Project #2 Datapath

Team: Baby Geniuses (Michelle Blum, Christopher Dubois)
February 19, 2017

Overview

A General Purpose Computer (GPC) typically receives two pieces of data, performs an operation on this data, and produces an output. For this assignment, we were asked to design a GPC that produces a result using an Arithmetic and Logic Unit (ALU). This was accomplished through the use of registers, which aided in selecting the data for processing, and the use of multiplexers and demultiplexers to connect and direct the registers to the ALU symbol file.

We started by coding modules for the smaller components that would eventually make-up our GPC. Through our knowledge of the functions of a multiplexer, demultiplexer, D flip-flop (DFF), and ALU, we were able to generate Verilog code for each. After creating symbol files from each of the modules, we formed our top-level entity, the "8bit_GPC_test.bdf" file. In the block diagram, we connected all of our Verilog modules using buses, and then assigned LEDs and switches in order to visually test the functions on the DE2-115 board.

It took us a while to ensure that our program was appropriately functioning due to the fact that we kept mixing up the intent of the different switches. Another issue that we encountered was figuring out the direction that the data was flowing. Since feedback was involved, the data was not simply flowing in a single direction, which caused us to debate if certain elements of the BDF should be multiplexers or demultiplexers. Once we slowly and logically worked our way through the GPC block diagram file (BDF), we were able to decide which switches should produce which outputs. This also allowed us to finally decipher the elements of the circuit that should be a multiplexer, and the elements that should be a demultiplexer.

By the time we arrived to number 3, we knew that our code was performing successfully. However, we were not even close to being done with the GPC. Inputting all of the data and steering this data with switches was too confusing. To remove the switches, we added our Datapath Controller (DPC) module from Project #1. The next goal was to remove the necessity to input data. This was accomplished through of a Single Part ROM module. The function of this module was to allow the use of a text file with all of the binary data already written.

The last module we added was a clock divider. A clock was necessary because we needed data to enter the circuit, without overriding data that had previously entered. The appropriate moment for new data was after the previous data had been stored within the DFF. A correctly paced clock ensured the functionality of our circuit.

The work that was performed in this assignment allowed us to learn the true complexity of even the most general computer. This was the first time that we had ever been asked to design such a complex circuit. This helped us to realize how every little aspect that we have learned about (i.e - Multiplexers, Registers, etc.) is interrelated. Knowing how to program and implement a few of these concepts is useless; we must learn about all of them in order to create the level of functionality within machines to which we have become accustomed.

Solutions

Q1:

In question 1, we were asked to recreate the datapath from **Figure 1** using 8-bit multiplexers and latches:

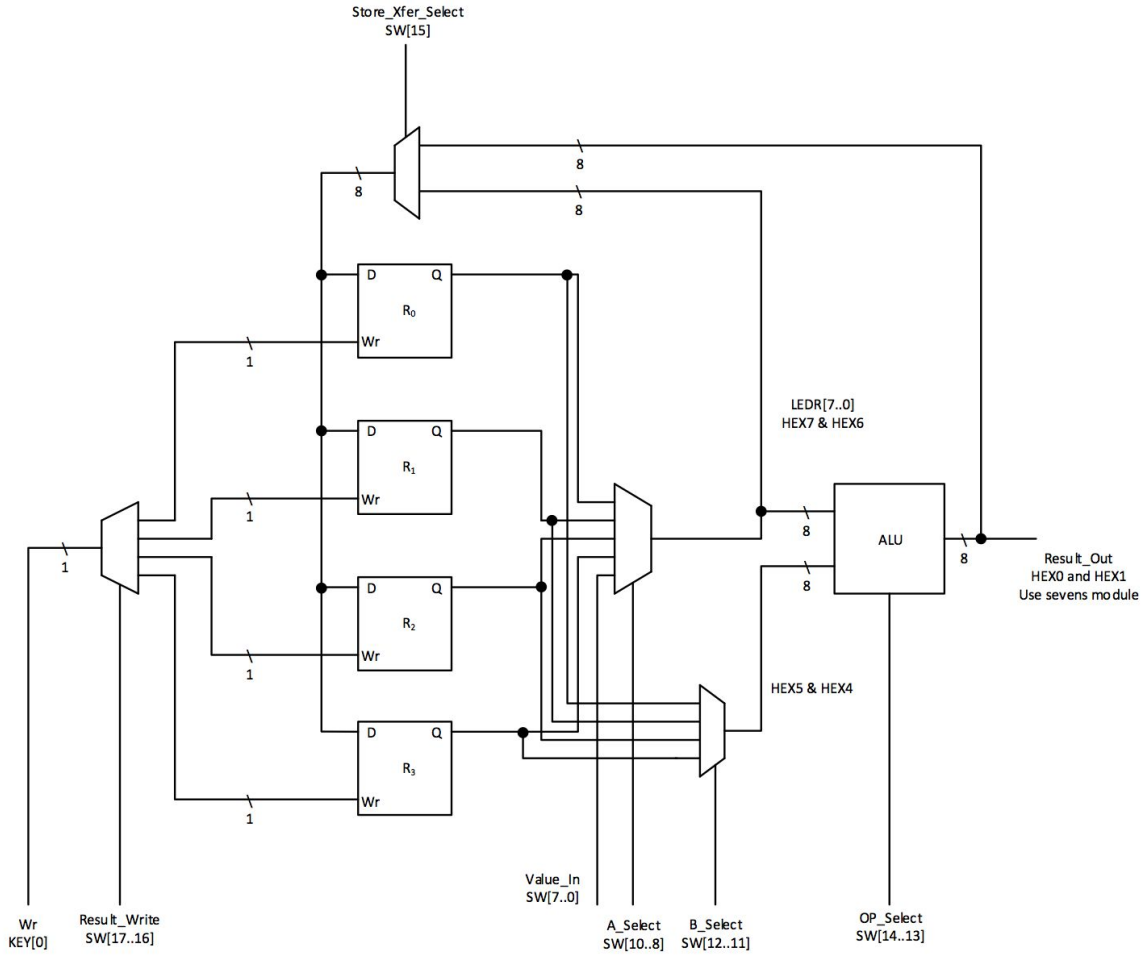

Figure 1: The circuit given by Question 1

To perform this task, we created several Verilog modules specific to the task, including a 2:1, 4:1, and 5:1 multiplexer, a 1:4 decoder, and an 8-bit D flip-flop. For the multiplexers we created multiple input wires and a single output wire that was case dependant. Paradoxically, for the demultiplexer, the value of a

single input decided which of multiple output options to select. Lastly, the DFF module contained an always block with a constant assignment of the letter D to the letter Q. Four of these were used in the BDF as the memory, or sequential part of our circuit. These modules can be seen in the Appendix section of this report.

Originally, we mistook the 2:1 multiplexer for a 1:2 decoder because of its orientation, but were quickly able to fix the error once we realized our perceived flow of the circuit was incorrect. We also initially used the provided ALU module, but found that it was easier to work with our own module from last semester's Introduction to Digital Systems course, as shown in **Code Block 1**. This used a case structure to decide the operation, and then performed the subsequent operation using the associated operator.

```
module ALU_New (input [7:0] a, input [7:0] b, input [1:0] SW,
output reg [7:0] result);
      always @(SW)   begin
            case(SW)
                  2'b00: result = a + b;
                  2'b01: result = a - b;
                  2'b10: result = a && b;
                  2'b11: result = a || b;
                  endcase
end
endmodule
```

Code Block 1: Arithmetic and Logic Unit

We assigned all the pins designated in the given diagram, using switches 0 through 17 for input to multiplexers where needed as well as selecting the ALU operation, and used the KEY0 button for writing the value to a given D flip flop. After quite a bit of tinkering and rearrangement, we were able to get the circuit in a serviceable order. **Figure 2** shows the circuit and all of the necessary connections made to get it running:

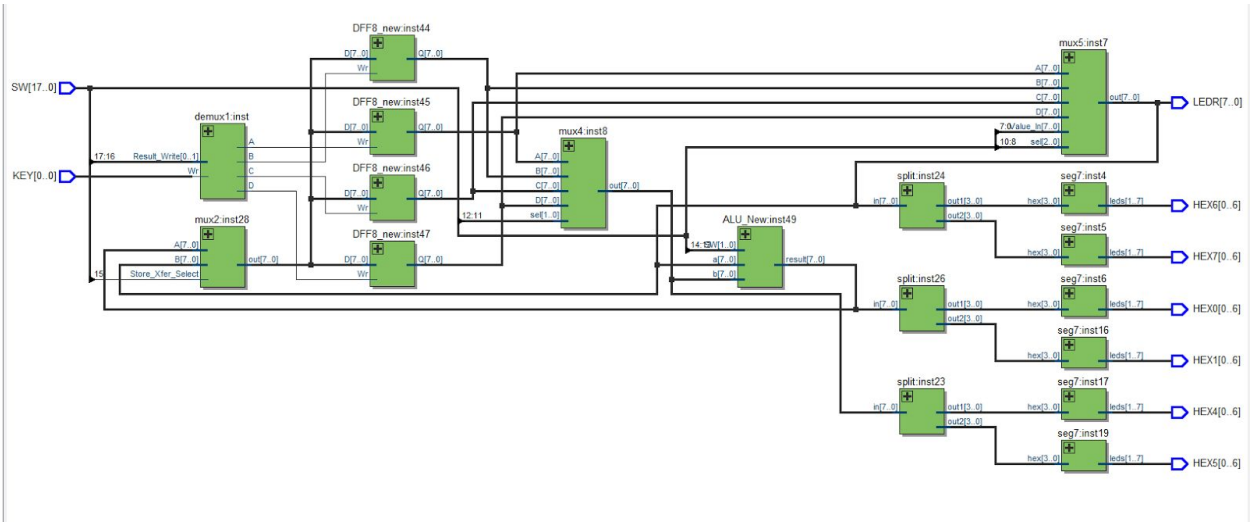

Figure 2: RTL diagram of the created circuit

The circuit functions in an interesting way. First, some 8-bit value is inputted through the Value_In input port. This value is then both sent to the 2:1 multiplexer and the ALU as the first operand; the output value of the ALU is also sent to the 2:1 multiplexer. The multiplexer then uses the select value (Select_Xfer_Select) to choose which of the two values advances through it. We then use the 1:4 decoder to select which register (R0, R1, R2, or R3) the value gets stored into. Once this is determined, these stored values can be sent to either the 5:1 multiplexer (whose output value is displayed on HEX6 & HEX7), the 4:1 multiplexer (whose output value is displayed on HEX4 & HEX5) , or both; these values, in turn, are used as the operands in the ALU. The ALU can either add, subtract, AND, or OR the two values. The output value is then fed back to the 2:1 multiplexer, while also being outputted to HEX0 & HEX1. This creates for a cyclic circuit that allows for basic arithmetic function and storage of the answer.

Q2:

When we initially tested our circuit, we were extremely disappointed to find out that it was not in working order. Luckily, since we understood how the circuit was supposed to function, through using the switches on the DE2-115 board, we were able to see that our issue was that no value was being saved into the DFFs. After trying a variety of fixes, what finally worked was removing our reset input. We had reset as part of the always block sensitivity, and as a case option. We believe that removing the input solved our problem because the reset value was not allowing our always block to run.

Now, with a functional circuit, we were able to complete the table as shown in **Table 1** below:

Operation	SW[17..16]	SW[15]	SW[14..13]	SW[12..11]	SW[10..8]	SW[7..0]
Copy RC->RA	RC	1	-	-	000	-
Load RC->Ext_in	**RC**	**1**	**-**	**-**	**100**	**Ext_in**
Add RC->RA + RB	RC	0	00	01	000	-
Sub RC->RA - RB	**RC**	**0**	**01**	**01**	**000**	**-**
And RC->RA & RB	**RC**	**0**	**10**	**01**	**000**	**-**

Or RC->RA | RB	**RC**	**0**	**11**	**01**	**000**	**-**
Addi RC->immed + RB	RC	0	00	01	100	immed
Subi RC->immed - RB	**RC**	**0**	**01**	**01**	**100**	**immed**
Andi RC->immed & RB	**RC**	**0**	**10**	**01**	**100**	**immed**
Ori RC->immed | RB	**RC**	**0**	**11**	**01**	**100**	**immed**

Table 1: Operational Instruction table

This table demonstrates the necessary binary input (using switches) to perform the operation listed in the left-most column. For example, the binary code required to copy the value in Register A to Register C is 101000000000000000. We came to this number by performing the action in question (replacing RA with R0, RB with R1, etc.) and recorded where the switches were at that point. The dashes in the table represent a value that does not influence the binary command, which we have used 0's for in our aforementioned example.

Q3:

For this question, we directly imported our DPC module from the last project. As the instructions suggested, we added an 18-bit input wire named code_data, as well as an internal 18-bit reg variable named INS and an 18-bit output wire named datapath_out. The significance of the 18-bit width is that the DPC will essentially be used in a system to replace the switches used in testing and create an automated version of it, run by outside instruction. The final version of the module is shown here in **Figure 3**:

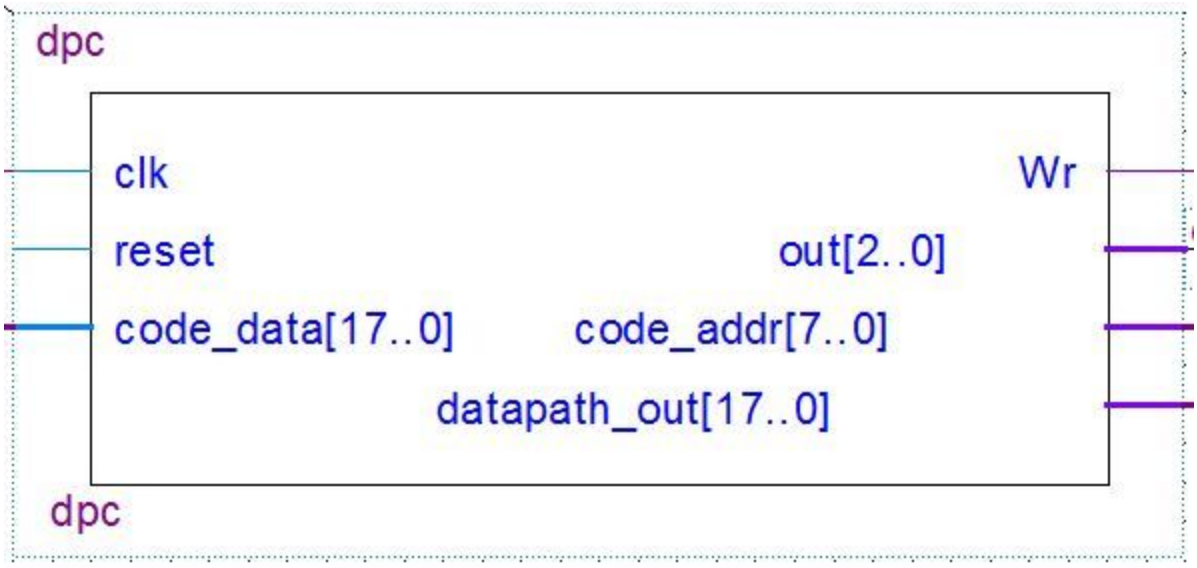

Figure 3: The modified Datapath Controller

Q4:

This question essentially instructed us step-by-step as to the actions to partake in, but did not explain why we should take these actions. The reason would be discovered later in questions 6 and 7.

We were asked to use the template of a Single Port ROM module (as seen in **Code Block 2)**. The only edit we were to make to this template was to change the name and the parameters. As would soon be revealed, the parameter DATA_WIDTH was set to 18 because, as shown in **Table 1** each string of data required 18 bits. The parameter for ADDR_WIDTH was changed to 8 because, in the end, an 8-bit value is displayed as the ALU result on the HEX displays.

Then, we created a symbol file from this module and added an instance of this to our top level entity. At this point, we predicted that this module would be used to replace us manually inputting the data. This prediction stemmed from the fact that our Datapath Controller inserted in question 3 replaced all the switches, and as a result removed our method of inputting the 18-bit data.

```
// Quartus Prime Verilog Template
// Single Port ROM

module instruction_rom
#(parameter DATA_WIDTH=18, parameter ADDR_WIDTH=8)
(
	input [(ADDR_WIDTH-1):0] addr,
	input clk,
	output reg [(DATA_WIDTH-1):0] q
);

	// Declare the ROM variable
	reg [DATA_WIDTH-1:0] rom[2**ADDR_WIDTH-1:0];

	// Initialize the ROM with $readmemb.  Put the memory contents
	// in the file single_port_rom_init.txt.  Without this file,
	// this design will not compile.

	// See Verilog LRM 1364-2001 Section 17.2.8 for details on the
	// format of this file, or see the "Using $readmemb and
$readmemh"
	// template later in this section.

	initial
	begin
		$readmemb("single_port_rom_init.txt", rom);
	end

	always @ (posedge clk)
	begin
		q <= rom[addr];
	end

endmodule
```

Code Block 2: Single Port ROM modified as Instruction ROM

Q5:

In this question, we are asked to rewire the system in a way that the circuit can run in an automated manner. In order to do this, we replaced all of the 18 switches with a bit of the datapath_out output from the DPC, effectively automating the program. We then also connected the instruction_rom and the DPC through varying inputs and outputs, as well as added a clock for automation purposes. We used the provided clock_divider module to split the ticks into seconds.

One vital piece not included in the instructions was wiring the Wr output of the DPC to the Wr input of the 1:4 decoder, which essentially makes storing the values possible by telling the D flip flops to write. We also wired the designated outputs (LEDs and HEXs) to the output ports. The result of inserting these additional modules is shown in **Figure 4** below:

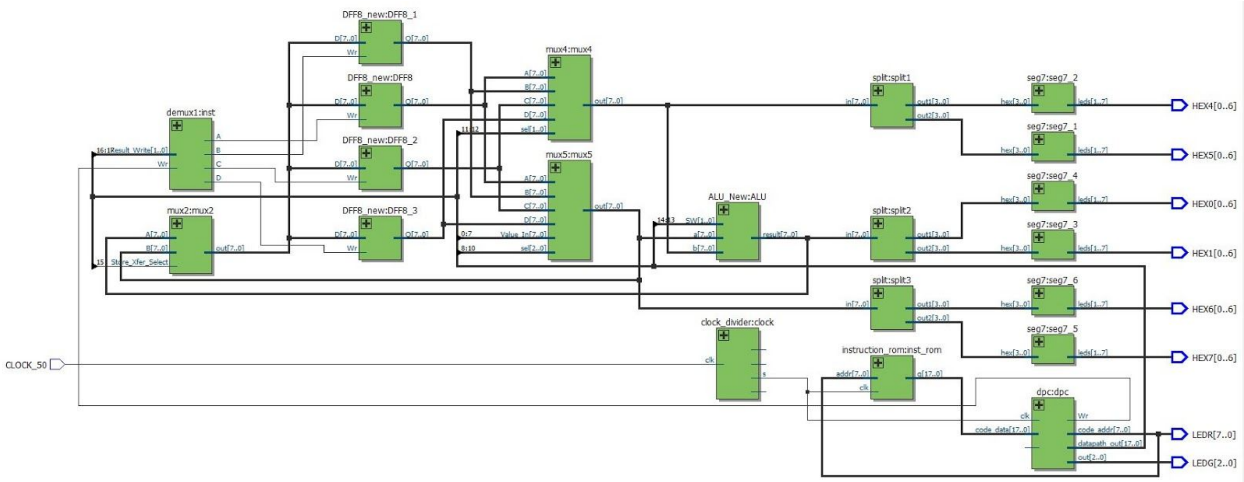

Figure 4: RTL diagram of final circuit, complete with DPC and Instruction ROM

Q6:

Prior to beginning this question, we attempted to run our circuit, and received an error message concerning a non-existent text file. Upon reading this question, we understood exactly what the error meant and created a text file entitled “single_port_rom_inti.txt” and added it to our Quartus project.

Our prediction from question 4 was correct; the Single Port ROM module was to be used to replace our manual input of the data. However, we never predicted how this would be accomplished. The necessary creation of a text file filled-in this missing piece of the puzzle.

The question stated that the values listed within this text file will be the 18-bit long value that we formed in question 2. Using the same logic as was used for the table, we created additional 18-bit data values and hand-compiled them in the next question.

Q7:

We created the binary values the same way that we created the value in question 2: by inputting the information into the DE2-115 board. For example, for the first value, R0 was represented by 00, the value of the first multiplexer is being loaded, making the next digit 0. The next four digits do not matter. After that, the value is 100, which sends the fifth value through as the output, and lastly, 13h converted to binary is 00010011. The final product is displayed on the first line of **Code Block 3** below. This method

of forming the binary data was repeated for the other three instructions, and then the last instruction was repeated a multitude of times.

```
0010000010000010011
0110000010000011001
1000001000000000000
1000000010000000000
1000000010000000000
1000000010000000000
1000000010000000000
1000000010000000000
1000000010000000000
1000000010000000000
1000000010000000000
```

Code Block 3: single_port_rom_init.txt in its entirety, used as instructions for the General Purpose Computer

The program is using the instructions to load two separate values into two separate D flip flops, add those two values through the ALU addition operation, store the result of that into another flip-flop, and then add that stored value to one of the original operands.

In order to make this program loop, you could have an if statement in the DPC (in the case statement) where it runs a certain function if the PC is above a certain value or not equal to a certain value. You could specify it even further with the if statements running between certain values of PC, making for precise numbers of loops.

Appendix

The following is code used as the solution to number 1 (demux1.v [Demultiplexer], mux2.v [2:1 multiplexer], mux4.v [4:1 multiplexer], mux5.v [5:1 multiplexer], DFF8_new.v [D flip flop], seg7.v [7 segment HEX display], and split.v [Split 8-bit values into two 4-bit values]).

```
module demux1(
      input Wr, input [1:0]Result_Write,
      output reg A, B, C, D);

      parameter sel0 = 2'b00, sel1 = 2'b01, sel2 = 2'b10, sel3 =
2'b11;

      always @(~Wr) begin
            case (Result_Write)
                  sel0:
```

```
                    A = ~Wr;
               sel1:
                    B = ~Wr;
               sel2:
                    C = ~Wr;
               sel3:
                    D = ~Wr;
               default:
                    A = ~Wr;
          endcase
     end
endmodule
```

```
module mux2(
     input [7:0]A, input [7:0]B, input Store_Xfer_Select,
     output reg [7:0]out);

     always @* begin
          case (Store_Xfer_Select)
               1'b0:
                    out = A;
               1'b1:
                    out = B;
               default:
                    out = 1;
          endcase
     end
endmodule
```

```
module mux4 (
     input [7:0]A, input [7:0]B, input [7:0]C, input [7:0]D, input
[1:0]sel,
     output reg [7:0] out
);


     parameter sel0 = 2'b00, sel1 = 2'b01, sel2 = 2'b10, sel3 =
2'b11;

     always @* begin
          case (sel)
               sel0:
```

```
                    out = A;
               sel1:
                    out = B;
               sel2:
                    out = C;
               sel3:
                    out = D;
               default:
                    out = A;
          endcase
     end

     endmodule
```

```
module mux5 (
     input [7:0]A, input [7:0]B, input [7:0]C, input [7:0]D, input
[7:0]Value_In, input [2:0]sel,
     output reg [7:0] out
);


     parameter sel0 = 3'b000, sel1 = 3'b001, sel2 = 3'b010, sel3 =
3'b011, sel4 = 3'b100;

     always @* begin
          case (sel)
               sel0:
                    out = A;
               sel1:
                    out = B;
               sel2:
                    out = C;
               sel3:
                    out = D;
               sel4:
                    out = Value_In;
               default:
                    out = A;
          endcase
     end

     endmodule
```

```
module DFF8_new(D, Wr, Q);


input wire[7:0] D;
input wire Wr;
output reg[7:0] Q;

always @(posedge Wr)
begin
           Q <= D;
end


endmodule
```

```
module seg7(
      input [3:0] hex,
      output reg [1:7] leds); // a..g

      always @(hex)
            case(hex)
                  0: leds = 7'b0000001;
                  1: leds = 7'b1001111;
                  2: leds = 7'b0010010;
                  3: leds = 7'b0000110;
                  4: leds = 7'b1001100;
                  5: leds = 7'b0100100;
                  6: leds = 7'b0100000;
                  7: leds = 7'b0001111;
                  8: leds = 7'b0000000;
                  9: leds = 7'b0000100;
                  10: leds = 7'b0001000; // A
                  11: leds = 7'b1100000; // b
                  12: leds = 7'b0110001; // C
                  13: leds = 7'b1000010; // d
                  14: leds = 7'b0110000; // E
                  15: leds = 7'b0111000; // F
            endcase
endmodule
```

```
module split (
  input [7:0] in,
```

```
      output [3:0] out1, output [3:0] out2
);

  assign out1 = in[3:0];

  assign out2 = in[7:4];

endmodule
```

The following code is used as the solution to number 5 (clock_divider.v [Clock]).

```
module clock_divider #(parameter CLOCKFREQ = 50000000)(
      input clk,
      output reg s,          //Seconds
      output reg t,          //Tenths of seconds
      output reg h,          //Hundredths of seconds
      output reg m           //milliseconds
      );

      integer millicount = 0;
      integer hundredcount = 0;
      integer tencount = 0;
      integer secondcount = 0;

      //Divide the clock to get milliseconds:
      always @ (posedge clk)
      begin
            if (millicount == (CLOCKFREQ / 2000 - 1))
            begin
                  millicount <= 0;
                  m <= ~m;
            end
            else
                  millicount <= millicount + 1;
      end

      //Divide the millisecond output to get hundredths:
      always @ (posedge clk)
      begin
            if (hundredcount == (CLOCKFREQ / 200 - 1))
            begin
                  hundredcount <= 0;
                  h <= ~h;
            end
            else
```

```
                hundredcount <= hundredcount + 1;
      end

      //Divide to get tenths:
      always @ (posedge clk)
      begin
            if (tencount == (CLOCKFREQ / 20 - 1))
            begin
                  tencount <= 0;
                  t <= ~t;
            end
            else
                  tencount <= tencount + 1;
      end

      //Divide the tenths output to get seconds:
      always @ (posedge clk)
      begin
            if (secondcount == (CLOCKFREQ / 2 - 1))
            begin
                  secondcount <= 0;
                  s <= ~s;
            end
            else
                  secondcount <= secondcount + 1;
      end

endmodule
```