EEC011P5G Microcomputers and Embedded Systems
Introduction

1. State which of the following statements are true / false with suitable justification.

    1) An airplane is a distributed embedded systems.
    2) A lift controller is a centralized embedded system.
    3) A washing machine controller is a distributed embedded system.
    4) A new type of pacemaker is used that communicates with a body area network. This is a centralized embedded system.
    5) A pacemaker is a soft real-time system.
    6) A traffic light controller is a hard real-time system.
    7) A device driver is a hard real-time system.
    8) A neonatal resuscitator is a hard real-time system.
    9) A critical section appears in only one thread of a concurrent program.
    10) Mutual exclusion when ensured will ensure deadlock freedom.

2. A mealy FSM is defined as a 6-tuple M =< $Q, q_0, \Sigma, \Delta, \delta, \lambda$>. Consider the FSM shown in Figure 1. For this FSM, identify the 6 components. Also, define all the different components of M.

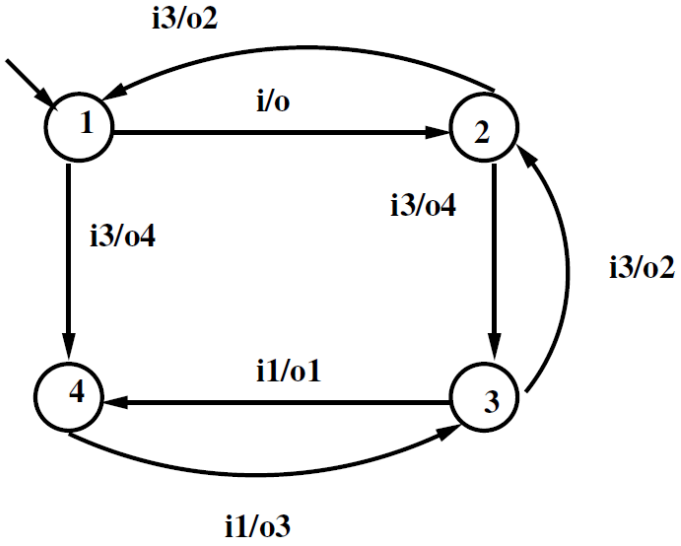

Figure 1 An example of a Mealy machine (FSM)

3. Distinguish between reactive systems and real-time systems.

4. What is the problem of priority inversion and relate it to the Mars rover failure? You are encouraged to read about this on the Internet and answer this question.

5. Assume that `occupied` is a global variable, which has been initialised to `false` and `flag` is a variable that is local to individual processes. Consider the TestAndSet (TS) instruction implemented using the following approaches. Which approach is correct? Also, can TS be implemented as a C-function? Justify your answers to both these questions.

```
Approach-1: flag:=true; occupied:=flag;
Approach-2: flag:=false; occupied:=flag;
Approach-3: flag:=occupied; occupied:=true;
Approach-4: occupied:=flag; flag:=true;
```

Consider the Test-and-Set (TS) instruction again. Let the initial value of a global variable called `occupied` be `true`. Let `code` be local to each process. Consider a process executing the following code segment. Determine the value of `occupied` and `code` when P( ) finishes. Show your working.

```
Process P( ){
TS(occupied);
if(code)
occupied=false;
else
occupied = true;
TS(occupied);
TS(occupied);
}
```

6. Consider a simple two process critical section implementation without using any mutex locks. Here, the solution uses conventional busy waiting using two variables called `need`[1] and `need`[2]. A process request entry to critical section by setting its need (`need`[1] is for Process-1 and `need`[2] for Process-2). A variable called `turn` is used to assign priority to a given process (if `turn=1` then Process-1 has higher priority). Consider the solution to critical section problem only using busy waiting as follows:

```
need[1]=false; need[2]=false;
RequestEntry:: need[me] = true;
               while need[other]{
                  if turn != me{
                     need[me]=false;
                  }
               }
               --critical section--

Release:: need[me]=false;
          turn=other;
```

Does this solution satisfy the requirements such as {*mutual exclusion, lack of deadlock and lack of indefinite postponement*}? Justify.

HINT: an execution sequence where mutual exclusion is preserved may be as follows - Let `turn` be 2. Process-1 may be scheduled first setting `need`[1] to `true`. Then process-2 is scheduled. process-2 sets `need`[2] to `true`. Then process-1 is scheduled again so that it enters the *entry protocol* and since `need`[2] is `true`, it enters inside the while and sets `need`[1] to `false` (as `turn`=2). Now process-2 is scheduled again, and it can enter the CS while process 1 is locked (busy waiting) in the while loop.

7. Consider the following three processes, HOG, P[0] and P[1]. These three processes are coded in pseudo-code:

```
Hog::
   while(1);

P[0]::
   int j=0;
```

```
   char a='A';
   while(j < 10){
       printf(``%c'', a);
       ++j;
   }

P[1]::
 int j=0;
   char b='B';
   while(j < 10){
       printf(``%c'', b);
       ++j;
   }
```

Now consider the following process `OutPut` which is running under a specific scheduler. `Cobegin..coend` allows the creation of three concurrent processes in the OS which are schedulable.

```
OutPut::
      cobegin
          Hog || P[0] || P[1]
      coend
```

Describe the output (if any) and whether or not the program will terminate for this concurrent program. Scheduling is extremely unfair, i.e., it lets the process that appears first (in a queue) to run to completion before selecting the next.

8. Consider the precedence graph of 4 concurrent processes shown in Figure 2. Use a ***mutex-locking*** based solution to enforce this precedence relation between processes. Declare the required number of mutex variables. In the main program, invoke the processes concurrently. Your solution must be developed in such a manner that correctness of the program is invariant to the scheduling order. The sample code for a process may be as follows:

```
Process P( ){
   wait for all predecessor processes
   execute the body of the process
   signal all successors
}
```

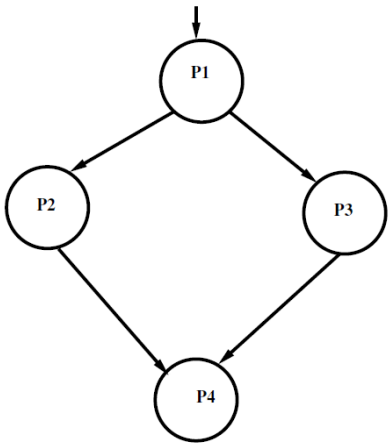

Figure 2 A precedence graph