EET 325
Microprocessors

Drexel University
Engineering Technology

Lab #2 – Adding memory
(and modular design in IceStudio)

Objectives

The objectives of this lab exercise are to demonstrate how existing Verilog components can be made into reusable modules and combined with new code to create more complex designs; and to introduce the usage of on-chip SRAM memory blocks for reading and writing, including power-on defaults.

Introduction

In Lab 2, we will build on the four-digit seven-segment display that we produced in Lab 1.
First, we will make a more general four-digit display driver, which we will then use with the input DIP switches to build a memory. We will then add an address buffer, standing in for a program counter. Finally, we will include write functionality.

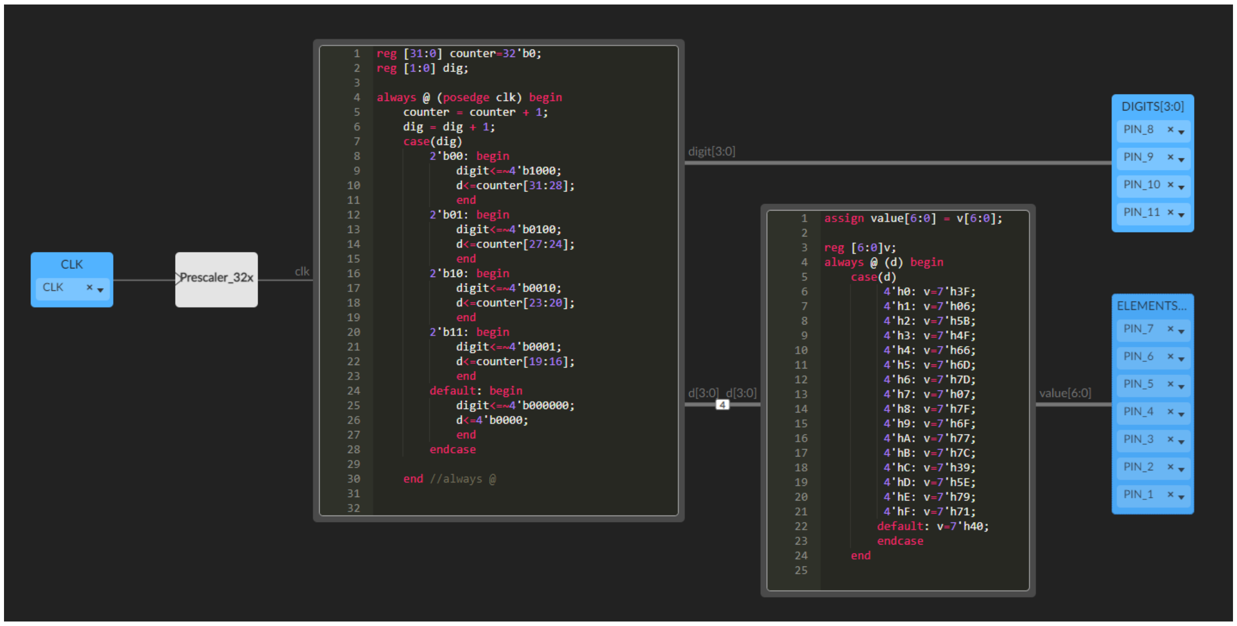
Setup:

Open ICEStudio on your computer. Plug your TinyFPGA BX into the breadboard, if it isn't already. Load up your final project from Lab 1 and use Save As to make a copy of it as Lab 2.

(If you don't have a copy of your old code, you can use mine, but you'll probably have to change the pinouts to match your design…)

Exercise 1: Turning a code block into a module (for hierarchical design)

At the end of Lab 1, you should have something like this:



```verilog
reg [31:0] counter=32'b0;
reg [1:0] dig;

always @ (posedge clk) begin
    counter = counter + 1;
    dig = dig + 1;
    case(dig)
        2'b00: begin
            digit<=~4'b1000;
            d<=counter[31:28];
            end
        2'b01: begin
            digit<=~4'b0100;
            d<=counter[27:24];
            end
        2'b10: begin
            digit<=~4'b0010;
            d<=counter[23:20];
            end
        2'b11: begin
            digit<=~4'b0001;
            d<=counter[19:16];
            end
        default: begin
            digit<=~4'b000000;
            d<=4'b0000;
            end
        endcase

    end //always @
```

```verilog
assign value[6:0] = v[6:0];

reg [6:0]v;
always @ (d) begin
    case(d)
            4'h0: v=7'h3F;
            4'h1: v=7'h06;
            4'h2: v=7'h5B;
            4'h3: v=7'h4F;
            4'h4: v=7'h66;
            4'h5: v=7'h6D;
            4'h6: v=7'h7D;
            4'h7: v=7'h07;
            4'h8: v=7'h7F;
            4'h9: v=7'h6F;
            4'hA: v=7'h77;
            4'hB: v=7'h7C;
            4'hC: v=7'h39;
            4'hD: v=7'h5E;
            4'hE: v=7'h79;
            4'hF: v=7'h71;
        default: v=7'h40;
        endcase
    end
```

Figure 1. The final product from Lab 1: a four-digit counter

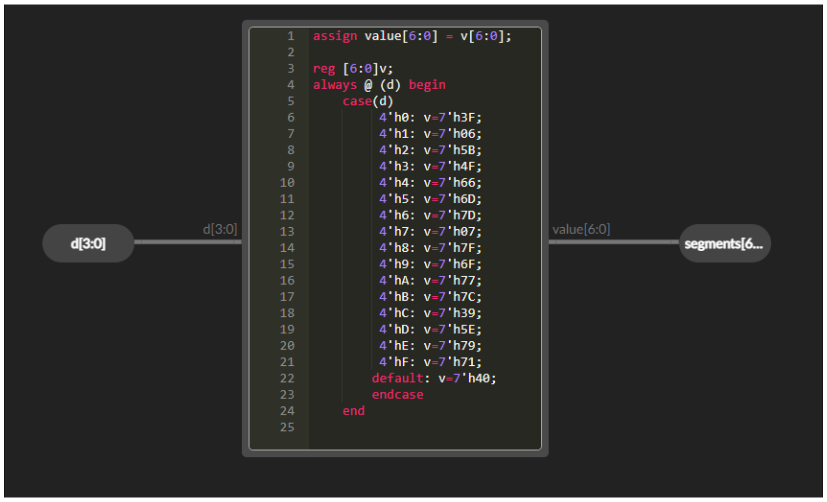
First, let's condense the seven-segment decoder into a module, since we've tested it and know that it works. (It's a four-input combinational circuit, and the counter project in Lab 1 runs through all 16 possibilities, so if you saw it count from 0 to F properly, it's been tested well enough for our purposes.)

Select the block containing the Verilog code for the decoder (the right code block).
Once it is selected (shaded blue), press ctrl-C to copy it.

Open a new Verilog project (while keeping Lab 2 open). Click on the workspace in this new project and press ctrl-V to paste in a copy of the code block.

Add input block **d[3:0]** and output block **segments[6:0]**. →→**Uncheck the "FPGA Pin" option.** ←←
(*If your seven-segment display block includes the decimal point, use* ***segments[7:0]*** *instead.)*

Your new project should look like Figure 2, below. Save it as "sseg_decoder.ice"
The input and output blocks should be grey. If they are blue, go back and uncheck FPGA pin.



```
assign value[6:0] = v[6:0];

reg [6:0]v;
always @ (d) begin
    case(d)
            4'h0: v=7'h3F;
            4'h1: v=7'h06;
            4'h2: v=7'h5B;
            4'h3: v=7'h4F;
            4'h4: v=7'h66;
            4'h5: v=7'h6D;
            4'h6: v=7'h7D;
            4'h7: v=7'h07;
            4'h8: v=7'h7F;
            4'h9: v=7'h6F;
            4'hA: v=7'h77;
            4'hB: v=7'h7C;
            4'hC: v=7'h39;
            4'hD: v=7'h5E;
            4'hE: v=7'h79;
            4'hF: v=7'h71;
        default: v=7'h40;
        endcase
    end

```

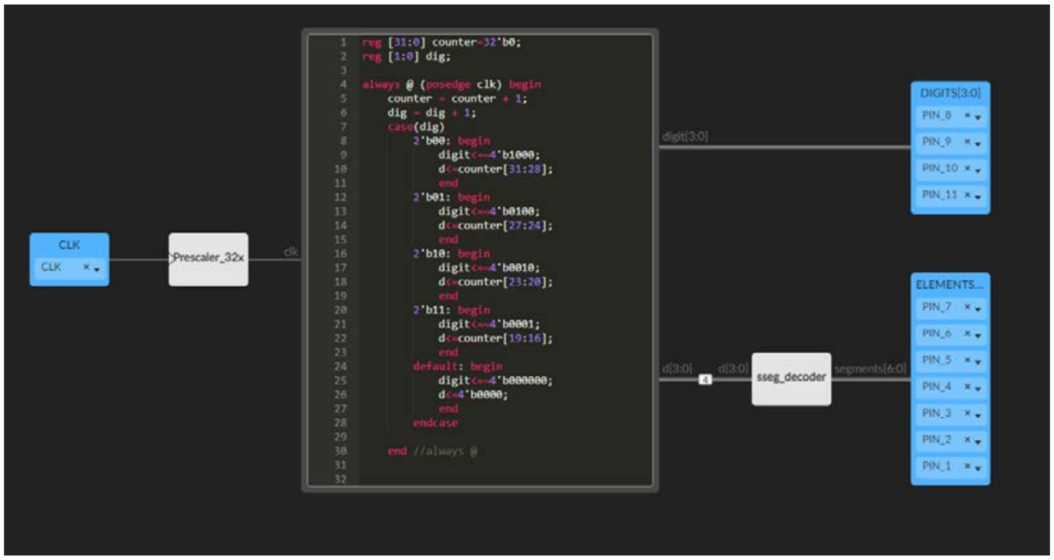
Figure 2. The seven-segment decoder code, ready to use as a module.

Close this new project and go back to the Lab 2 project.

Choose File → Add As Block…, and select the sseg_decoder.ice file that you just created.
You should now have a light grey block with input d[3:0] and output segments[6:0] (or [7:0]).

Select the code block containing the decoder code, and delete it.
In its place, wire up the module you just added.

If your new module doesn't have a name:
- Double-click it to "zoom in" to that module
- Click the lock to enable edits
- Edit → Project Information
- Change the name to sseg_decoder and click the lock to save the changes.



```verilog
reg [31:0] counter=32'b0;
reg [1:0] dig;

always @ (posedge clk) begin
    counter = counter + 1;
    dig = dig + 1;
    case(dig)
        2'b00: begin
            digit<=~4'b1000;
            d<=counter[31:28];
            end
        2'b01: begin
            digit<=~4'b0100;
            d<=counter[27:24];
            end
        2'b10: begin
            digit<=~4'b0010;
            d<=counter[23:20];
            end
        2'b11: begin
            digit<=~4'b0001;
            d<=counter[19:16];
            end
        default: begin
            digit<=~4'b000000;
            d<=4'b0000;
            end
        endcase

    end //always @
```

Figure 3. The Lab1 project with the decoder module

Exercise 2: Making a general-use four-digit display

Save your work (as Lab2_ex1 or similar) and then save a new copy (as Lab2_ex2 or similar).

This is progress, but what we really want is to be able to send four nibbles (two bytes) to a display module and have it handle sending it to the display for us.

We already have most of the code that we need – we only need to change from displaying different parts of a counter register to displaying the values of four input blocks.

First, let's add the four 4-bit inputs that we need. **Keep the clock.**
Double-click the top border of the counter code block, and add four inputs as shown:
da[3:0], db[3:0], dc[3:0], dd[3:0]
(You can name them a, b, c, and d if you name your output something other than "d".)

Next, modify the Verilog code inside the block to change what is sent to the output **d** for each digit.
For example, for case 00, we have **d <= counter[31:28];**
This first input should read from input A, so change this to **d <= da;**
d and **da** are the same bit width, so no bit-selection is necessary.

Change the other three cases, similarly. (Leave the code for "digits" alone.)

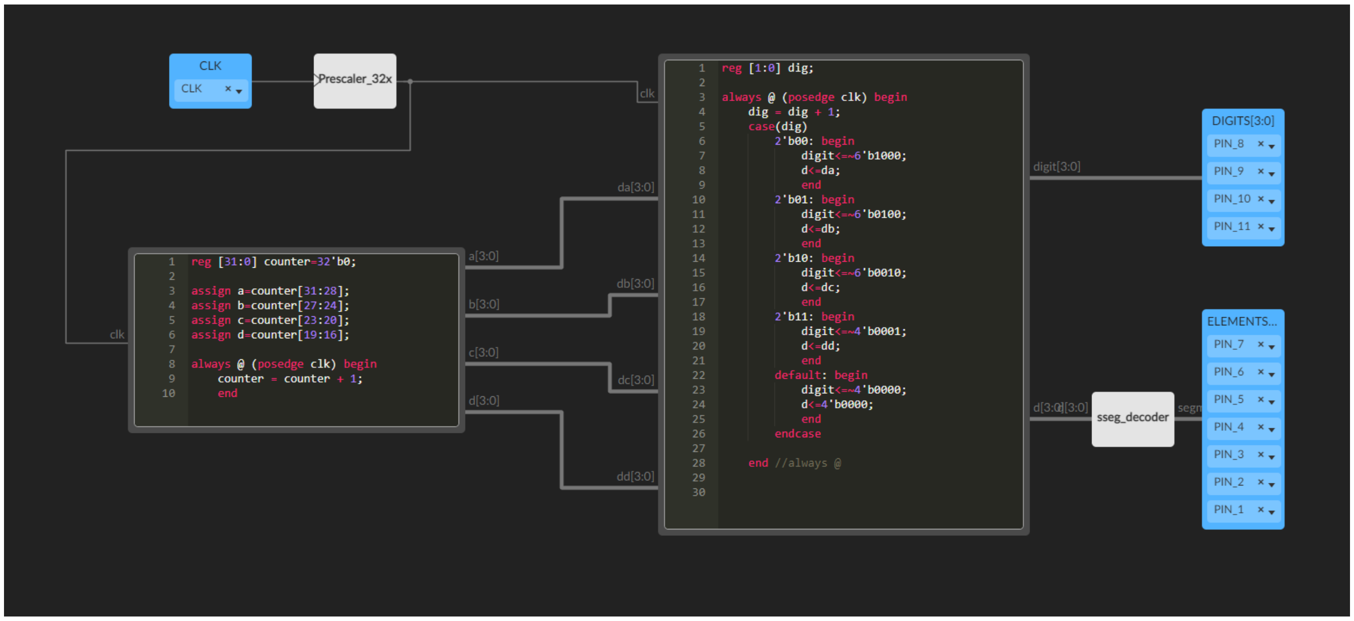
Grab a copy of the counter declaration at the top, then delete it and the "counter = counter + 1;" line.

Create a new code window with input **clk** and outputs **a[3:0], b[3:0], c[3:0] and d[3:0].**
This will give us the same counter functionality to test our display.

In the new code window, paste the definition for the counter along with an **always@(posedge clk)** block to increment the counter when the clock comes along.

Assign the outputs so that **a** is the highest four bits of the counter; **b** is the next four, and so on.
[31:28], [27:24], [23:20], [19:16].

Wire this module's **clk** input to the output from the clock prescaler module (See Figure 4, below.)
(If your counter goes too fast to see the last digit, check the clock connection.)



```
reg [31:0] counter=32'b0;

assign a=counter[31:28];
assign b=counter[27:24];
assign c=counter[23:20];
assign d=counter[19:16];

always @ (posedge clk) begin
    counter = counter + 1;
    end
```

```
reg [1:0] dig;

always @ (posedge clk) begin
    dig = dig + 1;
    case(dig)
        2'b00: begin
            digit<=~6'b1000;
            d<=da;
            end
        2'b01: begin
            digit<=~6'b0100;
            d<=db;
            end
        2'b10: begin
            digit<=~6'b0010;
            d<=dc;
            end
        2'b11: begin
            digit<=~4'b0001;
            d<=dd;
            end
        default: begin
            digit<=~4'b0000;
            d<=4'b0000;
            end
        endcase

    end //always @
```

Figure 4. The project at the end of Exercise 2.

CHECKPOINT: Working counter with modular decoder and a digit multiplexer (Figure 4.)

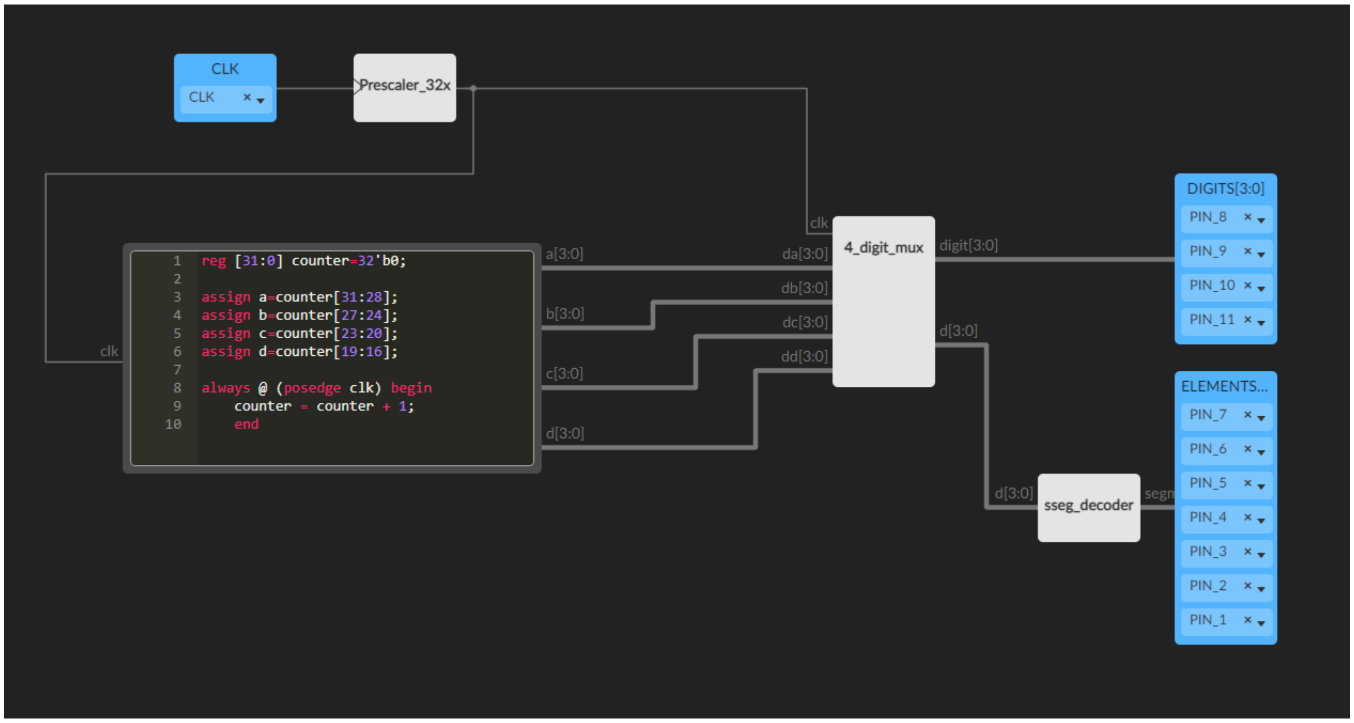
Stop for a moment and imagine building all of this with chips and breadboards.
RTL, and the ability to create circuitry by coding, is powerful.
…And we're only using ~1% (so far) of a relatively small, low-cost FPGA.

Exercise 3: Adding and wiring a SRAM memory block

Save your project (as Lab 2_ex2), then use Save As to save another copy as Lab 2b.
(IceStudio files don't take much space, and it's often useful to be able to have access to prior work.)

First, some preparation:

Package up your digit-multiplexer code block into its own module, as we did in Exercise 2.
Call the new module something like **digit_multiplexer.ice** and replace the code block with it.
(This will give us more room to work and clean up the design.)



```
reg [31:0] counter=32'b0;

assign a=counter[31:28];
assign b=counter[27:24];
assign c=counter[23:20];
assign d=counter[19:16];

always @ (posedge clk) begin
    counter = counter + 1;
    end
```

Figure 5. The project with the four-digit mux modularized.

CHECKPOINT: Counter still runs properly with the modular design.

Next, we want to add some memory to the design. For now, we'll use a pure 8-bit design, giving a memory size of 2^8 addresses, or 256 bytes. (*After all, 256 bytes should be enough for anyone, right?)*

We could make memory out of logical blocks, but it's very inefficient. LUTs are programmed when the bitstream is read in from the Flash memory, and can't function as RAM except to hold one bit each, so we would need 2,048 of them, which is over 1/4 of the chip's capacity of 7680 blocks.

Fortunately, Lattice (the FPGA manufacturer) included 32 "4k" blocks of RAM with 4096 bits (or 512 bytes) each. One of these is sufficient for our purposes, and in fact, we'll only use half.
If this somehow ends up being not enough for your project, there's a total of 16kB available on-chip, but you'd need to provide for up to 14 address lines (2^14 = 16384 = 16"k")

The easiest way to add block memory is to include a memory collection from GitHub.
Browse to https://github.com/FPGAwars/iceMem . Click on the down arrow next to the green **<> Code** button, and choose Download Zip. (There's a copy on BB Learn, but using GitHub is good practice.) Note where you downloaded the file.

In IceStudio, click on Tools → Collections → Add, and select the ZIP file you downloaded.
(You can call it IceMem or something similar.)

Click on the three bars at lower left, and open the Collection Manager.
Navigate through the IceMem menu to IceMem → SMemory → Bytes → memory-256B, and click.
A green dot will appear next to it, and you should be left holding a memory module. Place it somewhere.

(Exercise 3, continued.)

Next, let's add the inputs from the DIP switches. (***If you wired up four for Lab 1, add four more.)***
Add an input module called **switches[7:0]** or similar. These should be FPGA pins.
Assign these pins to the FPGA pins that you wired to your DIP switches, with the top entry (bit 7) corresponding to the leftmost switch (the most significant bit, or MSB).

Delete your Verilog code block with the counter – we will replace it with memory functionality.

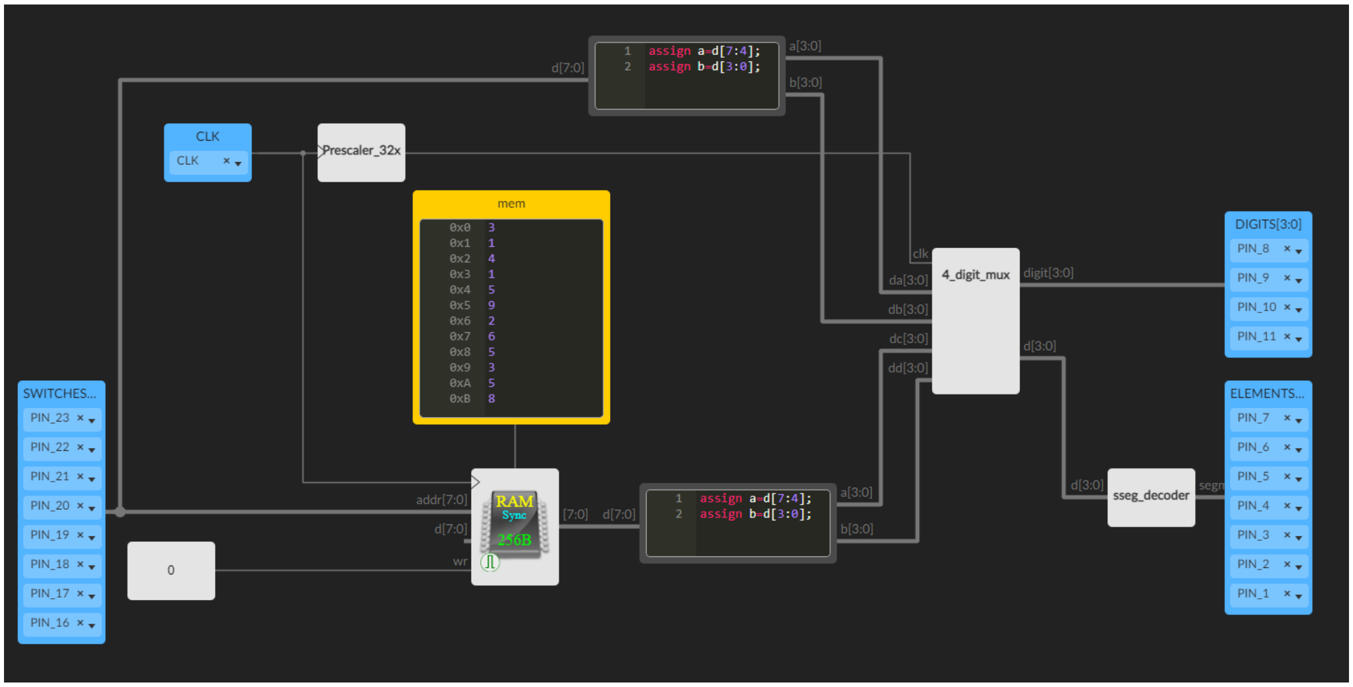
Wire the **switches[7:0]** input block into the **addr[7:0]** input of the SRAM module.

Go back to the Collection Manager and select Default collection → bit → 0.
This is a hardwired "0" input. Wire it to the **wr** input of the SRAM. (This makes it read-only, for now.)

Next, let's give our memory some initial values. From the toolbox, select a memory block.
It doesn't really matter what you call it – I called mine **mem**. I recommend Hexadecimal format.
You can resize this block to give you some room to work.
Wire the memory block to the SRAM (it works like a parameter, so is wired in from the top.)
Add some values for your memory to remember. I like to use the first several digits of Pi, but use whatever you like.

Next, we need to split the byte output from the memory into two four-bit nibbles.
Make a new Code block with input **d[7:0]** and outputs **a[3:0]** and **b[3:0]**.
Use two **assign** statements to make **a** equal to **d[7:4]** and **b** equal to **d[3:0]**.
(The correct code is shown below, but see if you can come up with the Verilog syntax yourself.)
Send **a** and **b** to the right two digits of your display: **dc** and **dd**.

Finally, we need to split the address byte. This is exactly the same task as before, so we can simply copy the Verilog code block that we just created, paste in another copy, and wire it up.
Do this, and connect the switches output to its input, and the outputs to display inputs **da** and **db**.



```
assign a=d[7:4];
assign b=d[3:0];
```

```
0x0  3
0x1  1
0x2  4
0x3  1
0x4  5
0x5  9
0x6  2
0x7  6
0x8  5
0x9  3
0xA  5
0xB  8
```

```
assign a=d[7:4];
assign b=d[3:0];
```

Figure 6. The project at the end of Exercise 3 – a read-only memory.

Reset your board and build and upload your project.

CHECKPOINT: The left two digits should show the address selected, and the right two digits should show the contents of the memory at that location (if you defined it in the memory block.) Memory locations not defined are probably set to zero (…but don't rely on this!)

Take a moment to double-click on the SRAM chip. It's not an inscrutable "black box," but specific constructions in Verilog code. Specifically, just one line actually allocates the memory:

reg [DATA_WIDTH-1:0] mem[0:SIZE-1];

…the rest of the code simply tells the compiler how to coordinate reads and writes.
Unlike most physical RAM chips, this memory has separate data lines for input and output, which will make our job easier. We can provide data to the input d port, and it will only be written to memory when the WR input is high when the clock edge comes along. Until then, ***data_out[7:0]*** *will show the contents of the memory specified by* ***addr[7:0]****.*

Back out of the SRAM module view without making any changes to it. ("*If it ain't broke…*")

Exercise 4: Writing to the memory

Save your project (as Lab2_ex3 or similar) and then save it again (as Lab2_ex4 or similar).

In order to write to the memory, we'll have to make the DIP switches do double duty as address and data inputs. So we'll need "load address" and "write" buttons.
Wire up two buttons as pull-down SPST inputs. Use the 3V3 input and a pull-down 1k to Ground.
MAKE SURE YOU DO NOT WIRE TO 5V, PLEASE – THESE ARE 3V3 PARTS!!

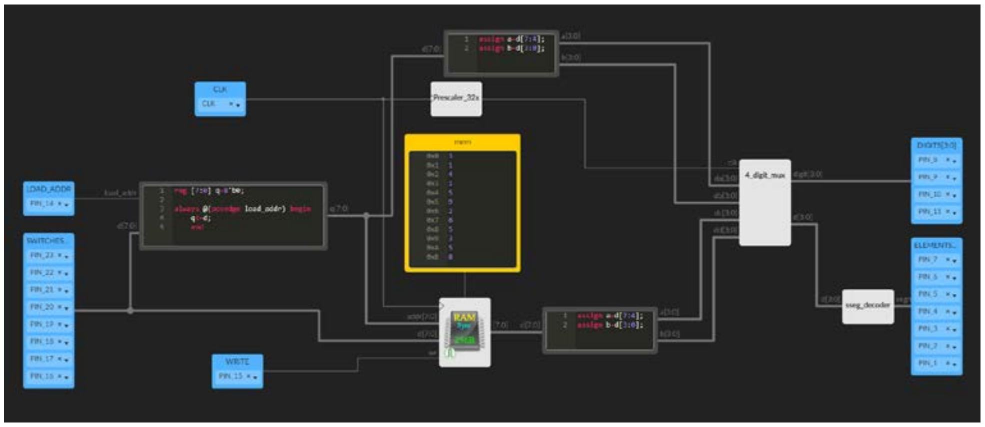
Setting up the Write pin is easy. Create an input block with that pin number, and use it to replace the logic 0 block that we wired into the **wr** input of the SRAM block.
Now the memory will still normally read back the data from the current address, until the Write button is pressed, **wr** goes high, and the new memory value will be loaded when the next clock comes along.

Wiring up the "load address" button is a little more involved.
Disconnect the address bus from the switches. Wire it instead to the **d[7:0]** input of the SRAM.
Rewire the connection into the top splitter (displays A and B) to the output of this new block.

We need to create a Verilog code block for the address buffer.
Create a new code block with inputs **load_addr** and **d[7:0]** and output **q[7:0]**.
Create a register for the output: **reg [7:0]q;** in the new code block.

Use a single **always** block to do the following:

- Whenever **load_addr** goes high, set **q** equal to **d**.
  (The correct code is shown below, but try to figure out the Verilog syntax yourself. Please ask if you have any questions about the syntax and/or the logic of the design, since each lab will be building on the foundations laid by the previous ones!)



```
reg [7:0] q=8'b0;

always @(posedge load_addr) begin
    q<=d;
    end
```

```
assign a=d[7:4];
assign b=d[3:0];
```

Figure 7. The completed Read/Write memory project – the foundation for a computer!

CHECKPOINT: Load_Addr changes the address, and data display follows memory contents.
Pressing Write overwrites the memory with the current switch settings.

PLEASE LEAVE YOUR PROJECT ASSEMBLED; we will build on it later!
(Put your group members' names on a note attached to your project.)
PLEASE SAVE YOUR FILES – they will be very useful later!

Discussion

In your lab report, please make sure you cover the following:

- What equipment and software did you use? (Just a few sentences is fine; the idea is you want other engineers to be able to replicate and build on your work.)

- What did you set out to make the board do? How? Was it successful? If so, how does it work? If not, what do you think went wrong, and what could be done to address this?
  (Contact me for help if you're having problems with any lab. ICEStudio is still experimental, so we may have to develop workarounds if we run into bugs.)

- (Make sure to include any code that you create or modify, and note where it goes in the project.)

- Include a block diagram of your project. Describe the function of any modules included (such as the digit multiplexer and seven-segment decoder.)

- Include a schematic of your project (logic switches, including pull-downs; LEDs and series resistors, etc.) Include the new Write button that you wired up in Exercise 4.

- How much of the FPGA resources are we using in terms of:
  - Logic cells
  - RAM blocks
  - I/O pins (go by what we have available, not what IceStudio thinks!)
    - Notably, how many free I/O pins do we have remaining?

END OF LAB 2