Final Exam

Date: May 12, 2017

UT EID: ________________________

Printed Name: ____
Last, First

Your signature is your promise that you have not cheated and will not cheat on this exam, nor will you help others to cheat on this exam:

Signature:

Instructions:

- Closed book and closed notes. No books, no papers, no data sheets (other than the last two pages of this Exam)
- No devices other than pencil, pen, eraser (no calculators, no electronic devices), please turn cell phones off.
- Please be sure that your answers to all questions (and all supporting work that is required) are contained in the space (boxes) provided. Do Not write answers on back of pages
- You have 180 minutes, so allocate your time accordingly.
- Unless otherwise stated, make all I/O accesses friendly.
- Please read the entire exam before starting.

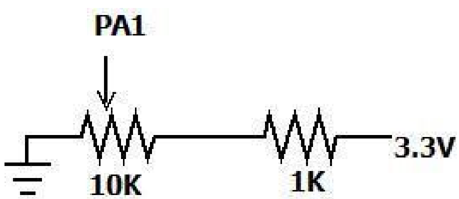
Problem 1	15	
Problem 2	15	
Problem 3	15	
Problem 4	10	
Problem 5	15	
Problem 6	15	
Problem 7	15	
Total	100	

[15 points] Problem 1: Fundamentals. Answer the following short questions in the boxes provided.

(i) (**2pt**) Explain what happens when you execute the following Stack instructions.
PUSH {R0, R2}
POP {R2, R0}

R0 and R2 do not get updated.

(ii) (**3pt**) Given the following series configuration of a 10kΩ potentiometer and a 1kΩ fixed resistor, what is the range of voltages at the input pin PA1?



0 to 3V

(iii) (**1pt**) Complete the following statement: A ____________ local variable is allocated permanent space in the RAM.

static

(iv) (**2pt**) The input signal of the ADC has frequency components ranging from 200Hz to 2kHz. What must the sampling frequency of the ADC be to faithfully reproduce the signal?

>= 4kHz

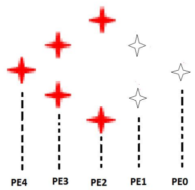
(v) (**4pt**) Let the bus clock frequency be 80MHz. Calculate the Sys[illegible] so that the timer resets every 1 μs?

79

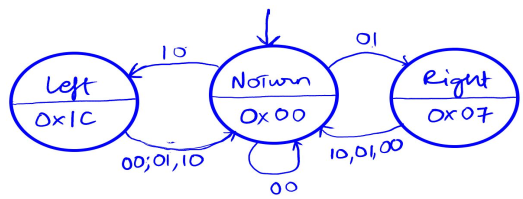
(vi) (**3pt**) A Moore FSM that you implemented in Lab5 repeated a sequence of 4 steps over and over. Complete that procedure below.

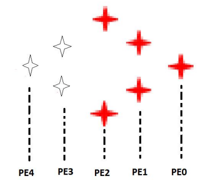
1. Perform output (depends on present state).
2. Wait (optional)
3. Read input.
4. Go to next state (depends on input and present state).

(15) Problem 2: Finite State Machine. You may recall the bicycle with turn indicators from the first midterm. The outputs are the five LEDs on PE4 to PE0 which flash when indicating a turn:



Left Indication





Right Indication

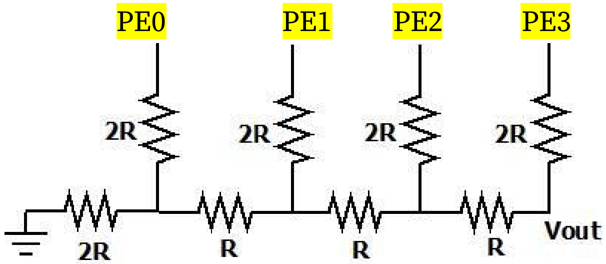
The input was an accelerometer reading which is now abstracted, so you can call the function: `uint8_t get_direction()`, which returns `00, 01 or, 10` to indicate to stay straight, turn right or turn left respectively. You are required to flash the LEDS at 5Hz with 50% duty-cycle. The state-transition graph for a Moore FSM implementation is given above (without the wait times).
Complete the code below by adding state `#defines`, blanks in the struct, FSM array size and entries, state initialization, and the FSM loop.

```
//#defines here        5Hz with 50% duty-cycle is 10 times/sec => wait = 100ms
#define NoTurn 0
#define Right 1
#define Left 2
struct State{
  uint8_t out;   // output produced in this state
  uint32_t wait; // delay in ms units;
                 // Can call delayms(count) to wait for count milliseconds
  uint8_t next[3]; // list of next states
};
typedef struct State SType;
SType FSM[3] = {
    {0x00, 100, {NoTurn, Right, Left},
    {0x07, 100, {NoTurn, NoTurn, NoTurn},
    {0x1C, 100, {NoTurn, NoTurn, NoTurn}
}
uint8_t curState = NoTurn; //set the initial state here
int main() {
      uint8_t in;
      // All Port Initialization done for you
      // Complete the FSM loop below
      …
      while(1){
          GPIO_PORTE_DATA_R = FSM[curState].out; //1: Output
          delayms(FSM[curState].wait; //2: Delay
          In = get_direction(); //3: Input
          curState = FSM[curState].next[in]; //4: Change state


      }
}
```

[15 points] Problem 3: Hardware.

Part a(10 pt): You are given the following R-2R ladder DAC circuit for a 4-bit DAC connected to the microcontroller port pins PE3 (MSB), PE2, PE1, PE0 (LSB). The output of the DAC is connected to a speaker.



(i) (**4p**t) Mark the microcontroller port pins on the circuit schematic.

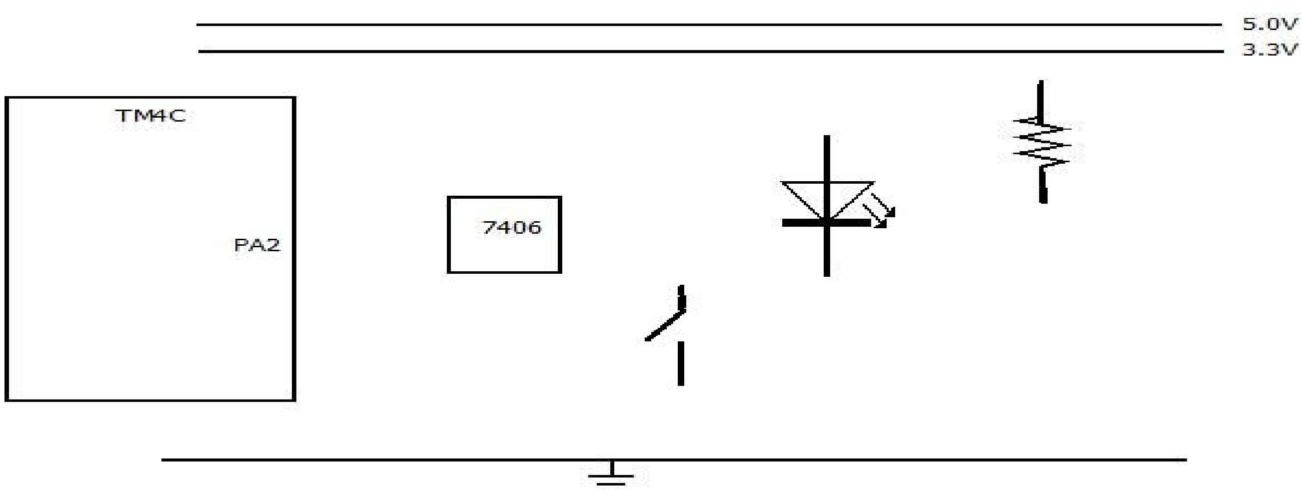
(ii) (**3pt**) A few rows in the table below have been completed for you. Complete the rest of the table. (Note that the values of Vout are rounded, and that for a R-2R ladder circuit the exact values of Vout are slightly different).

PE3	PE2	PE1	PE0	Vout (V)
0	0	0	0	0.0
0	0	0	1	0.2
0	0	1	0	0.4
0	0	1	1	0.6
0	1	0	0	0.8
1	1	1	1	3.0

(iii) (**3pt**) What is the range, resolution, and precision of this DAC?

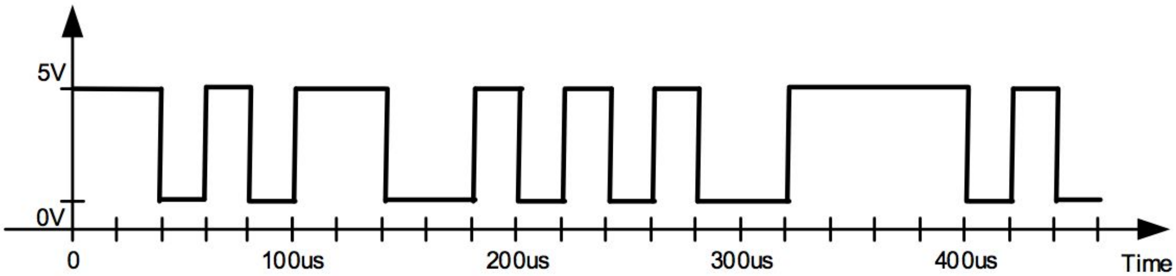
3V, 0.2V, 16

Part b(5pt): You are given a TM4C microcontroller, a 7406 driver, an LED whose desired operating point is 1.6V and 1.5mA, and resistors (of your choice). Interface this LED to PA2 using ***positive*** logic. Show your connections clearly. Assume the V_{OL} of the 7406 is 0.5V. Assume the microcontroller output voltages are V_{OH} = 3.1V and V_{OL} = 0.1V. Specify values for any resistors needed. Show equations of your calculations used to select resistor values. We do not need drivers for this. For positive logic: connect PA2 to the anode of LED, the cathode to resistor to ground. R = (3.1-1.6)/1.5m = 1k ohms.



[10 points] Problem 4: UART.

Part a (6pt): Assuming start, stop, and data bits only, mark the frame boundaries and data bits. Assume no breaks during transmission (frames are sent back-to-back).



Part b (4pt): What is the baud rate and bandwidth of this channel?

40--60us is the start bit, then 8 bits of data, then stop bit. Also, give points if they use 10us instead of 20us as the clock period. Baud rate: 1/20us. B/w: 8/10*1/20 bits/us.

[15 points] Problem 5: FIFO.

Consider a struct named `fix_pt_t` that contains two 8-bit integer fields `whole` and `frac`. Implement a FIFO of `fix_pt_t` elements. The FIFO supports the basic interface: `put`, `get`, `is_empty`, and `is_full`, which are all functions. The function `put` is passed a `fix_pt_t` element by value and `get` removes a value from the FIFO and places it in a parameter passed by reference. Let the maximum FIFO size be a defined constant `MAX_SZ`.

In addition, `get` returns the delay, in terms of number of elements added or removed from the FIFO, between when the element itself was added to the fifo with a `put` and until it is taken out with the `get`. That is, the delay represents how many times `put` or `get` were called between the insertion and removal of an element. Fill out the FIFO code below.

Note: *A correct implementation of the FIFO with struct elements is worth 10 points; implementing the delay feature will get you the full 15.*

```
#define MAX_SZ 32
// Define your struct fix_pt_t here. 1 point.
struct fix{
  uint8_t whole;
  uint8_t frac;
}
typedef const struct fix fix_pt_t;

// global variables for the FIFO. 2 points.

uint8_t delay=0;
uint32_t PutI=0,GetI=0;
fix_pt_t fifo[MAX_SZ];
uint8_t delays[MAX_SZ];
```

// implementations of `put(fix_pt_t elem)`, `uint8_t get(fix_pt_t *elem)`, `is_empty()`, `is_full()`.
// 1 point each for `is_full` and `is_empty`. 5 points each for `put` and `get with delay`.

```
int is_empty(void){
  if(PutI==GetI) return 1;
  return 0;
}
int is_full(void){
  if((PutI+1)%MAX_SZ)==GetI) return 1;
  return 0;
}
int put(fix_pt_t elem){
  if(is_full()) return 0; // failure, full
  fifo[PutI].whole = elem.whole;
  fifo[PutI].frac = elem.frac;
  delays[PutI] = delay-1; // don't count this one
  delay++;
  PutI = (PutI+1)%MAX_SZ;
  return 1;
}
uint8_t get(fix_pt_t *pt){ uint8_t d;
  if(is_empty ()) return 0; // failure, empty
  pt->whole = fifo[GetI].whole;
  pt-> frac = fifo[GetI].frac;
  d = delay-delays[GetI];
  delay++;
  GetI = (GetI+1)%MAX_SZ;
  return d;
}
```

[15 points] Problem 6: Sampling and ADC. You are responsible for an input module, which uses the TM4C123's ADC0 to provide samples to a sound-processing software module. The processing module requires a buffer of 100ms (100/1000 seconds) and assumes all samples in the buffer are 8-bit binary fixed-point numbers that linearly represent the sound input (S_in) within the range [0, 1.28) or ***0<=S_in<1.28***. The output is a speaker with a frequency range of 20Hz-25KHz. The microphone you use has the following response: ***V_mic = (S_in + 1.28)*1V***; the ADC accepts the range 0 - 2.56V.

The pertinent (and incomplete) **ADC initialization** code is (see ADC part of attached handout):

```
ADC0_PC_R      = 0x03;
ADC0_EMUX_R   |= 0xF000;
ADC0_IM_R     |= 0x0008;
ADC0_ACTSS_R  |= 0x0008;
```

Part a (2pt): How many samples per second does the ADC produce? _____ **250**___________KHz

Part b (2pt): What interrupt handler is called for every sample? ___**ADC0Seq3_Handler**___

Part c (5pt): What is the **minimum reasonable size** for the processing buffer (in bytes)?_ **5000**____B
Why this number (<20 words)?

Nyquist rate is >50KHz, 100ms means 5K samples

Part d (6pt): In the box on the next page, complete the interrupt handler which is triggered for each sample produced by the ADC. Write code in any blank area to pass samples to the buffer used by the processing module. There is way more blank space than necessary. Write C code, but if you can't, write pseudocode for partial credit. For example, if you need to disable or enable interrupts but don't know how, write something like:

```
EnableInterrupts();  // enable interrupts
DisableInterrupts(); // disable interrupts
```

<u>**Hint**</u>: *If you put every sample into the the buffer then you will need a much larger buffer than*

```
void ProcessingBufferPut(uint8_t x); // call to add to processing buffer

void XXX_Handler() { // called for each ADC0 sample
  static uint32_t count=0;
  uint8_t x;

  if (count ==  5 <FILL_THIS_IN>) { // hint: what rate do we need?
    count=0;

    x = (ADC0_SSFIFO_R - 2048) >> 3; // 2048 is 1.28V so subtract to get
                                     // S_in. But, value is now 0 - 2047
                                     // And we want an 8-bit number so
                                     // shift by 3.
    ProcessingBufferPut(x);

  }

  ++count;

   ADC0_ISC_R = 8; // Acknowledge interrupt

}
```

[15 points] *Problem 7*: Variable Fundamentals. The code below is AAPCS compliant. The relevant code given in ARM assembly is complete (main is not given). The C part is very incomplete, but you need not fully complete it nor should you worry about fully understanding the ASM. The question is about variables and types rather than directly on what the code does. Hint `bar` is called first. Recall that each variable can be local or global and permanent or temporary (e.g., local temporary, global permanent, local permanent).

```
VAR_a SPACE 4

baz
  cmp  r0, #0
  bge  BazRet
  eor  r0, r0, #0xffffffff
  add  r0, r0, #1
BazRet
  bx   lr

PartC
w equ  ???
z equ  #4
foo
  push {lr,r11,r4,r5}
  sub  sp, sp, #8
  mov  r11, sp

  mov  r4, r0
  mov  r5, #0
LabelFooA
  cmp  r1, r5
  bls  LabelFooB
  ldrsb r0, [r4,r5]
  str  r5, [r11,#z]
  ldr  r5, [r11,#w]
  blx  r5
  ldr  r5, [r2]
  add  r5, r5, r0
  str  r5, [r2]
  ldr  r5, [r11,#z]
  add  r5, r5, #1
  b    LabelFooA
LabelFooB
  ldr  r4, =VAR_a
  ldr  r0, [r4]
  add  r0, r0, #1
  str  r0, [r4]
  cmp  r3, #0
  bge  LabelFooC
  mov  r0, #0
LabelFooC
  add  sp, sp, #8
  pop  {lr,r11,r4,r5}
  bx   lr

bar
  push {lr,r9}
  sub  sp, sp, #8
  ldr  r9, =baz
  str  r9, [sp]
```

```
uint32_t baz(???     a) {
   // does something useful and returns
   // see ASM, but do not convert to C
}
???     foo(int8_t A[8],
            ??? // more parameters here
            ) {
  // Code in foo that you need not
  // complete in C. But, line below is
  // part of the code that is important
   ++a;
 // More code
}
uint32_t bar(int8_t A[8],
             uint32_t len,
             // another param
            ) {
 // code that you don't need to complete
}
```

Part a (2pt): What **sort of variable** is r3 just below PartA in ASM (e.g., global permanent)?

Local temporary

Part b (3pt): How many parameters does `foo` take?

5 (one on the stack)

Part c (2pt): What is ??? just below PartC in ASM?

#24

Part d (2pt): What **sort of variable** is **`a`** inside foo (look at C side)?

Local permanent (static)

Part e(2pt): What is the C99 type of `baz`'s input param?

int8_t

Part f (2pt): Which of foo's parameters are pass-by-reference (zero or more) -- indicate by parameter number (e.g., param8, param9).

Param1, param3

```
PartA
  and   r3, r2, #0xf
  bl    foo
  add   sp, sp, #8
  pop   {r9,pc}
```

Part g (2pt): What is the type of `foo`'s last input param? Use C syntax but explain in words otherwise.

Function pointer: uint32_t (*func)(int8_t)