EECS 373 Midterm 1

20 February 2024

No calculators, reference material, internet, or
communicating with others about the exam (except course staff).

Name

UM Uniqname

Sign below to acknowledge the Engineering Honor Code: “I have neither given nor received aid on this examination, nor have I concealed a violation of the Honor Code.” “Concealed” should be interpreted as “have failed or will fail to report”.

Signature

1 [13 pts.] Short questions

1. [3 pts.] If the embedded market is larger than the general-purpose computing market, why is it common for embedded systems designers to work on smaller teams? Select the single most correct answer.

   - ◯ Embedded systems are generally less expensive than general-purpose computers so the total market capitalization for embedded systems is lower, leading to less money to hire embedded systems engineers.
   - ◯ ✓The market is fragmented, with more varieties of applications and products. This increases the number of teams required, reducing the number of embedded systems designers per team.
   - ◯ Embedded system designers are more productive than general-purpose computer system designers, so fewer are needed to accomplish the same amount of work.
   - ◯ Debugging complexity is linearly related to processor clock frequency, making low-frequency embedded microcontroller based systems much easier to debug than high-frequency general-purpose systems.
   - ◯ There are a very wide range of general-purpose processors available, but only a few embedded processors. This enables fewer embedded system designers to manage processor-based design complexity.
   - ◯ Unlike general-purpose computer system designers, embedded systems designers use FPGAs in the prototyping process, making it easy for fewer designers to complete the same work.

2. [3 pts.] Indicate all system design decisions that will likely increase total design and debugging time.

   - ◯ Adding assertions for facts that appear to be obviously correct to one's code.
   - ◯ ✓Deferring all error checking to run-time.
   - ◯ Adding infinite loop traps to unused ISRs that should never be invoked.
   - ◯ Writing special values to memory locations to enable checking whether they were inappropriately overwritten.
   - ◯ ✓Ensuring that all tests are done on the entire system in a realistic environment approximating its expected future use.

   Compile-time error checking is better for errors it is capable of detecting. Sub-systems should be tested before integration.

3. [3 pts.] Consider a virtual timer system in which the bookkeeping information on individual timers is stored in a container sorted in order of increasing timer firing times. You must decide whether to implement the container using an array or linked list. Indicate all the application characteristics that indicate that an array would be better, or for which a linked list would provide no advantages over an array.

   - ◯ Virtual timers will most frequently be periodic/recurring.
   - ◯ ✓New timers will always have firing times greater than all previously created timers.
   - ◯ The application requires floating point math.
   - ◯ The number of virtual timers required is unknown and unbounded at compile time.
   - ◯ The timers must execute callbacks provided via function pointers.

   This one was a little tricky to grade because I wanted to make sure that correct answers based on any reasonable interpretation of the question would receive credit. If "unknown and unbounded" was selected, no credit was given. That would make it difficult to determine the size of the array. If that was unmarked and "firing times greater" was marked, then full credit was given. That would make appending (instead of insertion, which may be expensive in an array) practical. The other ones didn't favor arrays or lists so marks or lack of marks on those received no penalty.

4. [2 pts.] If we were to model the debugging time of a complex system as 2^{e+v} minutes where v is the number of vertices and e is the number of edges in a component interaction graph, then what is the expected debugging time (in minutes) for a system with four components, each of which may interact with any other component? Treat this as an undirected graph, i.e., an interaction between A and B is captured by the same edge as an interaction between B and A.

   $2^{4+6} = 1024$.

5. [2 pts.] Continuing from the prior question, if we were to limit the number of interactions to two per component, what would the expected debugging time (in minutes) be?

   $2^{4+4} = 256$.

2 [10 pts.] MMIO

1. [5 pts.] Indicate all of the following that are true.

   ◯ Accessing MMIO requires special load and store instructions.

   ◯ ✓Accessing MMIO can be slower than accessing physical memory.

   ◯ All MMIO transactions happen entirely on the AHB.

   ◯ Using MMIO is always faster than using a dedicated I/O bus.

   ◯ MMIO is only used to communicate with peripherals external to the microcontroller.

2. [5 pts.] Consider the following code segment.

```
volatile uint32_t *addr = 0x1234ABCD;
uint32_t x = *addr;
x = *addr;
```

   Use at most one sentence to indicate what may go wrong if the volatile keyword on the first line were not used.

   Volatile is used to ensure the compiler does not not assume that the value of the memory location pointed to may only be changed by the program. It prevents elimination of (needed) reads in cases where memory values may change outside the program, e.g., due to buttons and sensors.

3 [20 pts.] APB

You are designing a smart brooder for chicks. You are given an APB memory-mapped I/O hardware device that has three distance sensors and three LEDs. Each distance sensor is configured to output a high signal when one or more chicks is near it and a low signal when no chick is near it. The goal is to allow remote monitoring of chick locations to enable adjustment to brooder temperature profiles. The LEDs are active low. Each distance sensor has an associated read-only 32-bit memory address, with the sensor's output transmitted in bit 31. The LEDs have a shared write-only 32-bit memory address. LEDs A, B, and C are controlled by bits in positions 0, 1, and 2, respectively. Below is the implementation of the Verilog module that can be controlled by writing to the memory-mapped I/O registers.

```
module chicks_in_a_house (
input PCLK, PRESERN, PSEL, PENABLE, PWRITE
output PREADY, PSLVERR,
input [7:0] PADDR,
input [31:0] PWDATA,
output [31:0] PRDATA,
input dist_sen_a, dist_sen_b, dist_sen_c,
output led_a, led_b, led_c);

  assign PSLVERR = 0;
  assign PREADY = PENABLE;

  wire enable, read_a_en, read_b_en, read_c_en;

  assign enable = PSEL & PENABLE & PWRITE & (PADDR == 0);
  assign read_a_en = PSEL & PENABLE & ~PWRITE & (PADDR == 4);
  assign read_b_en = PSEL & PENABLE & ~PWRITE & (PADDR == 8);
  assign read_c_en = PSEL & PENABLE & ~PWRITE & (PADDR == 12);

  assign PRDATA[31] =
  (read_a_en) ? dist_sen_a :
  (read_b_en) ? dist_sen_b :
  (read_c_en) ? dist_sen_c :
  1'bz;

  always @(posedge PCLK) begin
    if (enable) begin
      led_a <= PWDATA[0];
      led_b <= PWDATA[1];
      led_c <= PWDATA[2];
    end
  end
endmodule
```

1. [12 pts.] In the following sub-problems, you will write a C program that reads in the state of the distance sensors and turns on the corresponding LED. Distance sensor A is associated with LED A, B with B, and C with C. Assume that PSEL is configured to be high when memory locations 0x80080000–0x8008000F are accessed. You may assume that the distance sensor measurement will not change while the functions are running.

(a) [4 pts.] Write the body of a function that reads one of the distance sensors, where the sensor selection index ranges from 0-2.

```
uint32_t read_distance_sensor (uint32_t index) {

  return *(uint32_t *)(0x80080004 + index * 4) & (0b1 << 31);
}
```

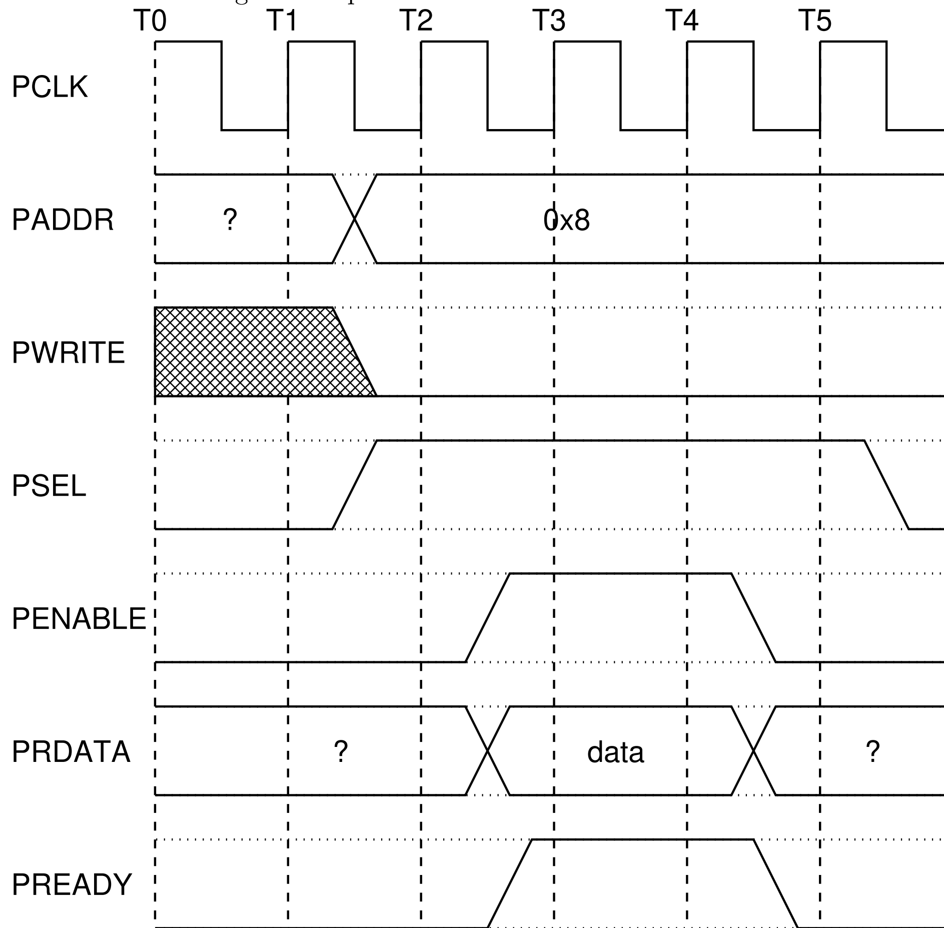
(b) [4 pts.] Write the body of a function that writes to the LED indicated by the index.

```
void write_led (uint32_t index, uint32_t val) {

  // Clear bit
  *(uint32_t *)(0x80080000) &= ~(0b1 << index);

// Write bit
  *(uint32_t *)(0x80080000) |= (val << index);
}
```

(c) [4 pts.] Write a function that uses the `read_distance_sensor` and `write_led` functions to set the LEDs appropriately based on the distance sensor states.

```
void update_leds {

  for (uint32_t i = 0; i < 3; ++i) {
    write_led(i, read_distance_sensor(i));
  }
}
```

2. [8 pts.] Fill in the unspecified signals in the timing diagram bwlow for a read from distance sensor B. The PWRITE and CLK signals are provided.



4 [20 pts.] Embedded ANSI C

Your friend is writing ANSI C for an ARM Cortex M4 microcontroller. Assume access to a linker script that gives the following memory layout.

High memory address	
Heap	1 KB
Data	50 KB
Stack	1 KB
Text	100 KB
Low memory address	

Assume the stack grows toward lower memory addresses. Your friend has written the following main.c file.

```
1  #include <stdint.h>
2
3  #define SIGNIFICANT_VALUE 0xDEADBEEF
4
5  /* Adds el to buffer at index, then increments index, wrapping
6   * around if needed.
7   * Implementation not shown, assume it is correctly implemented */
8  extern void pushCircularBuffer(char* buffer, int size, uint32_t* index, char el);
9
10 #define BUFFER_SIZE 10000
11 uint32_t currentIndex = 0;
12
13 int main(void) {
14 uint32_t *peripheralRegister = (uint32_t *) 0x40000000;
15 char circularBuffer[BUFFER_SIZE];
16
17  while (1) {
18      if (*peripheralRegister == SIGNIFICANT_VALUE) {
19          incrementCounter();
20          pushCircularBuffer(circularBuffer, BUFFER_SIZE, &currentIndex, 'Y');
21      } else {
22          pushCircularBuffer(circularBuffer, BUFFER_SIZE, &currentIndex, 'N');
23      }
24   }
15 }
16
17 void incrementCounter(void) {
18     static int counter = 0;
19     ++counter;
20 }
```

1. [8 pts.] For the following symbols, indicate what region of memory their underlying value is stored in. If a symbol doesn't correspond to something stored in memory, write "n.a.". If a value is a pointer, indicate where the pointer is stored, not the address it points to.

Symbol	**Region**
peripheralRegister	**Stack**
currentIndex	**Data**
circularBuffer	**Stack**
incrementCounter	**Text**
SIGNIFICANT_VALUE	**n.a. or Text accepted**

2. [4 pts.] Your friend attempts to cross-compile and link their program with the ARM GNU toolchain, and uses the -O3 and -Werror flags. The program fails to compile. Give the number of one line of the program that can be changed to correct the compilation problem, and write the new contents of the line. It is fine to replace any numbered line, even one that is currently empty.

The author of this question identified a problem with it so the version shown here differs from that appearing in the exam. In grading, he sought to avoid penalizing students for ambiguity or problems with the question. If you think you were penalized inappropriately, please submit a grade change request via Gradescope.

Line number: **12**
New line contents: **void incrementCounter(void);**

3. [4 pts.] After your change, the binary is successfully built and flashed to the microcontroller. However, a HARD_FAULT signal is generated soon after the program starts execution. Describe the most likely cause of the hard fault, and indicate the change you would make to correct it. Reference line numbers when possible.
   A stack overflow. A 10 kB array is allocated on the stack in line 15, despite the linker script only providing 1 kB of stack space. The array should be moved to global scope OR declared static OR reduced in size to (well) below 1 kB OR modify the linker script to provide a larger stack. Another answer would be peripheralAddress points to a 36 bit address. Dereferencing it on a 32 bit architecture will likely lead to a hard fault, especially if it was truncated to 0x0. Double check the address (likely remove an extra 0).

4. [4 pts.] After your change, the program no longer faults. However, after single-stepping with a debugger, you observe unexpected behavior: the counter doesn't increment despite the peripheral register holding SIGNIFICANT_VALUE. Modify one line to fix the unexpected behavior. Indicate the line you changed, write the revised version, and explain why your solution is correct below.

   Line number: **14**
   New line contents and explanation:
   The peripheral register pointer needs to be declared with the volatile keyword to ensure the value is actually pulled from memory on each iteration and not optimized out.

5 [25 pts.] Assembly and ABI

1. [10 pts.] Assembly.

   The function `rand` is an ABI compliant function with the following prototype: `int rand(void)`. The function `swap_randomly` has the following prototype: `void swap_randomly(int *a, int *b)`. The following ARM assembly code attempts to implement the `swap_randomly` function. The empty boxes are places where you might later add code; ignore them for now.

   ```
   1  swap_randomly:
   2  push {r6, r7}
   3    ldr r6, [r0]  // load int from a
   4    ldr r7, [r1]  // load int from b
   5  push {r0, r1, lr}
   6    bl rand
   7
   8    and r0, r0, #1
   9    cmp r0, #0
   10 pop {r0, r1, lr}
   11   beq end
   12   str r7, [r0]
   13   str r6, [r1]
   14
   15 end:
   16 pop {r6, r7}
   17   bx lr
   ```

   Unfortunately, `swap_randomly` it is not ABI-compliant. Make the additions in the boxes above needed for it to be ABI compliant, correct, and efficient. There exists a correct solution in which only four lines of code are added in total to some of the numbered but empty lines in the version shown above.

2. [15 pts.] ABI.

   Write an ARM assembly language procedure that implements the C function `naive_shuffle` in an ABI-compliant manner. Clearly comment code and label which registers each value represents. Code comments enable partial credit. Assume that we are using the ABI-compliant implementation of the function `swap_randomly`.

   ```
   void naive_shuffle(int *array, int input_size) {
     int i = 1;
     while (i < input_size) {
       int *a = array + i;
       int *b = a - 1;
       swap_randomly(a, b);
       ++i;
     }
   }
   ```

Although you are not required to use the following assembly instructions, some might be helpful: ADD, B, BEQ, BGE, BL, BX, CMP, LDR, LSL, MUL, MOV, POP, PUSH, SUB, and STRB.

```
naive_shuffle:
push {r4, r5}
  mov r2, #1 // r2 = i
  mov r4, r0 // r4 = array
  mov r5, r1 // r5 = input_size
loop:
  cmp r2, r5
  bge done
  lsl r3, r2, #2  // r3 = i * 4
  add r0, r4, r3 // r0 = a
  sub r1, r0, #4 // r1 = b
  push {r0, r1, r2, r3, lr}
  bl swap_randomly
  pop {r0, r1, r2, r3, lr}
  add r2, r2, #1
  b loop
done:
  pop {r4, r5}
  bx lr
```

6 [12 pts.] Interrupts

1. [3 pts.] Which of the following maps interrupt numbers to ISR addresses?

   ◯ ✓NVIC

   ◯ MMIO

   ◯ APB bus

   ◯ AHB bus

   ◯ EXTI controller

2. [5 pts.] A single interrupt occurs during the execution of a process. Label the following 1–5 in the order in which they happen. 1 is what happens first and 5 is what happens last. One blank will have two numbers in it.

   - **3** Interrupt pending bit goes high.
   - **2** External event in the world.
   - **4** Executing in ISR mode.
   - **1 and 5** Executing in thread mode.

3. [4 pts.] Use at most three sentences to describe a scenario in which tail chaining happens. Be specific and include relative times of when events happen. How does tail chaining improve performance of a system?

   Interrupt A occurs at t=0 and executes until t=2. At t=1, interrupts B of lower preemption priority occurs. The core will put registers on the stack at t=0 and execute the ISR for interrupt 1, but at t=2 execution will shift directly to the ISR for interrupt B without returning to the normal execution mode. This is known as tail chaining and it improves performance as there is no need to push/pop state to/from the stack between the execution of ISR A and ISR B.

This page may be used for work. Please hand it in with the exam and reference it from the associated questions if you would like it to be considered when determining partial credit.