A Silly Scope

16-bit Processor Data Sheet

Matt Mammarelli
Stephen Glass
Joseph Foderaro

May 9, 2017

Features

- **Efficient, 16-bit microprocessor**
- **Lightweight RISC Architecture**
    - **32 Implemented Instructions**
    - **16 x 16 Register File**
    - **16-bit ALU**
- **Large Program and Data Memories**
    - **Memory:** Two 256 x 16 M9K blocks
- **Architecture**: Harvard
    - Individual instruction and program memory blocks for maximum performance
- **Max clock speed:** 50 MHz
- **Intermediate Instructions supported:** BCLR, BSET
- **Advanced Instructions:** MUL

Table of Contents

List of Figures

List of Tables

Architecture Overview

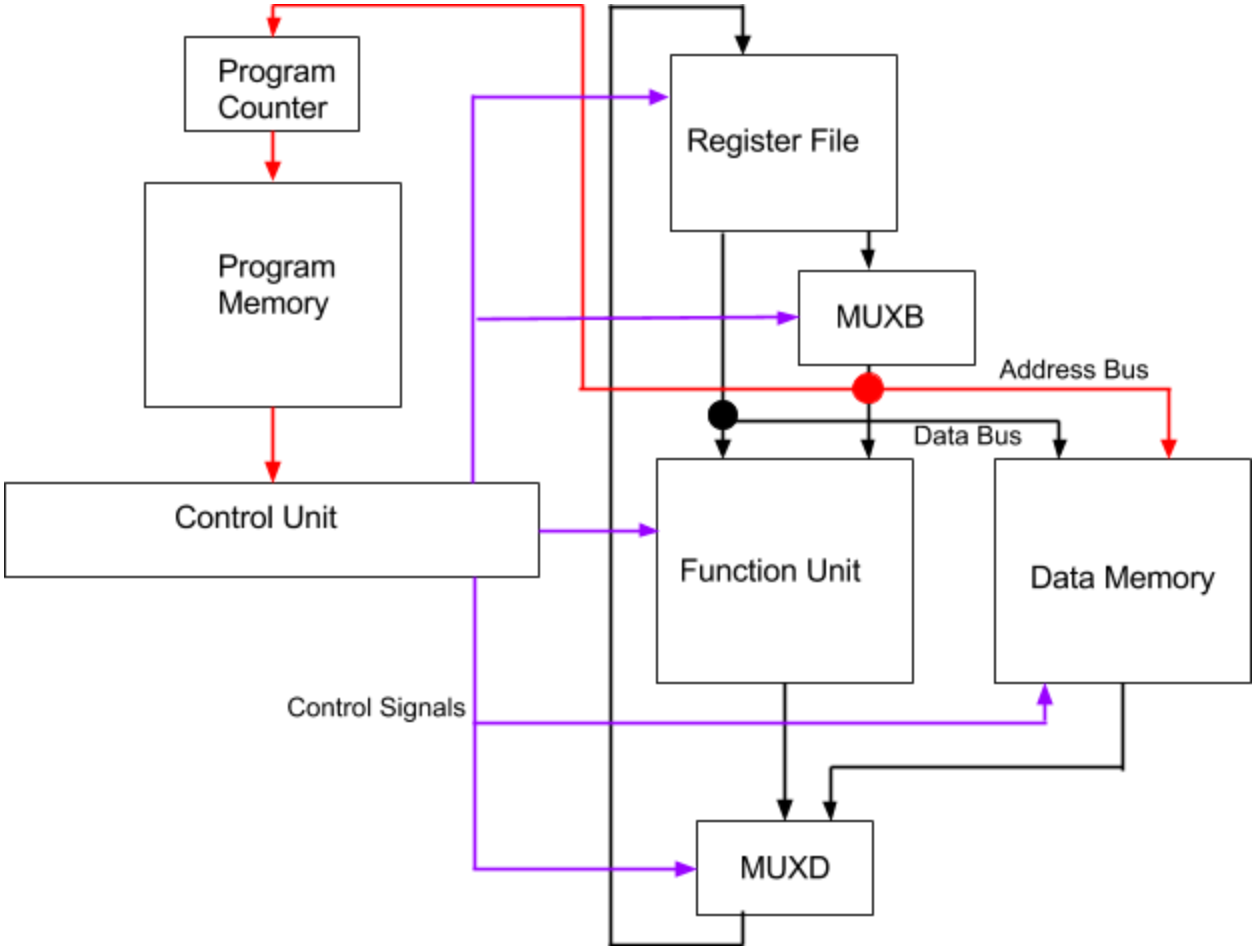


Figure 1: Overview of a Harvard Architecture Processor

This processor is a Harvard architecture which means that it contains separate program and data memory blocks. In the figure above, the black lines represent the Data Bus, the red lines represent the Address Bus, and the purple lines represent the Control Signals coming from the Control Unit. The Register File is a 16x16 unit which means that it contains 16 registers which are all 16 bit in width. The MUXB is a multiplexor which selects between a constant value and the data from the Register File. The Function Unit is also known as the Arithmetic Logic Unit (ALU) and contains all of the arithmetic, logical, and shifting functions that this processor can perform. The Data Memory has the ability to store data at a designated address for use at a later time. The MUXD is another multiplexor which selects between the Function Unit and the Data Memory units. The Program Counter is a unit which will increment an 8 bit address which will be assigned to each instruction inputted to the processor. The Program Memory will contain all of the instructions that the processor will execute. The Program Counter will increment a 16

bit address after every clock cycle which is fed into the Program Memory for addressing. Finally, the Control Unit will output a Control Word which contains all of the control signals that attach to other components in the processor. The Control Unit will be attached to units such as the multiplexor selects and the Register File address inputs.

Register File Design

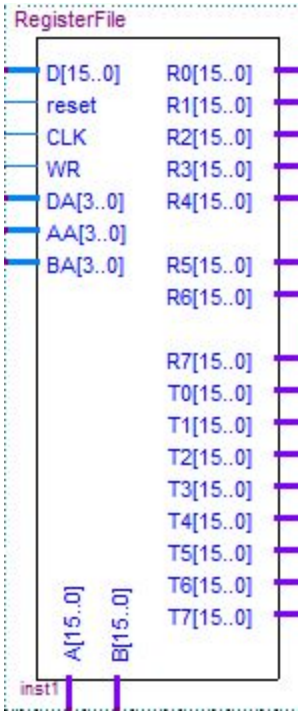


Figure 2: Register File Block

This 16x16 Register File contains the inputs of a 16 bit data (D), a reset, clock (CLK), a write enable (WR), a 4 bit Destination Address (DA), a 4 bit A Address (AA), and a 4 bit B address (BA). The Register File contains 16 bit outputs for each register from R0 to T7 which are individually selected by the DA. The data outputted from the 16 bit A and B outputs are selected by AA and BA.

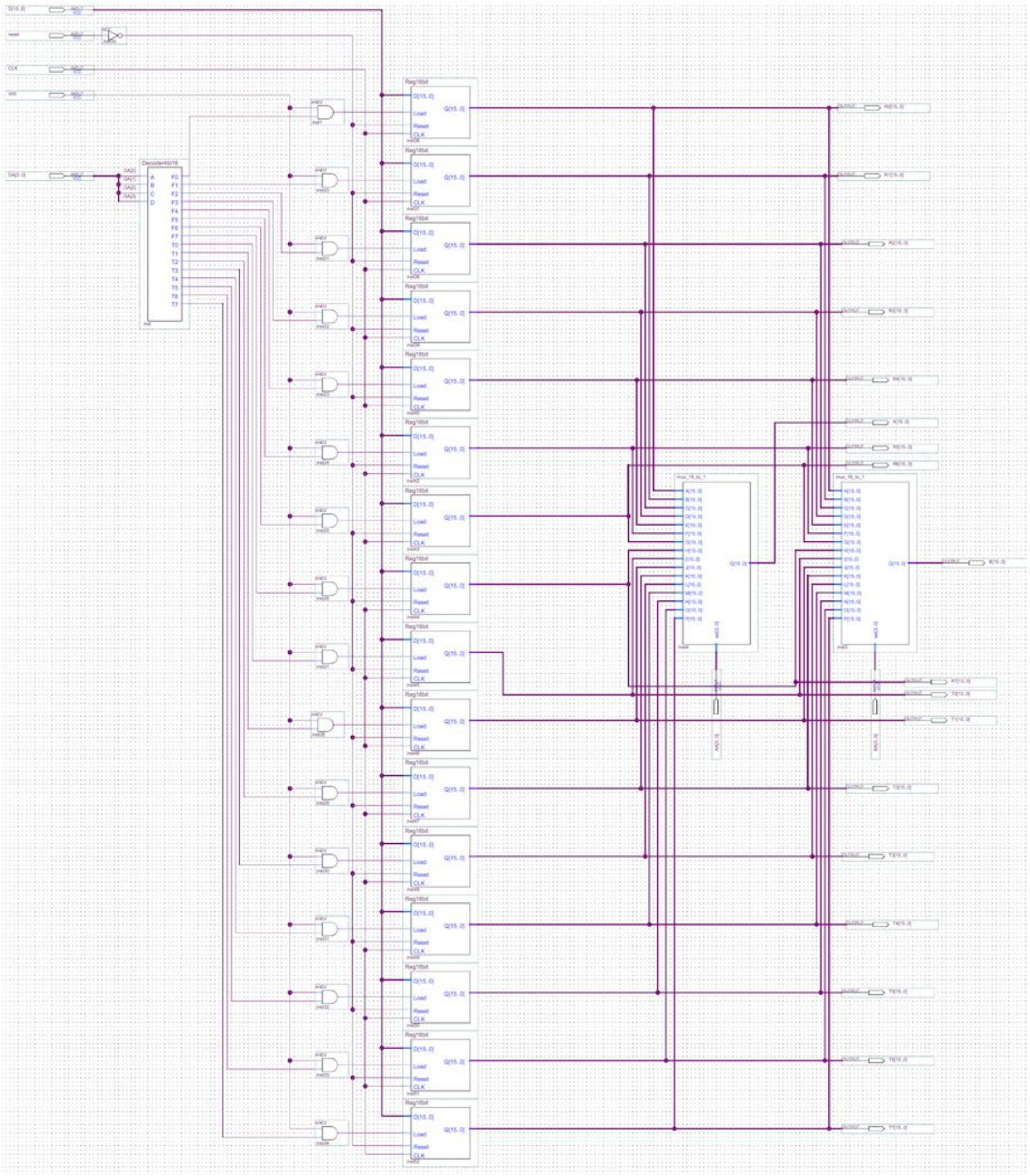


Figure 3: Register File Schematic

In figure 3, there are several components to take note of. Starting from the left, there is a the DA which feeds into a 4 to 16 decoder. A 4 to 16 decoder takes a 4 bit address and individually sets one of the bits of its 16 bit output high depending on the input address. Each output bit of this decoder is fed to an AND gate with the WR input and outputs to the load of each register cell. This design ensures that the load of the register cell will only be high when both the write enable and the designated decoder bit is high. There are 16 bit register cells, each of which internally are consisted of a cascade of 16 D-Latch Flip Flops which use the reset, clock, and Data (D) inputs. The outputs of each of these Register Cells become the outputs for each 16 bit register

R0 through T7. The outputs also connect to two 16 to 1 multiplexers used to produce the 16 bit A and B outputs.

A verilog testbench was created and ran in ModelSim-Altera to verify the functionality of the Register File. The testbench iterates through every register and loads it with the value provided in D. At the end of the simulation, all registers contain the value D which indicates that the register file is working correctly.

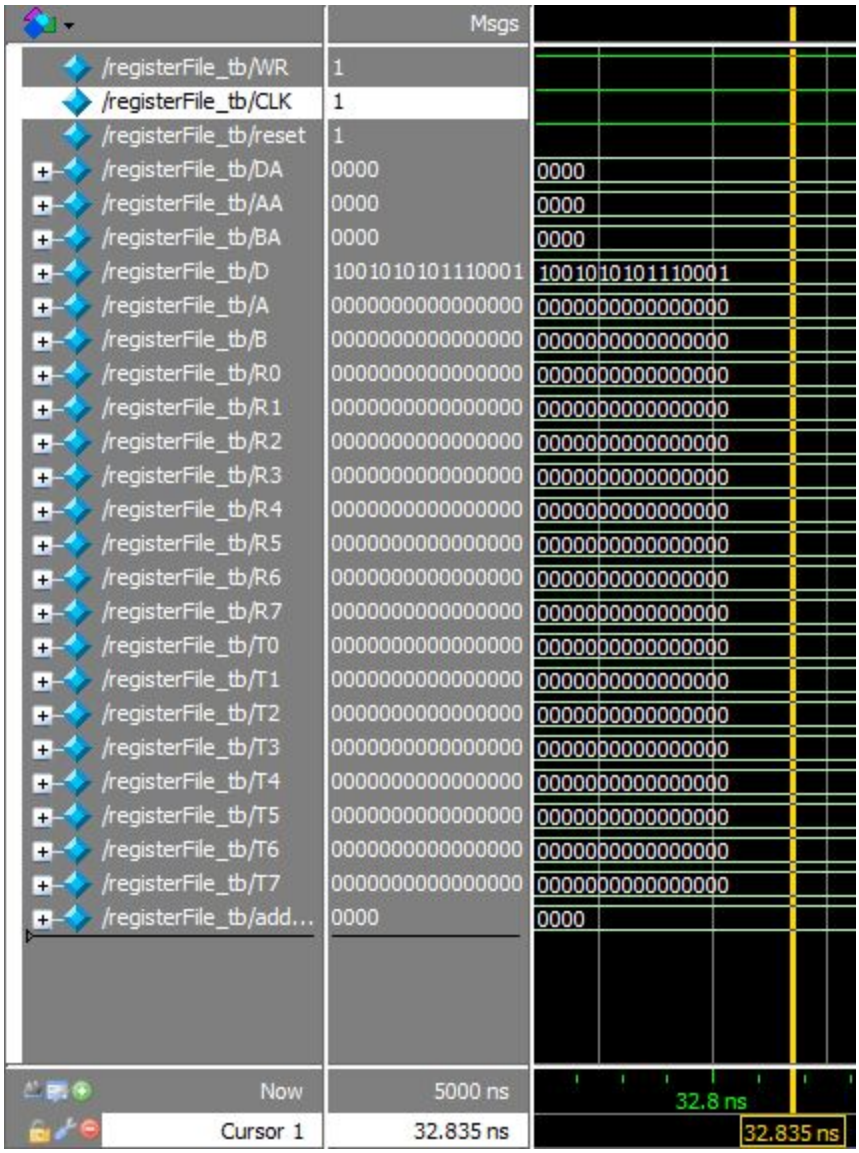


Figure 4: Beginning of Simulation showing all registers are empty.

See Appendix registerFile_tb.
Above D=16'b1001010101110001, WR =1'b1, and reset=1'b1;

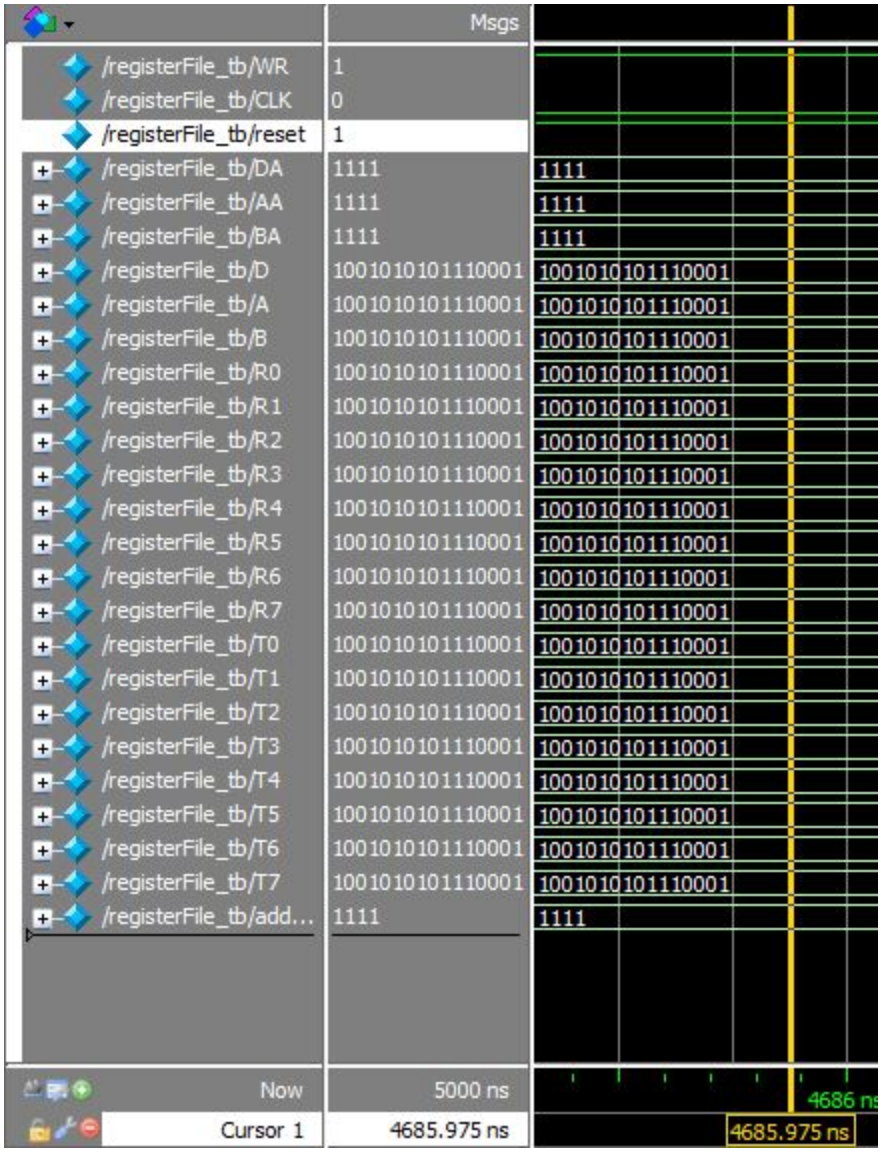


Figure 5: End of Simulation showing all registers contain value of D

See Appendix registerFile_tb..
Above D=16'b1001010101110001, WR =1'b1, and reset=1'b1;

ALU Design

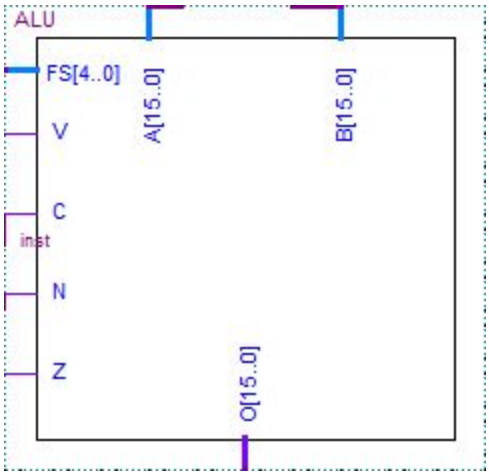


Figure 6: ALU block diagram

Figure 6 shows a 16 bit Ripple Carry Adder ALU. This Function Unit ALU has the inputs of a 5 bit Function Select, a 16 bit A data line, and a 16 bit B data line. Outputs consist of a 16 bit output, a zero bit flag (Z), a negative bit flag (N), a carry bit flag (C), and an overflow bit flag (V). Essentially, once the A and B inputs are set, a function will be performed according to the bits passed to FS, and there will be an output to O. The flags will be set depending on the output of the function. The zero bit will be set high if the output is 16 bits of 0. The negative flag will be set high if the most significant bit of the output is 1. The carry flag will be set high if there is a carry in the arithmetic from one bit of the ALU Cell to the next. The overflow bit will be set high if both the N and C flags are set high.

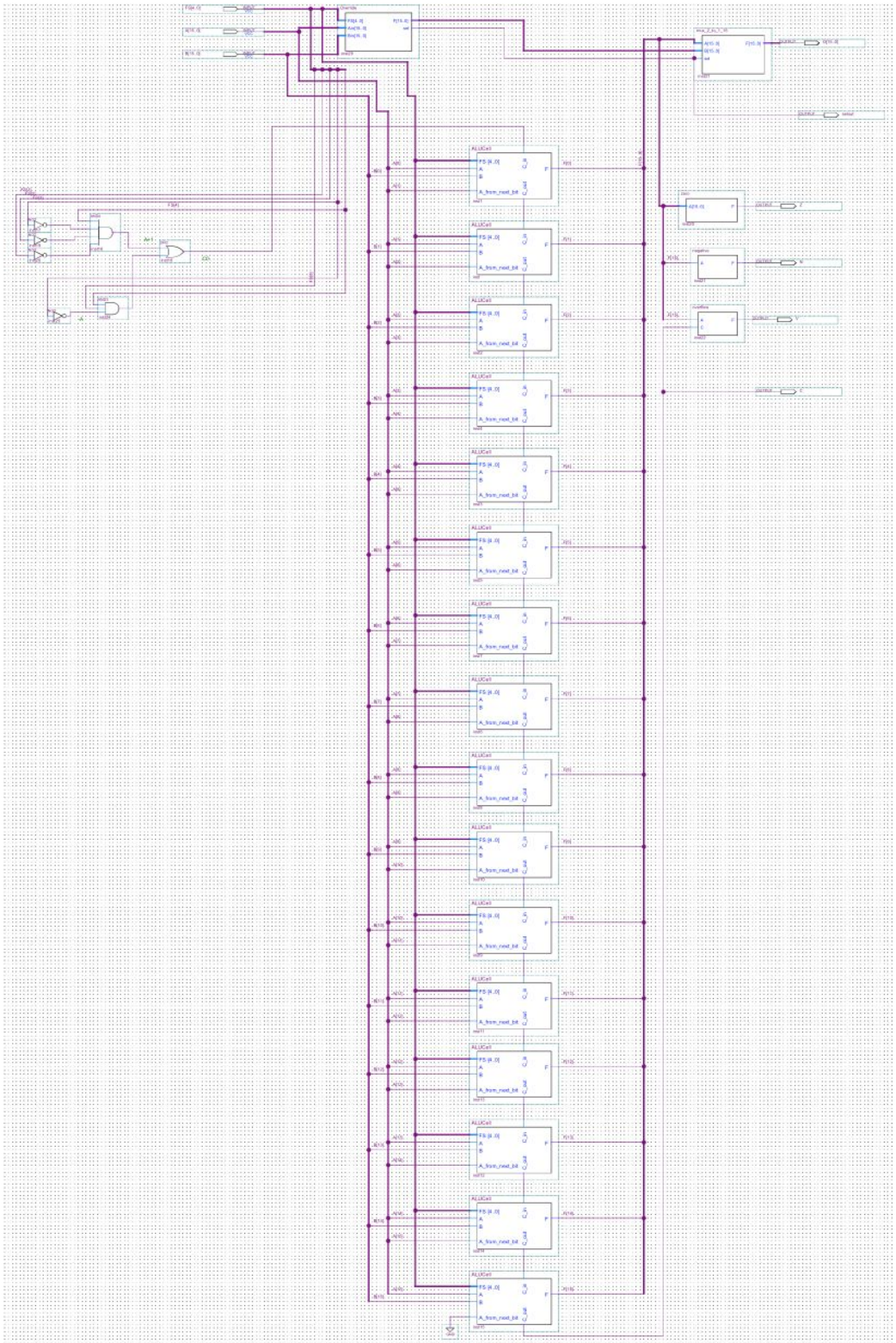


Figure 7: ALU Schematic

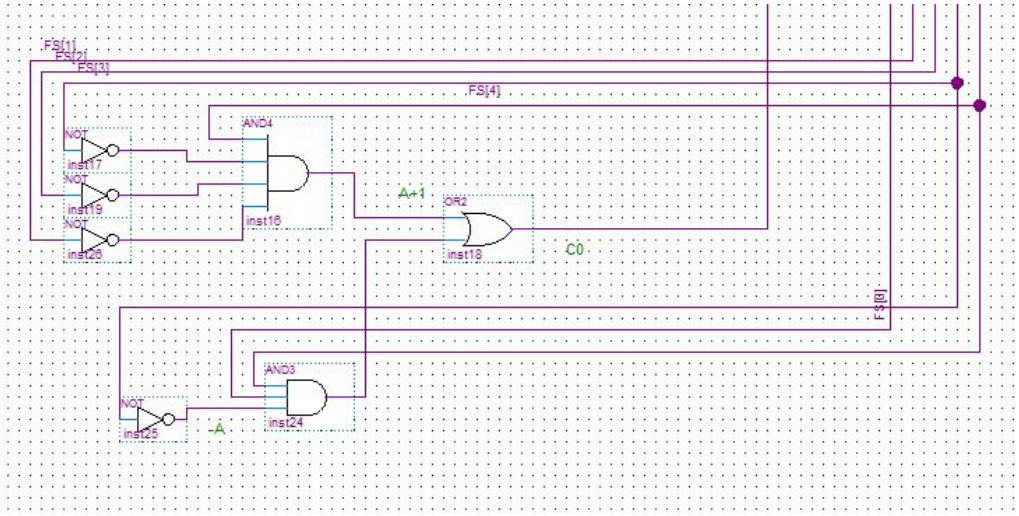


Figure 8: Carry In Logic for first ALU

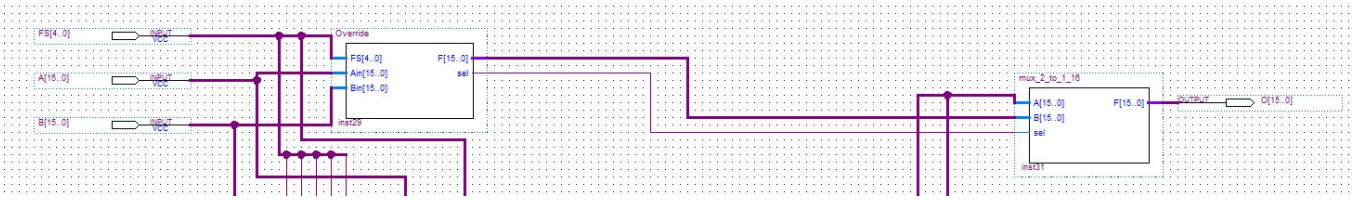


Figure 9: Override Block and Multiplexor Select

See Appendix Override.

This 16 bit ALU consists of 16 individual ALU Cells shown stacked vertically in the middle of the schematic which are individually responsible for each of the bits that are provided from the input. For instance, one ALU cell will receive the A[0] and B[0] bits and output a F[0] bit while another cell will take A[5] and B[5] bits and output a F[5] bit. On the left of the schematic exists carry in circuitry for the first ALU Cell so that all functions will work properly. The top of the schematic implements an ALU override so that additional functions not traditionally supported by the ALU can be easily implemented without changing circuitry. The multiplexer selects between traditional ALU output and the override so that both are supported.

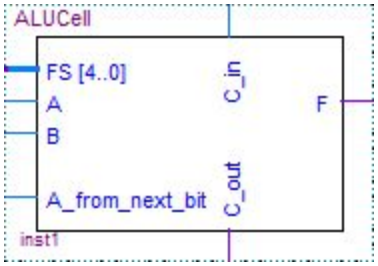


Figure 10: ALU Cell Block

The ALU Cell Block has the inputs of one bit of A data, one bit of B data, a Function Select, a Carry in (C_in), and the A from next bit. Outputs consist of a Carry Out (C_out) and the result of the arithmetic for one bit of A and B (F). A carry bit will propagate between the Carry out of one ALU Cell to the Carry In of the next ALU Cell which is why this design is known as a Ripple Carry Adder. The A_from_next_bit is used for shifting functions.

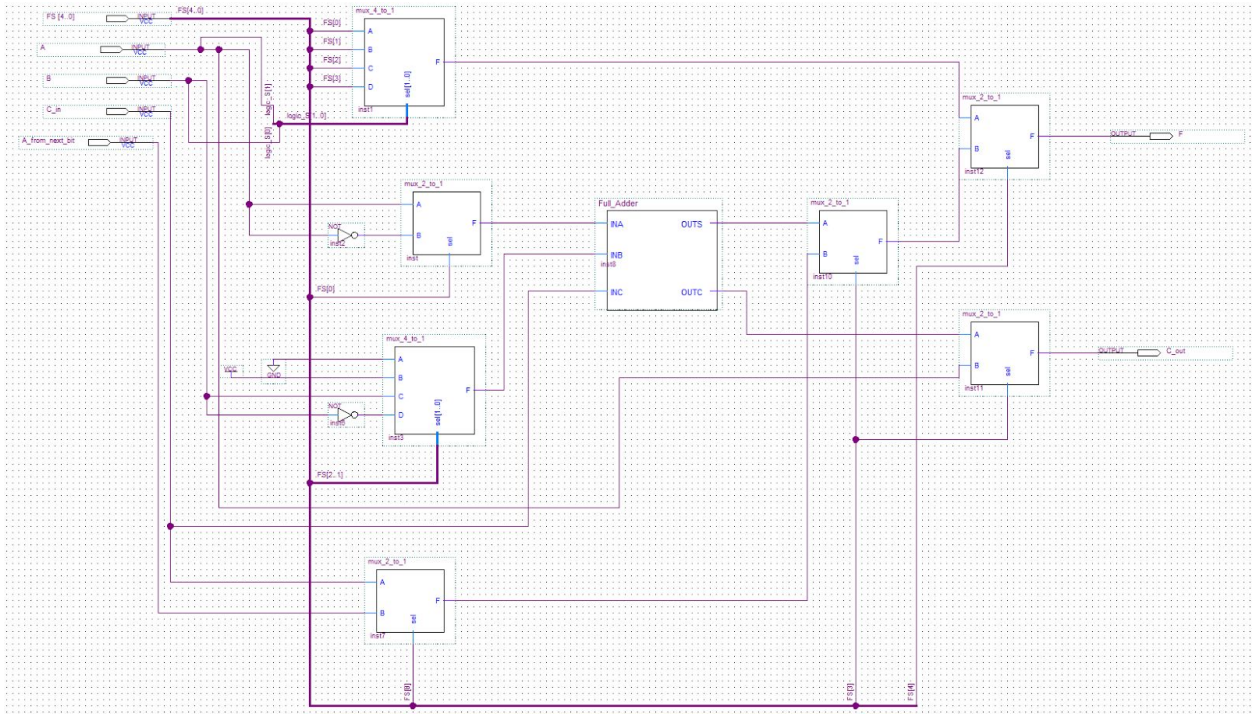


Figure 11: ALU Cell Schematic

The ALU Cell in figure 11 consists primarily of several multiplexers which designate where signals for certain functions will be sent. The 4 to 1 multiplexer at the top handles the simple functions like A&B, A|B, A^B. The shifting operations of A<<1 and A>>1 are handled by the bottom left 2 to 1 multiplexer using the A_from_next_bit to complete the operation. The full adder in the middle of the schematic handles functions like A+B and A-1. The operations of

F=A+1, F=A-B, and F=B were handled with the override block in the ALU since these operations were either not reliably executing in the cell or were not supported.

As stated our ALU design includes a ripple carry adder. This design, although simple, causes a delay because the carry has to propagate through every cell before there is an output. The alternative design, known as a carry lookahead adder, solves this delay problem by calculating the carry signals in advance which allows for a much more efficient and faster ALU. We opted for the simpler design to ensure reliability with the rest of the processor.
The ALU is controlled by a string of 5 bits to the FS input in order to execute a desired operation on A and B. Below is a table of all of the function select codes with additional carry in logic (C_0) and the Register Transfer Language (RTL) for each function.

Table 1: Function selection signals for each ALU operation

Function	F[4]	F[3]	F[2]	F[1]	F[0]	C_0	RTL
A + 1	1	0	0	0	0	1	R[DR] ← R[SA] + 1
A + B	1	0	1	0	0	0	R[DR] ← R[SA] + R[SB]
A – B	1	0	1	1	0	1	R[DR] ← R[SA] - R[SB]
A – 1	1	0	0	1	0	0	R[DR] ← R[SA] - 1
-A	1	0	0	0	1	1	R[DR] ← ~R[SA] + 1
0	0	0	0	0	0	X	NO ACTION
A	0	1	1	0	0	X	R[DR] ← R[SA]
~A	0	0	0	1	1	X	R[DR] ← ~R[SA]
A & B	0	1	0	0	0	X	R[DR] ← R[SA] ^ R[SB]
A | B	0	1	1	1	0	X	R[DR] ← R[SA] v R[SB]
A ^ B	0	0	1	1	0	X	R[DR] ← R[SA] XOR R[SB]
A >> 1 (shift in 0)	1	1	0	0	1	0	R[DR] ← sr R[SA]
A << 1 (shift in 0)	1	1	0	0	0	X	R[DR] ← sl R[SA]

B	0	1	0	1	0	x	R[DR] ← R[SB]

A verilog testbench was created and run through Modelsim-Altera to ensure that the ALU was working properly. This testbench iterates through every function select code to verify that all built-in functions of the ALU are working properly

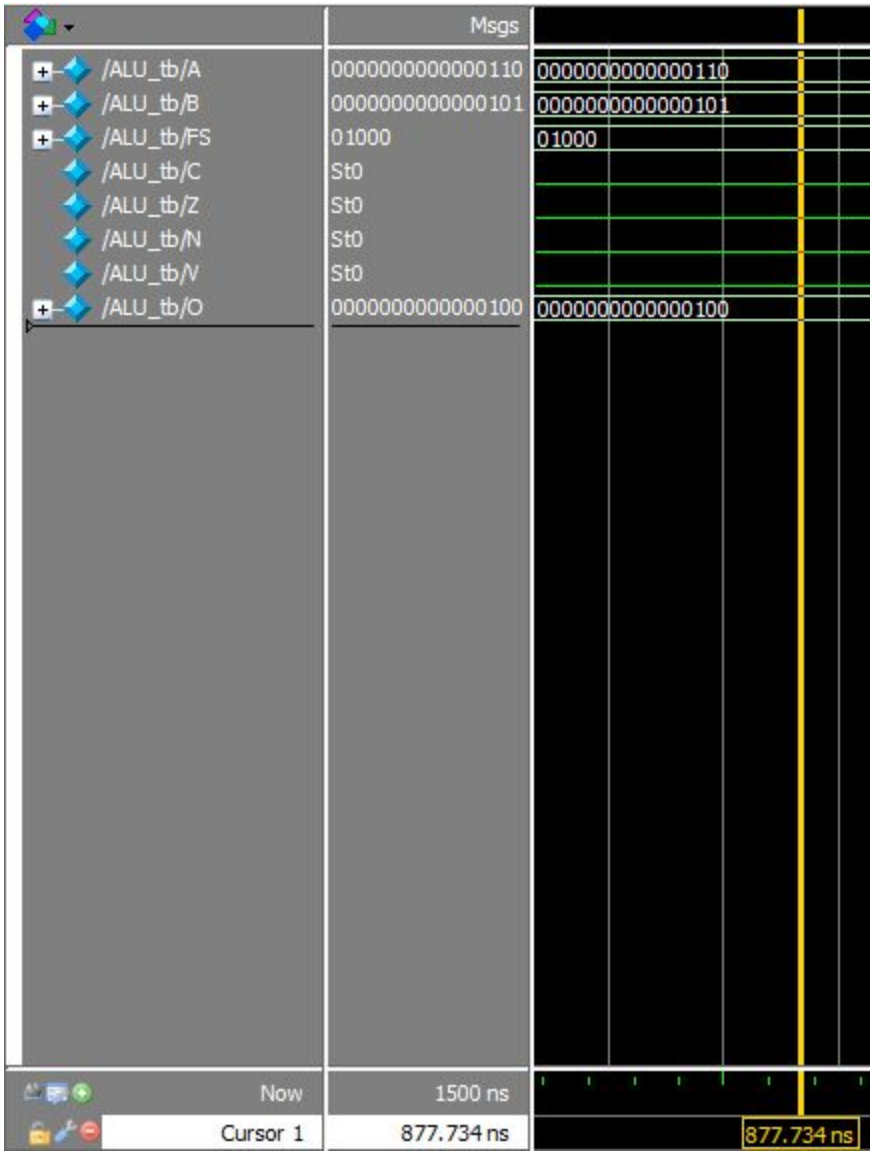


Figure 12: Testbench showing A & B

See Appendix ALU_tb.

In the above testbench A is set to 16’b0000000000000110 and B is set to 16’b0000000000000101 and the output F is shown to be 16’b0000000000000100 as it should.

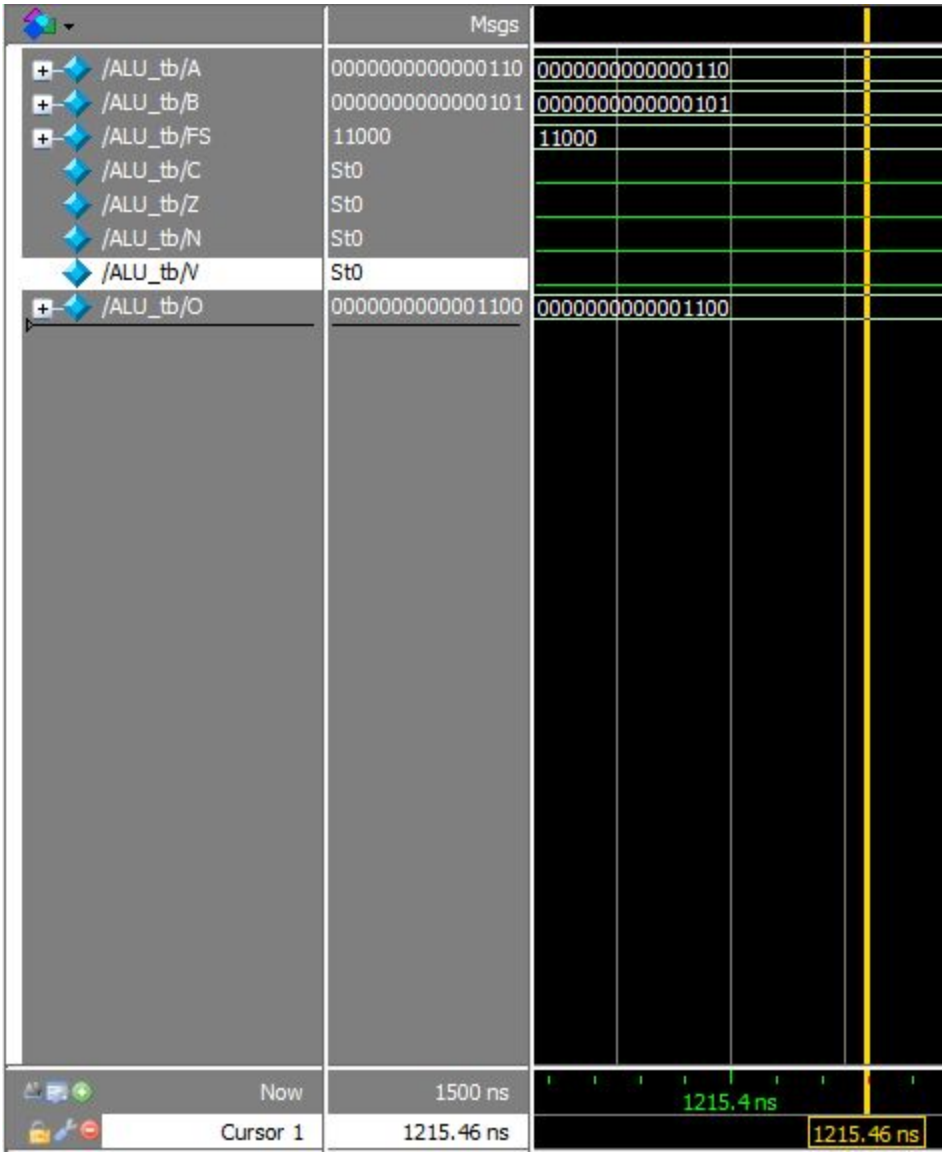


Figure 13: Testbench showing A << 1

See Appendix ALU_tb.

In the above testbench A is set to 16’b0000000000000110 and B is set to 16’b0000000000000101 and the output F is shown to be 16’b0000000000001100 as it should.

Memory organization

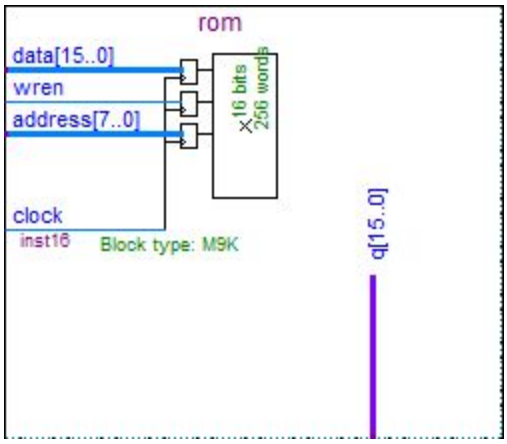


Figure 14: Program Memory/Data Memory Block

This processor is a Harvard architecture which means that it contains separate program and data memory blocks. To implement this, two 256x16 word M9K blocks were instantiated using Altera's Megawizard. The memory blocks contain a clock, 8 bit address, write enable, and 16 bit data line inputs with a 16 bit Q output as seen in figure 14. For the program memory, the address is provided by the program counter while the data is provided either externally or already programmed in. For the data memory, the address is provided by the A bus line while the data is provided by the B bus line. When the memory is being written to, the WREN line is set high while the required data and address are provided. The data will then be written to the memory block at the designated address and output that data at Q. When reading from memory, the WREN is set low and an address is provided which will output the data from that address at Q. Both reading and writing are timed to occur on the negative edge of the clock cycle. Doing this prevents reading from the memory taking two clocks instead of just one.

Datapath Design

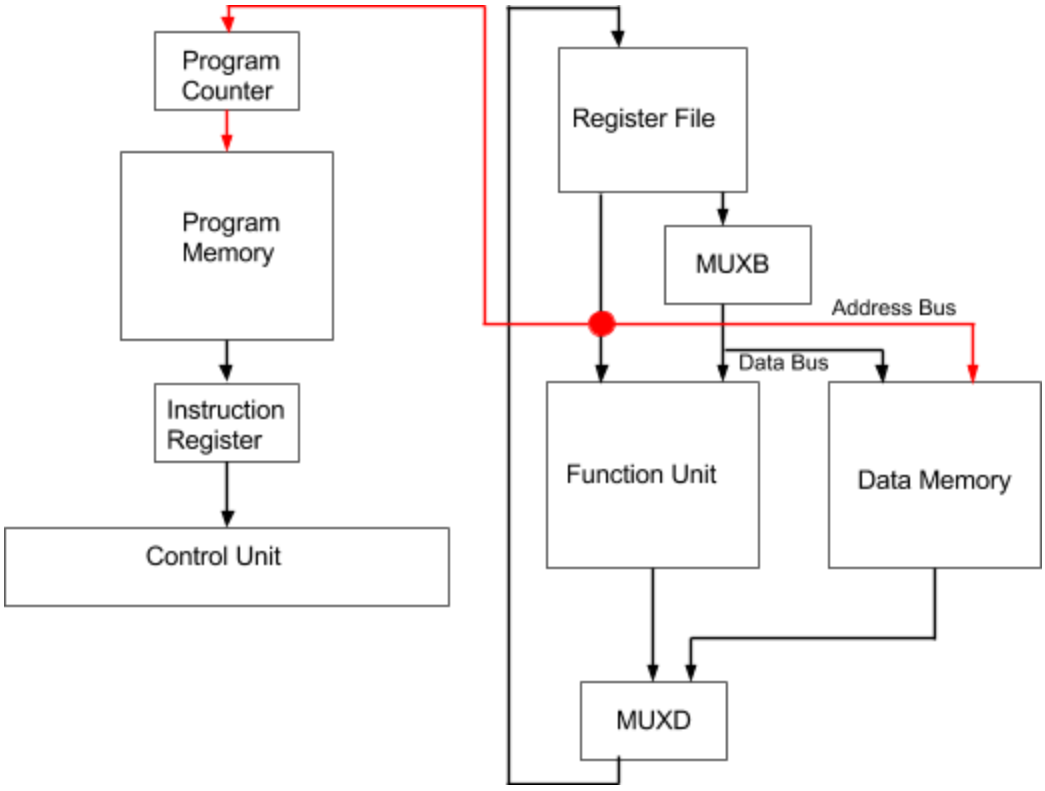


Figure 15: Datapath Block Diagram

The Datapath consists of the Register File, ALU, Data Memory, multiplexers B and D, the program counter, program memory, and the Instruction Register. The Control Unit is shown but is a part of the control section of the processor.

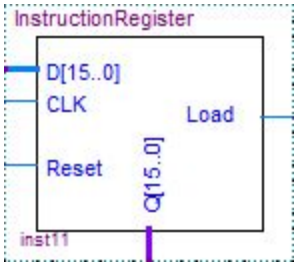


Figure 16: Instruction Register Block

The Instruction register has reset, clock (CLK), 16 bit data (D), and Load inputs with a 16 bit output (Q).

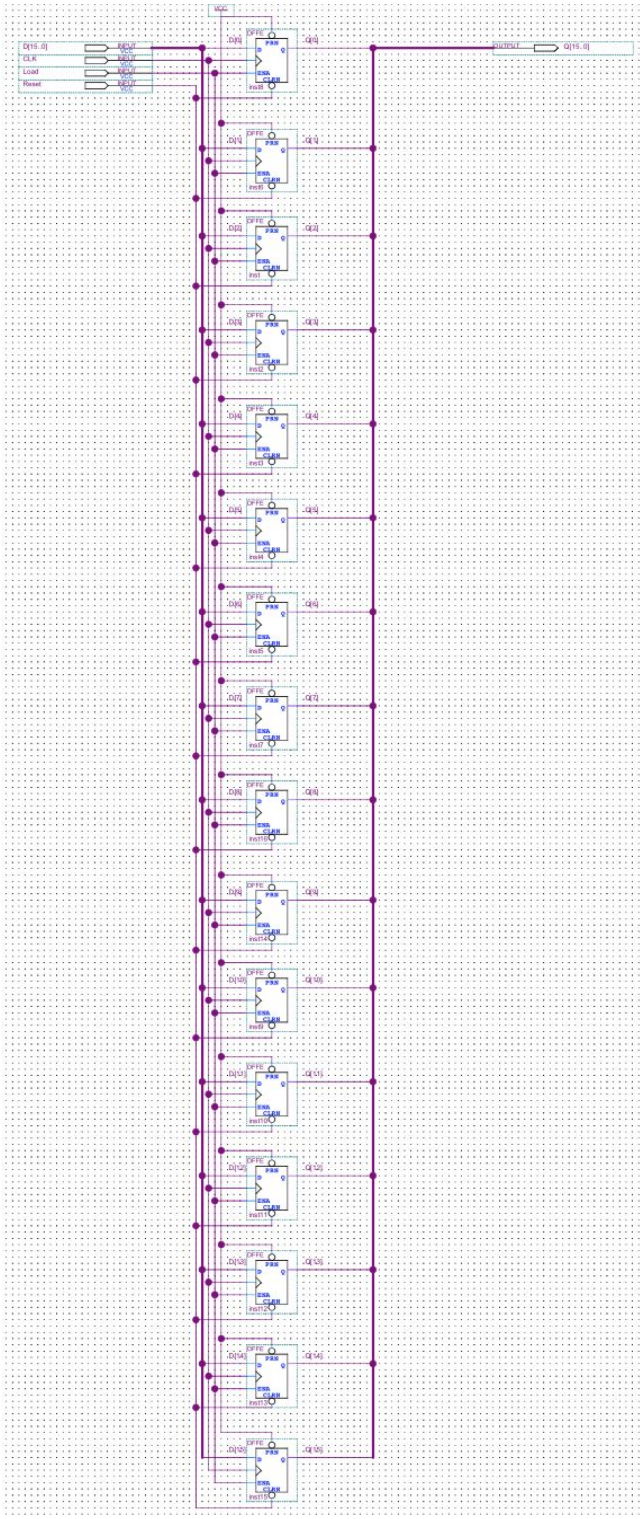


Figure 17: Instruction Register Schematic

Figure 17 shows a diagram of the 16 bit Instruction Register. It is composed of 16 D-Latch flip flops and will store values on a rising clock edge when load is high and reset is high as reset is active low.

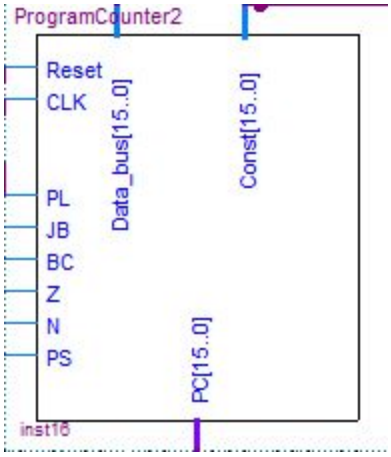


Figure 18: Program Counter Block

The Program Counter has the following inputs; a 16 bit Data_Bus, a 16 bit Constant, a negative bit flag (N), a zero bit flag (Z), a branch control flag (BC), a jump/branch control flag (JB), a program load (PL), a program select (PS), a clock, and a reset. The output will be a 16 bit address (PC). Only the the least significant 8 bits will be passed as an address; however, due to a constraint with the design of the control unit and opcodes.

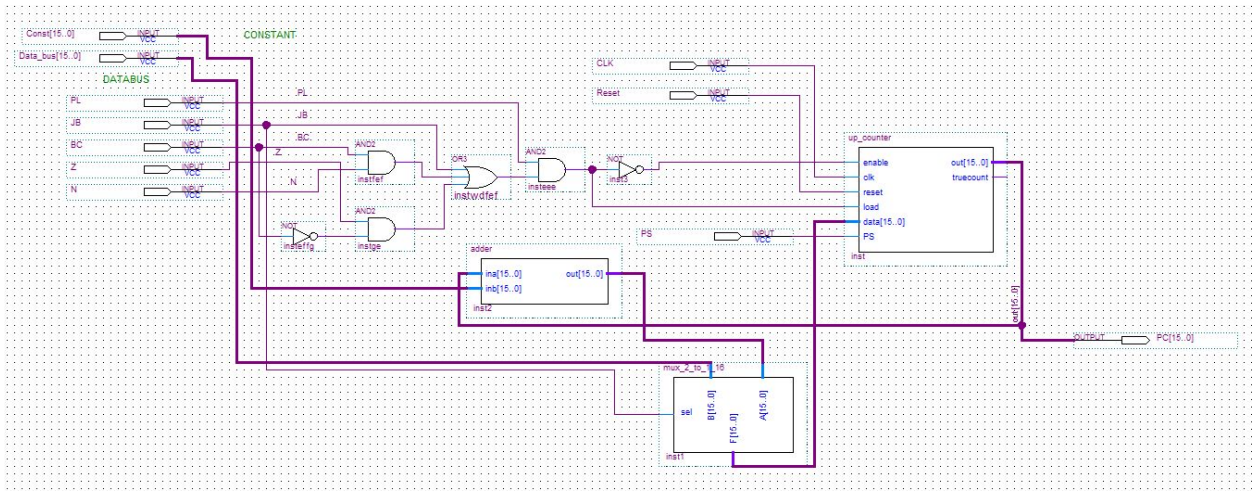


Figure 19: Program Counter Schematic

In the schematic shown in figure 19, there exists control logic, a 16 bit adder, a 2 to 1 multiplexer, and a 16-bit counter.

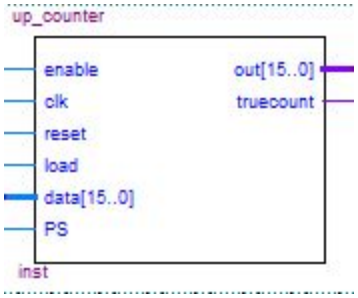


Figure 20: 16-bit counter block

The counter has clock (clk), reset, load, enable, PS, and a 16 bit data_in input with a 16 bit data output (out). When enable is high and load is low, the counter will add one bit to the data_in and output it. When enable is low and load is high, the counter will pass the value in data_in through to the output.

Generally, the program counter will be enabled and increment an address with each clock cycle while it will load otherwise when certain flags are set. Below is a table which shows conditions which will lead to the 16 bit counter loading or incrementing.

Table 2: Program Counter Function

PL	BC	JB	Z	N	Result
1	0	1	0	0	Load
1	0	0	1	0	Load
1	1	0	0	1	Load
0	x	x	x	x	Increment

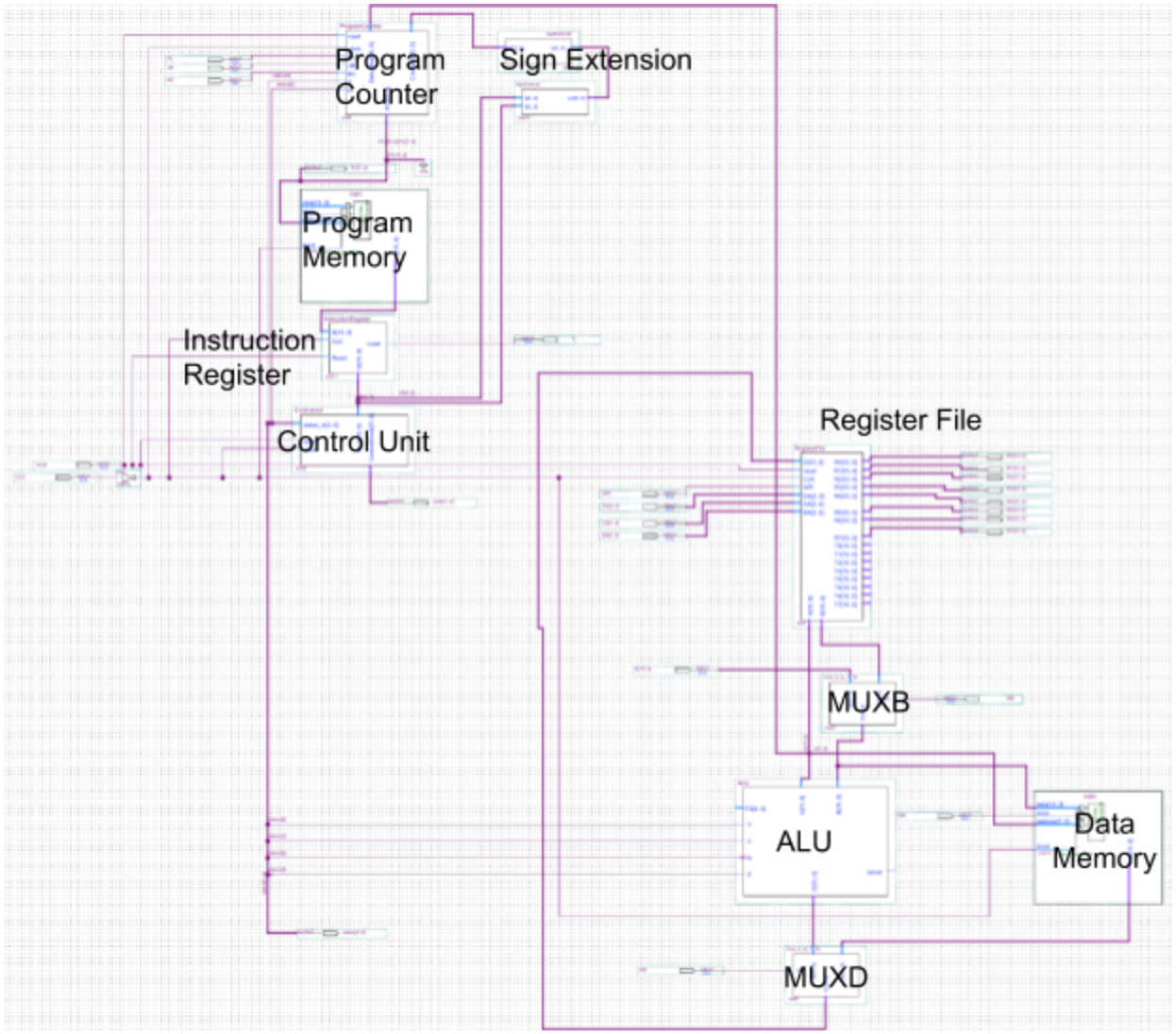


Figure 21: Datapath Schematic

In order to create a Control Unit for this Datapath, the Control Word and signals had to be established. The Control word is simply the combination of all of the control signals together into one string of bits. For this datapath, a 38 bit long Control Word was created and shown in the table below.

Table 3: Control Word: 39'b

PL	AA	BA	DA	MB	FS	MD	RW	MW	JB	BC	IL	PS	K
1'b	3'b	3'b	3'b	1'b	5'b	1'b	1'b	1'b	1'b	1'b	1'b	1'b	16'b

All of the control signals and their functions are listed in the table below.

Table 4: Control Signals and Functions

Signal	Function
PL	PROGRAM LOAD (1 FOR BRANCH OR JUMP, 0 OTHERWISE)
AA	A SELECT ADDRESS
BA	B SELECT ADDRESS
DA	DESTINATION ADDRESS
MB	MUX FOR B SELECT
FS	FUNCTION SELECT
MD	MUX FOR D BUS SELECT
RW	REGISTER FILE WRITE ENABLE
MW	MEMORY WRITE ENABLE
JB	JUMP OR BRANCH
BC	BRANCH CONTROL
IL	INSTRUCTION LOAD
PS	PROGRAM COUNTER SELECT(0-Hold,1-increment)
K	CONSTANT

Table 5: Control Signal MB

Value	**Result**
0	Register File B out
1	Constant

Table 6: Control Signal MD

Value	**Result**
0	FS out
1	Memory out

Table 7: Control Signals PL, JB, BC

PL	**JB**	**BC**	**PC operation**
0	X	X	Count up
1	1	X	JUMP
1	0	1	BRANCH ON NEGATIVE (ELSE COUNT UP)
1	0	0	BRANCH ON ZERO(ELSE COUNT UP)

The design of this datapath strives to be as streamlined as possible in order to ensure reliability and ease of testing. With this datapath and control word, it is possible to read and write to both registers and memory. Performing arithmetic operations on the data in these registers is easy with the ALU directly following the Register File. Branching and Jumping is possible due to the Branch/Jump control built into the program counter. The use of literals in arithmetic functions is also easy due to the 16 bit constant integrated into the control word. Sign extension is also an ability that is built into the feedback of the program counter. Overall, this datapath is robust and elegant with its simplicity.

Control Unit Design

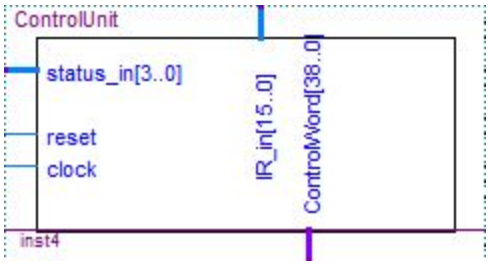


Figure 22: Control Unit Block

The Control Unit has a 16 bit instruction register input (IR_in), along with clock, reset, and a 4 bit status_in which contains the status signals of Zero, Negative, Carry, and Overflow.

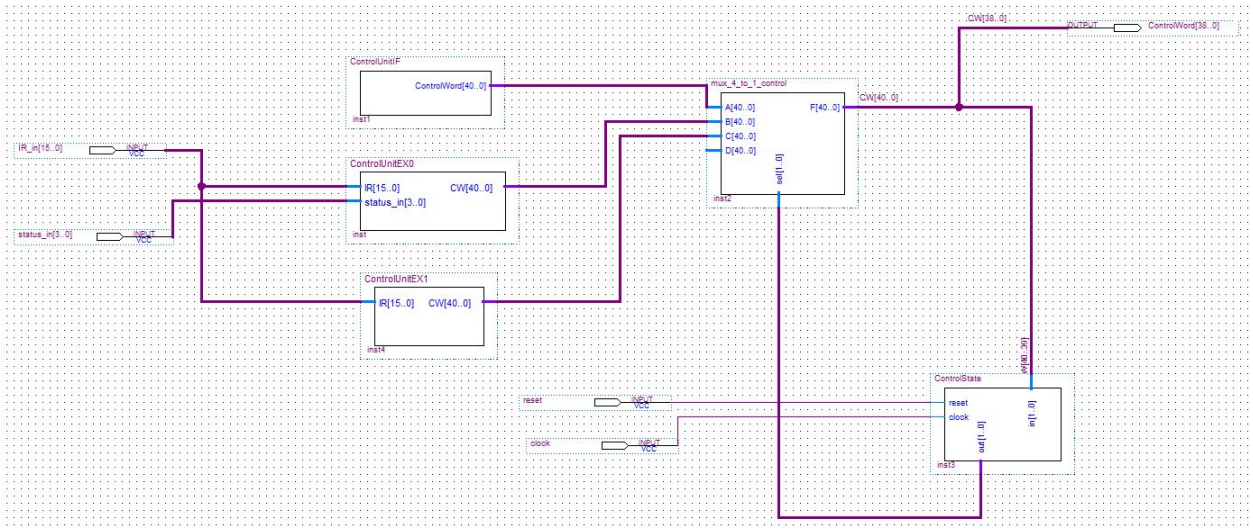


Figure 23: Control Unit Top Level Schematic

The top level schematic seen in figure 23 shows the different states that the implemented instructions could be in. These include Instruction Fetch (IF), EX0, and EX1. All instructions will use the IF and EX0 states but not all will take advantage of the EX1 state.

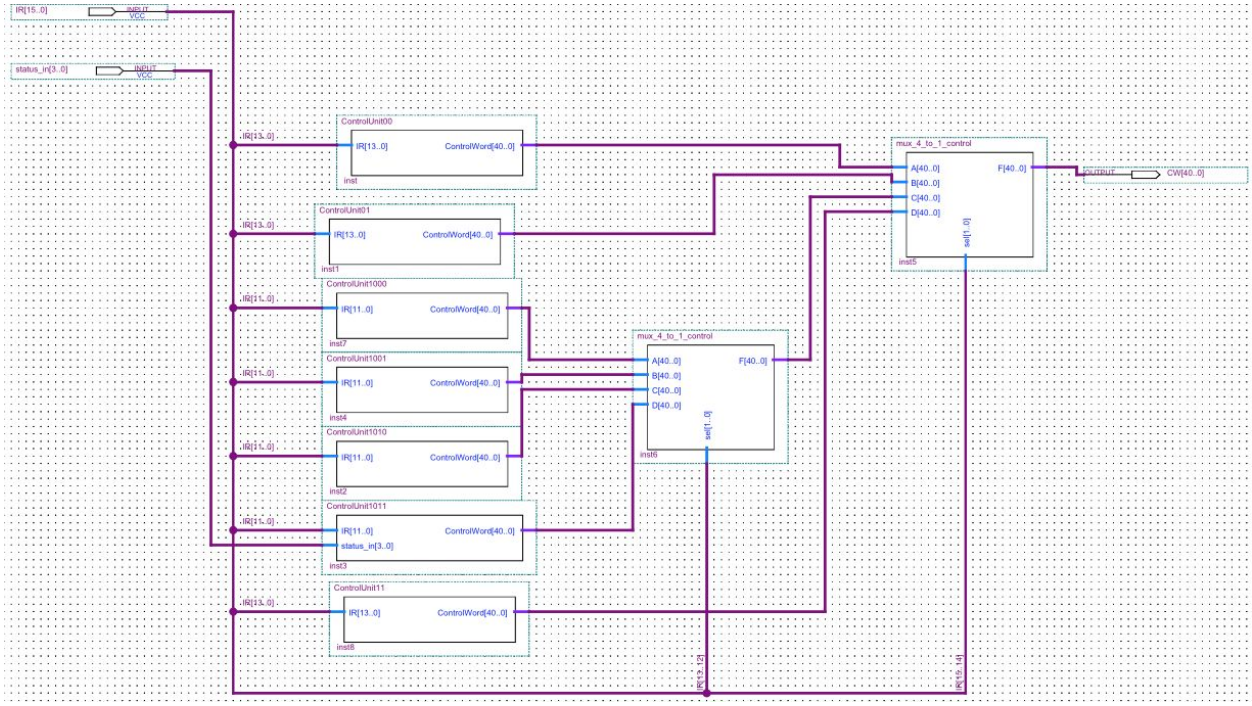


Figure 24: Control Unit EX0 Schematic

Figure 24 distinguishes different instructions based on the beginning of their opcodes. The opcode is a string of bits that will begin every instruction and designated what function it will be performing. The EX0 state breaks up the opcodes into ones with most significant bits (MSB) starting with 00, 01, 1000, 1001, 1010, 1011, and 11. Two 4 to 1 multiplexers are used in order to select which control unit implementation to use based on the opcodes.

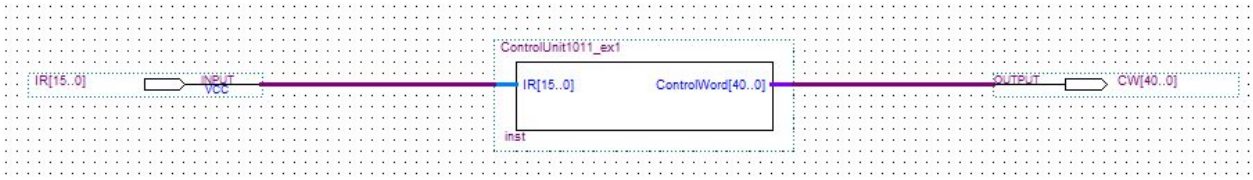


Figure 25: Control Unit EX1

The Control Unit EX1 is used by Branch on Zero (BRZ) and Branch on Negative (BRN).

State Diagrams used by all of the built in instructions based on their opcode are shown below.

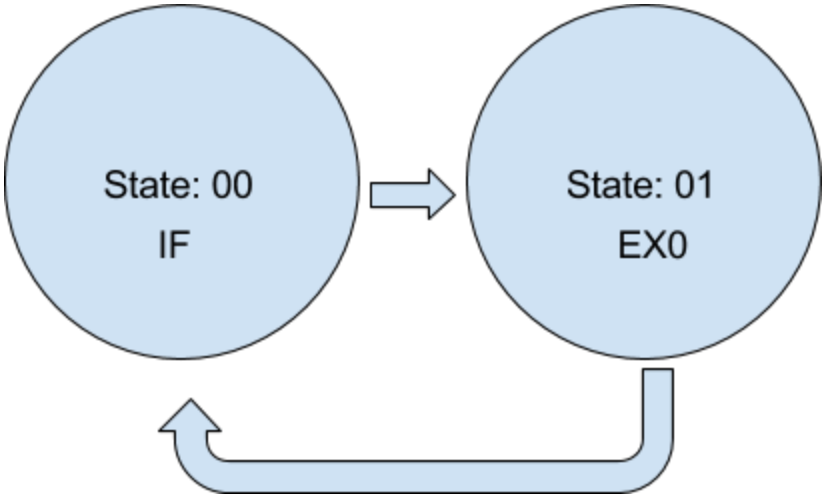


Figure 26: State Diagram for Opcodes MSB 00, 01, 1000, 1001, 1010, 11

Initially the instruction will be fetched and then the first state being EX0 will be executed. After that the next instruction will be fetched

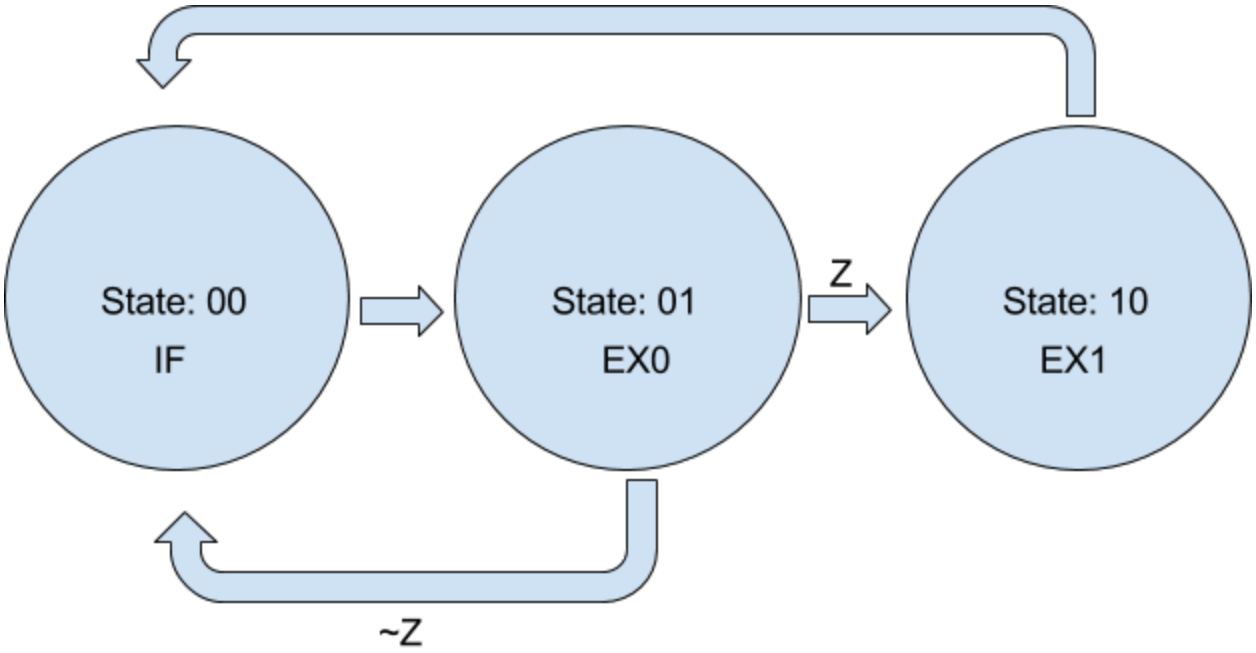


Figure 27: State Diagram For Opcode MSB 1011 BRZ

Initially the instruction will be fetched and then the first state being EX0 will be executed. If the Zero Flag is low, the next instruction will be fetched. If the Zero flag is high, the next state being EX1 will execute. Then the next instruction will be fetched.

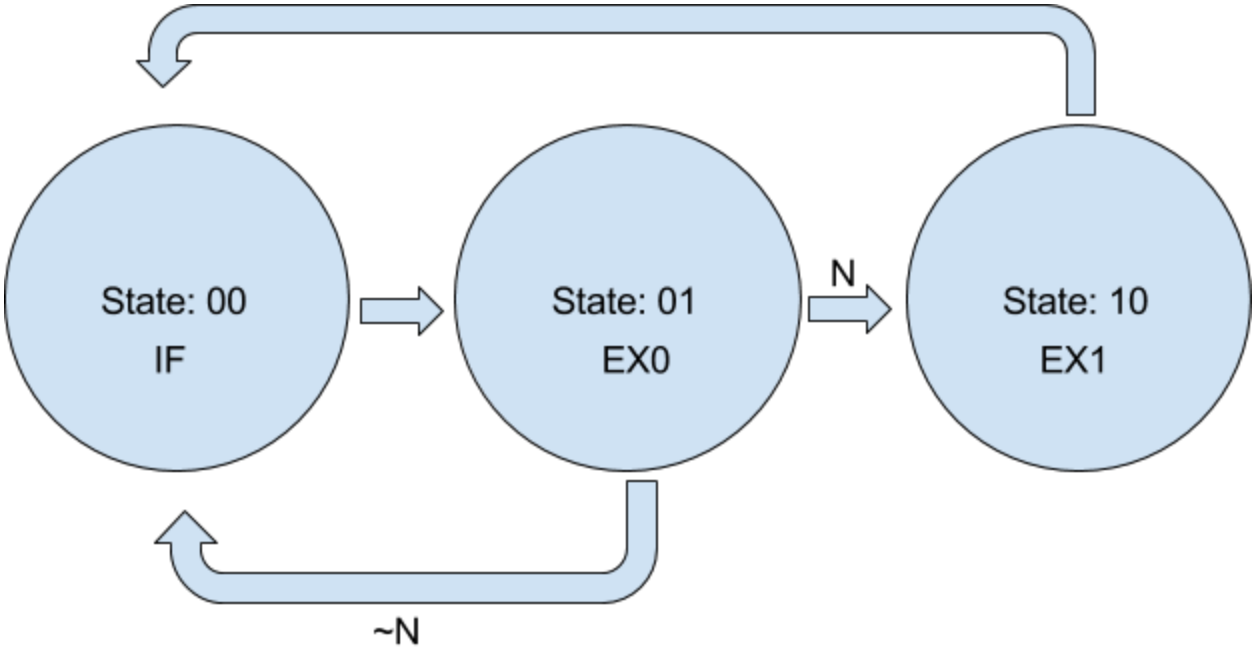


Figure 28: State Diagram For Opcode MSB 1011 BRN

Initially the instruction will be fetched and then the first state being EX0 will be executed. If the Negative Flag is low, the next instruction will be fetched. If the Negative flag is high, the next state being EX1 will execute. Then the next instruction will be fetched

A verilog testbench was created and run in Modelsim-Altera in order to test the functionality of the control unit. An instruction that will ADD R7 and R4 and write into register R7 will be performed while showing the control word for each state. This instruction should start in the instruction fetch state, proceed to the EX0 state, then return to the instruction fetch state.

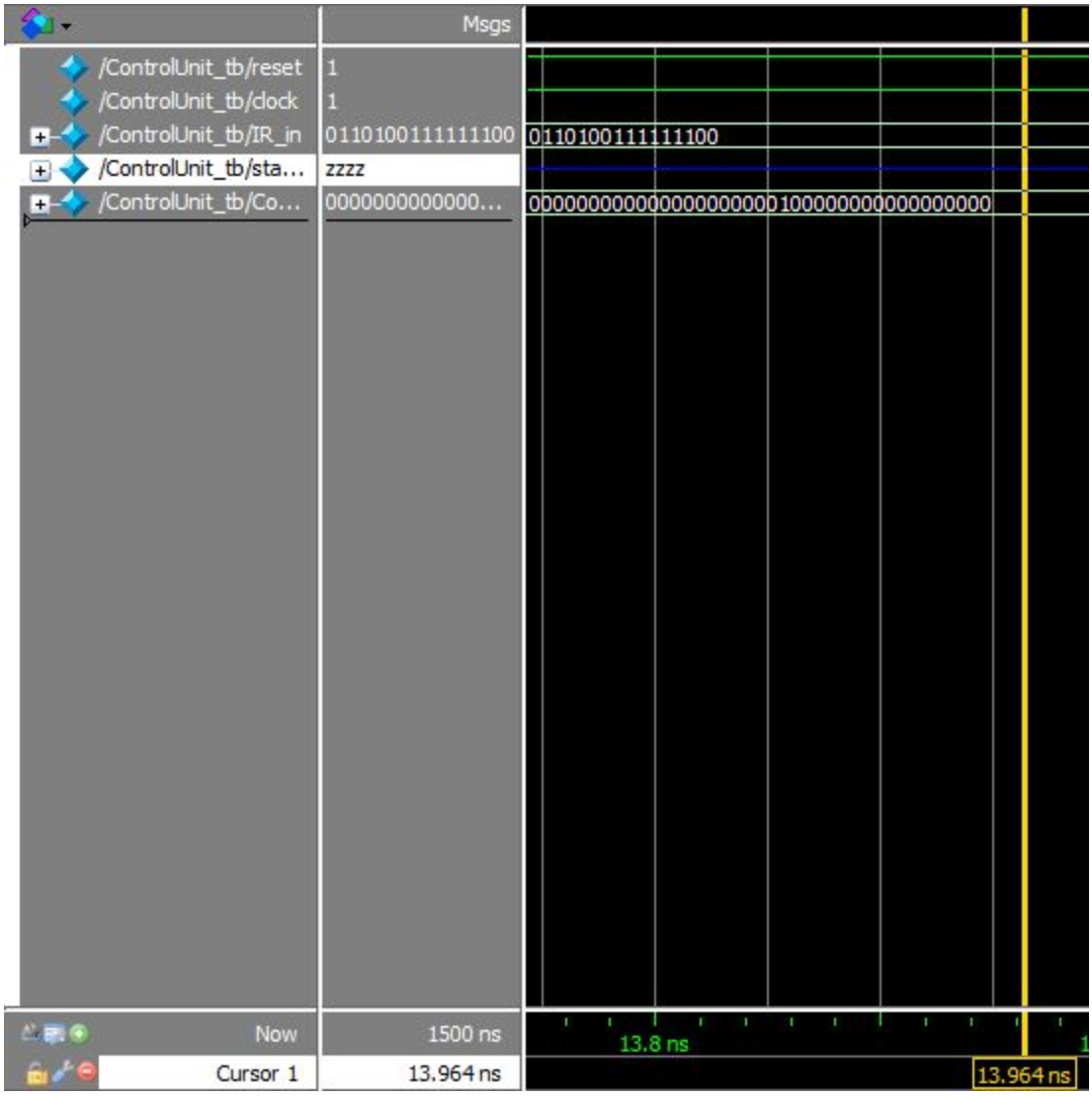


Figure 29: Shows the simulation starting with the control word for state IF 2'b00

See Appendix ControlUnit_tb.

The simulation starts off with a control word of
39'b000000000000000000000100000000000000000
Indicating that the program is starting in the Instruction Fetch (IF) state since the instruction load is high.

The Instruction to perform the operation R[7] ← R[7] + R[4] is 16'b0110100111111100. Below is a table which shows how this instruction is delineated.

Table 8: Register Format of ADD Instruction

IR[15:9]	**IR[8:6]**	**IR[5:3]**	**IR[2:0]**
7 bit Opcode	3 bit DR	3 bit SA	3 bit SB
7'b0110100	3'b111	3'b111	3'b100

Control Word: 38'b00000000000000000000010000000000000000
PL = 1'b0
AA = 3'b000
BA = 3'b000
DA = 3'b000
MB = 1'b0
FS = 5'b00000
MD = 1'b0
RW = 1'b0
MW = 1'b0
JB = 1'b0
BC = 1'b0
IL = 1'b1
PS = 1'b0
K = 16'b0

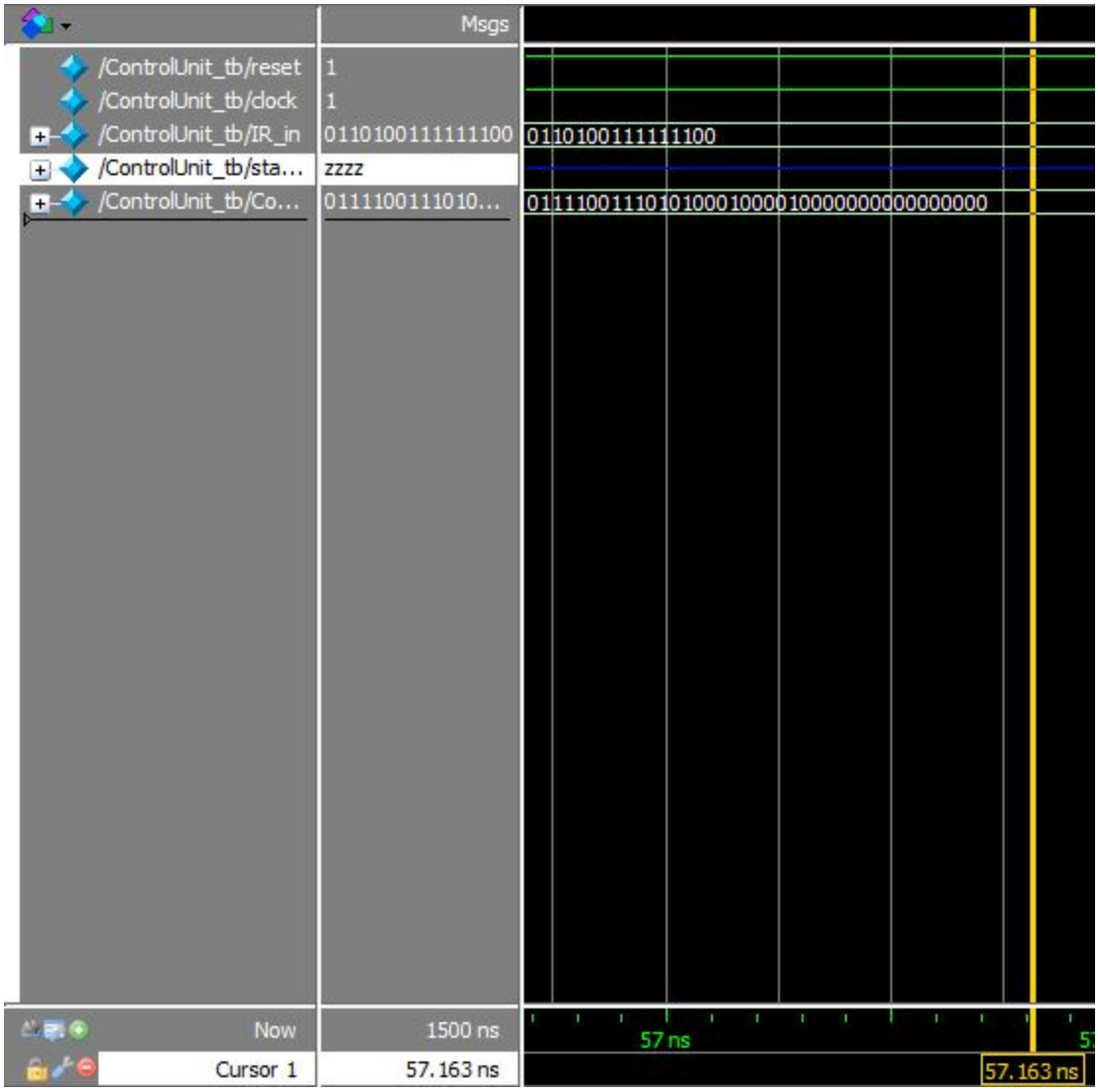


Figure 30: Shows the control word for state EX0 2'b01

See Appendix ControlUnit_tb.

The simulation then goes into state EX0 with the Control word of
39'b011110011101010001000010000000000000000
As indicated by values being set for other signals in the control word. Also, the instruction load (IL) is low while the program load (PL) is high.

The Instruction to perform the operation R[7] ← R[7] + R[4] is 16'b0110100111111100. Below is a table which shows how this instruction is delineated.

Table 9: Register Format of ADD Instruction

IR[15:9]	**IR[8:6]**	**IR[5:3]**	**IR[2:0]**
7 bit Opcode	3 bit DR	3 bit SA	3 bit SB
7'b0110100	3'b111	3'b111	3'b100

Control Word: 39'b0111100111010100010000100000000000000000
PL = 1'b0
AA = 3'b111
BA = 3'b100
DA = 3'b111
MB = 1'b0
FS = 5'b10100
MD = 1'b0
RW = 1'b1
MW = 1'b0
JB = 1'b0
BC = 1'b0
IL = 1'b0
PS = 1'b1
K = 16'b0

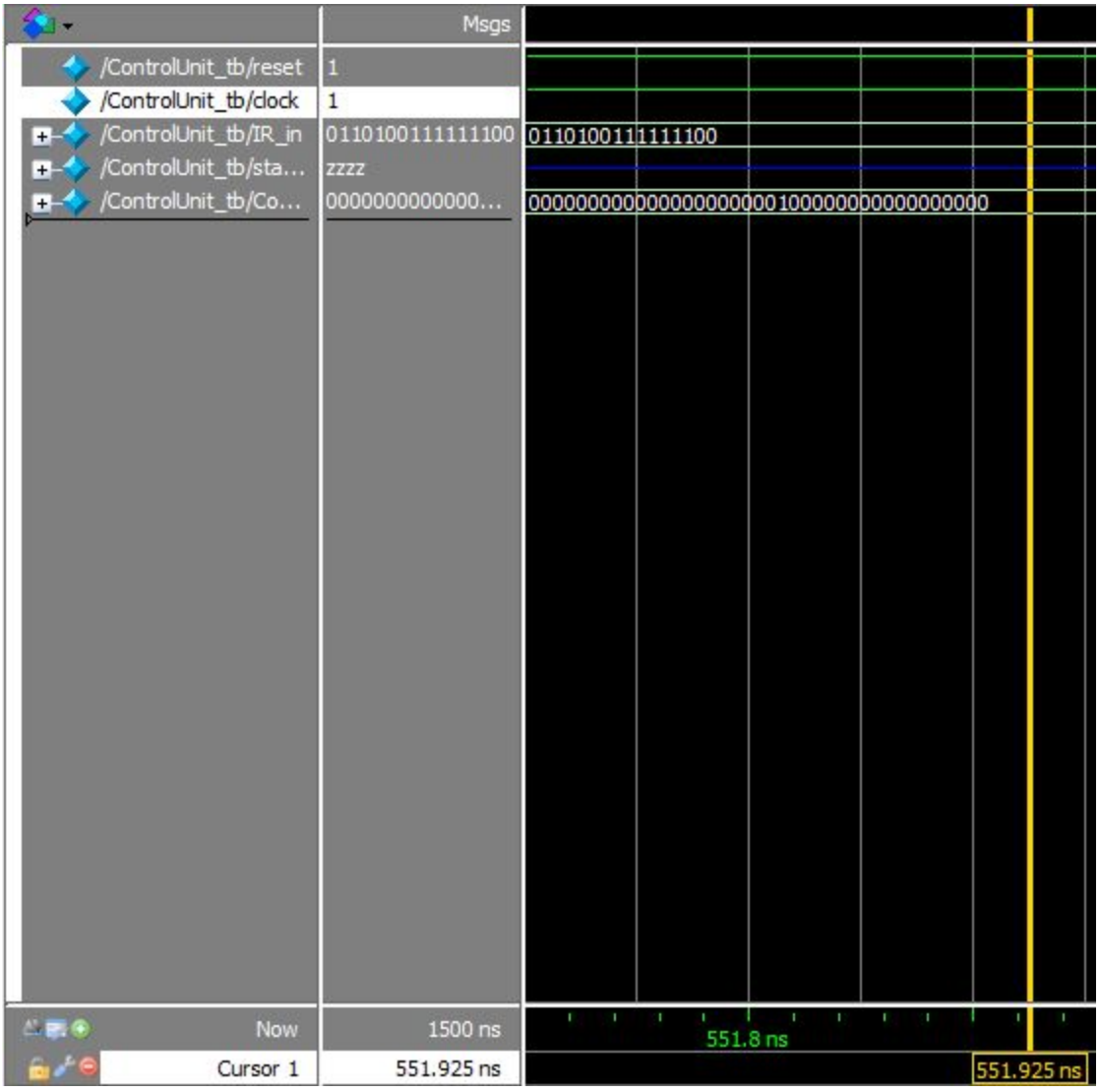


Figure 31: Shows the simulation ending with the control word for state IF 2'b00

See Appendix ControlUnit_tb.

The simulation ends with a control word of
39'b000000000000000000000010000000000000000
Indicating that the program is ending in the Instruction Fetch (IF) state since the instruction load is high.

CPU Design

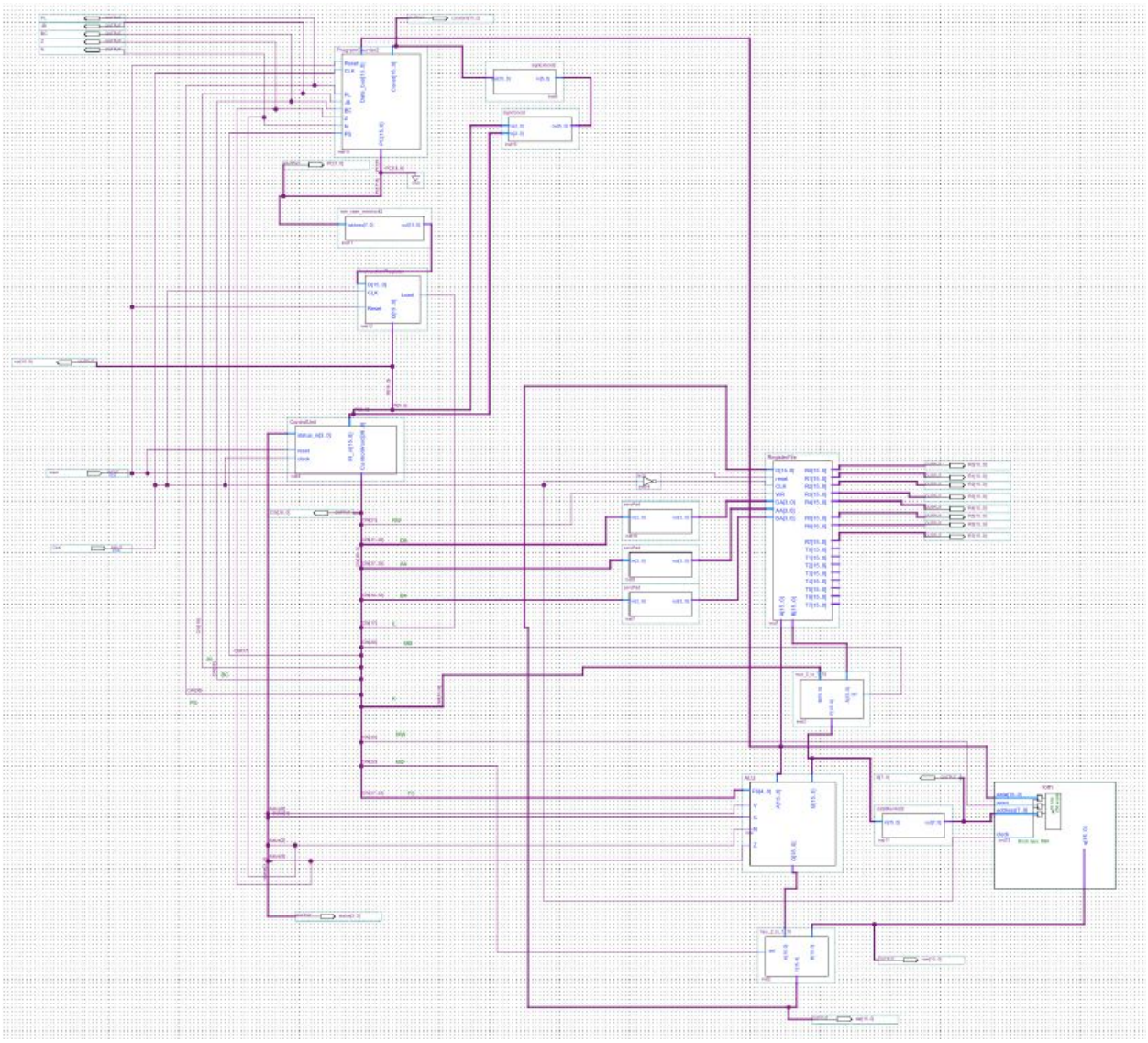


Figure 32: Processor Datapath and Control Schematic

Seen is figure 32 is the full schematic used for the processor. It includes all elements of the datapath along with the control unit and signals that connect to the other components.

A testbench in verilog was created and run in Modelsim-Altera in order to verify the functionality of the processor. In the testbench, several instructions will be executed and the registers will update accordingly.

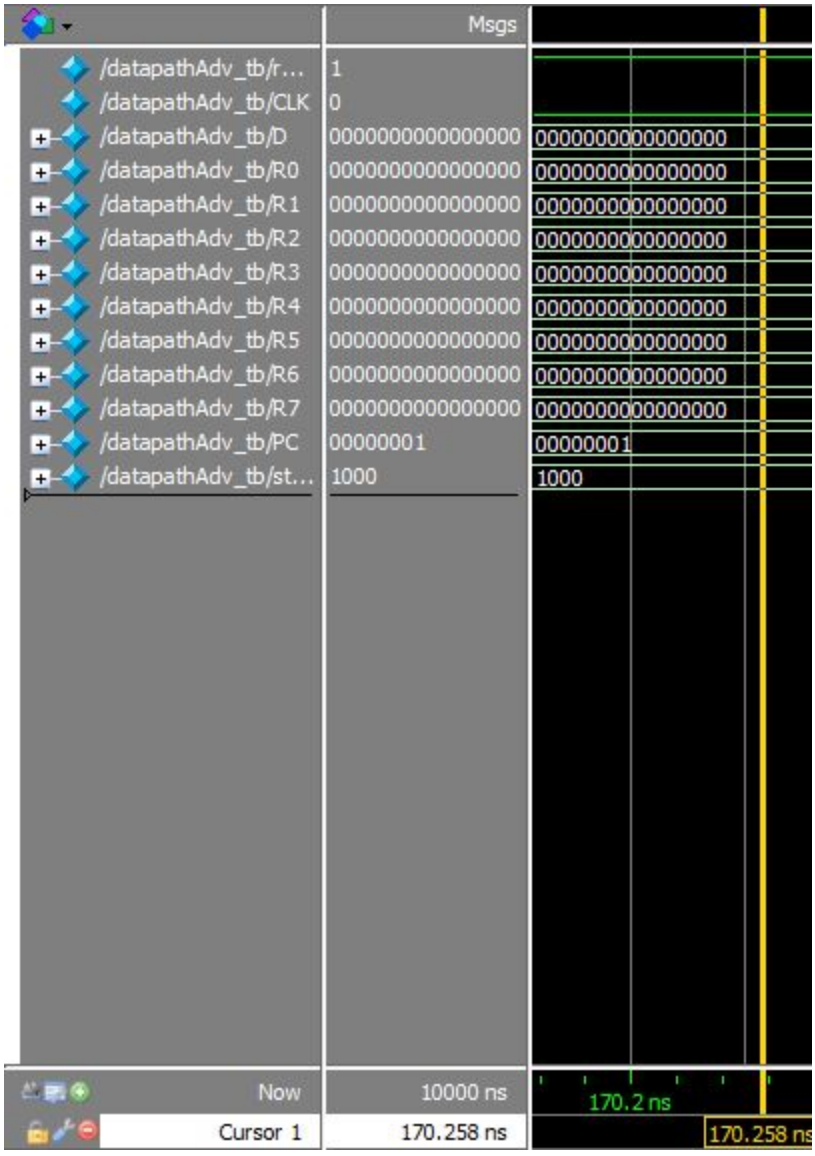


Figure 33: Instruction:NOP. Start of the testbench

See Appendix datapathAdv_tb

This is the start of the testbench where the first instruction is NOP and is shown in D as 16'b0000000000000000 and all of the registers are empty.

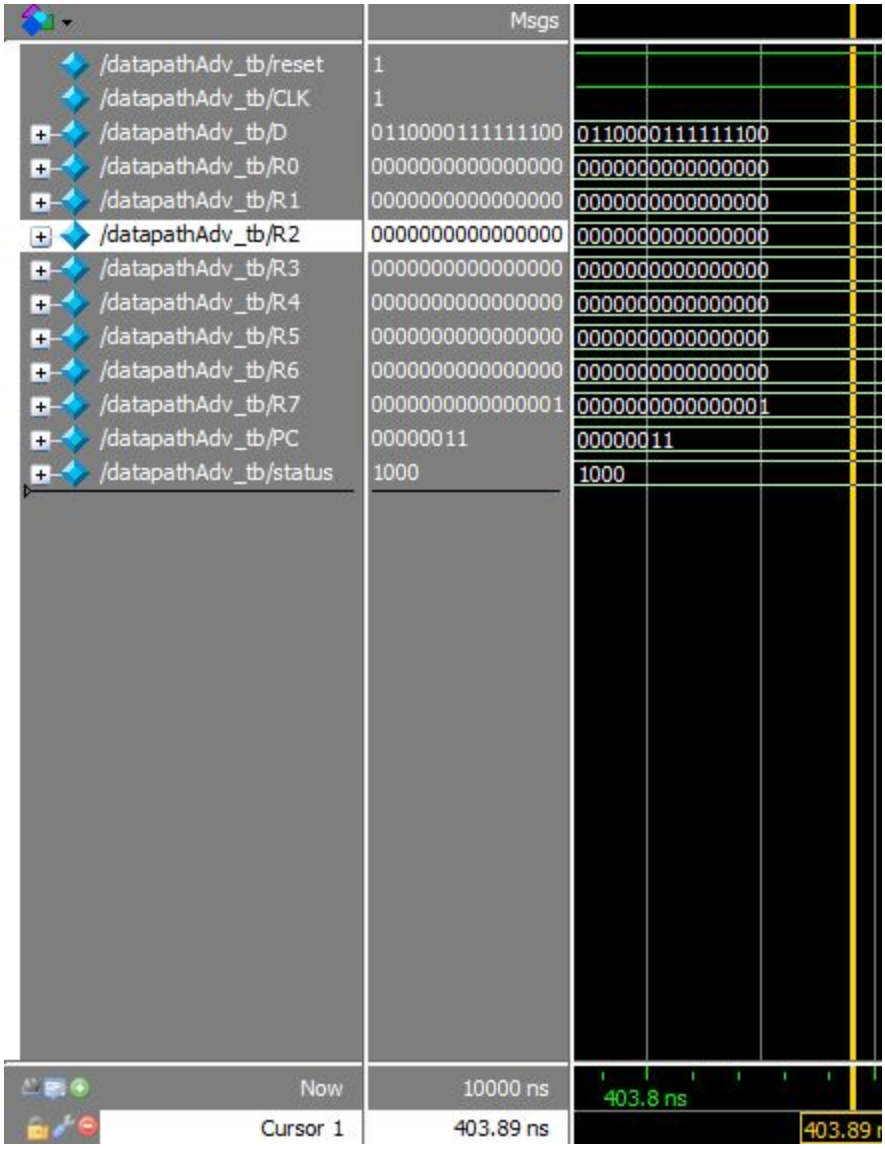


Figure 34: Instruction: INC. R[7] ← R[7] + 1

See Appendix datapathAdv_tb.

D:16’b0110000111111100

This instruction increments The value in R7 and writes it back into R7. Originally R7 is 16’b0 and then it becomes 16’b0000000000000001 after this instruction executes.

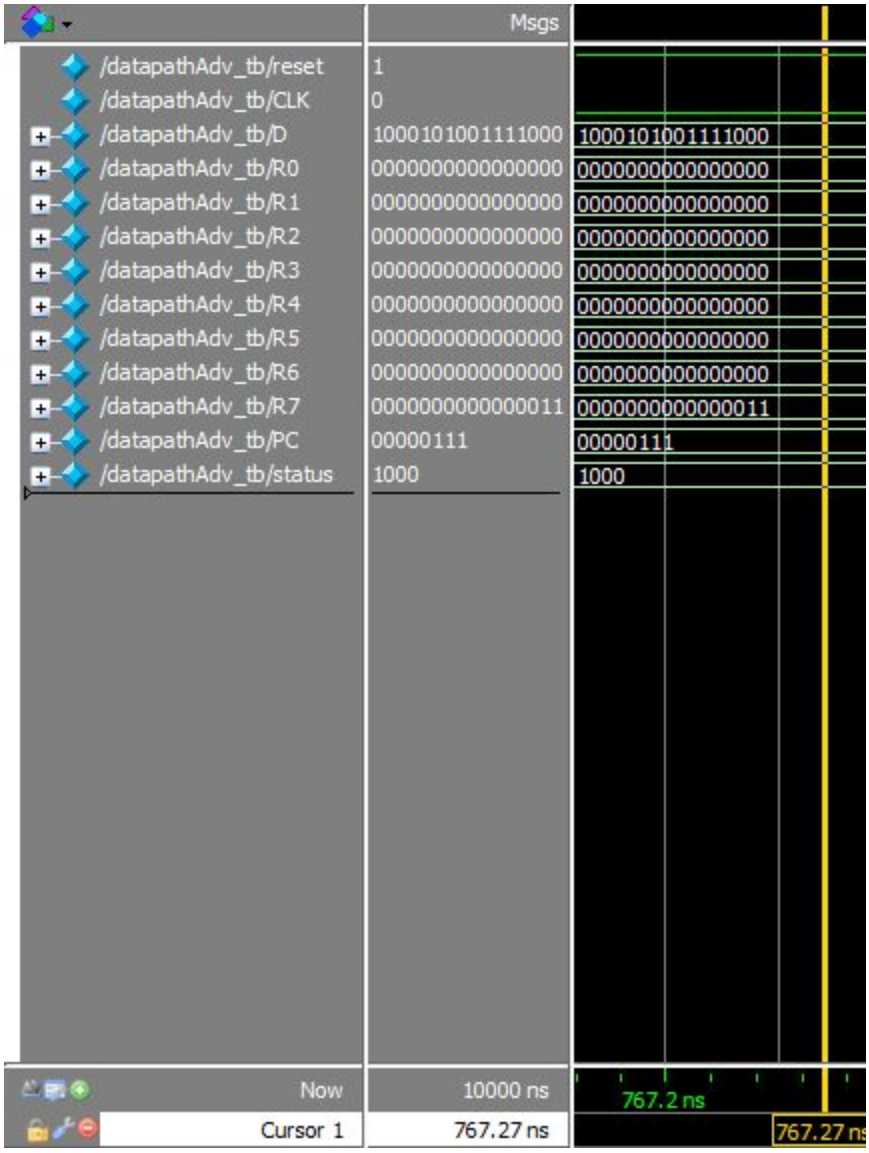


Figure 35: Instruction: STR. M[R[1]] ← R[7]

See Appendix datapathAdv_tb.

D: 16'b1000101001111000

This instruction stores the value of 2'b11 in R7 into Memory Address 3'b001.

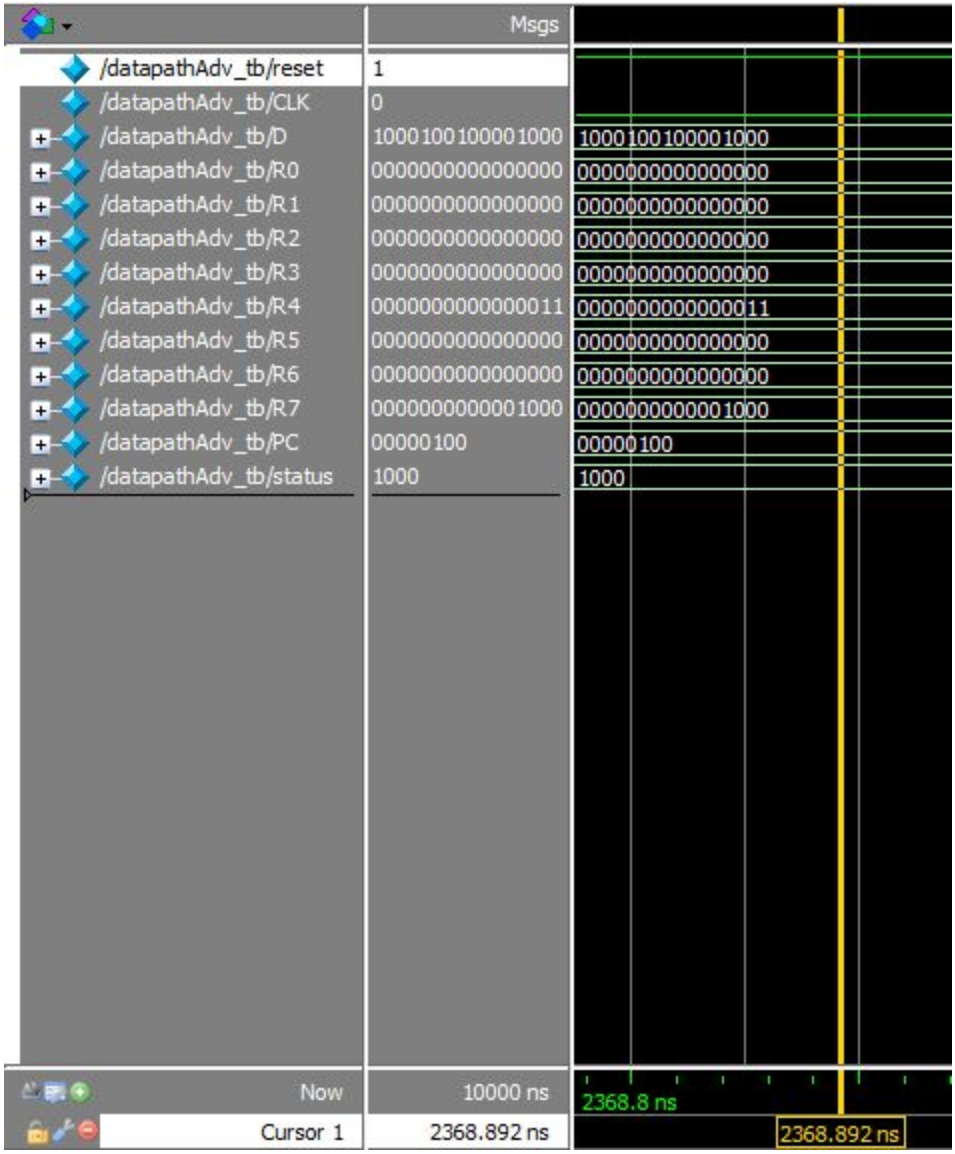


Figure 36: Instruction: LDR. R[4] ← M[R[1]]

See Appendix datapathAdv_tb.

D: 16'b1000100100001000

This instruction loads the value of 2’b11 in Memory address 3’b001 into R4.

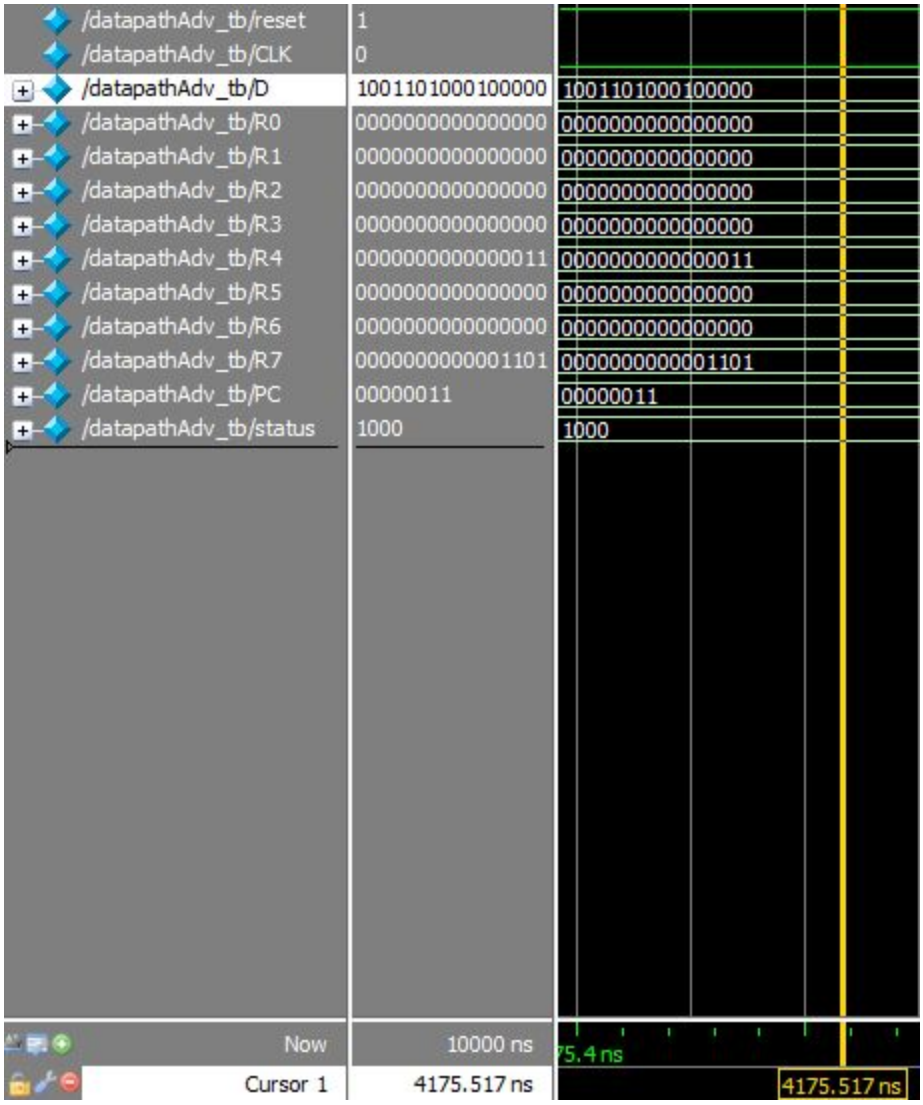


Figure 37: Instruction: JMPR. PC ← R[4]

See Appendix datapathAdv_tb.

JMPR: 16'b1001101000100000

This instruction makes the Program Counter jump to address in R4 of 2’b11

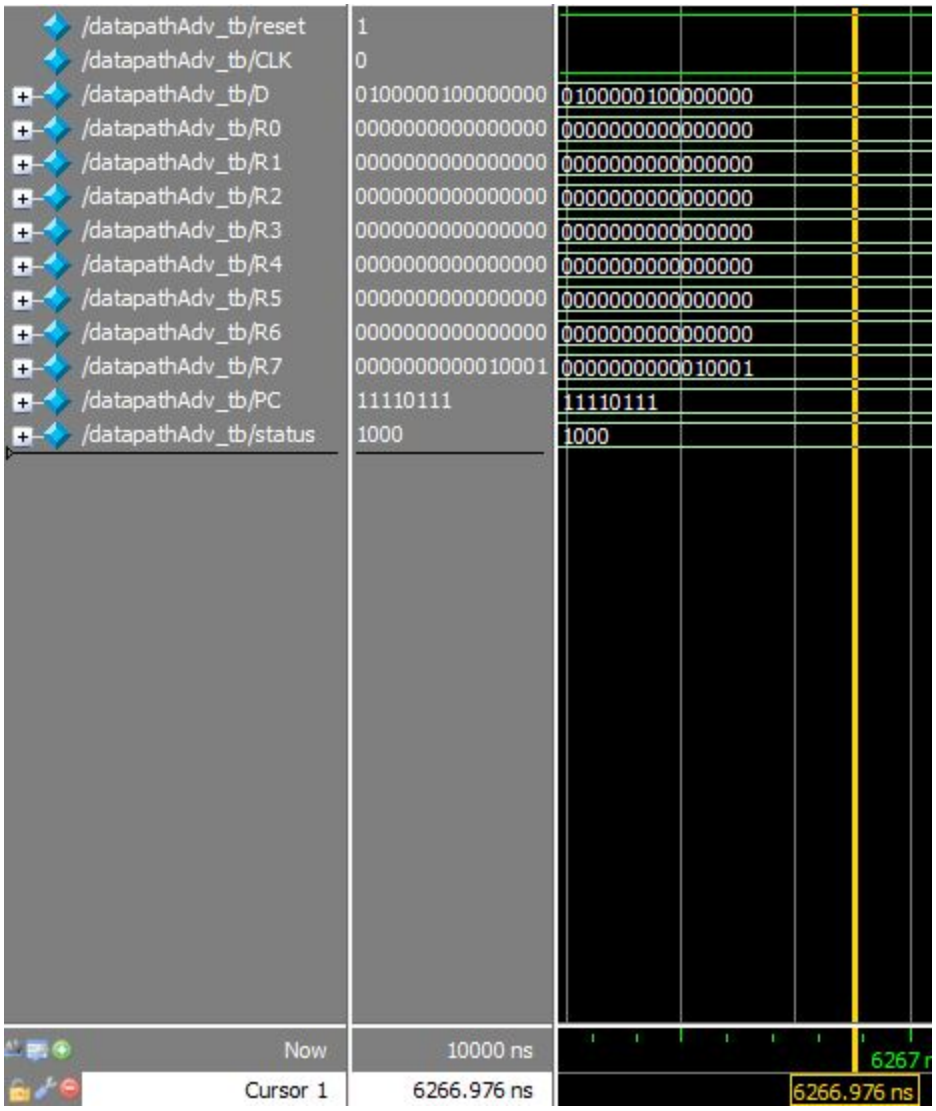


Figure 38: Instruction: CLR. R4 ← 0

See Appendix datapathAdv_tb.

CLR: 16’b0100000100000000

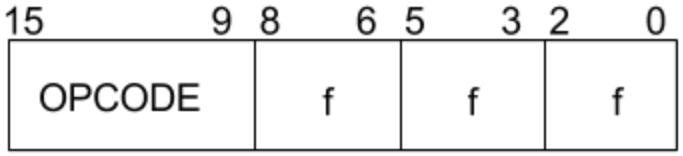
This instruction sets the value in R4 to 16’b0.

Instruction Set

Table 10: Opcode Field Descriptions

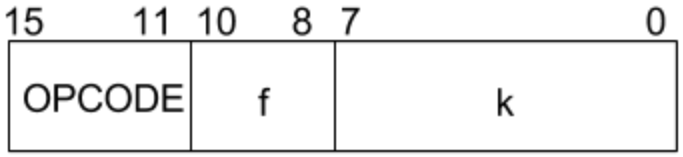
Field	Description
f	Register file address
k	Literal value
x	Don’t care
i	Index

Register Operations



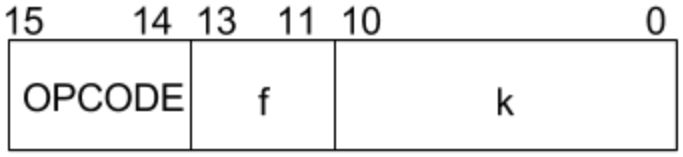
f = 3 bit address

Literal Operations



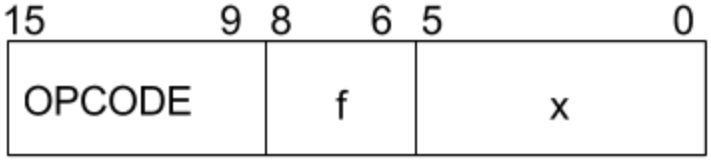
f = 3 bit address
k = 8 bit literal

LRI



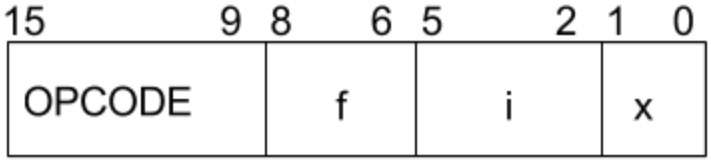
f = 3 bit address
k = 11 bit literal

LRLI-1



f = 3 bit address
x = 6 bit don't care

BSET and BCLR



f = 3 bit address
i = 4 bit index
x = 2 bit don't care

Figure 39: Instruction Formats

Table 11: Instruction Set Summary

Mnemonic	Description	Opcode	FIELDS			RTL
NOP	No Operation	0000000	3-bit D	3-bit SA	3-bit SB	No action
INC	Increment A by 1	0110000	3-bit D	3-bit SA	3-bit SB	R[DR] ← R[SA] + 1
ADD	Add A + B	0110100	3-bit D	3-bit SA	3-bit SB	R[DR] ← R[SA] + R[SB]
SUB	Subtract A - B	0110110	3-bit D	3-bit SA	3-bit SB	R[DR] ← R[SA] - R[SB]
DEC	Decrement A by 1	0110010	3-bit D	3-bit SA	3-bit SB	R[DR] ← R[SA] - 1
NEG	Negative -A	0110001	3-bit D	3-bit SA	3-bit SB	R[DR] ← ~R[SA] + 1
SHR	Shift A Right by 1	0111001	3-bit D	3-bit SA	3-bit SB	R[DR] ← sr R[SA]
SHL	Shift A Left by 1	0111000	3-bit D	3-bit SA	3-bit SB	R[DR] ← sl R[SA]
CLR	Clear A to zero	0100000	3-bit D	3-bit SA	3-bit SB	R[DR] ← 0
SET	Set all A bits to 1	0101111	3-bit D	3-bit SA	3-bit SB	R[DR] ← FFFF
NOT	Invert all A bits	0100011	3-bit D	3-bit SA	3-bit SB	R[DR] ← ~R[SA]
AND	AND A & B	0101000	3-bit D	3-bit SA	3-bit SB	R[DR] ← R[SA] ^ R[SB]
OR	OR A | B	0101110	3-bit D	3-bit SA	3-bit SB	R[DR] ← R[SA] v R[SB]
XOR	XOR A ^ B	0100110	3-bit D	3-bit SA	3-bit SB	R[DR] ← R[SA] XOR R[SB]
MOVA	Transfer A	0101100	3-bit D	3-bit SA	3-bit SB	R[DR] ← R[SA]
MOVB	Transfer B	0101010	3-bit D	3-bit SA	3-bit SB	R[DR] ← R[SB]
ADI	Add A + Literal	00001	3-bit Address	8-bit Literal		R[DR] ← R[SA] + se-lit
SBI	Subtract A - Literal	00010	3-bit Address	8-bit Literal		R[DR] ← R[SA] - se-lit
ANI	AND A & Literal	00011	3-bit Address	8-bit Literal		R[DR] ← R[SA] ^ zf-lit
ORI	OR A | Literal	00101	3-bit Address	8-bit Literal		R[DR] ← R[SA] v zf-lit
XRI	XOR A ^ Literal	00110	3-bit Address	8-bit Literal		R[DR] ← R[SA] XOR zf-lit
LRI	Load literal value to register	11	3-bit Destination	11-bit Literal		R[DR] ← se lit
LRLI-1	Load register with long immediate literal	1000010	3-bit Destination	6-bit Don't Care		R[DR] ← lit
LDI	Load data from memory at immediate address	10100	3-bit Address	8-bit Literal		R[DR] ← M[zf AD]
STI	Store data to memory at immediate address	10101	3-bit Address	8-bit Literal		M[zf AD] ← R[SA]
LDR	Load data from memory at register address	1000100	3-bit D	3-bit SA	3-bit SB	R[DR] ← M[R[SA]]
STR	Store data to memory at register address	1000101	3-bit D	3-bit SA	3-bit SB	M[R[DR]] ← R[SA]
JMPI	Jump to immediate address	1001100	9-bit Literal			PC ← zf AD
JMPR	Jump to address in register	1001101	3-bit D	3-bit SA	3-bit SB	PC ← R[SA]
BRZ	Branch to address in register if zero	10110	3-bit Address	8-bit Literal		if(R[SA] == 0) PC ← PC+se AD
BRN	Branch to address in register if negative	10111	3-bit Address	8-bit Literal		if(R[SA] < 0) PC ← PC+se AD
BCLR	Clear specified bit in register	1001000	3-bit Address	4-bit bit field index	2-bit Don't Care	R[DR].b = 0
BSET	Set specified bit in register	1001001	3-bit Address	4-bit bit field index	2-bit Don't Care	R[DR].b = 1

Table 12: Control word signals for control unit state EX0

INS	PL	AA	BA	DA	MB	FS	MD	RW	MW	JB	BC	IL	PS	K
NOP	0	X	X	X	X	X	X	0	0	0	0	0	1	X
INC	0	IR[5:3]	X	IR[8:6]	X	10000	0	1	0	X	X	0	1	X
ADD	0	IR[5:3]	IR[2:0]	IR[8:6]	0	10100	0	1	0	X	X	0	1	X
SUB	0	IR[5:3]	IR[2:0]	IR[8:6]	0	10110	0	1	0	X	X	0	1	X
DEC	0	IR[5:3]	X	IR[8:6]	X	10010	0	1	0	X	X	0	1	X
NEG	0	IR[5:3]	X	IR[8:6]	X	10001	0	1	0	X	X	0	1	X
SHR	0	IR[5:3]	X	IR[8:6]	X	11001	0	1	0	X	X	0	1	X
SHL	0	IR[5:3]	X	IR[8:6]	X	11000	0	1	0	X	X	0	1	x
CLR	0	X	X	IR[8:6]	1	X	0	1	0	X	X	0	1	16’B0
SET	0	X	X	IR[8:6]	1	X	0	1	0	X	X	0	1	16’B1
NOT	0	IR[5:3]	X	IR[8:6]	0	00011	0	1	0	x	x	0	1	x
AND	0	IR[5:3]	IR[2:0]	IR[8:6]	0	01000	0	1	0	x	x	0	1	x
OR	0	IR[5:3]	IR[2:0]	IR[8:6]	0	01110	0	1	0	x	x	0	1	x
XOR	0	IR[5:3]	IR[2:0]	IR[8:6]	0	00110	0	1	0	x	x	0	1	x
MOVA	0	IR[5:3]	X	IR[8:6]	0	01100	0	1	0	x	x	0	1	x
MOVB	0	X	IR[2:0]	IR[8:6]	0	01001	0	1	0	x	x	0	1	x
ADI	0	IR[10:8]	X	IR[10:8]	1	10100	0	1	0	X	X	0	1	IR[7:0]
SBI	0	IR[10:8]	X	IR[10:8]	1	10110	0	1	0	X	X	0	1	IR[7:0]
ANI	0	IR[1	X	IR[1	1	010	0	1	0	0	0	0	1	IR[7:

		0:8[		0:8]		00							0]
ORI	0	IR[10:8]	X	IR[10:8]	1	01110	0	1	0	0	0	0	1	IR[7:0]
XRI	0	IR[10:8]	X	IR[10:8]	1	00110	0	1	0	X	X	0	1	IR[7:0]
LRI	0	X	X	IR[13:11]	1	11111	0	1	0	X	X	0	1	IR[10:0]
LRLI-1	0	x	x	x	x	x	x	x	x	x	x	x	1	x
LDI	0	IR[10:8]	IR[10:8]	IR[10:8]	1	X	1	1	0	X	X	0	1	IR[7:0]
STI	0	IR[10:8]	IR[10:8	IR[10:8]	1	X	1	0	1	X	X	0	1	IR[7:0]
LDR	0	IR[5:3]	X	IR[8:6]	X	X	1	1	0	X	X	0	1	X
STR	0	IR[8:6]	IR[5:3]	X	X	X	X	0	1	X	X	0	1	X
JMPI	1	X	X	X	1	11111	0	0	0	1	0	0	1	IR[8:0]
JMPR	1	IR[5:3]	X	IR[5:3]	0	01100	0	0	0	1	0	0	1	X
BRZ	1	IR[10:8]	X	x	x	x	x	0	0	0	0	0	1	IR[7:0]
BRN	1	IR[10:8]	X	x	x	x	x	0	0	0	0	0	1	IR[7:0]
BCLR	0	IR[8:6]	X	IR[8:6]	1	01000	0	1	0	0	0	0	1	IR[5:2]
BSET	0	IR[8:6[	X	IR[8:6]	1	01110	0	1	0	0	0	0	1	IR[5:2]

Table 13: Control word signals for control unit state EX1

INS	PL	AA	BA	DA	MB	FS	MD	RW	MW	JB	BC	IL	PS	K
BRZ	1	IR[10:8]	X	X	X	X	X	0	0	0	0	0	1	X
BRN	1	IR[10:8]	X	X	X	X	X	0	0	0	0	0	1	X

These instructions have been implemented into the control unit and can be accessed using the opcodes and address fields shown in table 10. Most instructions require a 3 bit destination address, a 3 bit SA and a 3 bit SB. However, many instructions requiring a literal use the SA and SB fields to define that literal.

Advanced Instructions

The processor discussed in this datasheet supports an advanced Multiply instruction:

Table 14: Instruction for the multiply advanced operation

Mnemonic	Description	Opcode		Fields		RTL
MUL	Multiply A times B	0110111	3-bit D	3-bit SA	3-bit SA	R[DR] ← R[SA] * R[SB]

The multiply instruction is implemented as an additional block extension to the ALU. Data inputs to the ALU are also inputs to the multiply block via the same bus line. The output of the ALU is input to the advanced multiply block. If the instruction is the multiply instruction, the block will multiply the A and B inputs and output. Inversely, if the instruction is not a multiply instruction the multiply block works as a multiplexer and allows the output of the ALU to be the output.

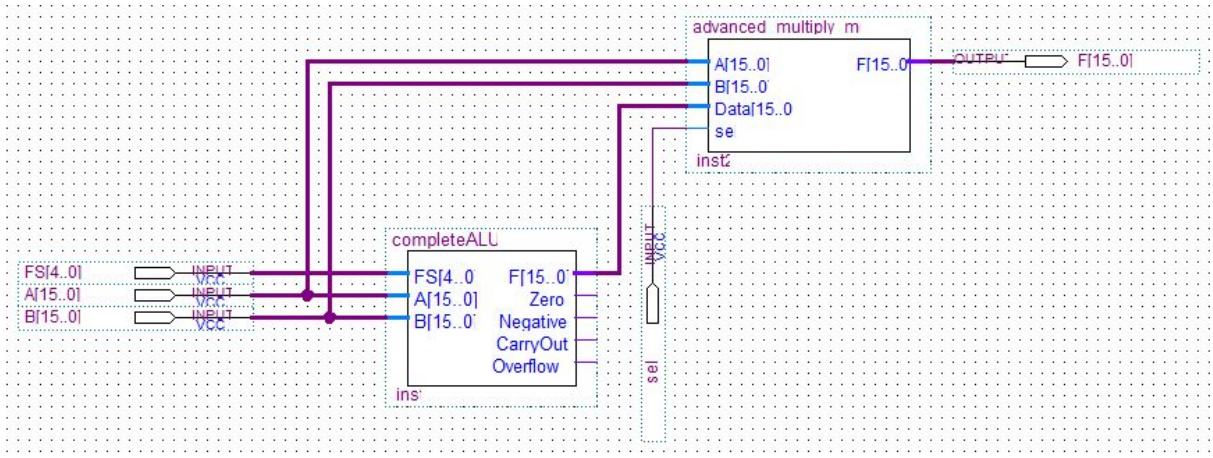


Figure 40: Block diagram of the multiply instruction implemented in the datapath

Example Programs

Matt Mammarelli

Table 15: Example Program Demonstrating CPU Operations

Assembly	Binary
NOP	16’b0
INC R7,R7	16'b0110000111111100
INC R7,R7	16'b0110000111111100
INC R7,R7	16'b0110000111111100
INC R7,R7	16'b0110000111111100
INC R7,R7	16'b0110000111111100
LRI R4, 13	16'b1110000000001011
INC R7,R7	16'b0110000111111100
INC R7,R7	16'b0110000111111100
STI R4, 3	16'b1010110000000011
INC R7,R7	16'b0110000111111100
INC R7,R7	16'b0110000111111100
LDI R2, 3	16'b1010001000000011
BSET R4.15	16'b1001001100111100
BCLR R4.0	16'b1001000100000000

My program will increment R7 multiple times. Then it will load a literal value into R4.Then it will increment R7. Then it will store the value in R4 at the immediate address of 3 in memory. It will then increment R7. It will then load R2 from memory at the immediate address of 3. It will then set the most significant bit of R4 to be 1. It will then set the least significant bit of R4 to be 0. This example program demonstrates simple instructions as well as the two intermediate instructions of BSET and BCLR.

Stephen Glass

Table 16: Example program demonstrating CPU operations (Stephen Glass)

Assembly	Binary
LRI R0, 5	16’b1100000000000101
LRI R1, 2	16’b1100100000000010
LRI R2, 10	16’b1101000000001010
BSET R0.0010	16’b1001001000001000
BCLR R2.0010	16’b1001000010110100
MUL R5, R1, R2	16’b0110111101001010

The program will begin by loading immediate values 5, 2, 10 into register values R0, R1, R2 respectively. BSET is demonstrated by setting the second bit of the value stored in R0 to one. BCLR is demonstrated by setting the second bit of value stored in R2 to 0. Finally, the multiply instruction (MUL) is demonstrated by multiplying the values of R1 and R2 into R5.

Joe Foderaro

Table 17: Example program demonstrating CPU operations (Joe Foderaro)

Assembly	Binary
NOP	16’b0
LRI R0, 67	16'b111000001000011
INC R1,R1	16'b0110000001001000
SHL R1,R1	16'b0111000001001000
SHL R1,R1	16'b0111000001001000
MUL R2,R0,R1	16'b0110111010000001
SHR R2,R2	16'b0111001010010000
SHR R2,R2	16'b0111001010010000
SHR R2,R2	16'b0111001010010000
SHR R2,R2	16'b0111001010010000
SHR R2,R2	16'b0111001010010000

BCLR R2.1000	16'b1001000100000000

This program loads the value 67 into R0, increments the value (0) in R1, and shifts that left twice so that the value in R1 is 4. The program then multiplies 4 and 67 giving you 268 which is 100001100 and this is stored into R2. It then shifts this value to the right 5 times leaving 1000 in R2. Finally bit 3 is cleared and R2 is left with 0000 stored in it.

Performance

Maximum Frequency achieved without error: 50MHz.
The performance of our processor could be first improved starting with redesigning the ALU. If a carry lookahead adder design was used instead of the ripple carry adder design it would perform much faster. Pipelining this processor would also improve performance as multiple instructions could be executing one clock cycle after another without delay. Other components like the Register File, the Program Counter, and the Control Unit are already as optimized so focusing on improving these in the future wouldn't improve the speed much. Our instruction set is also small so this wouldn't be impacting the performance much.

As discussed in the errata section the performance can also be improved by combining the instruction fetch in the control unit with the program select from the program counter. The program counter can be connected to the enable pin of the instruction register directly to load a new instruction when the previous cycle has completed.

Errata

Stephen Glass
Table 18: Errata (problems or improvement) (Stephen Glass)

Problem (or improvement)	Cause	Solution
Overclocking demonstration displayed "DCE!" instead of the expected "NICE!"	The final jump register (JMPR) instruction (h42) was not implemented correctly in the control unit.	Correcting the control word in the control unit for the jump register instruction improved the output of the overclocking to "DICE!" as opposed to "DCE!"
Improvement: Harvard architecture took one clock cycle longer for each instruction	The instruction fetch block in the control unit topic level is unnecessary in the Harvard architecture as the program counter hold signal can be used to load the next instruction.	Remove the instruction fetch from the control unit and connect the program counter select (PS) signal directly to the enable pin of the instruction register.

Appendix

Register File 4 to 16 Decoder

```
1   module Decoder4to16_2 (A,F);
2
3   input [3:0] A;
4   output reg [15:0] F;
5
6
7   always @(A)
8   begin
9   F=0;
10     case (A)
11     4'b0000: F=16'b0000000000000001;
12     4'b0001: F=16'b0000000000000010;
13     4'b0010: F=16'b0000000000000100;
14     4'b0011: F=16'b0000000000001000;
15     4'b0100: F=16'b0000000000010000;
16     4'b0101: F=16'b0000000000100000;
17     4'b0110: F=16'b0000000001000000;
18     4'b0111: F=16'b0000000010000000;
19     4'b1000: F=16'b0000000100000000;
20     4'b1001: F=16'b0000001000000000;
21     4'b1010: F=16'b0000010000000000;
22     4'b1011: F=16'b0000100000000000;
23     4'b1100: F=16'b0001000000000000;
24     4'b1101: F=16'b0010000000000000;
25     4'b1110: F=16'b0100000000000000;
26     4'b1111: F=16'b1000000000000000;
27
28     endcase
29
30     end
31
32
33  endmodule
34
35
```

Register File 16 to 1 Multiplexer

```
1   module mux_16_to_1 (A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,sel);
2
3      input [15:0] A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P;
4      input [3:0] sel;
5      output [15:0]Q;
6      reg [15:0]Q;
7
8      always @ (A or B or C or D or E or F or G or H or I or J or K or L or M or N or O or
   P or Q or sel)
9
10     case (sel)
11     4'b0000: Q=A;
12     4'b0001: Q=B;
13     4'b0010: Q=C;
14     4'b0011: Q=D;
15     4'b0100: Q=E;
16     4'b0101: Q=F;
17     4'b0110: Q=G;
18     4'b0111: Q=H;
19     4'b1000: Q=I;
20     4'b1001: Q=J;
21     4'b1010: Q=K;
22     4'b1011: Q=L;
23     4'b1100: Q=M;
24     4'b1101: Q=N;
25     4'b1110: Q=O;
26     4'b1111: Q=P;
27
28     endcase
29
30
31
32
33  endmodule
34
```

Register File Testbench

```
1   //Matt Mammarelli
2   // Modelsim-ASE requires a timescale directive
3   `timescale 1 ns / 1 ns
4
5   module registerFile_tb();
6
7   reg WR, CLK, reset;
8   reg [3:0] DA, AA, BA;
9   reg [15:0] D;
10  wire [15:0] A, B, R0, R1, R2, R3, R4, R5, R6, R7, T0, T1, T2, T3, T4, T5, T6, T7;
11  reg [3:0] address;
12
13  RegisterFile dut (.WR(WR),.CLK(CLK),.reset(reset),.DA(DA),.AA(AA),.BA(BA),.D(D),.A(A),.B
    (B),.R0(R0),.R1(R1),.R2(R2),.R3(R3),.R4(R4),.R5(R5),.R6(R6),.R7(R7),.T0(T0),.T1(T1),.T2(
    T2),.T3(T3),.T4(T4),.T5(T5),.T6(T6),.T7(T7));
14
15
16  initial begin
17        CLK <= $random(50);
18        reset <= 1'b0;
19        D <= 16'b1001010101110001;
20        $monitor("hellow");
21        $monitor("CLK = %b", CLK);
22        #0 WR <= 1'b1;
23
24        #0 address <= 4'b0000;
25        #300 address <= 4'b0001;
26        #300 address <= 4'b0010;
27        #300 address <= 4'b0011;
28        #300 address <= 4'b0100;
29        #300 address <= 4'b0101;
30        #300 address <= 4'b0110;
31        #300 address <= 4'b0111;
32        #300 address <= 4'b1000;
33        #300 address <= 4'b1001;
34        #300 address <= 4'b1010;
35        #300 address <= 4'b1011;
36        #300 address <= 4'b1100;
37        #300 address <= 4'b1101;
38        #300 address <= 4'b1110;
39        #300 address <= 4'b1111;
40
41
42
43
44  end
45
46  always
47        #10 CLK <= ~ CLK; //clock changes every 10 ns
48
49  always @ (posedge CLK)
50  begin
51     DA <= address;
52     AA <= address;
53     BA <= address;
54     $monitor("Address = %b", address); //monitor only prints when there is a change,
    display always prints every clock cycle
55     $display("R0 = %b", R0);
56     $display("R1 = %b", R1);
57     $display("R2 = %b", R2);
```

```
58      $display("R3 = %b", R3);
59      $display("R4 = %b", R4);
60      $display("R5 = %b", R5);
61      $display("R6 = %b", R6);
62      $display("R7 = %b", R7);
63      $display("T0 = %b", T0);
64      $display("T1 = %b", T1);
65      $display("T2 = %b", T2);
66      $display("T3 = %b", T3);
67      $display("T4 = %b", T4);
68      $display("T5 = %b", T5);
69      $display("T6 = %b", T6);
70      $display("T7 = %b", T7);
71
72  end
73
74
75
76  initial
77        #5000 $stop;
78
79
80
81  endmodule
82
```

ALU Override

```
1   module Override(FS,Ain,Bin,F,sel);
2   input [4:0] FS;
3   input [15:0] Ain,Bin;
4   output reg [15:0] F;
5   output reg sel;
6
7
8
9
10
11  always @(Ain or Bin or FS)
12  begin
13  sel = 1'b0;
14
15  case(FS)
16  5'b10000: begin
17  F = Ain+1'b1;
18  sel = 1'b1;
19
20  end
21  5'b10110: begin
22  F = Ain-Bin;
23  sel = 1'b1;
24
25  end
26  5'b11111: begin
27  F = Bin;
28  sel = 1'b1;
29
30  end
31
32
33  endcase
34
35  end
36
37  endmodule
38
```

ALU Testbench

```
1   //Matt Mammarelli
2   // Modelsim-ASE requires a timescale directive
3   `timescale 1 ns / 1 ns
4
5   module ALU_tb();
6
7   //inputs
8   reg [15:0] A, B;
9   reg [4:0] FS;
10
11  //outputs
12  wire C, Z, N, V;
13  wire [15:0] F;
14
15
16  ALU dut (.A(A),.B(B),.FS(FS),.C(C),.Z(Z),.N(N),.V(V),.F(F));
17
18  //reg A,B;
19  //reg C_in;
20  //reg A_from_next_bit;
21  //wire F;
22  //wire C_out;
23
24  //ALUCell dut (.A(A),.B(B),.C_in(C_in),.A_from_next_bit(A_from_next_bit),
    .FS(FS),.F(F),.C_out(C_out));
25
26  initial begin
27
28        $monitor("hellowww");
29
30
31        A <= 16'b0000000000000110;
32        B <= 16'b0000000000000101;
33
34        //A<=1'b0;
35        //B<=1'b1;
36        //C_in <= 1'b1;
37        //A_from_next_bit <= 1'b0;
38
39        //selects the function you want to perform
40
41        #0 FS <= 5'b10100; //A+B
42        #100 FS <= 5'b10010; //A-1
43        #100 FS <= 5'b10001; //-A
44        #100 FS <= 5'b00000; //0
45        #100 FS <= 5'b01100; //A
46        #100 FS <= 5'b00011; //~A
47        #100 FS <= 5'b01000; //A&B
48        #100 FS <= 5'b01110; //A|B
49        #100 FS <= 5'b00110; //A^B
50        #100 FS <= 5'b11001; //A>>1
51        #100 FS <= 5'b11000; //A<<1
52        #100 FS <=5'b10000;  //A+1
53        #100 FS <= 5'b10110; //A-B
54
55
56  end
57
58  always @(F or C or Z or N or V or FS or A or B) begin
59        $display("FS = %b", FS);
```

```
60          $display("A = %b", A);
61          $display("B = %b", B);
62          $display("Answer = %b", F);
63          $display("Carry Unsigned = %b", C);
64          $display("Zero = %b", Z);
65          $display("Negative = %b", N);
66          $display("Overflow Signed = %b", V);
67          $display("");
68
69    end
70
71
72
73
74    initial
75          #1500 $stop;
76
77
78
79
80
81
82
83    endmodule
84
```

Control Unit Testbench

```
1   //Matt Mammarelli
2   // Modelsim-ASE requires a timescale directive
3   `timescale 1 ns / 1 ns
4
5   module ControlUnit_tb();
6
7   reg    reset;
8   reg    clock;
9   reg    [15:0] IR_in;
10  wire   [3:0] status_in;
11  wire   [38:0] ControlWord;
12
13  ControlUnit dut (.reset(reset),.clock(clock),.IR_in(IR_in),.status_in(status_in),.
    ControlWord(ControlWord));
14
15  initial
16  begin
17
18  reset<=1'b1;
19  clock<=1'b0;
20
21  IR_in <= 16'b0110100111111100 ; //ADD
22  //IR_in<=16'b1110110000000000;
23
24  #10 clock<=1'b1;
25  #200 clock<=1'b0;
26  #200 clock<=1'b1;
27  #200 clock<=1'b0;
28  #200 clock<=1'b1;
29
30
31
32
33  end
34
35  initial
36  #1500 $stop;
37
38
39  always @(ControlWord or IR_in)
40  begin
41  $display("IR_in = %b", IR_in);
42  $display("CW = %b", ControlWord);
43
44  end
45
46  endmodule
```

CPU Testbench

```
1   //Matt Mammarelli
2   // Modelsim-ASE requires a timescale directive
3   `timescale 1 ns / 1 ns
4
5   module datapathAdv_tb();
6
7   reg    reset;
8   reg    CLK;
9   reg    [15:0] D;
10  wire   [15:0] R0;
11  wire   [15:0] R1;
12  wire   [15:0] R2;
13  wire   [15:0] R3;
14  wire   [15:0] R4;
15  wire   [15:0] R5;
16  wire   [15:0] R6;
17  wire   [15:0] R7;
18  wire [7:0] PC;
19  wire [3:0] status;
20
21  DatapathAdv dut (.reset(reset),.CLK(CLK),.D(D),.PC(PC),.status(status),.R0(R0),.R1(R1),.
    R2(R2),.R3(R3),.R4(R4),.R5(R5),.R6(R6),.R7(R7));
22
23  initial begin
24
25  CLK <= 1;
26  reset <= 1'b1;
27  D <= 16'b0110000111111100 ; //INC
28  #600 D <= 16'b1000101001111000 ; //STR  set M address 001 to R7
29  #400 D <= 16'b0110000111111100 ; //INC
30  #1000 D <= 16'b1000100100001000 ; //LDR  set r4 to M address 001
31  #400 D <= 16'b0110000111111100 ; //INC
32  #1000 D <= 16'b0; //NOP
33  #400 D <= 16'b1001101000100000 ; //JMPR to R4 = 11
34  #400 D <= 16'b0110000111111100 ; //INC
35  #700 D <= 16'b1011000000000111 ; //BRZ
36  #200 D <= 16'b0; //NOP
37  #700 D <= 16'b0100000100000000 ; //CLR
38  end
39
40  always
41        #50 CLK <= ~ CLK; //clock changes every 10 ns
42
43
44
45
46  initial
47  #10000 $stop;
48
49
50  endmodule
51
```