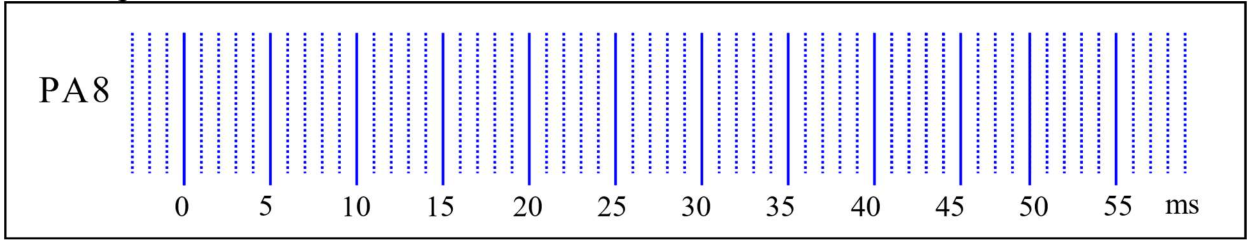
UT EID: ______________________ First: _______________ Last: ______________________

Instructions:

- Closed book and closed notes. No books, no papers, no data sheets (other than the addendum)
- 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. *Anything outside the boxes/blanks will be ignored in grading*. You may use the back of the sheets for scratch work.
- You have 120 minutes, so allocate your time accordingly.
- For all questions, unless otherwise stated, find the most efficient (time, resources) solution.
- Unless otherwise stated, make all I/O accesses friendly and all subroutines AAPCS compliant.
- *Please read the entire exam before starting.*

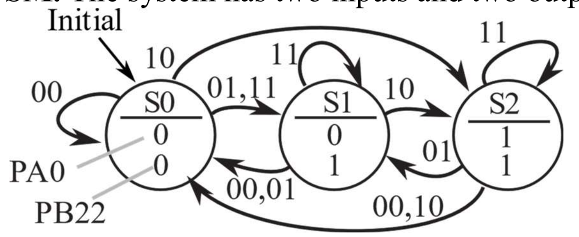
(10) Question 1. Communications/UART. The baud rate is 250 bits/sec. The UART1 output uses PA8. The microcontroller outputs one frame with data equal to 0x32 (binary 0b00110010, ASCII '2'). Draw the PA8 output signal as a function of time for this one frame. Assume the frame begins at time = 0. Assume the UART is idle before and after this one frame.



(10) Question 2. Friendly GPIO access. There is a 3-bit DAC connected to PB2-0. You can assume PB2-0 are already initialized as outputs. Implement friendly functions that output to the DAC in both C and assembly. Assume input parameter is a value limited from 0 to 7.

`// R0 has 3-bit data`	`void DAC3_Out(uint32_t data){`
`DAC3_Out:`	

(10) Question 3. Moore FSM. The system has two inputs and two outputs.



```
#define S0 0
#define S1 1
#define S2 2
```

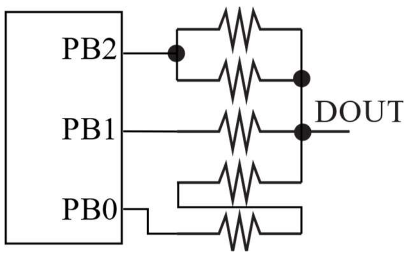
Part a) Write C code to define a struct for this FSM. Each state has an output value for PA0 and a separate output value for PB22. Each state also has four next states, but no time delay.

```
typedef const state state_t; // do not change any code given to you
```

Part b) Complete C code to define the state transition table in ROM. No engine is required.

```
state_t FSM[3]={ // parts a and b must compile together
  {0,0,{S0,S1,S2,S1},
  {0,1,{S0,S0,S2,S1},
```

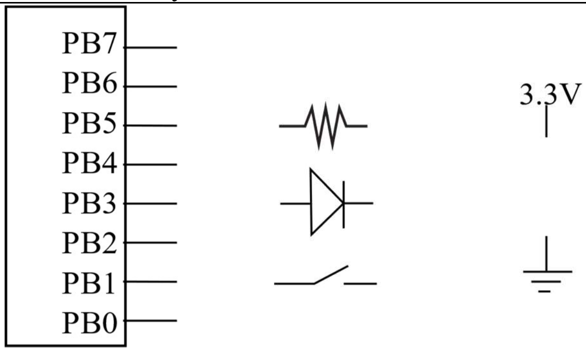
(10) Question 4. DAC, Ohm's Law, KCL, KVL. All resistors are 10k. Assume V_{OH} is 7V, V_{OL} is 0V, and PB2-0 are outputs. What is the DOUT voltage if the software writes a 100_2 to Port B.



(10) Question 5. LED. You are given this function, which should turn on an LED,

```
void LED_On(){
  GPIOB->DOUT31_0 &= ~0x04;
}
```

and are asked to interface an LED to the microcontroller, so the software operates as intended. The LED parameters are $I_d = 1.8\text{mA}$, $V_d = 1\text{V}$. The microcontroller output voltages are $V_{OL}=0.5\text{V}$ and $V_{OH} = 3.2\text{V}$. Show the LED interface that makes this software work. Include math to determine any resistor values needed.



(10) Question 6) FIFO queue. There is exactly one line in **Get** that must be changed. Circle the line containing the bug in **Get**. Show the correction required so it operates correctly. *Hint*: execute two calls to **Put** and draw the resulting data structure. Then, execute two calls to **Get** to see if your correction fixes the bug.

```
uint32_t PutI;  // should be 0 to 7
uint32_t GetI;  // should be 0 to 7
int32_t static FIFO[8];
void Init(void){
  PutI = GetI = 0;
}
int Put(int32_t data){
  if(((PutI-1)&7) == GetI) return 0;
  FIFO[PutI] = data;
  PutI = (PutI-1)&7;
  return 1;
}
int Get(int32_t *datapt){

  if(PutI == GetI) return 0;

  *datapt = FIFO[GetI];

  GetI = (GetI+1)&7;

  return 1;

}
```

(10) Question 7. Local variables. The subroutine **mySub** has one call by value input parameter and one output parameter. The function must be AAPCS compliant. The C version is

```
uint32_t mySub(uint32_t x){ uint32_t z=10; return z*x);
```

Typical calling sequences are

```
    LDR  R0,=1000                     uint32_t y;
    BL   mySub                        y = mySub(1000);
    MOVS R4,R0 // set y
```

The input parameter **x** is passed in R0, but will be saved as a local on the stack. The subroutine allocates one 32-bit local variable **z,** and it uses R7 frame pointer addressing to access the locals. The binding for these two locals are

```
.equ x, [          ]  // binding for the input parameter x

.equ z, [          ]  // binding for 32-bit local variable z

mySub: PUSH {R7,LR}
       PUSH {R0}      // parameter x is saved on the stack

     [                    ] // allocate z

     MOV R7,SP         // establish frame pointer
//---------start of body------------------
     MOVS R0,#10

     [                    ] // set z = 10, using R7

    LDR  R2,[R7,#x]  // R2 is input parameter x (1000)
    LDR  R3,[R7,#z]  // R3 is z (10)
    MULS R2,R2,R3    // R2 is z*x (10000)
//---------end of body--------------------
// balance the stack and return z*x

    [                    ]
```

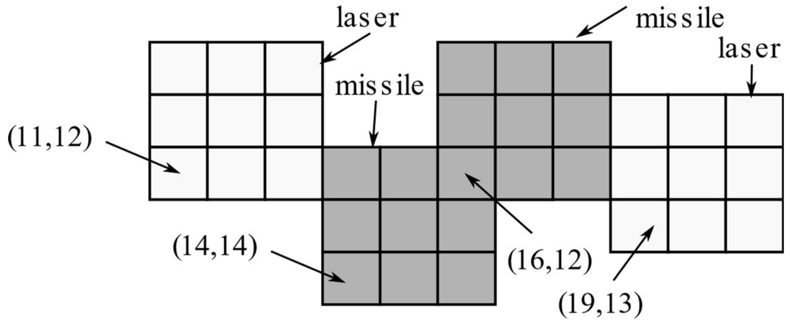
Fill in the above boxes with one or more lines of assembly code.

(15) Question 8. Arrays in assembly. Translate this C to assembly (assume I is initialized to 0)

```
uint16_t Buff[100];

uint32_t I;



void Dump(uint16_t x){
  if(I < 100){
    Buff[I] = x;
    I++;
  }
}
```

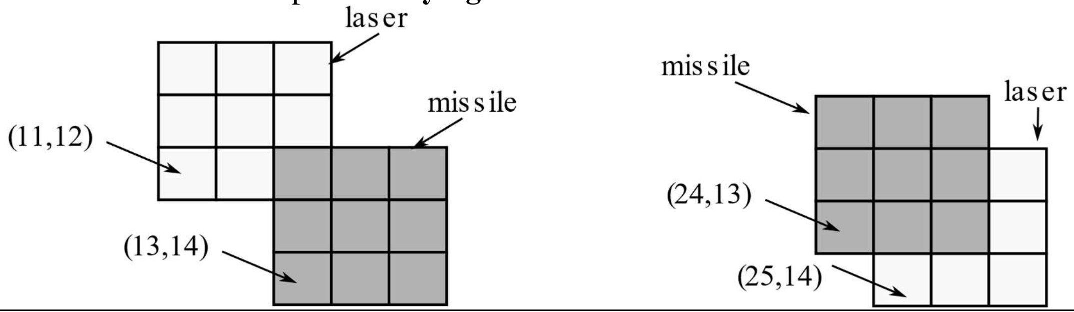
```
  .data




  .text
Dump:
```

(15) Question 9. Collisions. Consider a game with 10 missiles and 20 lasers. There are two sprite arrays, **Missiles** and **Lasers**. Consider each sprite as a 3 by 3-pixel square. The (x,y) coordinate of a sprite is its lower left pixel. You may assume the arrays have been populated with data before your function is called. A **collision** is defined as the overlap of any pixel of a missile with any pixel of a laser. This first figure has no collisions. Ignore collisions from missile to missile, and ignore collisions from laser to laser.



```
typedef enum {dead,alive,dying} status_t;
struct sprite{
  int32_t x;        // x coordinate, in pixels
  int32_t y;        // y coordinate, in pixels
  int32_t vx;       // x velocity, in pixels/frame
  int32_t vy;       // y velocity, in pixels/frame
  status_t life;};  // dead or alive
typedef sprite sprite_t;
sprite_t Missiles[10];
sprite_t Lasers[20];
```

Implement a C function that searches for collisions. If an **alive** missile overlaps with an **alive** laser, set both life parameters to **dying**. Do not worry about collisions involving 3 or more sprites touching at the same time. This second figure shows two collisions. Your function should set the **life** parameters for these four sprites to **dying**.



```
void Collisions(void){
```