RISC Computer

Technical Report

Kyle Lavorato (10141235)

April 9th, 2017

Abstract

A simple RISC computer consisting of a RISC processor, memory, and I/O was designed and implemented on an Altera Cyclone chip. Verilog was used for an all-HDL design approach. In addition to the required datapath and control unit components, the computer also consists of a 16x16 carry look ahead adder (CLA) and 32bit booth multiplier, which takes two signed binary numbers in two's complement notation and computes a value.

An assembler was written in the language Python to parse an input stream of instructions into a 32bit stream to be used in conjunction with the MiniSRC computer. This assembler was used during Phase Three of the project to assemble the test program to load it into the MiniSRC's memory to be run. A custom RAM module was written for the MiniSRC instead of using the Altera Megafunction pre-built Oneport RAM. A new register increment instruction was added to the MiniSRC, as register increment is a very useful hardware instruction for optimizing code.

The test programs had an average cycle per instruction (CPI) of 10. Though efficient, this number could be improved upon by more effectively asserting the control signals. Only 9.5% of the total chip area was used in the design.

The complete design of the datapath and control unit were successfully completed and implemented into the computer program. Full implementation onto the Altera Cyclone chip was not successful, as the wrong chip was being used for download.

The main recommendation for future improvement on the program would be to improve the naming convention used for wires and signals used in the beginning of the development process. As the proper naming convention was not adopted until later in the process, it made it difficult to keep track of various computer components.

Contents

Project Specifications

The purpose of the project was to design, simulate, implement and verify a simple RISC computer consisting of a RISC processor, memory, and I/O. There were four separate phases of the project, each consisting of the design of various parts of the datapath and control unit, with the final phase entailing their implementation on the Altera Cyclone III chip. The team decided to use an all-HDL design approach, using Verilog for all the components.

Design

The following section provides detail on various aspects of the computer's design.

Datapath Overview

The datapath was designed as a single 32-bit bus architecture as seen in Figure 1 below.

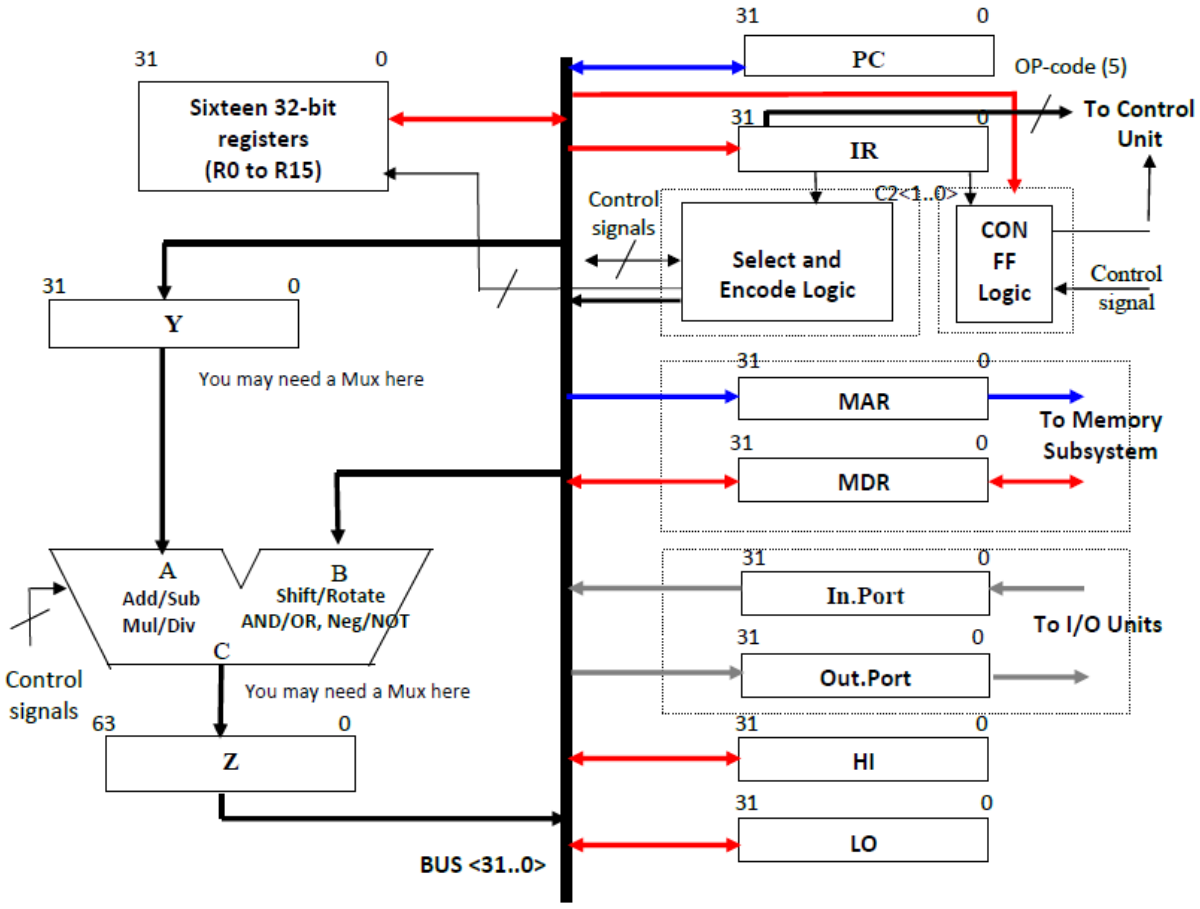

Figure 1: The design schematic for the single bus architecture

Due to the single bus design, at any one time, only one value can be on the bus, which is determined through bus arbitration logic. The bus arbitration logic is generated through a multiplexer, where all the components that are connected and capable of outputting to the bus are fed as inputs. There is an encoder that generates a select signal for the multiplexer. This encoder is fed by single bit control signals

(one hot style input) to generate a select code to let that signal through the multiplexer. The bus arbitration logic design is shown in Figure 2 below.

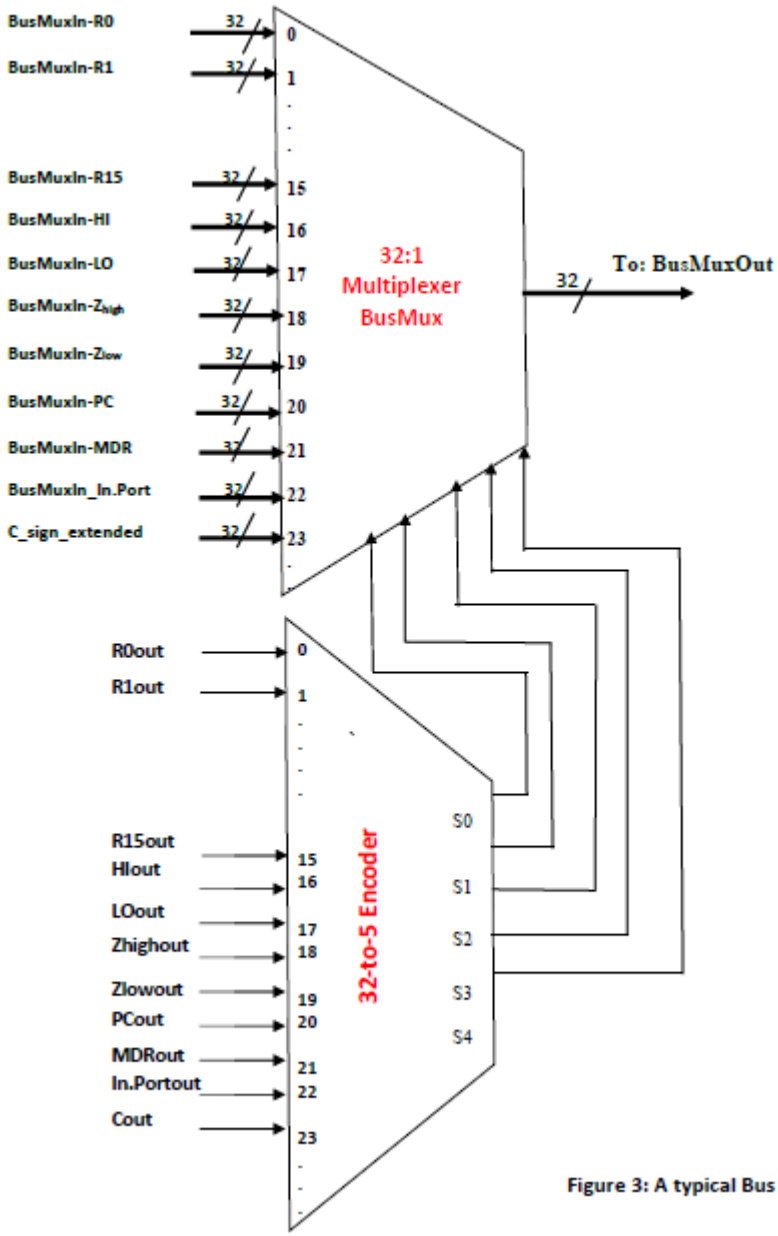

Figure 3: A typical Bus

Figure 2: Schematic of the bus arbitration logic

There is a separate enable logic module for controlling what can accept the bus input. At any time, the bus output is fed to every connected component that has an arrow feeding into it in Figure 1, though they will only change their value if their enable is set. The enables for each component are generated in two separate groups.

There is a select and encode logic module that generates all the enables for the 16 registers from IR's value; its high-level schematic can be seen below in Figure 3.

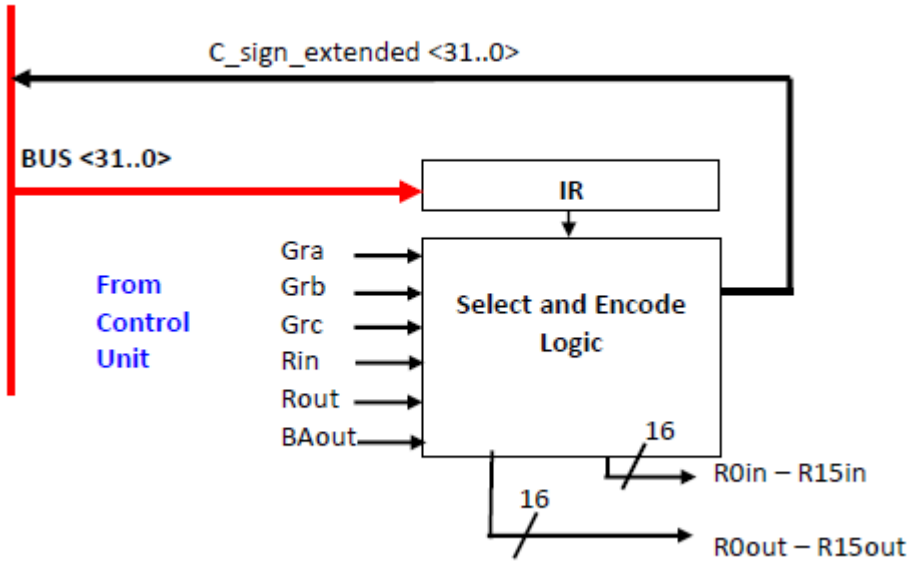

Figure 3: High-level schematic of the select and encode logic

The remaining enable values are generated by the Control Unit and will be discussed in the next section. The datapath contains a memory subsystem interface with the MAR and MDR registers. The memory subsystem is defined in Figure 4 below.

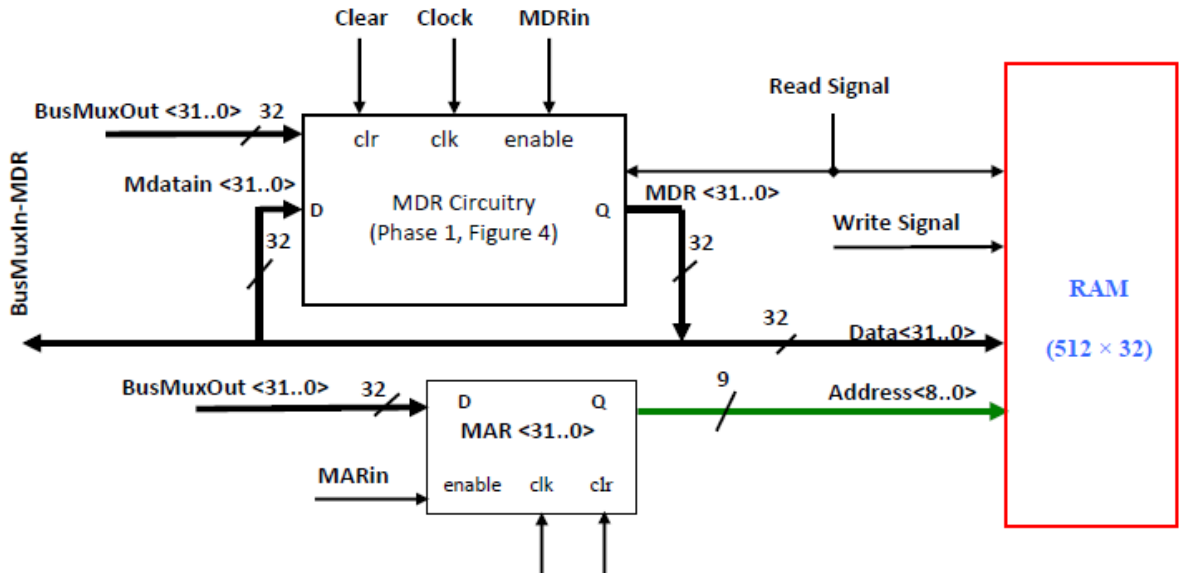

Figure 4: Schematic of the memory subsystem

The designed memory subsystem, select and encode logic, R0 register, CON FF logic and I/O ports all closely resemble the above figures. For the CON FF logic module, a MegaFunction implementation of the comparator circuit was used.

Registers

For the computer, various special signals are inputted, most having an 'in' and 'out' input. The ones labeled 'in' or "_en' are the enable signals. When they are set to 1, the register value can be changed. When equal to 1, the value on the bus is taken and put in the register.

There is another set of input signals labeled 'out', which are the inputs for the encoder. The encoder selects which values go on the bus. It takes each of the signals as input, and provides a select signal. Then the bus takes the select signal and puts that value on the bus.

Bus

'busDataIn' places a number into the In.Port. Then, InPort_out takes what is in the In.Port and places it on the bus. The variable 'r2In' is used to tell Reg 2 to store what is currently on the bus.

ALU

It takes two cycles to load the ALU. It begins by taking the value on the bus and putting it in Reg Y. The next cycle then loads a new value on the bus and puts that value in ALU_B_In. The old value from Reg Y goes to ALU_A_In so that both numbers are in the ALU. The ALU takes opcode, which tells it which instruction to run (add, subtract, multiply, etc). The result of the operation is passed into Reg Z, which is a 64-bit register used to hold the result of the operation in the ALU. As the bus can only hold 32 bits of data, Reg Z is used to output a HI or LO. It choses whether it wants the HI or LO based on whether Zlow_en or Zhigh_en is selected, which instructs Reg Z to either use the top 32 bits or bottom 32 bits.

Please note that the ADD, SUB, DIV, SHL, SHR, ROL, ROR operations were all done using the Megawizard function of Quartus.

Carry Look Ahead Adder

The 16x16 carry look ahead adder (CLA) can be seen in *cla16.v.* The adder uses a two-level design. It was created by combing four 4-bit CLAs. The 16-bit unit accepts both the group propagate 'PG' and group generate 'GG' from each of the four 4-bit CLAs. The carry input for each CLA is generated. The adder was tested using *cla16_tb.v.*

Booth Multiplier

The booth multiplier seen in *booth32.v* uses the booth algorithm to multiply numbers. It takes two signed binary numbers in two's complement notation and computes a value. The multiplier was tested using *booth32_tb.v.*

Control Unit Overview

The control unit is responsible for generating the control sequences of the remaining un-generated control signals. At any time, the signals that are required to be asserted are a result of the current cycle of the instruction being executed. Therefore, the control unit is designed as a Finite State Machine (FSM) with each state representing a different cycle. Each state has predefined sets of which control signals to assert and begins by setting all control signals to low. The base state is called the reset_state, where it is determined which it enters sequentially into the instruction fetch states 0, 1 and 2. Once fetch2 has

completed, it must be determined what instruction is executing to choose the next state. The state uses a large case statement to choose the next state based on the opcode that has been fetched. Once it is in an instruction specific state, such as add3, it will sequentially continue to add4 and add5, where it will then move back to the reset_state. This is the case for all instruction states, except the ones that have been optimized, as explained in the following section. The overview of the control unit's interface can be seen in the schematic in Figure 5. All the inputs to the control unit are generated off the datapath, except for the ConFF signals. These are generated by the ConFF module, which is not connected directly to the datapath. As seen by the schematic in Figure 6, the ConFF module is connected to the IR and then generates its signals and feeds them directly to the Control Unit.

Code Optimization

The team decided to optimize the code needed to generate the cycles in the control unit. Instead of making a cycle for every instruction (e.g. add3 add4 add5, sub3, sub4, sub5), the code reused the cycle states from other instructions that asserted the same control sequences. This added a small amount of extra overhead since it required extra case statements for examining opcodes, but it saved more code overall. Notes can be seen on things using other cycles in the comments of the control unit.

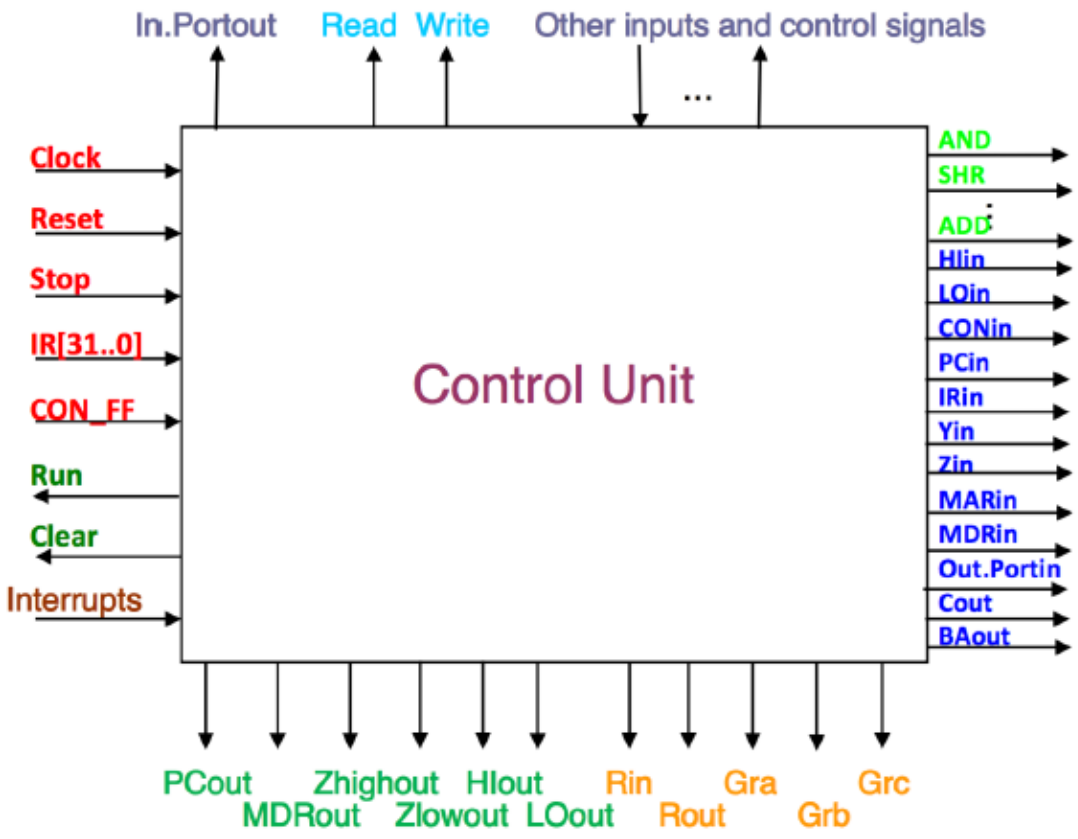

Figure 5: Schematic of the Control Unit

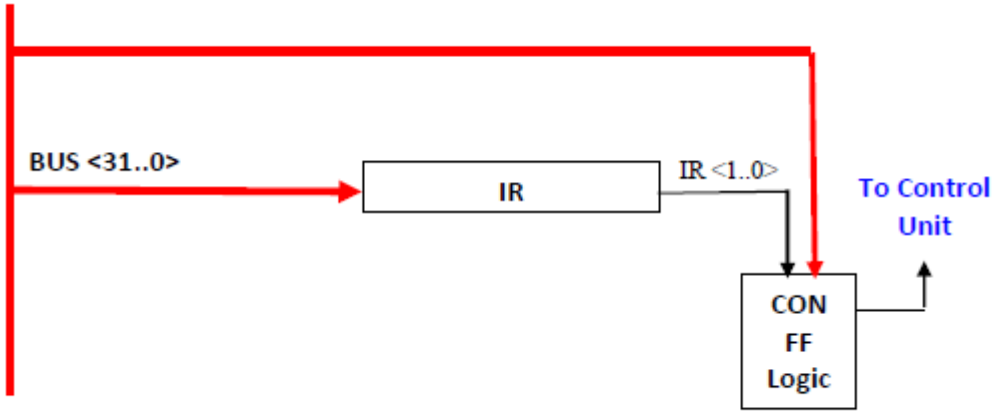

Figure 6: Schematic of the ConFF module

Bonus Design

Assembler.py

An assembler is a crucial part to any RISC style computer as the human understandable machine instructions must be parsed into a form that the computer can understand. An assembler has been written in the language Python to parse an input stream of instructions into a 32 bitstream to be used in conjunction with the MiniSRC computer. This assembler was used during Phase Three of the project to assemble the test program to load it into the MiniSRC's memory to be run.

The assembler reads the contents of the file "code.asm" as input and assembles it into the file "assembled.asm." There are two modes of operation, which can be changed by setting the "bitstring" variable to True or False. The first mode of operation (True) simply assembles the instructions into the bitstring form, with the fields of the instruction (e.g. opcode_ra_rb_c) separated by underscores for visibility. The second mode operation (False) assembles the instructions into the bitstring form and appends a prefix "ram[x] <= 32'b", where x is the current memory location. This mode of operation is used to directly assemble the instructions into a program to manually load into the ram512.v file of the MiniSRC.

The assembler itself is a very simple script. It uses lookup tables (Python dictionaries) to match human machine instruction to a bitstring value. This makes the assembler very sustainable as it is very easy for a person with no programming experience to add a new instruction. The fields of the instructions are read and parsed in order of appearance. The bitstring is then assembled, temporarily excluding the constant, additionally making sure that each value is placed in the correct field depending on the instruction. Finally, the constant is sign extended from 8 bits to bring the total size of the instruction to 32 bits. The source code of the assembler is available in Appendix A. The execution of the assembler on the test program from Phase Three is available in Appendix B.

Inferred RAM

A custom RAM module has been written for the MiniSRC instead of using the Altera Megafunction pre-built Oneport RAM. The file ram512.v, available in the source code in Appendix C, is an asynchronous inferred RAM module. Like the Megafunction RAM, the module accepts an address and read/write

signals to define a mode of operation. The RAM has been written to be asynchronous to ensure that the memory operations will be complete and ready for input/output by the next clock cycle. The RAM has been tested fully and there have been no hazards detected from the asynchronous operation.

The memory contents of the RAM are defined by “reg [31:0] ram [0:511];” creating an array called ram of size 512x32. Each memory location of ram is to hold a full 32 bits, allowing the MiniSRC to use increments of one for addressing PC. Finally, the RAM is preloaded with the instructions for the program prior to the execution of the test bench. This is done by setting the value of any memory location with a statement such as “ram[0] <= 32'b00001_0011_0000_0000000000001010111;.”

Inc Instruction

A new register increment instruction has been added to the MiniSRC. Register increment is a very useful hardware instruction for optimizing code. Incrementing a value is a very common line of code in all programs (e.g. i++) so having an instruction to optimize it saves high amounts of machine code during program compilation. Not only does the instruction allow lower code generation, but it also is able to execute in four cycles instead of the five cycles of a typical add instruction. Shown below is the documentation for the new instruction:

Increment	R[Ra] ←	R[Ra] + $1	inc Ra

(11011xxxxxxxxxxx----------)

The increment instruction has been added to the alu.v file as it is an arithmetic instruction and its implementation can be seen in the source code in Appendix C. A testbench of the instruction can be seen below in the Testing section.

Memory Stack

A built-in hardware stack module has also been written for the MiniSRC computer. Hardware stacks are a very useful, fast way to store and retrieve values, especially to preserve register contents upon entrance and exit of a subroutine. The memory stack consists of a 32x32 array of memory, initialized similar to that in the “ram512.v” file and an internal Stack Pointer register. The interface to the module is also identical to that of the RAM, excluding the address as that is handled by the internal stack pointer. The memory stack is written in “stack32.v” and can be found in the source code in Appendix C.

When the write signal is asserted, the value will be stored in memory at the location of the stack pointer, which always points to the top empty location of the stack. Once the write is complete, the stack pointer will be incremented to again be pointing at the top free location of the stack. When the read signal is asserted, the stack pointer will first be decremented to be pointing at the top value in the stack. Then the value is read and outputted. This is analogous to popping from the stack. For efficiency, the old value is not deleted and is instead marked as free space to be overwritten.

The stack is accessed through the use of two new instructions “ststk” and “ldstk” for store and load on the stack respectively. Shown below is the documentation for the new instructions:

Load from Stack	R[Ra] ←	S[SP]	ldstk Ra

(11110xxxxxxxxxxx----------)

Store on Stack	S[SP] ←	R[Ra]	ststk Ra

(11111xxxxxxxxxxx----------)

The MDR is used for both the memory stack and the RAM modules so slight modifications needed to be made. Since both modules will be outputting to the same source, blocking needed to be added to both the memory stack and RAM so they do not assert any signal when not in use. The memory stack and RAM were altered to output a bitstring of 32 '0's when not reading a value. This allows the two outputs to be ORed together and then the result of that 'or' be the input to the MDR. This makes the architecture and design of the memory stack more efficient, as it can use the same interface as the RAM and mostly the same control signals.

The memory stack and new instructions are tested in the Testing section below.

Testing

The bonus sections have been tested in tandem with the below test program, which utilizes the assembler to assemble it, the RAM to store the instructions, and executes instructions with increment and the memory stack. Functional simulations of the tests can be seen in the next section.

code.asm

```
ldi R3, 6
ststk R3
inc R3
ldstk R3
halt
```

assembled.asm

```
00001_0011_00000000000000000000110
11111_0011_00000000000000000000000
11101_0011_00000000000000000000000
11110_0011_00000000000000000000000
11100_00000000000000000000000000
```

Functional Simulations

ldi R3, 6 + ststk R3

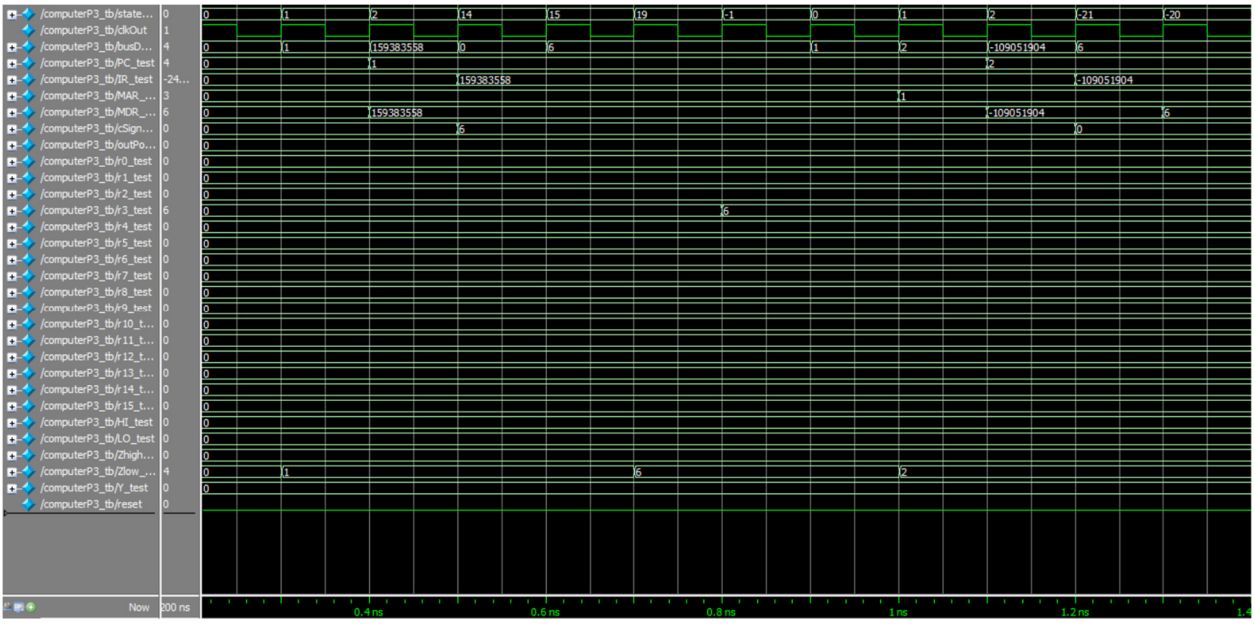

Figure 7: Waveform for the ldi R3, 6 + ststk R3 instructions

inc R3 + ldstk R3

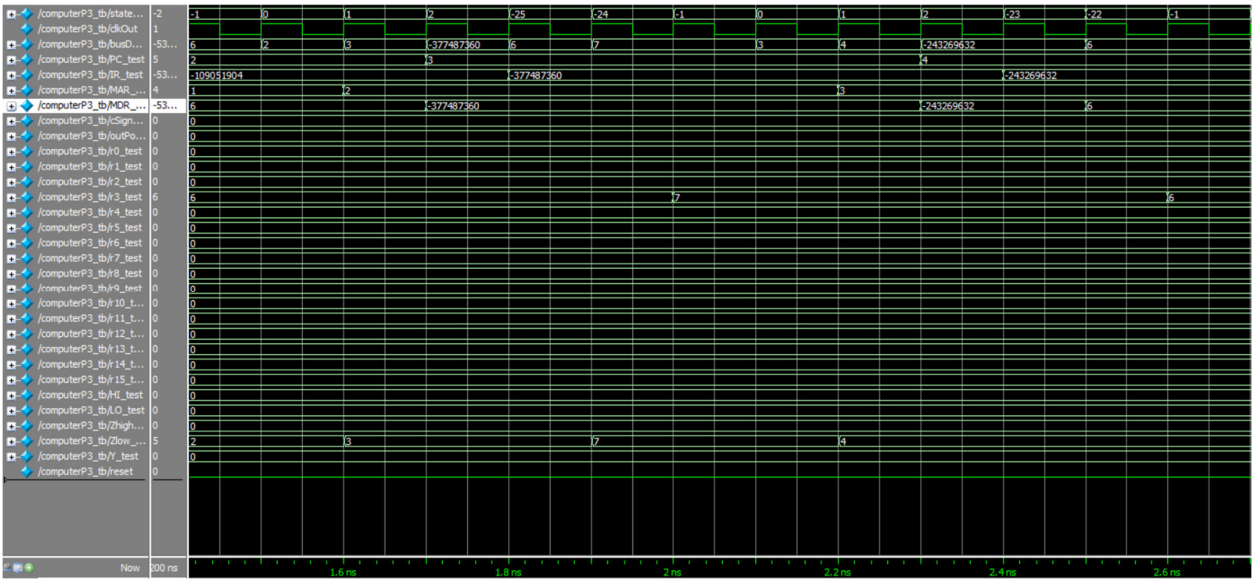

Figure 8: Waveform for the inc R3 + ldstk R3 instructions

Shown above is the execution of the simple program to test the bonus elements. In the first picture, the ldi instruction loads the value 6 into register three at the end of state 19. Then the ststk instruction executes. The value of 6 from register three can be seen in the MDR in state -20, which indicates that it is being sent out to the stack memory. Later the same value will be retrieved from the stack.

In the second waveform, the inc instruction is used on register three. The value 6 of register three can be seen on the bus going into the ALU in state -25 and is then incremented by one, ending up in Zlow at the start of cycle -24. Then register three's value is set to 7 at the end of state -24. Next is the ldstk instruction, to restore register three to its original stored value. The value of the MDR can be seen to be 6 as the value is retrieved from the stack in state -22. Then the value of register three is set back to 6 at the end of state -22. This test indicates that all new instructions and functionality is working as intended.

Results

The maximum frequency of operation for the simulation and chip was 50MHz, due to the 50MHz oscillator available at FPHA pins PIN_G21 and PIN_B12.

From the Phase Three simulations, it was determined that there were approximately 40 instructions asserted over 385 cycles. This means that the test programs had an average cycle per instruction (CPI) of approximately 10.

385 cycles / 40 instructions = 9.6 = ~10 cycles/instruction

This number is relatively low, showing that the control signals were asserted in an efficient manner. However, there is definitely room for improvement. More efficiently using the control signals could have resulted in a lower CPI.

To determine the percentage of chip area used, the team looked at the total number of pins used in the program. Since 33 out of the 347 pins available were used, the percentage of chip area used was approximately 9.5%.

Conclusion

The complete design of the datapath and control unit were successfully completed and implemented into the computer program. Future development on the project could include the completion of the fourth phase of the assignment, as the team was unable to successfully test the program on the Altera Chip. This would allow the full functionality of the computer to be evaluated.

Recommendations

Potential improvements could be implemented in the usage of the control signals to lower the programs CPI. Since 10 cycles/instruction could still be considered relatively high, the team could look into lowering the number to 5 cycles/instruction to ensure more efficient assertion of the signal