Final Exam

Date: May 16, 2019

UT EID: _______KEY_________________

Printed Name: _________Solution_____________________Key___________________
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:

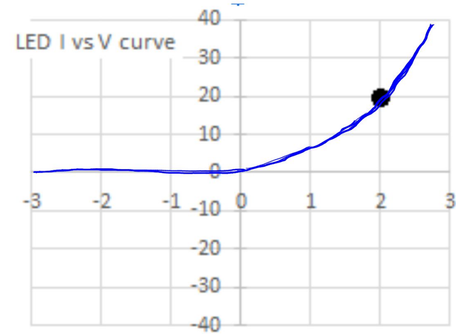
Signature:

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	10	
Problem 3	15	
Problem 4	15	
Problem 5	5	
Problem 6	10	
Problem 7	15	
Problem 8	15	
Total	100	

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



a

(i) (**3pt**) Consider an LED with a desired operating point of 2V, 20 mA. The 2V, 20mA point is shown as dot on the graph. Make a rough sketch of the LED current as a function of LED voltage. Include all voltages from -3 to +3V

For parts ii) iii) iv) consider this piece of C code

```
static uint16_t x=5;
uint16_t Operate(const uint16_t y){
  static uint16_t z=6;
```

For parts ii) iii) and iv) answer one letter A-H

A) Forces compiler to not optimize the access
B) Places the variable in nonvolatile ROM
C) Forces the variable to be placed in a register
D) Places the variable in volatile RAM
E) Makes the variable private to the file
F) Makes the variable private to the function
G) Restricts function from modifying parameter
H) Places the variable to be placed on the stack

(ii) (**2pt**) Why do we add the keyword `static` to the variable `x`?

E

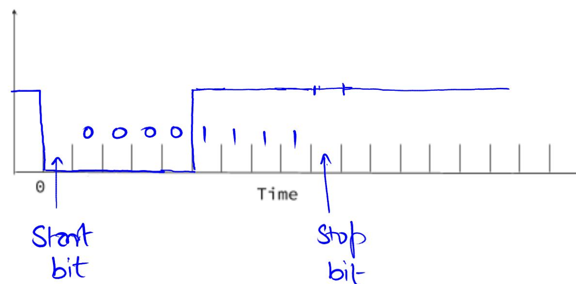
(iii) (**2pt**) Why do we add the keyword `const` to the y parameter?

G

(iv) (**2pt**) Why do we add the keyword `static` to the variable `z`?

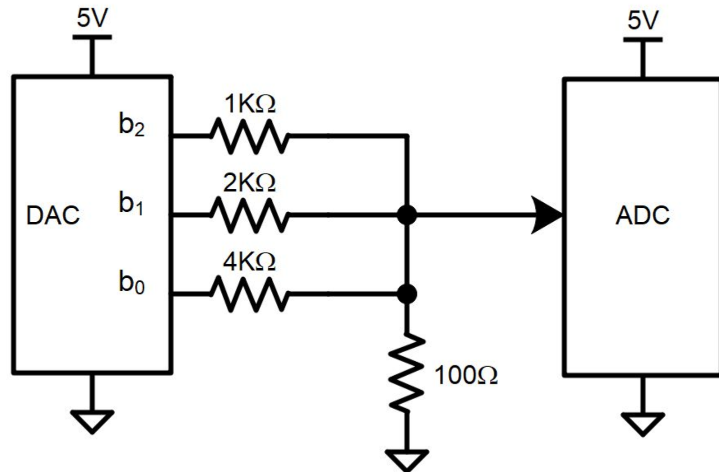
D

(v) (**6pt**) Draw the UART waveform for a frame when a byte **`0xf0`** is transmitted. Each bit time is one time slot on the plot. The frame starts at `Time=0` and goes right. Assume the UART is idle prior to this frame.



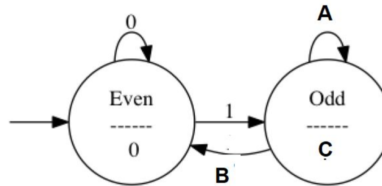
[10 points] Problem 2: DAC.
Given the 3-bit DAC with the 5V reference, provide the NUMERICAL EXPRESSION for the input voltage seen at the ADC input for the following DAC output codes with the 100 Ω load attached. The ADC reference is also 5V.

For each question below, a numerical answer is not required. Instead, provide a NUMERICAL EXPRESSION(you may leave it as unreduced) for your answer.



	DAC output code (binary) $b_2\ b_1\ b_0$	ADC input voltage expression (V)
(a)	0 0 0	0V
(b)	0 0 1	R1=4000ohms, R2=1000||2000||100=87ohms ADC = 5V*R2/(R1+R2) = 0.107V
(c)	0 1 0	R1=2000ohms, R2=1000||4000||100=89ohms ADC = 5V*R2/(R1+R2) = 0.213V
(d)	1 0 0	R1=1000ohms, R2=4000||2000||100=93ohms ADC = 5V*R2/(R1+R2)=0.425V
(e)	1 1 1	R1=1000||2000||4000=571ohms, R2=100ohms ADC = 5V*R2/(R1+R2)=0.745V

[15 points] Problem 3: Finite State Machine. The following state-machine is designed to be an Odd 1's detector. It detects if there are an odd number of 1's in a stream of 0's and 1's. It outputs a **1** when the stream thus far has an odd number of 1's, a **0** otherwise.



1. (3 points) What are the values of **A, B** and **C** in the FSM state-graph shown above.

A = 0	**B = 1**	**C = 1**

2. (7 points) The code below encodes this FSM in software. Fill in the missing pieces (marked ________).

```
#define Odd __1_____

#define Even ___0____
struct State {
   uint32_t out;

   uint32_t next[___2____];
}
typedef struct State State_t;
State_t FSM[2]={

{___0____,{Even,Odd____}},

{___1____,{Odd,Even____}}
};
```

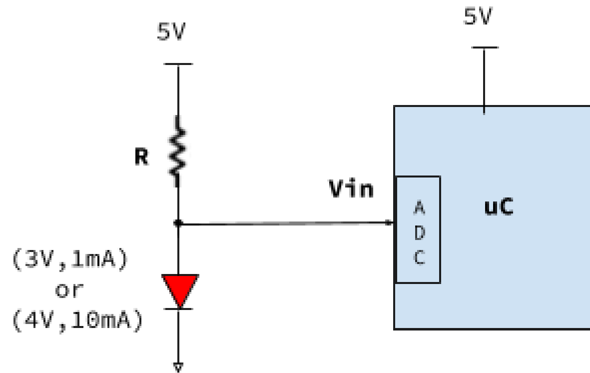
```
int main(){
   uint32_t CS=Even,in;
   while(1){
      doOutput(FSM[CS].out);
      in = getInput();
      CS = FSM[CS].next[in]; // XXX
   }
}
```

3. (5 points) Give the assembly code conversion of the one line of C code marked **XXX**, above. You may assume that CS and in are allocated in registers R1 and R2 respectively.

```
LDR R0,=FSM
MOV R3,#12   ; Size of the state struct is 12 bytes
MUL R1,R3
ADD R0,R1    ; R0 has FSM[CS]
ADD R0,#4    ; R0 has FSM[CS].next (offset to next field is 4)
LSL R2,#2    ; times 4: each element of next is 4 bytes
LDR R1,[R0,R2] ; new CS
```

[15 points] Problem 4: LED Interfacing and ADC

Valvano mixed up his two bags of LEDs. Now there is one bag with both types in it. You will help him sort them. The two possible LED types have operating points of either (3V,1mA) or (4V,10mA). A microcontroller interfaces to the circuit below, monitoring the voltage and computing the current. This allows you to determine which type of LED is connected. The ADC converter voltage range is [0V, 5V] (different from the TM4C range which is [0,3.3V]).



The LEDs do not need to light up, the test system must be able to distinguish one type from the other without destroying the LED. For each question below, a numerical answer is not required. Instead, provide a NUMERICAL EXPRESSION (you may leave it unreduced) for your answer.

(a) (5pts) Compute the resistor values for the two LED types.

R (ohms) = **5-4)/10mA = 100 ohms**

R (ohms) = **5-3)/1mA = 2000 ohms**

(b) (5pts) If you can choose only one of the above resistor values to distinguish between the two LED types, which is an appropriate choice and why?

If you use R=100 ohms, the 1mA LED will have I>>1mA
So, use R=2000 so low current LED is not damaged
I = (5-4)/2000 = 0.5mA (will not light up)
I= (5-3)/2000 = 1mA (will light up)

(c) (5pts) If using a 13-bit ADC converter, what is the ADC digital value when the current is 1mA? Show your work.

=1mA, R=2000, V_R=2V, V_{ADC}=3V, N = 3*8191/5 = 4915

[5 points] Problem 5: Write a C function that takes a null-terminated string as an input and outputs the string to the UART using busy-wait synchronization (assuming UART0 initialize is already done).

```
_void___UARTOutString(__char *string__){
 while(*string){ // stop on NULL, do not send NULL
  while((UART0_FR_R&0x20) != 0); // busy-wait on TXFF
  UART0_DR_R = *string;   // send one
  string++;     // next character in string
 }


}
```

[10 points] Problem 6: Translate the following C code to assembly. The local variable v6 must be allocated in the stack. Follow AAPCS guidelines for parameter-passing.

```
uint32_t var_test (uint32_t v1, uint32_t v2, uint32_t v3, uint32_t v4, uint32_t v5){
  uint8_t v6 = v5 * v4; // assembly code MUST store v6 on stack
  return v6;
 }
int main(void){
  var_test(11,12,13,14,15);
  while(1){};
}
```

```
V6 EQU  0   ; Binding
var_test:
  PUSH {R4,LR}
  LDR R4,[SP,#8] ;get v5 from stack
  SUB SP,#1   ; Allocate v6 on stack
;SUB ok if #1, #4, or #8 (AAPCS)
  MUL  R4,R3 ; R4 <- v5(R4) * v4(R3)
  STRB R4,[SP,#v6] ;v6=v5*v4
  LDRB R0,[SP,#v6]
  ADD  SP,#1 ;Deallocate v6
;ADD constant matches SUB constant
  POP {R4,LR}


 BX LR
```

```
main:
  MOV  R0,#15
  PUSH {R0}   ;pass v5
  MOV  R0,#11 ;pass v1
  MOV  R1,#12 ;pass v2
  MOV  R2,#13 ;pass v3
  MOV  R3,#14 ;pass v4
  BL   var_test ; Call subroutine
  ADD  SP,#4  ;discard v5
Loop
  B   Loop
```

[15 points] Problem 7: Queue and/or Stack.

You will complete the implementation of a data structure (called **QS**) that provides both a FIFO (*First In First Out like a **Queue***) and LIFO (*Last In First Out like a **Stack***) access mechanism. A wrap-around circular buffer (see figure below) is used to store the items being enqueued/dequeued and pushed/popped. A partially complete implementation of the data structure is given below with two missing subroutines (on next page) which you have to complete. You are not allowed to add any other variables to the implementation. Note that the same routine, QS_Add can be used to both enqueue an item and push an item to **QS**. The two routines you are to implement involve removing an item, when **QS** is treated as a Queue (QS_DeQ) and, when **QS** is treated as a Stack (QS_Pop).

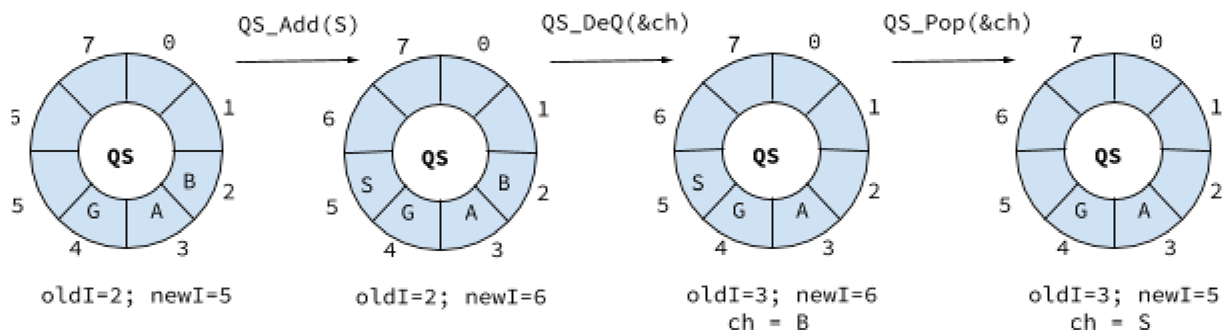
```
// Variables and Constants
#define N  8   // Capacity is 7
#define Fail  0
#define Success 1
char QS[N];     // Data store
// index of oldest item
uint8_t oldI;
// index of next item to be added
uint8_t newI;

// Initialize the QS, making it empty
// Input: None
// The QS is empty
void QS_Init(){
   oldI = newI = 0;
}
```

```
// Adds an item to QS
// Input: data has item to add
// Output: Success or Fail
uint8_t QS_Add(char data){
  if ((newI+1)%N == oldI)
     return Fail;
  QS[newI] = data;
  newI = (newI+1)%N;
  return Success;
}
```

The following figure shows how the three operations work. The leftmost figure shows the state of QS after some arbitrary Add, DeQ and Pop operations have been performed, leaving 3 items with oldest (B) at index 2 and newest (G) at index 4. The second figure from left shows the state of QS after adding an item (S). The third figure shows the state after an item(B) is de-queued, and the last shows the state after an item (S) is popped.

__Note__: The figure only shows valid items; inaccessible items need not be cleared on a dequeue/pop operation.



(**6 points**)

```
// De-queues oldest item from QS
// Input: Pointer to data to hold item removed
// Output: Success or Fail
uint8_t QS_DeQ(char *data){
  if (newI == oldI)
     return Fail;
  *data = QS[oldI];
  oldI = (oldI+1)%N;
  return Success;
}
```

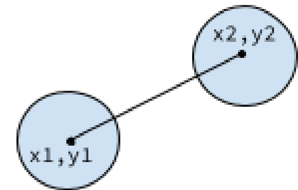
(**9 points**)

```
// Pops newest item from QS
// Input: Pointer to data to hold item removed
// Output: Success or Fail
uint8_t QS_Pop(char *data){
   if (oldI==newI) return Fail;
   if (newI==0) {
      *data = QS[Size-1];
      newI = Size-1;
   } else {
      *data = QS[newI-1];
      newI = newI-1;
   }
   return Success;
}
```

[15 points] Problem 8: programming, design.

Consider a game with birds flying in a 2-D world. The parameters of a bird are stored as fields in a structure. The parameters `x,y` are the center coordinates of the bird in 2-D space, which are the integer components of 32-bit signed binary fixed-point numbers. The resolution of the fixed-point number system is 1/256 meters. For example, if a position parameter is 2.125 meters, then the integer stored in memory is 2.125*256 = 544. You will write two functions, one to detect collision and one to process elastic bouncing. The movement occurs every 10ms (100 Hz), so the units of velocity are meters/10ms. We define a bird as a circle of radius 0.25 meters. A bird that is `dead` or `lost` will not move, will not be drawn, and will not be considered for collision detection. A bird that is `alive` will move, will be drawn, and will considered for collision detection. You may not add any fields to this structure. You may not add more global variables. You do not need to move or draw the birds.

```
typedef enum {dead,lost,alive} status_t;
typedef struct{
  int32_t x,y;       // 2-D position in 1/256 meter
  int32_t vx,vy;     // 2-D velocity in 1/256 meter/10ms
  const int16_t *image; // pointer image to draw
  status_t life;     // dead/alive
} Bird_t;
```



There are 100 birds in the game, defined in global RAM like this

```
Bird_t Flock[100];
```

The first function you will write will determine if two birds have collided. For this function, you assume both birds are alive. A collision is defined if the distance from the center of one bird to the center of the other bird is less than or equal to 0.5 meters. Think about how to do this without using square root; however, partial credit will be given if you call an existing integer **sqrt()** function. No credit will be given for using floating point. The prototype for your first function is

```
// Input: pointers to two birds
// Output: true if these two birds have collided, false if not collided
int Collision(Bird_t *p1, Bird_t *p2);
```

The second function you will traverse the array of 100 birds, and check if any two birds have collided. A collision event can occur between any two birds that are both **alive**. Consider birds 1 and 2 with masses m_1, m_2, and velocities u_1, u_2 before collision, v_1, v_2 after collision. An elastic collision in physics conserves both momentum (m*v) and kinetic energy ($0.5*m*v^2$). In an elastic collision,

$$v_1 = (m_1 - m_2)*u_1/(m_1+m_2) + 2*u_2*m_2/(m_1+m_2)$$

$$v_2 = (m_2 - m_1)*u_2/(m_1+m_2) + 2*u_1*m_1/(m_1+m_2)$$

If the two birds have the same mass ($m_1=m_2$), this simplifies to swapping the velocities

$$v_1 = u_2$$

$$v_2 = u_1$$

You will implement elastic collisions in both x and y dimensions, More specifically, if a collision occurs, swap the `vx` between the two birds, and swap the `vy` between the two birds. You must call the `Collision` function in part a) to determine if a collision has occurred. Be careful not to swap the velocities twice. E.g., if bird 12 has collided with bird 37, then bird 37 will also have collided with bird 12. On a collision, please change the `vx vy` of both birds once and not twice. You do not have to specifically handle the case with 3 or more birds colliding at the same time. The prototype for your second function is

```
void ElasticBounce(void);
```

You may use this swap function if you wish

```
void swap(int32_t *pt1, int32_t *pt2){int32_t data;
  data = *pt1;  *pt1 = *pt2;   *pt2 = data;
}
```

(7) Part a) Write the collision function in C

```
int Collision(Bird_t *p1, Bird_t *p2){
// 0.5m is 0.5*256=128, 0.5*0.5m^2 is 128*128=16384
  int32_t dx,dy,dd;
  dx = (p1->x - p2->x);   // difference in x
  dy = (p1->y - p2->y);   // difference in y
  dd = dx*dx+dy*dy;       // distance squared
  return dd <= 16384;
}
```

(8) Part b) Write the bounce function in C

```
void ElasticBounce(void){
  int i,j; // indices into flock
  for(i=0; i<99; i++){
    if(Flock[i].life == alive){
       for(j=i+1; i<100; i++){
         if(Flock[j].life == alive){
      if(Collision(&Flock[i],&Flock[j]){
             swap(&Flock[i].vx,&Flock[j].vx; // swap velocities
             swap(&Flock[i].vy,&Flock[j].vy; // swap velocities
          }
         }
       }
    }
  }


}
```