EEE095/6S 2022 Class Test 2

11 October 2022 [100] 1.5 hours

Instructions:

Please **use the attached answer sheet** for answering the questions. ***NB:*** write your student number and name on the first page in the box at the top of the answer sheet. The first page of the answer sheet is for answering multiple choice questions (Section A), Yes/No questions (Section B) and the second page onwards is for answering the short answer questions (Section C).
Note: The appendix provides cheat sheets for ARM assembly and Verilog.

Section 1: Multiple Choice [35 marks]

Q1.1

Assume you have two embedded platforms, A and B, both of which have SPI and GPIO built in. If you wanted to connect these two quickly to running high-speed comms with minimal additional hardware, which one of these options would you recommend? (select only one option)

a) Connect A and B via I2C
b) Connect A and B via SPI
c) Connect A and B via USB
d) Connect A and B via RS232

Q1.2

Consider you are connecting platforms via I2C. Ponder these points and select which one of these is most accurate. (select only one option).

a) I2C utilizes more wires, yet is still slower than SPI.
b) I2C is superior to RS232 as it provides full-duplex data transfer
c) I2C has benefit over SPI as its data line can be shared among multiple masters and slaves.
d) I2C saves interface lines, similarly to RS232, as it eliminates the need for a shared clock.

Q1.3

While there are multiple benefits to I2C, which one of these is a key disadvantage? (select only one option).

a) I2C utilizes more wires, yet is still slower than SPI.
b) I2C does not provide easy supports for multiple Vdd levels.
c) I2C is full duplex, but this causes complication in scheduling acknowledgements.
d) I2C frame overhead costs time that can cause reduced comms efficiency.

Q1.4

Consider the differences between CISC and RISC. Which one of the statements below is false concerning CISC instructions? (select only one option)

a) It has a large number of instructions (usually 100 or more instructions).
b) All instructions are of the same length.
c) Some instructions perform specialized tasks and are used only infrequently.
d) Often has instructions, besides load and store, that can manipulate memory directly.

Q1.5

A so-called “cheap and nasty” DAC simply uses PWM, usually via an amplifier and small capacitor, to generate analogue output signals. But a major drawback of this approach is… (select one option)

a) That the output voltage cannot change very rapidly, it increments or decrements over time.
b) That it requires intense use of the DMA resources.
c) It hogs a GPIO pin for doing the PWM.
d) It is limited to the voltage ranges that the GPIO has (usually 0V to 3.3V).

Q1.6

The following circuit illustrates a rather useful and low cost DAC design… which happens to be one of your lecturer’s favourite designs as it is fairly simple and effective. What is this useful DAC design approach called … (select one option)

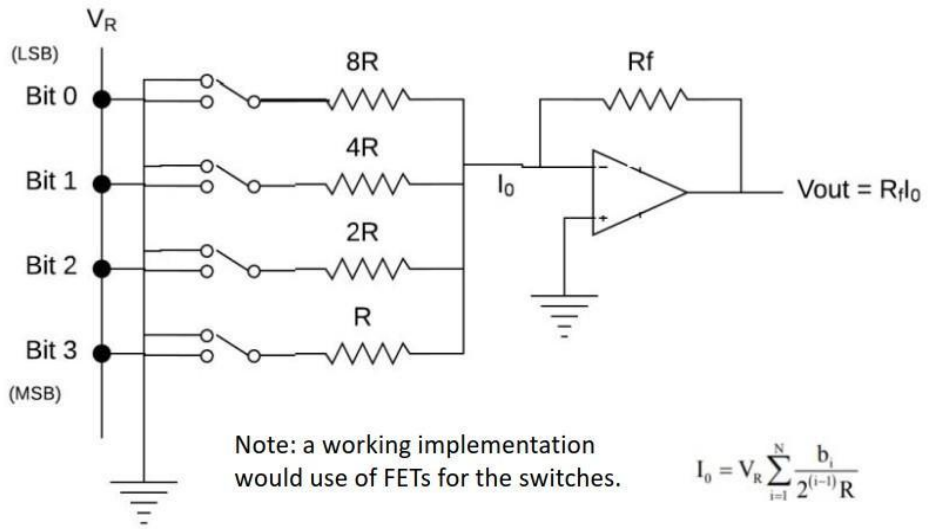


Figure 1: A much liked DAC... but can you remember its name which captures the main feature of its design?

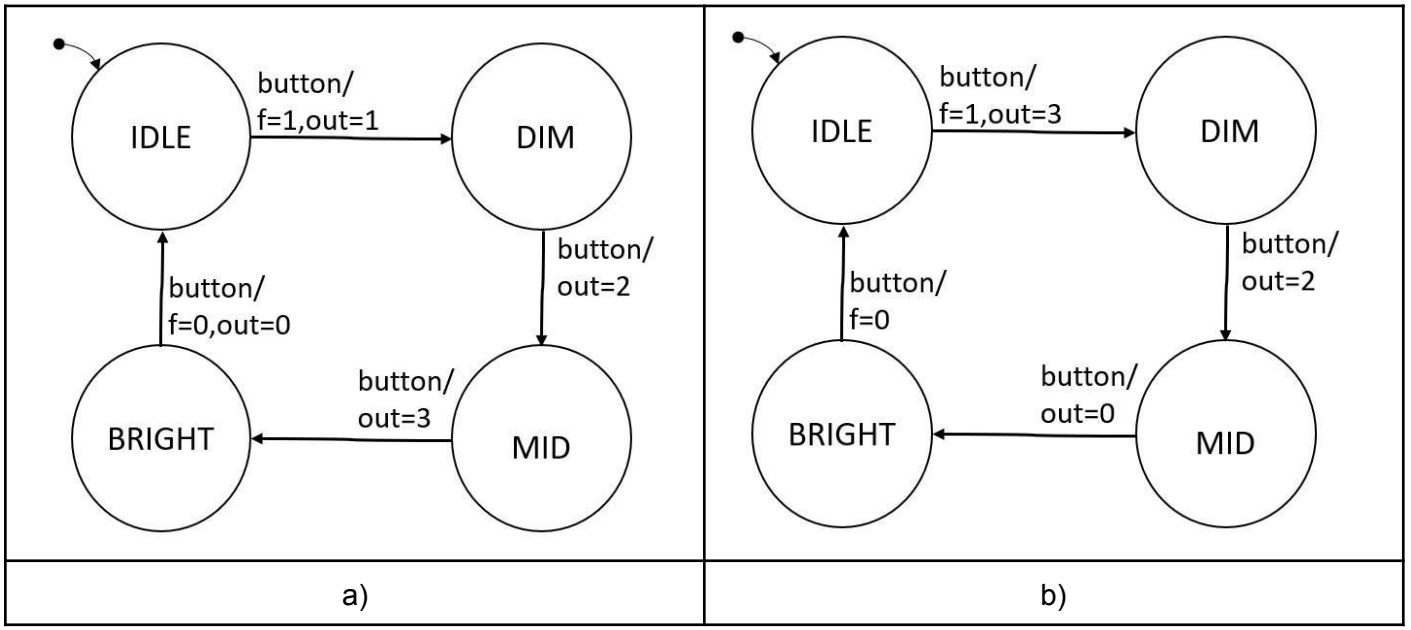
a) The R2R Ladder DAC.
b) The Parallel Resistors DAC.
c) The Convergent Binary Search DAC.
d) The Binary Weighted DAC.

Q1.7

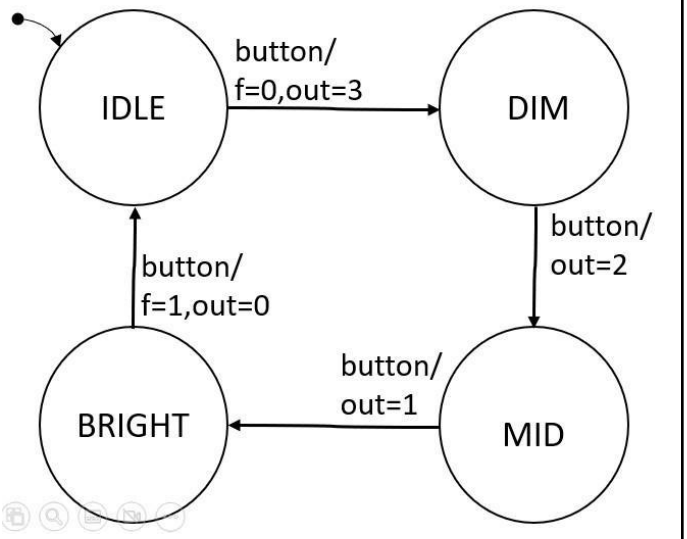
Consider that you need to provide a finite state machine diagram for the following flashlight system, that works as follows:

- The flashlight has one button, the ‘button’ is an input, and a 2-bit output pin, called ‘out’.
- The out line is used to connect one resistor (if out=1 or out=2), two resistors (if out=3) or no resistors (if out=0, i.e. a direct connection) in series with the LED (the resistors are all the same resistance).
- The ‘f’ line turns on power to the LED that passes through zero or more resistors.
- In idle mode (which is the state it starts in) the flashlight is just waiting for the button to be pressed.
- The first time the button is pressed, the LED of the flashlight is set to the dim state.
- The second time the button is pressed, the LED is at medium brightness, the ‘mid’ state.
- The third time the button is pressed, the LED is at maximum brightness, the ‘bright’ state.
- The fourth time the button is pressed, the LED is turned off and the idle mode is resumed.

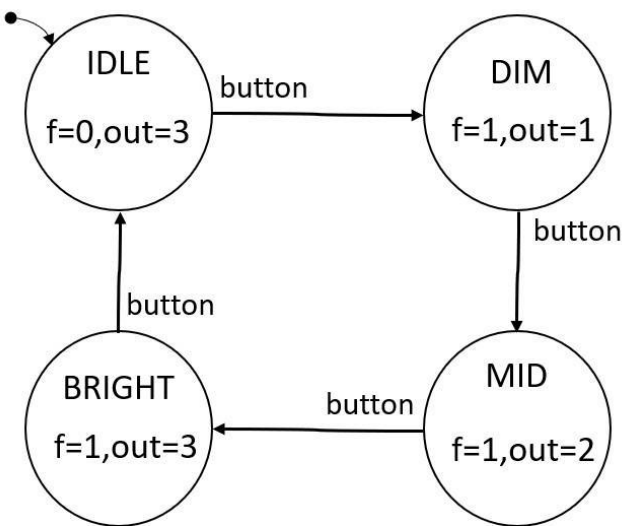
Which one of the following drawings gives the most correct representation for a Mealie machine that describes the flashlight design as mentioned above (select only one of the options)



a) b)



c)



d)

Section 2: Yes or No Answers [25 marks]

Q2.1
Lecture 1 went into various Embedded Systems definitions. Of these, the 'Brisbane' definition was indicated to be particularly apt and particilarly favoured by your lecturer. Is the below definition the correct Brisbane definition? (Answer Yes or No)

"Brisbane, Australia (2009) Definition":
Embedded systems are information processing systems within a larger system, for which the technical problem centres on managing time and concurrency of computation.

Q2.2
The teams that develop *complex* embedded systems typically include domain experts. Would it be correct to say that the domain expert is someone who knows little about the environment or location in which the system to be developed will be installed, but does know much detail about how electronic systems are built and configured? (Yes or No)

Q2.3
We have discussed SPI quite a bit in this course in regards to communication protocols, but we seldom mention the full name for what this acronym stands for… Does SPI stand for "Serial Programming Interface"? (Yes or No)

Q2.4
The term UART refers to the a standard serial protocol that is nowadays commonly used for asynchronous data transfer between two platforms of the same type.

Q2.5
Assume the following code snippet is valid Veilog. When run in simulation, the value of 'd' that is printed will be 0, as opposed to x. (Yes or No)

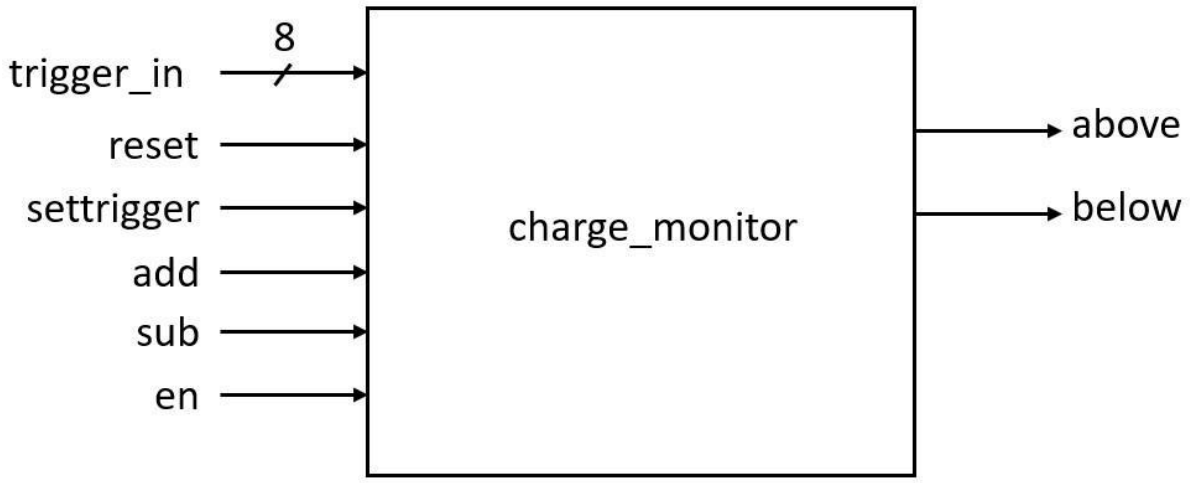
```
// combine does some logical function on input pins
module combiner (input a, b, c, output d);
 assign d = ~((a & b) | c);
endmodule

// this is the testbench for seeing what combiner does
module combiner_tb ();
 reg a, b, c;  // define some register to hold values
 wire d;       // this just links to the output pin of combiner

 // instantiate the module udner test
 combiner mtb(a,b,c,d);

 initial
  begin
   // link monitor simulation operation to see how pins change
   $monitor("%b %b %b -> %b\n",a,b,c,d);
   a = 0;
   #5 a = 1;
   #5 b = 0;
   #5 c = 1;
  end
endmodule
```

Section 3: Short Answer Questions [40 marks]

The answer sheet provides answer blocks in which you are to provide answers for these questions; note that the answesheets will be scanned into Gradescope, so please try to keep your answers on the pages even if they don't fit fully into the boxed space procided.

Q3.1 Verilog Coding [25 marks]

Consider the following block diagram. The C++ code on the next page (which I've converted to be mostly C code to improve understanding) indicates what this module needs to do.



The basic concept is a module that keeps track of the number of Wh that a battery had been charged with. If a positive edge is sent on the add line, this means 1Wh of charge has been added. If a positive edge is sent on the sub line, it means a 1Wh of charge has been used. The system can only keep track of discrete

1Wh units, it cannot do fractions (but don't worry about this approximation or reasoning for this). Basically, the module has an internal 8-bit register that is called 'level' and works like a counter that is incremented or decremented to keep track of the Wh of charge available. Furthermore the module can be set up with a trigger level (pass to it via trigger_in and sending a positive edge on the settrigger line). If the current charge level (i.e. the value of the 'level' register) falls under the trigger, the below output line is set high; and if the charge level goes above the trigger level then the above line is set high.

Transform the C code into a Verilog module. You only need to implement the charge_monitor module, not the surrounding test code given in the main() function (the main function is just code to test the module and show its operation). Assume datetype **bool** in C is equivalent to **bit** in Verilog. A printout of what the code does is shown after the code.

Notes on marking

- Module interface declaration : 5 marks
- Handling setting of trigger : 5 marks
- Handling positive edges on add and on sub : 2x4 = 8 marks
- Accounting for reset and en lines: 4 marks
- Comments & neatness: 3 marks

```
/** Change_monitor operation : a C program showing how the charge_monitor module needs to work. */
#include <iostream>
#include <stdio.h>
using namespace std;

typedef unsigned char BYTE; /* just defining what a BYTE is in C */

/** monitor: This function is just added to the C program to print out the signal lines, it is essentially
mimicking what a $monitor function linked to these signal lines would likely show. */
void monitor (BYTE  trigger_in, bool reset, bool settrigger, bool add, bool sub, bool en,
                bool above, bool below)   {
   static int t = 0;
   printf("#%02d trigger = %d reset=%d settrigger=%d add=%d sub=%d en=%d above=%d below=%d\n",
        t,trigger_in, reset, settrigger, add, sub, en, above, below);
   t++; // t is just used to keep track of time, equivalent to number of times monitor is called in this case.
}

/** This is the function that needs to be turned into a Verilog module. Assume that all the parameters
    that are of datatype bool would be a bit in Verilog. */
void charge_monitor (
       BYTE  trigger_in, // a value we want to be alerted to if we are under or over this charge level
       bool  reset,        // hold this high, regardless of enable, to set level to 0
       bool  settrigger,  // send a positive edge if we want to set the trigger value
       bool  add,          // add a Watt of power to battery
       bool  sub,          // indicate used a Watt of power from battery
       bool  en,           // enable line (when high module is active)
       bool* above,      // variable parameter, set high if above trigger
                           // (since you might not all know C++, and & params, I'm instead using C here)
       bool* below       // this is a variable parameter to say level is below trigger
    )
{
```

```
// NB: if you don't know what a static keyword is, basically treat it like a global variable.
static BYTE level;    // level: stores the value of the charge level
static BYTE trigger; // need to keep an internal copy of what trigger value was sent
// these variables are just used for capturing the previous values of input values, so that a
// posedge or negedge can be detected on the signal lines.
static bool prev_settrigger, prev_add, prev_sub;

//  DEBUG: this call to monitor is just to print out the input and output lines //
monitor (trigger,reset,settrigger,add,sub,en,*above,*below);

// --- this is the start of the operations that the module is to provide: --- //

// check if reset is high
if (reset) {
   level = 0;
   prev_settrigger = 0;
   prev_add        = 0;
   prev_sub        = 0;
} else {
   // if reset is low....

   if (en) {
      // if reset is low and en is high

      // check if there is a positive edge on settrigger, and if so then
      // set the module's trigger register to the sent trigger input value
      if (settrigger & !prev_settrigger) trigger = trigger_in;

      // if there is a positive edge on add, increment the level
      if (add        & !prev_add)        level++;

      // if there is a positive edge on sub, decrement the level
      if (sub        & !prev_sub)        level--;

      // check if trigger conditions are met:

      // if level is below trigger level then set below to high
      if (level < trigger) *below = 1; else *below = 0;

      // if level is above trigger level then set above to high
      if (level > trigger) *above = 1; else *above = 0;
   } // end of if (en)

   // save the state of the inputs in the prev_ variables so that we
   // can tell in C if the input has a positive (0 to 1) or negative (1 to 0) edge
   // you can essentially ignore these three lines for Verilog coding
   prev_settrigger = settrigger;
   prev_add        = add;
   prev_sub        = sub;
} // end of else
```

```
} // end of module

// ---------- MAIN FUNCTION ----------
/** Main function: entry point to this program
    NB: you don't need to implement this in Verilog, this main function
    Is simply provided as an example of how to test the check_module C function
    that is mimicking the operation of the wanted monitor_charge module. */
int main()
{
   cout << "Welcome to charge monitor!" << endl;

   // let's test the operation and demonstrate what the function  is doing

   // define some variables (would be regs in Verilog)
   BYTE  trigger    = 0;
   bool  reset      = 0,    settrigger = 0,    add        = 0,   sub        = 0,
         en         = 0,    above      = 0,    below      = 0;
   int   i; // just used for looping

   // print out value of the lines
   monitor (trigger,reset,settrigger,add,sub,en,above,below);

   // lets run some test vectors!!
   // assume that monitor calls here are mimiching the monitor function in verilog

   // reset the module
   reset = 1;
   charge_monitor (trigger,reset,settrigger,add,sub,en,&above,&below);

   // enable the module
   reset = 0;
   en    = 1;
   charge_monitor (trigger,reset,settrigger,add,sub,en,&above,&below);

   // assign the trigger value
   trigger    = 5;
   settrigger = 1;
   charge_monitor (trigger,reset,settrigger,add,sub,en,&above,&below);

   // take settrigger back to 0
   settrigger = 0;
   charge_monitor (trigger,reset,settrigger,add,sub,en,&above,&below);

   // add in 7Wh of power
   for (i=0; i<7; i++) {
      add        = 1;
      charge_monitor (trigger,reset,settrigger,add,sub,en,&above,&below);
      add        = 0;
      charge_monitor (trigger,reset,settrigger,add,sub,en,&above,&below);
   }
```

```
    // end simulation
    cout << "Finished" << endl;

    return 0;
}
```

Result of running this code

```
Welcome to charge monitor!
#00 trigger = 0 reset=0 settrigger=0 add=0 sub=0 en=0 above=0 below=0
#01 trigger = 0 reset=1 settrigger=0 add=0 sub=0 en=0 above=0 below=0
#02 trigger = 0 reset=0 settrigger=0 add=0 sub=0 en=1 above=0 below=0
#03 trigger = 0 reset=0 settrigger=1 add=0 sub=0 en=1 above=0 below=0
#04 trigger = 5 reset=0 settrigger=0 add=0 sub=0 en=1 above=0 below=1
#05 trigger = 5 reset=0 settrigger=0 add=1 sub=0 en=1 above=0 below=1
#06 trigger = 5 reset=0 settrigger=0 add=0 sub=0 en=1 above=0 below=1
#07 trigger = 5 reset=0 settrigger=0 add=1 sub=0 en=1 above=0 below=1
#08 trigger = 5 reset=0 settrigger=0 add=0 sub=0 en=1 above=0 below=1
#09 trigger = 5 reset=0 settrigger=0 add=1 sub=0 en=1 above=0 below=1
#10 trigger = 5 reset=0 settrigger=0 add=0 sub=0 en=1 above=0 below=1
#11 trigger = 5 reset=0 settrigger=0 add=1 sub=0 en=1 above=0 below=1
#12 trigger = 5 reset=0 settrigger=0 add=0 sub=0 en=1 above=0 below=1
#13 trigger = 5 reset=0 settrigger=0 add=1 sub=0 en=1 above=0 below=1
#14 trigger = 5 reset=0 settrigger=0 add=0 sub=0 en=1 above=0 below=0
#15 trigger = 5 reset=0 settrigger=0 add=1 sub=0 en=1 above=0 below=0
#16 trigger = 5 reset=0 settrigger=0 add=0 sub=0 en=1 above=1 below=0
#17 trigger = 5 reset=0 settrigger=0 add=1 sub=0 en=1 above=1 below=0
#18 trigger = 5 reset=0 settrigger=0 add=0 sub=0 en=1 above=1 below=0
Finished
```

Q3.2 Verilog Coding [15 marks]

Some ARM programming… but, since it's the last question, something pretty quick and easy 😉

Consider that we want to convert the following *squared_equal* simple function into ARM assembly that can be assembled using GAS (full marks is you are appropriately using the C ABI). This routine just check if the square of the first input parameter, X, is equal to the second input parameter Y. Accordingly, these two parameters are somehow sent via registers. Try to provide an assembly solution that would be as close to workable as possible.

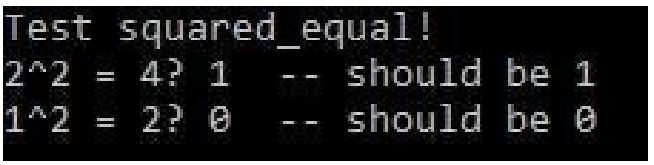
```
/** squared_equal function that returns true if input X squared is equal to input Y */
int squared_equal ( int X, int Y )
{
    int sqX = X * X;        // work out the square of X
    if (sqX == Y) return 1;  // they are equal
    return 0;               // they are not equal
}
```

Convert just this function to assembly

Example of how the function might get used… (this is just to aid understanding of the function)

```
/** main function for testing the squared_equal function */
int main()
{
    int res;
    printf("Test squared_equal!\n");

    // test 2,4 ... should be equal
    res = squared_equal(2,4);
    printf("2^2 = 4? %d  -- should be 1\n",res);

    // test 1,2 ... should be unequal
    res = squared_equal(1,2);
    printf("1^2 = 2? %d  -- should be 0\n",res);

    return 0;
}
```

Console output when above program executes



end of test