Q1. Answer the following. **[4+3+3 = 10 marks]**

a) Consider the following three instructions stored in consecutive memory locations in a byte addressable computer:

```
MOV   R1,R2          // R1 = R2
ADD   R3,LOCA        // R3 = R3 + Mem[LOCA]
SUB   R4,LOCB        // R4 = R4 - Mem[LOCB]
```

The first instruction is stored at memory location 2000 (in hexadecimal), the addresses LOCA and LOCB are 3004 (in hexadecimal) and 3100 (in hexadecimal) respectively. The contents of memory location LOCA and LOCB are 3E (in hexadecimal) and AE (in hexadecimal). Assume each instruction is of size 32 bits (i.e. 4 bytes). Answer the following, assuming that instruction fetch and decode are already over. **[1+1.5+1.5=4]**

i) What will be the content of program counter (PC) while the second instruction (**ADD R3,LOCA**) is getting executed?

ii) What will be the content of memory address register (MAR) while the second instruction (**ADD R3,LOCA**) is getting executed?

iii) What will be the content of memory data register (MDR) while the third instruction (**SUB R4,LOCB**) is getting executed?

(i) The value of the PC will be: 2008 (in hexadecimal) [1 mark]

(ii) The content of MAR will be: 3004 (in hexadecimal) [1.5 marks]

(iii) The content of MDR will be: AE (in hexadecimal) [1.5 marks]

b) Consider a 0-address machine where the operands are stored in the stack, with instructions like **ADD**, **SUB**, **MUL**, **DIV**, **PUSH X**, and **POP X**, where '**X**' refers to a memory location. Show the assembly language code to evaluate the following expression: **[3]**

```
A = (B * (C + D)) – (E * F)
```

The Postfix form of the given expression on the RHS is: B C D + * E F * -

The assembly language code is as follows:

```
PUSH  B
PUSH  C
PUSH  D
ADD
MUL
PUSH  E
PUSH  F
MUL
SUB
POP   A
```

[3 marks]
(Deduct 1 mark if POP A not mentioned)

c) Consider the following code segment for a 1-address accumulator-based machine. Assume that the variables **X**, **Y** and **Z** in memory are initialized with the values 50, 20 and 15 respectively (all in decimal). What will be the final values of **X**, **Y** and **Z**?

```
LOAD   X          ACC = 50
ADD    Y          ACC = 70
SUB    Z          ACC = 55
```

```
STORE   X       X = 55
LOAD    Y       ACC = 20
SUB     X       ACC = -35
STORE   Z       Z = -35
```

X = 55 **[1 mark]**

Y = 20 **[1 mark]**

Z = -35 **[1 mark]**

Q2. Answer the following. **[3+4+3 = 10 marks]**

a) In one sentence each, state how an operand stored in memory can be accessed using: (i) register indirect addressing, (ii) direct addressing, and (iii) indirect addressing.

[1 + 1 + 1 marks]

i) **Register indirect addressing: The specified register contains the address of a memory location; the operand is stored in that memory location.**

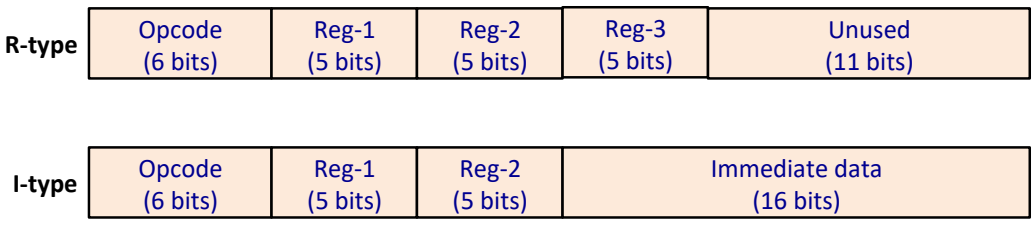
ii) **Direct addressing: The address of a memory location is stored as part of the instruction; the operand is stored in that memory location.**

iii) **Indirect addressing: The address of a memory location is stored as part of the instruction; the specified memory location contains the address of another memory location where the operand is stored.**

b) Consider a processor with the following specifications. The processor uses 32-bit (4-byte) instruction format. It has 32 registers (R0, R1, …, R31), each of 32-bits size. There are a total of 50 different instructions, which are categorized into two types – ***R-type*** and ***I-type***. In the R-type instruction, three register operands can be specified. In the I-type instruction, two register operands and an immediate data can be specified. Some example instructions are as follows:

```
ADD     R2,R5,R9        // R2 = R5 + R9     (R-type)
ADDI    R3,R5,#25       // R3 = R5 + 25     (I-type)
LOAD    R1,73(R4)       // R1 = Mem[R4+73]  (I-type)
```

Show a possible instruction format for the R-type and I-type instructions, clearly mentioning the size and purpose of the different fields in the instruction. **[4]**



[4 marks]

The 6-bit opcode specifies one of 50 different instructions.

Each 5-bit Reg field specifies a register. The R-type instruction requires three registers, while the I-type instruction requires two registers.

The 16-bit "Immediate data" field in the I-type instruction either specifies a 16-bit immediate operand, or a 16-bit offset.

c) In the design of the control unit for a processor, there are 118 control signals that can be divided into three groups **G1**, **G2** and **G3**. **G**1 contains 40 control signals that can be activated either one (or multiple) at a time. **G2** contains 55 control signals that are mutually exclusive (that is, at most one of them can be active at a given time). Similarly, **G3** contains 23 control signals that are also mutually exclusive. For a micro-programmed control unit, suggest a suitable microinstruction word encoding that requires the smallest number of bits without sacrificing concurrency. Show a schematic diagram showing how the control signals are decoded.

G1 must be encoded using 40 bits as the control signals may bet activated more than one at a time.

G2 can use vertical encoding as at most one can be active at a time --- requires $\log_2 55 = 6$ bits.

G3 can use vertical encoding as at most one can be active at a time --- requires $\log_2 23 = 5$ bits.

Therefore, size of the control word = 40 + 6 + 5 = 51 bits.

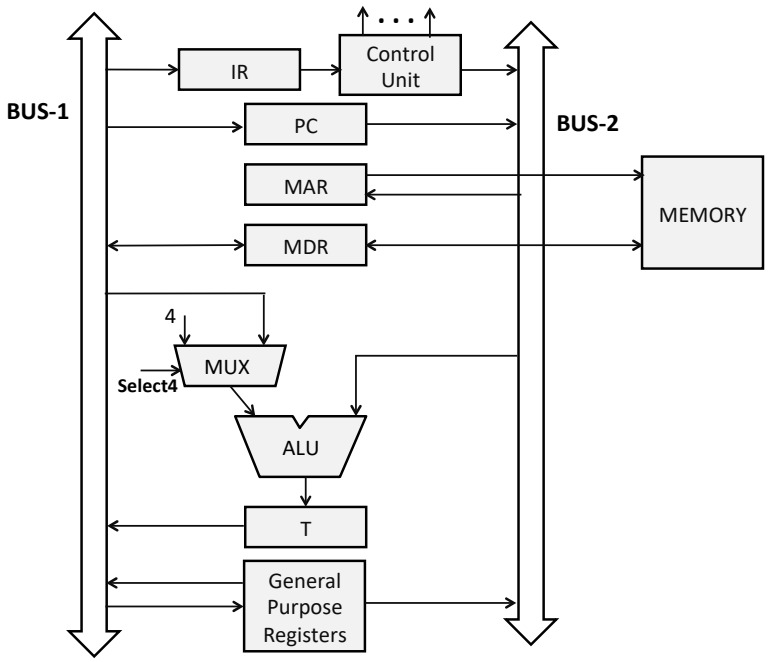
We need a (6 x 55) and a (5 x 23) decoder for the groups G2 and G3.

The schematic diagram must show the partition of the three groups, and the decoders.

(Give marks if the decoder sizes are mentioned as 6 x 64 and 5 x 32.

[2 marks for bit size calculation; 1 mark for showing the decoders]

Q3. Consider the two-bus processor architecture as shown in the diagram below. In addition to the standard registers IR, PC, MAR and MDR, there are eight general-purpose registers R0, R1, …, R7 in a register bank, and a temporary register T. All the registers are 32-bits in size. MAR and MDR are connected to the memory system as shown. Two registers can be simultaneously read from the register bank, one through BUS-1 and the other through BUS-2. Follow the usual convention for naming the control signals; in addition, Ri_{out}^{B1} and Ri_{out}^{B2} respectively denote the control signal to read the content of register Ri to the bus BUS-1 and BUS-2 respectively. **[4+3+3 = 10 marks]**



a) Show the control signals along with the time steps for fetching and executing the following instructions:

```
SUB     R1,R2       // R1 = R1 - R2
STORE   X,R3        // Mem[X] = R3
```

[4]

SUB R1,R2:

T1: PC_{out}, MAR_{in}, Select4, T_{in}, Read, ADD

T2: WMFC, T_{out}^{B1}, PC_{in}, MDR_{in}

T3: IR_{in}, MDR_{out}

T4: $R1_{out}^{B1}$, $R2_{out}^{B2}$, ~Select4, SUB, T_{in}

T5: T_{out}^{B1}, $R1_{in}$, END **[2 marks]**

STORE X,R3:

T1: PC_{out}, MAR_{in}, Select4, T_{in}, Read, ADD

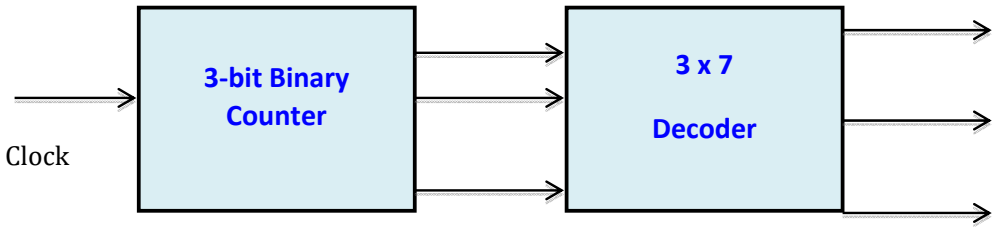
T2: WMFC, T_{out}^{B1}, PC_{in}, MDR_{in}

T3: IR_{in}, MDR_{out}

T4: IR_{out}^{addr}, MAR_{in}, Read

T5: WMFC, MDR_{out}, $R3_{in}$, END **[2 marks]**

b) Draw the schematic diagram to generate the time-step signals T1, T2, … using a step counter/decoder circuit, assuming that the maximum number of time steps in an instruction is 7. **[3]**



c) For hardwired control unit design, show the logic equations to generate the control signals **MAR_{in}** and **T_{in}** with respect to the two instructions given in part (a).

The MAR_{in} signal gets generated:

- **SUB instruction, T1 step**
- **STORE instruction, T1 step**
- **STORE instruction, T4 step**

The T_{in} signal gets generated:

- **SUB instruction, T1 step**
- **SUB instruction, T4 step**
- **STORE instruction, T1 step**

MAR_{in} = T1 + T4 . STORE **[1.5 marks]**

T_{in} = T1 + T4.SUB [1.5 marks]

Q.4 (a) Consider a scenario where a CPU has to run 5 processes with different arrival and burst times. Calculate the **average waiting time** for the given scenario with the following scheduling algorithms: **[2.5 +2.5 = 5]**

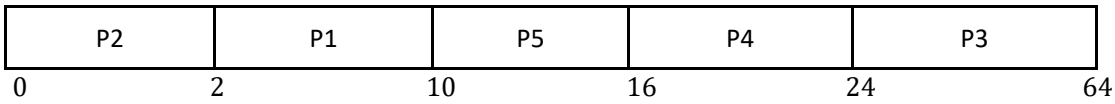
i) Non-preemptive Shortest Job First (SJF)

ii) Round Robin (RR) Scheduling (time quantum = 5ms)

Process	**P1**	**P2**	**P3**	**P4**	**P5**
Arrival Time	0	0	4ms	6ms	8ms
Burst Time	8ms	2ms	40ms	8ms	6ms

Solution: (i)

Gantt Chart:



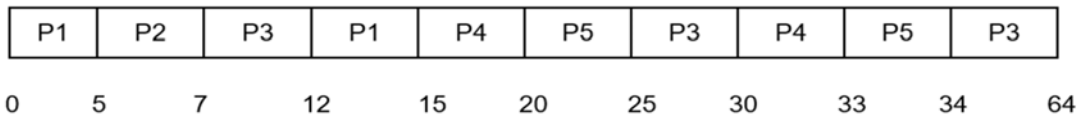
Waiting Times:

P1 = (2 - 0) = 2
P2 = (0 - 0) = 0
P3 = (24 - 4) = 20
P4 = (16 - 6) = 10
P5 = (10 - 8) = 2

Average waiting time = (2 + 0 + 20 + 10 + 2) / 5 = 34 / 5 = 6.8 msec

Solution: (ii)

Time Quantum = 5 msec



Waiting Times:

P1 = (12 - 5) = 7

P2 = (5 - 0) = 5

P3 = (7 - 4) + (25 - 12) + (34 - 30) = 20

P4 = (15 - 6) + (30 - 20) = 19

P5 = (20 - 8) + (33 - 25) = 20

Average waiting time = (7+5+20+19+20)/5 = 71/5 = 14.2 msec

Q4. (b) Consider three processes P1, P2, P3 with CPU burst times 2, 4, 8 units. All processes arrive at the time zero. Consider the preemptive longest remaining time first (LRTF) scheduling algorithm. In this algorithm, the process with the longest next remaining CPU time will be scheduled first. In LRTF, ties are broken by giving priority to the process with the lower process id. Compute the average turnaround time with the help of Gantt chart. Assume that the time units are integral. **[5] (4+1)**

Solution:

All processes arrive at time 0. Process with the longest remaining processing time is P3.

P3	P3	P3	P3	P2	P3	P2	P3	P1	P2	P3	P1	P2	P3	

TAT: Completion Time - Arrival Time
P1 = 12 - 0 = 12 (As all the jobs arrive at time 0)
P2 = 13 - 0 = 13
P3 = 14 - 0 = 14
Average TAT : (12+13+14)/3 = 13

Q4. (c) A process-management module uses a prioritized round-robin (RR) scheduling policy (that is, the ready queue is implemented as a priority queue). New processes are assigned an initial quantum of length "q". Whenever a process uses its entire quantum without blocking, its new quantum is set to twice its current quantum (for its next turn). If a process blocks before its quantum expires, its new quantum is reset to "q". You assume that every process requires a finite total amount of CPU time.

Is starvation possible for each of the following cases? The scheduler gives higher priority to (i) processes that have larger quanta, and (ii) processes that have smaller quanta. Justify our answers in both the cases.
[1+2+2=5]

Solution:

In a prioritized round-robin scheduling policy where the ready queue is implemented as a priority queue and new processes are assigned an initial quantum of length q, with the quantum doubling upon each full usage, starvation is not possible in either case. Here's the justification for each case:

1. **Higher Priority for Larger Quanta:**
   No, starvation is not possible in Round Robin (RR) Scheduling. Considering that a process will eventually terminate, the worst-case scenario is that a CPU-bound process will continue to execute until it completes its burst time or time quantum. Once it finishes execution, one of the lower-priority processes in the ready queue will be selected to execute next. This ensures fairness in scheduling as all processes, regardless of their priorities or characteristics, will eventually get a chance to execute due to the preemptive nature of Round Robin scheduling.
2. **Higher Priority for Smaller Quanta:**
   Yes, starvation is possible in this scenario. Considering a CPU-bound process that exhausts its time quantum while running on the processor. Now, if this process's time quantum is doubled, and there's a continuous flow of I/O-bound processes entering the system, they will always have a shorter time quantum compared to the process with the doubled quantum. As a result, these incoming I/O-bound processes will be selected for execution before the process with the extended time quantum, causing the latter to starve due to a lack of CPU time allocation.

Q.5 (a) Consider the following C code snippet?

…

```
for (int i=0; i<n; i++) {
      fork();
      printf("\nTest1");
      if (fork()==0){
            printf("\nTest2”);
      }
}
printf("\nTest3");
```

…

How many times “Test1”, “Test2” and “Test3” will be printed? Justify. **[5]**

Solutions:

Test1: $2/3\ (4^n - 1)$

Test2: $2/3\ (4^n - 1)$

Test3: 4^n

Q.5 (b) What are Singular, Orphan and Zombie processes? Give example scenarios. **[5]**

Singular Process: A singular process refers to a unique process that serves a specific purpose and typically runs in isolation, meaning there is only one instance of it running at any given time. Example Scenario: A singular process could be a system-wide daemon responsible for managing system resources or performing critical tasks such as scheduling, logging, or network management. An example could be the init process (PID 1) in Unix-like operating systems, which is the ancestor of all other processes and is responsible for starting and managing system services.

Orphan Process: An orphan process is a process whose parent process has terminated or completed, leaving the orphaned process running in the system's process table. If parent terminated without invoking wait , process is an orphan. Example Scenario: Consider a scenario where a user starts a process from a shell session (the parent process). If the user then closes the shell session or the parent process unexpectedly terminates, but the child process continues running, it becomes an orphan process. In Unix-like systems, orphan processes are typically adopted by the init process (PID 1), which ensures they are re-parented and managed until they complete.

Zombie Process: A zombie process, also known as a defunct process, is a process that has completed execution but still has an entry in the process table, as its parent process hasn't yet read its exit status. While zombie processes consume minimal system resources, they can clutter the process table if not properly handled. If no parent waiting (did not invoke wait()) process is a zombie Example ***Scenario***: Imagine a scenario where a parent process creates a child process to perform a specific task. After completing its task, the child process terminates, but the parent process fails to read its exit status using system calls like wait(). As a result, the child process becomes a zombie process. In such cases, the parent process should execute the wait() system call to reap the exit status of its terminated child processes and allow the operating system to remove their entries from the process table.

Q.5 (c) Consider the following C-program (Assume all function calls are successful).

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFSIZE 100

main(){
      int fd[2], i;
      char mybuf[BUFSIZE];

      pipe(fd);

      if (fork() > 0) {
            close(______fd[0]_________);
            for (i=0; i < 10; i++) {
                  printf("Parent Process..\n");
                  strcpy(mybuf, "Msg from Parent");
                  write(______fd[1]_________, mybuf, BUFSIZE);
                  printf("Parent writes: %s\n", mybuf);
                  sleep(1);
            }
      }
      else {
            close(__fd[1]_____________);
            for (i=0; i < 10; i++) {
                  printf("\t\t\t Child  Process..\n");
                  read(__________fd[0]______, mybuf, BUFSIZE);
                  printf("\t\t\t Child reads: %s\n", mybuf);
                  sleep(2);
            }
      }
}
```

Fill-up the blank spaces with appropriate expressions. Justify your answer. **[5]**