CMR INSTITUTE OF TECHNOLOGY

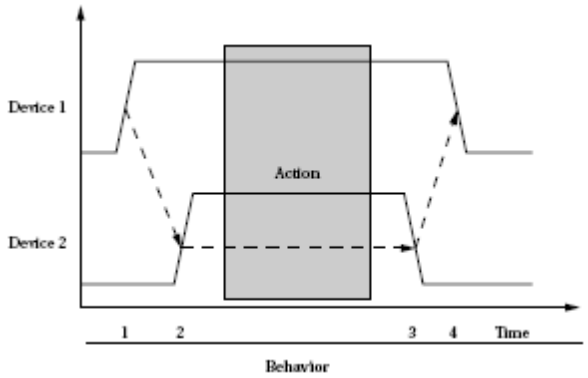
USN | | | | | | | | | | |

Internal Assesment Test – II Solution

Sub:	Embedded Computing Systems							Code:	10CS72
Date:	07/11/2017	Duration:	90 mins	Max Marks:	50	Sem:	VII	Branch:	CSE

Answer Any FIVE FULL Questions

		Marks	OBE CO	OBE RBT
1(a)	Explain the four-cycle handshake protocol.	[4]	CO2	L2



The basic building block of most bus protocols is the ***four-cycle handshake***, illustrated in the Figure. The handshake ensures that when two devices want to communicate, one is ready to transmit and the other is ready to receive. The handshake uses a pair of wires dedicated to the handshake: ***enq*** (meaning enquiry) and ***ack*** (meaning acknowledge). Extra wires are used for the data transmitted during the handshake. The four cycles are described below.

1. *Device 1* raises its output to signal an enquiry, which tells *device 2* that it should get ready to listen for data.

2. When *device 2* is ready to receive, it raises its output to signal an acknowledgment. At this point, *devices 1* and *2* can transmit or receive.

3. Once the data transfer is complete, *device 2* lowers its output, signaling that it has received the data.

4. After seeing that *ack* has been released, *device 1* lowers its output.

At the end of the handshake, both handshaking signals are low, just as they were at the start of the handshake. The system has thus returned to its original state in readiness for another handshake-enabled data transfer.

(b)	What is inter-process communication (IPC)? Explain the different IPC techniques.	[6]	CO2	L2

Inter Process Communication (IPC) is the mechanism provided by the OS as part of the process abstraction through which the processes/tasks communicate with each other.

Some of the important IPC mechanisms adopted by various kernels are explained below:

1. Shared Memory
   Processes share some area of the memory to communicate by the process is written to the shared memory area. Other processes which require this information can read the same from the shared memory area.

   Some of the different mechanisms adopted by different kernels are as below:
   a. Pipes: Pipe is a section of the shared memory used by processes for communicating. Pipes follow the client-server architecture. A process which creates a pipe is known as a pipe server and a process which connects to a pipe is known as pipe client. It can be unidirectional, allowing information flow in one direction or it can be bidirectional, allowing bi-directional information flow. Generally, there are two types of pipes supported by the OS. They are:
      Anonymous pipes: They are unnamed, unidirectional pipes used for data transfer between two processes.
      Names Pipes: They are named, unidirectional or bidirectional for data exchange between two processes.
   b. Memory Mapped Objects: This is a shared memory technique adopted by some real-time OS for allocating shared block of memory which can be accesses by multiple process simultaneously. In this approach, a mapping object is created and physical storage for it is reserved and committed. A process can map the entire committed physical area ir a block of it to its virtual address space. All read-write operations to this virtual address space by a process is directed to its committed physical area. Any process which wants to share data with other processes can map the physical memory area of the mapped object to its virtual memory space and use it for sharing the data.

2. Message Passing
   Message passing is an (a)synchronous information exchange mechanism used for Inter Process/Thread communication. The major difference between shared memory and message passing is that through shared memory lots of data can be shared whereas only limited amount of data is passed through message passing. Also, message passing is relatively fast and free from synchronization overheads compared to shared memory. Based on the message passing operation between the processes, message passing is classified into:
   a. Message Queue: Usually the process which wants to talk to another process posts the message to a First-In-First-Out (FIFO) queue called ‘message queue’, which stores the message temporarily in a system

defined memory object, to pass it to the desired process. Messages are sent and received through *send* and *receive* methods. The messages are exchanged through the message queue. It should be noted that the exact implementation is OS dependent. The messaging mechanism is classified into synchronous and asynchronous based on the behavior of the message posting thread. In asynchronous messaging, the message posting thread just posts the message to the queue and it will not wait for an acceptance (return) from the thread to which the message is posted. Whereas in synchronous messaging, the thread which the message is posts the message enters waiting state and waits for the message result from the thread to which the message is posted. The thread which invoked the send message becomes blocked and the scheduler will not pick it up for scheduling.

b. Mailbox: Mailbox is an alternative to 'message queues' used in certain RTOS for IPC, usually used for one way messaging. The thread which creates the mailbox is known as 'mailbox server' and the threads which subscribe to the mailbox are known as 'mailbox clients'. The mailbox server posts messages to the mailbox and notifies it to the clients which are subscribed to the mailbox. The clients read the message from the mailbox on receiving the notification. The process of creation, subscription, message reading and writing are achieved through OS kernel provided API calls.

c. Signaling: Signaling is a primitive way of communication between processes/threads. *Signals* are used for asynchronous notifications where one process/trhead fires a signal, indicating the occurrence of a scenario which the other process(es)/thread(s) is waiting. Signals are not queued and they do not carry any data.

3. Remote Procedure calls and Sockets

Remote Procedure Call (RPC) is the IPC mechanism used by a process to call a procedure of another process running on the same CPU or on a different CPU which is interconnected in a network. In object oriented language terminology RCP is also known as *Remote Method Invocation (RMI)*. RPC is mainly used for distributed applications like client-server applications. The CPU/process containing the procedure which needs to be invoked remotely is known as server. The CPU/process which initiates an RPC request is known as client.

Sockets are used for RPC communication. Socket is a logical endpoint in a two-way communication link between two applications running on a network. Sockets are of different types, namely, Internet Sockets (INET),

UNIX sockets, etc. The INET sockets works on internet communication protocols, such as TCP/IP and UDP. They are classified into stream sockets and datagram sockets.

Stream sockets are connection oriented, and they use TCP to establish a reliable connection.

Datagram sockets rely on UDP for communication. The UDP connection is unreliable when compared to TCP.

2(a) List the different program optimization techniques. Explain any one technique with an example. [4] CO1 L1

1. Expression simplification
2. Dead code elimination
3. Procedure inlining
4. Loop transformations
5. Register allocation
6. Scheduling
7. Instruction selection

Expression Simplification:

This is a useful area for machine-independent transformations. Laws of algebra are used to simplify expressions. For example, distributive law is applied to rewrite the following expression:

```
a*b+a*c;
```

Re-written as `a*(b+c);`

The new expression has only teo operations rather than three for the original form. This is certainly cheaper because it is both faster and smaller.

(b) Analyze the following ARM assembly code: [6] CO4 L4

```
LDR r0,a
LDR r1,b
ADD r2,r0,r1
STR r2,w
LDR r0,c
LDR r1,d
ADD r2,r0,r1
STR r2,x
LDR r1,c
ADD r0,r1,r2
STR r0,u
LDR r0,a
LDR r1,b
SUB r2,r1,r0
STR r2,v
```

Answer the following:

i. Write the sample C code fragment for the above ARM assembly code.
ii. Draw register lifetime graph for the C code in (i).
iii. Modify the C code from (i) using operator scheduling for register allocation.
iv. Draw lifetime graph for the modified C code in (iii).
v. Generate ARM code for the modified C code in (iii).
vi. Draw DFG for (i) and (iii).

vii. Write the sample C code fragment for the above ARM assembly code

i. w=a+b;
ii. x=c+d;
iii. u=c+x;
iv. v=b-a;

viii. Draw a lifetime graph that shows uses of register in register allocation from the above C statement.

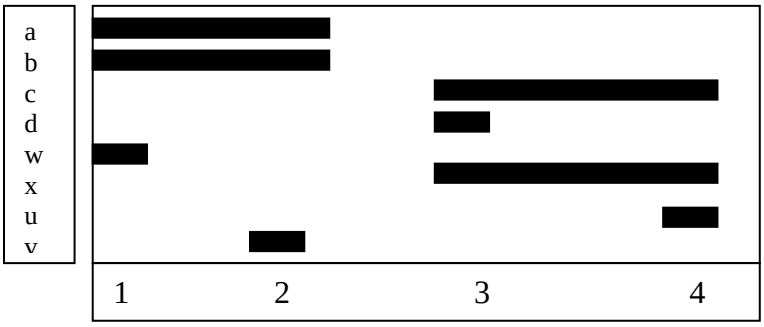
a
b
c
d
w
x
u
v

1 2 3 4

ix. Modify the obtained C code statement using operator scheduling for register allocation

i. w=a+b;
ii. v=b-a;
iii. x=c+d;
iv. u=c+x;

x. Draw a lifetime graph for the modified C code

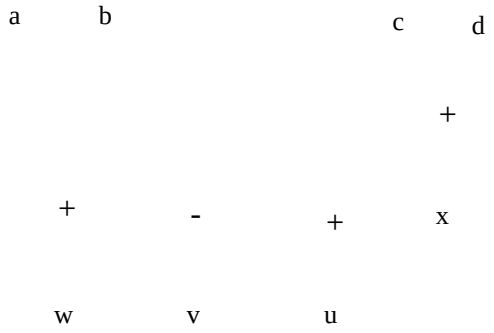


xi. Write a ARM assembly code for the modified C code using register allocation.

```
LDR r0,a
LDR r1,b
ADD r2,r0,r1
STR r2,w
SUB r2,r1,r0
STR r2,v
LDR r0,c
LDR r1,d
ADD r2,r0,r1
STR r2,x
LDR r1,c
ADD r0,r1,r2
STR r0,u
```

xii. Draw DFG for (i) and (iii)

DFG for (i) and (iii) are the same, as shown below.

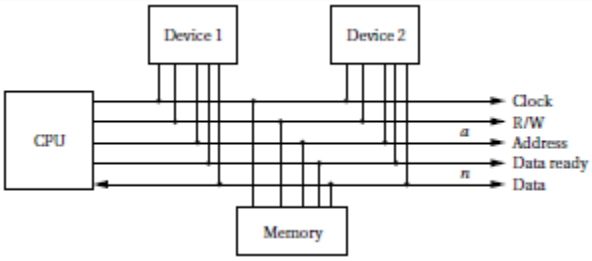


3(a) Explain the term *Bus Master*. Illustrate different components/signals on a bus. [4] CO2 L2

Bus master It is a device that can initiate its own bus transfer is known as a ***bus master***. Devices that do not have the capability to be ***bus masters*** do not need to connect to a bus request and bus grant.

The major components of a bus are as follows:

- ***Clock*** provides synchronization to the bus components,
- ***R/W*** is true when the bus is reading and false when the bus is writing,
- ***Address*** is an *a*-bit bundle of signals that transmits the address for an access,
- ***Data*** is an *n*-bit bundle of signals that can carry data to or from the CPU, and
- ***Data ready*** signals when the values on the data bundle are valid.



(b) Three processes with ID's P1, P2, P3 with estimated execution completion time 5, 10, 7ms respectively enters the ready queue together in the order P1, P2, P3. Process P4 with estimated execution completion time 2ms enters the ready queue after 5 ms. Which of the following scheduling policies is best for this scenario? Justify your choice. [6] CO3 L3

i. FIFO

P1 P2 P3 P4

0 5 6 15 16 22 23-24

Process	Waiting Time (WT) (ms)	Turn Around Time (TAT) (ms)
P1	0	5
P2	5	15
P3	15	22
P4	22-5 = 17	24-5=19
Average	**9.25**	**15.25**

ii. preemptive SJF

P1 P4 P3 P2

0 5 6 7 8 14 15 24

Process	Waiting Time (WT) (ms)	Turn Around Time (TAT) (ms)
P1	0	5
P2	14	24
P3	7	14
P4	5-5=0	7-5=2

Average	**5.25**	**11.25**

iii. RR (Time slice 2ms)

P1 P2 P3 P4 P1 P2 P3 P1 P2 P3 P2 P3 P2

0 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

Process	**Waiting Time (WT) (ms)**	**Turn Around Time (TAT) (ms)**
P1	10	15
P2	14	24
P3	15	22
P4	6-5=1	8-5=3
Average	**10.0**	**16.0**

From the above three scheduling policies, we see that preemptive SJF yields the least average waiting time (AWT) and Average Turn Around Time (ATAT). Thus, preemptive SJF is the best scheduling policy for the given scenario.

4(a) Precisely differentiate the following: [4] CO2 L2

i. Cooperating vs competing processes
ii. DFG vs CDFG programming models
iii. Deadlock vs livelock
iv. Anonymous pipes vs named pipes.

i. Cooperating vs Competing processes

Cooperating processes	**Competing processes**
In this model, one process requires the inputs from other processes to complete its execution	In this model, the processes do not share anything among themselves, but they compete for the system resources.
They can further be classified as cooperation though sharing, and cooperation though communication	Here the classification in based on the type of resource being shared like file, display device etc
E.g. Process B requiring data X from Process A to execute.	E.g. Process A needs printer P to print file f1, and Process B also needs Printer P to print file f2.

ii. DFG vs CDFG programming models

DFG	**CDFG**
Data Flow Graph is a model of program with no conditionals.	Control/Data Flow Graph model uses DFG as an element, adding constructs to describe control.
Precisely, only one entry point and one exit point.	May have multiple exit points depending on how a program is written.
Constitutes one basic block consisting of operators as nodes and variables as edges.	Usually contains more than one basic blocks. Constitutes two types of nodes, viz., decision nodes and data flow nodes.
For the given input data set, ALL statements are executed.	Based on the input data set, only selected paths may execute.

e.g. C fragment:
`x=a+b;`
`y=a-c;`
`z=x+d;`
	e.g. C fragment:
	`i=0;`
	`if (a<b)`
	`  i+=10;`
	`else`
	`  i-=10;`

iii. Deadlock vs Livelock

Deadlock	**Livelock**
Condition in which a process is waiting for resource held by another process, which in turn is waiting for a resource held by the first process.	Condition in which a process always does something but is unable to make any progress towards execution completion.
Situation where none of the processes are able to make any progress in their execution due to the cyclic dependency of resources. This ends up in none of the resources being utilized.	Situation where progress seem to happen all the time but actually no real execution. This is similar to the situation ‘always busy, doing nothing’.
e.g. Pa blocked by Pb for resource. Pb blocked by Pa for resource.	e.g. Both Pa and Pb needs x and y for completion. step 1: Pa holds x, Pb holds y step 2: Pa drops x, Pb drops y

	Repeat steps 1 and 2

iv. Anonymous pipes vs named pipes

Anonymous pipes	**named pipes**
Does not hold an explicit name. Identifiable only via handler.	Has an explicit name in addition to a handler
The pipe ceases to exist once the creator process is terminated	The pipe goes on to exist even after creator process terminates
Usually unidirectional	Bidirectional

(b) Explain the working of DMA with illustration. [6] CO2 L2

Direct memory access (DMA) is a bus operation that allows reads and writes not controlled by the CPU. A DMA transfer is controlled by a ***DMA controller***, which requests control of the bus from the CPU. After gaining control, the DMA controller performs read and write operations directly between devices and memory.

Figure below shows the configuration of a bus with a DMA controller. The DMA requires the CPU to provide two additional bus signals:

■ The ***bus request*** is an input to the CPU through which DMA controllers ask
for ownership of the bus.

■ The ***bus grant*** signals that the bus has been granted to the DMA controller.

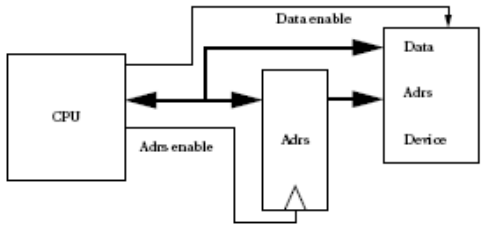


FIGURE 4.8
Bus signals for multiplexing address and data.

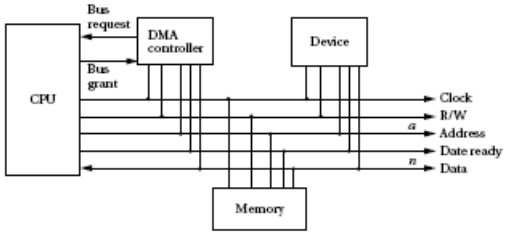


FIGURE 4.9
A bus with a DMA controller.

A device that can initiate its own bus transfer is known as a ***bus***

master. Devices that do not have the capability to be ***bus masters*** do not need to connect to a bus request and bus grant. The DMA controller uses these two signals to gain control of the bus using a classic four-cycle handshake. The bus request is asserted by the DMA controller when it wants to control the bus, and the bus grant is asserted by the CPU when the bus is ready.

The CPU will finish all pending bus transactions before granting control of the bus to the DMA controller. When it does grant control, it stops driving the other bus signals: R/W, address, and so on. Upon becoming bus master, the DMA controller has control of all bus signals (except, of course, for bus request and bus grant).
Once the DMA controller is bus master, it can perform reads and writes using the same bus protocol as with any CPU-driven bus transaction. Memory and devices do not know whether a read or write is performed by the CPU or by a DMA controller.

After the transaction is finished, the DMA controller returns the bus to the CPU by deasserting the bus request, causing the CPU to deassert the bus grant.

The CPU controls the DMA operation through registers in the DMA controller.

A typical DMA controller includes the following three registers:
■ A starting address register specifies where the transfer is to begin.
■ A length register specifies the number of words to be transferred.
■ A status register allows the DMA controller to be operated by the CPU.

The CPU initiates a DMA transfer by setting the starting address and length registers appropriately and then writing the status register to set its start transfer bit. After the DMA operation is complete, the DMA controller interrupts the CPU to tell it that the transfer is done.

What is the CPU doing during a DMA transfer? It cannot use the bus. As illustrated in Figure 4.10, if the CPU has enough instructions and data in the cache and registers, it may be able to

continue doing useful work for quite some time and may not notice the DMA transfer. But once the CPU needs the bus, it stalls until the DMA controller returns bus mastership to the CPU.

To prevent the CPU from idling for too long, most DMA controllers implement modes that occupy the bus for only a few cycles at a time. For example, the transfer may be made 4, 8, or 16 words at a time. As illustrated in Figure 4.11, after each block, the DMA controller returns control of the bus to the CPU and goes to sleep for a preset period, after which it requests the bus again for the next block transfer.

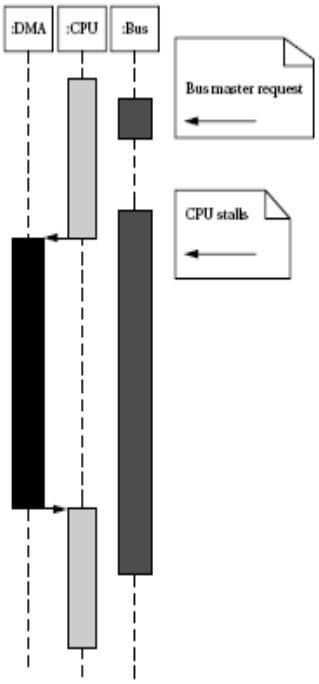


FIGURE 4.10
UML sequence diagram of system activity around a DMA transfer.

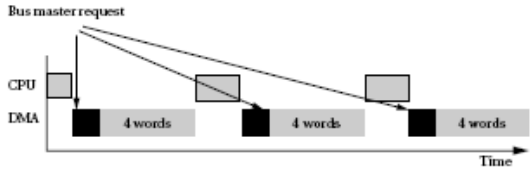


FIGURE 4.11
Cyclic scheduling of a DMA request.

5(a) What is RTOS? List the different services of RTOS, and explain any one in detail. [4] CO1 L1

RTOS stands for Real-Time Operating System, which is a type of operating system that implements policies and rules concerning time-critical allocation of system resources. RTOS decides which applications should run in which order, and how much time needs to be allocated for each application. E.g. Windows CE, QNX, VxWorks MicroC/OS-II.

Services of RTOS:

1. Real-time Kernel
    a. Task/Process management
    b. Task/Process scheduling

c. Task/Process synchronization
d. Error/Exception handling
e. Primary and Secondary Memory Management
f. File System Management
g. I/O system/ Device Management
h. Interrupt Handling
i. Time Management
j. Protection systems

2. Hard Real-Time
3. Soft-Real-Time

Hard/Soft -Real time:
RTOS must adhere to the timing constraints of the processes/ applications. Based on the type of deadline, RTOS can be classified as either hard real time or soft real time system.
RTOS that strictly adhere to the deadline associated to tasks without any slippage are referred to as hard real-time systems. Missing any deadline may produce catastrophic results, including permanent data loss, irrecoverable damages and/or safety concerns. Here, the principle is '*A late answer is wrong answer*'. E.g. Anti-lock Braking System (ABS), Airbag control in vehicles.
RTOS that does not strictly guarantee meeting deadlines, but offers best effort to meet the deadline are referred to as soft real-time systems. Missing deadlines for tasks are acceptable if the frequency of missing deadline is within the compliance limit of the Quality of Service (QoS). E.g. Automatic Teller Machine (ATM), Audio-Video playback systems.

(b) Analyze the following C fragment: [6] CO4 L4

```
if (a < b) {
  if (c < d)
    x = 1;
  else
    x = 2;
}
else{
  if (e < f)
    x = 3;
  else
    x = 4;
}
```

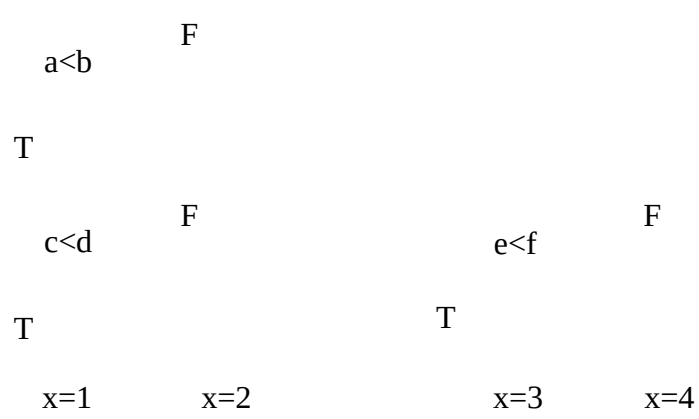
i. Generate ARM assembly code.
ii. Draw the CDFG.
ii. Find the cyclomatic complexity of the CDFGs.

v. Generate ARM assembly code.

```
ADR R0, a ;
LDR R1, [R0]; R1 <- a
ADR R0, b
```

```
LDR R2, [R0]; R2 <- b
ADR R0, c ;
LDR R3, [R0]; R3 <- c
ADR R0, d
LDR R4, [R0]; R4 <- d
CMP R1, R2
BGE outer_else
CMP R3, R4
BGE inner_else1
MOV R5, #1
JUMP after
inner_else1:MOV R5, #2
JUMP after
outer_else: ADR R0, e
LDR R3, [R0] ;  R3 <- e
ADR R0, f
LDR R4, [R0] ; R4 <- f
CMP R3, R4
BGE inner_else2
MOV R5, #3
JUMP after
inner_else2: MOV R5, #4
after: ADR R0, x
STR R5, [R0] ; R5 -> x
```

v. Draw the CDFG.



vi. Find the cyclomatic complexity of the CDFGs.

Cyclomatic Complexity M = e – n + 2p
From the above CDFG,
Number of edges, e = 10
Number of nodes, n= 8
Number of exit points, p = 1
Therefore, Cyclomatic Complexity M = 10-8+2=4

6(a)	For the following basic block given below, rewrite it in single-assignment form, and then generate the DFG.
	`r=a+b-c;`
	`s=2*r;`
	`t=b-d;`
	`r=d+e;`
		[4]	CO3	L3

Single-Assignment Form:

```
r1=a+b-c;
s=2*r1;
t=b-d;
r2=d+e;
```

a b c d e

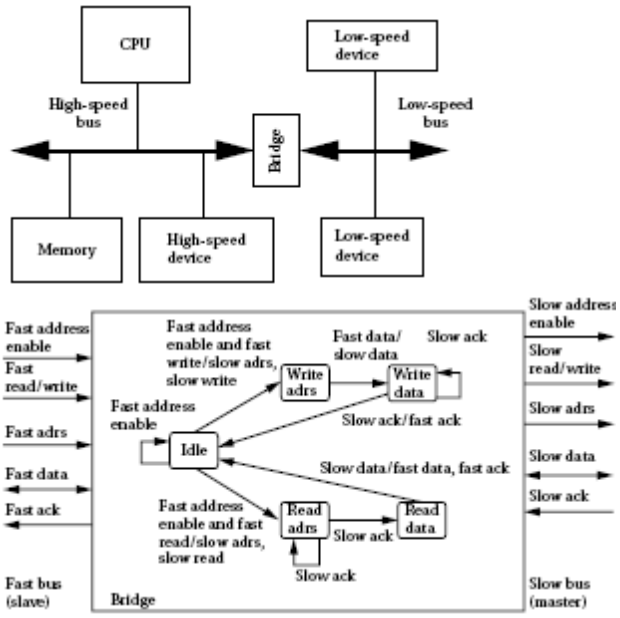
\+ - +

\- 2

r1 t r2

* s

(b)	Explain how bridge can be used to connect different speed systems.	[6]	CO2	L2



A microprocessor system often has more than one bus. As shown in the Figure, the high-speed devices may be connected to a high-performance bus, while lower-speed devices are connected to a different bus. A small block of logic known as a ***bridge*** allows the buses to connect to each other. There are several good reasons to use

multiple buses and bridges:
■ Higher-speed buses may provide wider data connections.
■ A high-speed bus usually requires more expensive circuits and connectors.

The cost of low-speed devices can be held down by using a lower-speed, lower-cost bus. The bridge may allow the buses to operate independently, thereby providing some parallelism in I/O operations.

Consider the operation of a bus bridge between a fast bus and a slow bus as illustrated in the Figure. The bridge is a slave on the fast bus and the master of the slow bus. The bridge takes commands from the fast bus on which it is a slave and issues those commands on the slow bus. It also returns the results from the slow bus to the fast bus—for example; it returns the results of a read on the slow bus to the fast bus. The upper sequence of states handles a write from the fast bus to the slow bus. These states must read the data from the fast bus and set up the handshake for the slow bus. Operations on the fast and slow sides of the bus bridge should be overlapped as much as possible to reduce the latency of bus-to-bus transfers.
Similarly, the bottom sequence of states reads from the slow bus and writes the data to the fast bus. The bridge serves as a protocol translator between the two bridges as well. If the bridges are very close in protocol operation and speed, a simple state machine may be enough. If there are larger differences in the protocol and timing between the two buses, the bridge may need to use registers to hold some data values temporarily.

7(a) Consider the following loop. [4] CO3 L3

```
int N=8, M=4;
for (i = 0; i < N*M; i++)
  x[i] = a[i] * c[i];
```

i. Optimize the code applying code motion technique.

ii. Optimize the code applying loop unrolling 2 times.

iii. Optimize the code applying code motion technique.

```
int N=8, M=4;
temp = N*M;
for (i = 0; i < temp; i++)
  x[i] = a[i] * c[i];
```

iv. Optimize the code applying loop unrolling 2 times.

```
int N=8, M=4;
temp = N*M;
for (i = 0; i < temp; i+=2)
            x[i] = a[i] * c[i]
```

```
x[i+1] = a[i+1] * c[i+1]
```

(b) What is task synchronization? Explain the different task synchronization techniques. [6] CO2 L2

The act of making processes aware of the access of shared resources by each process to avoid conflicts is known as Task synchronization. Task synchronization is essential for 1. Avoiding conflicts in resource access (racing, deadlock, livelock, starvation) in a multitasking environment, and 2. Ensuring proper sequence of operation across processes. E.g. producer-consumer problem.

Different task synchronization techniques to address them are as follows:

1. Mutual Exclusion through busy waiting/spin lock:
   Busy waiting is the simplest method for enforcing mutual exclusion. The busy waiting technique uses a lock variable for implementing mutual exclusion. Each process/thread checks this lock variable before entering the critical section. The lock is set to '1' by a process/thread if the process/thread is already in its critical section; otherwise the lock is set to '0'. The major challenge in implementing the lock variable based synchronization is the non-availability of a single atomic instruction which combines the reading, comparing and setting of the lock variable. To address this issue is tackled by combining the actions of reading the lock variable, testing its state and setting the lock into a single step, with a combined hardware and software support. Most processors support a single instruction 'Test and Set Lock' (TSL) for testing and software support. This instruction call copies the value of the lock variable and sets it to a nonzero value.
   The lock based mutual exclusion implementation always checks the state of a lock and waits till the lock is available. This keeps the processes/threads always busy and forces the processes/threads to wait or spin in one state till the availability of the lock for proceeding further. Hence, this synchronization is got the name 'Busy Waiting' or 'Spin Lock'. For the same reason, this mechanism leads to underutilization, wastage of processor time and power consumption.

2. Mutual Exclusion through Sleep and Wake up:
   An alternative to 'busy waiting' is the 'Sleep & Wakeup' mechanism. When a process is not allowed to access the critical section that has been locked by another process, the process undergoes 'Sleep' and enters 'Blocked' state. The process which is blocked on waiting for access to the critical section is awakened by the process which currently owns the critical section. Sleep & Wake can be implemented in different ways. Few of them are listed below.

Semaphores: It is a sleep and wake up based mutual exclusion for shared memory access, which limits the access of resources by a fixed number of processes/threads. This is further classified into two: *Binary semaphore* and *Counting Semaphore*. The binary semaphore, also called *mutex*, provides exclusive access to shared resource by allocating the resource to a single process at a time, and not allowing other process to access it when it is being owned by a process. Counting Semaphore, on the other hand, maintains a count between zero and a value. It limits the usage of a resource to the maximum value of the count supported by it.

Critical Section Objects: In Windows CE, the critical section object is same as the mutex object, except that Critical section object can only be used by the threads of a single process (Intra process). The piece of code which needs to be made critical section is places in the 'critical section' area by the process. The memory area which is to be used as the 'critical section' is allocated by the process. Once the critical section is initialized, all threads in the process can use it using an API call for getting exclusive ownership of the critical section.

Events: Event object is a synchronization technique which uses the notification mechanism for synchronization. In the concurrent execution we may come across situations which demand processes to wait for a particular sequence for its operations. For example, in producer-consumer threads, the consumer should wait to consume the data for producer to produce the data, and likewise, producer should wait for consumer to consume data. Event objects are helpful to implement notification mechanisms in such scenarios. A thread/process can wait for an event and another thread/process can set this event for processing by the waiting thread/process.

8(a) Precisely differentiate the following: [4] CO2 L2

i. PC vs PLC
ii. Counting semaphore vs mutex
iii. Non-preemptive scheduling vs preemptive scheduling
iv. Asynchronous vs Synchronous message passing

i. PC vs PLC

PC	**PLC**
Program counter: Points to the next instruction to be executed	Program Location Counter: Points to the next instruction to be parsed to generate addresses to each instruction.

Branches to the location where the branch target address points to	No effect of branch instruction, as it only keeps track of instruction address (no actual execution)
May pass over the same instruction more than once	Exactly passes each instruction once

ii. Counting semaphore vs Binary semaphore

Counting semaphore	**Binary semaphore**
It is a sleep and wake up based mutual exclusion for shared memory access, which limits the access of resources by a fixed number of processes/threads.	Aka Mutex is also a counting semaphore, but restricting the access to only one process/thread at any given time.
Maintains count between 0 and N, where N is the number of processes/threads that can access resource at a given time.	Uses 1-bit value tracking. That is counts 0 to 1.
e.g Network card supporting N ports for N parallel communication.	e.g. Printer uses a 1-bit lock stating whether it is in use or not. A process can use a printer only if it is not locked.

iii. Non-preemptive scheduling vs Preemptive Scheduling

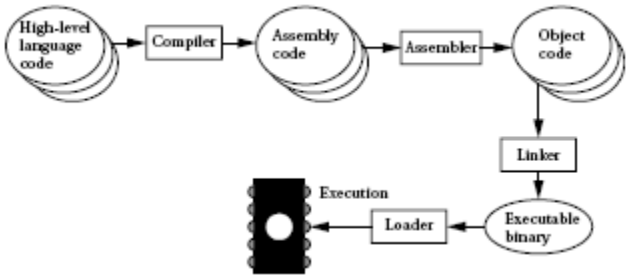
Non-preemptive scheduling	**Preemptive Scheduling**
In a multitasking model, this allows the currently executing task/process to run until it terminates or enters wait state, waiting for an IO or system resource.	In this multitasking model, every task in the Ready queue gets a chance to execute by evicting the currently running process. Here the scheduler temporarily pre-empts (stops) the currently running process.
E.g First Come First Served (FCFS), Last Come First served (LCFS), Shortest Job First (SJF), Priority based scheduling.	E.g. Preemptive SJF, Round Robin (RR).

iv. Non-blocking vs Blocking communication

Non-blocking	**Blocking communication**
Allows the process to continue execution after sending the	After sending communication, the process goes to waiting state until it

communication.	receives a response.
e.g. Synchronous RPC waiting for acknowledgement for every message sent.	e.g. Asynchronous RPC, where the calling process continues its execution while remote process executes the procedure.

(b) Explain the role of assemblers and linkers in the compilation process with a neat diagram. [6] CO2 L2



Assemblers

When translating assembly code into object code, the assembler must translate opcodes and format the bits in each instruction, and translate labels into addresses. In this section, we review the translation of assembly language into binary.

Labels make the assembly process more complex, but they are the most important abstraction provided by the assembler. Labels let the programmer (a human programmer or a compiler generating assembly code) avoid worrying about the locations of instructions and data. Label processing requires making two passes through the assembly source code as follows:

1. The first pass scans the code to determine the address of each label.

2. The second pass assembles the instructions using the label values computed in the first pass.

As shown in Figure, the name of each symbol and its address is stored in a ***symbol table*** that is built during the first pass. The symbol table is built by scanning from the first instruction to the last. (For the moment, we assume that we know the address of the first instruction in the program). During scanning, the current location in memory is kept in a ***program location counter (PLC)***. Despite the similarity in name to a program counter, the PLC is not used to execute the program, only to assign memory locations to labels. For example, the PLC always makes exactly one pass through the program, whereas the program counter makes many passes over code in a loop.

At the start of the first pass, the PLC is set to the program's starting address and the assembler looks at the first line. After examining the line, the assembler updates the PLC to the next location (since ARM instructions are four bytes long,

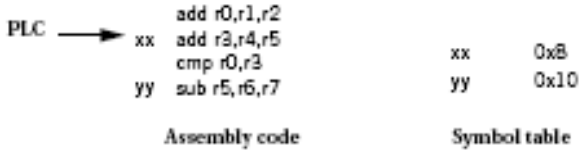
the PLC would be incremented by four) and looks at the next instruction. If the instruction begins with a label, a new entry is made in the symbol table, which includes the label name and its value. The value of the label is equal to the current value of the PLC.

At the end of the first pass, the assembler rewinds to the beginning of the assembly language file to make the second pass. During the second pass, when a label name is found, the label is looked up in the symbol table and its value substituted into the appropriate place in the instruction.

To know the starting value of the PLC, we assume the simplest case of absolute addressing. In this case, one of the first statements in the assembly language program is a pseudo-op that specifies the ***origin*** of the program, that is, the location of the first address in the program. A common name for this pseudo-op (e.g., the one used for the ARM) is the ORG statement

```
ORG 2000
```

which puts the start of the program at location 2000. This pseudo-op accomplishes this by setting the PLC's value to its argument's value, 2000 in this case. Assemblers generally allow a program to have many ORG statements in case instructions or data must be spread around various spots in memory.



Assemblers allow labels to be added to the symbol table without occupying space in the program memory. A typical name of this pseudo-op is EQU for equate. For example, in the code

```
ADD r0,r1,r2
FOO EQU 5
BAZ SUB r3, r4,#FOO
```

the EQU pseudo-op adds a label named FOO with the value 5 to the symbol table. The value of the BAZ label is the same as if the EQU pseudo-op were not present, since EQU does not advance the PLC. The new label is used in the subsequent SUB instruction as the name for a constant. EQUs can be used to define symbolic values to help make the assembly code more structured.

The ARM assembler supports one pseudo-op that is particular to the ARM instruction set. In other architectures, an address would be loaded into a register (e.g., for an indirect access) by reading it from a memory location. ARM does not have an instruction that can load an effective address, so the assembler supplies the ADR pseudo-op to create the address in the register. It does so by using ADD or SUB instructions to generate the address. The address to be loaded can be register relative, program relative, or numeric, but it must assemble to a single instruction. More complicated address calculations must be explicitly programmed.

The assembler produces an object file that describes the instructions and data in binary format. A commonly used object file format, originally developed for Unix but now used in other environments as well, is known as COFF (common

object file format).The object file must describe the instructions, data, and any addressing information and also usually carries along the symbol table for later use in debugging.
Generating relative code rather than absolute code introduces some new challenges to the assembly language process. Rather than using an ORG statement to provide the starting address, the assembly code uses a pseudo-op to indicate that the code is in fact relocatable. (Relative code is the default for the ARM assembler.)
Similarly, we must mark the output object file as being relative code. We can initialize the PLC to 0 to denote that addresses are relative to the start of the file. However, when we generate code that makes use of those labels, we must be careful, since we do not yet know the actual value that must be put into the bits. We must instead generate relocatable code. We use extra bits in the object file format to mark the relevant fields as relocatable and then insert the label's relative value into the field. The linker must therefore modify the generated code—when it finds a field marked as relative, it uses the addresses that it has generated to replace the relative value with a correct, value for the address.

Linking
Many assembly language programs are written as several smaller pieces rather than as a single large file. Breaking a large program into smaller files helps delineate program modularity. If the program uses library routines, those will already be preassembled, and assembly language source code for the libraries may not be available for purchase. A ***linker*** allows a program to be stitched together out of several smaller pieces. The linker operates on the object files created by the assembler and modifies the assembled code to make the necessary links between files.

Some labels will be both defined and used in the same file. Other labels will be defined in a single file but used elsewhere as illustrated in Figure. The place in the file where a label is defined is known as an ***entry point***. The place in the file where the label is used is called an ***external reference***. The main job of the loader is to *resolve* external references based on available entry points. As a result of the need to know how definitions and references connect, the assembler passes to the linker not only the object file but also the symbol table. Even if the entire symbol table is not kept for later debugging purposes, it must at least pass the entry points. External references are identified in the object code by their relative symbol identifiers.
The linker proceeds in two phases. First, it determines the address of the start of each object file. The order in which object files are to be loaded is given by the user, either by specifying parameters when the loader is run or by creating a ***load map*** file that gives the order in which files are to be placed in memory.

Given the order in which files are to be placed in memory and the length of each object file, it is easy to compute the starting address of each file. At the start of the second phase, the loader merges all symbol tables from the object files into a single, large table. It then edits the object files to change relative addresses into addresses.

This is typically performed by having the assembler write extra bits into the

object file to identify the instructions and fields that refer to labels. If a label cannot be found in the merged symbol table, it is undefined and an error message is sent to the user.

Controlling where code modules are loaded into memory is important in embedded systems. Some data structures and instructions, such as those used to manage interrupts, must be put at precise memory locations for them to work. In other cases, different types of memory may be installed at different address ranges. For example, if we have EPROM in some locations and DRAM in others, we want to make sure that locations to be written are put in the DRAM locations.

Workstations and PCs provide ***dynamically linked libraries***, and some embedded computing environments may provide them as well. Rather than link a separate copy of commonly used routines such as I/O to every executable program on the system, dynamically linked libraries allow them to be linked in at the start of program execution. A brief linking process is run just before execution of the program begins; the dynamic linker uses code libraries to link in the required routines. This not only saves storage space but also allows programs that use those libraries to be easily updated. However, it does introduce a delay before the program starts executing.

```
label1   LDR r0,[r1]
         ...
         ADR a
         ...
         B label2
         ...
var1     % 1
```

External references	Entry points
a	label1
label2	var1

File 1

```
label2   ADR var1
         ...
         B label3
         ...
x        % 1
y        % 1
a        % 10
```

External references	Entry points
var1	label2
label3	x
	y
	a

File 2

GOOD LUCK!