Name _____ **ECE mailbox** _____

Problem	Max pts	Score
1	17	
2	23	
3	20	
4	20	
5	20	
Total	100	

This exam is <u>open book</u>, <u>open notes</u>. Calculators are allowed. Internet access is not permitted. Show all work for full credit. Partial credit is also assigned.

1) [17 pts] Answer the following questions about real-time I/O.
   a) [10 pts] A typical hardware FIFO for an input peripheral interrupts when the FIFO is **half-full**. Why is this better than interrupting when **full** or **non-empty**?

If the FIFO interrupts when <u>full</u>, the <u>relative deadline</u> (until FIFO overflow) is <u>much shorter</u> (ISR can tolerate very little latency) than interrupting when half-full or non-empty.

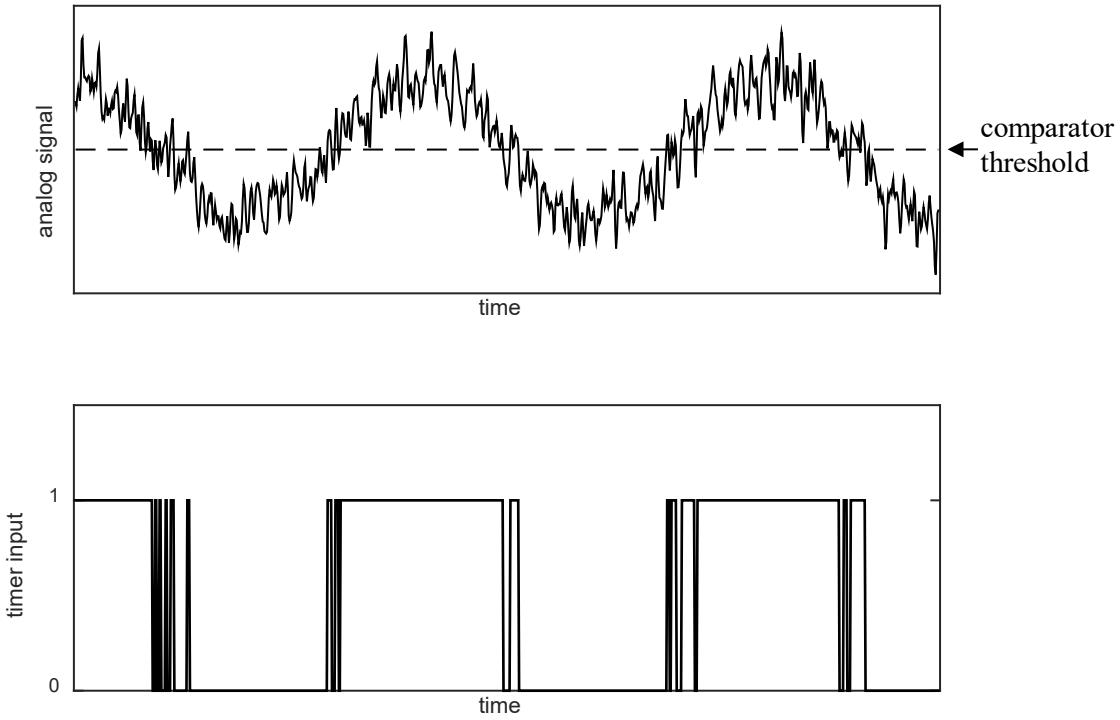
If the FIFO interrupts when <u>non-empty</u>, the <u>CPU load is much higher</u> (due to much shorter period) than interrupting when half-full or full.

Interrupting when half-full is a compromise between CPU load and the relative deadline (latency tolerance) for the ISR/task handling the FIFO.

   b) [7 pts] Why is the data payload size of a CAN (Controller Area Network) data frame limited to 8 bytes max? This is certainly not beneficial for data throughput.

The size of the payload affects the duration of the data frame on the CAN bus. CAN supports strict prioritization of messages, for RMS-like scheduling to guarantee all messages are delivered on time. Since a data frame cannot be preempted, the duration of the longest data frame determines the <u>latency</u> for the highest priority message transmitted through CAN.

2) [23 pts] Your embedded system is measuring the period of an analog signal that behaves as in the following figure. The signal is passed through an analog comparator, then fed to the input of a timer in capture mode.



a) [10 pts] Your period measurements often come out much smaller than the actual analog signal period. What do you think is causing this problem? On the "timer input" axes above, **fill in** what you think the timer input signal approximately looks like.

The noise in the input signal is causing multiple transitions in the timer input signal for each threshold crossing of the analog signal. These multiple transitions are being interpreted as very short waveform periods. Period is typically measured between two rising edges of the timer input signal.

b) [8 pts] What adverse effect could the issue in part (a) have on the schedule of the real-time system? Explain.

The edges of the timer input signal are triggering interrupts. There can be many such edges in a short time interval during the threshold crossing of the analog signal, so many interrupts. The timer capture ISR can temporarily starve lower priority real-time tasks of the CPU, causing them to miss deadlines.

c) [5 pts] What can be done to eliminate this problem? Hint: The solution would likely require hardware changes.

Any one of the following is sufficient for full credit:

- **Low-pass filter the analog signal to reduce high-frequency noise.**
- **Increase the hysteresis window size of the analog comparator.**
- **Add GPIO input conditioning that cleans up edges, similar to hardware debouncing, to the timer input**.

3) [20 pts] Convert the following C code fragment to ARM assembly. Register assignments are indicated in the comments. You may use other general purpose registers as temporary storage. Notes: You do not need to write any assembly code for the variable declarations. The arrays are stored in memory. This is not a complete function. You do not need to worry about the function call convention.

```
int32_t A[40];   // r0 = &A[0];
int32_t i, x, y; // r1 = i; r2 = x; r3 = y;

i = 0;
while (i < 40) {
    x = A[i] & 0xff;
    if (x == y) {
        break; // exit the loop
    }
    A[i] = x;
    i++;
}
```

```
        mov r1, #0        ; i = 0;

loop1   cmp r1, #40
        bge done1         ; if i >= 40, exit loop

        lsl r4, r1, #2    ; r4 = i * 4; // array offset
        ldr r2, [r0, r4]  ; x = copy of A[i];
        and r2, r2, #0xff ; x = x & 0xff;

        cmp r2, r3
        beq done1         ; if x == y, exit loop

        str r2, [r0, r4]  ; A[i] = x;
        add r1, r1, #1    ; i++;

        b   loop1         ; continue loop
done1
```

4) [20 pts] Convert the C function `f2()` to ARM assembly. Assume the function `f1()` is implemented elsewhere. Make sure to follow the C calling conventions for passing arguments and return values, as well as preserving registers that need to be preserved. Remember that certain registers can be overwritten by a function call.

```
int32_t f1(int32_t x, int32_t y);

int32_t f2(int32_t a)
{
    int32_t b = f1(a, 5);
    return b + f1(a, b - 5);
}
```

```
; upon entry into the function:
;   r0 = a;
; inside the function:
;   r4 = a;
;   r5 = b;
f2      push {r4, r5, lr} ; preserve registers on the stack

        mov  r4, r0       ; r4 = a;

                          ; 1st argument is already a
        mov  r1, #5       ; 2nd argument = 5;
        bl   f1           ; call f1();
        mov  r5, r0       ; b = return value of f1();

        mov  r0, r4       ; 1st argument = a;
        sub  r1, r5, #5   ; 2nd argument = b - 5;
        bl   f1           ; call f1();

        add  r0, r5, r0   ; r0 = b + (return value of f1());

        pop  {r4, r5, pc} ; restore registers and return (value in r0)
```

5) [20 pts] Determine the number of clock cycles required to execute the following ARM assembly function on a Cortex-M4 system. Show your work. You need to analyze the control flow in this function.

```
; void initab(int32_t a[], int32_t b[], int32_t c);

initab  mov r3, #0
; Data operation: 1 cycle

loop1   lsl r12, r3, #2
; Data operation: 1 cycle

        str r2, [r0, r12]
; STR with register offset,
; not pipelined: 2 cycles
        str r2, [r1, r12]
; STR with register offset,
; not pipelined: 2 cycles
        add r3, r3, #1
; Data operation: 1 cycle

        cmp r3, #3
; Data operation: 1 cycle

        ble loop1
; Branch to label
; Taken: 2 cycles (when r3 <= 3)
; Not taken: 1 cycle (when r3 > 3)
        bx  lr
; Branch to register,
; always taken: 3 cycles
```

Loop executes 4 times:
at `cmp r3, #3`
`r3 = 1, 2, 3, 4`
`ble` not taken in the last iteration

Total cycles = 1 (mov)
+ (1 (lsl) + 2 (str) + 2 (str) + 1 (add) + 1 (cmp) + 2 (bne)) * 3
+ 1 (lsl) + 2 (str) + 2 (str) + 1 (add) + 1 (cmp) + 1 (bne)
+ 3 (bx)

= <u>39</u>