Dylan Thornburg and Ryan Kinris

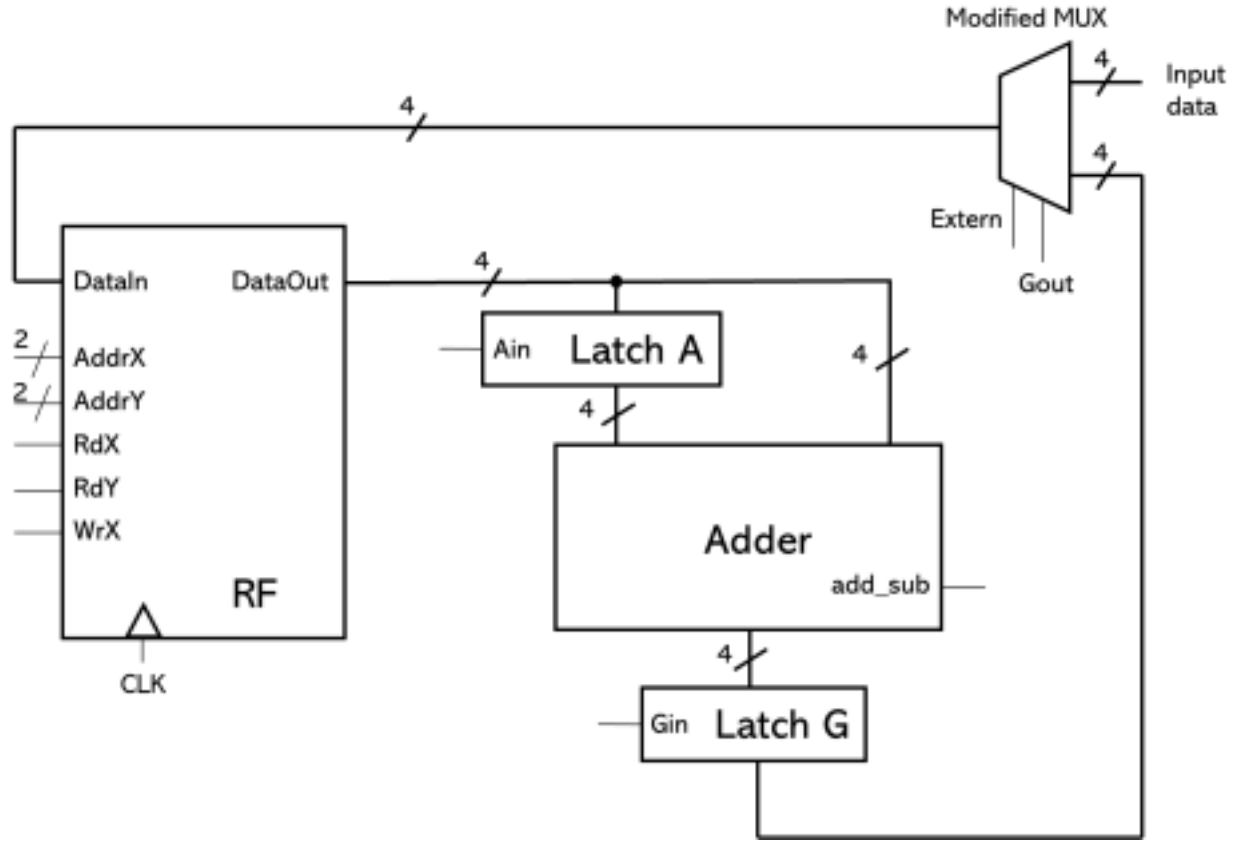
SANTA CLARA UNIVERSITY	ECEN 122 Winter 2024	Hoeseok Yang
Laboratory #2: State machine control of simple datapath **For lab section on January 23/26, 2024**		

I. OBJECTIVES

Given a simple CPU datapath and the definition of just a few operations, define and implement a state machine that correctly controls the datapath in response to any set of provided inputs.

PROBLEM STATEMENT

You will be given a datapath (register file, ALU, and associated connections) that has the ability to do simple addition and subtraction of 4-bit numbers. The datapath will come with a set of control signals that need to be asserted in the correct sequence in order for the operations to be executed properly. This sequencing will be provided by a state machine that you will design and implement based on the details provided below.



Explanation of the datapath

The diagram above is similar to what we have discussed in class, with a few notable distinctions. First, the data bus coming out of the register file (RF) is separate from the bus used to provide data into the register file. This bus will be driven solely by the RF. The *RdX* and *RdY* control signals will be used to drive the selected value contained in the register identified by the corresponding address. If *RdX* and *RdY* are either both 0 or both 1, the register file will drive the value 0 onto the bus. This difference in functionality will be important when we look into the details of how the move operation will work with this design.

The other notable distinction is a modified MUX in the upper righthand corner. Note that this modified MUX has two select signals: *Extern* and *Gout*. These signals will cause the corresponding input to pass through and thus be driven into the *DataIn* port of the RF. As with the read control signals in the RF, if *Extern* and *Gout* are either both 0 or both1, the output of this "MUX" will be 0. (See the provided source code, modified_mux.v.)

Explanation of the connections to the datapath

The address values supplied to the RF will come from the testbench (See l2_tb.v) you will use to test your design. The RF will contain just four registers, so there will be 2 bits for *AddrX*, and 2 bits for *AddrY*.

Similarly, the input data will come from the testbench.
For this lab we will be using just a 4-bit datapath, so all the data (not address) buses in the diagram are 4 bits wide.

The remaining signals in the diagram are all the control signals that will be coming from the state machine that you will design and implement.

Operations

We will implement the four operations that we have discussed in our simple datapath in class:

- A **load** operation, which causes the value provided via the external input bus to be written to register X, i.e., the register identified by *AddrX*.
- A **move** operation, which causes the value in register Y to be copied to register X, i.e., X = Y. • An **add** operation, which causes the value in register X to be updated by adding the value in register Y to it, i.e., X = X + Y.
- A **subtract** operation, which causes the value in register X to be updated by subtracting the value in register Y from it, i.e., X = X - Y.

We will use a 2-bit opcode, again coming from the testbench, to indicate which of the four operations should be performed.

Mapping operations to control signals

Let's walk through each of the four operations to make it clear which control signals need to be asserted and to understand any ordering/timing requirements that need to be in place.

- Load: The load operation needs to assert the *Extern* signal, which will allow the value on the external pins to be placed on the bus connected to the input of the RF, and the *WrX* signal, which will cause the value on the RF's data input (*DataIn*) to be written to the register specified by *AddrX*. These two signals can be (in fact, must be) asserted at the same time.

- Move: The sequencing of this operation is going to be different than what we have discussed in class. The reason being that, in this design, the output of the RF is not directly connected to the input of the RF. So, any data we read out of the RF will need to go thru the ALU to get to the bus connected to the input of the RF. How can we do that without changing the value? We can add (or subtract, it doesn't matter) 0. How can we get 0? The definition of this particular RF is such that if neither *RdX* or *RdY* is asserted then it will drive 0 on its output bus. We are going to have to stretch the execution of this operation across three cycles
    - First, we need to read out Y (assert *RdY*) and capture that in the staging latch labeled A (assert *Ain*).
    - Next, we need to "read" 0 (i.e., deassert *RdY*) and then capture the computed result in the staging register called G (assert *Gin*).
    - Finally, we need to get this value to pass through the mux that controls what shows up at the input to the RF (assert *Gout*) and cause this value to be written to register X (assert *WrX*).

[We are going to look at subtract before add, because the implementation of add leads to some choices that can be made.]

- Subtract: The timing of the subtract operation is similar to the move operation; it needs to be stretched over three cycles. Note that subtraction is not commutative; Y-X is not the same thing as X-Y. Given the hardware we have, and given that we're trying to compute X-Y, we need to read out X first (assert *RdX*) and capture the value in staging register A (assert Ain) and then read Y in the next cycle (assert *RdY*). We also need to make sure our subtract control signal to ALU is asserted and that the result is captured in staging register G (assert *Gin*). The last cycle is the same: assert Gout and *WrX*.

- Add: Again, the timing of the add operation is the same as both move and subtract. But a difference between add and subtract is that adding is a commutative operation; it doesn't matter whether we compute X+Y or Y+X. So when executing the add operation it doesn't matter whether we read X first or Y first, as long the value is captured in register A (assert Ain) and then we assert the other read control in the next cycle. However, we will find in the future that it will be useful to have consistency in the semantics for add and subtract. So you should implement your add operations such that X is read first and stored in staging register A, and then read Y to perform the addition operation.

Controlling execution

Given what has been described so far, we are going to be using a total of 10 bits of input from the testbench: 4 for the external data, 2 each for the "addresses" for registers X and Y, and 2 to indicate the desired operation (i.e., the opcode). In the testbench, it may be difficult to keep track of the timing of your inputs and when you

intend particular operations to start. So, we are going to use more control signal to manage this, which we will call *exec*.

The *exec* signal should cause the following behavior:

- When *exec* is 0, and your design will just sit idle (assuming it is not in the middle of some previous operation).
- When *exec* changes to a 1, your design will execute whatever you've instructed it to execute via the inputs from the testbench.
- Your design will ignore any further changes to the testbench inputs unless *exec* is changed back to 0, at which point asserting it to 1 will cause your design to execute the next specified operation.

State machine to properly sequence the control signals

Now that we have talked through and described how this simple CPU is intended to work, we need to get to the ultimate business at hand and design the state machine that is going to make sure that it functions as intended. We are going to implement this circuit in a Moore machine (l2_sm.v) with nine states as follows:

- IDLE: the state machine remains idle until you assert *exec*.

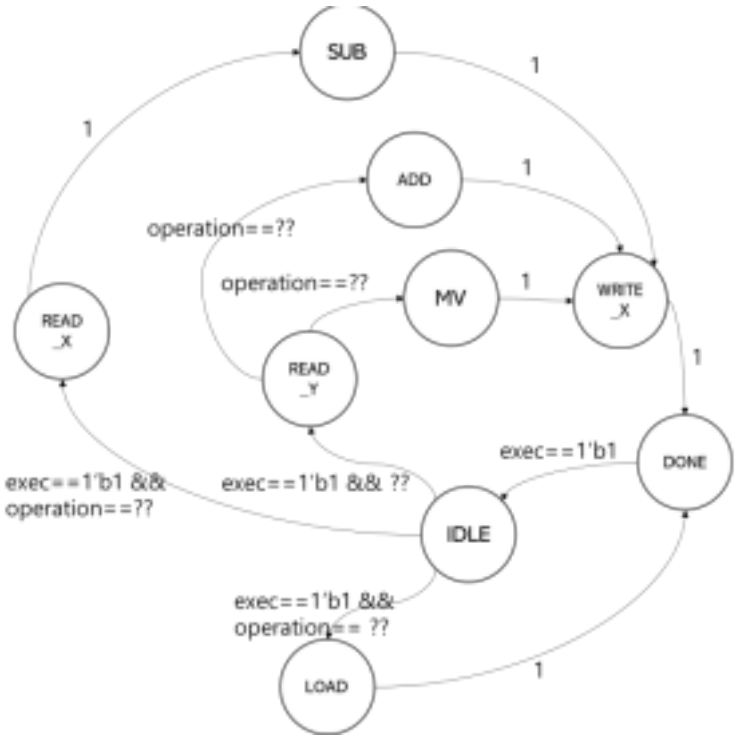
- LOAD: take the value from instruction and write it to the designated (by *AddrX*) register. • READ_Y: take the stored value from a register (addressed by *AddrY*) and store it in latch A. • READ_X: take the stored value from a register (addressed by *AddrX*) and store it in latch A. • ADD: perform addition (need to properly control *RdX/RdY* and *add_sub*) and store the result in latch G.
- SUB: perform subtraction (need to properly control *RdX/RdY* and *add_sub*) and store the result in latch G.
- MV: take the value stored in latch A and store it in latch G, bypassing the adder (adding 0). • WRITE_X: write the value stored in latch G into a register (addressed by *AddrX*). • DONE: do nothing and go back to IDLE when *exec* is deasserted.

We assume the following encoding for opcode (operation in l2_SM.v):

- 0 (2'b00): load
- 1 (2'b01): move
- 2 (2'b10): subtract
- 3 (2'b11): add

II. PRE-LAB

(i) Complete the given Moore machine that will properly execute these operations given the datapath described for this lab. Because there are a number of outputs, if you were to fully specify the 0 or 1 value of every output on every arc, or in every state, that would lead to a very busy state diagram. So only specify the control signal (or signals, if there are more than 1) that needs to assert under a given set of conditions. The absence of listing a signal will imply that it is not asserted, i.e., it's 0.



(ii) The control logic specified in a Moore machine above, has two inputs (*operation* and *exec*) and eight outputs (*RdX*, *RdY*, *WrX*, *Ain*, *Gin*, *Extern*, *Gout*, and *add_sub*). Based on the state definitions given above, complete the following truth table. (Completed during lab report)

State	*RdX*	*RdY*	*WrX*	*Ain*	*Gin*	*Extern*	*Gout*	*add_sub*
IDLE	0	0	0	0	0	0	0	0
LOAD	0	0	1	0	0	1	0	0
READ_Y	0	1	0	1	0	0	0	0
READ_X	1	0	0	1	0	0	0	0
ADD	1	0	0	0	1	0	0	0
SUB	0	1	0	0	1	0	0	1
MV	0	0	0	0	1	0	0	0
WRITE_X	0	0	1	0	0	0	1	0
DONE	0	0	0	0	0	0	0	0

III. LAB PROCEDURE

(i) The first thing to do with your lab partner is reconcile your pre-labs. Compare your state diagrams and truth tables. If they are somewhat different, discuss with each other what made the differences and make a decision between the two of you about which state diagram you will try to implement.

(ii) Since only one person can really type up a Verilog module, one of you should get started on coding up the state machine that you have decided to pursue. You are to fill in two always statements in the given skeleton (l2_sm.v). The first one is about state transitions while the other is to implement the truth table you completed in your pre-lab.[1]

(iii)In the meantime, the other person can bring up Vivado and start pulling in the other pieces of the system, which the TA will make available to you. Make all connections that are not dependent on the state machine being instantiated (in l2_tb.v).[2]

(iv)Once the state machine is coded, import this into Vivado and finish making the connections.

(v) The TA will provide a testbench, which will cause the design to be simulated across a number of cycles and will cause each of the four operations to exercised. After running the simulation, examine the resulting waveforms to see if you can determine if your design is functioning properly.

(vi)Once you have determined that your design appears to be working properly, demonstrate to the TA.

[1] See pages 34-38 of the Verilog reference for the syntax of *always* and *case* statements,

[2] See pages 29 and 30 for of the Verilog reference for connecting Verilog modules in a structural specification.

IV. REPORT

For your lab report, include the source code for your working state machine. In addition, include answers to the following questions.

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

• 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.

CODE (only l2_sm is included because it is the only file we changed):

```
module l2_SM(input clk,
             input execute,
             //input [3:0] input_data (part of instruction),
             //input [1:0] regXaddr (part of instruction),
```

```
                //input [1:0] regyaddr (part of instruction),
                input [1:0] operation, // opcode (part of instruction)
                output reg _Extern,
                output reg Gout,
                output reg Ain,
                output reg Gin,
                //output reg [1:0] AddrX,
                //output reg [1:0] AddrY,
                output reg RdX,
                output reg RdY,
                output reg WrX,
                output reg add_sub,
                output [3:0] cur_state);

//defining all my states - 8 total
parameter IDLE          =  4'b0000;
parameter LOAD          =  4'b0001;
parameter READ_Y   =  4'b0010;
parameter READ_X   =  4'b0011;
parameter ADD           =  4'b0100;
parameter SUB           =  4'b0101;
parameter MV            =  4'b0110;
parameter WRITE_X =  4'b0111;
parameter DONE          =  4'b1000;

reg [3:0] state = IDLE; // initial state being IDLE

assign cur_state = state;

initial begin //instead of reset
 state <= IDLE;
end


/* 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;
```

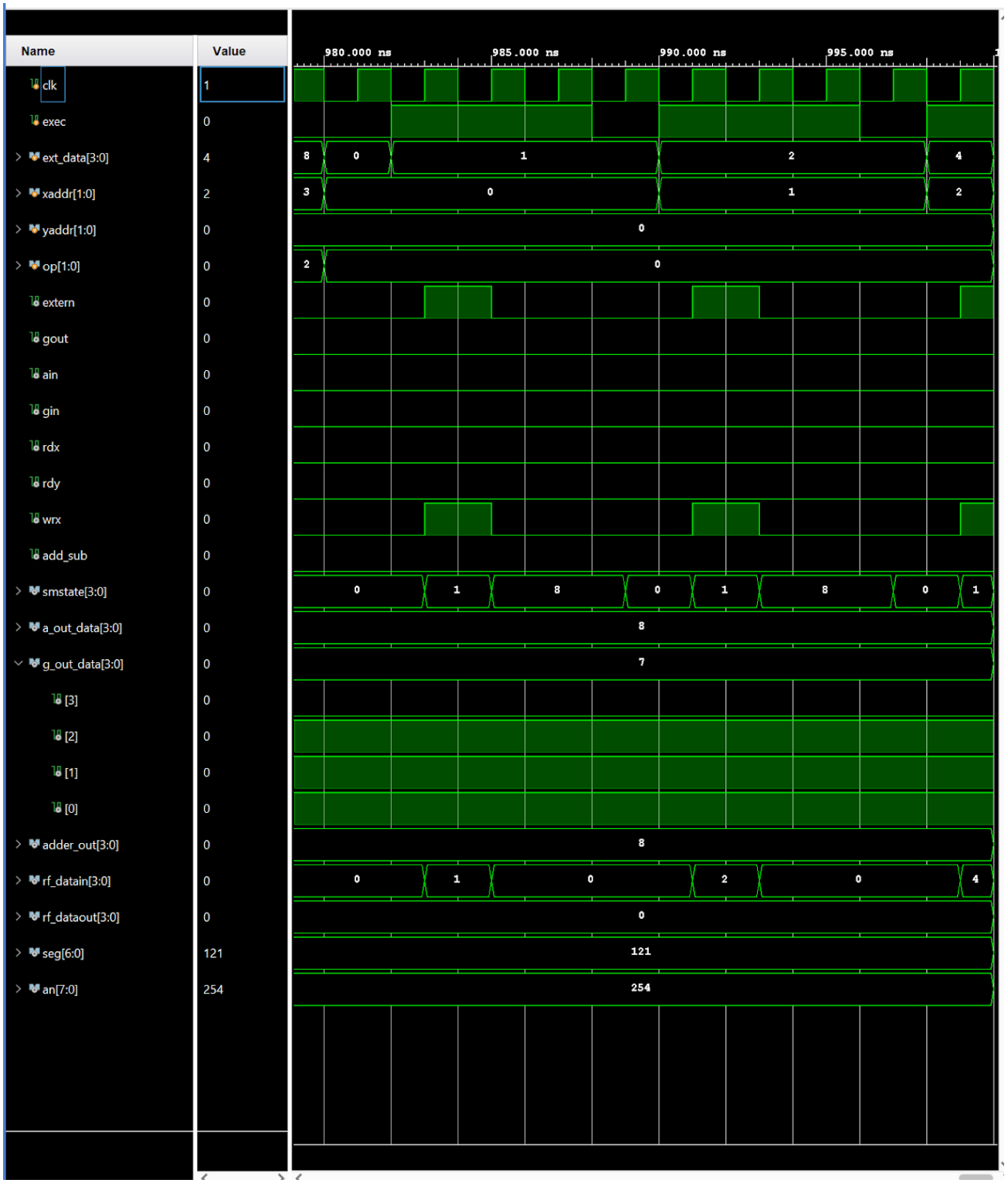
```
        Ain = 1'b0;
        Gin = 1'b1;
        RdX = 1'b1;
        RdY = 1'b0;
        WrX = 1'b0;
        add_sub = 1'b0;
    end
SUB:
    begin
                    _Extern = 1'b0;
        Gout = 1'b0;
        Ain = 1'b0;
        Gin = 1'b1;
        RdX = 1'b0;
        RdY = 1'b1;
        WrX =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

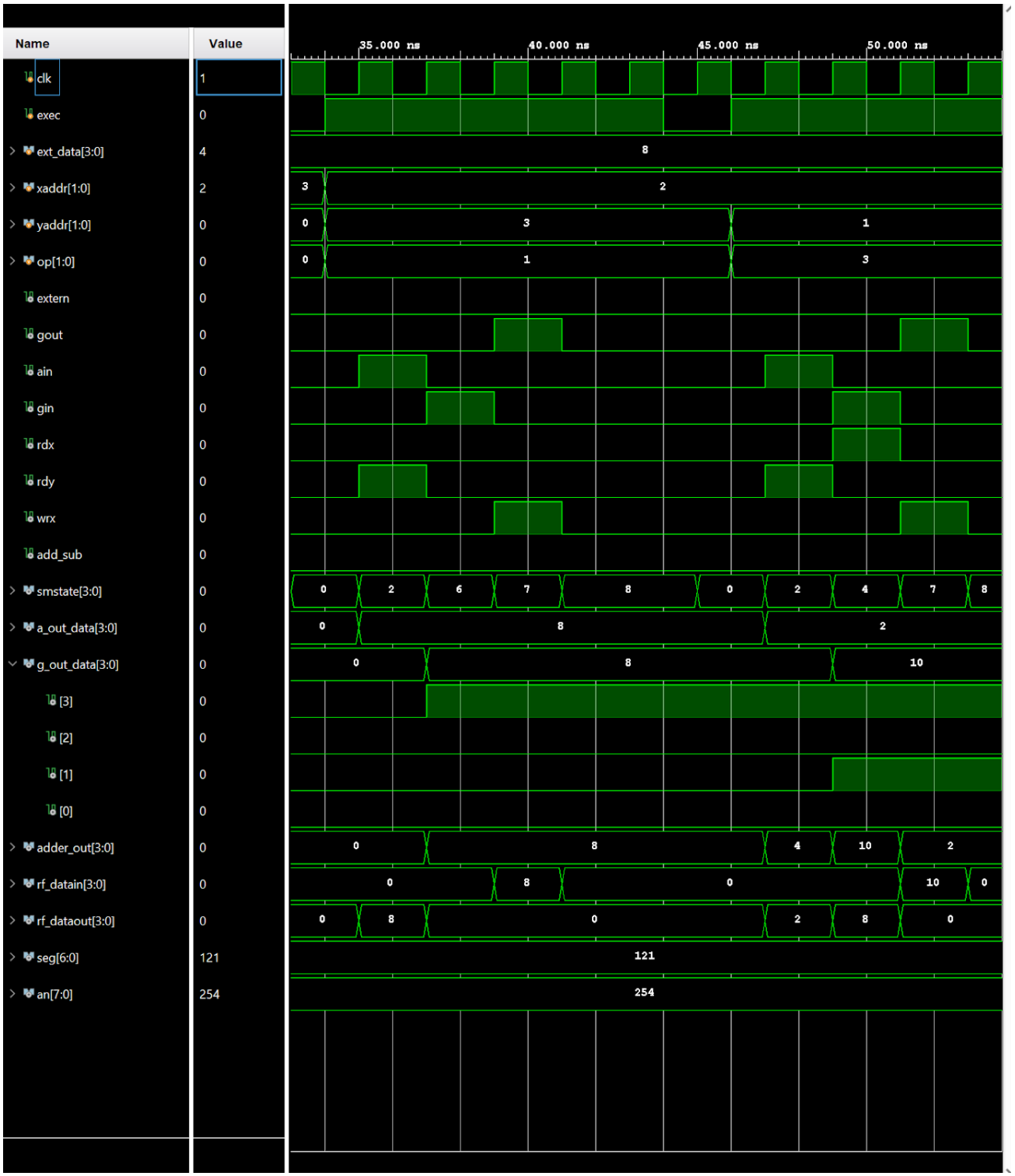
```
            Gin = 1'b0;
            RdX = 1'b0;
            RdY = 1'b0;
            WrX = 1'b0;
            add_sub = 1'b0;
        end
    endcase
end

/* 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(operation == 1) state <= MV;
        if(operation == 3) state <= ADD;
                    end

            READ_X: begin
                        state <= SUB;
                    end

            ADD: begin
                        state <= WRITE_X;

                end

            SUB: begin
```

```
                state <= WRITE_X;


            end

        MV: begin
                state <= WRITE_X;

            end

        WRITE_X: begin
                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


endmodule

/*

encodings
00 - load
01 - move
10 - subtract
11 - add

*/
```

Screenshots:

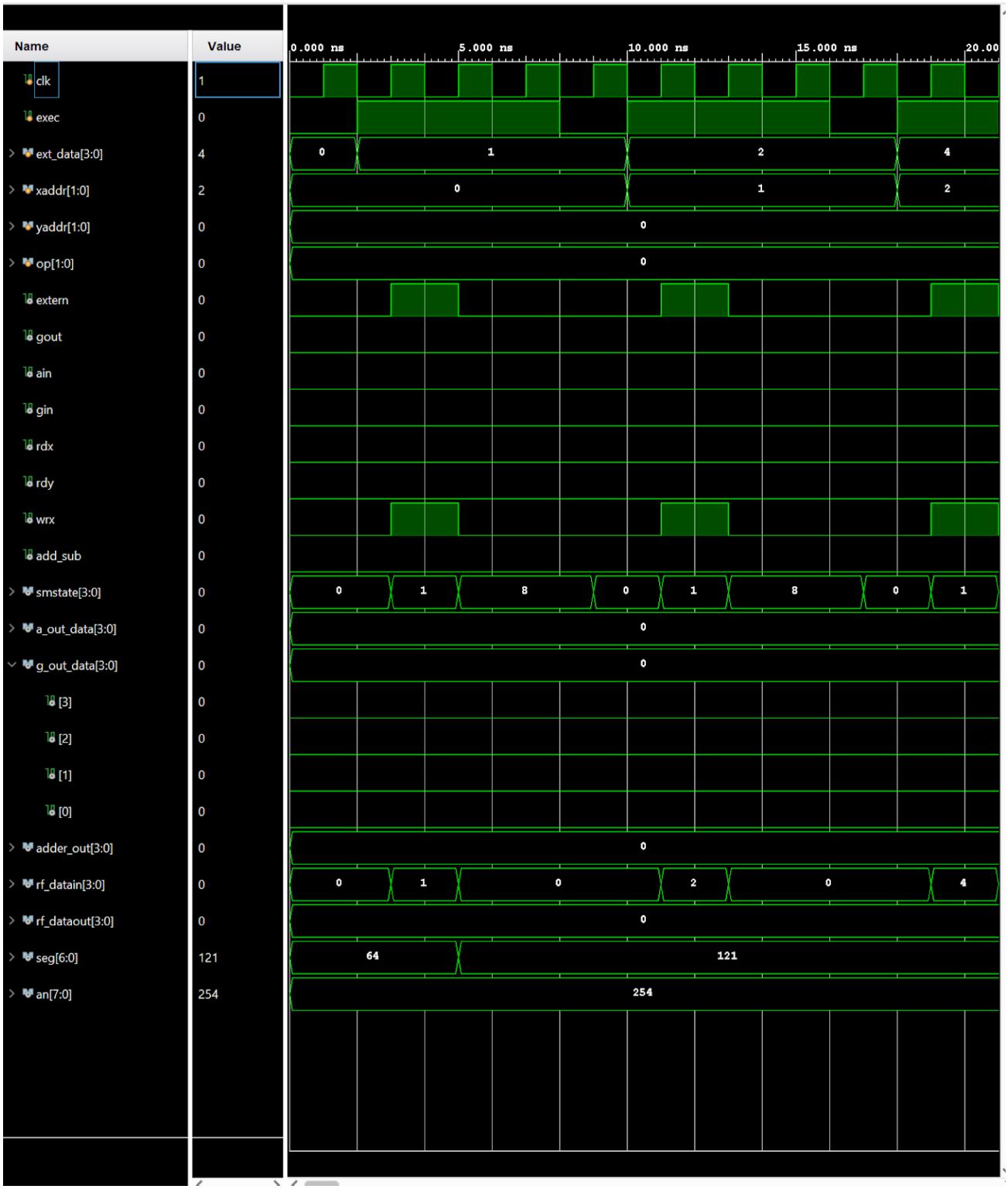
End of simulation (g_out_data = 7)

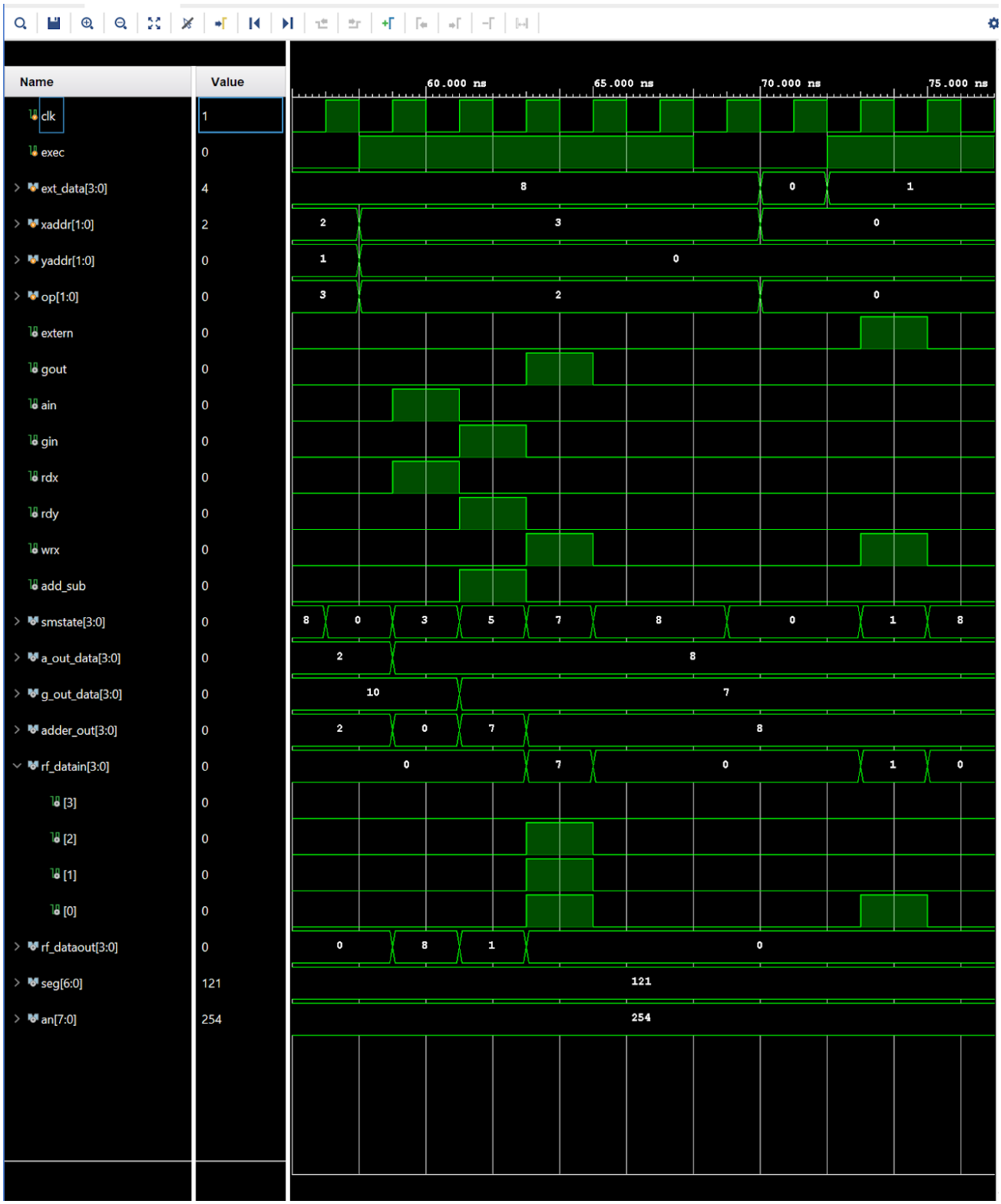




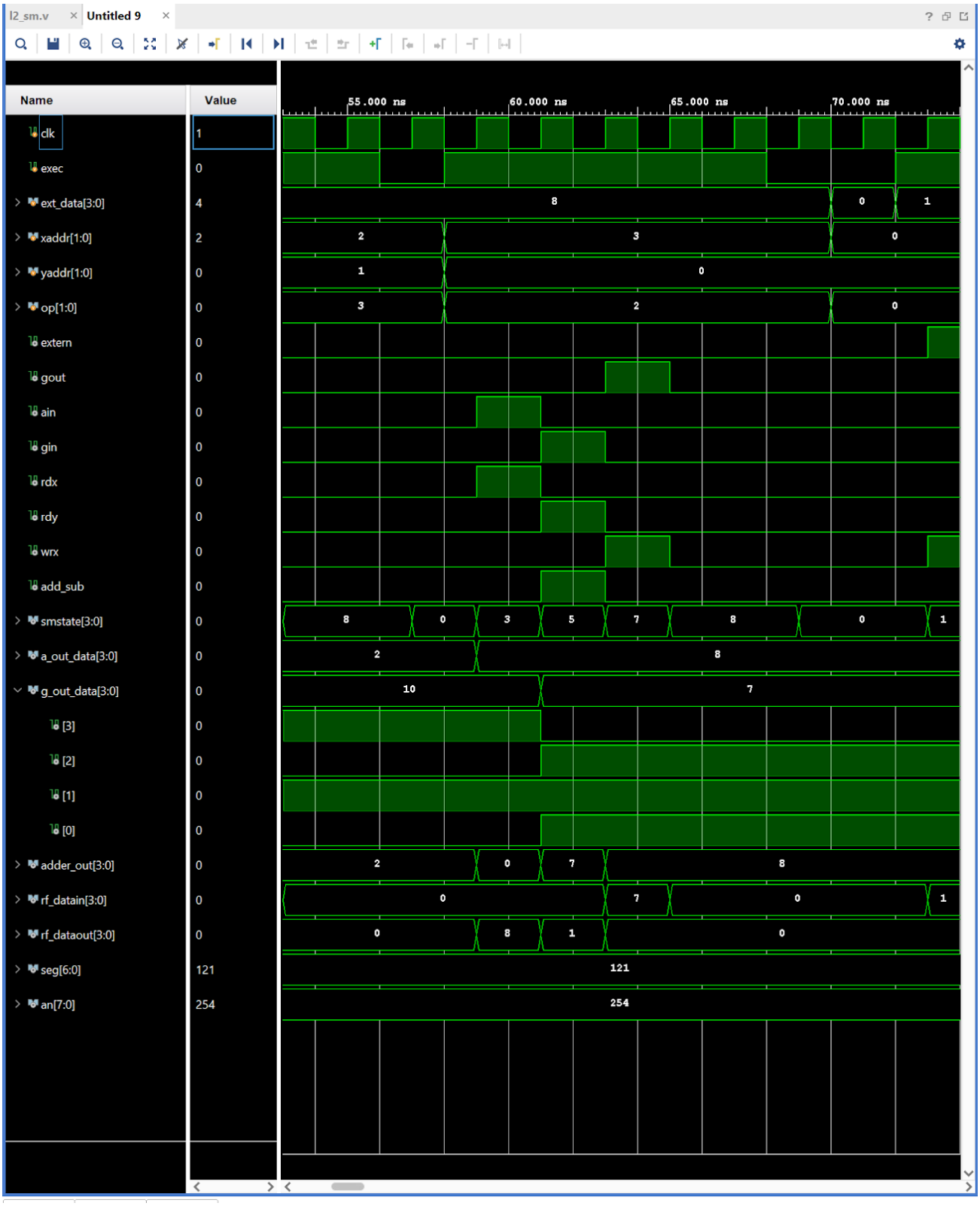
Extra screenshot 1 (move and add) ^^^

Extra screenshot 2 (load)





64 nano-second screenshot (rf_datain = 7) ^^^^



Extra screenshot 3 (subtract)^^^

Questions:

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

We had some issues regarding the logic of the states. We fixed this by examining the waveforms and changing our bit values for each state.

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.

When demonstrating to the TA we came to no roadblocks/challenges with our work, but one thing we did not think about was the register value for register 3. This was something that we did not expect to look for when trying to run the program as nothing in the lab document suggested that's what would be tested specifically. The value was correct regardless.

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.

We will implement an always 0 input (by grounding and calling it the 0 register) that gets switched into the adder before the MV state. This way when we want to move in a value using the adder, we do not need AddrX and Addry to be zero. This would perform similarly to the x0 register in RISC-V software.