Cpr E 281 LAB12
ELECTRICAL AND COMPUTER ENGINEERING
IOWA STATE UNIVERSITY

Registers and Register Files

PRELAB!

Read the entire lab, and **complete** the prelab questions (Q1-Q2) on the answer sheet **before** coming to the laboratory.

1.0 Objectives

In this lab you will create a register file to store data. Once the data has been stored, further data processing may be used to conduct operations based on the stored data. Re-read the lecture slides on registers and register files and complete the prelab before you come to the lab.

2.0 Setup

Begin by creating a folder named **Lab12**, and then creating four subfolders **Lab12\step1**, **Lab12\step2**, and **Lab12\step3**.

A register file, is a series of interconnected parallel-access registers. Here, we will use a Block Design File to create the one-bit parallel-access register, then use Verilog to create larger registers to suit our needs.

While it is entirely possible to create a register file using only Schematic Captures or with only Verilog, we shall employ these methods together to demonstrate how Block Diagram Files may be used in a Verilog file and vice versa.

3.0 Parallel-Access 4-bit Register

Create a new project with a new schematic capture. Give this project the name **register**. Design the one-bit parallel access register with inputs Load, In, CLRN, and Clock, and with output Out. A sample diagram is shown below in Figure 1, but **you should also include the CLRN connection to your schematic**. Use a D flip-flop as memory and use the Quartus builtin **busmux** component as the multiplexer to choose the data source for the D flip-flop. Set the busmux width to 1 by right-clicking the busmux, going into Properties, selecting Parameter, and setting the WIDTH to 1. Save this file as **register.bdf**.

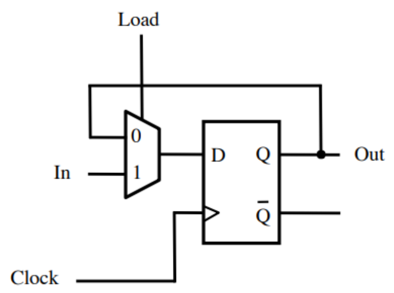


Figure 1: One-bit parallel access register.

Please note that several of the Quartus built-in components such as the **busmux** and **dff2** are not from the **primitives** folder when selecting them from the Symbol tool. These components will not be able to be brought directly into Questa ModelSim without extra effort (you may have to instead use an alternative implementation that includes the features you need by using the primitives).

When you have finished recreating the register in schematic capture, simulate this register on the FPGA. Assign Clock to a pushbutton, Out to any LED, and use a rocker switch for CLRN, In, and Load. Observe the operation of the register as follows:
To change the output of the board, the clock signal must produce a positive edge and Load must be 1. Once these requirements have been met, the output Out will change to match the input In. Fill in the characteristic table on your answer sheet.

After you have verified the proper operation of your one-bit register, create a symbol for your register. You will use multiple copies of this one-bit register to make much more elaborate register circuitry.

Make a new .bdf file, name it **reg4b.bdf**, and then verify that the files **register.bdf** and **register.bsf** are in this project folder. Make this new **.bdf** file the top-level entity for your project. This can be done by either pressing Ctrl-Shift-J on the reg4b file, or selecting **Project -> Set as Top-Level Entity** from the Quartus menu. Delete the pin assignments that you have made in **Assignments -> Pin Planner**; you will be reassigning these pins shortly.

Next, you will create a 4-bit register. Place four copies of the 1-bit **register** in your circuit schematic file. Connect all of the CLOCK inputs to the same input pin CLK. Similarly, connect all of the LOAD inputs to the same input pin LD and all of the CLRN inputs to the same input pin CLRN.

The input and output connections will be made as a bus. Add one input pin for the data input and name it **IN[3..0]**. The same method will be used for the output; name the output pin **OUT[3..0]**.

Connect the bus wires individually to each **register** as shown below, in Figure 2. Notice that each wire connecting to the IN bus and OUT bus connection is given the same name with a corresponding index (IN3, OUT3, IN2, OUT2, etc.). Make sure that the indexing you use is consistent; i.e., if a register receives **IN3**, it should also produce **OUT3**.

Once this circuit is complete, you will have four input pins (CLK, LD, CLRN, and IN[3..0]) and one output pin (Out[3..0]). This will be the register in the register file that you will create in the next step.

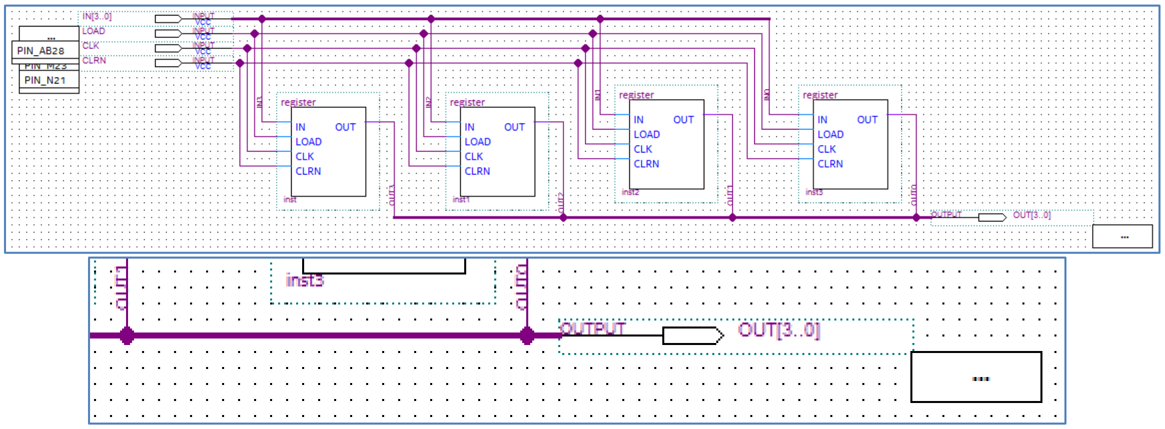


Figure 2: Four-bit parallel access register.

4.0 Register File

Now, it is time to create a register file, which is the next component in the hierarchy that you will create for this lab. The register file will have one write port, two read ports, and data will be stored in eight 4-bit registers. Therefore, you need to start by making eight copies of the four-bit register that you created in Part 3.0.

In addition to the eight registers, the register file also includes the controlling components that specify which register will be read and which register will be written. In order to read from the registers, you will use a multiplexer that will select which register output will become the output of the entire register file. To write to the registers, you will use a decoder. Since there are eight registers, you will need an 8-to-1 multiplexer and a 3-to-8 decoder.

The ports of the register file are as follows:
DATAP: First output of 4-bit data from the register file.
DATAQ: Second output of 4-bit data from the register file.
RP: 3-bit Read address that specifies which register will send its output through DATAP.
RQ: 3-bit Read address that specifies which register will send its output through DATAQ.
WA: 3-bit Write address that indicates which register will update its data.
LD_DATA: 4-bit data that will be written into the register specified by the address in WA.
WR: When this is zero, no register will update its data. When this is one, the register specified by WA will update its value with the value in LD_DATA.
CLK: A typical clock connection.

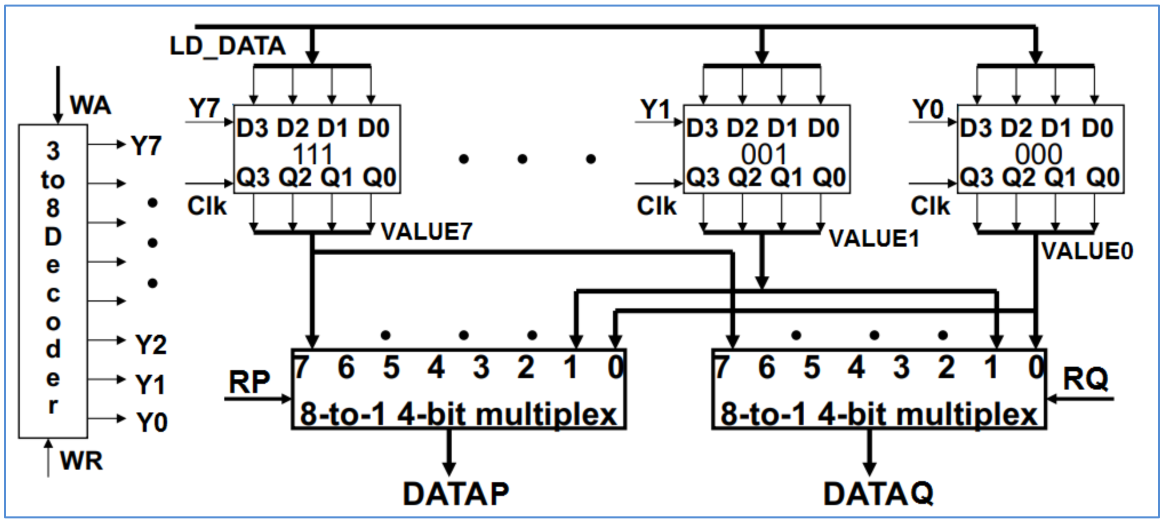


Figure 3: Rough schematic for the register file.

Start a new project in step2 and name it **regfile**. Open a new **Verilog** file. In this file, you will include several components that will provide the needed functionality for the register file. These components will include the the 4-bit registers that you created in the previous step; be sure to bring the **register** and **reg4b** bdf files into this new folder.

The contents of the register file are described in the following subsections.

4.1 Multiplexer

The multiplexer that will read from the register file will receive all of the outputs from each **reg4b** that exists within the regfile. Each of these outputs is a four-bit value and the multiplexer must then select one of these four-bit values to output. What you will create for this register file is a 4-bit 8-to-1 multiplexer. You will create this 4-bit 8-to-1 multiplexer in Verilog similarly to how you created the 1-bit 4-to-1 multiplexer in Lab 08. Make this multiplexer in a separate Verilog file named **Mux8_4b**. You will bring this file together with the other components in section 4.4.
Intermediate multiplexer values are four bit connections and should be declared as such. Although it is certainly possible to create the multiplexer without any intermediate expressions, it may be beneficial to include them. A four-bit intermediate connection can be declared as **wire [3:0] X**; this 4-bit wire X can be used to connect an output from a 2-to-1 multiplexer to another 2-to-1 multiplexer's input. There are other declarations that we need to do in this lab; see section 4.5 for more details.
Note that the output of each multiplexer is a single 4-bit value. The individual bits from the output bus can be referenced in Verilog using brackets. For instance, the individual bits from an output bus **DATAP** can be referenced as **DATAP[0], DATAP[1], DATAP[2],** and **DATAP[3]**. This will be used when we bring our components together in 4.4.

4.2 Decoder

The decoder will assert one of the LOAD lines for the eight registers, depending on the value of the write address. These LOAD lines will specify which register's value will be updated on the next clock edge. Also, add an ENABLE line to this decoder to allow for the opportunity to maintain all register values unchanged. To accomplish this, create a 3-to-8 decoder with ENABLE in Verilog. This, again, will be in a separate file named **Decoder3to8**.

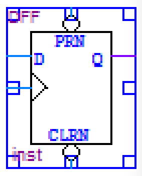
4.3 Register Collection

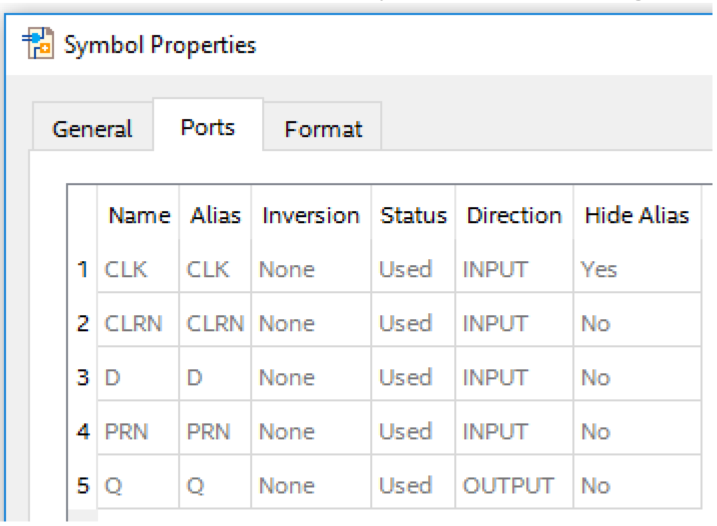
It is simple to implement this register file as a BDF, but we're going to implement it in Verilog. This requires us to instantiate a reg4b object with a Verilog file. How can this be done? Consider the following example:

```
module register(IN, LOAD, CLK, OUT, PRESET_N, CLEAR_N);
    input IN, LOAD, CLK, PRESET_N, CLEAR_N;
    output OUT;

    assign D = LOAD ? IN : OUT;
    DFF my_dff(.D(D), .CLK(CLK), .PRN(PRESET_N), .CLRN(CLEAR_N), .Q(OUT));
endmodule
```

This creates the same 1-bit register that you implemented in part 3.0, but the multiplexer is implemented with the ternary operator as it was used in Lab 08. The code instantiates a DFF (i.e., a D Flip-Flop), which is a native Quartus symbol, in the Verilog file.





	Name	Alias	Inversion	Status	Direction	Hide Alias
1	CLK	CLK	None	Used	INPUT	Yes
2	CLRN	CLRN	None	Used	INPUT	No
3	D	D	None	Used	INPUT	No
4	PRN	PRN	None	Used	INPUT	No
5	Q	Q	None	Used	OUTPUT	No

Figure 4: D Flip-Flop with its list of Ports, as seen from its Properties window.

The ports of the DFF are specified within its construction parameter list as follows: the ports are **D**, **CLK**, **PRN**, **CLRN**, and **Q**. In the code, the ports are specified between the parentheses as follows: `(.D(), .CLK(), .PRN(), .CLRN(), .Q())`

You can create duplicate objects from the code files that you've already used similarly to how the DFF was instantiated above while in Verilog. For example:

```
reg4b my_register(.IN(LD_DATA), .LOAD(Y0), .CLK(CLK), .OUT(R0), .CLRN(CLRN))
```

This will create one of the eight 4-bit registers from Part 3.0 that you will need to make the register file. You will have to specify the remaining connections to the other **reg4b** entities (see in Q5 in the prelab).

4.4 Bringing Everything Together

Now, you are ready to create the register file in Verilog. Start by declaring the module inputs and outputs as follows:

```
module regfile(DATAP3, DATAP2, DATAP1, DATAP0, DATAQ3, DATAQ2, DATAQ1, DATAQ0,
    RP2, RP1, RP0, RQ2, RQ1, RQ0, WA2, WA1, WA0, LD_DATA, WR, CLRN, CLK);

    // address and control ports
    input RP2, RP1, RP0, RQ2, RQ1, RQ0, WA2, WA1, WA0, WR, CLRN, CLK;

    // input data port
    input [3:0] LD_DATA;

    // output data ports
    output DATAP3, DATAP2, DATAP1, DATAP0, DATAQ3, DATAQ2, DATAQ1, DATAQ0;
```

Note that the register addresses are declared as individual bit connections, but the input data port is declared as a bus. The bus width is specified with its highest and lowest index as part of the declaration **input [3:0]**. This declares an input connection that has a width of four bits. Each bit can be accessed individually using the indices **[0], [1], [2],** and **[3]**. This will also be shown below.

The decoder can be added in the same way that you added the registers in part 4.3. A separate decoder file can be integrated with the register file module like this:

```
Decoder3to8 my_decoder(.EN(WR), .W2(WA2), .W1(WA1), .W0(WA0), .Y0(Y0), .Y1(Y1),
.Y2(Y2), .Y3(Y3), .Y4(Y4), .Y5(Y5), .Y6(Y6), .Y7(Y7));
```

The inputs to the decoder are also inputs to the register file. They determine which register, if any, will update its value. The outputs Y0 to Y7 are intermediate wires in Verilog that connect to the register **LOAD** connections as follows:

```
reg4b my_reg0(.IN(LD_DATA), .LOAD(Y0), .CLK(CLK), .OUT(VALUE0), .CLRN(CLRN))
```

This connects the decoder to register 0. The register's load connection is connected directly to the **Y0** output of the decoder. The output of the register is placed on the connection labeled **VALUE0**; this output will also become the output of the register file if address 0 is specified as either of the two read addresses. The data, clock, and clear inputs are inputs to the entire register file and will serve the same purpose for all registers. The remaining seven registers will be connected in the same way, except they will receive the corresponding Y output from the decoder and will output to the appropriate VALUE connection.

NOTICE: The multiplexer output is a 4-bit intermediate wired connection and must be declared as such. See section 4.5 for more information.

Once all of the registers have been placed in the file, we can use the multiplexer to select from one of the eight register outputs (**VALUE0, VALUE1, VALUE2, …**). Since our register file will have two read ports (**DATAP** and **DATAQ**), we will use two multiplexers to decide the output on each port. For instance, the output DATAP will be decided by a multiplexer that processes the values based on the read address in RP. In other words,

```
Mux8_4b my_muxP(.S2(RP2), .S1(RP1), .S0(RP0), .W0(VALUE0),
     .W1(VALUE1), .W2(VALUE2), .W3(VALUE3), .W4(VALUE4), .W5(VALUE5),
     .W6(VALUE6), .W7(VALUE7), .F(DATAP))
```

A similar result will be done for read address in RQ.

Note that the output of each multiplexer is a single connection that is 4-bits in width (e.g. **DATAP**), but the outputs of the register file, as shown previously, are four individual bits (e.g. **DATAP3, DATAP2, DATAP1,** and **DATAP0**). To make this connection work, we need to transform each of the four-bit bus connections (e.g., **DATAP**) into its four constituent wire connections. We can do this by referencing the indices of the bus connection as follows:

```
assign DATAP3 = DATAP[3];
```

Here, the square brackets are used to isolate a particular bit from the four-bit bus connection. The assign statement merely takes this wire off of the bus and connects it to the output of the register file. Do this for the remaining output data connections.

4.5 Wire Declarations

Wire declarations are also necessary. Previously, we did not have to declare intermediate wired connections in Verilog, since all intermediate values were just one bit in width. In this case, the output of the register file is a 4-bit value. If this output were left undeclared, then the compiler would assume this wire to be only one bit wide, as done before, which will create errors when these wires are used in place of the required four-bit connections. To avoid this problem, we will declare all of our intermediate buses as four-bit wires, as shown below.

```
// wire declarations
wire [3:0] VALUE0, VALUE1, VALUE2, VALUE3, VALUE4, VALUE5, VALUE6, VALUE7;
wire [3:0] DATAP, DATAQ;
```

These values are the outputs of the register file and the outputs of the multiplexer. They are all four-bit values that are neither a module input nor a module output. These declarations should be placed in the register file module with the declarations for the input and output ports.

After all of these components are specified and connected appropriately, the register file should be functional. Connect the input pins as follows: the 4-bit LD_DATA on toggle switches SW17 to SW14, the 3-bit write address WA on SW12 to SW10, the 3-bit read address RP on SW8 to SW6, the 3-bit read address RQ on SW5 to SW3, CLK on Pushbutton 3, WR on Pushbutton 2, and CLRN on Pushbutton 0. Connect each set of four outputs to a set of four adjacent LEDs. The LEDs connected to DATAP should be to the left of the LEDs connected to DATAQ.

Notice: The pushbuttons on the Altera Board are 0 when pressed and 1 when not pressed. The pushbuttons on the daughterboard are 1 when pressed and 0 when not pressed.

Your circuit should now be able to store eight 4-bit numbers. Test your circuit and once you understand how it works, demonstrate your result to the TA.

5.0 Interaction of the Register File with Other Components

Finally, you will interface the newly created register file with other components that you have created in previous labs. Before closing the register file project, make a symbol for the register file. Now, create a new project and name this project **lab12step3**. Create a new schematic file and save it. Bring the seven_seg_decoder files from Lab05 and the adder files from Lab07 into this new project. You will also need to copy the files used in step2 into the new folder; these files will be: **Decoder3to8.v**, **Mux8_4b.v**, **regfile.v**, **regfile.bsf**, **reg4b.bdf**, **register.bdf**, and **register.bsf**.

Once these files have been successfully copied into the new folder, connect the components as shown in the figure below. To set the width of the **busmux**, right-click the busmux symbol, select Properties, select Parameter, and set the **WIDTH** value to 4.

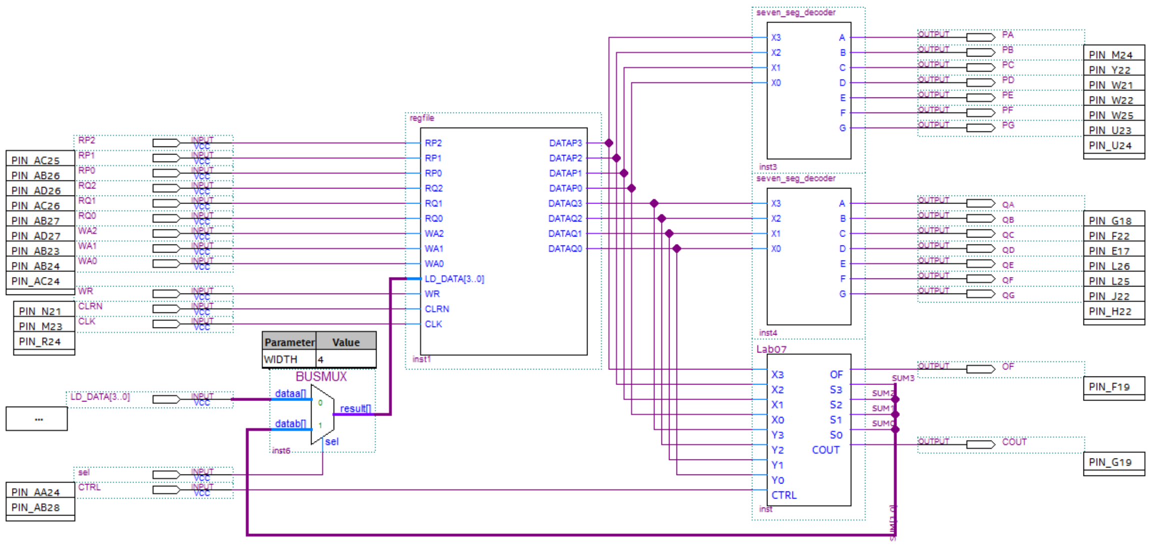


Figure 5: Register file with 4-bit adder and seven_seg_decoders connected.

The output of the 4-bit adder from Lab07 is connected onto a bus. The four wires that constitute the 4-bit bus connection are named and then placed on the bus, each with a unique address. The bus also has to be named **SUM[3..0]** so that the incoming connections are organized appropriately.

Don't forget to connect the busmux to the bus that contains the sum and assign a pin to its select input.

Once your circuit has been connected, demonstrate how it works to your TA.

6.0 Complete

You are done with this lab. Close all lab files, exit Quartus Prime, log off the computer, power down the DE2-115 board, and hand in your answer sheet. **Don't forget to write down your name and your lab section number**.