Fault, and Bus Fault.
  - System Handlers: NMI, PendSV, SVCall, SysTick, and the above-mentioned fault exceptions are considered system exceptions. Usually these handlers are implemented as part of the operating system.
- The Cortex-M3 processor uses a vector table that contains the addresses of the service routines. When an interrupt from a particular source occurs, the corresponding interrupt service routine is executed.
- After accepting an interrupt, the processor fetches the service routine address from the vector table using the instruction bus interface.
- It can be relocated to any arbitrary address using the associated configuration registers.
- When an exception takes place, the following sequence of operations is performed:

- Execution of the current instruction is either completed or terminated without completing, depending on the value of the interrupt continuable instruction (ICI) field in the xPSR register.
- The current status is preserved by pushing registers R0–R3, R12, LR, PC, and PSR onto the stack
- The processor reads the exception number field from the updated value of the xPSR register and performs vector fetch by reading the exception handler starting address from the interrupt vector table.
- Before execution of the interrupt service routine begins, the processor updates the stack pointer, link register (LR), and program counter (PC).The LR is loaded with a specific value to signify the type of interrupt return as will be explained in this section later.
- Specific interrupt service routine corresponding to the interrupt source is performed.

5b) Differentiate between Cortex-M and 8086 with respect to architectures.

Feature	Cortex-M Architecture	8086 Architecture
Type	RISC (Reduced Instruction Set Computer)	CISC (Complex Instruction Set Computer)
Data Width	32-bit architecture	16-bit architecture
Instruction Set	Simple, fixed-length instructions	Complex, variable-length instructions
Pipeline	Supports pipelining for faster execution	Limited pipelining (uses instruction queue)
Memory Architecture	Harvard architecture (separate instruction & data paths)	Von Neumann architecture (common memory for code & data)
Registers	Large register set (R0–R15 + special registers)	Limited registers (AX, BX, CX, DX, etc.)
Performance	High efficiency and low power	Lower efficiency compared to Cortex-M
Application	Embedded systems and real-time control	Early general-purpose computing systems

6a) Demonstrate LDR and STR memory access instructions.

- LDR and STR instructions allow data to be transferred between processor and memory.
- The load register, LDR, instruction is used to transfer data from memory to the processor register, while store register, STR, instruction is used to transfer data from processor register to memory.
- Following are the Memory access instructions

```
LDR R2, [R1]      ; load R2 with data from memory pointed to by
LDR R0, = NUM1    ; load R0 with the value of constant NUM1 (thi
                  ; constant might represent the memory address)
STR R4, [R3]      ; store R4 to a memory location addressed by R
```

- The first LDR instruction retrieves data from memory address specified by the value of register R1 and transfers it to register R2.
- The second LDR instruction loads the immediate value to the destination register and in that context it is functionally equivalent to the MOV instruction used for loading immediate value.
- These instructions differ in the maximum immediate value that can be used.
- The maximum permissible immediate value for MOV instruction is either 8-bit for 16-bit encoding or 12-bit for 32-bit encoding.
- The last instruction stores the contents of R4 to the memory location addressed by the value in R3.
- In Cortex-M architecture, memory access instructions include load-store and push-pop.
- Load register (LDR) instruction transfers data from memory to processor registers
- Store register (STR) instruction transfers data from registers to memory
- The LDM and STM instructions are used to load or store multiple data.
- If the processor is required to access a large memory segment sequentially, then one possible implementation is to maintain a pointer to the starting address of that memory segment.
- This pointer, which is the starting memory address of the memory segment, is made available in one of the registers of the processor.
- This address is termed the *base address.*
- An *offset* from this address can be used to generate a relative address and this mode of memory access is called *offset addressing.*

Immediate Offset Addressing:

- When the offset is an immediate value from the base address stored in a register, the resulting addressing mode is called immediate offset addressing
- The load register instruction is used to fetch 32-bits of information from the memory and put the value in the destination register Rt.
- By default, the LDR and STR instructions perform 32-bit data transfers.

Register Offset Addressing:

- In Register offset addressing mode, the base address is contained in a register while the offset value is also in a register.
- The offset is an immediate value.

6b) Illustrate various data processing instructions in Cortex-M.

(Consider any 4)

In **ARM Cortex-M**, data processing instructions are used to perform arithmetic and logical operations on data stored in registers.

- The addition operation is implementable using one of the three possible instructions depending on the requirement of operation

- The three different instructions available for the addition operation are ADD, ADDW, and ADC.
- The ADD instruction simply adds two numbers and can be encoded using either 16-bit or 32-bit formats.
- The ADDW instruction is always 32-bit encoded

  Following are the Syntax illustration for basic arithmetic instructions

```
ADD{S}{cond}{.size}   {Rd,}   Rn,   Operand2
ADC{S}{cond}{.size}   {Rd,}   Rn,   Operand2
SUB{S}{cond}{.size}   {Rd,}   Rn,   Operand2
SBC{S}{cond}{.size}   {Rd,}   Rn,   Operand2
RSB{S}{cond}{.size}   {Rd,}   Rn,   Operand2
ADDW{cond}            {Rd,}   Rn,   #imm12
SUBW{cond}            {Rd,}   Rn,   #imm12
```

```
ADD  R2, R1, R3       ; R2 = R1 + R3
ADC  R2, R1, R3       ; R2 = R1 + R3 + C, here C represent
                      ; value of carry flag
SUBS R8, R6, #240     ; R8 = R6 - 240, sets the flags on
                      ; the result
RSB  R4, R4, #1280    ; R4 = 1280 - R4,
```

- The subtraction operation is implementable using subtract (SUB, SUBW), subtract with carry (SBC), and reverse subtract (RSB) instructions.
- The functionality of SUB and SUBW is similar to ADD and ADDW except that the operation performed is subtraction.
- In SBC the current value of the carry flag is first inverted and then it is subtracted from the result of subtraction.
- In case of RSB instruction, the order of operands is reversed as will be clarified

Multiply and Divide Instructions: illustration of multiplication instruction syntax with 32-bit result

- The Cortex-M3 processors also support 32-bit multiplication instructions with or without accumulate and produce 64-bit results.
- Unsigned multiply long (UMULL), unsigned multiply with accumulate long (UMLAL), signed multiply long (SMULL), and signed multiply with accumulate long (SMLAL) instructions are supported by the Cortex-M3 hardware architecture.

MUL{*S*}{*cond*}   {*Rd*, }   *Rn*,   *Rm*
MLA{*cond*}   *Rd*,   *Rn*,   *Rm*,   *Ra*
MLS{*cond*}   *Rd*,   *Rn*,   *Rm*,   *Ra*

where

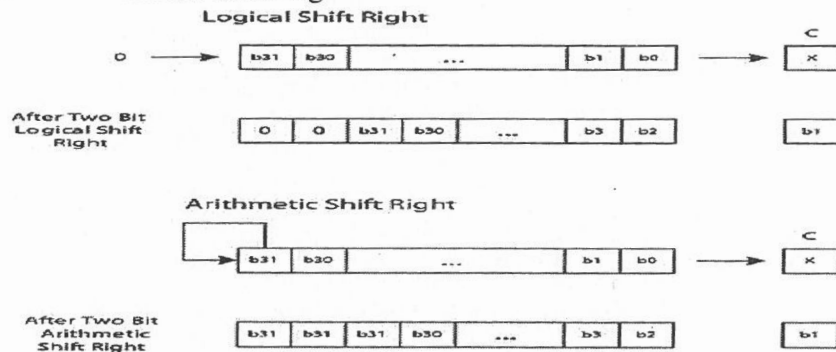
Rn, *Rm*	These are the two registers containing the values to be multiplied.
Rd	This is the destination register that holds the least significant 32-bits of the result of operation.
Ra	This is the register holding the value to be added to or subtracted from.

```
MUL    R1, R2, R5        ; Multiply, R1 = R2 x R5
MLA    R1, R2, R3, R5    ; Multiply with accumulate,
                         ; R1=(R2 x R3)+R5
MULS   R1, R2, R2        ; Multiply with flag update,
                         ; R1=R2 x R2
MLS    R1, R5, R6, R7    ; Multiply with subtract,
                         ; R1=R7-(R5 x R6).
```

Shift, Rotate & Logical Instructions:

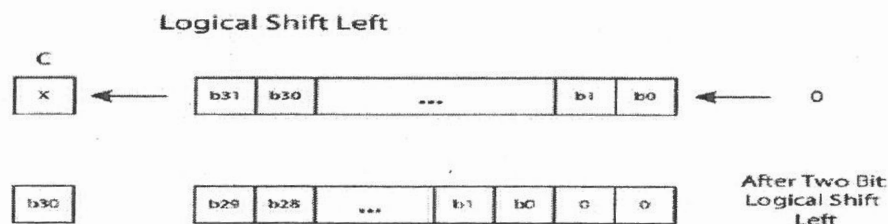
- Shift operations move the bits in a register left or right by a specified number of bits, termed the shift length.
- The Cortex- M3supports *logical shift left* (LSL), *logical shift right* (LSR), *arithmetic shift right* (ASR), *rotate right* (ROR), and *rotate right extended* (RRX).

Logical and Arithmetic Shift Right:



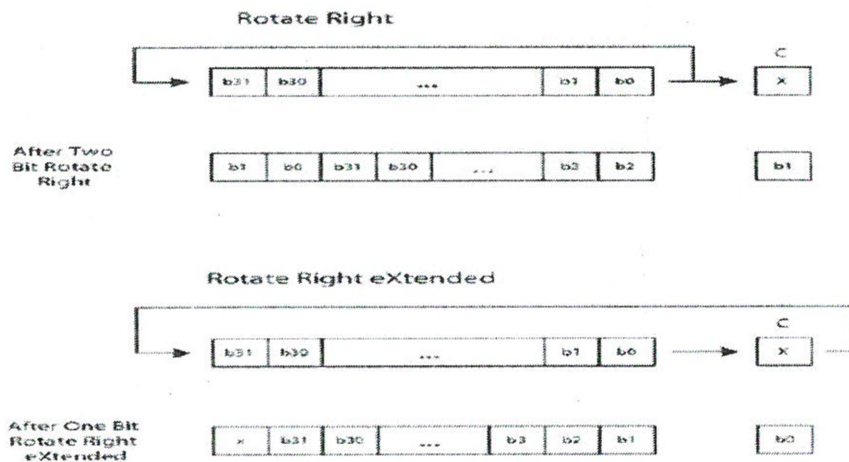
- The logical shift right (LSR) instruction shifts the bits in the register by the specified value and zeros are entered to the register from its left side.
- Arithmetic shift right (ASR) by n bits moves the left-hand 32- n bits of the register Rm to the right by n places, into the right-hand 32-n bits of the result.

Logical Shift Left:



- In logical shift left (LSL) instruction, the bits entering the register from least significant bit (LSB) side are always zero, while the bits exiting the register are discarded.
- Shift left by one-bit position leads to multiplication by 2.

Rotate Right and Rotate Right Extended:



- ➢ The rotate right (ROR) instruction rotates the contents by a specified value without making carry flag part of the rotation.
- ➢ The variable n is the number of bit positions to rotate and $1 \leq n \leq 31$
- ➢ The special rotate right extended (RRX) instruction has a slightly different behavior from the usual rotate operations.
- ➢ One key difference is that no count is specified and this instruction always rotates by one bit.

Logical Instructions:

- ➢ The logical instructions support bitwise AND, exclusive OR (EOR), simple OR (ORR), OR negative (ORN), and bit clear (BIC) operations
- ➢ The AND, ORR, and EOR instructions perform bitwise AND, OR, and Exclusive OR operations between instruction parameters Rn and Operand2

Data Movement Instructions:

```
MOV{S}{cond}    Rd,    Operand2
MOV{cond}       Rd,    #imm16
MVN{S}{cond}    Rd,    Operand2
MOVW{cond}      Rd,    #imm16
MOVT{cond}      Rd,    #imm16
```

- ➢ Data movement inside the processor can be performed using move (MOV and MOVW), move negative (MVN), and move top (MOVT) instructions.
- ➢ The MOV instruction copies the value from Operand2 to the register Rd.
- ➢ The MOVW instruction provides the same functionality as that of MOV instruction
- ➢ The MVN instruction takes the value of Operand2

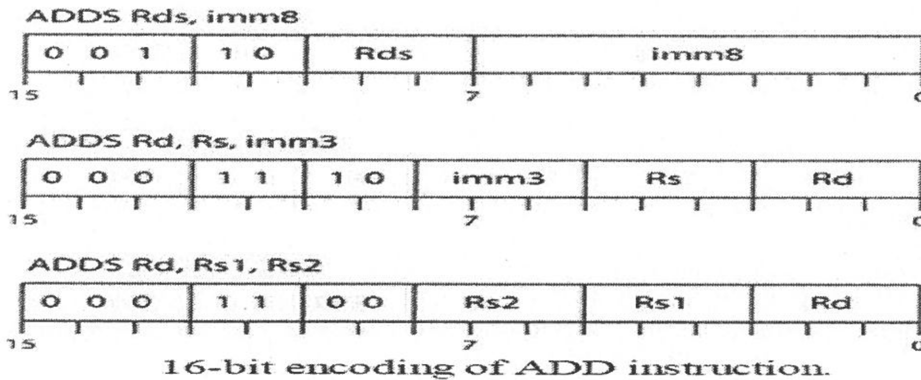
7a) Demonstrate 16 bit instruction encoding in Cortex-M

- ➢ Instruction encoding is the process of assigning a unique binary codeword to each assembly instruction.
- ➢ The main purpose of an assembler is to translate each assembly instruction to an equivalent code word.
- ➢ The assembly instructions are encoded to 16-bit code words, while some other instructions are encoded using 32bit code words.

```
ADDS R2, #16          ; R2 = R2 + 16, Instruction is 16-
bit encoded
ADDS R4, R2, #6       ; R4 = R2 + 6, Instruction is 16-
bit encoded
ADDS R6, R5, R3       ; R6 = R5 + R3, Instruction is 16-
bit encoded
```

ADD instruction illustration for 16-bit encoding.

- ➢ The first two instructions in illustrate two possibilities resulting in 16-bit encodings, primarily due to the appropriate choice of immediate values.
- ➢ The third instruction, illustrating register to register addition, is also encoded using 16-bit encoding



16-bit encoding of ADD instruction.

- ➢ If the same register is used as source and destination operand, the immediate value can have an 8-bit size.
- ➢ Destination and source registers allows only a 3-bit immediate value operand.

7b) Demonstrate the execution flow of a loop using branch instructions.

- ➢ The software development process effectively translates the desired task to a corresponding explicit set of instructions that can be executed by the processor.
- ➢ To handle these complexities associated with the real-life software programs, the processor architecture supports branch and control instructions.
- ➢ In **ARM Cortex-M**, loops are implemented using **branch instructions** that repeatedly execute a block of code until a condition is met.

Execution Flow of a Loop:

1. Initialize a loop counter in a register.
2. Execute the instructions inside the loop.
3. Update/decrement the loop counter.
4. Use a compare instruction (CMP) to check the condition.
5. Use a conditional branch instruction (BNE, BEQ, etc.) to jump back to the loop start if the condition is not satisfied.
6. Exit the loop when the condition becomes false.

The for Loop:

- ➢ A *for loop* consists of initialization, condition check, execution, and update.
- ➢ A for loop is employed when the number of iterations for executing a set of instructions is predetermined.
  - ➢ Syntax for (i=0; i<50; i++)
    Array [i] ='*';

Execution Flow:

1. Initialize loop variable
2. Check condition using CMP
3. If condition is false → branch to end
4. Execute loop body
5. Update loop variable
6. Branch back to condition

The While Loop:

A *while loop* checks the condition before execution.

```
count=0;
 i=0;
while (Array([i]= !='\n')
count ++
```

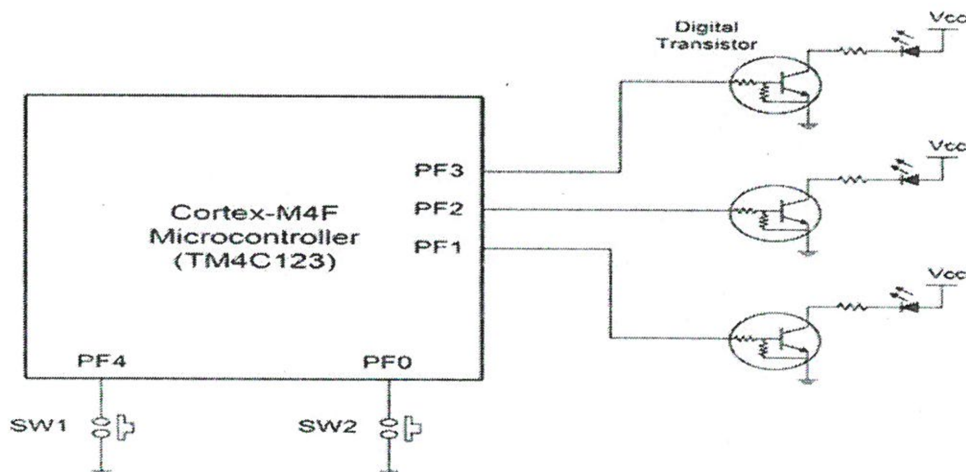
Execution Flow:

- Check condition first
- If false → exit loop
- Execute loop body
- Branch back to condition
- Consider a scenario where the number of loop iterations is not known in advance. Rather, the loop is iterated until a certain condition is fulfilled. For such situations, a while loop is used.

8a) Analyze LED and switch interfacing with Cortex-M with neat diagram.

- The TM4C123 (TI Tiva-C ARM Cortex-M4F) board contains three on-board LEDs (Red, Green, Blue) and two switches (SW1, SW2).
- LEDs are configured as digital outputs, and switches are configured as digital inputs using GPIO pins.
- LEDs are connected through digital NPN transistors to prevent GPIO current overloading and to ensure operation in cut-off and saturation regions.
- Switches use the internal pull-up resistor of TM4C123.Switch released → Logic HIGH, Switch pressed → Logic LOW.
- Alternatively, pull-down resistors or PNP transistor LED configuration can be used, which reverses the driving logic.

Fig: Connection diagram for TM4C123 on board LEDs and switches



Output Interfacing for LED:

For LED interfacing we need to configure Port F pins 1, 2 and 3 as outputs. The basic steps to configure the required Port F pins are given below.

- Enable the clock to the GPIO port.
- Select the general purpose IO functionality by clearing the alternate function select register (GPIOAFSEL).
- Configure the GPIO direction for digital output function.
- Configure the PAD for the digital operation.

Input Interfacing for Switch:

Mechanical switches are commonly used to feed any parameters to the digital systems. The switches can be interfaced to a microcontroller using digital inputs. The software program for switch interfacing can be implemented using one of the following two methods.

- Polling Based
- Interrupt Based

Polling Based:

- In case of polling based method the GPIO pin connected with the switch is polled

frequently enough, in the software, to avoid missing any key presses.

- By frequently enough, it is meant that the time interval between two consecutive read

operations of the GPIO port pin connected with the switch and determining whether the switch is pressed or not, is smaller than the minimum time the switch is kept pressed by the user.

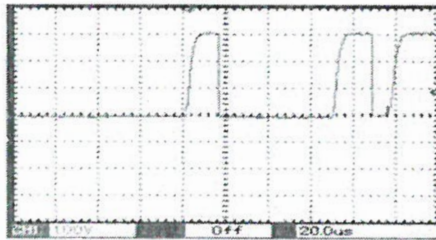
Interrupt Based:

- In this method the GPIO pin is configured as an external interrupt and any key press leads to an interrupt.
- The edge triggering of the interrupt can be configured on rising or falling edge, depending on the switch hardware connectivity.

Switch Bouncing:

- Electrical switches that use mechanical contacts to close or open a circuit are subject to bouncing of the contacts.
- Switch inputs are asynchronous and are not electrically clean.
- Switch contacts are usually made of springy metals.
- When the contacts strike together, their momentum and elasticity act together to cause them to bounce one or more times before making steady contact.
- It results in a pulsating electric signal instead of a clean transition.
- For a microcontroller GPIO pin configured as input, this switch bouncing can be interpreted as multiple switch presses.
- Both hardware as well as software solutions exist to get around the switch bouncing problem.

➢ Analog filtering using an RC delay to filter out the rapid changes in switch output can be used as hardware solution, as shown in Figure.



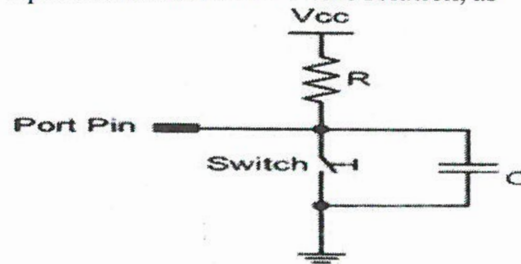


Fig (a): The switch bouncing when the switch is released Fig(b) : The hardware denouncing of the switch

➢ The design task is to choose R and C such that the input threshold is not crossed while bouncing is still occurring.
➢ In case of software solution to the switch bouncing problem, at the beginning of polling the switch is checked for being pressed.
➢ If the switch is detected as pressed, then a delay is inserted and the switch is checked again for being pressed.
➢ If the switch is detected as pressed again, we declare that the switch is pressed.
➢ The rate at which the switch should be polled has to be fast enough that no switch press is missed.

Switch Interfacing Illustration:

➢ There are two user switches on TM4C123 evaluation board.
➢ User switch labeled SW1 is connected to Port F pin 4 (abbreviated as PF4), while
➢ the second switch SW2 is connected to Port F pin 0 (PF0).
➢ The basic configuration steps are similar to the LED interfacing.
➢ Enable the clock to the GPIO port.
➢ Select the general purpose IO functionality by clearing the alternate function select register (GPIOAFSEL).
➢ Configure the GPIO direction for digital input.
➢ Configure the PAD for digital operation.
➢ In addition, enable the internal pull-up resistor by setting the corresponding bit of GPIOPUR register.
➢ This is required for proper functioning of the switch, as no external pull-up resistor is placed on the board

8b) compare and contrast serial communication and parellel in UART communication

Feature	Serial Communication (UART)	Parallel Communication
Data Transfer	Transfers data **one bit at a time**	Transfers **multiple bits simultaneously**
Number of Lines	Uses **single data line (TX/RX)**	Uses **multiple data lines**
Speed	Slower compared to parallel	Faster for short distances
Distance	Suitable for **long-distance communication**	Limited to **short distances**
Cost & Complexity	**Low cost and simple** wiring	**More complex and costly**
Example	UART, RS-232	Data buses in computers

Serial Communication

- Uses single line for data transfer.
- Simpler wiring.
- Suitable for longer distances (hundreds of meters).
- Slower than parallel communication.
- Requires:
  - Parallel-to-Serial converter (transmitter)
  - Serial-to-Parallel converter (receiver)
- Data is transferred one bit at a time.

Parallel Communication

- Uses multiple lines to transfer data.
- Faster data transfer.
- Requires more wiring.
- Suitable for short distances (few meters).
- Example uses:
  - Memory interfaces
  - Graphics displays

9a) Analyze how to interface a segment display ad TM4C123 microcontroller.

- ➢ A seven-segment display consists of seven LED segments (a–g) and an additional decimal point (DP) LED. By selectively turning ON segments, decimal, hexadecimal, and some special characters can be displayed.
- ➢ All segment LEDs share a common terminal, which determines the type of display: common anode or common cathode, as shown in above Fig.
- ➢ In a common cathode display, all cathodes are connected to ground, and required segments glow when logic high is applied. In a common anode display, the common terminal is connected to logic high, and segments glow when logic low is applied.
- ➢ Different digits are formed by turning ON specific segment combinations (e.g., to display digit 2, segments a, b, d, e, g are ON). The segment patterns for digits and characters are summarized in tables (0–9, A–F).

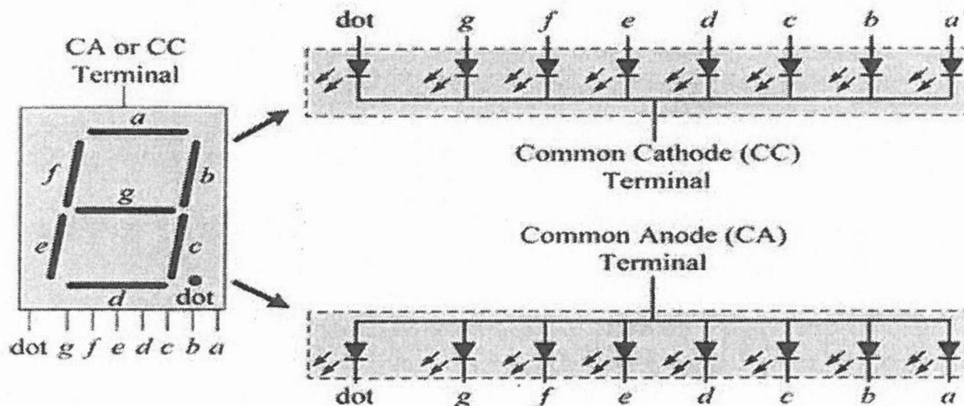


Fig: Common anode and common cathode configurations for seven segment display

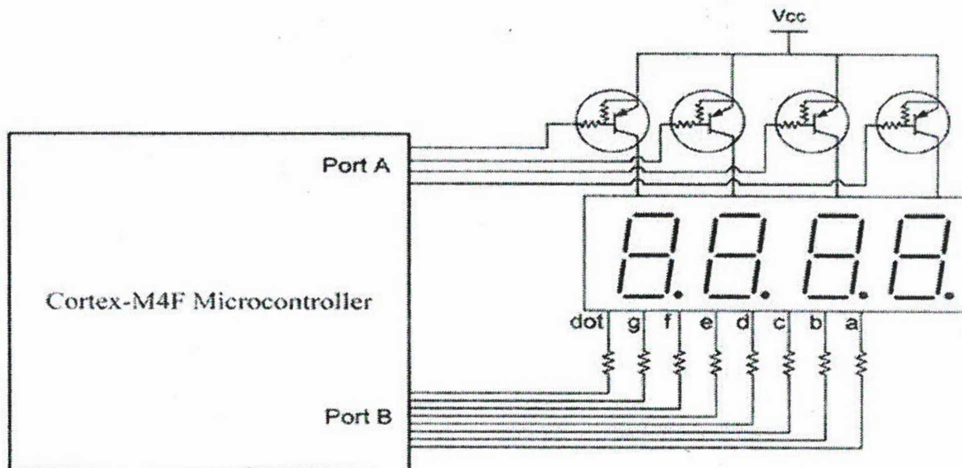


Fig: Four-digit seven-segment display connections using time multiplexing. The display used is of common anode type

The figure shows interfacing of a four-digit seven-segment display with a ARM Cortex-M4F using time multiplexing, and the display is of common anode type.
Basic connection:

- The display has 4 digits, each with segments (a–g and dot).
- Port B of the microcontroller is connected to segment lines (a–g, dp) through current-limiting resistors.
- **Port A** is connected to the **common anodes** of each digit via **transistors** (used as switches).
- The common anodes are connected to **Vcc**.

Working principle (Common Anode):

- In a common anode display, a segment glows when its cathode is LOW (0).
- So:

  Segment ON → logic 0 from Port B
  Segment OFF → logic 1

Time multiplexing Operation:
Only one digit is activated at a time by enabling its transistor (Port A).
The microcontroller:

1. Activates Digit 1 → sends segment data
2. Quickly switches to Digit 2 → sends new data
3. Repeats for Digit 3 and Digit 4

This process happens very fast (milliseconds), so all digits appear ON continuously due to persistence of vision.
Transistors act as switches to enable one digit at a time. They provide sufficient current to drive each digit.

9b) Analyze how to configure ADC for analog input conversion

- Enable ADC clock gating control.
- Configure GPIO pins as ADC inputs.
- Configure ADC module settings.
- Configure ADC sample sequencer.
- Configure ADC interrupts.

  Therefore, The **TM4C123 ADC** uses **two 12-bit SAR** ADC modules with 12 analog inputs, 1 Msps sampling rate, and four programmable sample sequencers with FIFO buffers for efficient multi-channel analog signal conversion.

ADC Clock Gating Control Configuration:

- The **TM4C123 microcontroller** has **two ADC modules**: **ADC0** and **ADC1**.

- The clock for each ADC module is controlled using the **RCGC_ADC_R register Bit assignments:**
- **Bit 0 → ADC0**
- **Bit 1 → ADC1**
- Writing **1** to the corresponding bit **enables the clock** for that ADC module.
- After enabling the clock, a **delay of 3 clock cycles** is required **before accessing ADC**

registers.

- Example usage: Setting **Bit 0 = 1** enables the clock for **ADC0.**

GPIO Configuration as ADC Input

- To use a **GPIO pin as an analog input**, digital functionality must be disabled first. Steps:
- **Clear the corresponding bit in GPIO_DEN_R register** to disable digital function.
- **Set the corresponding bit in GPIO_AMSEL_R register** to enable analog mode.
- **GPIO_AMSEL_R register offset: 0x528 from the GPIO port base address.**
- **Bits 0–7** of this register correspond to **GPIO pins 0–7.**
- Each pin can be **individually configured for analog functionality.**
- To enable analog functionality on a GPIO pin, the corresponding bit in the GPIO_DEN_R register must be cleared to disable the digital function.

ADC Sample Sequencer Configuration:

- The selection of an analog input channel for each sequencer is completely independent and can be performed separately. It is also possible to configure same analog input for multiple channels of the same sequencer. Each sample sequencer analog input can be configured for single ended or differential type of input.
- When using a sequencer, it is not necessary to use all of its channels. Rather any arbitrary channel, of the sequencer, can be configured for end of sequence. For instance, any of the channels from 0 to 7 can be configured as end of sequence for sequencer 0. When a sequencer is triggered, sampling is started at the programmed sampling rate and continues until end of sequence has been reached for that sequencer.

ADC Interrupt Configuration:

Interrupt configuration for each analog input channel can be performed separately as mentioned in the description of ADC sequencer control configuration register. In addition, each sequencer can be a source of interrupt as well, which can be configured using the ADC interrupt configuration registers.

10a) Analyze basic timing interfaces in TM4C123 microcontroller.

- Each microcontroller has two types of timing interfaces, one used to generate the system clock to operate the microcontroller at desired frequency, and the second based on a timer which is primarily a peripheral module.
- One primary use of a timer module is to generate timing intervals of desired value.
- The timer module can be configured to operate either as an input or an output device.
- When used as an input device, it can be configured to time or count external events.
- When used as an output device, it can generate signals of varying duty cycle as well as frequency.

- At the basic level, the timer can be used to generate accurate timing signals, measure time intervals, generate interrupts at specific time intervals, and count events of interest.
- Timer overflow sets a timer overflow flag, which can be polled by the user or can generate a timer interrupt (when configured), and this flag must be cleared by user software.
- The user can measure time elapsed since the previous overflow by reading the current timer value.
- If counting down, the timer underflows at 0x0000, and after underflow, the counter is loaded with 0xFFFF or a user specified reload value to start the next count down.
- The underflow condition also sets the timer overflow flag, and the basic timer block diagram uses either a 16-bit or 32-bit counter with a register holding the counter reload value.

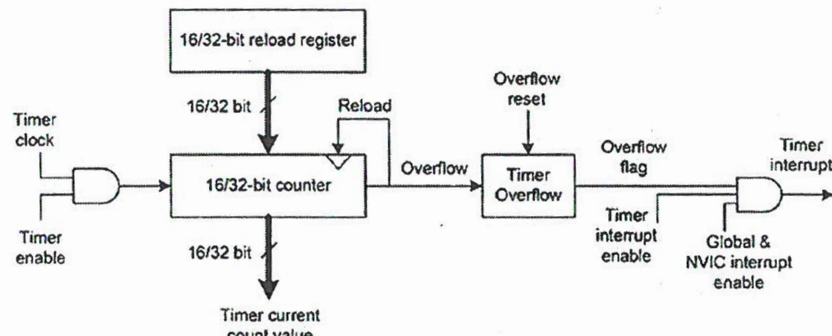


Figure : Basic timer block diagram with reload and timer interrupt capability.

- A 16-bit timer is assumed to be operating in count down mode with reload value 0xFFFF and clock input frequency 4 MHz.
- $2^{16}/4x10^{6}$=16.384ms
- For generating a timer interrupt every 10 ms, two possible choices exist:
- Change the frequency of the input clock signal to 6.5536 MHz.
- Start the timer count down from 0x9C40 (40000) instead of 0xFFFF.
- The second option, i.e., loading the timer with 0x9C40 (40000), is more viable for generating the desired timer overflow interval.
- The timer is required to generate a timer interrupt every Δt seconds, operating at frequency f Hz with w-bit width in count down configuration.
- The reload value X_d is obtained such that the timer underflows after reaching 0x0000 to produce the required interrupt interval.

  X_d=f*delta t
- When the timer is counting up, it overflows after reaching 0xFFFF, and the corresponding reload value $x_u$is obtained using the modified expression.

  $X_u=2^w$ - f*delta t

10b) Evaluate the generation of pulse width modulation using timer as an output device.

- Timers as an output device, the main purpose of a timer is to generate signals with desired attributes.
- A timer can generate a periodic signal (square wave shape) of certain frequency and can also generate pulses of varying width.
- A timer can be used to generate time delays.

Pulse Width Modulation:

- Pulse width modulation (PWM) is a technique to generate variable width pulses while maintaining a constant signal frequency.

- The time period t_p and the pulse width t_w must be specified such that $t_w < t_p$ and the duty cycle d is defined as $d = t_w / t_p \times 100$
- In a PWM signal, the frequency remains the same and only the duty cycle

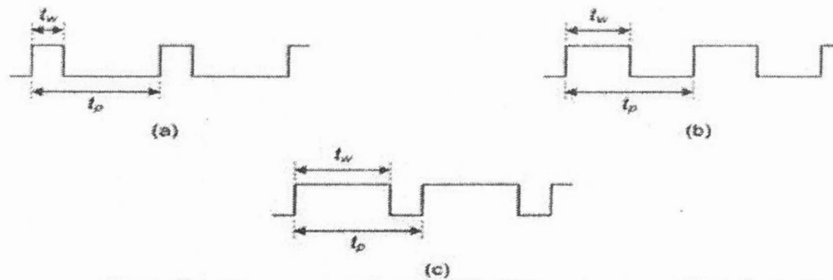


Figure : Same frequency PWM signal with different duty cycle values, (a) 25%, (b) 50%, and (c) 75%.

changes.

- The below PWM block diagram uses the system clock frequency, which can be scaled using an optional frequency prescaler before feeding it to the counter.
- The timer is configured as a down counter, and when it starts with count value 0, it is loaded with the reload value proportional to the PWM time period.
- The reload signal sets the flip-flop output to logic high, and the counter begins decrementing on each clock cycle.
- The counter value is continuously compared with the duty cycle value, and when the counter counts below the duty cycle value, the comparator output resets the flip-flop.
- When the counter reaches 0, an underflow (overflow condition) occurs, generating the reload signal and completing one PWM cycle.
- The overflow condition reloads the counter and starts the next PWM cycle
- The timer reload value 0xzzzz determines the frequency (or alternatively time period) of the PWM signal, while the pulse width is determined by subtracting the value in duty cycle register from 0xzzzz.
- The PWM waveform is generated by comparing the duty cycle value against the timer current count, and if the duty cycle register value is updated at the beginning of each PWM cycle, the resulting PWM signal can have an arbitrary pattern embedded in the duty cycle variations.
- If the duty cycle register values are updated following a sinusoidal pattern, the resulting PWM signal will have pulse widths varying sinusoidally, which is done in PWM based DC to AC inverters.
- When the counter operates in countdown mode, the counter values plotted as a function of time appear as a sawtooth waveform with negative slope, whereas in count up mode the counter values appear as a sawtooth waveform with positive slope.
- The PWM generated using a sawtooth signal is called edge aligned PWM, while the PWM signal generated using a triangular waveform is called center aligned PWM.
- The triangular wave shape is constructed by configuring the timer as an up-down counter, where the counter first counts up to a specified value and then counts down to zero to complete one cycle of the PWM signal.

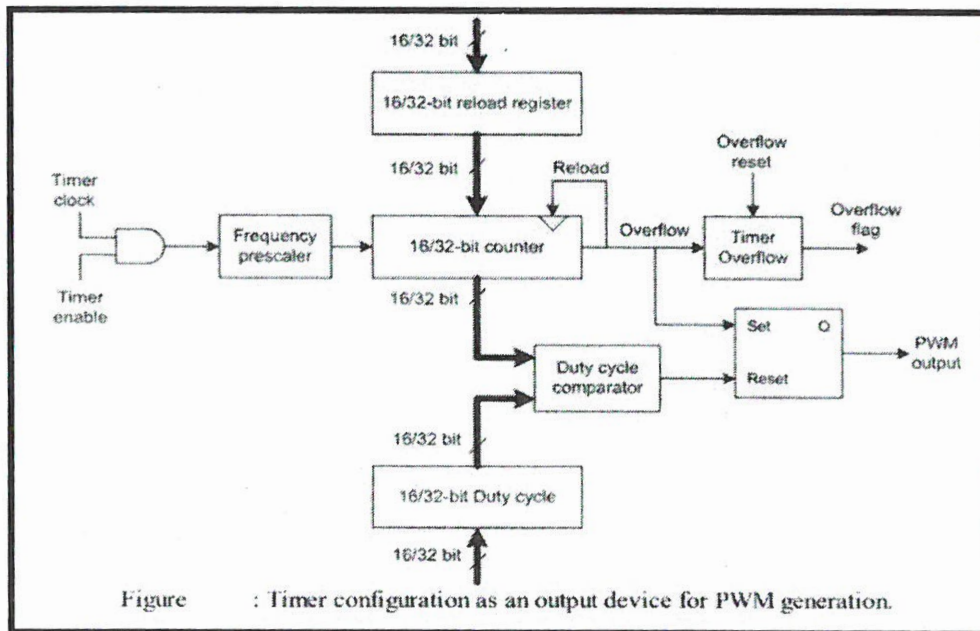


Figure : Timer configuration as an output device for PWM generation.

- PWM frequency f_{PWM} is configured by using an appropriate timer reload value for a given timer clock frequency f_{clk} and for given PWM and timer clock frequencies, the reload value t_{rl} is obtained as $t_{rl} = f_{clk} / f_{pwm}$

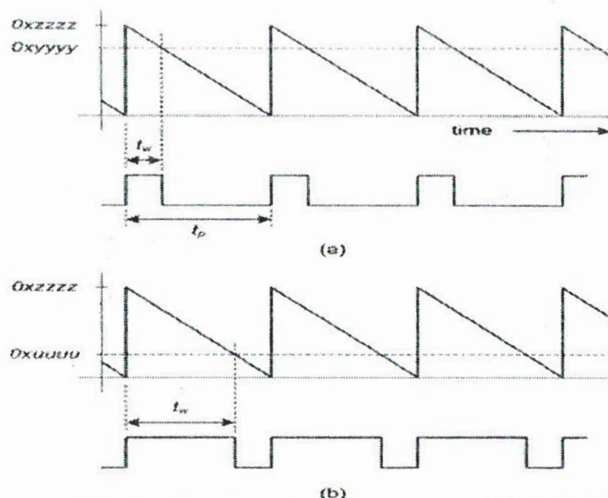


Figure PWM signal generation illustration for two different duty cycle values. For a given timer clock input frequency, the PWM frequency is determined by the reload value 0xzzzz. The value 0xyyyy in (a) corresponds to 25% duty cycle, while 0xwwww in (b) results in 75% duty cycle.

- The PWM resolution determines the minimum change in the duty cycle and is quantified in bits of units; for instance, an 8-bit PWM signal will have the smallest pulse of $\frac{100}{256}$ % duty cycle, which is the smallest possible change in the duty cycle value, and using 10-bit PWM provides four times better resolution than 8-bit PWM
- Since the parameter t_{rl} corresponds to 100% duty cycle of the PWM signal, it can be related to the PWM resolution (r_{PWM} in bits) as $r_{pwm} = \log_2 (t_{rl})$
- Combining the expressions, the relationship between PWM frequency and PWM resolution (in bits) is given by $r_{pwm} = \log_2 (f_{clk} / f_{pwm})$

11a) Analyze the working of Systick timer in TM4C123 microcontroller.

- The TM4C123 microcontroller is equipped with different timer modules having varying capabilities, and an application may use a timer independently or multiple timers in combination.
- A specific timer mode of operation may not be available on each timer module, requiring an intelligent assignment of timers.
- The Systick timer is integrated as part of the Cortex-M processor and is referred to as the system timer.
- All other timer modules are integrated as peripheral devices, and their memory space (except Systick timer) is allocated in the peripheral address space.
- The Systick timer is assigned memory space in the system address space, and Systick and general purpose timer modules are discussed in detail with practical applications.
- The Cortex-M processor integrates a system timer called Systick, which is based on a 24-bit down counter and reloads on counting to zero (wrap-on-zero), and its current value is cleared by write operation (clear-on-write).
- The Systick timer has a simple control configuration and can generate timer ticks of specified interval for RTOS, such as 10 ms timer ticks for scheduling tasks.
- Systick can be used as a simple counter for measuring time elapsed while executing a task and can also be configured as a high speed alarm timer based on the system clock.
- The Systick timer has three registers for its proper functioning, with their memory address allocation, reset values and descriptions provided, and the bit field allocations shown separately.

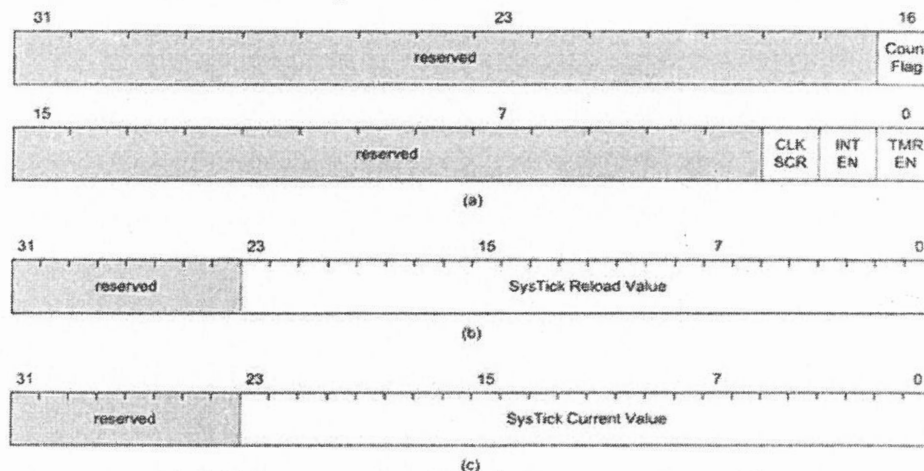


Figure : Bit field allocations for Systick timer registers. (a) Systick control (STCTRL) register, (b) reload value register (STRELOAD), and (c) current value register (STCURRENT).

- The Systick reload and current value registers use least significant 24 bits for holding their values, and to generate timer ticks every N clock cycles, the reload register should be initialized with $N - 1$.
- The maximum reload value that can be configured is 0x00FFFFFF, and the control and status register bit field descriptions are provided separately.
- The Systick current value register can be read at any time, but it is clear-on-write; writing any value clears the register and also clears the 'Count Flag' bit in STCTRL register.
- The Systick timer can be configured to generate 10 ms timer interrupts, which can be used as timer ticks by the system software.
- The Systick timer is a system timer with limited capability.

- The peripheral timer modules integrated in the TM4C123 microcontroller are highly flexible with powerful features.
- This flexibility and feature set make the use of these timer modules more complex.
- The timer modules have further capabilities and can be used as an input device as well as an output device.

11b) Design the steps to generate variable frequency signals using timers.

- The PWM generation using timer is one specific use of timer as an output compare device, where two parameters t_p and t_w can be configured; in case of PWM, the parameter t_w is varied while t_p is kept constant.
- Another possibility is to vary parameters t_p and t_w simultaneously, but keep the ratio $\frac{t_w}{tp}$ constant; by keeping this ratio constant, the resulting signal has a fixed duty cycle, and since t_p determines the periodicity and hence signal frequency, a variable frequency signal with constant duty cycle can be generated, where the frequency can be varied either gradually or abruptly.
- If parameter t_p is varied while t_w is kept constant, the resulting signal will have variable duty cycle as well as variable frequency, and an increase (decrease) in frequency will also result in a corresponding increase (decrease) in duty cycle.
- The joint variation of signal frequency and duty cycle can be decoupled by making both t_p and t_w variable, and the possible choices for parameters t_p , t_w and the signals generated as a result are tabulated, with the condition $t_w < t_p$ to be satisfied.

Table: Different signals generated based on parameters tp and tw.

Parameter t_p	t_w	$\frac{t_w}{t_p}$	Description
Fixed	Fixed	Fixed	Constant duty cycle as well as fixed frequency signal.
Fixed	Variable	Variable	Variable duty cycle but fixed frequency signal. This is PWM signal.
Variable	Variable	Fixed	Variable frequency but constant duty cycle signal.
Variable	Fixed	Variable	Both frequency as well as duty cycle are variable. These variations are coupled i.e. when frequency increases the duty cycle also increases and vice versa.
Variable	Variable	Variable	Both frequency as well as duty cycle are variable. These variations are decoupled i.e., frequency and duty cycle can be varied arbitrarily.