Project #1 Datapath Controller

Team: Baby Geniuses (Michelle Blum, Christopher Dubois)
February 5, 2017

Overview

Every processor has a collection of functions, and this collection is modeled by a Datapath. The Datapath Controller (DPC) ensures that these functions are appropriately executed by controlling the flow between parts of the processor. For this assignment we were asked to demonstrate the stages of the the Instruction Cycle through the creation of a DPC. The process that a DPC follows can be demonstrated with a Moore Finite State Machine.

We applied the "Four State Moore Finite State Machine" Quartus template to our top-level module entitled dpc.v. The template was easily adjusted to accommodate the 5-state cycle that our code was to implement. In order to test our completed module, we used a block diagram named dpc_test.bdg. In the block diagram, we assigned a clock and a reset signal to the module under test (MUT). To verify that the MUT was functioning as intended, we assigned green LEDs to represent each state, and a red LED to represent the output at each state on the DE2-115 board.

After ensuring that we fully understood how DPCs function, we turned our focus towards transferring this function to our Verilog code. Since we knew what to anticipate, we were aware when a minor error in code or our block diagram must have been present. For example, we would occasionally forget semicolons or not fully connect the wires in the block diagram file (BDF).

Once we arrived at parts 1c and 1e, we saw that our code was performing successfully. Being able to validate the success of our program through observation on the DE2-115 board was extremely helpful. At the completion of step 1c, our code already demonstrated the steps of the Instruction Cycle implemented by a DPC. However, with the addition of a counter from steps 1d and 1e, we were also able to show the continuous cyclic nature of the DPC.

The work that was performed in this assignment allowed us to reinforce our knowledge pertaining to the steps of the Instruction Cycle and the usefulness of state machines. We need to learn to visualize a way to code the function of concepts that are central to processors, such as the Instruction Cycle. By understanding the versatility of Moore/Mealy Machines, and by beginning to increase our understanding of other approaches, such as registers, this process of visualization should become second-nature.

Solutions

Q1:

After inserting the "Four State Moore Machine" Quartus template, we immediately noticed that the code was split into two always blocks. This increased the readability of the code by placing a single required component of a Moore Machine into each block - the output and the order or succession. The first block defined each state's output, while the second block defined the order of the cycle.

Our Moore Machine needed five states, not four. To add a state, we added a fifth parameter to the template. By doing this, we also needed an additional situation in each case structure. Each of the states corresponded to a step in the Instruction Cycle: RESET, FETCH, DECODE, EXECUTE and WRITE.

RESET is the first step in the Instruction Cycle, and once this state occurs, the only way to return is by performing a separate action (in our case, by pressing a button). The cycle never returns to RESET on its own. Once the RESET step is complete, the Instruction Cycle fetches the instruction. FETCH is the step in the cycle that is returned to upon each rotation. Following FETCH is the DECODE component of the cycle, which interprets the instruction. Now that the instruction is able to be read by the processor, the next step is to execute the instruction and then to write the output.

For our code, which is partially displayed in Code Block 1, the output is named Wr. It is one-bit in width because the output is either true or false, or 0 and 1 (binary output). Paradoxically, our state variables were three-bits wide because there were five options. The values were 000 (RESET), 001 (FETCH), 010 (DECODE), 100 (EXECUTE), 110 (WRITE).

The initial code contained two inputs, in and Reset. We removed the input in because a Moore Machine does not require any additional input. As for the other initial input, as aforementioned, reset is the name of the first step in the Instruction Cycle that is never returned to unless an outside action occurs. Whenever the input Reset becomes logic 0, the Instruction Cycle returns to the RESET state. We used the pin assignment associated with KEY[0] for the Reset input, as shown in Figure 1. This way, whenever the button is pressed, the input becomes logic 0, and while released, the input remains at logic 1.

However, logic 0 signifying the signal to be sent is not intuitive. To counter this, we added a "not" gate to our BDF (visible in Figures 1, 2, and 3). The "not" gate allowed us to state in our code that logic 1 should activate the lines within the RESET case, and that logic 0 allows the cycle to continue.

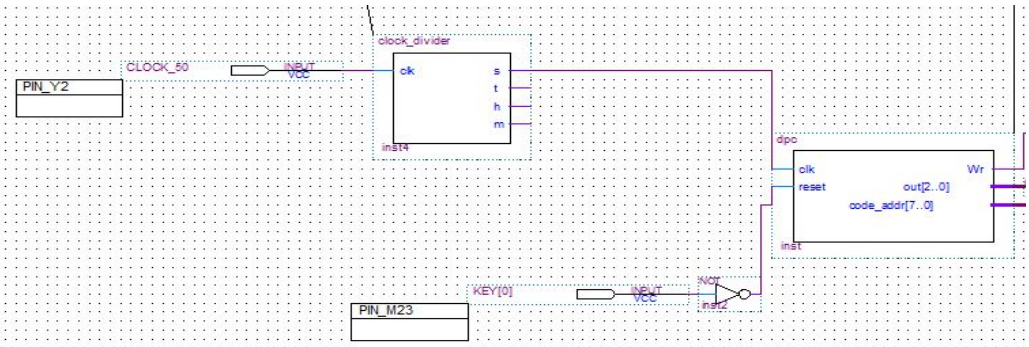

Figure 1: BDF symbol for the DPC module and its inputs

In order to facilitate the testing of our code, we created an output wire for the five states. This way, in the BDF, we were able to connect a bus to the new output that associated each state variable with three green LEDs. Each LED signified a bit of the state's value. The output, Wr, was simply connected to a single red LED.

```
always @ (posedge clk or posedge reset) begin
		if (reset)
			begin
				state = RESET;
				code_addr = 0;
				PC = 0;
			end
		else
			case (state)
				RESET:
					begin
						Wr= 1'b0;
						out = 3'b000;
						state = FETCH;
					end

				FETCH:
					begin
						Wr= 1'b0;
						out = 3'b001;
						state = DECODE;
					end
				DECODE:
					begin
						Wr= 1'b0;
						out = 3'b010;
						PC = PC + 1;
						code_addr = PC;
						state = EXECUTE;
					end

				EXECUTE:
					begin
						Wr= 1'b0;
						out = 3'b100;
						state = WRITE;
					end

				WRITE:
					begin
						Wr= 1'b1;
						out = 3'b110;
						state = FETCH;
					end

				default:
					state = RESET;

			endcase
```

Code Block 1: An excerpt of the main Verilog code, dpc.v, which illustrates the always statement that determines state and output

To complete the process of assigning pins to each of our inputs and outputs, we implemented the clock divider module (visible in Figures 2 and 3). This module allowed us to use the seconds output from the clock to drive our circuit. We also needed to assign the pin for the 50 MHz clock on the DE2-115 board to the input of the clock divider.

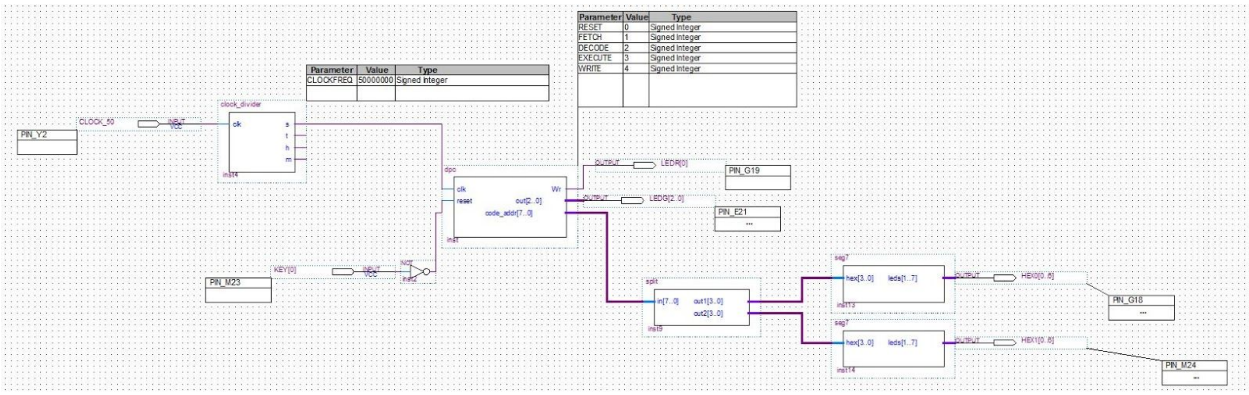

Figure 2: The complete BDF file for testing the Datapath Controller

Now, we were ready to test our circuit. The output Wr appeared to lag behind the state variable. We tried to resolve this issue through the use of both continuous/non-continuous assignments, through the reordering of code, etc. Unfortunately, nothing seemed to fix our issue. We could not find an error in our code, and none of the lab assistants that we asked could find one either. The only successful solution to this problem was to combine both always statements into one. Since the output and state were both within the same statement, the concept of one lagging behind the other would have been impossible. Although this may lessen the readability of the code, we found functionality to be a priority.

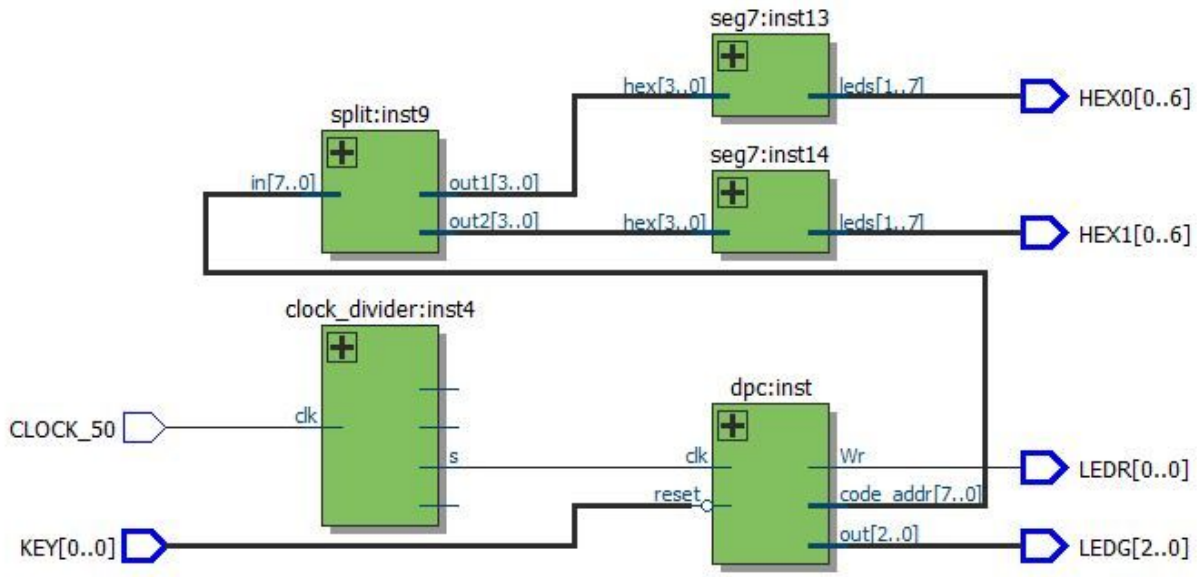

Figure 3: The RTL diagram that shows the diagram in clearer detail

Once the code started working, we moved onto number 1d, which requested that we create a counter variable that increments whenever the state changed from FETCH to DECODE. We created an 8-bit reg variable entitled PC, which is an acronym for “Program Counter”. When there were two always blocks, we placed the incrementation (PC = PC + 1) into the second block, where the order that the cycle

followed was stated. This is because the always statement that contained the output at each variable has no impact on moving between states. However, with only one always block, the incrementation was placed into the FETCH case, but after the line that stated the next state. This way, the compiler is aware that the line is to be executed in between the two states, not whenever the FETCH state occurs.

By taking the concept of an incrementing variable, we created an output entitled code_addr that would show the value of this variable on two hex digits. In order to have an output that is constantly changing, a continuous assignment to the incrementing variable had to be added to our code. This was done by creating an 8-bit wire variable and setting it equal to the 8-bit reg variable each time that PC changed.

To display the count on two hex digits, a 7-segment decoder is needed. This is because there are seven segments that make-up a single hex digit, and in order to represent a number, the DE2-115 board needs to be told which of these segments to light-up. The board knows which segments to light-up through a binary representation of each digit. However, for the hex digits, logic 0 represents a lit segment, while logic 1 represents a non-lit segment.

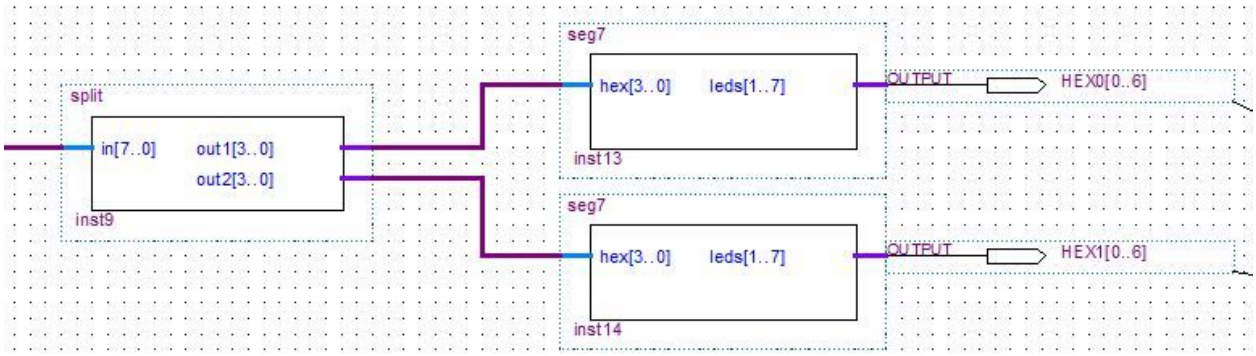

Figure 4: BDF diagram of the split module and the 7 segment displays

Two of the 7-segment decoders were necessary (one for HEX0 and one for HEX1). The next challenge was to connect our 8-bit counter variable to the 4-bit inputs of the 7-segment decoder modules. In order to do this, we created a module entitled split.v that splits the 8-bit outputs into two 4-bit variables. We then connected our dpc.v module to the split.v module, which was then connected to the 7-segment decoder modules on the BDF (visible in Figure 4). We were then able to run our dpc_test.bdf file and see a counter on two hex digits that incremented whenever the state changed from FETCH to DECODE. When the KEY[0] button is activated and the system is returned to the RESET state, the counter also returns to zero.

Appendix

The following is code used as the solution to number 1 (DPC.v [Datapath Controller], split.v [Splitting 8-bit signal into 2 4-bit signals], clock_divider.v [used to convert the on-board clock of the DE2-115 to seconds], and seg7.v [used to display a 4-bit value onto the 7-segment display]):

```
// Datapath Controller
// 5-State Moore state machine

// A Moore machine's outputs are dependent only on the current
state.
// The output is written only when the state changes.  (State
// transitions are synchronous.)

module dpc
(
      input clk, reset,
      output reg Wr, output reg [2:0]out, output reg [7:0]code_addr
);

      // Declare state register
      reg        [2:0]state;
      reg        [7:0]PC;

      // Declare states
      parameter RESET = 0, FETCH = 1, DECODE = 2, EXECUTE = 3, WRITE
= 4;

      // Determine the next state
      always @ (posedge clk or posedge reset) begin
            if (reset)
                  begin
                        state = RESET;
                        code_addr = 0;
                        PC = 0;
                  end
            else
                  case (state)
                        RESET:
                              begin
                                    Wr= 1'b0;
                                    out = 3'b000;
                                    state = FETCH;
```

```
                        end

                    FETCH:
                        begin
                            Wr= 1'b0;
                            out = 3'b001;
                            state = DECODE;
                        end
                    DECODE:
                        begin
                            Wr= 1'b0;
                            out = 3'b010;
                            PC = PC + 1;
                            code_addr = PC;
                            state = EXECUTE;
                        end

                    EXECUTE:
                        begin
                            Wr= 1'b0;
                            out = 3'b100;
                            state = WRITE;
                        end

                    WRITE:
                        begin
                            Wr= 1'b1;
                            out = 3'b110;
                            state = FETCH;
                        end

                    default:
                        state = RESET;

                endcase
    end

endmodule
```

```
//Split Module - used to divide the 8-bit signal of the PC counter
into two separate 4-bit signals

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
```

```
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
```

```
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
```