EECS 373 Practice Midterm / Homework #3
Winter 2015
Due February 19th

Name: ___________________________ Uniquename: ___________________________

Sign the honor code:

I have neither given nor received aid on this exam nor observed anyone else doing so.

<table>
<thead>
<tr>
<th>Problem #</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
</tr>
</tbody>
</table>

NOTES:
- PUT YOU NAME/UNIQUE NAME ON EVERY PAGE TO ENSURE CREDIT!
- Can refer to the ARM Assembly Quick Ref. Guide and 1 page front/back cheat sheet
- Can use a basic/scientific calculator (but not a phone, PDA, or computer)
- Don’t spend too much time on any one problem.
- You have 80 minutes for the exam.
- The exam is 9 pages long, including the cover sheet.
- Show your work and explain what you are doing. Partial credit w/o this is rare.
1) **Fill-in-the-blank or circle the best answer. [10 pts, all or no points for each question]**

a) The ARM EABI specifies that registers \( r0 - r3 \) are caller-save.

b) The ARM Thumb-2 instruction set has **16-bit/32-bit/16- and 32-bit** encodings.

c) To generate 8 ns pulses, a PWM controller with a 50 MHz clock needs a \( 40\% \) duty cycle.

d) An ARM EABI-compliant procedure that calls another procedure should **always**/

   **sometimes**/never save and restore the \( lr \) register.

   \[ \text{\textasteriskcentered} \]

   \( **UART/\text{SPI/I2C}** \) supports flow control.

e) **UART/\text{SPI/I2C}** has built-in support for error checking by parity.

f) \( **UART/\text{SPI/I2C}** \) has built-in support for error checking by parity.

   \[ \text{\textasteriskcentered} \]

g) SPI bus transfers are **asynchronous/synchronous** and use **dedicated chip select lines/*addresses embedded in frames**.

h) In the Verilog hardware description language, an @ ( \( \text{posedge clock} \) ) block

   would be used when implementing a flip-flop.

i) If multiple devices share a single interrupt line and generate interrupts at the same rate,

   then processor workload **remains constant/grows linearly/grows quadratically** with

   the number of devices.

j) By default, uninitialized global variables go in the **text/data/.bss** section.

   \[ \text{\textasteriskcentered} \]

   This was a bad question (sorry). The DTS/CTS signals in UART

   are also a form of flow control (although less powerful than I2C's).

   We would accept either (or both) UART and I2C as answers.
Name: 

2) **Hardware Design.** [5pts]

Imagine an arm with 3 positions: 0, 1, and 2. A motor controls this arm. The motor can be driven forward by asserting MOTOR_FWD and in reverse by asserting MOTOR_REV. It is an error to assert both signals.

The arm position is measured by an encoder that reports current arm position as an 8-bit value. Mechanical stops prevent the arm from going past 0 to the left or 255 to the right. The positions are at encoder values 20, 127, and 235. You may assume that the arm has no inertia, that is, if neither MOTOR_FWD nor MOTOR_REV are asserted, the ARM_POSITION will not change.

Write a hardware module to control this arm. Your module should move the arm to the TARGET_POSITION only when the MOVE_REQ signal is asserted. POSITION will not change while MOVE_REQ is high. Your module should assert MOVE_DONE when the arm is in the desired position. It should then wait until MOVE_REQ is de-asserted and de-assert MOVE_DONE in response. MOVE_REQ will not re-assert until MOVE_DONE is de-asserted.

```verilog
module arm_control(
    input CLOCK,

    // control interface
    input [1:0] TARGET_POSITION,
    input MOVE_REQ,
    output MOVE_DONE,

    // arm interface
    input [7:0] ARM_POSITION
    output MOTOR_FWD,
    output MOTOR_REV
)

`define POS_0 8'd20
`define POS_1 8'd127
`define POS_2 8'd235

reg MOVE_DONE, MOTOR_FWD, MOTOR_REV;
reg next DONE, next FWD, next REV;

always @(posedge CLOCK) begin
    if (next FWD == 1)
        next REV = 0;
    end
    endmodule
```
3) Memory-Mapped I/O. [20 pts]

Using only basic circuit elements (e.g. AND and OR gates), sketch the glue logic required to interface the arm control module from Question 2 to the APB. Recall, the PSEL line is the peripheral select (i.e. it goes high when the processor is addressing the arm). The POSITION will be attached to data bits [1:0]. You should be able to change the arm position by writing to memory and read the current value by reading. A memory write should not return until the move is complete. The APB signals are: PCLK, PADDR, PWRITE, PSEL, PENABLE, PWDATA, PRDATA, and PREADY.

Sketch a timing diagram of an APB transaction that moves the arm from position 1 to position 2. Assume the arm peripheral is at address 0x20000000. You only need to fill in relevant signals.

Rubric (1pt/each)

A) PSIZE 1?
DATA correct?
PADDR correct?
SEL 1?
B) SELECT, ADDR, DATA, WRITE SET before PENABLE
C) PENABLE low in response to ENABLE
D) PREADY assertion before other changes
E) End transaction by pulling ENABLE low
4) ARM Assembly Language.  10 pts

<table>
<thead>
<tr>
<th>Encoding T1</th>
<th>ARMv7-M</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV&lt;==&lt;Rd,#&lt;imm8&gt;</td>
<td></td>
</tr>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td>1 1 1 1 0 1 0 1 1 0 0 0</td>
</tr>
</tbody>
</table>

\[ d = \text{UInt}(Rd); \text{setflags} = \text{HIITBlock}(); \text{imm32} = \text{ZeroExtend}(|\text{imm8}|, 32); \text{carry} = \text{APSAR.C}; \text{shift_t, shift_n} = (\text{SRType}, \text{LSL}, 0); \]

<table>
<thead>
<tr>
<th>Encoding T1</th>
<th>All versions of the Thumb instruction set.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV&lt;==&lt;Rd,#&lt;imm8&gt;</td>
<td></td>
</tr>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td>0 0 1 0 0</td>
</tr>
</tbody>
</table>

\[ d = \text{UInt}(Rd); \text{setflags} = \text{HIITBlock}(); \text{imm32} = \text{ZeroExtend}(|\text{imm8}|, 32); \text{carry} = \text{APSAR.C}; \text{shift_t, shift_n} = (\text{SRType}, \text{LSL}, 0); \]

a) What does the instruction 0x5088 do when executed?

Store the value at MEM[9(R1)+R2]

Write the value of register R# to the address R1+R2

b) Fill in machine code in hex around 0x5088 such that the arm peripheral from the previous question moves to position 1 when the instructions are executed.

| 0x2200  | MOVs, R2, #0 |
| 0x2180  | MOVs, R1, #0x80 |
| 0xEF2A20100  | MOV R1, 0x20000000 |
| 0x2001  | MOVs R#, #1 |

Note: order (mostly) doesn't matter [except MOVT]
- Can flip R1 and R2

Rubric
- 2 pts: Set R# to 1
- 4 pts: Set R1+R2 to 0x2000080
- 2 pts: movr after movs
5) Assembly, C, and the ABI. 15 pts

Recall the arm peripheral is at address 0x20000010 and encodes the arm position in the bottom two data bits.

a.) Write a C function, `bool check_move(int change)`, that takes a move and returns 1 if it is a possible move and 0 if the move cannot be made. The change argument describes the change of arm position, e.g., change “2” would move an arm from position 0 to position 0+2 = 2 and change “-1” would return an error if the arm were in position 0. This function should **not** move the arm.

```c
bool check_move(int change) {
    int32_t current = *(volatile int32_t *) 0x20000010;
    if ((current + change) > 0)
        return 0;
    if ((current + change) < 0)
        return 0;
    return 1;
}
```

b.) Write an assembly function `int move_arm(unsigned steps, bool reverse)` that moves the arm peripheral the requested number of steps forward or backward. Your function **must** call the `check_move()` function from (a) to verify if the requested move is valid before attempting to move the arm. `move_arm()` should return the current arm position, regardless of whether the arm moved or not.

The assembly code should assemble without errors. Write clearly and **comment the code**.

```
label:
move_arm:
    push r4, r5, r6    // callee-save
    cmp r5, #1
    it eq    // if reverse, -steps
        neg eq r5
    movs r5, r6    // save steps
    bl check-move   // call check
    -5 pts if no attempt to call check-move
    // 1 for other errors
    // ARM moves if check succeeds
    add eq r4, r5    // conditionally set new arm position
    streq r4, [r2, #0]    // load arm pos.
    mov r4, 0x80    // load periph address
    mov r6, 0x20000010
move r6, r5
    // set up return value & return
    pop r4, r5, r6    // callee, restore
    // pop pc or bx lr to return
    // This is 15 instructions + 1 EC for 13 instrs + 3 EC for
    // less than 13 instrs Must be correct for any EC.
```

Rubric

-1 point for each error
6) Write a C function `void enable_interrupts(int x)` that enables interrupt `x`. You need not check to validate that `x` is a legal interrupt number. The table below might be useful.

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Reset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xE000E004</td>
<td>ICTR</td>
<td>RO</td>
<td>-</td>
<td>Interrupt Controller Type Register, ICTR</td>
</tr>
<tr>
<td>0xE000E100 - 0xE000E11C</td>
<td>NVIC_ISER0 - NVIC_ISER7</td>
<td>RW</td>
<td>0x00000000</td>
<td>Interrupt Set-Enable Registers</td>
</tr>
<tr>
<td>0xE000E180 - 0xE000E19C</td>
<td>NVIC_ICER0 - NVIC_ICER7</td>
<td>RW</td>
<td>0x00000000</td>
<td>Interrupt Clear-Enable Registers</td>
</tr>
<tr>
<td>0xE000E200 - 0xE000E21C</td>
<td>NVIC_ISPR0 - NVIC_ISPR7</td>
<td>RW</td>
<td>0x00000000</td>
<td>Interrupt Set-Pending Registers</td>
</tr>
<tr>
<td>0xE000E280 - 0xE000E29C</td>
<td>NVIC_ICPR0 - NVIC_ICPR7</td>
<td>RW</td>
<td>0x00000000</td>
<td>Interrupt Clear-Pending Registers</td>
</tr>
<tr>
<td>0xE000E300 - 0xE000E31C</td>
<td>NVIC_IABR0 - NVIC_IABR7</td>
<td>RO</td>
<td>0x00000000</td>
<td>Interrupt Active Bit Register</td>
</tr>
<tr>
<td>0xE000E400 - 0xE000E4EC</td>
<td>NVIC_IPR0 - NVIC_IPR59</td>
<td>RW</td>
<td>0x00000000</td>
<td>Interrupt Priority Register</td>
</tr>
</tbody>
</table>

```c
void enable_interrupts(int x) {
    uint32_t *base = (uint32_t *) 0xE000E100;  // 2's for MMU2 syntax
    unsigned idx = x >> 5;  
    unsigned bit = x & 0x1f;  
    uint32_t mask = 1 << bit;  
    *(base + idx) = mask;  
}
```
7) Startup and Interrupts.

In addition to the motor, the system has two buttons. Button0 is wired to Interrupt 7 and Button1 is wired to Interrupt 8.Whenever Button0 is pressed, the arm should move left (decrement position). If Button1 is pressed, the arm should move right (increment position).

Using the functions from Questions 5 and 6, write any remaining code required such that the arm goes to position 1 when the system powers on and the buttons behave as intended.

```c
// vector table

1. Word reset_fn
2. Word reset_fn
3. Word reset_fn
4. Word reset_fn
5. Word reset_fn
6. Word reset_fn
7. Word decr_fn
8. Word incr_fn

5 pts for vector table
1 pt for Vector top
2 pt reset handler
2 pt int7/int8 handlers
-1 pt per minor error
if appropriate

void reset_fn (void) {
    enable_interts (7);
    enable_interts (8);
    while (1) {
        asm("sti ;");
    }
}

void incr_fn (void) {
    more_arm (1, 0);
}

void deccr_fn (void) {
    more_arm (1, 1);
}

5 pts for main function
2 pts for enabling interrupts
2 pts for while loop
(or other non-busy wait idea)
1 pt for "sti"
(or other non-busy wait idea)

5 pts for interrupts handlers
2 pts / handler
1 pt for correct more_arm calls
```
8) Logic Design [15 pts].

The motor controller actually requires a PWM signal to operate. Given a 50% duty cycle, the motor will not move. At 70% it will drive forward, and at 30% it will drive in reverse. Design a digital circuit with inputs MOTOR_FWD and MOTOR_REV that takes the system clock of 100 MHz (CLkin) and converts it to an output clock of 5 MHz with a 30, 50, or 70% duty cycle (MOTOR_CLK). You may use synchronously resettable D flip-flops and n-bit binary counters (make sure to specify the value of n). You may express combinational logic as Boolean expressions or using standard logic gates. You may assume that MOTOR_FWD and MOTOR_REV will never assert at the same time. Label things and write neatly.
move_arm:
  push {r4, r5, lr}

  cmp r1, 1
  it eq
  negeq r0
  movs r5, r0
  bl check_fn
  cmp r0, 1

  ldr r4, addr
  ldr r0, [r4, 0]

  itt eq
  addseq r0, r5
  streq r0, [r4, 0]

  pop {r4, r5, pc}

addr: .word 0x20000080