Laboratory Exercise B

Programmable Processor

TCES 330 Digital Systems Design

Spring 2016

Authors:

Maxim Kulish

Wayvern McCormick

Zeeshan Karim

Table of Contents

**Requirements2**

**Design3**

**Test Procedures5**

**Observations6**

**Conclusion6**

**Appendix7**

Laboratory Assignment A

The purpose of this assignment is to develop a programmable processor in Verilog, following the outline provided by the assignment materials. The processor will have a 16 bit wide instruction set, and instructions for NOOP, STORE, LOAD, ADD, SUBTRACT, and HALT will be implemented. Finally the processor will be loaded onto the DE2-115 board for verification, using KEY[2] as the clock signal, KEY[1] as a synchronous reset, and the HEX displays to display relevant data from the processor.

Requirements:

* Implement the six-instruction programmable processor as described in Processor16.pptx, provided by the instructor, using Verilog.
* Interface to the DE2-115 board as follows:
  + KEY[2] acts as the system clock
  + KEY[1] acts as a synchronous system reset
  + Display the program counter instruction register, state machine current state, both ALU inputs, and the ALU output using the HEX displays.
  + HEX 3, 2, 1 and 0 are used to display the contents of the instruction register
  + SW[17:15] determine what HEX 7, 6, 5 and 4 display as follows:
    - SW[17:15] = 0:

HEX 7, 6 display program counter, HEX 5, 4 display current state

* + - SW[17:15] = 1:

HEX 7, 6, 5, 4 display A side input to the ALU

* + - SW[17:15] = 2:

HEX 7, 6, 5, 4 display B side input to the ALU

* + - SW[17:15] = 3:

HEX 7, 6, 5, 4 display output from the ALU

* + - SW[17:15] = 4:

HEX 7, 6, 5, 4 display the next state variable

* + - SW[17:15] = 5: Unused
    - SW[17:15] = 6: Unused
    - SW[17:15] = 7: Unused
  + The top level module instantiates the processor module, a multiplexer for selecting the output of the displays, and a key conditioner.
* Implement the processor using the same interface as that defined in the ModelSim testbench, provided by the instructor.
* The following sample program will be compiled and loaded into instruction memory:

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

D[CD] = RF[0];

HALT

* Data memory initially contains:

D[6] = 0x10AC

D[B] = 0xCC05

D[1B] = 0x01B5

D[8A] = 0xA040

* Use the Quartus In System Memory Content Editor to view the contents of the instruction memory and data memory, and verify them.
* Run TimeQuest and generate a sdc file for the 50 MHz system clock, and the clock derived from KEY[2].
* Simulate the circuit using ModelSim and the testbench provided by the instructor.
* Upload the design to the DE2-115 board and verify its functionality.

Design:

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

The overall objective of the lab was to design a processor to work with an ALU and perform the following operations:

NOOP – Performs no operation.

STORE – Stores value from specified register file address to specified data memory address.

LOAD – Loads value from specified data memory address into specified register file address.

ADD – Takes the value in register file address A, sums it with the value in register file address B and stores the sum into register file address C.

SUBTRACT – Takes the value in register file address A, subtracts from it the value in register file address B and stores the difference into register file address C.

HALT – Causes the processor to perform a hard stop. The only way to move to a different state is to perform a reset.

Each instruction follows a 4 digit hexadecimal format; with the first digit always representing the opcode of the function, and the remaining digits containing relevant memory addresses. The format of each instruction implemented is described in the following table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Instruction Code | | | | |
| Instruction | OpCode | First Hex | Second Hex | Third Hex |
| NOOP | 0 | 0 | 0 | 0 |
| Store | 1 | Source Register | Destination Data Memory | |
| Load | 2 | Source Data Memory | | Destination Register |
| Add | 3 | Register A | Register B | Register C |
| Subtract | 4 | Register A | Register B | Register C |
| Halt | 5 | 0 | 0 | 0 |

The overall task of implementing the processor needed to be broken up into several modules:

The top level module, LabB.v, implements the processor to the DE2-115 board using the hardware specified in the requirements and uses the multiplexer designed in MUX16\_8\_to\_1.v to choose between items displayed on the 7-segment displays. Also, the module ButtonSync.v was used to ensure using the key as a clock only puts out one clock pulse at a time and the module KeyFilter.v was used to limit the button being pushed to 10 per second. The RTL view for this module is available in Figure 1. ButtonSync.v and KeyFilter.v were provided by the professor, Robert Gutmann.

Processor.v is a higher level module that instantiates the datapath and controller for the processor. It also provides some outputs to the top level module for display on the hex displays. The RTL view for this module is available in Figure 2.

Controller.v contains the state machine that drives the processor and instantiates the instruction memory as a 128 X 16 ROM LPM module. The state machine contains 11 states:

* Init – Clears the PC and sets next to Fetch
* Fetch – Loads the next instruction from the instruction register and sets next to Decode.
* Decode – Increments the PC and decides which state is next based in the first 4 bits of the instruction.
* NOOP – Does no operation and sets next to Fetch.
* Load\_A – Pulls the data address from the instruction and sets next to Load\_B.
* Load\_B – Pulls the data address and the register file address for A from the instruction, sets the write enable for port A to true, and sets next to Fetch.
* Store\_A – Gets the register file address for A from the instruction and sets next to Store\_B.
* Store\_B – Gets the data address from the instruction, sets data write to true, and sets next to Fetch.
* Arith\_A – Gets the register file addresses for A and B from the instruction, decides which ALU instruction to perform depending on the first 4 bits of the instruction and sets next to Arith\_B.
* Arith\_B – Gets the register file address for B from the instruction, sets write for B to true, decides which ALU instruction to perform depending on the first 4 bits of the instruction, and sets next to Fetch.
* Halt – Does nothing and sets next to Halt, causing the processor to stop and stay in this state unless reset.

States that enable or disable hardware, such as load and store states, utilize enable signals that are sent to the Datapath module to allow proper functionality. Additionally memory addresses are output to the Datapath module after being read from the instruction ROM. These outputs can be seen in the RTL view for Processor.v, coming out of the Controller module.

Because this modules functionality is mostly handled by the state machine, a RTL view is less useful; and therefore a state diagram has instead been included in Figure 4.

Datapath.v takes the addresses, control signals, and ALU selection signal from the Controller module and performs the appropriate tasks. Also, the data memory is instantiated as 256 X 16 RAM LPM module; and the register file is instantiated using a 16 X 16 2-port RAM LPM module. ALU operations are performed using the ALU74381.v module provided by the professor, Robert Guttmann. The inputs and output from the ALU are output from the Datapath module; for display on the DE2-115 board. An RTL view of this module is available in Figure 3.

Test Procedures:

In order to test the completed processor, we first implemented the example pseudocode provided by the professor in our assembly language. Below is a table describing each of the instructions, and the hex codes that represent them for our processor.

|  |  |  |  |
| --- | --- | --- | --- |
| Program Instructions | | | |
| PC | Instruction | Action | Hex Code |
| 00 | Load | D[0B] 🡪 R[0] | 20B0 |
| 01 | Load | D[1B] 🡪 R[1] | 21B1 |
| 02 | Load | D[06] 🡪 R[2] | 2062 |
| 03 | Load | D[8A] 🡪 R[3] | 28A3 |
| 04 | Subtract | R[0] – R[1] = R[0] CC05 – 01B5 = CA50 | 4010 |
| 05 | Add | R[0] + R[2] = R[0] CA50 + 10AC = DAFC | 3020 |
| 06 | Subtract | R[0] – R[3] = R[0] DAFC – A040 = 3ABC | 4030 |
| 07 | Store | R[0] 🡪 D[CD] | 10CD |
| 08 | Halt | End | 5000 |

Hex codes were then placed in the Instructions.mif file, in order by PC count, so they could be loaded into the design when compiling. Additionally, the data values described in the Requirements section were added to Data.mif, at the specified memory locations. After this, the program was tested in ModelSim using the testbench provided by the professor. Finally, the program was loaded to the DE2-115 board for verification.

Observations:

After successful compilation of the design, we observed the total logic elements used and ran TimeQuest to obtain FMax, the maximum operating frequency of our design. The design used a total of 619 logic elements, and is capable of operating at up to 98.23 MHz.

After adding our compiled sample code in Instructions.mif, and the specified data into Data.mif, running the testbench provided by the instructor on the processor module resulted in the printout shown in Figure 7.

Finally, loading the design to the DE2-115 board and manually running the code cycle by cycle allowed us to view the instructions in hexadecimal on the hex displays, as well as view the ALU inputs and outputs. The operation performed as expected, as we were able to view the instructions loaded from the instruction memory on the first 4 displays, and the inputs and outputs of the ALU on the remaining displays.

Viewing the Instruction Memory Content Editor (Figure 5) showed the compiled code we had placed in the Instructions.mif file. Additionally, the Data Memory Content Editor (Figure 6) displays the original data we had included in the Data.mif file, as well as the result of our program stored in memory location 0xCD.

Conclusion:

We were successful in developing the processor as described in the assignment materials, as well as implementing NOOP, STORE, LOAD, ARITH, and HALT instructions. Additionally, a sample program was implemented and loaded in to the instruction memory. The functionality was successfully tested using ModelSim, and by interfacing the processor module with the DE2-115 board. Interestingly, this Verilog project only made use of 619 logic elements on our FPGA, less than 1% of the total supported; demonstrating the ability of this type of development to design and test much more complex projects.

Appendix:

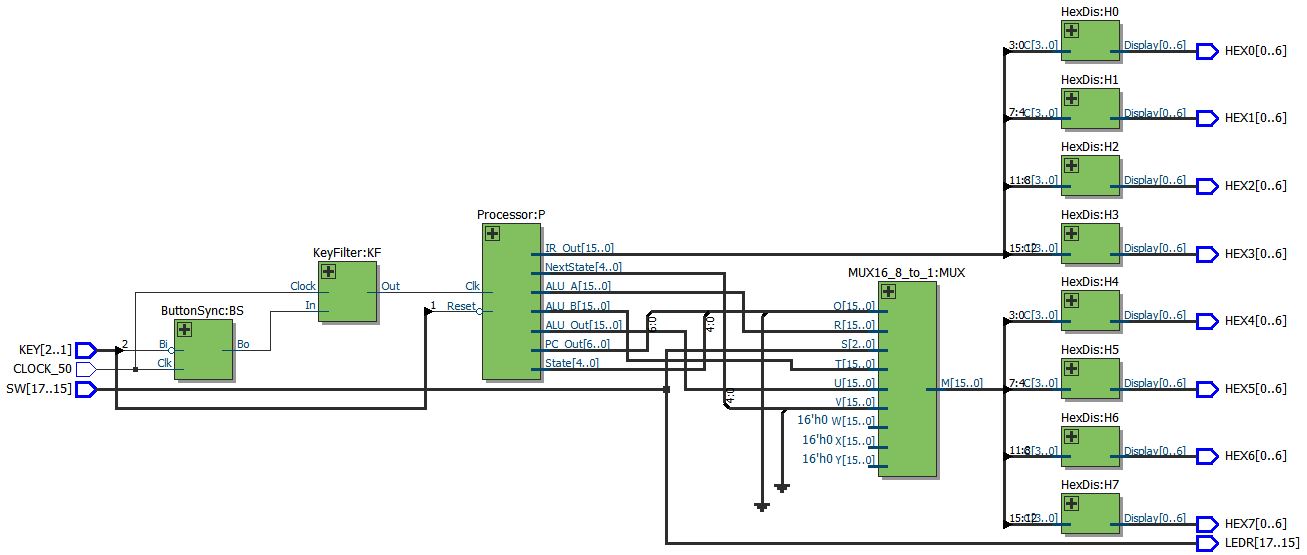


Figure 1 – Top Level RTL View

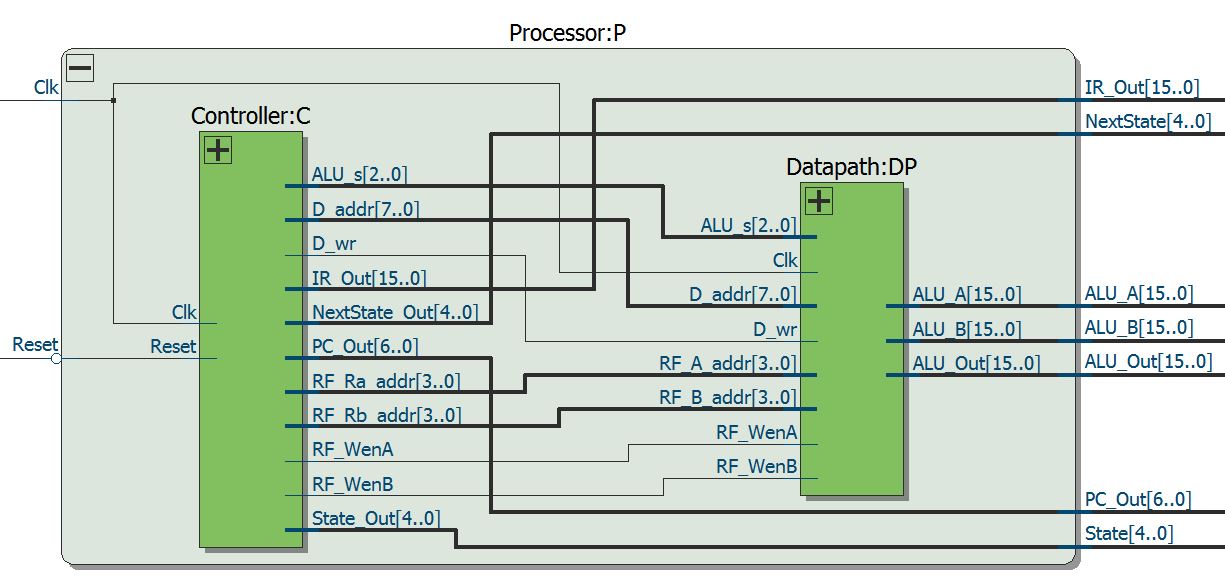


Figure 2 – Processor Module RTL View

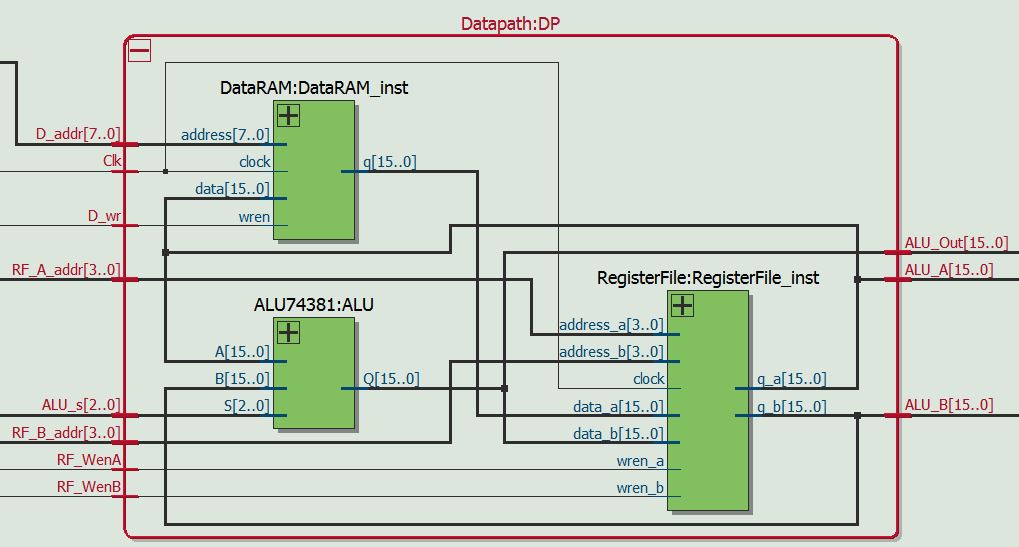


Figure 3 – Datapath module RTL View

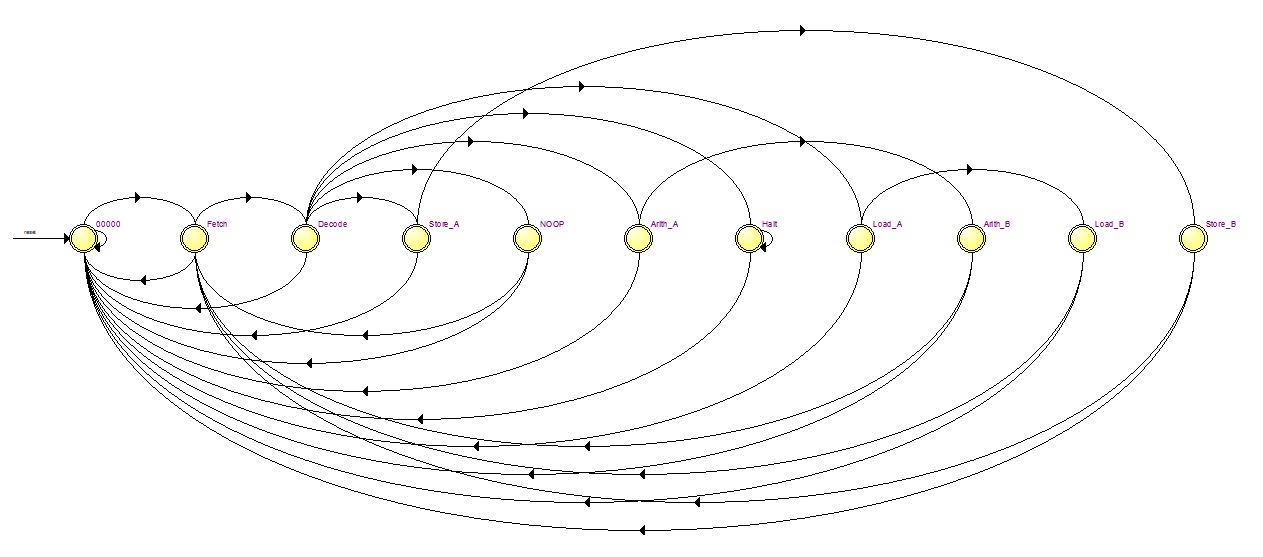


Figure 4 – Controller State Machine

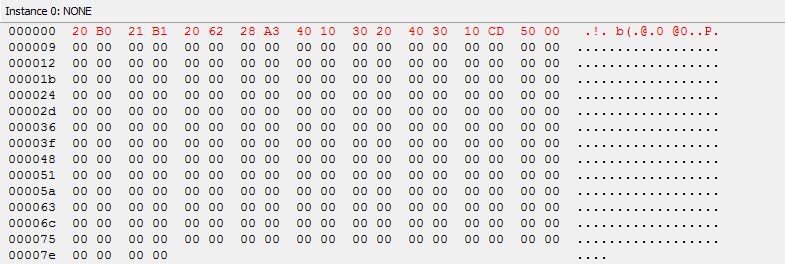


Figure 5 – Instruction Memory Content Editor

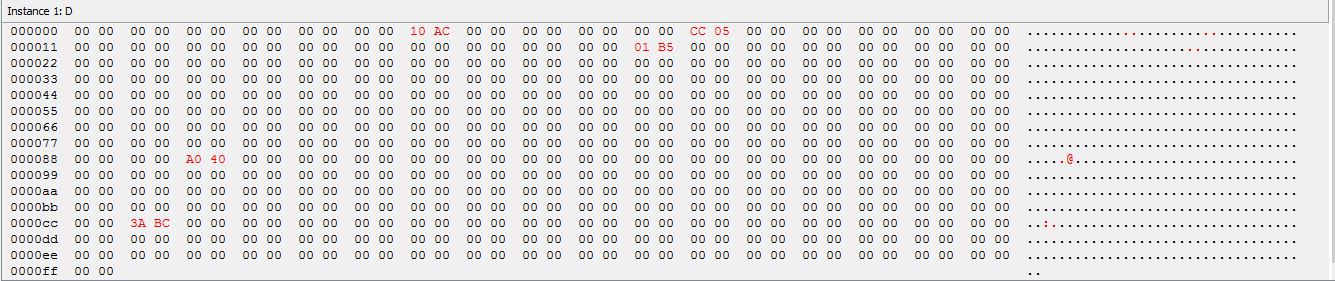


Figure 6 – Data Memory Content Editor (after sample program stores value)

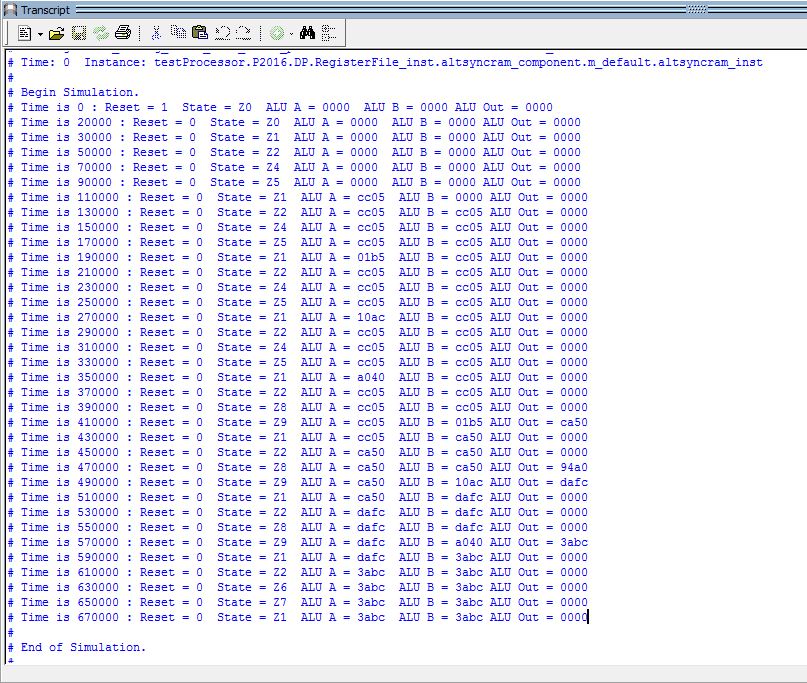


Figure 7 – ModelSim Testbench Printout