EE 332 Real Time Systems
Midterm Examination Solution
Friday February 13, 2004
2:30 pm to 4:30 pm

Student Name	
Student Number	

Question	Mark
#1	/ 15
#2	/ 20
#3	/ 25
TOTAL	/ 60

General:

- Two hours (2:30 pm to 4:30 pm)
- Open book and open notes

1. 15 Marks (3 marks each)

a) Why is it important for a real-time system to have a watchdog timer? Is a timer interrupt routine an appropriate point in the software to reset the watchdog timer, why or why not?
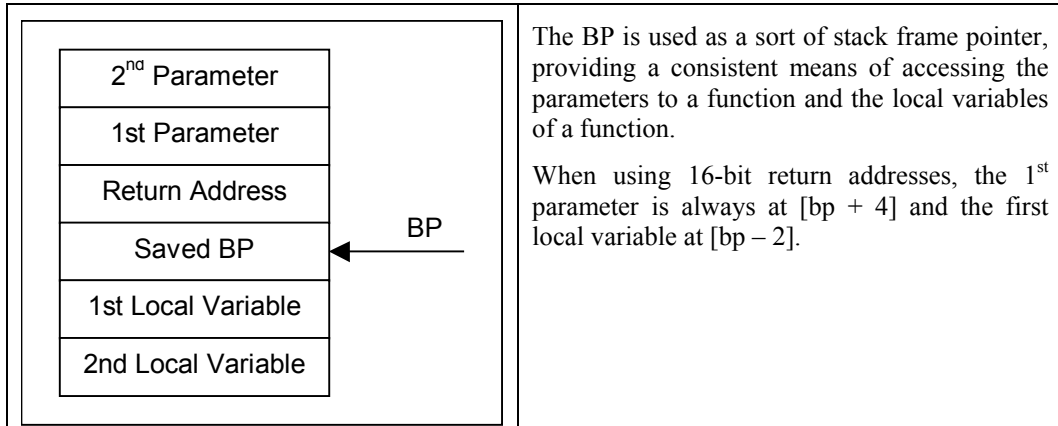
A watchdog timer guards against system malfunctions such as deadlocks (infinite loops), power glitches, time-overload by resetting the system, especially important for unattended/remote systems.

A timer interrupt routine is NOT an appropriate point in the software to reset the watchdog timer as all it may indicate that interrupts are still working, it does not guard against failures such as infinite loops in the foreground processes.

b) Show the Moore Finite State Automata to accept the words "cab", "cob", "cat", "cot" but no others from the alphabet L={a, b, c, t, o }.
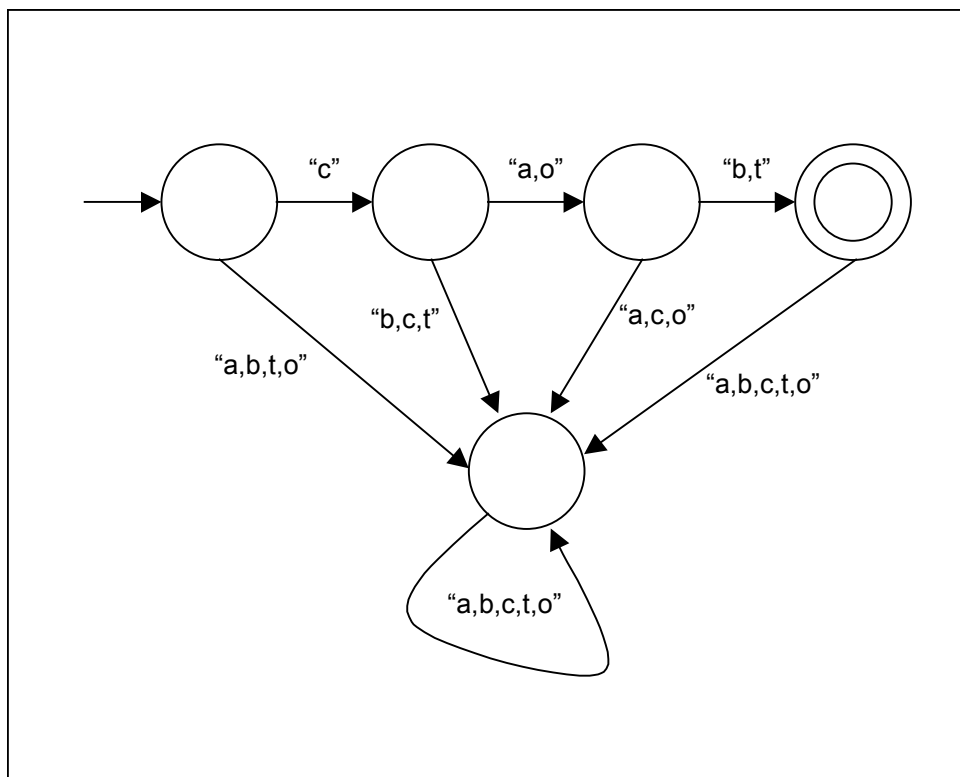
c) **Describe how the BP register is used to access the parameters and local variables for C functions on an 8086 microprocessor (e.g., what is the address of the first parameter and what is the address of the first local variable).**

2nd Parameter	The BP is used as a sort of stack frame pointer, providing a consistent means of accessing the parameters to a function and the local variables of a function.
1st Parameter	
Return Address	
Saved BP ← BP	When using 16-bit return addresses, the 1st parameter is always at [bp + 4] and the first local variable at [bp – 2].
1st Local Variable	
2nd Local Variable	

d) **Describe the difference between recursion and reentrancy.**

Recursion is the ability of a routine to call itself directly or indirectly (via other intermediate routines).

Reentrancy is the ability of a routine to be used concurrently by more than one task.

By inference, recursive routines are also reentrant

e) **Give two advantages and two disadvantages of Polled Loop systems.**

Advantages:

- Simple to write and debug
- The response time is easy to determine
- Excellent for handling high-speed data channels

Disadvantages:

- Not generally sufficient to handle complex systems
- Inherently waste CPU time
- May be too slow response time for certain parts of the system

2. 20 Marks (10 + 10)

Consider a hypothetical microprocessor called the CPU332v2. The CPU332v2 has the following properties:

- **One 16-bit accumulator ACC**
- **Three 16-bit general purpose registers R0, R1, R2**
- **A 16-bit stack pointer SP**
- **A "push" instruction can push any register (ACC, R0-R2) onto the stack.**
- **A "pop" instruction can pop any register (ACC, R0-R2) from the stack.**
- **Code addresses are 32-bits in size, saved in a "little-endian" format (low order byte to high order byte).**

a) **Show the "pseudo assembler" for a "Yield( )" function for the CPU332v2. The pseudo assembler should show the order of pushes, pops, and saving and restoring of stack pointers. Indicate which registers are pushed and in what order. The code to determine the next task to run can be glossed over (but you should still indicate where in your Yield( ) function this occurs).**

```
Yield:
    Push ACC
    Push R0
    Push R1
    Push R2
    Move SP to StackPointers[ CurrentTask ]
    <Determine next task to run, set new value for CurrentTask >
    Move StackPointers[ CurrentTask ] to SP
    Pop R2
    Pop R1
    Pop R0
    Pop ACC
    Return
```

b) **The interface of created tasks is required to be:**

```
void Task( short sparm, long lparm );
```
sparm is a 16-bit integer, lparm is a 32-bit integer (assume also little-endian)

For your Yield( ) function from (a), show the organization, addresses and values of the initial stack for a Task whose entry address is 0x11223344 and whose stack space goes from 0x2800 to 0x28FF (inclusive). Remember to show required initial values for any values on the stack (or XXXX for don't cares). Also indicate the value of the "initial stack pointer" (suitable to start the task via a call to Yield( )). Show the stack for two potential different modes of operation of the CPU332v2:

i. **Stack grows from low addresses to high addresses. For a push, the stack pointer is first incremented and then the value is written to the stack:**

Address	Value	Description
0x2814	XXXX	R2
0x2812	XXXX	R1
0x2810	XXXX	R0
0x280E	XXXX	ACC
0x280C	0x1122	Task Address (Most Significant Word)
0x280A	0x3344	Task Address (Least Significant Word)
0x2808	XXXX	Dummy or Task Terminate (MSW)
0x2806	XXXX	Dummy or Task Terminate (LSW)
0x2804	Value of sparm	sparm
0x2802	Value of lparm	lparm (MSW)
0x2800	Value of lparm	lparm (LSW)

| Initial SP = | 0x2814 | |

ii. **Stack grows from high addresses to low addresses. For a push, the value is written to the stack and then the stack pointer is decremented:**

Address	Value	Description
0x28FE	Value of lparm	lparm (Most Significant Word)
0x28FC	Value of lparm	lparm (Least Significant Word)
0x28FA	Value of sparm	sparm
0x28F8	XXXX	Dummy or Task Terminate (MSW)
0x28F6	XXXX	Dummy or Task Terminate (LSW)
0x28F4	0x1122	Task Address (Most Significant Word)
0x28F2	0x3344	Task Address (Least Significant Word)
0x28F0	XXXX	ACC
0x28EE	XXXX	R0
0x28EC	XXXX	R1
0x28EA	XXXX	R2

| Initial SP = | 0x28E8 | |

3. 25 Marks (10 + 15)

Consider the following sequence 1, 1, 1, 3, 5, 9, 17, ... Except for the first three numbers, each number is the sum of the preceding three numbers. A recursive routine for determining a particular number in the sequence is given by:

```
int f3( int num )
{
   int  result;

   if( num < 3 )
   {
      result = 1;
   }
   else
   {
      result =  f3( num - 3 ) + f3( num - 2 ) + f3( num - 1 );
   }

   return result;
}
```

a) For a call of "f3(5)", show how the recursive algorithm works by showing the values of the parameters and the order of each call to f3( ), how many times f3( ) is called, and the return value of each f3( ).

```
f3( 5 )
    f3( 2 )
    return 1
    f3( 3 )
        f3(  0 )
        return 1
        f3( 1 )
        return 1
        f3( 2 )
        return 1
    return 3
    f3( 4 )
        f3( 1 )
        return 1
        f3( 2 )
        return 1
        f3( 3 )
            f3(  0 )
            return 1
            f3( 1 )
            return 1
            f3( 2 )
            return 1
        return 3
    return 5
return 9
```

Total of 13 calls to f3( ).

b) **Given the following 8086 assembler code generated from f3( ) C code, fill in the table on the following page showing the maximum length call stack when you call "f3(5)". Indicate the stack address, element value (if the value is not determinable, indicate with "XXXX"), description of what the element represents, and the function the code was in when it pushed the value. Start with the parameter and return address of the call to "f3(5)" and assume the stack pointer has the value 0xFFF6 before the initial "5" parameter is pushed.**

```
    ; int f3( int num )
    ; {
_f3
        push    bp
        mov     bp,sp

    ;   int     result;
        sub     sp,2

    ;   if( num < 3 )
        cmp     word ptr [bp+4],3
        jge     short @2@86

    ;   {
    ;       result = 1;
        mov     word ptr [bp-2],1

    ;   }
        jmp     short @2@114

@2@86:
    ;   else
    ;   {
    ;       result =  f3( num - 3 ) + f3( num - 2 ) + f3( num - 1 );
        mov     ax,word ptr [bp+4]
        add     ax,-3
        push    ax
        call    near ptr _f3
        pop     cx

        push    ax

        mov     ax,word ptr [bp+4]
        add     ax,-2
        push    ax
        call    near ptr _f3
        pop     cx

        pop     dx
        add     dx,ax
        push    dx

        mov     ax,word ptr [bp+4]
        dec     ax
        push    ax
        call    near ptr _f3
        pop     cx

        pop     dx
        add     dx,ax
        mov     word ptr [bp-2],dx
    ;   }

@2@114:
    ;   return result;
        mov     ax,word ptr [bp-2]

    ; }
        mov     sp,bp
        pop     bp
        ret
```

Student Name: _____

Address	Value	Description	Function
0xFFF4	0x0005	Parameter to f3(5)	main( )
0xFFF2	XXXX	Return address to function calling f3(5)	
0xFFF0	XXXX	Saved bp	f3(5)
0xFFEE	XXXX	Local variable result	
0xFFEC	0x0004	Result of f3(2) + f3(3)	
0xFFEA	0x0004	Parameter to f3(4)	
0xFFE8	XXXX	Return address to f3( )	
0xFFE6	0xFFF0	Saved bp	f3(4)
0xFFE4	XXXX	Local variable result	
0xFFE2	0x0002	Result of f3(1) + f3(2)	
0xFFE0	0x0003	Parameter to f3(3)	
0xFFDE	XXXX	Return address to f3( )	
0xFFDC	0xFFE6	Saved bp	f3(3)
0xFFDA	XXXX	Local variable result	
0xFFD8	0x0001	Result of f3(0)	
0xFFD6	0x0001	Parameter to f3(1)	
0xFFD4	XXXX	Return address to f3( )	
0xFFD2	0xFFDC	Saved bp	f3(1)
0xFFD0	XXXX	Local variable result	
		OR	
0xFFD8	0x0002	Result of f3(0) + f3(1)	
0xFFD6	0x0002	Parameter to f3(2)	
0xFFD4	XXXX	Return address to f3( )	
0xFFD2	0xFFDC	Saved bp	f3(2)
0xFFD0	XXXX	Local variable result	