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

Real-time I/O

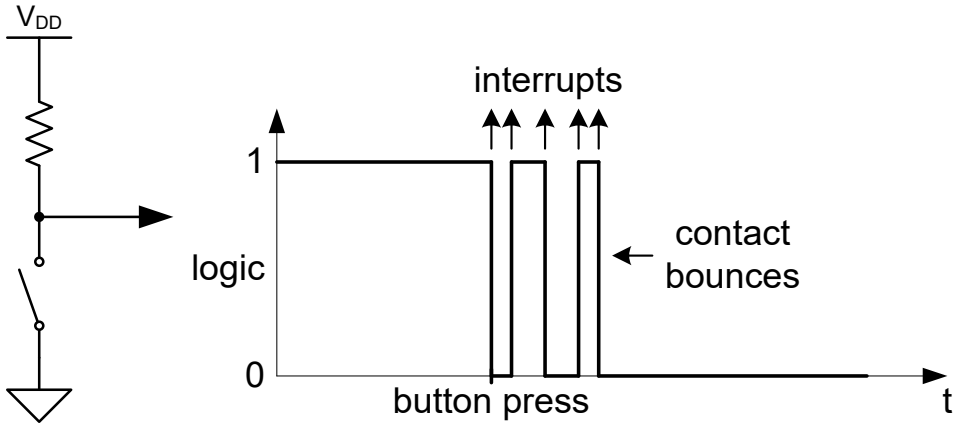
1) Short answers.

   a) How does the CAN (Controller Area Network) interface handle multiple nodes transmitting at the same time?

CAN logic levels are Dominant (driven) and Recessive (high impedance). A Dominant bit overrides a Recessive bit.

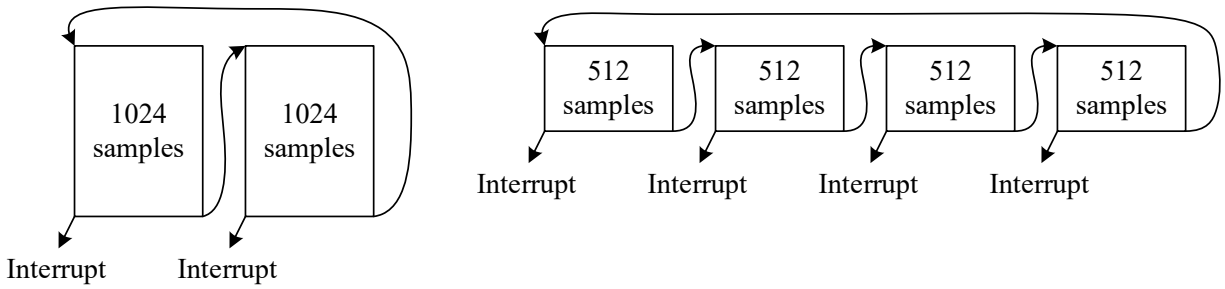
Each transmitting node listens to its own transmission. If the received bit differs from the transmitted bit (because the transmitted Recessive bit was overridden by a Dominant bit from another node), the node stops transmitting (loses arbitration) and waits for the message to finish before retrying.

b) For button debouncing, explain why it is better to sample the input signal in a **periodic task** rather than having the edges of the input signal cause **interrupts**. A typical button input circuit is shown below. Assume a working debouncing algorithm in either case.



A periodic task has predictable CPU load and fits into the RMS schedule of the system. If input edges cause interrupts, the contact bounces may trigger an unpredictable number of interrupts in a short time. The button ISR may temporarily starve lower priority real-time tasks of the CPU and cause them to miss deadlines.

c) In Lab 3 you configured the ADC DMA in ping-pong mode with **two 1024-sample buffers** (left diagram). For a generic real-time system (not your lab), what is the primary benefit to using **four 512-sample buffers** (right diagram) instead? Assume that on every DMA interrupt, the DMA ISR keeps the continuous DMA transfer going, and signals a Task to process the completed DMA buffer.



With 512-sample buffers, the processing Task has a longer relative deadline. Therefore, this Task can tolerate higher latency.

After a DMA interrupt indicating completion of a buffer, the relative deadline for the processing Task is when the DMA needs to access that buffer again. With two 1024-sample buffers, the relative deadline is 1024 sample periods (time to fill one buffer). With four 512-sample buffers, the relative deadline is 1536 sample periods (time to fill three buffers).

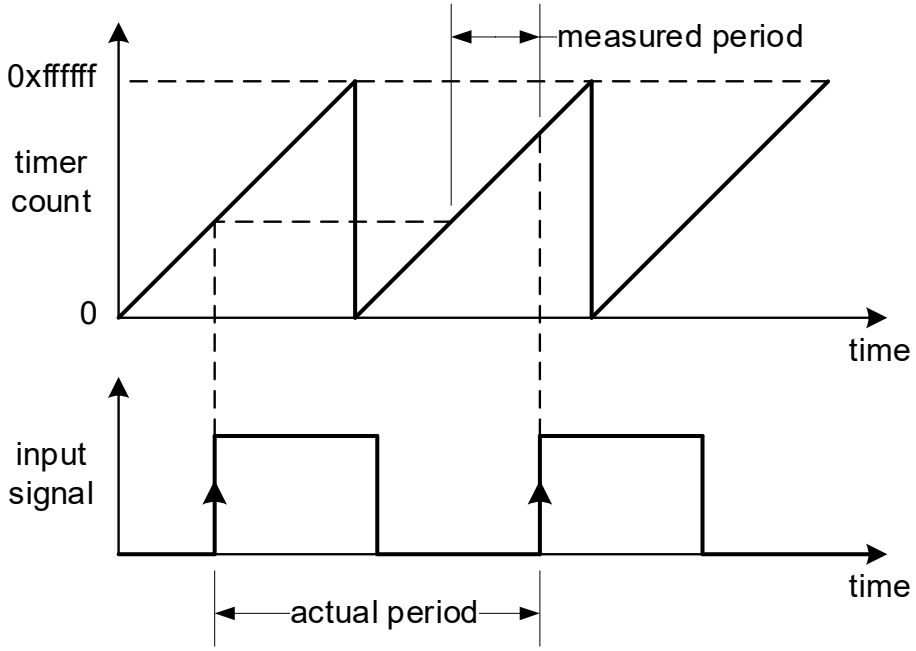
Timer I/O

2) You are using a 24-bit timer (up counting, fed by the CPU clock) in capture mode to measure the period of an input square wave. The ISR is shown below. What type of measurement error occurs when the input waveform period exceeds 0xffffff CPU clock cycles? Explain using a graph of timer count versus time.

```
uint32_t period; // measured period in clock cycles

void TimerCaptureISR1(void) { // ISR for capture interrupts
    static uint32_t last_count = 0;
    uint32_t count;

    <clear timer capture interrupt flag>;
    count = <read captured timer count>;
    period = (count - last_count) & 0xffffff;
    last_count = count;
}
```



The `& 0xffffff` operation in the ISR always produces a period measurement <= 0xffffff CPU cycles. An input waveform period exceeding 0xffffff CPU cycles results in a measurement that is less than the actual period by an integer multiple of the timer period (0x1000000 CPU clock cycles).

ARM assembly

3) 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.

```
uint32_t i;      // r0 = i;
uint32_t A[32];  // r1 = &A[0];

if (i < 32) {
    if (A[i] & 7 == 7) {
         A[i]++;
    }
}
else {
    i = 0;
}
```

```
        cmp r0, #32       ; compare i to 32
        bhs else1         ; if i >= 32, goto else1

        lsl r2, r0, #2    ; r2 = i * 4; // array offset
        ldr r3, [r1, r2]  ; r3 = copy of A[i];
        and r4, r3, #7    ; r4 = r3 & 7;
        cmp r4, #7        ; compare r4 to 7
        bne done1         ; if r4 != 7, goto done1

        add r3, r3, #1    ; r3++;
        str r3, [r1, r2]  ; A[i] = r3;
        b   done1         ; goto done1
else1
        mov r0, #0        ; i = 0;
done1
```

ARM assembly functions

4) 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 a);

int32_t f2(int32_t x, int32_t y)
{
    return f1(x) - x + f1(y) - y;
}
```

```
; upon entry into the function:
;   r0 = x;
;   r1 = y;
; inside the function:
;   r4 = x; // then reused to hold (f1(x) – x)
;   r5 = y;
f2      push {r4, r5, lr} ; preserve registers on the stack

        mov  r4, r0       ; r4 = x;
        mov  r5, r1       ; r5 = y;

        bl   f1           ; call f1(x); // argument already set up
        sub  r4, r0, r4   ; r4 = (return value of f1(x)) - x;

        mov  r0, r5       ; argument #1 = y;
        bl   f1           ; call f1(y);
        add  r0, r4, r0   ; r0 = r4 + (return value of f1(y));
        sub  r0, r0, r5   ; r0 = r0 – y; // r0 = return value

        pop  {r4, r5, pc} ; restore registers and return
```

Instruction-level performance analysis

5) Determine the **maximum** number of clock cycles required to execute the following ARM assembly function on a Cortex-M4 system. Show your work.

```
; void func(int *p1, int *p2);

func    ldr r2, [r0]
; LDR without offset, not following another LDR/STR: 2 cycles

        ldr r3, [r1]
; LDR without offset, pipelined after another LDR/STR: 1 cycle

        cmp r2, r3
; Data operation: 1 cycle

        bne done1
; Branch with immediate destination: taken        2 cycles
;                                    not taken    1 cycle
        mov r2, #0
; Data operation: 1 cycle

        str r2, [r0]
; STR without offset: 1 cycle

        str r2, [r1]
; STR without offset: 1 cycle

done1   bx  lr
; Branch with register destination: always taken   3 cycles
```

There are two paths of execution, depending on whether the `bne` branch is taken or not.

`bne` not taken:

clock cycles = 2 + 1 + 1 + 1 (bne) + 1 (mov) + 1 (str) + 1 (str) + 3 (bx) = 11

`bne` taken:

clock cycles = 2 + 1 + 1 + 2 (bne) + 3 (bx) = 9

<u>maximum</u> number of clock cycles = <u>11</u>.