Laboratory Exercise B
Processors

TCES 330 Digital Systems Design
6/1/2016

Authors: Vladislav Psarev, Brandon Watt and Andrew Gates

Table of Contents

Laboratory Assignment B

The purpose of this lab is to understand how to implement a programmable processor. By using modules that work as a Processor, Control Unit, Data Path, Data RAM, Register RAM, Instruction ROM, and an ALU we were able to accomplish this. The processor will take in 16-bit input from a memory file and perform a corresponding operation with that input. The possible operations are load, store, add, subtract, halt, and no operation (NOOP). The processor will output to all eight of the seven segment displays. The seven segment displays can be changed depending on the value of switches 17-15, they can display ALU A side, ALU B side, ALU output, or the next state.

Requirements:

The objectives of the lab assignment are described in this section of the report.

We were given button sync and key conditioner modules to resolve the button bounce issues, as well as the ALU. All the register files for the RAM and ROM were constructed using the LPMs. For dataROM.mif we had to decode the instructions, the decoding can be see in figure 7. Key two acts as the system clock, going through the button sync and key conditioner modules, and key one is a synchronous reset. Switches 17 through 15 control what HEX 7 - 4 of the seven segment displays show. HEX 3 - 0 show the contents of the instruction register.

- Implement the six-instruction programmable processor described in Processor16.pptx using Verilog.
- Your Quartus project should be in a folder called LabB. Your top-level module will instantiate your processor module and interface to the DE2 board as follows:
    - KEY[2] acts as your system clock; (so we don't wear out KEY[0]
    - KEY[1] acts as a synchronous system reset;
        - Note that reset will not be able reload memory contents, but it should reinitialize your state machine and reset (clear) all your registers
    - Several internal variables are brought to the top-level for debug and display purposes. These include the program counter, the instruction register, state machine current state, both inputs to the ALU, and the ALU output.
    - HEX3, 2, 1, and 0 always display the current contents of the IR;
    - SW[17:15] determines what HEX 7, 6, 5, and 4 display as follows:
        - 0 => HEX7, HEX6 = PC; HEX5, HEX4 = Current State; Note: for this to correspond to the values you set for your states, you should tell Quartus to use your state machine encoding.
        - 1 => HEX7, 6, 5, 4 = ALU_A (A-side input to ALU)
        - 2 => HEX7, 6, 5, 4 = ALU_B (B-side input to ALU)

        - 3 => HEX7, 6, 5, 4 = ALU_Out (ALU output)
        - 4 => Next State (FSM next state variable, if you use one)
        - 5 Unused
        - 6 Unused
        - 7 Unused
    - Your top level module (LabB) should instantiate a processor module, a multiplexer for selecting displays and a key conditioner (described below). Your top level RTL view should look very close to Figure 1.
- I will provide a high-level ModelSim testbench, so the Processor module you turn in must use this signature (if your processor does not conform to this signature, change your processor, not my testbench):

```
module Processor( Clk, Reset, IR_Out, PC_Out, State,
NextState,  ALU_A, ALU_B, ALU_Out );
     input Clk; // processor clock
     input Reset; // system reset
     output [15:0] IR_Out; // Instruction register
     output [4:0] PC_Out; // Program counter
     output [3:0] State; // FSM current state
     output [4:0] NextState; // FSM next state (or 0 if you
     don't use one)
     output [15:0] ALU_A; // ALU A-Side Input
     output [15:0] ALU_B; // ALU B-Side Input
     output [15:0] ALU_Out; // ALU current output
```

  Feel free to add debug outputs to your processor module in the unused slots, but make sure the outputs listed above operate as specified. Also, you will need to take those variables out when you turn in your project so that your processors uses the signature shown above.

- Turn in the following sample program compiled and loaded into Instruction Memory:

  RF[0] = D[0B] - D[1B] + D[06] - D[8A];
  D[CD] = RF[0];
  HALT

  Data memory should initially contain
  D[6] = 0x10AC
  D[B] = 0xCC05

D[1B] = 0x01B5
D[8A] = 0xA040

- Make sure that the In System Memory Content Editor can display the contents of your memories instruction memory and your data memory; it cannot be used on your register file (dual ported) memory.
- Make sure Quartus recognizes your processor state machine (as a state machine). Note that there will be a second state machine in your Quartus design: the ButtonSync state machine. In Quartus State Machine View you can select which state machine to view in the upper left corner of the state machine window.
- Data memory should be a 256 X 16 Quartus RAM LPM with Memory Content Editor enabled.
- The Register File should be a 2-Port RAM as discussed in RegisterFile.pptx. Note: you cannot use the In System Memory Content Editor with 2-Port devices.
- The ALU is as discussed in ALU.pptx and the Verilog is provided for you.
- The Instruction memory should be a 128 X 16 Quartus ROM LPM with Memory Content Editor enabled.
- The controller state machine should be similar to the one shown in Processor16.pptx.
- Run TimeQuest and include the sdc file in your Quartus project. Note that there will be two clocks in this project (the 50 MHz clock for the key conditioning circuits and the processor clock derived from the KEY). Refer to KeyConditioner.pptx.
- I will provide a ModelSim testbench. Run this testbench and include a screenshot of the ModelSim Transcript area where the results are printed in your report. Of course you can also include the waveforms in your report, if you wish along with other tests that you might have used.
- Write your report using the same outline as before. You know by now to be very explicit and detailed about your test procedure and test results. Photographs are allowed!

Design:

This section of the report describes our analysis of the requirements for this laboratory exercise and the resulting project design.

The processor was designed using the layout shown in figure 10, the processor contains two separate modules, the Datapath and Control Unit. The datapath module is composed of the data RAM, register RAM and Arithmetic Logic Unit. Both the RAM units are created using the LPMs and the ALU is given to us. The datapath module simply requires that the RAMs and ALU are connected properly, how they should be connected can be seen in figure 6. The Control Unit contains the instruction memory, program counter, instruction register and statemachine.

The instruction memory is a ROM module made by the LPMs, the program counter is just a counter tracking the number of programs loaded, the instruction register is a register that holds the current instruction and the state machine sends out the necessary control signals. The state machine should have at least 9 states: Initial, Fetch, Decode, Load, Store, Add, Subtract, Halt and NOOP. The state diagram of the Control Unit state machine can be seen in figure 5. For our state machine we simplified it by creating an Arithmetic state that handled both the add and subtract operations. The Arithmetic state would check the four most significant bits of the instruction and send the corresponding instruction and data to the ALU.

Test Procedures:

The following test procedures will be used to verify this laboratory exercise satisfies the requirements given in the Requirements section, above.

ModelSim Test:

1. Set up ModelSim inside of Quartus using the given processor test file.
2. Run the test and observe the output.
3. Compare the results to the expected output given. The states don't have to match but all other values should.
4. This concludes the ModelSim test.

On Board Test:

1. Check the instructions mif file and be sure to understand it's instructions.
2. Load the program onto the board.
3. Press KEY[2] until the first instruction appears on HEX 3-0.
4. Compare the value displayed on HEX 3-0 to the expected instruction.
5. Go through inputs 0-4 on switches 17-15 and compare each of the outputs on the HEX 7-4 to the values you expect.
6. Press KEY[2].
7. Repeat steps 4 through 6, until you reach the halt instruction.
8. Hold KEY[1] and press KEY[2] until the program resets to the initial instruction.
9. Walk through the instructions again and occasionally hold KEY[1] and press KEY[2] to make sure the reset works.
10. This concludes the on board test.

Test Results:

The following test results were what we achieved in the lab.

ModelSim Test:

1. The simulation using ModelSim produced the output shown in Figure 9, in Appendix A. Our output had the same Time, ALU A, ALU B and ALU Out outputs as the ModelSim output that was provided to us. Only a few of the State outputs were different, which was acceptable. Our final non-zero ALU Out output was 3ABC, similar to the provided ModelSim Output.

On Board Test:

1. Our instructions mif file had the correct encoded instructions and it performs the instructions given in the procedure. Our encoding logic for the instructions given can be found in Figure 7 of Appendix A.
2. Our program compiled successfully and was also successfully loaded onto the board. The RTL views of our program as well as the state machine transition diagram can be found in Figures 1-6 of Appendix A. Figure 8 shows the contents of the Instruction Memory ROM that was loaded onto the board.
3. Pressing KEY[2] once, the first instruction appears on HEX 3-0, which is 0x20B1
4. This first displayed instruction is also the first instruction in the Instruction Memory ROM, which is what we expected.
5. Binary 0 on switches 17-15 causes HEX 7-6 to display PC, which is 0, and is what we expected, since it is pointing to our first instruction.. HEX 5-4 displays the current state, which is 2, and is what we expected because we are currently fetching an instruction. Binary 1 on switches 17-15 causes HEX 7-4 to display ALU A, which is 0, and is what we expected since the current state is fetch. Binary 2 on switches 17-15 causes HEX 7-4 to display ALU B, which is 0, and is also what we expected for the same reason. Binary 3 on switches 17-15 causes HEX 7-4 to display ALU Out, which is 0, and is what we expected since the current state is fetch. Finally, binary 4 on switches 17-15 causes HEX 7-6 to display our next state, which is 5, and is what we expected since our next state will be LOAD_A, as the current instruction being read is a load instruction.
6. Pressing KEY[2] induces a clock pulse, moves us to the next state and increments PC.
7. Repeating steps 4 through 6 of our test until we reached the HALT instruction, all of the outputs on the HEX displays produced outputs which we had expected. The final contents of the Data Memory RAM can be seen in Figure 11. Note that Memory Address CD contains the value 0x3ABC.
8. Holding KEY[1] and pressing KEY[2] twice reset PC to 0. Releasing KEY[1] and pressing KEY[2] twice caused the current contents of IR, which HEX 3-0 displays, to be the initial instruction 0x20B1.
9. Walking through the instructions again and occasionally holding KEY[1] and pressing KEY[2], PC was set to 0, produced expected results and showed that the reset function works.

10. This concludes the on board test.

Observations:

The following observations are points of interest to be noted in the lab.

Some trouble we had when testing the processor in ModelSim is that we didn't initialize the state and next state values in the state machine. This was a bit odd because while the State and NextState values started out as unknowns (xx) they became known values (00), but the simulation stopped after the first instruction finished. It's possible that after the instruction finished ModelSim return the state and next state values to unknowns and that confused the simulation. Once the values were properly initialized the issue was resolved.

When we encoded the required instructions, we made the mistake of converting the last load instruction into its incorrect hex representation, which was also written to the mif file. There were no issues when we tested our processor on the board, but there were issues when we compared our ModelSim output to the ModelSim output that we were given. Our final hex number that was stored to data memory location CD was not the correct number because our final load instruction loaded from a memory location that contained the value zero. After we had discovered our mistake, we corrected the hex representation so that it would load the value stored in the memory location 8A. Our ModelSim output was correct after we fixed our mistake.

Conclusion:

In this lab we went over how to build a simple processor. This processor contains a Control Unit and a Datapath. The Control Unit contains the Instruction Memory (ROM), a PC register, an Instruction Register, and a State Machine. This State Machine determines what happens in the Datapath using selection signals, Register Addresses, and Data Memory Addresses. The Datapath contains the Data Memory (RAM), the Register File (2-port RAM) and finally an ALU that performs a few simple tasks, such as adding and subtracting two numbers. Our Processor was controlled with a manual Clock input assigned to KEY[2] and a reset was assigned to KEY[1]. Various outputs including ALU inputs and outputs, PC, Current State and Next State were displayed on hex displays HEX 7-0 with switches 17-15 as the selection switches used to determine which output was displayed. This lab was a great way to introduce us to how processors work by designing a simple processor. This lab also showed us how the different parts of the processor communicate with each other in order to complete a task.

Appendix A:

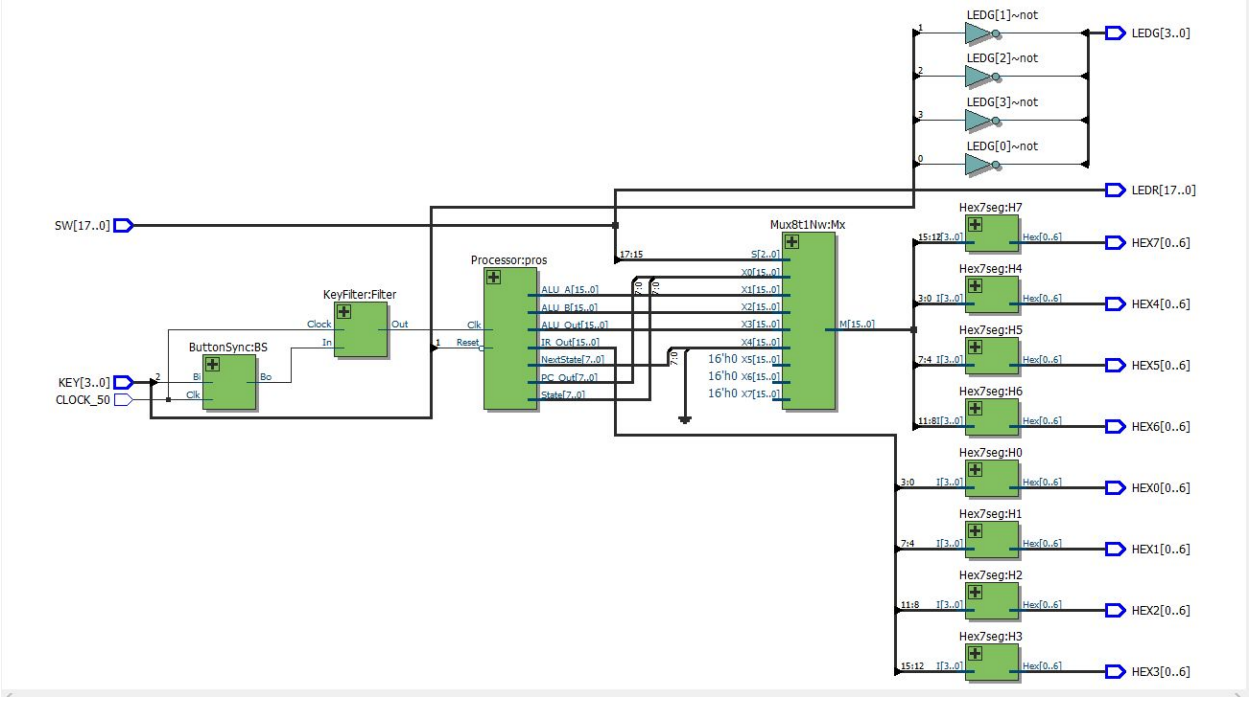

Figure 1. Top Level (LabB) RTL View

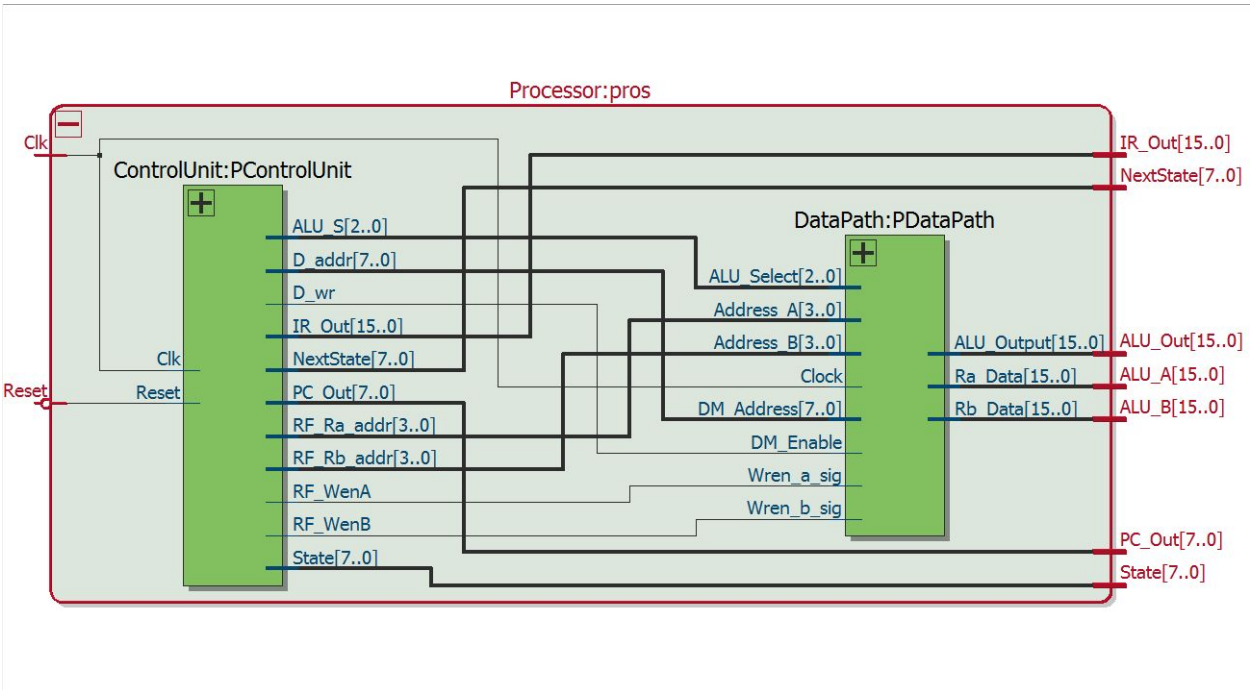

Figure 2. Processor RTL View

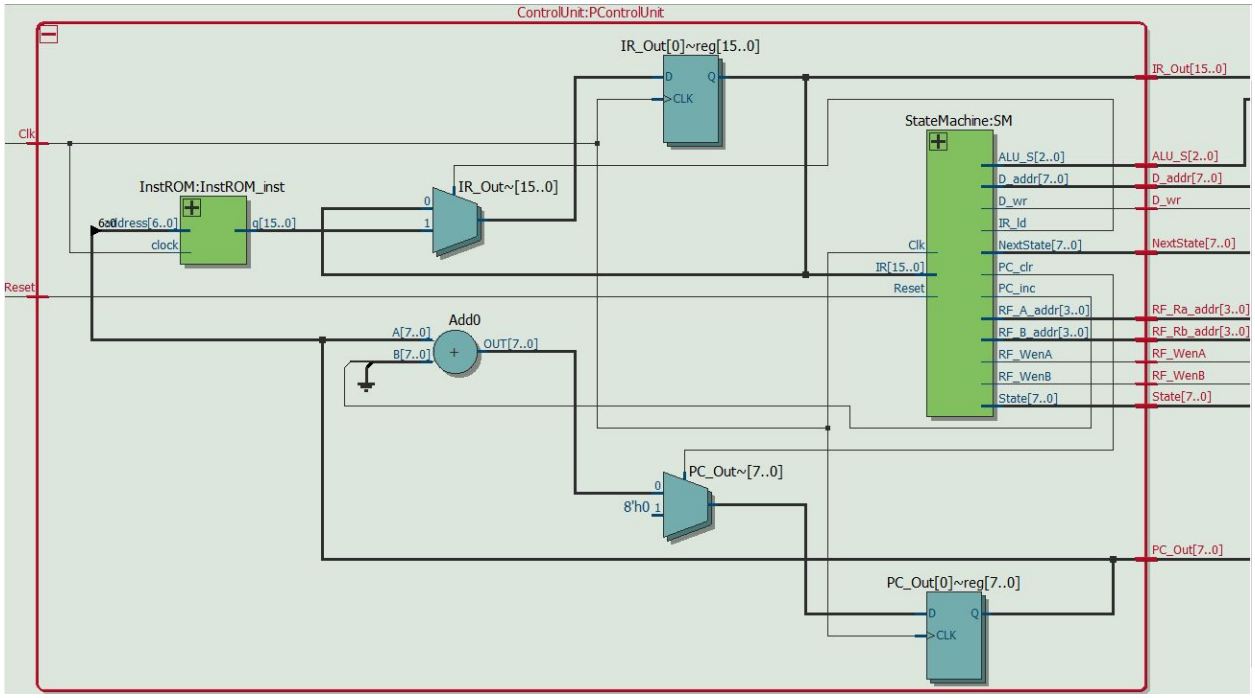

Figure 3. Control Unit RTL View

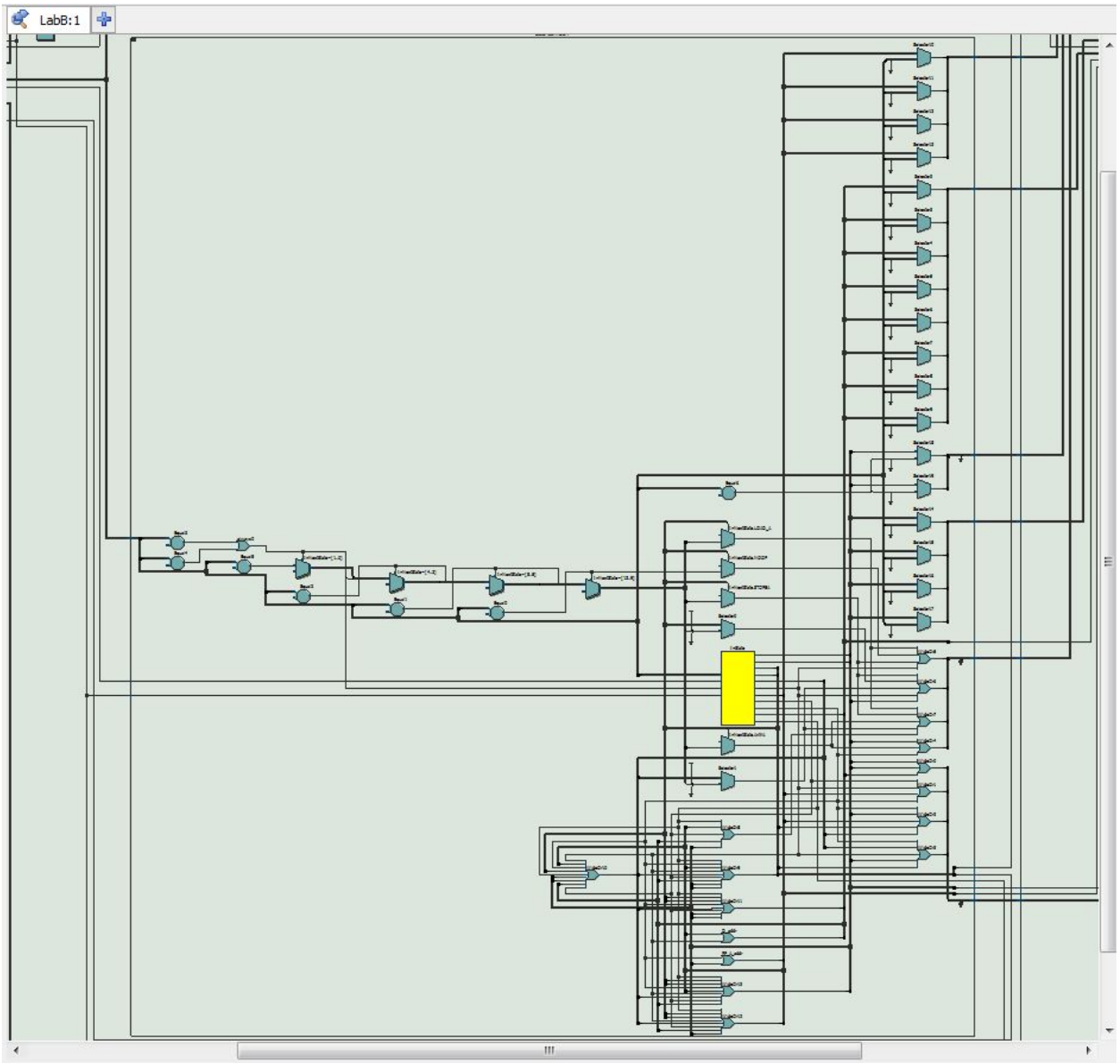

Figure 4. State Machine RTL View

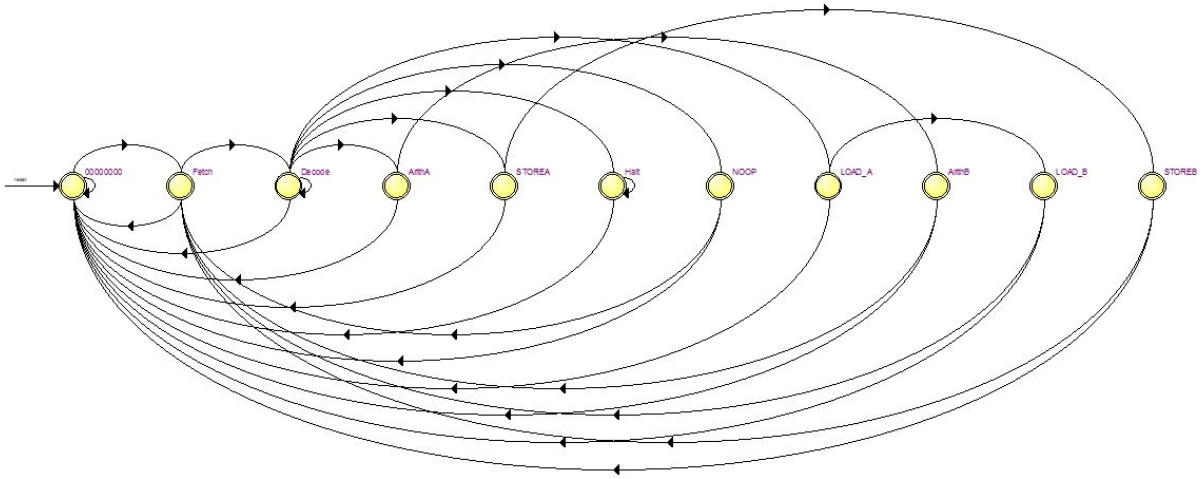

Figure 5. State Machine in State Machine Viewer

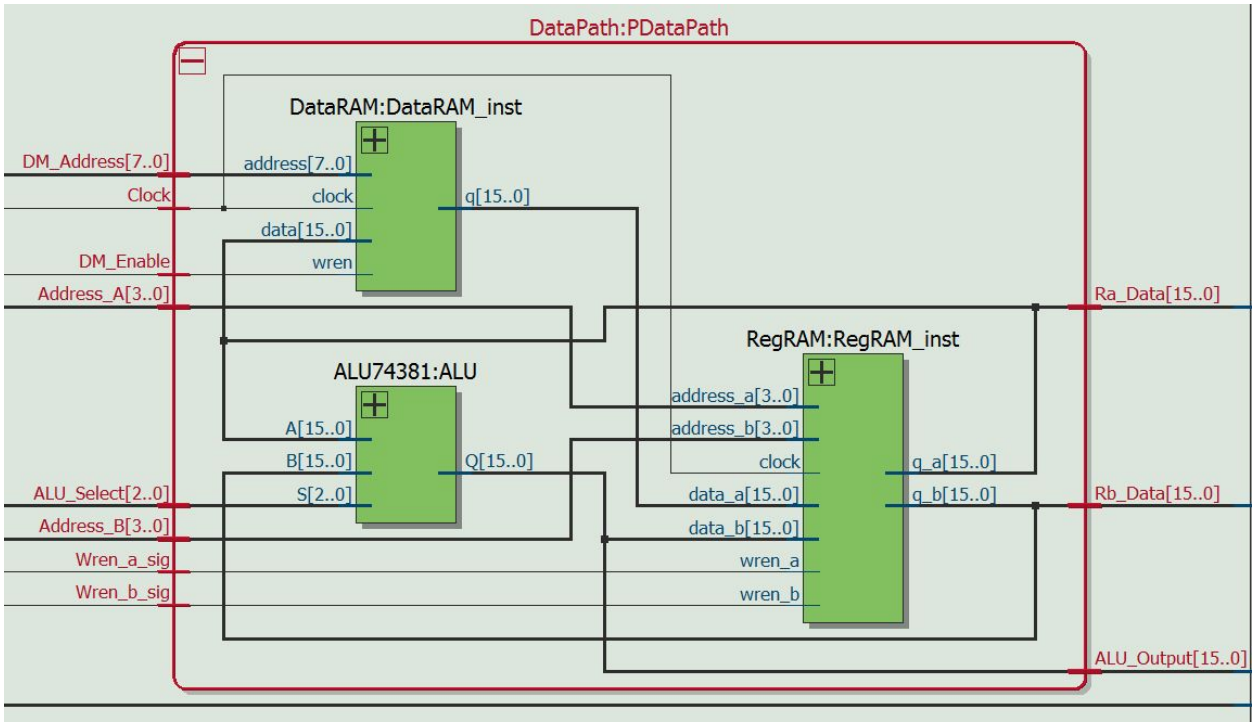

Figure 6. Data Path RTL View

RF[0] = D[0B] - D[1B] + D[06] - D[8A];

D[CD] = RF[0];

HALT

Load D[0B] into R1	0010 0000 1011 0001	0x20B1
Load D[1B] into R2	0010 0001 1011 0010	0x21B2
Sub R1 from R2 and save it into R1	0100 0001 0010 0001	0x4121
Load D[06] into R2	0010 0000 0110 0010	0x2062
Add R1 to R2 and save it into R1	0011 0001 0010 0001	0x3121
Load D[8A] into R2	0010 1000 1010 0010	0x28A2
Sub R1 from R2 and save it into R0	0100 0001 0010 0000	0x4120
Store R0 in D[CD]	0001 0000 1100 1101	0x10CD
HALT	0101 0000 0000 0000	0x5000

- General form : **operation source destination**

NOOP instruction – **0000 0000 0000 0000**

STORE instruction – **0001** $r_3r_2r_1r_0$ $d_7d_6d_5d_4d_3d_2d_1d_0$

LOAD instruction – **0010** $d_7d_6d_5d_4d_3d_2d_1d_0$ $r_3r_2r_1r_0$

ADD instruction – **0011** $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$

SUBTRACT instruction – **0100** $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$

HALT instruction – **0101 0000 0000 0000**

'r's are Register File locations (4 bits each)
'd's are Data Memory locations (8 bits each)

Figure 7. Instruction Encoding

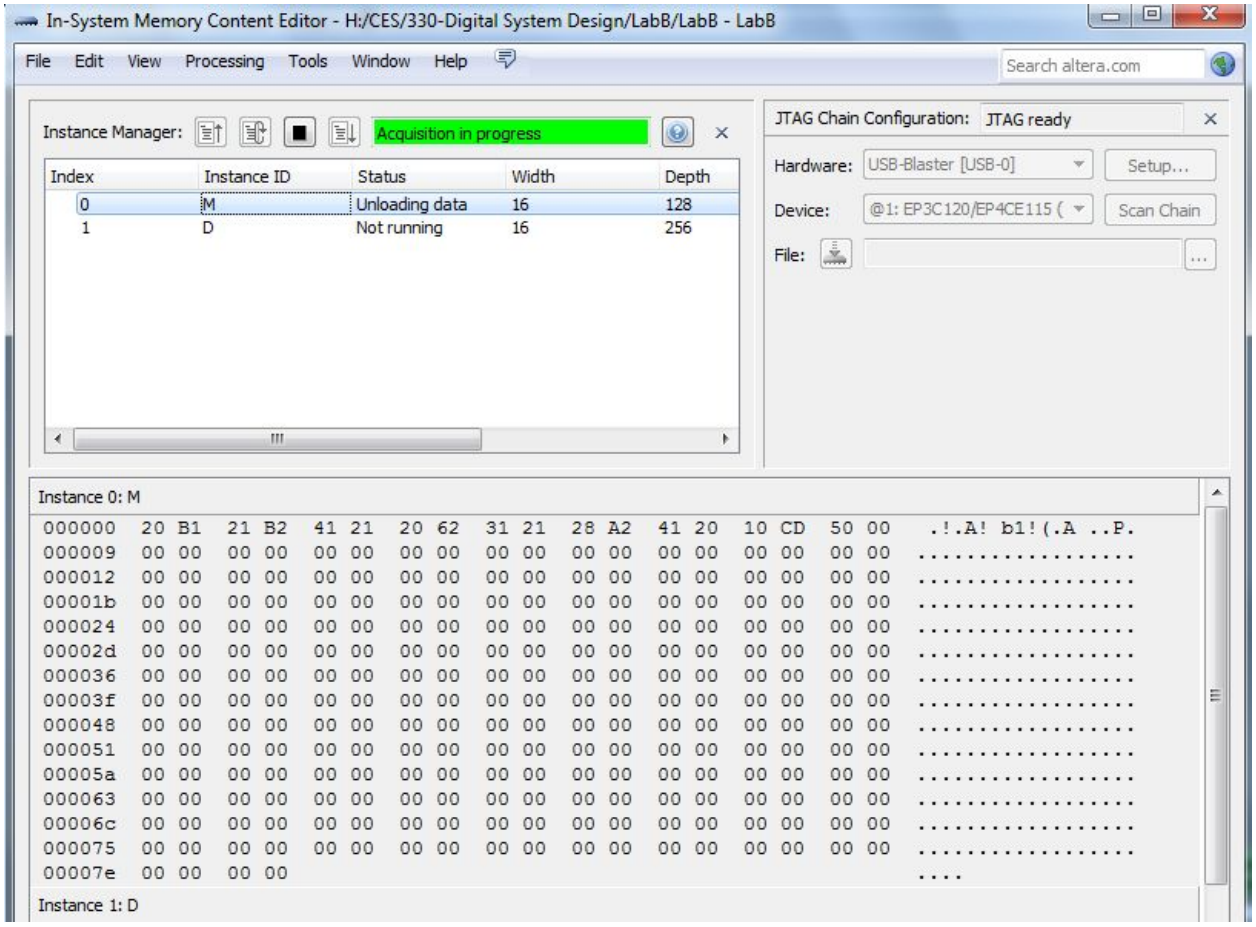

Figure 8. In-System Memory Content Editor - Instruction Memory

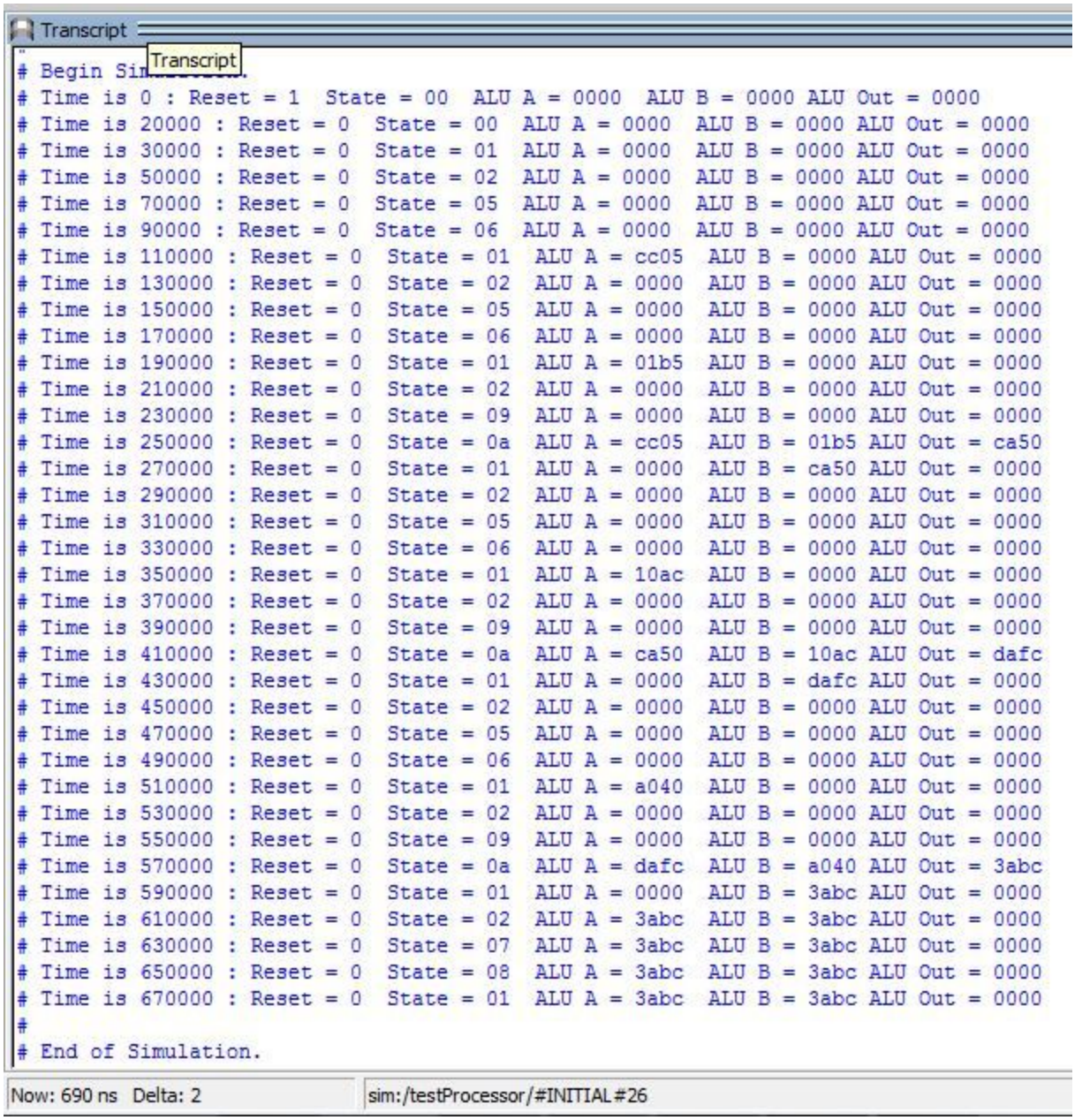

Figure 9. ModelSim Output

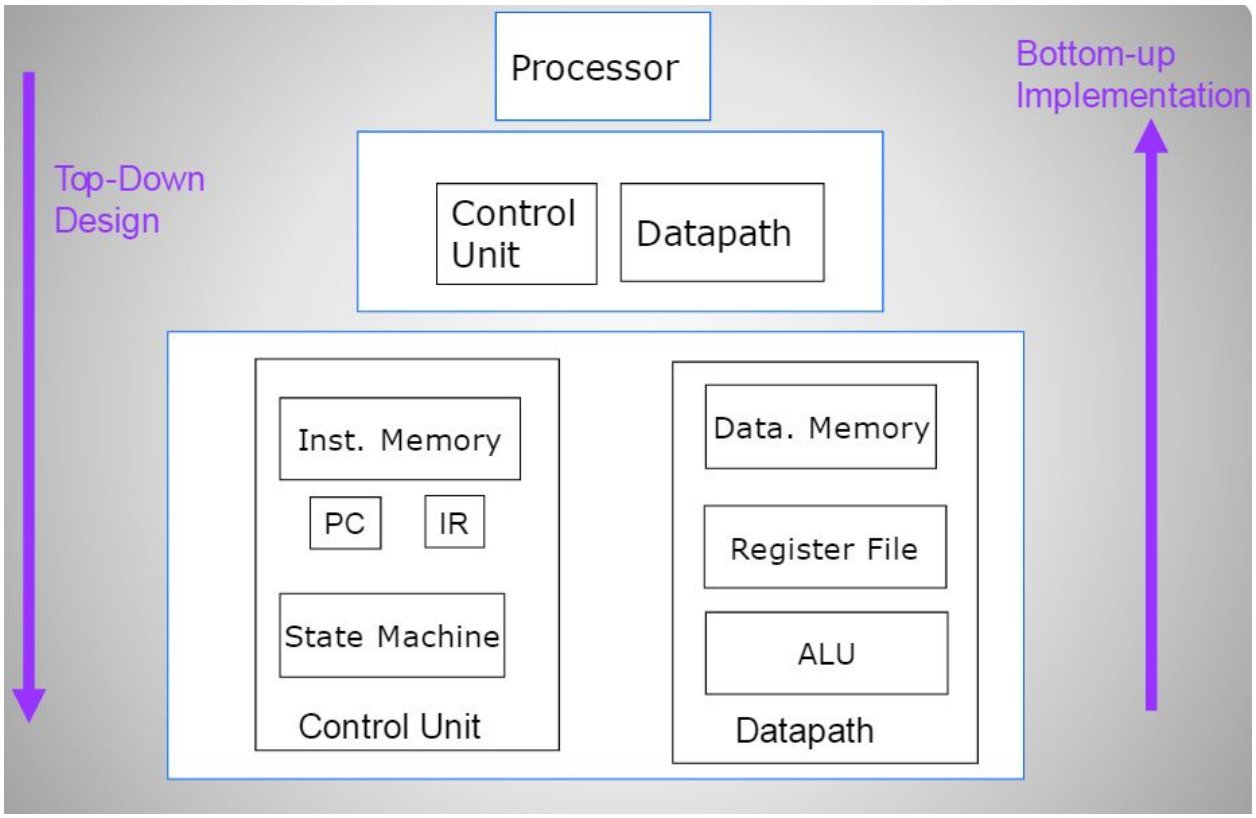

Figure 10. Processor Module Layout

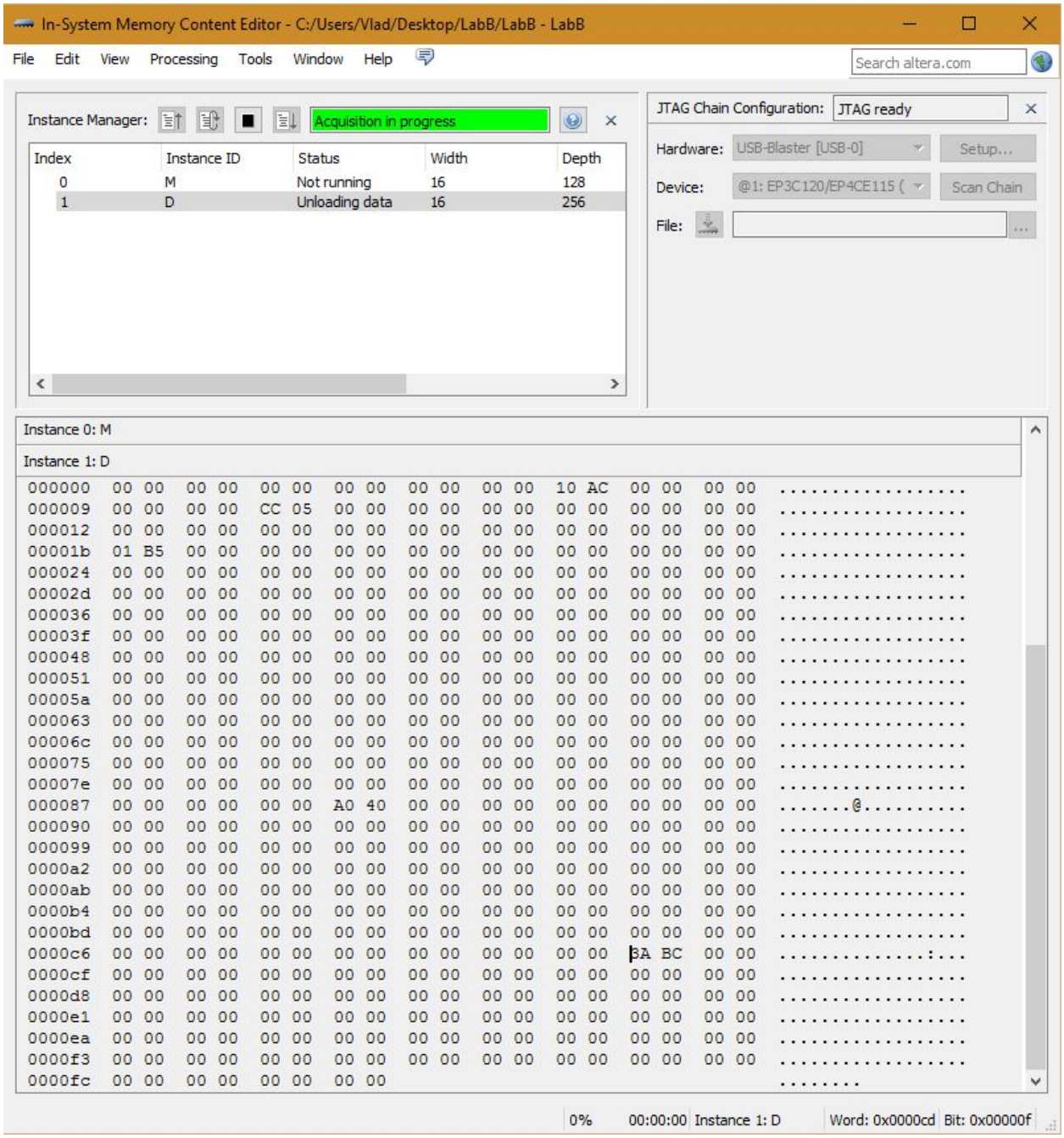

Figure 11. In-System Memory Content Editor - Data Memory - Final