374 Project Final Report

April 7, 2023

Group 4:
Liam Salass (20229595)
Abdellah Ghassel (20230384)

"We do hereby verify that this written lab report is our own work and contains our own original ideas, concepts, and designs. No portion of this report has been copied in whole or in part from another source, with the possible exception of properly referenced material."

Abstract

This document describes the design and implementation of a mini src computer, a 32-bit CPU with 32 by 512 RAM. The mini src computer has one input port and one output port and can handle all types of instructions (R-Type, I-Type, B-Type, J-Type, and M-Type). The CPU was coded entirely in Verilog, and test benched using ModelSim. The document also explains the logic and functionality of each CPU component, such as the ALU, the register file, the control unit, and the instruction memory, datapath, and select and encode logic. The document also demonstrates that the CPU can perform the XOR operation, an operation not required by the documentation. An evaluation of the size and performance of the computer is included. The computer had perfect functionality in test benching in ModelSim, but due to time constraints, the CPU was not ported to testing on an FPGA chip.

Table of Contents

Project Specification

This project involved designing and implementing a Mini SRC computer. The computer had to meet hardware and functionality criteria. We used Verilog code to create the computer's hardware components and functionality. The hardware contained components (Verilog Modules) for many general features, such as registers. Other Verilog components, such as the Control Unit, were used to meet functional specifications.

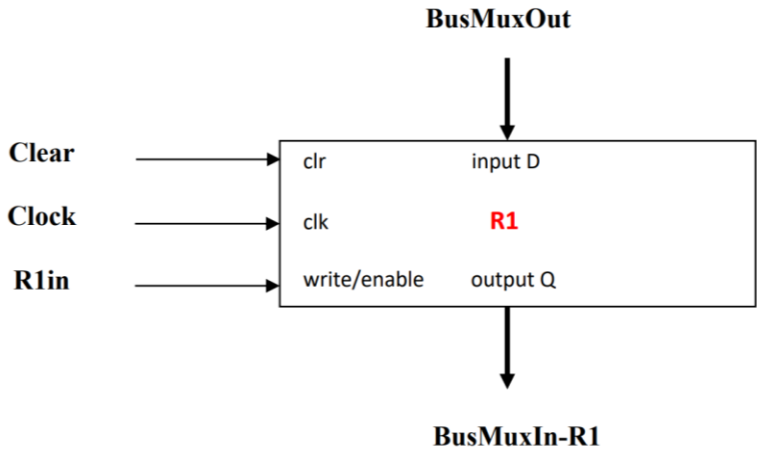
In the final stage of our project, our computer can read a pre-initialized .hex file into its RAM. From there, it can execute any code in the .hex file as long as it follows the correct instruction specifications (outlined later on). The computer exceeded all functional requirements of phase 1, phase 2, and phase 3, with all instructions correctly behaving when read from a .hex file into the memory and then having the computer be reset and run.

Due to time limitations, the computer was never loaded onto or run on an FPGA chip. However, our computer has more functionality than that specified by the project outline. The operation XOR was added to the list of function instructions. Our computer also performs branching in one less cycle than what was expected. Lastly, we implemented a booths version of multiplication and a non-restoring implementation of division, resulting in higher performance from the slower ALU operations.

Below is an outline of all the specifications of the computer and its components.

Register Components:

All registers are 32-bit and have a clr, clk, D, and Rin inputs. The only output of the registers is the Q value. The clear resets the register value to 0, the D is the 32-bit input from the datapath, and the select allows the value in D to be placed inside the register's internal memory.



Program Counter (PC) Register

The program counter register was used for holding the current program location in the memory's address. It would be incremented within the first three cycles of any instruction to point to the next instruction to be executed.

Instruction Register (IR)

Used to hold the value loaded from the memory location pointed to by the PC. This value is decoded to execute instructions in the execution cycles.

General Purpose Registers

Eight general-purpose registers were used to store data or memory addresses or perform operations with the values within them.

Argument Registers

There are four argument registers. They are similar to general-purpose registers but are used to pass arguments to functions or subroutines. This is only a design practice; ultimately, how they are used is up to the assembly program coder.

Return Registers

In our solution, there are two return registers. As argument registers, these registers can be used for anything; however, their purpose is to store the address of the instruction to be executed after a subroutine or function. They can be used for nested functions with a depth of 2 or 3.

Stack Pointer Register (SP)

The SP register holds the memory address of the stack. It can store general-purpose register values in memory before entering a function call and retrieve those values back once exiting. Only one SP register in the design.

Return Address Register (RA)

The RA register holds the old PC value when a jump and link (JAL) instruction is executed so that at the function end, that value can be placed back in the PC using a jump instruction. This will cause the PC to continue right after the JAL instruction and continue. There is only one RA in our CPU, and its select wire is or'd with a special flag wire for when a JAL instruction is executed so that the PC value can be stored within it.

HI And LO Registers

The HI register holds the extended value output from multiplication or the remainder from a division execution. The LO register contains the lower bits from multiplication or the quotient from the division.

Memory Registers

Unlike the registers specified above, these registers consist of different functionality specifically for reading and writing interactions with the memory. Only the memory data register can write values directly to the datapath.

Memory Data Register (MDR)

The MDR stores the value outputted or inputted into the memory and places it onto the datapath. Since the MDR takes two different inputs to D, a multiplexer was used. This is further elaborated in the implementation part of the report.

Memory Address Register (MAR)

The MAR is a regular 32-bit register. The lower 8 bits of its output are used to select where we should be pointing to in the memory. The location can either be read from or written to them.

Memory (RAM)

The memory functions are similar to a register that returns a value depending on where the MAR points to. It takes in the MAR output and MDR output as well as read, write, clock. It has one singular output that also goes to the MDR multiplexer.

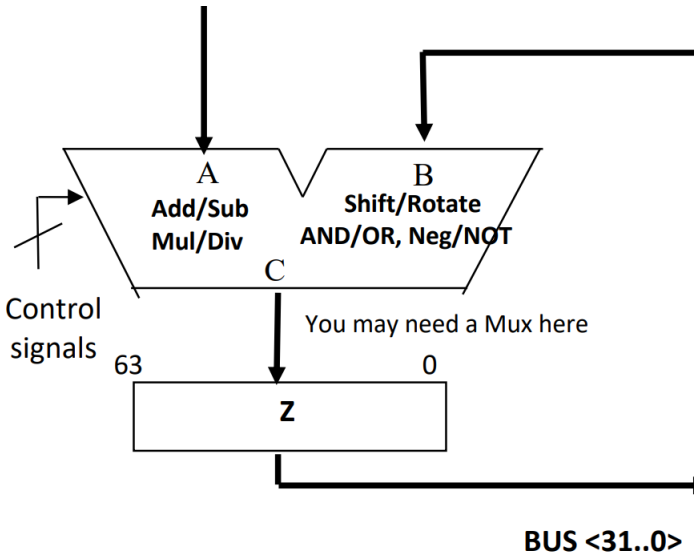
Input/Output

In-Port And Out-Port Registers

The I/O registers accept external inputs and outputs of external data. The inport can place its contents on the datapath, and the outport register can receive data from the datapath.

ALU Components

Two different ALUs were created and tested in our design. The first ALU is an asynchronous module-based ALU that uses individual modules for each type of computation. This file was called alu.v. All modules would be fed the ALU's inputs. Then, depending on the ALU opcode, the ALU selects which output from its submodules to place in the Z register. The second ALU computed addition, subtraction, multiplication, and division asynchronously (in the same way the first ALU did), while the rest were synchronous and computed within the ALU module. This Verilog file was named alu_test.v and will be referenced later in the report.



An intermediate register Y is wired to the input of the A value of the ALU to preserve the first value being entered into the ALU while the second value is placed on the datapath. The Z register was broken into two output registers, Zlow and Zhigh, to split the lower and higher output bits, respectively.

The control signals that went into the ALU included the opcode for determining which ALU operation to execute/select. Other inputs include clear, clock, increment PC flag, and the CONN_out (for branching).

ALU Operations

The ALU can perform the following operations: addition, subtraction, multiplication, division, shift right, shift left, shift right arithmetic, rotate right, rotate left, and, or, xor, negation, nor, and not. The Xor operation was not specified by the project outlines but added to the design as a bonus.

In all ALU implementations, addition, subtraction, multiplication, and division are computed in their respective modules. Each of these operations is executed using a separate module file (asynchronously, alu.v) or within the ALU component after being selected by the opcode (alu_test.v). The addition module uses a hierarchal carry look-ahead adder. Subtraction implements the addition module and negates the B input to create two complement values. The multiplication module uses the booth algorithm with bit pair recoding. The division algorithm uses the non-restoring algorithm. The code used in these modules can be found in the appendix section ALU Modules.

Instruction Specifications and Behaviour

Our Mini SRC computer could compute R-type, I-type, J-type, B-type, and M-type instructions. The computer could run the following R-type instructions: addition, subtraction, and, or, xor, shift left, shift right, shift right arithmetic, rotate left, rotate right, and load. I-type includes: addi, andi, ori, multiplication, division, negation, and not. B-type included: brzr, brnz, brmi, and brpl. J-type: in, out, mfhi, mflo, jr, and jal. Lastly, our M-type instructions: nop and halt. A list of all instructions and their opcodes can be found in the Instruction Opcode List section in the There are still some limitations and challenges that we would like to address in our future work. We have divided our future work into two categories: short term and long term.

Short term:

- Port to FPGA: We've only simulated on a Model Sim. We want to port our design to a FPGA board and run it on real hardware. This would allow us to verify the correctness of our design in a more realistic environment and measure our computer's actual speed and power consumption.
- Include more instructions: We would like to include more instructions, such as NOR or I-Type instructions. This would increase the functionality and expressiveness of our computer. This would require modifying the ALU and the control unit to support the new instructions.
- Tristate logic: Our computer uses multiplexers to select the inputs and outputs of different components. However, multiplexers are not very efficient in terms of area and power. We want to replace the multiplexers with tristate logic, which can enable or disable the output of a component depending on a control signal. This would reduce the number of gates and wires in our design and improve the area and power efficiency.
- Optimize design: Our mini SRC is not fully optimized regarding speed and area. We want to apply some optimization techniques, such as pipelining, to improve the performance and efficiency of our mini SRC. For example, we could pipeline the fetch-decode-execute cycle to increase the throughput of our mini SRC.
- Speed up clock and finite states in the control unit: Our mini SRC uses a relatively slow clock and the states are even slower. We would need to redesign our control unit to reduce the number of states and transitions and make them faster. We want to decrease the time spent in each state to achieve higher performance.

Long term:

- Bidirectional bus: Our mini SRC uses a single bus for data and addresses. This limits the speed and flexibility of the computer. We want to implement a bidirectional bus that can carry data

and addresses simultaneously. This would allow us to use one bus for both memory access and I/O operations, which would increase the effiecieny of our design.

Assembly language to .hex file: We have to manually write the .hex file containing our assembly code's binary representation. This is tedious and error-prone. We want to develop a tool that can automatically translate our assembly code into .hex file. This would make it easier for us to write and test our programs on our computers.

.

These instructions had unique opcodes decoded by the control unit for signal sending and the select and encode so that the correct registers can also be interacted with. How these components were implemented is explained in further detail within the Project Design and Implementation portion.

Datapath and Bus

All the above registers and the ALU were wired to the datapath and bus for proper data transfer functionality. Using a 32 to 1 MUX and a 32 to 5 encoder, the output of these registers could be selected and placed onto the Bus. With this implementation, only one register can write its values onto the bus at a time. The datapath is also the top-level entity of the whole system and interacts externally through the input and output ports of the computer.

Control Unit

The control unit we have behaves as a finite state machine. The first three cycles it performs are the fetch instruction cycles. Subsequently, depending on the value read into the IR, it will enter the execution cycle, which will take 0 to 5 more cycles to finish the instruction. The control unit outputs corresponding control signals to the datapath, which selects which registers can write data onto the bus, which registers accept data from the bus, ALU opcodes, or specific functionality flags that are further elaborated upon in theThe ALU's output is a 64-bit value stored in the C register. This value is divided into two 32-bit halves: the upper half goes to the ZHigh register, and the lower half goes to the ZLow register. The Bus gets the value from the ZLow register in the final instruction cycle for most operations. For Multiplication or Division, however, the HI and LO registers get the values from the ZHigh and ZLow registers, respectively. The HI and LO register data can be accessed using mfhi and mflo instructions.

Control Unit section of the Project Design and Implementation section.

Project Design and Implementation

Datapath

Our datapath is our top-level entity and immediately interacts with inputs from our test bench and I/O ports. The datapath wires all of our other entities together. The datapath takes in a `reset,` `clk,` `stop,` and `InPortData.` The InPortData is the external input from a device connected to the computer. The module outputs the `OutPortData` and `run.` The `run` output is high when the computer executes code. The reset signal will set all values in the registers to 0 and reset the ALU. The clk signal is the rate at which the clock runs for the whole system and is dictated by the test bench. The stop signal halts all program execution, similar to the IR reading a HALT opcode.

After the input-output declarations, the module has multiple declarations for functionality signals, such as R1-15in signals, R1-15 out signals, etc. All the Bus multiplexer inputs are also instantiated so that all

the registers can be passed to the bus module, which depending on the Rout signal, will put the output of that register on the BusMuxOut wire. The BusMuxOut wire is wired to both data inputs to the ALU, input to all registers, input to the MDR register, input to the PC register, and input to the Outport register. Subsequently, more minor data wires are declared. A 64-bit wire CRegOut is the output from the ALU and is split into the Zlow and Zhigh registers, accessed in the last cycles of any IR, and placed into their corresponding register. Wire R15jal is the R15 in wire or'd with the Jal_flag wire if the Jal instruction is called and we need to place the PC in the return register. The Jal_flag is an output of the control unit. The branch_flag wire overrides the ALU and causes it to add the following two values. This is used when we want to add an offset to the PC value for conditional branching. The branch_flag remains low if the branching conditions aren't met. In this case, no addition would occur in the ALU.

Following the internal wire declarations is the module instantiation for the control unit, R0 register, R1 to R15 registers, HI and LO registers, Zhigh and Zlow registers (takes ALU input from CRegOut), PC, in-port and out-port, MDR, MAR, Y register (feeds the ALU), RAM, CONN_FF logic, select and encode logic, the bus, and finally the ALU.

The datapath feeds the control unit, the Irdata, and signals for clk, reset, and stop. In turn, it outputs read/write, BAout, Rin, Rout, Gra, Grb, Grc, CONN_in, MARin, MDRin, HIin, LOin, Yin, Zin, PCin, IRin, incPC, InPortIn, OutPortIn, HIout, LOout, ZLowOut, ZHighOut, MDRout, Cout, InPortOut, PCout, Jal_flag, and alu_opcode data.

The general register (R1 to R15) takes input from the BusMuxOut and puts their data onto the bus via a multiplexer and encoder in the bus module. Hence all of their outputs are wired to the bus module. The R0 register is unique in that it also takes in the BAout signal, which determines if its output should be forced to 0 if BAout is high. The HI, LO, and PC registers also function and are wired similarly.

The ALU's output is a 64-bit value stored in the C register. This value is divided into two 32-bit halves: the upper half goes to the ZHigh register, and the lower half goes to the ZLow register. The Bus gets the value from the ZLow register in the final instruction cycle for most operations. For Multiplication or Division, however, the HI and LO registers get the values from the ZHigh and ZLow registers, respectively. The HI and LO register data can be accessed using mfhi and mflo instructions.

Control Unit

Our control unit was implemented as a finite-state machine. There are four primary states the computer can be within. These states are the reset state, halt state, fetch state, and execution state.

The first state the machine can enter is the reset state. This state can be accessed if and only if the `reset` signal is set to high. This is an external input to the datapath formed by either the testbench or wired to the reset pin on the FPGA. The reset state sets all register values to 0, the ALU state to 0, and all control flags to 0, by setting the `clr` signal to high. Once the `reset` signal is set back to low, the computer begins entering the fetch states.

The computer can enter the halt state through two means. Either the computer reads a HALT instruction in the IR, or the `stop` signal is set to high. When the computer enters the halt state, it stops executing and gets stuck in an infinite loop until `reset` is set to high.

There are three sub-states within the fetch states, Fetch0, Fetch1, and Fetch2. These will always run at the beginning of the instruction cycle. The first state, Fetch0, can be arrived at by either reset (as

specified above) or by the computer that has gotten to the last instruction in the execution phase and has now looped back to Fetch0. In the Fetch0 state, the PC value is placed on the Bus and is read in by the MAR and the ALU. Also, the incPC flag is set to high, which forces the ALU to add 1 to whatever value is in the B input to the ALU. In this state, Zin is also high, so the increment PC value is placed in the ZLowOut register. In Fetch1, the read and MDRin signals are set high so that the PC value placed in the MAR (in Fetch0) is the location from which we read the instruction. Also, the ZLowOut and PCin signals are set high so that the PC can now hold the incremented value, which points to the following memory location we will be reading from when we return to the fetch state again. Lastly, in Fetch2, the value in the MDR register is placed on the Bus and is read into the IR register for decoding.

Lastly, the control unit will enter the execution state. Within the execution state, there are 78 substates. There are so many substates because all the substates are unique per instruction, and some instructions have more substates than others. However, most substates follow similar pattens or have the same functionality. Most ALU operations generally involve placing either two (Rb and Rc) or one value (Rb) in the ALU and passing the ALU its correct opcode per instruction in the first two execution instructions. Then the last instruction is for fetching the value and placing it in the Ra register. If it's an immediate ALU operation, the C value is placed on the Bus instead of the Rb register. If it's not or negate, only one value from the Rb register is placed on the bus. On average, ALU operations have three execution states, so they take a total of 6 cycles. Store instructions take four instruction cycles; the first two add the offset to the value, and the last store the register value in the specified location. Load instructions take five cycles; two are used to add the offset to the location, three cycles to load the value from memory and then store it in the specified register. Load immediate takes three cycles, 2 for adding the immediate value by the offset and the last one to store the value in a register. Branch instructions take only four instructions. Note that it was specified by the specification documentation that branching normally would have five execution states; however, our implementation of the branching flag allows for the reduction of one state. The Ra register value is placed on the bus in the first branching state. This value is going to enter the CONN_ff logic and will cause the logic to send a signal to the ALU if the two values should be added. The PC value is then placed in the A register in the ALU, and the C sign extended portion of the IR is placed in the B input to the ALU. If the branching condition is met, then CON_ff will give the ALU the signal to either add or not add the two values. The output of the ALU is then stored in the PC. If the addition occurred in the ALU, the branching occurred, and the PC is now pointing to a new location. The jump instruction takes only one cycle, while Jump and link take two. This is because JAL requires placing the PC value in the R15 register (using the Jal flag) before jumping. in, out, mfhi, and mflo all need one execution state to move a singular value from one register and place it in another. Only one NOP instruction state that does nothing but return to Fetch0.

Additional Hardware/features

Register R0

The R0 register has slightly different behaviour from other registers. It has an addition BAout signal that can force it to output a 0 even if it's stored data is not equal to 0. This is used in I-Type instructions for loads and stores. If a register specifies no offset value, then the Rb selection bits will be all 0 and will correlate to using R0 to add the offset. But if R0 holds a value, we can't use it as an offset. So, the BAout will use the R0 register to ensure that a 0 is written onto the bus. Refer to the R0 Register in the Appendix to see how this was implemented.

The MDR Register

The MDR register is a module that contains a regular register and a two-to-one multiplexer. The two-to-one multiplexer is used for determining if the value from the bus should be placed inside the MDR or if the value coming from the memory should be placed within. Hence the module takes in two extra data inputs and one extra select signal. Refer to MDR Register in the There are still some limitations and challenges that we would like to address in our future work. We have divided our future work into two categories: short term and long term.

Short term:

- Port to FPGA: We've only simulated on a Model Sim. We want to port our design to a FPGA board and run it on real hardware. This would allow us to verify the correctness of our design in a more realistic environment and measure our computer's actual speed and power consumption.
- Include more instructions: We would like to include more instructions, such as NOR or I-Type instructions. This would increase the functionality and expressiveness of our computer. This would require modifying the ALU and the control unit to support the new instructions.
- Tristate logic: Our computer uses multiplexers to select the inputs and outputs of different components. However, multiplexers are not very efficient in terms of area and power. We want to replace the multiplexers with tristate logic, which can enable or disable the output of a component depending on a control signal. This would reduce the number of gates and wires in our design and improve the area and power efficiency.
- Optimize design: Our mini SRC is not fully optimized regarding speed and area. We want to apply some optimization techniques, such as pipelining, to improve the performance and efficiency of our mini SRC. For example, we could pipeline the fetch-decode-execute cycle to increase the throughput of our mini SRC.
- Speed up clock and finite states in the control unit: Our mini SRC uses a relatively slow clock and the states are even slower. We would need to redesign our control unit to reduce the number of states and transitions and make them faster. We want to decrease the time spent in each state to achieve higher performance.

Long term:

- Bidirectional bus: Our mini SRC uses a single bus for data and addresses. This limits the speed and flexibility of the computer. We want to implement a bidirectional bus that can carry data and addresses simultaneously. This would allow us to use one bus for both memory access and I/O operations, which would increase the effiecieny of our design.

Assembly language to .hex file: We have to manually write the .hex file containing our assembly code's binary representation. This is tedious and error-prone. We want to develop a tool that can automatically translate our assembly code into .hex file. This would make it easier for us to write and test our programs on our computers.
section for the implementation.

RAM

A 512 x 32 RAM was implemented into the design. The module functions similarly to a register but has three more inputs: read, write, and MARout. The read and write ports are control signals that indicate

whether the memory is being read from or written to. The MARout port is the memory address register output that specifies the memory location address to be accessed. The module uses a reg array called mem to store 512 words of 32 bits each. The initial block is used to initialize the mem array with values from a hex file using the $readmemh system task. The module uses an assign statement to assign the Q output to either a high-impedance value (ZZZZZZZZ) or the value of mem[MARout], depending on the write and read signals. The high-impedance value indicates that any source does not drive the output and can be overridden by another driver. The implementation can be located in RAM in the There are still some limitations and challenges that we would like to address in our future work. We have divided our future work into two categories: short term and long term.

Short term:

- Port to FPGA: We've only simulated on a Model Sim. We want to port our design to a FPGA board and run it on real hardware. This would allow us to verify the correctness of our design in a more realistic environment and measure our computer's actual speed and power consumption.
- Include more instructions: We would like to include more instructions, such as NOR or I-Type instructions. This would increase the functionality and expressiveness of our computer. This would require modifying the ALU and the control unit to support the new instructions.
- Tristate logic: Our computer uses multiplexers to select the inputs and outputs of different components. However, multiplexers are not very efficient in terms of area and power. We want to replace the multiplexers with tristate logic, which can enable or disable the output of a component depending on a control signal. This would reduce the number of gates and wires in our design and improve the area and power efficiency.
- Optimize design: Our mini SRC is not fully optimized regarding speed and area. We want to apply some optimization techniques, such as pipelining, to improve the performance and efficiency of our mini SRC. For example, we could pipeline the fetch-decode-execute cycle to increase the throughput of our mini SRC.
- Speed up clock and finite states in the control unit: Our mini SRC uses a relatively slow clock and the states are even slower. We would need to redesign our control unit to reduce the number of states and transitions and make them faster. We want to decrease the time spent in each state to achieve higher performance.

Long term:

- Bidirectional bus: Our mini SRC uses a single bus for data and addresses. This limits the speed and flexibility of the computer. We want to implement a bidirectional bus that can carry data and addresses simultaneously. This would allow us to use one bus for both memory access and I/O operations, which would increase the effiecieny of our design.

Assembly language to .hex file: We have to manually write the .hex file containing our assembly code's binary representation. This is tedious and error-prone. We want to develop a tool that can automatically translate our assembly code into .hex file. This would make it easier for us to write and test our programs on our computers.
Verilog Code section.

Multiplication Module

The multiplier we implemented used Booth's algorithm with bit pair encoding. The module outputs a 64-bit value instead of 32 bits to place the value in the LO and HI registers. The module was implemented by looping over 16 times and shifting the multiplier twice to the right each time. We look at the bottom 3 bits of the multiplier, which is our bit pattern. We perform different operations depending on the bit pattern's value through each loop. The multiplicand will be shifted twice to the left with each iteration. A temporary register is used to accumulate the sum of each loop, which will then be assigned as the product output. Depending on the bit pattern, we do the following operations:

- If the bit pattern is 001 or 010, it means that the multiplier has a 0 followed by a 1. In this case, the temporary register is increased by the multiplicand, which is equivalent to adding one times the multiplicand to the product.
- If the bit pattern is 011, it means that the multiplier has two consecutive 1s. In this case, the temp is increased by twice the multiplicand, which is equivalent to adding two times the multiplicand to the product.
- If the bit pattern is 100, it means that the multiplier has two consecutive 0s. In this case, the temp is decreased by twice the multiplicand, which is equivalent to subtracting two times the multiplicand from the product.
- If the bit pattern is 101 or 110, it means that the multiplier has a one followed by a 0. In this case, the multiplicand decreases the temp, which is equivalent to subtracting one times the multiplicand from the product.

Finally, after exiting the loop, the code assigns the temp value to the product output. The implementation of this code in Verilog can be found in the Multiplication (Booths Algorithm With Bitpair Encoding) section of the There are still some limitations and challenges that we would like to address in our future work. We have divided our future work into two categories: short term and long term.

Short term:

- Port to FPGA: We've only simulated on a Model Sim. We want to port our design to a FPGA board and run it on real hardware. This would allow us to verify the correctness of our design in a more realistic environment and measure our computer's actual speed and power consumption.
- Include more instructions: We would like to include more instructions, such as NOR or I-Type instructions. This would increase the functionality and expressiveness of our computer. This would require modifying the ALU and the control unit to support the new instructions.
- Tristate logic: Our computer uses multiplexers to select the inputs and outputs of different components. However, multiplexers are not very efficient in terms of area and power. We want to replace the multiplexers with tristate logic, which can enable or disable the output of a component depending on a control signal. This would reduce the number of gates and wires in our design and improve the area and power efficiency.
- Optimize design: Our mini SRC is not fully optimized regarding speed and area. We want to apply some optimization techniques, such as pipelining, to improve the performance and efficiency of our mini SRC. For example, we could pipeline the fetch-decode-execute cycle to increase the throughput of our mini SRC.

- Speed up clock and finite states in the control unit: Our mini SRC uses a relatively slow clock and the states are even slower. We would need to redesign our control unit to reduce the number of states and transitions and make them faster. We want to decrease the time spent in each state to achieve higher performance.

Long term:

- Bidirectional bus: Our mini SRC uses a single bus for data and addresses. This limits the speed and flexibility of the computer. We want to implement a bidirectional bus that can carry data and addresses simultaneously. This would allow us to use one bus for both memory access and I/O operations, which would increase the effiecieny of our design.

Assembly language to .hex file: We have to manually write the .hex file containing our assembly code's binary representation. This is tedious and error-prone. We want to develop a tool that can automatically translate our assembly code into .hex file. This would make it easier for us to write and test our programs on our computers.
.Conclusion and Future Work

Division Module

The division was achieved using the non-restoring algorithm. In the non-restoring algorithm, we first check whether the dividend or divisor is negative and flip their sign. Then we create a 64-bit register temp that's lower bits contain the value of the dividend. For 32 iterations (for the 32 bits in the quotient), we shift the temp register 1 to the left. Depending on whether the value in temp is negative or positive, we add the divisor to the top 32 bits or subtract it respectively. Then, if the value is still negative, we set the 0th bit of the temp register to 0; otherwise, it's set to 1. The code for the non-restoring algorithm can be found in the Division (Nonrestoring Algorithm) section of the There are still some limitations and challenges that we would like to address in our future work. We have divided our future work into two categories: short term and long term.

Short term:

- Port to FPGA: We've only simulated on a Model Sim. We want to port our design to a FPGA board and run it on real hardware. This would allow us to verify the correctness of our design in a more realistic environment and measure our computer's actual speed and power consumption.
- Include more instructions: We would like to include more instructions, such as NOR or I-Type instructions. This would increase the functionality and expressiveness of our computer. This would require modifying the ALU and the control unit to support the new instructions.
- Tristate logic: Our computer uses multiplexers to select the inputs and outputs of different components. However, multiplexers are not very efficient in terms of area and power. We want to replace the multiplexers with tristate logic, which can enable or disable the output of a component depending on a control signal. This would reduce the number of gates and wires in our design and improve the area and power efficiency.
- Optimize design: Our mini SRC is not fully optimized regarding speed and area. We want to apply some optimization techniques, such as pipelining, to improve the performance and efficiency of our mini SRC. For example, we could pipeline the fetch-decode-execute cycle to increase the throughput of our mini SRC.

- Speed up clock and finite states in the control unit: Our mini SRC uses a relatively slow clock and the states are even slower. We would need to redesign our control unit to reduce the number of states and transitions and make them faster. We want to decrease the time spent in each state to achieve higher performance.

Long term:

- Bidirectional bus: Our mini SRC uses a single bus for data and addresses. This limits the speed and flexibility of the computer. We want to implement a bidirectional bus that can carry data and addresses simultaneously. This would allow us to use one bus for both memory access and I/O operations, which would increase the effiecieny of our design.

Assembly language to .hex file: We have to manually write the .hex file containing our assembly code's binary representation. This is tedious and error-prone. We want to develop a tool that can automatically translate our assembly code into .hex file. This would make it easier for us to write and test our programs on our computers.

.

Select and Encode Logic

The Select and Encode logic partitions the register opcodes from the IR register. It can then be used with Gra, Grb, and Grc signals to select which register we wish to select. Once a Rin, Rout, or BAout signal is high along with the corresponding Gra, Grb, or Grc, the logic will set the corresponding R#in signal high or R#out signal high. The select and encode logic also sign extends the C value in the IR out. To see implementation, refer to Select and Encode Logic.

Conditional Branching Logic

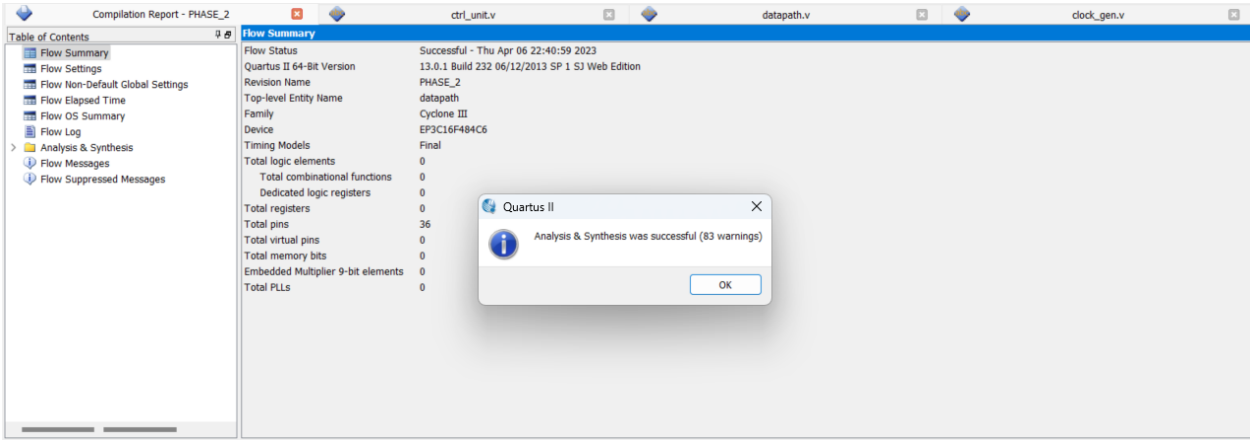
The Condition Branching Logic Module (CONN_ff) checks the branching conditions in the IR register and compares the Ra register value to the branching condition. If the value meets the branching criteria, it sends a high signal to the ALU to add the offset to the PC. Refer to CON FF logic for implementation in Verilog.

Evaluation Results

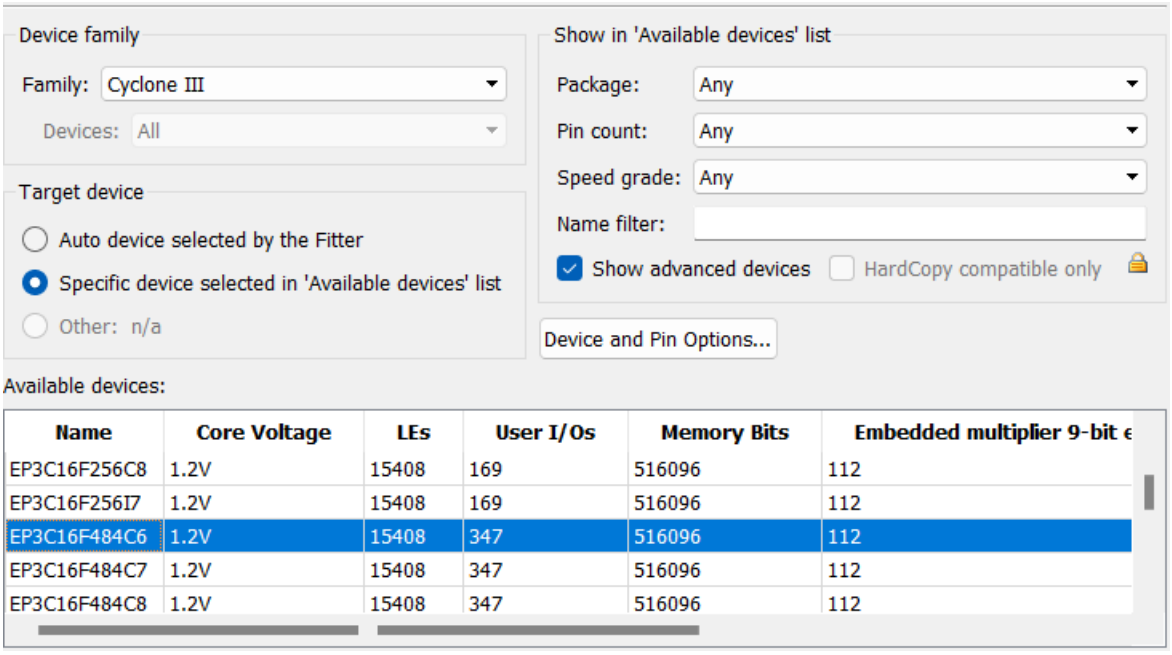
Maximum frequency of operation in simulation.

To determine the maximum frequency of operation in simulation, the following steps should be executed:

1. Synthesize the design in Quartus.



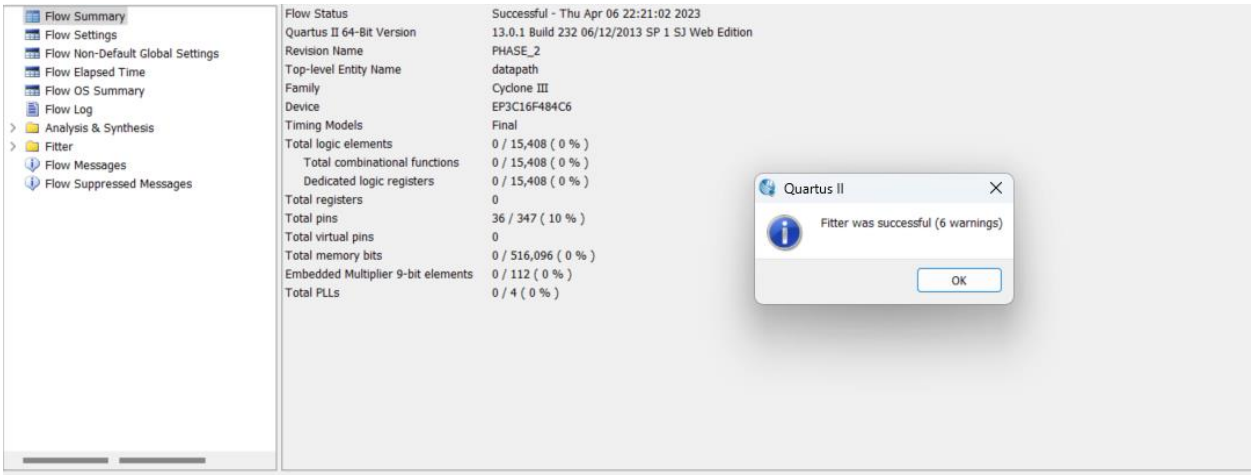
2. Assign a specific FPGA Device in Quartus



3. **Assign all the required input/output pins to the desired FPGA pins.**

N/A: Pin assignment was expected in Phase 4.

4. **Route the design in the FPGA using the Fitter and optimize performance:**



5. **Review the TimeQuest Timing Analyzer to determine the maximum frequency (Fmax) on our design.**

Note, since we got up to Phase 3 in this project, we did not set up the pin assignments, nor the .sdc file needed for TimeQuest Timing Analyzer.

Average Cycle Per Instruction (CPI) for the test programs

The first instruction occurs at 70ns due to a delay in resetting and clearing all the modules. Then, the final instructions end at 10,590 ns (including the xor for bonus). Considering a total of 40 instructions in the program, this leads to an average of 263 ns per instruction.

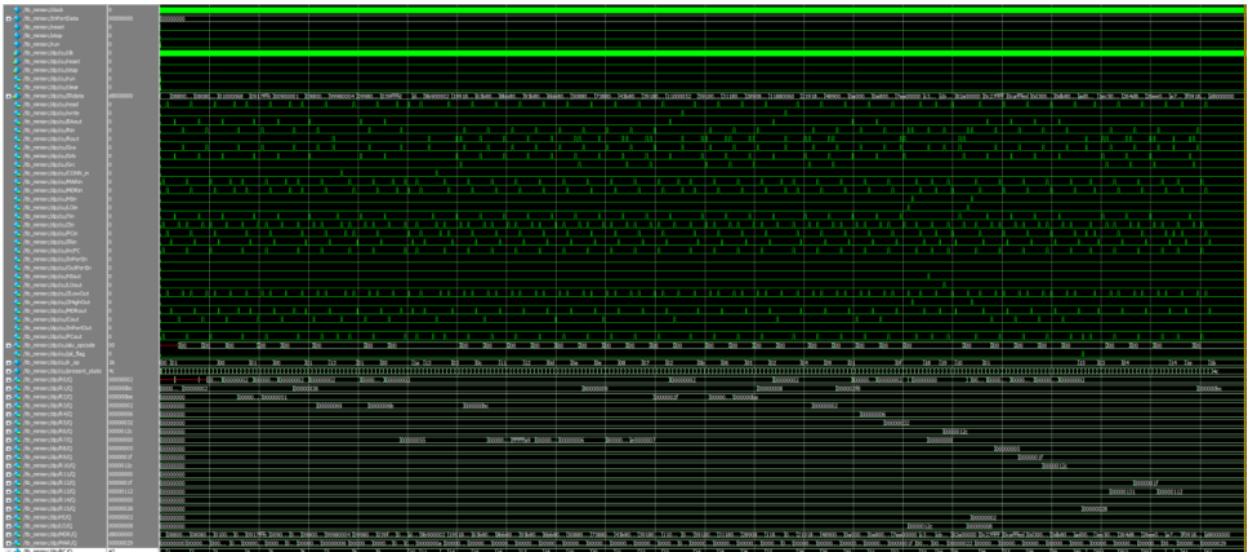


Figure 1: Entire wave form consisting of 40 instructions.

In addition, the duration was the xor instruction was focused on more detail. The first instruction is fetched at 10,310 ns and finishes executing at 10,550 ns; thus, its total time is 240 ns. Note that the cycles were not considered due to timing issues since the clock was run on override at a cycle of 2 ns.

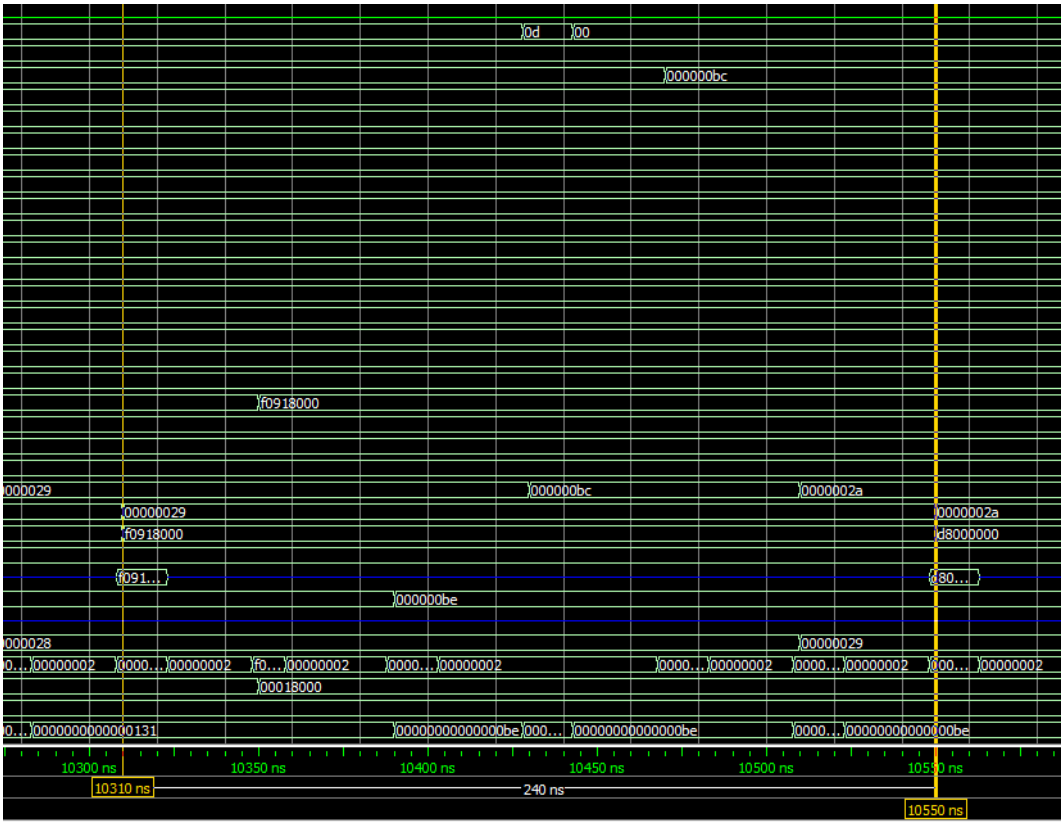


Figure 2: Length of xor instruction from the beginning of fetch to the end of execution.

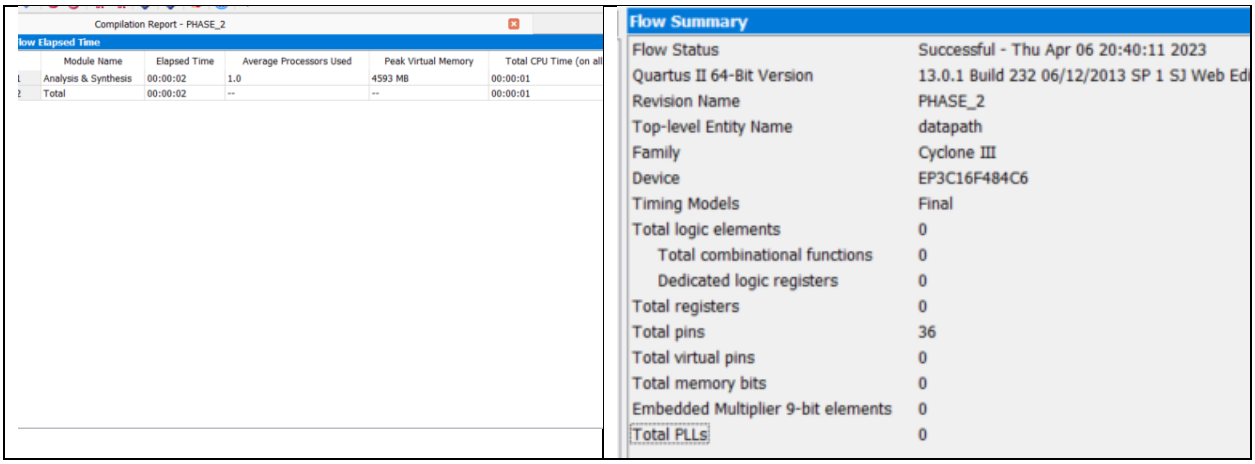
Percentage of Chip Area in Design

Flow Summary	
Flow Status	Successful - Thu Apr 06 20:58:07 2023
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	PHASE_2
Top-level Entity Name	datapath
Family	Cyclone III
Device	EP3C16F484C6
Timing Models	Final
Total logic elements	0 / 15,408 (0 %)
Total combinational functions	0 / 15,408 (0 %)
Dedicated logic registers	0 / 15,408 (0 %)
Total registers	0
Total pins	36 / 347 (10 %)
Total virtual pins	0
Total memory bits	0 / 516,096 (0 %)
Embedded Multiplier 9-bit elements	0 / 112 (0 %)
Total PLLs	0 / 4 (0 %)

There are only 36 pins utilized out of the 347 available on the chip, thus, our design only uses 10% of the total pins.

Other Metrics

Our project used 36 pins and 4593 MB of Peak Virtual Memory.



Phase 1 Bonus Metrics

In Phase 1, we experimented with different modules for each of the operations and made a clean module alu_test that contains the logic for all these components seen below. We made a special ALU testbench that went through each opcode for further verification. In addition, we made a nor arithmetic logic.

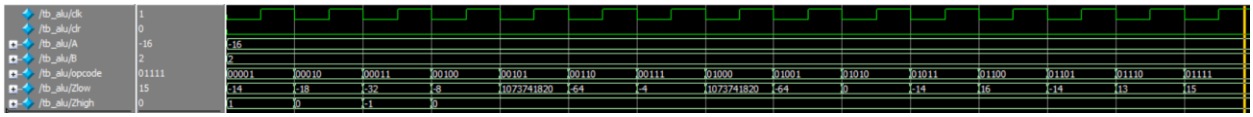


Figure 3: ALU testbench for values -16 and 2.

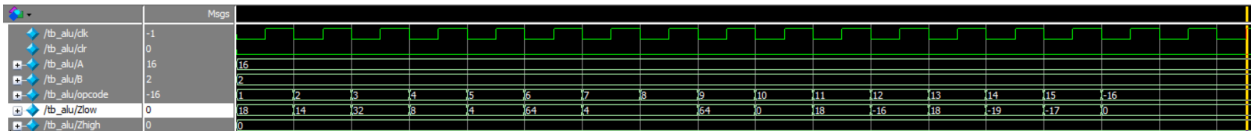


Figure 4: ALU testbench for values 16 and 2.

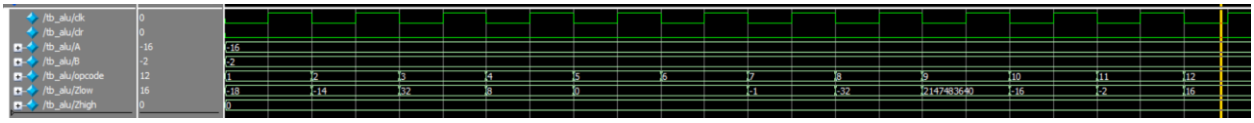


Figure 5: ALU testbench for values -16 and -2.

Phase 3 Bonus Metrics

As previously mentioned in phase 3, we added a xor instruction to our memory, which performs a bitwise exclusive OR operation on two operands. The Xor instruction was not part of the original CPU specs document but was added later. The Xor instruction was implemented by adding a new IR opcode (11110 in the first 5 bits), a new XOR alu function, and new XOR execution states to the control unit.

Discussion

This semester-long intensive project required us to design a simple RISC-style computer which was broken down into four phases, each more complex than the previous phase. This discussion will primarily focus on the motivation behind each phase of the project, the challenges we encountered and the extra implementations that made our design unique.

Phase 1 was fundamentally simple as it involved designing a Datapath, an arithmetic logic unit (ALU), the registers, multiplexers, and encoders/decoders. In addition, we had to design test benches to verify the functionality of the operations performed in the ALU. This was the most challenging as it was our first true introduction to Verilog language, and any minor logic errors in any of the modules led to substantial errors that took some time to debug. One of the challenges was how we wanted to design our ALU; initially, we had separate modules for each operation performed. Although this worked well to make our code very modular, it required instantiation of each operation module within our ALU and took a whole to debug issues. We evolved our design to place most of the operations within the ALU module except for multiplication, division and addition, as these were explicitly required as separate modules. This improved the readability of our code and eased the debugging process. In addition, we had an issue with multiplication which was revealed in the testbench. We assumed there was an error within our logic, and after verifying it line-by-line, it turned out to be an instantiation issue within the ALU module.

Phase 2 was enjoyable as it involved the design and functional simulation of the Mini SRC datapath, in particular the “Select and Encode” logic, “Memory Subsystem,” “CON FF” logic, and “Input/Output” ports, as well as load/store instructions, branch and jump instructions and immediate instructions. Some more complex challenges arose in this phase. For instance, the initial design of our 32-to-5 encoder involved a default case of don’t cares (5’bx). This minor issue leads to issues within Phase 2, trying to resolve unexpected outputs. It was a rigorous process to debug this flaw from an introductory module used across our whole CPU design. The process involved tracing the issue by determining which signals were high in the ModelSim simulation, then verifying each module and going over the code.

Furthermore, we experienced a sign extension issue between the two initial phases. In phase 1, it described *c-sign extended* as an input to the bus, which derived the logic for *Cout*. This was different in phase 2, in which we had to rewrite our logic as *c-sign extended* was now coming from the select and encode logic based on the 17^{th} bit of the IR register. To continue, we experienced issues with reading in the memory from the ram for each instruction. To resolve this, we filled our ram.hex file with zeroes. Then within our ram.hex file, we declared the opcode of our instruction directly, using *“ram[0] = “#hexopcode”.* Finally, since we sped up our clock from the standard, we could not use the clock cycle approach to increment our program counter (PC), thus we modified our ALU to take in incPC, to modify the PC register.

Finally, the motivation behind Phase 3 of the CPU Design Project was to gain a thorough understanding of the Control Unit's role in orchestrating the Mini SRC's operation. By analyzing the interaction between the Control Unit and the datapath components, the project aimed to ensure the seamless execution of instructions and enhance the Mini SRC's overall performance. Furthermore, this phase provided an opportunity to delve into the intricacies of control signal generation, branching and instruction flow management, and external inputs handling. This phase was especially challenging since the control unit was the longest testbench we made, consisting of around 40 instructions. Thus, from the initial run, it was clear there was a lot of debugging involved. The first major challenge turned out to be timing issues.

Our simulation revealed that the only instruction that was performed was the base case, load instruction, with an IR opcode of 5'b00000. After zooming into the simulation into the nanosecond degree, we finally discovered our simulation was off by a fraction of a hair. As frustrating as this was, since it required numerous hours to make this discover this error, it was a quick fix, which simply involved modifying the delays in our control unit and running our clock on a 2 ns cycle. Lastly, the control unit required lots of parameters for each all the states, IR opcodes, ALU opcodes and an average of three opcodes for each instruction. Unfortunately, this led to some duplicate opcodes used, which did not lead to any syntax errors, this was seen in repercussions in our simulation with unexpected outcomes for instructions. Once again, this was a minor but tedious issue with this phase as it involved reading through over a thousand lines of code countless times.

Bonus Functionality

Phase 1

Within phase 1, we wrote each ALU operation in two different ways and included additional operations such as XOR and NOR. Each of these operations is executed using a separate module file (asynchronously) or within the ALU component after being selected by the opcode. This meant we had two different ALU modules, that were both functional but used different basic modules, for instance, in the *AND* operation, we made a module that performed 32-bit bit-wise AND, which was used in our *alu.v* file and a case condition for logical and that was used in our second ALU, *alu_test.v (synchronous)*:

```
module and32 (a, b, c);
    input wire [31:0] a, b;
    output reg [31:0] c;
    reg [31:0] temp;
    integer i;

    always @(*) begin
        for (i = 0; i < 31; i = i + 1) begin
            temp[i] = ((a[i])&(b[i]));
        end
        c = temp;
    end

endmodule
```

Figure 6: bit-wise AND module for alu.v

```
log_and: begin
    C [31:0] <= A & B;
    C [63:32] <= 32'd0;
end
```

Figure 7: Logical AND case condition in alu_test.v

In addition, we performed bit-pair booth multiplication in a clean and elegant manner for easy readability and comprehension. This practice was reflected across all our code in this project.

Phase 2

In phase 2, we continued the use of best practices when writing our code and optimized the branch instructions by using six states instead of seven.

Phase 3

For phase 3, we included additional instructions such as *XOR* in our RAM. This was explained in detail in Other Metrics section of this report.

Conclusion and Future Work

This project aimed to design, simulate, implement, and verify a simple RISC Computer (Mini SRC) which comprised a simple RISC processor, memory, and I/O. The primary motivation behind this project was to gain a comprehensive understanding of the intricacies involved in designing a RISC-based computing system. Additionally, the project aimed to provide valuable insights into the integration of various datapath components, which were essential for the overall functionality of the Mini SRC. Lots of challenges were encountered with this project, and it was well worth the effort we put into it. We exceeded our expectations by completing phase 3 and were surprised by how much we learned to get there. Without a doubt, this was one of the hardest and most rewarding projects we have completed in our academic careers. The content was interesting and it gave us a strong introduction to the process development cycle in hardware projects.

Future Work

There are still some limitations and challenges that we would like to address in our future work. We have divided our future work into two categories: short term and long term.

Short term:

- Port to FPGA: We've only simulated on a Model Sim. We want to port our design to a FPGA board and run it on real hardware. This would allow us to verify the correctness of our design in a more realistic environment and measure our computer's actual speed and power consumption.
- Include more instructions: We would like to include more instructions, such as NOR or I-Type instructions. This would increase the functionality and expressiveness of our computer. This would require modifying the ALU and the control unit to support the new instructions.
- Tristate logic: Our computer uses multiplexers to select the inputs and outputs of different components. However, multiplexers are not very efficient in terms of area and power. We want to replace the multiplexers with tristate logic, which can enable or disable the output of a component depending on a control signal. This would reduce the number of gates and wires in our design and improve the area and power efficiency.
- Optimize design: Our mini SRC is not fully optimized regarding speed and area. We want to apply some optimization techniques, such as pipelining, to improve the performance and efficiency of our mini SRC. For example, we could pipeline the fetch-decode-execute cycle to increase the throughput of our mini SRC.
- Speed up clock and finite states in the control unit: Our mini SRC uses a relatively slow clock and the states are even slower. We would need to redesign our control unit to reduce the number of states and transitions and make them faster. We want to decrease the time spent in each state to achieve higher performance.

Long term:

- Bidirectional bus: Our mini SRC uses a single bus for data and addresses. This limits the speed and flexibility of the computer. We want to implement a bidirectional bus that can carry data and addresses simultaneously. This would allow us to use one bus for both memory access and I/O operations, which would increase the effiecieny of our design.

- Assembly language to .hex file: We have to manually write the .hex file containing our assembly code's binary representation. This is tedious and error-prone. We want to develop a tool that can automatically translate our assembly code into .hex file. This would make it easier for us to write and test our programs on our computers.

Appendix

Verilog Code

Datapath

```
module datapath (
    //Test Bench inputs/outputs but goes through datapath to control unit
    output run,
    input wire clk, reset, stop,
    //Inport data from external device
    input [31:0] InPortData
);

wire clr;

wire R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, R8out, R9out,
R10out, R11out, R12out, R13out, R14out, R15out;
wire R0in, R1in, R2in, R3in, R4in, R5in, R6in, R7in, R8in, R9in, R10in, R11in,
R12in, R13in, R14in, R15in;
wire branch_flag;
//control unit wires
wire PCout, read, write, BAout, Rin, Rout, Gra, Grb, Grc, CONN_in, MARin, MDRin,
HIin, LOin, Yin, jal_flag, R15jal,
        Zin, PCin, IRin, incPC, InPortIn, OutPortIn, HIout, LOout, ZLowOut,
ZHighOut, MDRout, Cout, InPortOut;
wire [4:0] opcode;

wire [31:0] BusMuxIn_R0,
            BusMuxIn_R1,
            BusMuxIn_R2,
            BusMuxIn_R3,
            BusMuxIn_R4,
            BusMuxIn_R5,
            BusMuxIn_R6,
            BusMuxIn_R7,
            BusMuxIn_R8,
            BusMuxIn_R9,
            BusMuxIn_R10,
            BusMuxIn_R11,
            BusMuxIn_R12,
            BusMuxIn_R13,
            BusMuxIn_R14,
            BusMuxIn_R15;

wire [31:0] IRdata,
            BusMuxIn_HI,
```

```
            BusMuxIn_LO,
            BusMuxIn_Zhigh,
            BusMuxIn_Zlow,
            BusMuxIn_PC,
            BusMuxIn_MDR,
            BusMuxIn_InPort,
            RamDataOut,
            Yout,
            d_pc,
            MARout,
            BusMuxOut,
            C_sign_extended,
            OutPortOut;

wire [63:0] CRegOut;


//Control unit initialization
ctrl_unit cu (
    //Test Bench inputs/outputs but goes through datapath
    .run(run),
    .clear(clr),
    .clk(clk),
    .reset(reset),
    .stop(stop),
    //Datapath inputs/outputs
    .IRdata(IRdata),
    .read(read),
    .write(write),
    .BAout(BAout),
    .Rin(Rin),
    .Rout(Rout),
    .Gra(Gra),
    .Grb(Grb),
    .Grc(Grc),
    .CONN_in(CONN_in),
    .MARin(MARin),
    .MDRin(MDRin),
    .HIin(HIin),
    .LOin(LOin),
    .Yin(Yin),
    .Zin(Zin),
    .PCin(PCin),
    .IRin(IRin),
    .incPC(incPC),
```

```
    .InPortIn(InPortIn),
    .OutPortIn(OutPortIn),
    .HIout(HIout),
    .LOout(LOout),
    .ZLowOut(ZLowOut),
    .ZHighOut(ZHighOut),
    .MDRout(MDRout),
    .Cout(Cout),
    .InPortOut(InPortOut),
    .PCout(PCout),
    .alu_opcode(opcode),
    .jal_flag(jal_flag)
);

assign R15jal = (R15in | jal_flag);

regR0 R0 (BAout, clr, clk, R0in, BusMuxOut, BusMuxIn_R0); //input signal is
always 0 for R0 (special reg)
reg32bit R2 (clr, clk, R2in, BusMuxOut, BusMuxIn_R2);
reg32bit R3 (clr, clk, R3in, BusMuxOut, BusMuxIn_R3);
reg32bit R1 (clr, clk, R1in, BusMuxOut, BusMuxIn_R1);
reg32bit R4 (clr, clk, R4in, BusMuxOut, BusMuxIn_R4);
reg32bit R5 (clr, clk, R5in, BusMuxOut, BusMuxIn_R5);
reg32bit R6 (clr, clk, R6in, BusMuxOut, BusMuxIn_R6);
reg32bit R7 (clr, clk, R7in, BusMuxOut, BusMuxIn_R7);
reg32bit R8 (clr, clk, R8in, BusMuxOut, BusMuxIn_R8);
reg32bit R9 (clr, clk, R9in, BusMuxOut, BusMuxIn_R9);
reg32bit R10 (clr, clk, R10in, BusMuxOut, BusMuxIn_R10);
reg32bit R11 (clr, clk, R11in, BusMuxOut, BusMuxIn_R11);
reg32bit R12 (clr, clk, R12in, BusMuxOut, BusMuxIn_R12);
reg32bit R13 (clr, clk, R13in, BusMuxOut, BusMuxIn_R13);
reg32bit R14 (clr, clk, R14in, BusMuxOut,  BusMuxIn_R14);
reg32bit R15 (clr, clk, R15jal, BusMuxOut,  BusMuxIn_R15);

reg32bit HI (clr, clk, HIin, BusMuxOut, BusMuxIn_HI);
reg32bit LO (clr, clk, LOin, BusMuxOut, BusMuxIn_LO);
reg32bit ZHigh (clr, clk, Zin, CRegOut[63:32], BusMuxIn_Zhigh);
reg32bit ZLow (clr, clk, Zin, CRegOut[31:0], BusMuxIn_Zlow);


//PC reg initialization, using specific incPC input to increment PC by 1 each
time set to high
regPC PC (clr, clk, PCin, BusMuxOut, BusMuxIn_PC);

//Input and output ports added to datapath (p2)
```

```
reg32bit InPort (clr, clk, InPortIn, InPortData, BusMuxIn_InPort);
reg32bit OutPort (clr, clk, OutPortIn, BusMuxOut, OutPortOut);

//MDR reg initialization

MD_reg32 MDR (.clr(clr), .clk(clk), .read(read), .MDRin(MDRin),
.BusMuxOut(BusMuxOut), .Mdatain(RamDataOut), .Q(BusMuxIn_MDR)); //special MDR reg
reg32bit MAR (clr, clk, MARin, BusMuxOut, MARout);

// Goes into ALU A input
reg32bit Y (clr, clk, Yin, BusMuxOut, Yout);

//Instruction register. IRdata doesn't go on the bus, but leads to CON
reg32bit IR (clr, clk, IRin, BusMuxOut, IRdata);

//Memory initialization
//RamDataOut used as MDR input since, using BusMuxIn_MDR would have two registers
writing to the same input.
ram myRam (.clk(clk), .read(read), .write(write), .MARout(MARout[8:0]),
.D(BusMuxIn_MDR), .Q(RamDataOut));

//Control Branch logic
CONN_FF myConn_ff (
    .IRdata(IRdata),
    .BusMuxOut(BusMuxOut),
    .CONN_in(CONN_in),
    .CONN_out(branch_flag)
);


//Select and Encode logic for selecting register functions based on opcode
select_and_encode mySAE (
    .irOut(IRdata),
    .Gra(Gra),
    .Grb(Grb),
    .Grc(Grc),
    .Rin(Rin),
    .Rout(Rout),
    .BAout(BAout),
    .R0in(R0in),
    .R1in(R1in),
    .R2in(R2in),
    .R3in(R3in),
    .R4in(R4in),
    .R5in(R5in),
```

```
    .R6in(R6in),
    .R7in(R7in),
    .R8in(R8in),
    .R9in(R9in),
    .R10in(R10in),
    .R11in(R11in),
    .R12in(R12in),
    .R13in(R13in),
    .R14in(R14in),
    .R15in(R15in),
    .R0out(R0out),
    .R1out(R1out),
    .R2out(R2out),
    .R3out(R3out),
    .R4out(R4out),
    .R5out(R5out),
    .R6out(R6out),
    .R7out(R7out),
    .R8out(R8out),
    .R9out(R9out),
    .R10out(R10out),
    .R11out(R11out),
    .R12out(R12out),
    .R13out(R13out),
    .R14out(R14out),
    .R15out(R15out),
    .C_sign_extended(C_sign_extended)
);

//bus
bus myBus (
    //encoder
    .R0out(R0out),
    .R1out(R1out),
    .R2out(R2out),
    .R3out(R3out),
    .R4out(R4out),
    .R5out(R5out),
    .R6out(R6out),
    .R7out(R7out),
    .R8out(R8out),
    .R9out(R9out),
    .R10out(R10out),
    .R11out(R11out),
    .R12out(R12out),
```

```
    .R13out(R13out),
    .R14out(R14out),
    .R15out(R15out),
    .HIout(HIout),
    .LOout(LOout),
    .ZHighOut(ZHighOut),
    .ZLowOut(ZLowOut),
    .PCout(PCout),
    .MDRout(MDRout),
    .InPortOut(InPortOut),
    .Cout(Cout),
    //multiplexer
    .BusMuxIn_R0(BusMuxIn_R0),
    .BusMuxIn_R1(BusMuxIn_R1),
    .BusMuxIn_R2(BusMuxIn_R2),
    .BusMuxIn_R3(BusMuxIn_R3),
    .BusMuxIn_R4(BusMuxIn_R4),
    .BusMuxIn_R5(BusMuxIn_R5),
    .BusMuxIn_R6(BusMuxIn_R6),
    .BusMuxIn_R7(BusMuxIn_R7),
    .BusMuxIn_R8(BusMuxIn_R8),
    .BusMuxIn_R9(BusMuxIn_R9),
    .BusMuxIn_R10(BusMuxIn_R10),
    .BusMuxIn_R11(BusMuxIn_R11),
    .BusMuxIn_R12(BusMuxIn_R12),
    .BusMuxIn_R13(BusMuxIn_R13),
    .BusMuxIn_R14(BusMuxIn_R14),
    .BusMuxIn_R15(BusMuxIn_R15),
    .BusMuxIn_HI(BusMuxIn_HI),
    .BusMuxIn_LO(BusMuxIn_LO),
    .BusMuxIn_Zhigh(BusMuxIn_Zhigh),
    .BusMuxIn_Zlow(BusMuxIn_Zlow),
    .BusMuxIn_PC(BusMuxIn_PC),
    .BusMuxIn_MDR(BusMuxIn_MDR),
    .BusMuxIn_InPort(BusMuxIn_InPort),
    .C_sign_extended(C_sign_extended),
    .BusMuxOut(BusMuxOut)
);

//alu
alu_test myAlu(
    .clk(clk),
    .clr(clr),
    .incPC(incPC),
    .CONN_out(branch_flag),
```

```
    .B(BusMuxOut),
    .A(Yout),
    .opcode(opcode),
    .C(CRegOut)
    );
endmodule
```

Control Unit

```
//Control unit for CPU
//Finite state machine logic for control
`timescale 1ns/1ps
module ctrl_unit(
    //Test Bench inputs/outputs but goes through datapath
    input clk, reset, stop,
    output reg run, clear,
    //Datapath inputs/outputs
    input [31:0] IRdata,
    output reg read, write,
    output reg BAout, Rin, Rout,
    output reg Gra, Grb, Grc,
    output reg CONN_in,
    output reg MARin, MDRin,
    output reg HIin, LOin,
    output reg Yin, Zin,
    output reg PCin, IRin, incPC,
    output reg InPortIn, OutPortIn,
    output reg HIout, LOout, ZLowOut, ZHighOut,
    output reg MDRout, Cout, InPortOut, PCout,
    output reg [4:0] alu_opcode,
    output reg jal_flag
);
    wire [4:0] ir_op;

    assign ir_op = IRdata[31:27];

    //Control unit states
    parameter   reset_state = 8'b00000000,
                //Fetch states
                fetch0 = 8'b00000001, fetch1 = 8'b00000010, fetch2= 8'b00000011,
                //Add states
                add3 = 8'b00000100, add4= 8'b00000101, add5= 8'b00000110,
                //Sub states
                sub3 = 8'b00000111, sub4 = 8'b00001000, sub5 = 8'b00001001,
                //Multiplication states
```

```
                mul3 = 8'b00001010, mul4 = 8'b00001011, mul5 = 8'b00001100, mul6
= 8'b00001101,
                //Division states
                div3 = 8'b00001110, div4 = 8'b00001111,div5 = 8'b00010000, div6 =
8'b00010001,
                //Or states
                or3 = 8'b00010010, or4 = 8'b00010011, or5 = 8'b00010100,
                //Xor states
                xor3 = 8'b10010101, xor4 = 8'b10010110, xor5 = 8'b10011111,
                //And states
                and3 = 8'b00010101, and4 = 8'b00010110, and5 = 8'b00010111,
                //Shift left states
                shl3 = 8'b00011000, shl4 = 8'b00011001, shl5 = 8'b00011010,
                //Shift right states
                shr3 = 8'b00011011, shr4 = 8'b00011100,  shr5 = 8'b00011101,
                //Shift right arithmatic states
                shra3 = 8'b10011011, shra4 = 8'b10011100,  shra5 = 8'b10011101,
                //Rotate left states
                rol3 = 8'b00011110, rol4 = 8'b00011111, rol5 = 8'b00100000,
                //Rotate right states
                ror3 = 8'b00100001, ror4 = 8'b00100010,  ror5 = 8'b00100011,
                //Negate states
                neg3 = 8'b00100100, neg4 = 8'b00100101, neg5 = 8'b00100110,
                //Not states
                not3 = 8'b00100111, not4 = 8'b00101000,  not5 = 8'b00101001,
                //Load states
                ld3 = 8'b00101010, ld4 = 8'b00101011, ld5 = 8'b00101100, ld6 =
8'b00101101, ld7 = 8'b00101110,
                //Load immediate states
                ldi3 = 8'b00101111, ldi4 = 8'b00110000, ldi5 = 8'b00110001,
                //Store States
                st3 = 8'b00110010, st4 = 8'b00110011, st5 = 8'b00110100, st6 =
8'b00110101, st7 = 8'b00110110,
                //Add immediate states
                addi3 = 8'b00110111, addi4 = 8'b00111000, addi5 = 8'b00111001,
                //And immediate states
                andi3 = 8'b00111010, andi4 = 8'b00111011, andi5 = 8'b00111100,
                //Or immediate states
                ori3 = 8'b00111101, ori4 = 8'b00111110, ori5 = 8'b00111111,
                //Branch if States
                br3 = 8'b01000000, br4 = 8'b01000001, br5 = 8'b01000010, br6 =
8'b01000011,
                //Move reg to PC state
                jr3 = 8'b01000100,
                //Move PC to reg state
```

```
            jal3 = 8'b01000101, jal4 = 8'b10000001,
            //Move HI to reg state
            mfhi3 = 8'b01000111,
            //Move LO to reg state
            mflo3 = 8'b01001000,
            //Move Inport to reg state
            in3 = 8'b01001001,
            //Move reg to Outport state
            out3 = 8'b01001010,
            //No operation state (go to fetch0)
            nop3 = 8'b01001011,
            //Halt state
            halt = 8'b01001100;

    //ALU opcodes
    parameter   alu_nop = 5'b00000,
            alu_addop = 5'b00001,
            alu_subop = 5'b00010,
            alu_mulop = 5'b00011,
            alu_divop = 5'b00100,
            alu_shrop = 5'b00101,
            alu_shlop = 5'b00110,
            alu_shraop = 5'b00111,
            alu_rorop = 5'b01000,
            alu_rolop = 5'b01001,
            alu_andop = 5'b01010,
            alu_orop = 5'b01011,
            alu_negop = 5'b01100,
            alu_xorop = 5'b01101,
            alu_norop = 5'b01110,
            alu_notop = 5'b01111;


    //IR opcodes
    parameter   ir_add  = 5'b00011,
            ir_sub  = 5'b00100,
            ir_mul  = 5'b01111,
            ir_div  = 5'b10000,
            ir_shl  = 5'b01001,
            ir_shr  = 5'b00111,
            ir_shra = 5'b01000,
            ir_rol  = 5'b01011,
            ir_ror  = 5'b01010,
            ir_and  = 5'b00101,
            ir_or   = 5'b00110,
```

```
            ir_xor  = 5'b11110, //Xor opcode is 11110
            ir_neg  = 5'b10001,
            ir_not  = 5'b10010,
            ir_ld   = 5'b00000,
            ir_ldi  = 5'b00001,
            ir_st   = 5'b00010,
            ir_addi = 5'b01100,
            ir_andi = 5'b01101,
            ir_ori  = 5'b01110,
            ir_br   = 5'b10011,
            ir_jr   = 5'b10100,
            ir_jal  = 5'b10101,
            ir_mfhi = 5'b11000,
            ir_mflo = 5'b11001,
            ir_in   = 5'b10110,
            ir_out  = 5'b10111,
            ir_nop  = 5'b11010,
            ir_halt = 5'b11011;


reg [7:0] present_state = halt;



always @(posedge clk, posedge reset, posedge stop) begin

    if (reset) begin    //reset the processor
        #5 present_state = reset_state;
        #10 present_state = fetch0;
    end

    if (stop) begin
        present_state = halt;
    end

    case (present_state)
        fetch0 : #40 present_state = fetch1;
        fetch1 : #40 present_state = fetch2;
        fetch2 : begin
                #40
                case (ir_op)
                ir_add  :  present_state = add3;
                ir_sub  :  present_state = sub3;
                ir_mul  :  present_state = mul3;
                ir_div  :  present_state = div3;
```

```
                ir_shl  :  present_state = shl3;
                ir_shr  :  present_state = shr3;
                ir_shra :  present_state = shra3;
                ir_rol  :  present_state = rol3;
                ir_ror  :  present_state = ror3;
                ir_and  :  present_state = and3;
                ir_or   :  present_state = or3;
                ir_xor  :  present_state = xor3;
                ir_neg  :  present_state = neg3;
                ir_not  :  present_state = not3;
                ir_ld   :  present_state = ld3;
                ir_ldi  :  present_state = ldi3;
                ir_st   :  present_state = st3;
                ir_addi :  present_state = addi3;
                ir_andi :  present_state = andi3;
                ir_ori  :  present_state = ori3;
                ir_br   :  present_state = br3;
                ir_jr   :  present_state = jr3;
                ir_jal  :  present_state = jal3;
                ir_mfhi :  present_state = mfhi3;
                ir_mflo :  present_state = mflo3;
                ir_in   :  present_state = in3;
                ir_out  :  present_state = out3;
                ir_nop  :  present_state = fetch0;
                ir_halt :  present_state = halt;
            endcase
        end
        //Add instruction
        add3    : #40 present_state = add4;
        add4    : #40 present_state = add5;
        add5    : #40 present_state = fetch0;

        //Sub instruction
        sub3    : #40 present_state = sub4;
        sub4    : #40 present_state = sub5;
        sub5    : #40 present_state = fetch0;

        //Mul instruction
        mul3    : #40 present_state = mul4;
        mul4    : #40 present_state = mul5;
        mul5    : #40 present_state = mul6;
        mul6    : #40 present_state = fetch0;

        //Div instruction
        div3    : #40 present_state = div4;
```

```
            div4    : #40 present_state = div5;
            div5    : #40 present_state = div6;
            div6    : #40 present_state = fetch0;

            //Or instruction
            or3     : #40 present_state = or4;
            or4     : #40 present_state = or5;
            or5     : #40 present_state = fetch0;

            //Xor instruction
            xor3    : #40 present_state = xor4;
            xor4    : #40 present_state = xor5;
            xor5    : #40 present_state = fetch0;

            //And instruction
            and3    : #40 present_state = and4;
            and4    : #40 present_state = and5;
            and5    : #40 present_state = fetch0;

            //Shift left instrcutions
            shl3    : #40 present_state = shl4;
            shl4    : #40 present_state = shl5;
            shl5    : #40 present_state = fetch0;

            //Shift right instructions
            shr3    : #40 present_state = shr4;
            shr4    : #40 present_state = shr5;
            shr5    : #40 present_state = fetch0;

            //Shift right arithmatic instructions
            shra3   : #40 present_state = shra4;
            shra4   : #40 present_state = shra5;
            shra5   : #40 present_state = fetch0;

            //Rotate left instructions
            rol3    : #40 present_state = rol4;
            rol4    : #40 present_state = rol5;
            rol5    : #40 present_state = fetch0;

            //Rotate right instructions
            ror3    : #40 present_state = ror4;
            ror4    : #40 present_state = ror5;
            ror5    : #40 present_state = fetch0;

            //Negate instructions
```

```
neg3    : #40 present_state = neg4;
neg4    : #40 present_state = neg5;
neg5    : #40 present_state = fetch0;

//Not instructions
not3    : #40 present_state = not4;
not4    : #40 present_state = not5;
not5    : #40 present_state = fetch0;

//Load instructions
ld3     : #40 present_state = ld4;
ld4     : #40 present_state = ld5;
ld5     : #40 present_state = ld6;
ld6     : #40 present_state = ld7;
ld7     : #40 present_state = fetch0;

//Load immediate instructions
ldi3    : #40 present_state = ldi4;
ldi4    : #40 present_state = ldi5;
ldi5    : #40 present_state = fetch0;

//Store instructions
st3     : #40 present_state = st4;
st4     : #40 present_state = st5;
st5     : #40 present_state = st6;
st6     : #40 present_state = st7;
st7     : #40 present_state = fetch0;

//Add immediate instructions
addi3   : #40 present_state = addi4;
addi4   : #40 present_state = addi5;
addi5   : #40 present_state = fetch0;

//And immediate instructions
andi3   : #40 present_state = andi4;
andi4   : #40 present_state = andi5;
andi5   : #40 present_state = fetch0;

//Or immediate instructions
ori3    : #40 present_state = ori4;
ori4    : #40 present_state = ori5;
ori5    : #40 present_state = fetch0;

//Branch instructions
br3     : #40 present_state = br4;
```

```
            br4     : #40 present_state = br5;
            br5     : #40 present_state = br6;
            br6     : #40 present_state = fetch0;

            //Jump register instructions
            jr3     : #40 present_state = fetch0;

            //Jump and link instructions
            jal3    : #40 present_state = jal4;
            jal4    : #40 present_state = fetch0;

            //Move from HI instructions
            mfhi3   : #40 present_state = fetch0;

            //Move from LO instructions
            mflo3   : #40 present_state = fetch0;

            //Input instructions
            in3     : #40 present_state = fetch0;

            //Output instructions
            out3    : #40 present_state = fetch0;

            //If restart, go to halt
            reset_state : begin
                #40 present_state = fetch0;
            end

            //halt state
            halt : begin
                present_state = halt;
            end
        endcase

    end

    always @(present_state) begin
        case (present_state)
            reset_state : begin
                run = 1; clear = 1;
                read = 0; write = 0;
                Gra = 0; Grb = 0; Grc = 0; Rin = 0;
                BAout = 0; Rin = 0; Rout = 0;
                CONN_in = 0;
                MARin = 0; MDRin = 0;
```

```
                HIin = 0; LOin = 0;
                Yin = 0; Zin = 0;
                PCin = 0; IRin = 0; incPC = 0;
                InPortIn = 0; OutPortIn = 0;
                jal_flag = 0;
                HIout = 0; LOout = 0; ZLowOut = 0; ZHighOut = 0;
                MDRout = 0; Cout = 0; InPortOut = 0; PCout = 0;
                #3 clear = 0; run = 0;
            end

            fetch0: begin
                #10 PCout = 1; MARin = 1; Zin = 1; incPC = 1;
                #15 PCout = 0; MARin = 0; Zin = 0; incPC = 0;
            end
            fetch1: begin
                #10 ZLowOut = 1; PCin = 1; read = 1; MDRin = 1;
                #15 ZLowOut = 0; PCin = 0; read = 0; MDRin = 0;
            end
            fetch2: begin
                #10 MDRout = 1; IRin = 1;
                #10 MDRout = 0; IRin = 0;
            end

            //Alu instruction states
            add3, sub3, and3, or3, shl3, shr3, shra3, ror3, rol3, addi3, andi3,
ori3, neg3, not3, xor3: begin
                #10 Grb = 1; Rout = 1; Yin = 1;
                #15 Grb = 0; Rout = 0; Yin = 0;
            end
            mul3, div3: begin
                #10 Gra = 1; Rout = 1; Yin = 1;
                #15 Gra = 0; Rout = 0; Yin = 0;
            end
            add4 : begin
                #10 Zin = 1; Grc = 1; Rout = 1; alu_opcode = alu_addop;
                #15 Zin = 0; Grc = 0; Rout = 0; alu_opcode = alu_nop;
            end
            sub4 : begin
                #10 Zin = 1; Grc = 1; Rout = 1; alu_opcode = alu_subop;
                #15 Zin = 0; Grc = 0; Rout = 0; alu_opcode = alu_nop;
            end
            and4 : begin
                #10 Zin = 1; Grc = 1; Rout = 1; alu_opcode = alu_andop;
                #15 Zin = 0; Grc = 0; Rout = 0; alu_opcode = alu_nop;
            end
```

```
            or4 : begin
                #10 Zin = 1; Grc = 1; Rout = 1; alu_opcode = alu_orop;
                #15 Zin = 0; Grc = 0; Rout = 0; alu_opcode = alu_nop;
            end
            xor4 : begin
                #10 Zin = 1; Grc = 1; Rout = 1; alu_opcode = alu_xorop;
                #15 Zin = 0; Grc = 0; Rout = 0; alu_opcode = alu_nop;
            end
            shl4 : begin
                #10 Zin = 1; Grc = 1; Rout = 1; alu_opcode = alu_shlop;
                #15 Zin = 0; Grc = 0; Rout = 0; alu_opcode = alu_nop;
            end
            shr4 : begin
                #10 Zin = 1; Grc = 1; Rout = 1; alu_opcode = alu_shrop;
                #15 Zin = 0; Grc = 0; Rout = 0; alu_opcode = alu_nop;
            end
            shra4 : begin
                #10 Zin = 1; Grc = 1; Rout = 1; alu_opcode = alu_shraop;
                #15 Zin = 0; Grc = 0; Rout = 0; alu_opcode = alu_nop;
            end
            ror4  :begin
                #10 Zin = 1; Grc = 1; Rout = 1; alu_opcode = alu_rorop;
                #15 Zin = 0; Grc = 0; Rout = 0; alu_opcode = alu_nop;
            end
            rol4 : begin
                #10 Zin = 1; Grc = 1; Rout = 1; alu_opcode = alu_rolop;
                #15 Zin = 0; Grc = 0; Rout = 0; alu_opcode = alu_nop;
            end
            addi4  :begin
                #10 Zin = 1; Cout = 1; alu_opcode = alu_addop;
                #15 Zin = 0; Cout = 0; alu_opcode = alu_nop;
            end
            andi4 : begin
                #10 Zin = 1; Cout = 1; alu_opcode = alu_andop;
                #15 Zin = 0; Cout = 0; alu_opcode = alu_nop;
            end
            ori4 : begin
                #10 Zin = 1; Cout = 1; alu_opcode = alu_orop;
                #15 Zin = 0; Cout = 0; alu_opcode = alu_nop;
            end
            mul4 : begin
                #10 Zin = 1; Grb = 1; Rout = 1; alu_opcode = alu_mulop;
                #15 Zin = 0; Grb = 0; Rout = 0; alu_opcode = alu_nop;
            end
            div4 : begin
```

```
                #10 Zin = 1; Grb = 1; Rout = 1; alu_opcode = alu_divop;
                #15 Zin = 0; Grb = 0; Rout = 0; alu_opcode = alu_nop;
            end
            neg4 : begin
                #10 Zin = 1; alu_opcode = alu_negop;
                #15 Zin = 0; alu_opcode = alu_nop;
            end
            not4 : begin
                #10 Zin = 1; alu_opcode = alu_notop;
                #15 Zin = 0; alu_opcode = alu_nop;
            end
            add5, sub5, and5, or5, shl5, shr5, shra5, ror5, rol5, andi5, addi5,
ori5, not5, neg5, xor5: begin
                #10 ZLowOut = 1; Gra = 1; Rin = 1;
                #15 ZLowOut = 0; Gra = 0; Rin = 0;
            end
            mul5, div5 : begin
                #10 ZLowOut = 1; LOin = 1; Rin = 1;
                #15 ZLowOut = 0; LOin = 0; Rin = 0;
            end
            mul6, div6 : begin
                #10 ZHighOut = 1; HIin = 1; Rin = 1;
                #15 ZHighOut = 0; HIin = 0; Rin = 0;
            end

            //Load instruction states
            ld3, ldi3: begin
                #10 Grb = 1; BAout = 1; Yin = 1;
                #10 Grb = 0; BAout = 0; Yin = 0;
            end
            ld4, ldi4 : begin
                #10 Zin = 1; Cout = 1; alu_opcode = alu_addop;
                #15 Zin = 0; Cout = 0; alu_opcode = alu_nop;
            end
            ld5 : begin
                #10 ZLowOut = 1; MARin = 1;
                #15 ZLowOut = 0; MARin = 0;
            end
            ldi5 : begin
                #10 ZLowOut = 1; Rin = 1; Gra = 1;
                #15 ZLowOut = 0; Rin = 0; Gra = 0;
            end
            ld6 : begin
                #10 read = 1; MDRin = 1;
                #15 read = 0; MDRin = 0;
```

```
            end
            ld7 : begin
                #10 MDRout = 1; Rin = 1; Gra = 1;
                #15 MDRout = 0; Rin = 0; Gra = 0;
            end

            //Store instruction states
            st3 : begin
                #10 Grb = 1; BAout = 1; Yin = 1;
                #10 Grb = 0; BAout = 0; Yin = 0;
            end
            st4: begin
                #10 Zin = 1; Cout = 1; alu_opcode = alu_addop;
                #15 Zin = 0; Cout = 0; alu_opcode = alu_nop;
            end
            st5 : begin
                #10 ZLowOut = 1; MARin = 1;
                #15 ZLowOut = 0; MARin = 0;
            end
            st6 : begin
                #10 write = 1; MDRin = 1; Rout = 1; Gra = 1;
                #15 write = 0; MDRin = 0; Rout = 0; Gra = 0;
            end

            //Branch instruction states
            br3: begin
                #10 Gra = 1; Rout = 1;
                #15 Gra = 0; Rout = 0;
            end
            br4: begin
                #10 PCout = 1; Yin = 1;
                #15 PCout = 0; Yin = 0;
            end
            br5: begin
                #10 Cout = 1; Zin = 1; CONN_in = 1; alu_opcode = alu_nop;
                #15 Cout = 0; Zin = 0; CONN_in = 0;
            end
            br6: begin
                #10 ZLowOut = 1; PCin = 1;
                #15 ZLowOut = 0; PCin = 0;
            end

            //Jump register and Jump and Link Register instructions
            jr3 : begin
                #10 Gra = 1; Rout = 1; PCin = 1;
```

```
                #10 Gra = 0; Rout = 0; PCin = 0;
            end
            jal3 : begin
                #10 PCout = 1; jal_flag = 1;
                #10 PCout = 0; jal_flag = 0;
            end
            jal4: begin
                #10 PCin = 1; Gra = 1; Rout = 1;
                #10 PCin = 0; Gra = 0; Rout = 0;
            end

            //Input Output instruction states
            in3: begin
                #10 InPortIn = 1; Rin = 1; Gra = 1;
                #15 InPortIn = 0; Rin = 0; Gra = 0;
            end
            out3: begin
                #10 OutPortIn = 1; Rout = 1; Gra = 1;
                #15 OutPortIn = 0; Rout = 0; Gra = 0;
            end

            //Mfhi and Mflo instruction states
            mfhi3: begin
                #10 HIout = 1; Rin = 1; Gra = 1;
                #15 HIout = 0; Rin = 0; Gra = 0;
            end
            mflo3: begin
                #10 LOout = 1; Rin = 1; Gra = 1;
                #15 LOout = 0; Rin = 0; Gra = 0;
            end

            //No operation instruction states
            nop3: begin
                #10 alu_opcode = alu_nop;
            end

            //If halted, show that not running
            halt: begin
                run = 0;
                clear = 0;
            end
    endcase
end

endmodule
```

Select and Encode Logic

```
module select_and_encode (
    input [31:0] irOut,
    input Gra, Grb, Grc,
    input Rin, Rout, BAout,
    output R0in, R1in, R2in, R3in, R4in, R5in, R6in, R7in, R8in, R9in, R10in,
R11in, R12in, R13in, R14in, R15in,
    output R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, R8out, R9out,
R10out, R11out, R12out, R13out,
    R14out, R15out,
    output [31:0] C_sign_extended
);
    wire Rout_or;
    wire [3:0] Ra, Rb, Rc, gra_and, grb_and, grc_and, dec_in;
    wire [15:0] dec_out;

    //Decoder initialization
    decoder4to16 myDecoder (.in(dec_in), .out(dec_out));

    //Partition IR reg output
    assign Ra = irOut[26:23];
    assign Rb = irOut[22:19];
    assign Rc = irOut[18:15];

    //If Gr is high, create 4 bit high, else low for anding later
    assign gra_and = Gra == 1 ? 4'b1111 : 4'b0000;
    assign grb_and = Grb == 1 ? 4'b1111 : 4'b0000;
    assign grc_and = Grc == 1 ? 4'b1111 : 4'b0000;

    //And Ra, Rb, Rc and w Gr input, then or values and input to decoder
    assign dec_in = (gra_and & Ra) | (grb_and & Rb) | (grc_and & Rc);

    //Or Rout and BAout, then covert to 16 bit for and'ing
    assign Rout_or = Rout | BAout;

    //Select wich output signal to send.
    assign R0in = dec_out[0] & Rin;
    assign R1in = dec_out[1] & Rin;
    assign R2in = dec_out[2] & Rin;
    assign R3in = dec_out[3] & Rin;
    assign R4in = dec_out[4] & Rin;
```

```
    assign R5in = dec_out[5] & Rin;
    assign R6in = dec_out[6] & Rin;
    assign R7in = dec_out[7] & Rin;
    assign R8in = dec_out[8] & Rin;
    assign R9in = dec_out[9] & Rin;
    assign R10in = dec_out[10] & Rin;
    assign R11in = dec_out[11] & Rin;
    assign R12in = dec_out[12] & Rin;
    assign R13in = dec_out[13] & Rin;
    assign R14in = dec_out[14] & Rin;
    assign R15in = dec_out[15] & Rin;

    assign R0out = dec_out[0] & Rout_or;
    assign R1out = dec_out[1] & Rout_or;
    assign R2out = dec_out[2] & Rout_or;
    assign R3out = dec_out[3] & Rout_or;
    assign R4out = dec_out[4] & Rout_or;
    assign R5out = dec_out[5] & Rout_or;
    assign R6out = dec_out[6] & Rout_or;
    assign R7out = dec_out[7] & Rout_or;
    assign R8out = dec_out[8] & Rout_or;
    assign R9out = dec_out[9] & Rout_or;
    assign R10out = dec_out[10] & Rout_or;
    assign R11out = dec_out[11] & Rout_or;
    assign R12out = dec_out[12] & Rout_or;
    assign R13out = dec_out[13] & Rout_or;
    assign R14out = dec_out[14] & Rout_or;
    assign R15out = dec_out[15] & Rout_or;

    assign C_sign_extended = irOut[17] == 1 ? {14'b11111111111111, irOut[17:0]} :
{14'b0, irOut[17:0]};
endmodule
```

Bus

```
module bus  #(parameter wordSize = 32)(
    //encoder signals
    input R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, R8out, R9out,
R10out, R11out, R12out, R13out, R14out, R15out,
    input HIout, LOout, ZHighOut, ZLowOut, PCout, MDRout, InPortOut, Cout,
    //multiplexer registers
    input [wordSize-1:0] BusMuxIn_R0, BusMuxIn_R1, BusMuxIn_R2, BusMuxIn_R3,
BusMuxIn_R4, BusMuxIn_R5, BusMuxIn_R6, BusMuxIn_R7, BusMuxIn_R8, BusMuxIn_R9,
```

```
BusMuxIn_R10, BusMuxIn_R11, BusMuxIn_R12, BusMuxIn_R13, BusMuxIn_R14,
BusMuxIn_R15,
    input [wordSize-1:0] BusMuxIn_HI, BusMuxIn_LO, BusMuxIn_Zhigh, BusMuxIn_Zlow,
BusMuxIn_PC, BusMuxIn_MDR, BusMuxIn_InPort, C_sign_extended,
    output [wordSize-1:0] BusMuxOut
);

wire [4:0] s;

encoder32to5 myEncoder (
    .ein({8'b0, Cout, InPortOut, MDRout, PCout, ZHighOut, ZLowOut, HIout, LOout,
        R15out, R14out, R13out, R12out, R11out, R10out, R9out, R8out, R7out,
R6out, R5out, R4out, R3out, R2out, R1out, R0out}),
    .eout(s)
);

mux32to1 BusMux(

    .data0(BusMuxIn_R0),
    .data1(BusMuxIn_R1),
    .data2(BusMuxIn_R2),
    .data3(BusMuxIn_R3),
    .data4(BusMuxIn_R4),
    .data5(BusMuxIn_R5),
    .data6(BusMuxIn_R6),
    .data7(BusMuxIn_R7),
    .data8(BusMuxIn_R8),
    .data9(BusMuxIn_R9),
    .data10(BusMuxIn_R10),
    .data11(BusMuxIn_R11),
    .data12(BusMuxIn_R12),
    .data13(BusMuxIn_R13),
    .data14(BusMuxIn_R14),
    .data15(BusMuxIn_R15),
    .data16(BusMuxIn_LO),
    .data17(BusMuxIn_HI),
    .data18(BusMuxIn_Zlow),
    .data19(BusMuxIn_Zhigh),
    .data20(BusMuxIn_PC),
    .data21(BusMuxIn_MDR),
    .data22(BusMuxIn_InPort),
    .data23(C_sign_extended),
    .data24(0),
    .data25(0),
    .data26(0),
```

```
    .data27(0),
    .data28(0),
    .data29(0),
    .data30(0),
    .data31(0),
    .s(s),
    .out(BusMuxOut)
    );

    endmodule
```

CON FF logic

```
module CONN_FF (
    input [1:0] IRin,
    input [31:0] BusMuxOut,
    input wire CONN_in,
    output reg CONN_out
);
    wire nor_bus, msb_bus;
     reg D;

    //Nor gate all bus inputs
    assign nor_bus = BusMuxOut == 0 ? 1'b1 : 1'b0;

    //Get MSB of bus
    assign msb_bus = BusMuxOut[31];

    //2 to 4 decoder and value a
    always @(*) begin
        case(IRin)
            2'b00: D = nor_bus;
            2'b01: D = ~nor_bus;
            2'b10: D = ~msb_bus;
            2'b11: D = msb_bus;
        endcase

        if(CONN_in)
            CONN_out = D;
        else
            CONN_out = 1'b0;
    end
endmodule
```

RAM

```
//512 x 32 RAM
module ram(
    input clk, read, write,
    input [8:0] MARout,     //address
    input [31:0] D,
    output [31:0] Q
);
    reg [31:0] mem [511:0];

    `ifdef MODEL_TECH
    initial $readmemh("../../ram.hex", mem);
    `else
    initial $readmemh("ram.hex", mem);
    `endif

    assign Q = (write || !read) ?  32'bZZZZZZZZ : mem[MARout];

   always @(posedge clk) begin
        //mem[0] = 32'hB1800000; //opcode for branch case 3 testbench
        if(write) mem[MARout] = D;
    end
endmodule
```

Registers

```
module reg32bit #(parameter qInitial = 0)(clr, clk, enable, D, Q);
    input wire clr, clk, enable;
    input wire [31:0]D;
    output reg [31:0]Q;

    initial Q = qInitial;

    //At positive clock edge begin
    always @(posedge clk) begin
        if (clr) //If clr is high set to 0
            Q <= 0;
        else if (enable) //If enable is high, read value from bus to Q
            Q <= D;
    end
endmodule
```

Special Registers

R0 Register

```
module regR0(BAout, clr, clk, enable, D, Q);
    input wire BAout, clr, clk, enable;
    input wire [31:0] D;
    output wire [31:0] Q;

    //At positive clock edge begin
    reg [31:0] regout;

    always @(posedge clk) begin
        if (clr) //If clr is high set to 0
            regout <= 0;
        else if (enable) //If enable is high, read value from bus to Q
            regout <= D;
    end

    assign Q = BAout ? 32'b0 : regout;

endmodule
```

MDR Register

```
module MD_reg32(
    input clr, clk, read, MDRin,
    input [31:0] BusMuxOut, Mdatain,
    output [31:0] Q
);

    wire [31:0] D;

     mux2to1 MDMux(
        .data_1(BusMuxOut),
        .data_2(Mdatain),
        .select(read),
        .out(D)
    );

    reg32bit MDR(
        .clr(clr),
        .clk(clk),
        .enable(MDRin),
        .D(D),
        .Q(Q)
    );
```

```
endmodule
```

ALU Modules

Synchronous ALU

```
module alu_test #(parameter wordSize = 32)(
    input wire clk, clr, incPC, CONN_out,
    //32-bit input registers A, B,
    input wire [wordSize-1:0] A, B,
    //5-bit opcode
    input wire [4:0] opcode,
    //64-bit output register C
    output reg [(wordSize*2)-1:0] C
);

    parameter   nop = 5'b00000,
                add = 5'b00001, sub = 5'b00010, mul = 5'b00011, div = 5'b00100,
shr = 5'b00101, shl = 5'b00110, shra = 5'b00111,
                ror = 5'b01000, rol = 5'b01001, log_and = 5'b01010, log_or =
5'b01011, log_neg = 5'b01100, log_xor = 5'b01101,
                log_nor = 5'b01110, log_not = 5'b01111;

    wire tempAdd, tempSub;

    wire [wordSize-1:0] temp32_out, add32_out, sub32_out;
    wire [(wordSize*2)-1:0] temp64_out, mul64_out, div64_out;

    alu_32add myAdd (.a(A), .b(B), .cin(1'd0), .s(add32_out), .cout(tempAdd));

    wire [wordSize-1:0] neg_b;
    assign neg_b = ~B + 1;
    alu_32add mySub (.a(A), .b(neg_b), .cin(1'd0), .s(sub32_out),
.cout(tempSub));

    alu_32div myDiv (.dividend(A), .divisor(B), .out(div64_out));
    alu_32mul myMul (.multiplicand(A), .multiplier(B), .product(mul64_out));

    always @(*) begin
        case(opcode)
            add: begin
                C [31:0] <= add32_out[31:0];
                C [63:32] <= 32'd0;
            end
            sub: begin
```

```
            // Calculate two's complement of b
            C [31:0] <= sub32_out[31:0];
            C [63:32] <= 32'd0;
        end
        mul: begin
            C [63:0] <= mul64_out [63:0];
        end
        div: begin
            C [63:0] <= div64_out[63:0];
        end
        shr: begin
            C [31:0] <= A >> B;
            C [63:32] <= 32'd0;
        end
        shl: begin
            C [31:0] <= A << B;
            C [63:32] <= 32'd0;
        end
        shra: begin
            C [31:0] <= $signed(A) >>> $signed(B);
            //C [31:0] <= $signed(A >>> B);
            C [63:32] <= 32'd0;
        end
        ror: begin
            C [31:0] <= (A >> B) | (A << ~B);
            C [63:32] <= 32'd0;
        end
        rol: begin
            C [31:0] <= (A << B) | (A >> ~B);
            C [63:32] <= 32'd0;
        end
        log_and: begin
            C [31:0] <= A & B;
            C [63:32] <= 32'd0;
        end
        log_or: begin
            C [31:0] <= A | B;
            C [63:32] <= 32'd0;
        end
        log_neg: begin
            C [31:0] <= -A;
            C [63:32] <= 32'd0;
        end
        log_xor: begin
            C [31:0] <= A ^ B;
```

```
                C [63:32] <= 32'd0;
            end
            log_nor: begin
                C [31:0] <= ~(A | B);
                C [63:32] <= 32'd0;
            end
            log_not: begin
                C [31:0] <= ~A;
                C [63:32] <= 32'd0;
            end
            default: begin
                C <= A;
            end
        endcase
        if (incPC) begin
            C <= B + 1;
        end
        if (CONN_out) begin
            C <= A + B;
        end
    end
endmodule
```

Asynchronous ALU

```
`timescale 1ns/10ps

module alu (A, B, opcode, clk, clr, incPC, CONN_out, control, C);
    input wire [31:0] A, B, control;
    input wire clk, clr, incPC, CONN_out;
    input wire [4:0] opcode;
    output reg [63:0] C;


    wire[31:0] neg_reg, nor_reg, or_reg, not_reg, xor_reg, and_reg, shl_reg,
shr_reg, shra_reg, ror_reg, rol_reg, add_reg, sub_reg,
    add_cout_reg, sub_cout_reg;
    wire [63:0] mul_reg, div_reg;


    parameter add = 5'b00001, sub = 5'b00010, mul = 5'b00011, div = 5'b00100, shr
= 5'b00101, shl = 5'b00110, shra = 5'b00111,
                    ror = 5'b01000, rol = 5'b01001, log_and = 5'b01010, log_or =
5'b01011, log_neg = 5'b01100, log_xor = 5'b01101,
```

```
                    log_nor = 5'b01110, log_not = 5'b01111;

    cla32bit addop (.a(A), .b(B), .cin(1'd0), .s(add_reg),
.cout(add_cout_reg[0]));
    sub32 subop (.a(A), .b(B), .cin(1'd0), .s(sub_reg), .cout(sub_cout_reg));
    mul32 mulop (.multiplicand(A), .multiplier(B), .product(mul_reg));
    div32 divop (.D(A), .O(B), .Q(div_reg[63:32]), .R(div_reg[31:0]));
    and32 andop (.a(A), .b(B), .c(and_reg));
    or32 orop (.a(A), .b(B), .c(or_reg));
    xor32 xorop (.a(A), .b(B), .c(xor_reg));
    neg32 negop (.in(B), .out(neg_reg));
    nor32 norop (.a(A), .b(B), .c(nor_reg));
    not32 notop (.in(B), .out(not_reg));
    shl32 shlop (B, A, shl_reg);
    shr32 shrop (B, A, shr_reg);
    shra32 shraop (B, A, shra_reg);
    ror32 rorop (B, A, ror_reg);
    rol32 rolop (B, A, rol_reg);


    always @(posedge clk) begin
        //if (opcode) begin -- don't need this since ur using cases
            case(opcode)
                add: begin
                    C [31:0] <= add_reg[31:0];
                    C [63:32] <= 32'd0;
                end
                sub: begin
                    C [31:0] <= sub_reg [31:0];
                    C [63:32] <= 32'd0;
                end
                mul: begin
                    C [63:0] <= mul_reg [63:0];
                end
                div: begin
                    C [63:0] <= div_reg[63:0];
                end
                shr: begin
                    C [31:0] <= shr_reg[31:0];
                    C [63:32] <= 32'd0;
                end
                shl: begin
                    C [31:0] <= shl_reg[31:0];
                    C [63:32] <= 32'd0;
                end
```

```
            shra: begin
                C [31:0] <= shra_reg[31:0];
                C [63:32] <= 32'd0;
            end
            ror: begin
                C [31:0] <= ror_reg[31:0];
                C [63:32] <= 32'd0;
            end
            rol: begin
                C [31:0] <= rol_reg[31:0];
                C [63:32] <= 32'd0;
            end
            log_and: begin
                C [31:0] <= and_reg[31:0];
                C [63:32] <= 32'd0;
            end
            log_or: begin
                C [31:0] <= or_reg[31:0];
                C [63:32] <= 32'd0;
            end
            log_neg: begin
                C [31:0] <= neg_reg[31:0];
                C [63:32] <= 32'd0;
            end
            log_xor: begin
                C [31:0] <= xor_reg[31:0];
                C [63:32] <= 32'd0;
            end
            log_nor: begin
                C [31:0] <= nor_reg[31:0];
                C [63:32] <= 32'd0;
            end
            log_not: begin
                C [31:0] <= not_reg[31:0];
                C [63:32] <= 32'd0;
            end
            default: begin
                C <= A;
            end
        endcase
        if (incPC) begin
            C <= B + 1;
        end
        if (CONN_out) begin
            C <= A + B;
```

```
            end
        end
    //end
endmodule
```

Addition (Carry Lookahead Adder)

```
//Carry look ahead adder

module alu_32add (a,b,cin,s,cout);
    input wire [31:0] a,b;
    input wire cin;
    output wire [31:0] s;
    output wire cout;
    wire c1;

    cla16bit cla1 (.a(a[15:0]), .b(b[15:0]), .s(s[15:0]), .cin(cin), .cout(c1));
    cla16bit cla2 (.a(a[31:16]), .b(b[31:16]), .s(s[31:16]), .cin(c1),
.cout(cout));

endmodule


module cla16bit (a,b,cin,s,cout);
    input wire [15:0] a,b;
    input wire cin;
    output wire [15:0] s;
    output wire cout;
    wire c1, c2, c3;

    cla4bit cla1 (.a(a[3:0]), .b(b[3:0]), .s(s[3:0]), .cin(cin), .cout(c1));
    cla4bit cla2 (.a(a[7:4]), .b(b[7:4]), .s(s[7:4]), .cin(c1), .cout(c2));
    cla4bit cla3 (.a(a[11:8]), .b(b[11:8]), .s(s[11:8]), .cin(c2), .cout(c3));
    cla4bit cla4 (.a(a[15:12]), .b(b[15:12]), .s(s[15:12]), .cin(c3),
.cout(cout));
endmodule


module cla4bit (a,b,cin,s,cout);
    input wire [3:0] a,b;
    input wire cin;
    output wire [3:0] s;
    output wire cout;
    wire c1, c2, c3;

    cla cla1 (.a(a[0]), .b(b[0]), .s(s[0]), .cin(cin), .cout(c1));
```

```
    cla cla2 (.a(a[1]), .b(b[1]), .s(s[1]), .cin(c1), .cout(c2));
    cla cla3 (.a(a[2]), .b(b[2]), .s(s[2]), .cin(c2), .cout(c3));
    cla cla4 (.a(a[3]), .b(b[3]), .s(s[3]), .cin(c3), .cout(cout));
endmodule


module cla (a,b,s,cin,cout);
    input wire a,b,cin;
    output wire s,cout;

    wire xor1, and1, and2;

    assign xor1 = ((a)^(b));
    assign and1 = ((a)&(b));
    assign and2 = ((xor1)&(cin));
    assign s = ((xor1)^(cin));
    assign cout = ((and1)|(and2));

endmodule
```

Subtraction

```
module sub32 (a, b, cin, s, cout);
    input wire signed [31:0] a, b;
    input wire cin;
    output wire signed [31:0] s;
    output wire cout;
    wire [31:0] temp;

    neg32 negate(.in(b), .out(temp));
    cla32bit add(.a(a), .b(temp), .cin(cin), .s(s), .cout(cout));

endmodule
```

Multiplication (Booths Algorithm With Bitpair Encoding)

```
//Booths algorithm multiplication.
module alu_32mul (
    input signed [31:0] multiplicand,
    input signed [31:0] multiplier,
    output reg signed [63:0] product
);

reg signed [31:0] multiplicand_reg;
reg signed [32:0] multiplier_reg;
reg signed [2:0] bitPattern;
```

```
reg signed [63:0] temp;

integer i;

always @(*) begin
    // Initialization
    multiplicand_reg = multiplicand;
    temp = 64'b0;
    multiplier_reg = (multiplier << 1);

    for (i = 0; i < 16; i = i + 1) begin

        bitPattern = multiplier_reg[2:0];

        case (bitPattern)
            // +1 x M
            3'b001, 3'b010: temp = temp + multiplicand_reg;
            // +2 x M
            3'b011: temp = temp + (multiplicand_reg + multiplicand_reg);
            // -2 x M
            3'b100: temp = temp - (multiplicand_reg + multiplicand_reg);
            // -1 x M
            3'b101, 3'b110: temp = temp - multiplicand_reg;
                default: temp = 64'bx;
        endcase

        multiplicand_reg = multiplicand_reg <<< 2;
        multiplier_reg = multiplier_reg >>> 2;
    end

    product = temp;
end

endmodule
```

Division (Nonrestoring Algorithm)

```
//Nonrestoring Algorithm for division

module alu_32div(
    input signed [31:0] dividend,
    input signed [31:0] divisor,
    output reg signed [63:0] out
);
    reg signed [63:0] temp;
    reg signed [31:0] D, O;
    integer i;

    always @(*) begin
        if (dividend[31])
            D = -dividend;
        else
            D = dividend;

        if (divisor[31])
            O = -divisor;
        else
            O = divisor;

        // Initialize temp with D and 32 zeros (A/Q reg)
        temp = {32'h0, D};
        // Perform division for 32 iterations
        for (i = 0; i < 32; i = i + 1) begin
            // Shift temp left by one bit
            temp = temp << 1;

            // If the most significant bit of temp is one,
            // add O to upper half of temp.
            if (temp[63] == 1'b1)
                temp[63:32] = temp[63:32]+ O;

            // Otherwise, subtract O from upper half of temp.
            else
                temp[63:32] = temp[63:32] - O;
            // If the most significant bit of temp is one,
            // set the least significant bit of quotient to zero.
            if (temp[63] == 1'b1)
                temp[0] = 1'b0;
            // Otherwise, set it to one.
```

```
            else
                temp[0] = 1'b1;
        end


        if ((dividend[31])||(divisor[31])) begin
            if (!((dividend[31])&&(divisor[31])))
                temp[31:0] = -temp[31:0];
        end
        // If the most significant bit of temp is one,
        // add 0 to remainder and set out accordingly.

        if (temp[63] == 1'b1) begin
            out = {temp[63:32] + 0, temp[31:0]};

        // Otherwise, set out accordingly without adding 0 to remainder.
        end else begin
            out = {temp[63:32],temp[31:0]};
        end
    end

endmodule
```

And Module

```
module and32 (a, b, c);
    input wire [31:0] a, b;
    output reg [31:0] c;
    reg [31:0] temp;
    integer i;

    always @(*) begin
        for (i = 0; i < 31; i = i + 1) begin
            temp[i] = ((a[i])&(b[i]));
        end
        c = temp;
    end

endmodule
```

OR Module

```
 module or32 (a,b,c);
    input wire [31:0]a,b;
    output wire [31:0]c;
    genvar i;
```

```
    generate
        for (i = 0; i < 31; i = i + 1) begin : loop
            assign c[i] = ((a[i])|(b[i]));
        end
    endgenerate

endmodule
```

XOR Module

```
module xor32 (a,b,c);
    input wire [31:0]a,b;
    output reg [31:0]c;
    integer i;

    always @(*)begin
        for (i = 0; i < 31; i = i + 1) begin
            c[i] = ((a[i])^(b[i]));
        end
    end

endmodule
```

Rotate Right

```
module ror32 (in, num_shifts, out);
    input wire signed [31:0] in;
    input wire signed [31:0] num_shifts;
    output wire signed [31:0] out;

    assign out = (in >> num_shifts) | (in << ~num_shifts);

endmodule
```

Rotate Left

```
module rol32 (in, num_shifts, out);
    input wire signed[31:0] in;
    input wire signed[31:0] num_shifts;
    output wire signed[31:0] out;

    assign out = (in << num_shifts) | (in >> ~num_shifts);
endmodule
```

Shift Left

```
module shl32 (in, num_shifts, out);
    input wire signed [31:0] in;
    input wire [31:0] num_shifts;
    output reg signed[31:0] out;

    always @(*) begin
        out[31:0] = in<<num_shifts;
    end
endmodule
```

Shift Right

```
module shr32 (in, num_shifts, out);
    input wire signed[31:0] in;
    input wire [31:0] num_shifts;
    output reg signed[31:0] out;

    always @(*) begin
        out[31:0] = in>>num_shifts;
    end
endmodule
```

Shift Right Arithmatic

```
module shra32 (in, num_shifts, out);
    input wire signed [31:0] in;
    input wire [31:0] num_shifts;
    output reg signed [31:0] out;

    always @(*) begin
        out[31:0] = in>>>num_shifts;
    end
endmodule
```

Not

```
module not32 (in, out);
    input wire signed[31:0] in;
    output wire signed[31:0] out;

    genvar i;
    generate
        for (i=0; i<32; i = i+1) begin : loop
            assign out[i] = !in[i];
        end
```

```
    endgenerate
endmodule
```

Negate

```
module neg32 (in, out);
    input wire signed[31:0] in;
    output wire signed[31:0] out;
    wire [31:0] temp;
    wire cout;

    not32 notop (.in(in),.out(temp));
    cla32bit add_op (.a(temp), .b(32'd1), .cin(1'd0), .s(out), .cout(cout));

endmodule
```

Nor Module

```
module nor32 (a,b,c);
    input wire [31:0]a,b;
    output reg [31:0]c;
    integer i;
    always @(*)begin
        for (i = 0; i < 31; i = i + 1) begin
            c[i] = ((a[i])~^(b[i]));
        end
    end
endmodule
```

(Never used)

Miscilanious Modules

4 To 16 Decoder

```
module decoder4to16 (
    input [3:0] in,
    output reg [15:0] out
);

   always@(*) begin
        out = 16'd0;
        case(in)
            4'b0000: out[0] = 1;
            4'b0001: out[1] = 1;
            4'b0010: out[2] = 1;
```

```
            4'b0011: out[3] = 1;
            4'b0100: out[4] = 1;
            4'b0101: out[5] = 1;
            4'b0110: out[6] = 1;
            4'b0111: out[7] = 1;
            4'b1000: out[8] = 1;
            4'b1001: out[9] = 1;
            4'b1010: out[10] = 1;
            4'b1011: out[11] = 1;
            4'b1100: out[12] = 1;
            4'b1101: out[13] = 1;
            4'b1110: out[14] = 1;
            4'b1111: out[15] = 1;
        endcase
    end
endmodule
```

32 to 5 Encoder

```
module encoder32to5 (input wire [31:0]ein, output reg [4:0] eout);
    always @(ein) begin
        if(ein[31]==1) eout=5'b11111;
        else if (ein[30]==1) eout=5'b11110;
        else if (ein[29]==1) eout=5'b11101;
        else if (ein[28]==1) eout=5'b11100;
        else if (ein[27]==1) eout=5'b11011;
        else if (ein[26]==1) eout=5'b11010;
        else if (ein[25]==1) eout=5'b11001;
        else if (ein[24]==1) eout=5'b11000;
        else if (ein[23]==1) eout=5'b10111;
        else if (ein[22]==1) eout=5'b10110;
        else if (ein[21]==1) eout=5'b10101;
        else if (ein[20]==1) eout=5'b10100;
        else if (ein[19]==1) eout=5'b10011;
        else if (ein[18]==1) eout=5'b10010;
        else if (ein[17]==1) eout=5'b10001;
        else if (ein[16]==1) eout=5'b10000;
        else if (ein[15]==1) eout=5'b01111;
        else if (ein[14]==1) eout=5'b01110;
        else if (ein[13]==1) eout=5'b01101;
        else if (ein[12]==1) eout=5'b01100;
        else if (ein[11]==1) eout=5'b01011;
        else if (ein[10]==1) eout=5'b01010;
        else if (ein[9]==1) eout=5'b01001;
        else if (ein[8]==1) eout=5'b01000;
```

```
        else if (ein[7]==1) eout=5'b00111;
        else if (ein[6]==1) eout=5'b00110;
        else if (ein[5]==1) eout=5'b00101;
        else if (ein[4]==1) eout=5'b00100;
        else if (ein[3]==1) eout=5'b00011;
        else if (ein[2]==1) eout=5'b00010;
        else if (ein[1]==1) eout=5'b00001;
        else eout=5'b00000;
    end
endmodule
```

2 to 1 MUX

```
module mux2to1(data_1, data_2, out, select);
    input wire [31:0] data_1, data_2;
    input wire select;
    output reg [31:0]out;

    always@(*)begin
        if (select)
            out <= data_2;
        else
            out <= data_1;
    end
endmodule
```

32 To 1 MUX

```
module mux32to1(data0, data1, data2, data3, data4, data5, data6, data7, data8,
data9, data10, data11, data12, data13, data14, data15, data16, data17, data18,
data19, data20, data21, data22, data23, data24, data25, data26, data27, data28,
data29, data30, data31, out, s);
    input wire [31:0] data0, data1, data2, data3, data4, data5, data6, data7,
data8, data9, data10, data11, data12, data13, data14, data15, data16, data17,
data18, data19, data20, data21, data22, data23, data24, data25, data26, data27,
data28, data29, data30, data31;
    input wire [4:0] s;
    output reg [31:0] out;

    always @(*) begin
        if (s==5'd31) out <= data31;
        else if (s==5'd30) out <= data30;
        else if (s==5'd29) out <= data29;
```

```
        else if (s==5'd28) out <= data28;
        else if (s==5'd27) out <= data27;
        else if (s==5'd26) out <= data26;
        else if (s==5'd25) out <= data25;
        else if (s==5'd24) out <= data24;
        else if (s==5'd23) out <= data23;
        else if (s==5'd22) out <= data22;
        else if (s==5'd21) out <= data21;
        else if (s==5'd20) out <= data20;
        else if (s==5'd19) out <= data19;
        else if (s==5'd18) out <= data18;
        else if (s==5'd17) out <= data17;
        else if (s==5'd16) out <= data16;
        else if (s==5'd15) out <= data15;
        else if (s==5'd14) out <= data14;
        else if (s==5'd13) out <= data13;
        else if (s==5'd12) out <= data12;
        else if (s==5'd11) out <= data11;
        else if (s==5'd10) out <= data10;
        else if (s==5'd9) out <= data9;
        else if (s==5'd8) out <= data8;
        else if (s==5'd7) out <= data7;
        else if (s==5'd6) out <= data6;
        else if (s==5'd5) out <= data5;
        else if (s==5'd4) out <= data4;
        else if (s==5'd3) out <= data3;
        else if (s==5'd2) out <= data2;
        else if (s==5'd1) out <= data1;
        else if (s==5'd0) out <= data0;
    end
endmodule
```

Contents of the Memory Before and After Program Execution

Before

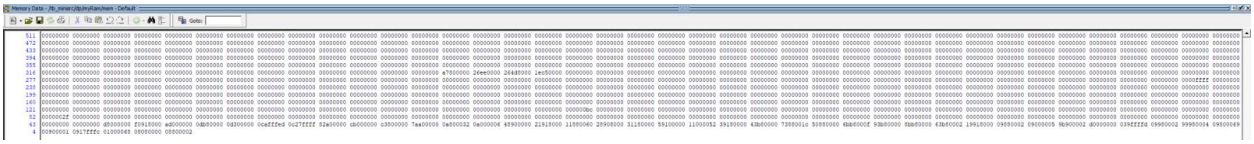
The instruction that were pre-loaded into the memory are as follows:

0: ldi R1, 2
1: ldi R0, 0(R1)
2: ld R2, 104
3: ldi R2, -4(R2)
4: ld R1, 1(R2)
5: ldi R3, 105
6: brmi R3, 4
7: ldi R3, 2(R3)
8: ld R7, -3(R3)

9: nop
10: brpl R7, 2
11: ldi R2, 5
12: ldi R3, 2(R1)
13: add R3, R2, R3
14: addi R7, R7, 2
15: neg R7, R7
16: not R7, R7
17: andi R7, R7, 15
18: ror R1, R1, R0
19: ori R7, R1, 28
20: shra R7, R7, R0
21: shr R2, R3, R0
22: st 82, R2
23: rol R2, R2, R0
24: or R2, R3, R0
25: and R1, R2, R1
26: st 96(R1), R3
27: sub R3, R2, R3
28: shl R1, R2, R0
29: ldi R4, 6
30: ldi R5, 50
31: mul R5, R4
32: mfhi R7
33: mflo R6
34: div R5, R4
35: ldi R8, -1(R4)
36: ldi R9, -19(R5)
37: ldi R10, 0(R6)
38: ldi R11, 0(R7)
39: jal R10
40: xor R1, R2, R3
41: halt
82: 00000026 (immediate value in hex)
104: 00000055 (immediate value in hex)
240: 0000FFFF (immediate value in hex)
300: add R13, R8, R10
301: sub R12, R9, R11
302: sub R13, R13, R12
303: jr R15

Note that the value on the left hand side is the decimal representation of the address in memory where the value or instruction was loaded into. Some of the values were immediate values that were to be loaded into register. These were located at 82, 104, and 240 in memory.

After



@0 08800002 08080000 01000068 0917fffc 00900001 09800069 99980004 09980002

@8 039ffffd d0000000 9b900002 09000005 09880002 19918000 63b80002 8bb80000

@10 93b80000 6bb8000f 50880000 7388001c 43b80000 39180000 11000052 59100000

@18 31180000 28908000 11880060 21918000 48900000 0a000006 0a800032 7aa00000

@20 c3800000 cb000000 82a00000 0c27ffff 0cafffed 0d300000 0db80000 ad000000

@28 **f0918000** d8000000 00000000 00000000 00000000 00000000 00000000 00000000

@30 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@38 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@40 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@48 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@50 00000000 00000000 0000002f 00000000 00000000 00000000 00000000 00000000

@58 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@60 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@68 000000bc 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@70 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@78 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@80 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@88 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@90 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@98 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@a0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@a8 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@b0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@b8 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@c0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@c8 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@d0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@d8 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@e0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@e8 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@f0 0000ffff 00000000 00000000 00000000 00000000 00000000 00000000

@f8 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@100 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@108 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@110 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@118 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@120 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@128 00000000 00000000 00000000 00000000 1ec50000 264d8000 26ee0000 a7800000

@130 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@138 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@140 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@148 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@150 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@158 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@160 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@168 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@170 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@178 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@180 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@188 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@190 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@198 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@1a0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@1a8 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@1b0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@1b8 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@1c0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@1c8 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@1d0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@1d8 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@1e0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@1e8 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@1f0 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

@1f8 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

Instruction Opcode List

IR format

Name	Fields					Comments
Field size	31..27 5 bits	26..23 4 bits	22..19 4 bits	18..15 4 bits	14..0 15 bits	All instructions are 32-bit long
R-Format	OP	Ra	Rb	Rc	Unused	Arithmetic/Logical
I-Format	OP	Ra	Rb	Constant C / Unused		Arithmetic/Logical; Load/Store; Imm.
B-Format	OP	Ra	C2	Constant C		Branch
J-Format	OP	Ra	Unused			Jump; Input/Output; Special
M-Format	OP	Unused				Misc.

IR Table

Instruction	Format	OP Code (Upper 5 bits)	Instruction Type
Addition	ADD Ra, Rb, Rc	00011	R-Type
Subtraction	SUB Ra, Rb, Rc	00100	R-Type
Shift Left	SHL Ra, Rb, Rc	01001	R-Type
Shift Right	SHR Ra, Rb, Rc	00111	R-Type
Shift Right Arithmetic	SHRA Ra, Rb, Rc	01000	R-Type
Rotate Left	ROL Ra, Rb, Rc	01011	R-Type
Rotate Right	ROR Ra, Rb, Rc	01010	R-Type
And	AND Ra, Rb, Rc	00101	R-Type
Or	OR Ra, Rb, Rc	00110	R-Type

Xor	XOR Ra, Rb, Rc	11110	R-Type
Multiplication	MUL Ra, Rb	01111	I-Type
Division	DIV Ra, Rb	10000	I-Type
Add Immediate	ADDI Ra, Rb, C	01100	I-Type
And Immediate	ANDI Ra, Rb, C	01101	I-Type
Or Immediate	ORI Ra, Rb, C	01110	I-Type
Negate	NEG Ra, Rb	10001	I-Type
Not	NOT Ra, Rb	10010	I-Type
Load	LD Ra, C(Rb)	00000	I-Type
Load Immediate	LDI Ra, C(Rb)	00001	I-Type
Store	ST C(Rb), Ra	00010	I-Type
Branch If Zero	BRZR Ra, C	10011	B-Type
Branch If Not Zero	BRNZ Ra, C	10011	B-Type
Branch If Positive	BRPL Ra, C	10011	B-Type
Branch If Negative	BRMI Ra, C	10011	B-Type
Return from Procedure	JR Ra	10100	J-Type
Jump and Link	JAL Ra	10101	J-Type
Input	IN Ra	10110	J-Type
Output	OUT Ra	10111	J-Type
Move from HI	MFHI Ra	11000	J-Type
Move from LO	MFLO Ra	11001	J-Type
Nop	NOP	11010	M-Type
Halt	HALT	11011	M-Type