Ayden Dauenhauer, Edward Park
Prof. Yang
ECEN 122L Tuesday 2:15 p.m.
23 January 2024

Lab 2 - State Machine Control of Simple Datapath

Verilog Code:

```
/* output state logic
input: state
output: 8 control signals
TODO: you need to complete the output logic in the following always statement
*/
        always@(*)
        begin
          case(state)
            IDLE:
              begin
                _Extern = 1'b0;
                Gout = 1'b0;
                Ain = 1'b0;
                Gin = 1'b0;
                RdX = 1'b0;
                RdY = 1'b0;
                WrX = 1'b0;
                add_sub = 1'b0;
              end
            LOAD:
              begin
                _Extern = 1'b1;
                Gout = 1'b0;
                Ain = 1'b0;
                Gin = 1'b0;
                RdX = 1'b0;
                RdY = 1'b0;
                WrX = 1'b1;
                add_sub = 1'b0;
              end
            READ_Y:
              begin
                _Extern = 1'b0;
                Gout = 1'b0;
```

```
            Ain = 1'b1;
            Gin = 1'b0;
            RdX = 1'b0;
            RdY = 1'b1;
            WrX = 1'b0;
            add_sub = 1'b0;
        end
    READ_X:
        begin
            _Extern = 1'b0;
            Gout = 1'b0;
            Ain = 1'b1;
            Gin = 1'b0;
            RdX = 1'b1;
            RdY = 1'b0;
            WrX = 1'b0;
            add_sub = 1'b0;
        end
    ADD:
        begin
            _Extern = 1'b0;
            Gout = 1'b0;
            Ain = 1'b0;
            Gin = 1'b1;
            RdX = 1'b1;
            RdY = 1'b0;
            WrX = 1'b0;
            add_sub = 1'b0;
        end
    SUB:
        begin
            _Extern = 1'b0;
            Gout = 1'b0;
            Ain = 1'b0;
            Gin = 1'b1;
            RdX = 1'b0;
            RdY = 1'b1;
            WrX = 1'b0;
            add_sub = 1'b1;
        end
```

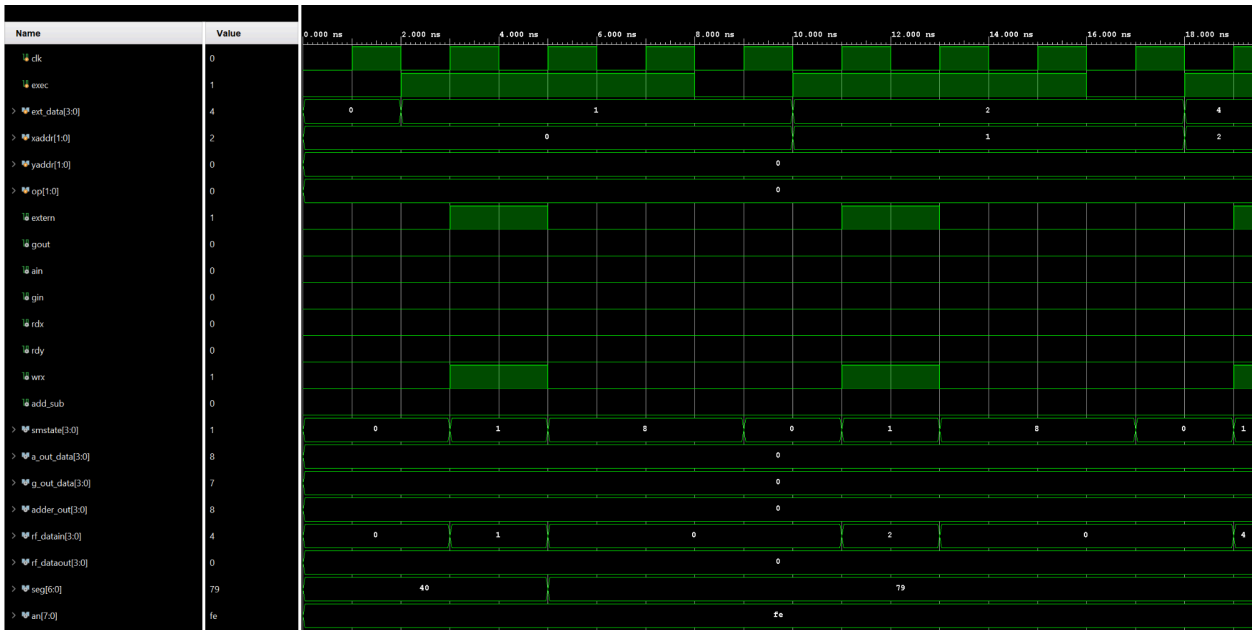
```
            MV:
               begin
                  _Extern = 1'b0;
                  Gout = 1'b0;
                  Ain = 1'b0;
                  Gin = 1'b1;
                  RdX = 1'b0;
                  RdY = 1'b0;
                  WrX = 1'b0;
                  add_sub = 1'b0;
               end
            WRITE_X:
               begin
                  _Extern = 1'b0;
                  Gout = 1'b1;
                  Ain = 1'b0;
                  Gin = 1'b0;
                  RdX = 1'b0;
                  RdY = 1'b0;
                  WrX = 1'b1;
                  add_sub = 1'b0;
               end
            DONE:
               begin
                  _Extern = 1'b0;
                  Gout = 1'b0;
                  Ain = 1'b0;
                  Gin = 1'b0;
                  RdX = 1'b0;
                  RdY = 1'b0;
                  WrX = 1'b0;
                  add_sub = 1'b0;
               end
         endcase
      end

/* Next state logic combined with flip-flops
input: clk, execute (stop/go signal), operation (opcode)
output: state
```

TODO: you need to complete this Next state logic (combined with flip-flops) in the following always statement

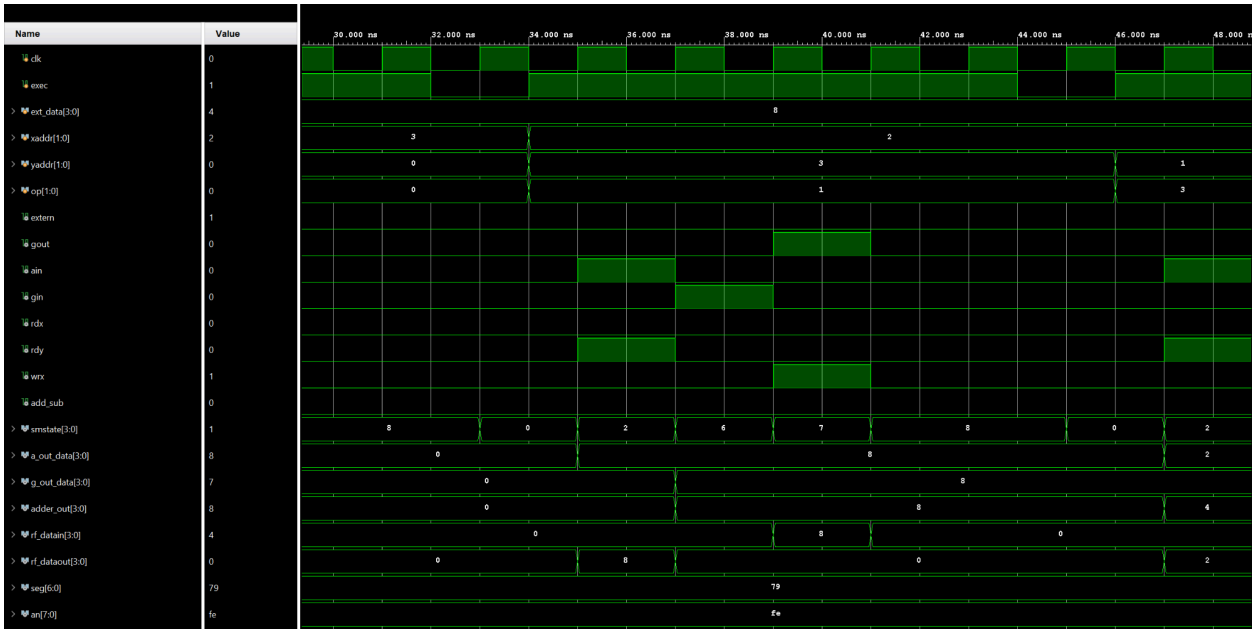
```
	*/
	always@(posedge clk)
	begin

	case(state)
		IDLE: begin
			if(execute == 1 && operation == 0) state <= LOAD;
			if(execute == 1 && operation == 1) state <= READ_Y;
			if(execute == 1 && operation == 3) state <= READ_Y;
			if(execute == 1 && operation == 2) state <= READ_X;
			end

		LOAD: begin
			state <= DONE; // always go to the DONE state at the next clock tick
			end

		READ_Y: begin
				if(execute == 1 && operation == 1) state <= MV;
				if(execute == 1 && operation == 3) state <= ADD;
			end

		READ_X: begin
				if(execute == 1) state <= SUB;
			end

		ADD: begin
				if(execute == 1) state <= WRITE_X;
			 end

		SUB: begin
				if(execute == 1) state <= WRITE_X;
			 end

		MV: begin
				if(execute == 1) state <= WRITE_X;
			end

		WRITE_X: begin
```

```
                    if(execute == 1) state <= DONE;
                end

            DONE: begin
                    //back to idle if execute back to low
                    if(execute == 0) state <= IDLE;

                end

                default: state <= IDLE;

            endcase

        end //end always
```

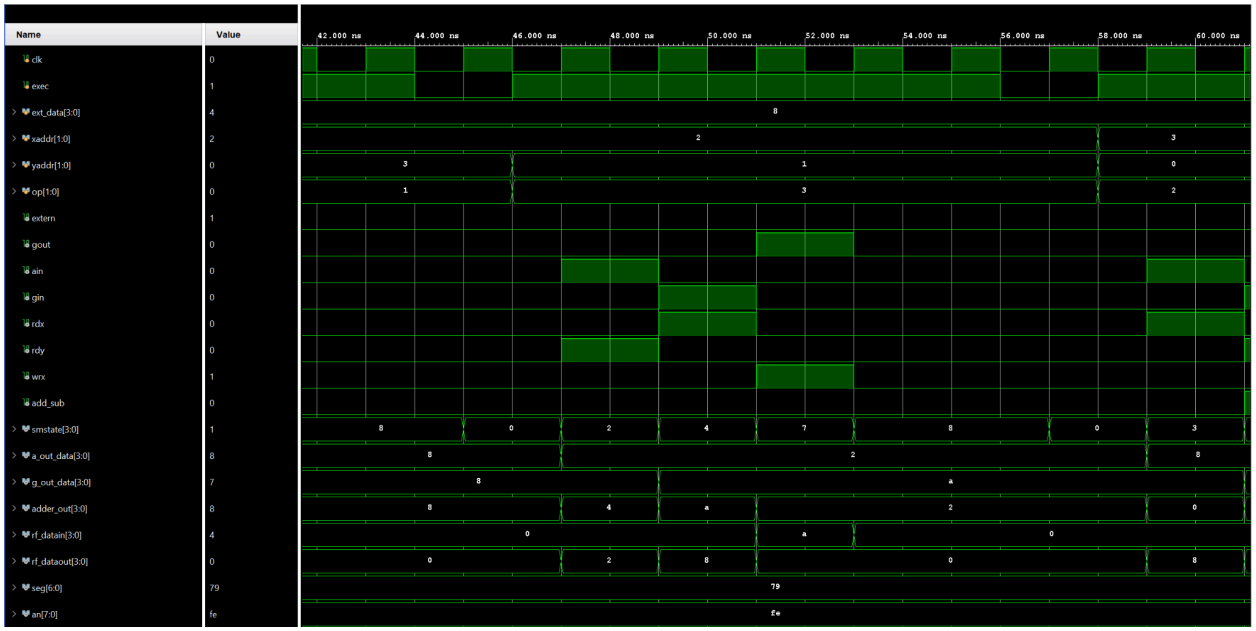
Load Simulation:



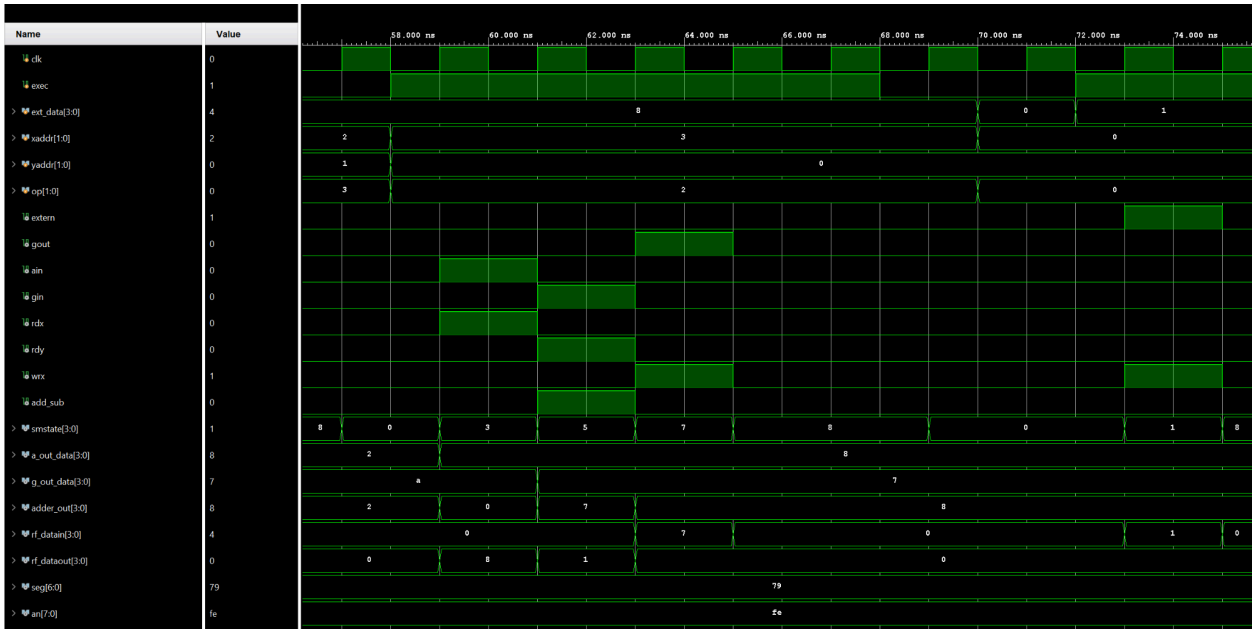
Move Simulation:



Add Simulation:



Sub Simulation:



Questions:

1. What problems did you encounter while implementing and testing your system?

Some of the simulations were incorrect and had wrong simulation results. To fix this, we changed the values of the variables.

2. Did any problems arise when demonstrating for the TA? In other words, did the TA ask you to demonstrate something that you did not think of yourself? What was the scenario that you were asked to demonstrate? Provide some thoughts about why you didn't think of this yourself.

We were asked to explain the inputs and outputs based on the simulation waveform. We did not think of this ourselves because our main focus was on the variables. With a quick rundown on how to read the waveform generators inputs and outputs, we were able to explain the add and subtract commands.

3. In this design, the value 0 is readily available by causing both read signals to be deasserted. If the design did not have that capability, what could you have done to implement the move operation? Note that the answer here will likely involve multiple operations, which is fine.

There could be multiple ways of doing this. The first is adding an AND gate with zero, from register 0. Since any value with an AND gate and a zero will give us 0, this will yield us a zero when we need to. Another method could be adding an XOR gate with itself since XOR only gives 1 when the two values are different. When we send in the same value in the XOR gate, it will give us an 0.