team **BOES**

Computer Architecture
ECE 09243 | Summer 2021

Professor Muhlbaier

Jeremy Beal
Michael Onyeije
Ali Elhamawi

BOES-16JAMB Processor Datasheet

A 16-bit Processor

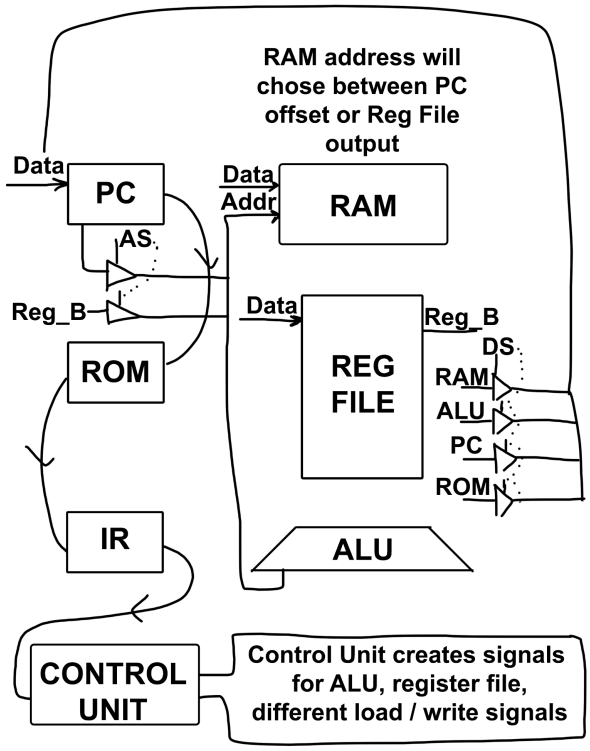
Section 1: The Overview

The BOES-16JAMB is a 16-bit processor based on Harvard Architecture, meaning that it makes use of two separate memory spaces for program instructions and data. There are a few advantages that come with this. A processor's operating bandwidth can improve, and a designer can incorporate different bus sizes between the two forms of memory. Our processor uses two separate memory components, a ROM (read-only memory) and a RAM (random access memory). These two forms of memory will be described in more detail in the **MEMORY SECTION.** The BOES-16JAMB uses a register file with 8 registers, all (at least) 16-bits. Data can be written from a desired register, or read from two desired registers at a time. The ALU in our processor is capable of doing common arithmetic and logic instructions along with logical shifts. Both the ALU and register file handle (at least) 16-bit input. More individualized information on each group member's processor can be found **HERE.** These sections will shed more light on specific memory sizes, architecture implementation, peripherals, and clock speed.

Table of Contents

Section 2: Architecture Overview

Figure 2-1: Simple Block Diagram (Jeremy)



This is a very simple version of my finished datapath. Data flows into the program counter, the program counter value flows into the separate ROM which passes instructions to the instruction register. The instruction register will load instructions into the control unit which will generate a control word that tells everything else what to do. More information about this process can be found in the **datapath section.** However, from an architecture standpoint, there are indeed two separate memory units, and the bulk of the selection happens using tri-state buffers (as seen with the AS and DS signals in the above figure).

Section 3: List of Figures

Architecture Overview

The Register File

The ALU

The Memory

The Datapath

The Control Unit

The CPU

The Peripherals

Instruction Set Architecture

Example Program

Performance

Personal Enhancements and Features (Jeremy)

Appendix

Section 4: List of Tables

The Register File

The ALU

The Datapath

Instruction Set Architecture

Section 5: The Register File

Figure 5-1: 8x16 Register File

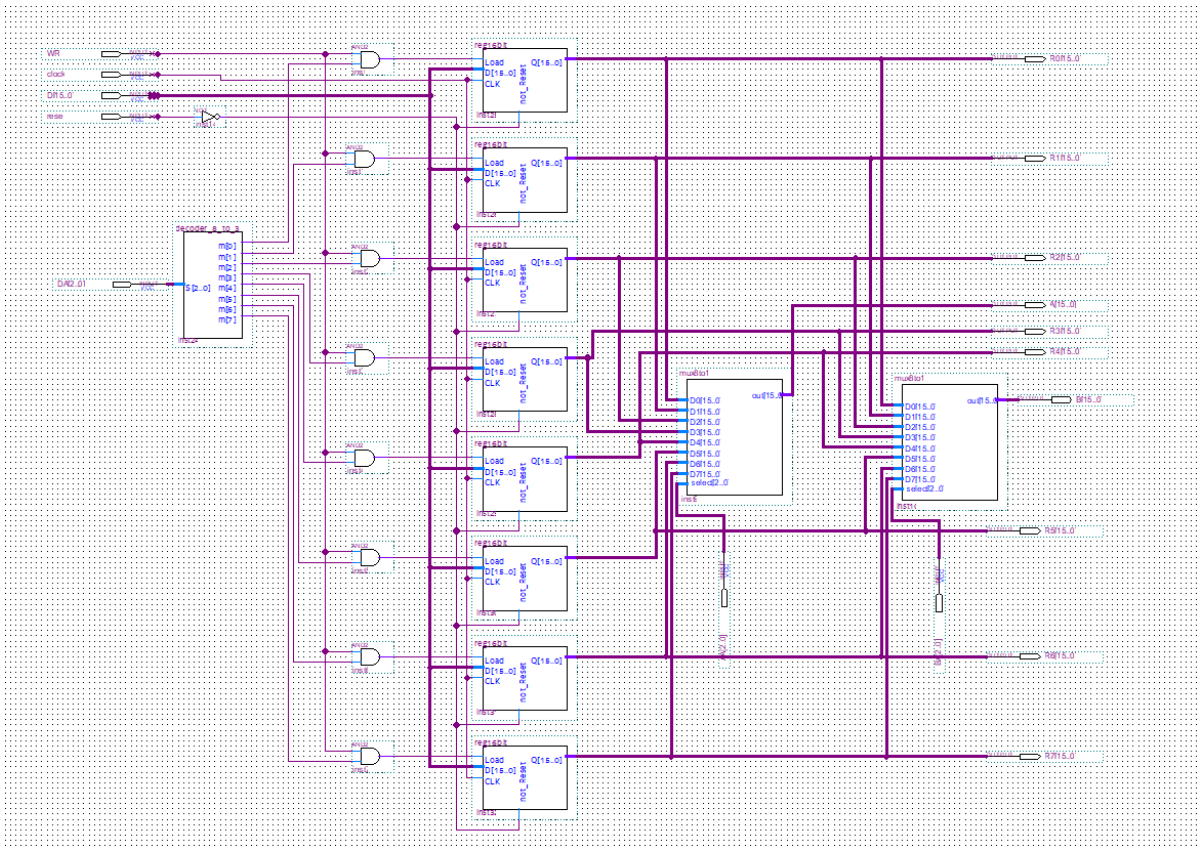
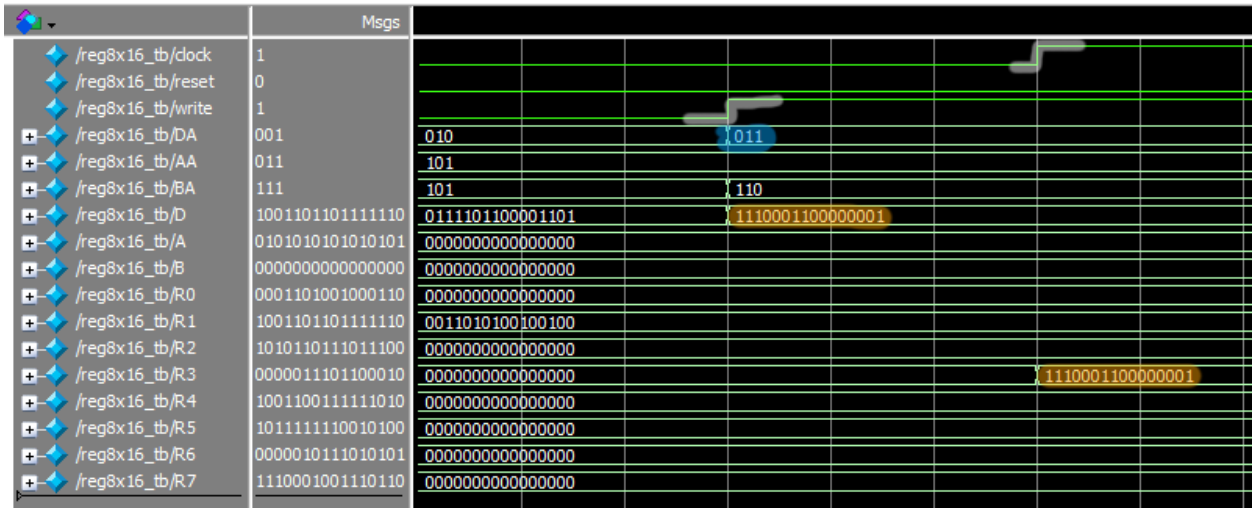


Figure 1-1 shows an 8x16 register file, 8 registers 16 bits each. There is a 4 bit destination select that chooses which register to load data into (this is sent through a 3-to-8 decoder). It has a write input which will load data when write is 1, and does not load data when write is 0. There are also 2, 16 bit 8-to-1 multiplexers that select the registers to connect to the outputs. The write signal and decoder outputs are sent through 8 AND gates before going into each of the registers.

Figure 5-2: Test bench of the Register File



Here, we can prove that the register file is working properly by looking at the wave diagram in ModelSim. Write is 1, meaning that data will be loaded. The data to be loaded is located on the D input (in orange) Specifically, it will be loaded onto register 011 based on our destination address (in blue). On the next clock edge, we get that.

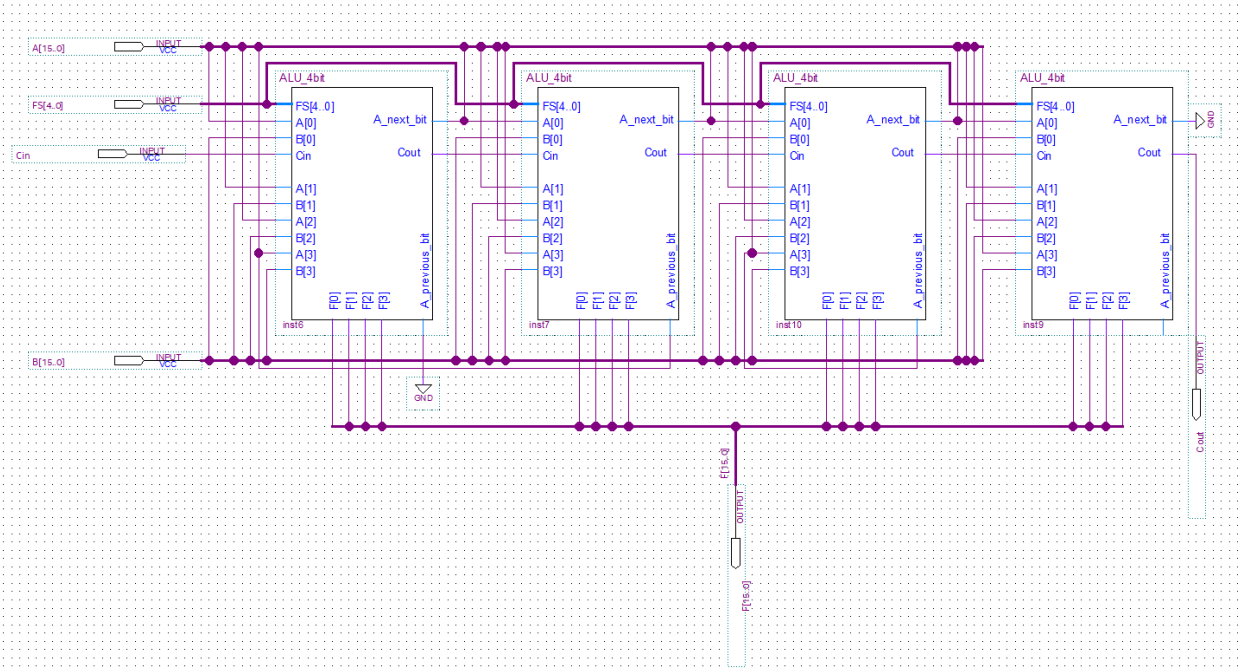
Table 5-1: Inputs and Outputs of Register File

Inputs / Outputs	Description
D	16-bit data input that loads into the registers.
DA	3 / 4 bit address that selects register to load into.
A	16-bit output from selected register.
AA	3 / 4 bit output to select A for output.
B	16-bit output from selected register.
BA	3 / 4 bit output to select B for output.
WR	Write a signal to enable load.
clock	Input to the clock of all registers.
reset	Resets all registers to zero.
R0 - R7/R15	Direct 16-bit register outputs.

Section 6: The ALU

The chosen ALU is constructed, essentially, of 16 1-bit ALU's. A 4-bit symbol file was created using 4 1-bit ALU's which made it relatively easy to implement a 16-bit version. This ALU can perform arithmetic operations (add / subtract etc.), logical operations (AND / OR / NOT etc.), and shifts to the right or left. A carry-look-ahead adder was implemented into this design for gate delay optimization. Schematics of the individual cells can be found in the **APPENDIX.**

Figure 6-1: 16-bit ALU from 4, 4-bit ALU's



Also included in this section is a very useful function select (FS) table complete with correct C_0 values that describes each function select input and what operation it executes.
Some values are omitted due to an obscure output that wouldn't necessarily be used.

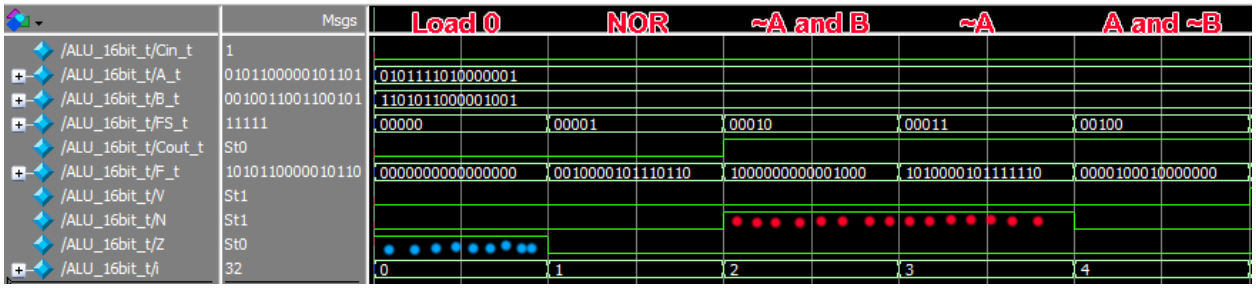
Table 6-1: Function Select Table

FS[4]	FS[3]	FS[2]	FS[1]	FS[0]	C_0	Operation
0	0	0	0	0	x	F = 0
0	0	0	0	1	x	F= A ~| B (NOR)
0	0	0	1	0	x	F = (~A) & B
0	0	0	1	1	x	F = ~A
0	0	1	0	0	x	F = A & (~B)
0	0	1	0	1	x	F = ~B
0	0	1	1	0	x	F = A ^ B (XOR)
0	0	1	1	1	x	F = A ~& B (NAND)
0	1	0	0	0	x	F = A & B
0	1	0	0	1	x	F = A ~^ B (not XOR)
0	1	0	1	0	x	F = B
0	1	0	1	1	x	Omitted
0	1	1	0	0	x	F = A
0	1	1	0	1	x	Omitted
0	1	1	1	0	x	F = A | B
0	1	1	1	1	x	F = 1'b1
1	0	0	0	0	0	F = A
1	0	0	0	0	1	F = A + 1
1	0	0	0	1	0	F = ~A
1	0	0	0	1	1	F = -A
1	0	0	1	0	0	F = A + 1
1	0	0	1	0	1	F = A + 2
1	0	0	1	1	0	F = -A
1	0	0	1	1	1	F = 1-A

1	0	1	0	0	0	F = A + B
1	0	1	0	0	1	F = A + B + 1
1	0	1	0	1	0	F = ~A + B
1	0	0	1	1	1	F = B - A
1	0	1	1	0	0	F = A + ~B
1	0	1	1	0	1	F = A - B
1	0	1	1	1	0	F = ~A + ~B
1	0	1	1	1	1	F =Omitted
1	1	0	0	0	0	F = A << 0
1	1	0	0	0	1	F = A << 1
1	1	0	0	1	0	F = A >> 1
1	1	0	0	1	1	F = A >>> 1

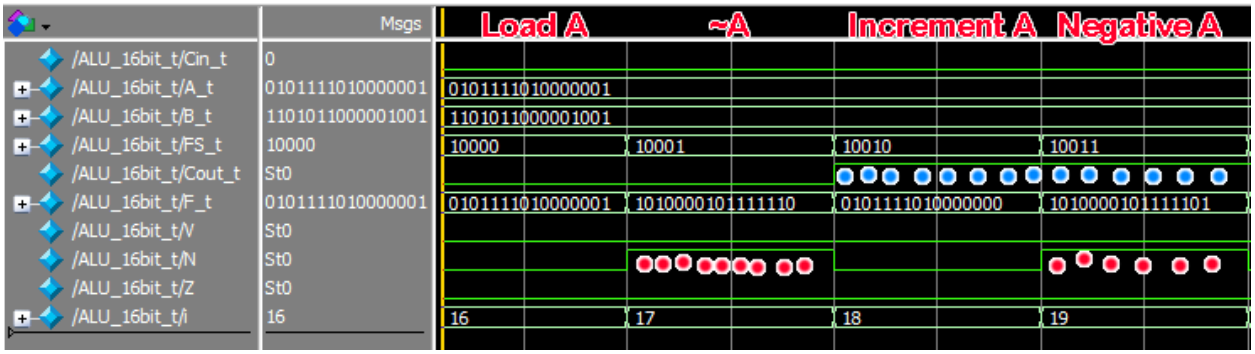
The remaining bits will simply perform the previous three shift operations, they will each repeat three times (until 11111) and were omitted from the table. Also, the omitted bits still have a function, that said function was just deemed unnecessary for showcasing.

Figure 6-2: Testbench of ALU (logic)



The figure above shows a testbench of the ALU, specifically the initial logic functions. The functions being performed are above in red and are working as expected. The blue dots represent the zero flag at work and the reg dots represent the negative flag (along with Cout).

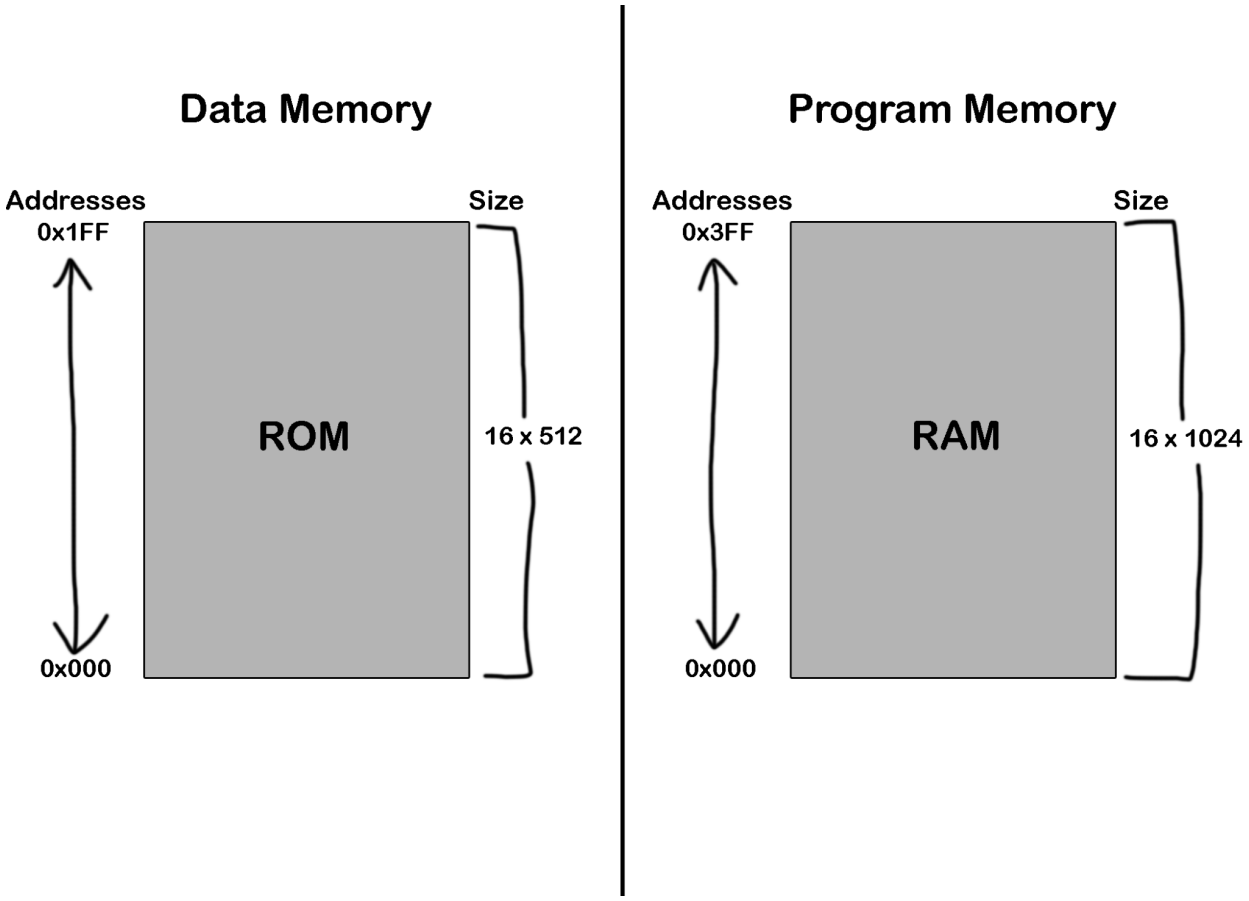
Figure 6-3: Testbench of ALU (arithmetic)



This snippet of the testbench shows some of the arithmetic operations (switched every 10 ticks) as listed at the top in red. Again everything seems to be working properly and the flags changes are shown with blue and red dots.

Section 7: The Memory

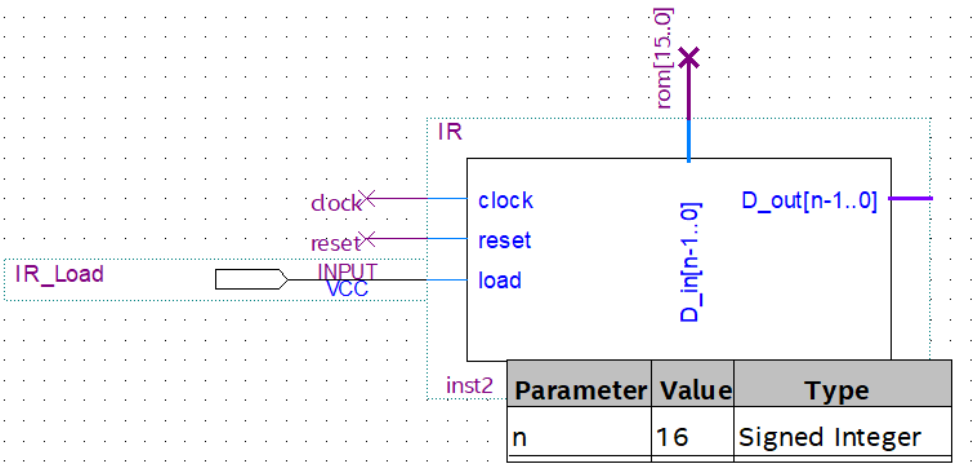
Figure 7-1: Memory Layout



Our processor is equipped with individual ROM and RAM modules. The size of the actual memory will vary depending on the group member as each member was required to have a slightly different size, this given size is just for figure purposes. At this size the RAM spans from 0x000 to 0x3FF while the ROM caps out at 0x1FF. The RAM accepts 10-bit addresses (variable) and 16-bit data, and has a write enable input. It outputs 16-bit data that is fed into a tri-state buffer for selecting purposes. The ROM has an 8-bit address input (variable) with a 16-bit output that also goes directly to a tri-state buffer. Advanced features and modifications can be found in some individual sections **HERE.**

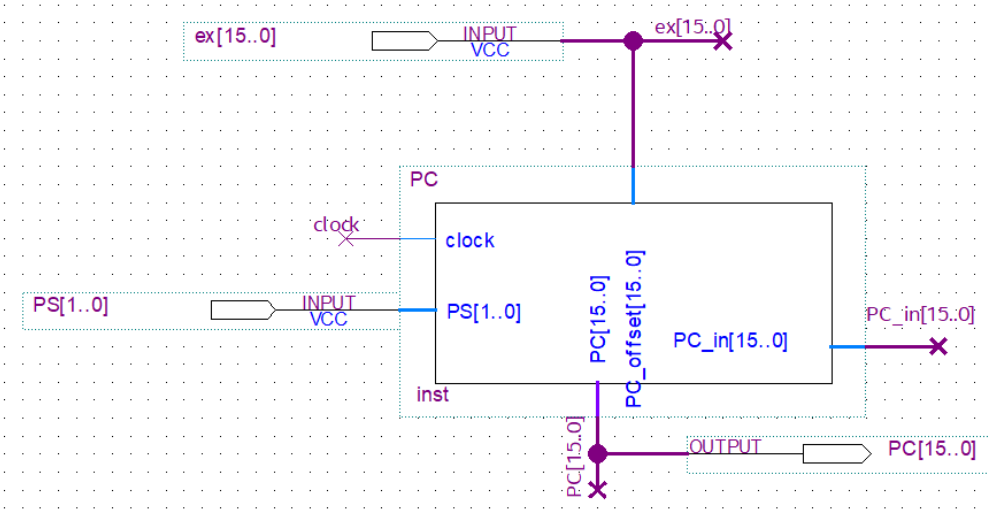
Section 8: The Datapath

Figure 8-1: Instruction Register Symbol



The instruction register is essentially a normal register with an IR_load input which when enabled will allow the IR to load instructions. The instruction register feeds directly into the control unit which will output specific control word values based on the bits of the IR output. The Verilog code for the instruction register can be found in the **APPENDIX**.

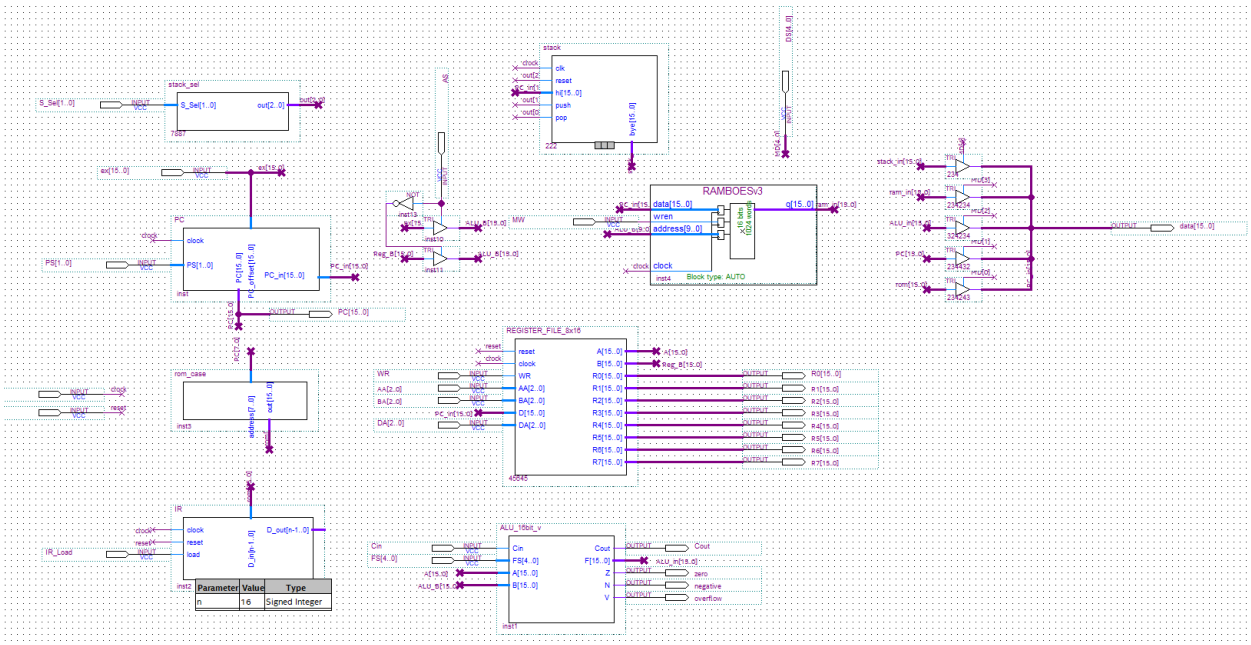
Figure 8-2: Program Counter Symbol



The program counter chosen in the design has a 2-bit selector, at 00 the program counter does nothing, 01 causes the program counter to increment by 1, while 11 will increment the program counter by a literal value (named ex). That literal value feeds into the PC_offset output and a tri-state buffer can select the PC output of the program counter (although not

used often). The PC_in output will feed into the different components of the datapath (RAM, ALU, Register File). The Verilog code for the program counter can be found in the **APPENDIX.**

Figure 8-3: Datapath using Tri-state Buffers



Note: An annotated version of this datapath can be found in the **APPENDIX.**

Figure 8-4: Datapath using Muxes

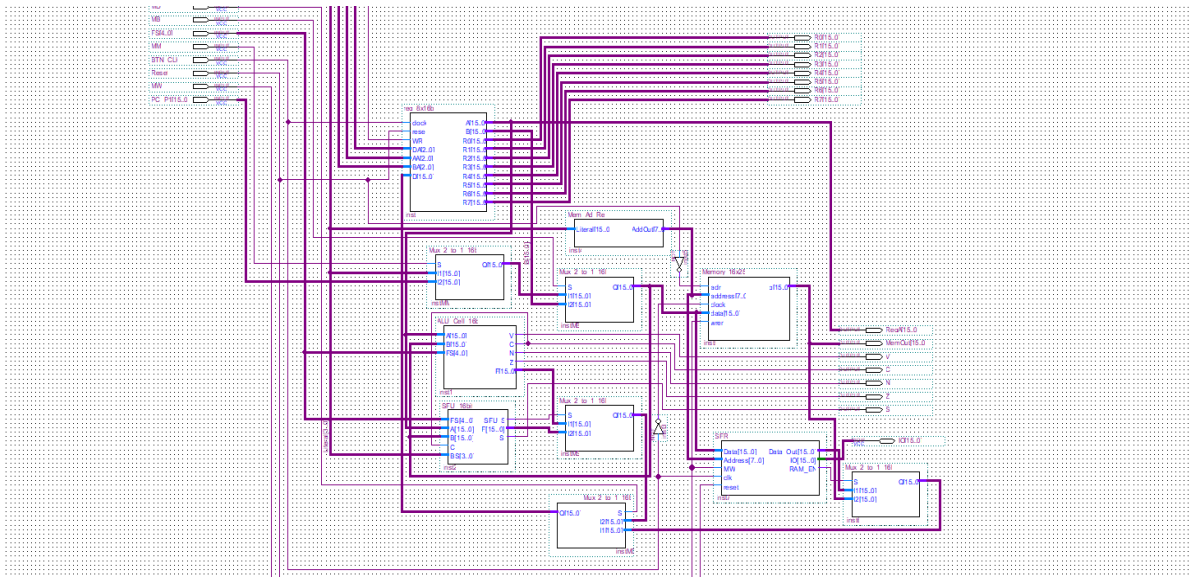


Figure 8-5: Control Word

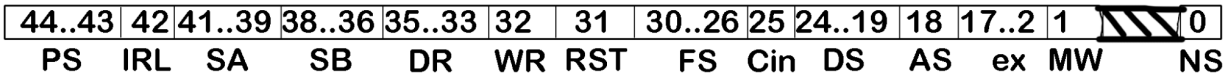


Figure 5-1: Table of Signals

PS[45:44]	Program Counter Selector, selects what the program counter is doing
IRL[43]	Instruction Register Load, is enabled the instruction register will load instructions
SA[42:40]	Source A from the register file, will be assigned to a register in the file
SB[39:37]	Source B from the register file, will be assigned to a register in the file
DR[36:34]	Destination Register from the register file, will be assigned to a register in the file
WR[33]	Write Register, when enabled data will be written onto a register
RST[32]	Reset, clears registers
FS[31:27]	Function Select, controls the ALU and what operations it will execute
Cin[26]	Cin value, paired with the ALU, will perform different operations based on the value of Cin (in some cases)
DS[25:21]	Data Bus Selector, selects the source of data to be loaded onto the data bus. DS[0] correlates to ROM, DS[1] to PC, DS[2] to the ALU, DS[3] to the RAM, DA[4] to the stack
AS[20]	Address Selector, selects either the literal value or B output of register
ex[19:4]	Literal Value, 16 bits, used as an address or immediate value for different instructions
MW[3]	Write Memory, when enabled memory will be written to
SS[2:1]	Stack Selector, selects between the inputs of the hardware stack
NS[0]	Next State

In reference to figure 4-3, this datapath was designed using tri-state buffers. Tri-state buffers select the output source that is loaded onto the data bus (DS) and two more tri-state buffers select between the literal value and B output of the register file. From a design standpoint, the components were connected by wire name to make everything look cleaner (an annotated version is included to help see what goes where). The datapath is of Harvard Architecture, there are separate buses for memory data addresses, along with two separate blocks for ROM and RAM. More benefits of this architecture will come in the control unit section.

Section 9: The Control Unit

Figure 9-1: Control Unit Final Schematic (1 of 2)

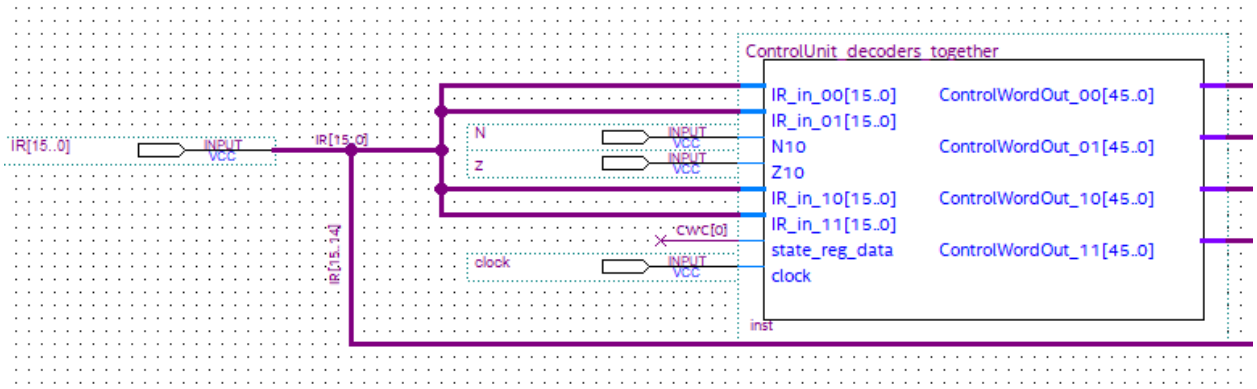
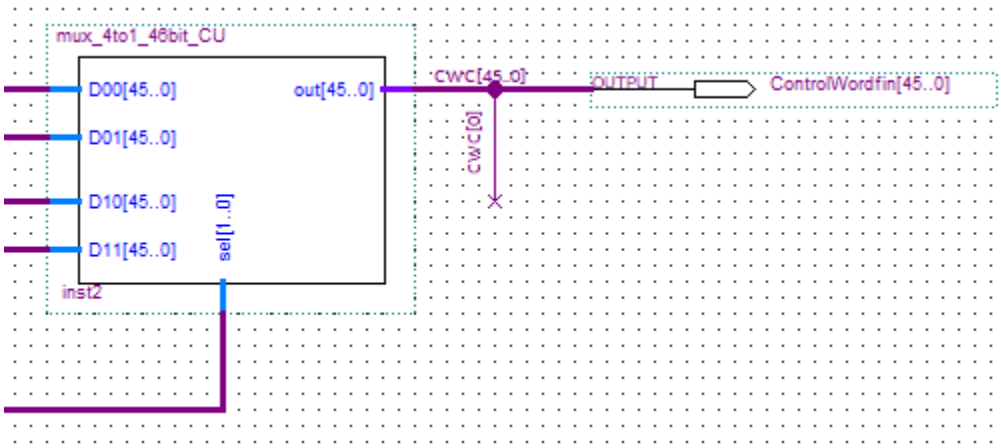
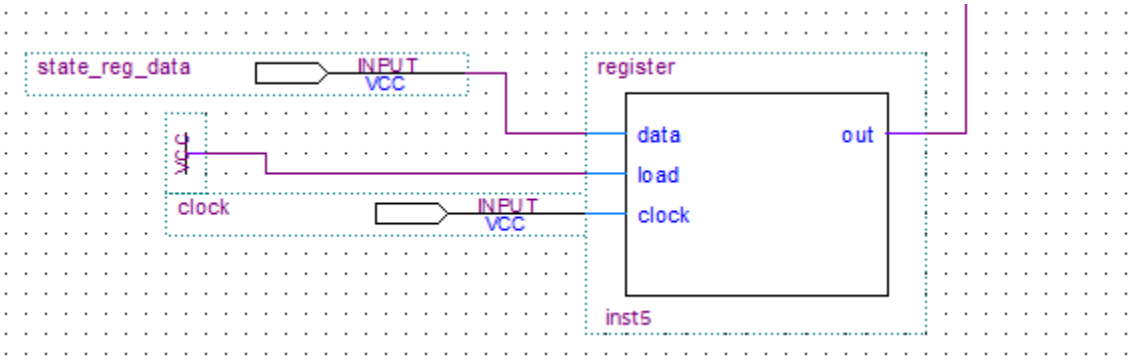


Figure 9-2: Control Unit Final Schematic (2 of 2)



Note: ControlWordOut_XX outputs connect to DXX inputs.

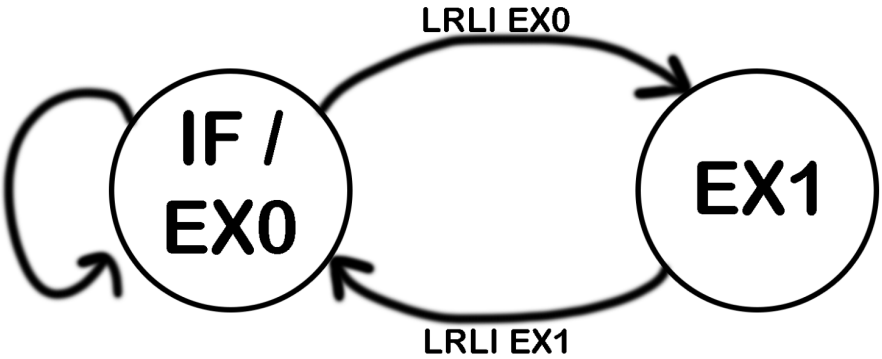
Figure 9-3: Control Unit Decoders Register



Note: out output connects to final (fourth) state input of 11 decoder

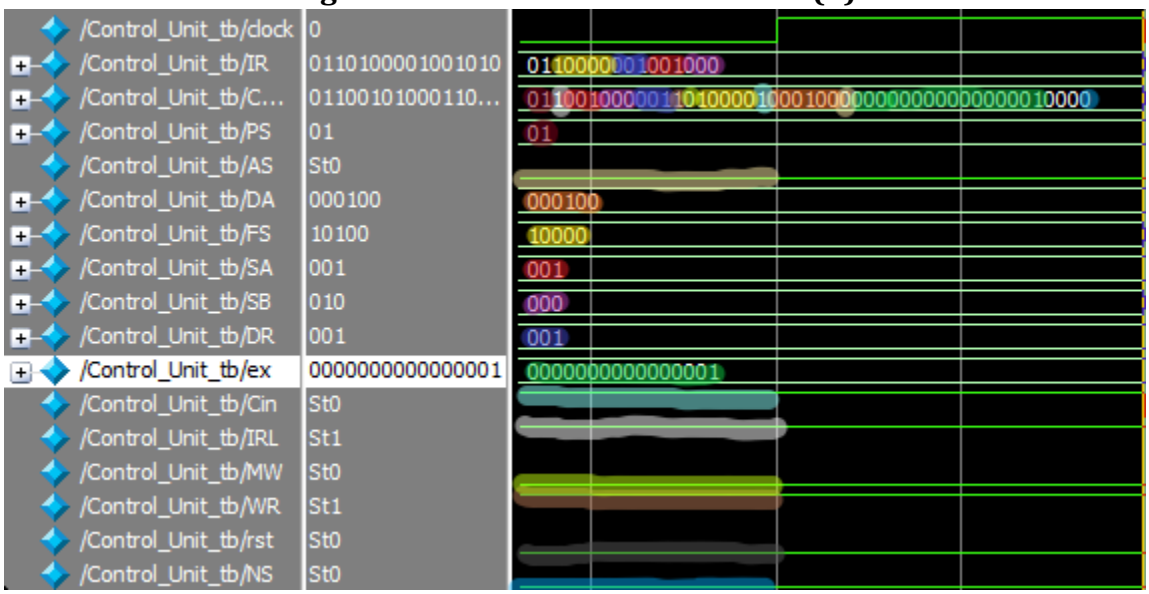
In order to create the control unit, four separate decoders were created for different opcodes starting with 00, 01, 10, and 11. After breaking the instructions down into their respective most significant two bits, tables were made to make things more readable and easier to work with (found in the instruction set architecture section). Some of the signals can be easily reduced to wires (for example if SA always belongs to the same bits of IR you already have your answer). Once those are dealt with, you can solve for the slightly more complex signals. K-maps were the method of choice for this process. Negative and Zero status signals were also included for branch instructions. Symbol files were created for each decoder, which were put into a schematic file and linked together. Each state input linked to the next state input until decoder 11 fed into the output of a register that held that data. The control word was broken down and bus/wire named into their signal and whether they corresponded to a 00, 01, 10, or 11 instruction. A symbol file was then created for that file, and brought into the file seen in the first two figures. As previously stated, the two most significant bits of IR are used as select bits, and are fed into a 46-bit 4-to-1 mux. The DXX inputs receive the different control words and the output ends up being the final control word based on the IR input.

Figure 9-4: Control Unit State Machine



The datapath utilizes Harvard Architecture. When looking at the state machine for the control unit, only two states are needed. Most instructions are single cycle because the instruction fetch and EX0 states are combined into one. This can be seen on the left of the diagram. There are only two instructions that require two cycles, CALL and LRLI. To do this, the program counter needs to be able to add an offset, and memory instructions need to be single cycle. Inverting the RAM clock solved that issue.

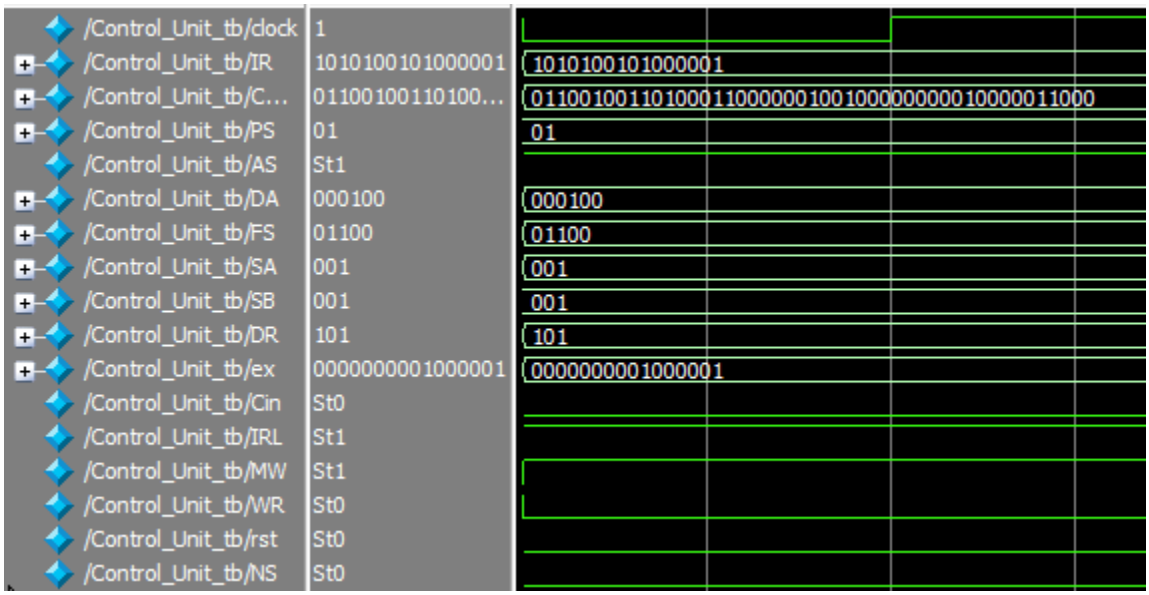
Figure 9-5: Control Unit Test Bench (1)



In this ModelSim screenshot, the increment operation is being executed. The two most significant bits of IR are 01, and along with the opcode and desired registers, this selects the corresponding control word that is shown on the screen. In an effort to make things more readable and easier to break down, most signals were broken down into their corresponding place in the control word and wired in the testbench. This makes it a bit easier to visualize where everything is going and what goes where. For the increment operation, The opcode goes in first followed by the destination register, source A register, and source B register (which really is a don't care). Although the literal value is set here, the AS signal is low so it is not being incorporated. Cin is high which is crucial for this operation since Cin must be 1 for the operation to actually be incremented

The following figures will not be labeled this way so it would be best to refer to the above figure or the control word figure located **HERE.**

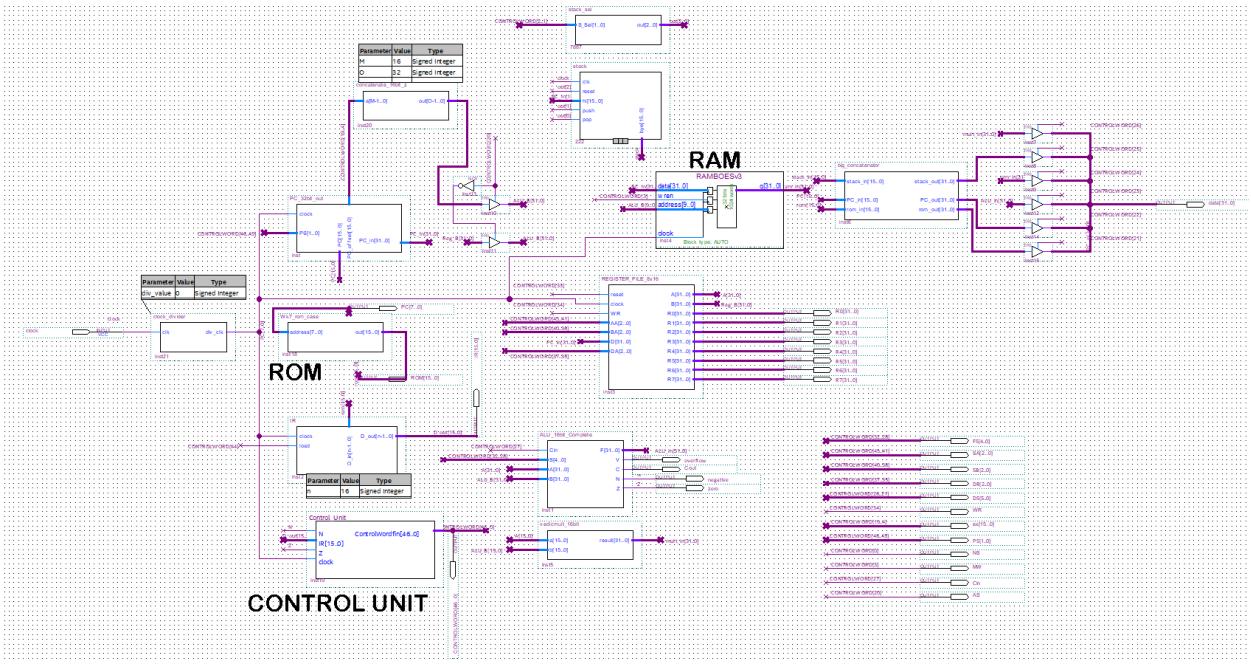
Figure 9-6: Control Unit Test Bench (2)



This snippet shows the control word generation for a store operation. A few key things here: The AS signal is 1 meaning the literal value is in use, the ex literal value is set to a specific address correlating to the one chosen in the IR input, write to memory is on and write to register is off, and SA is set to the correct value based on IR input. This would store whatever is in SA to address 41 (hex) in memory.

Section 10: The CPU

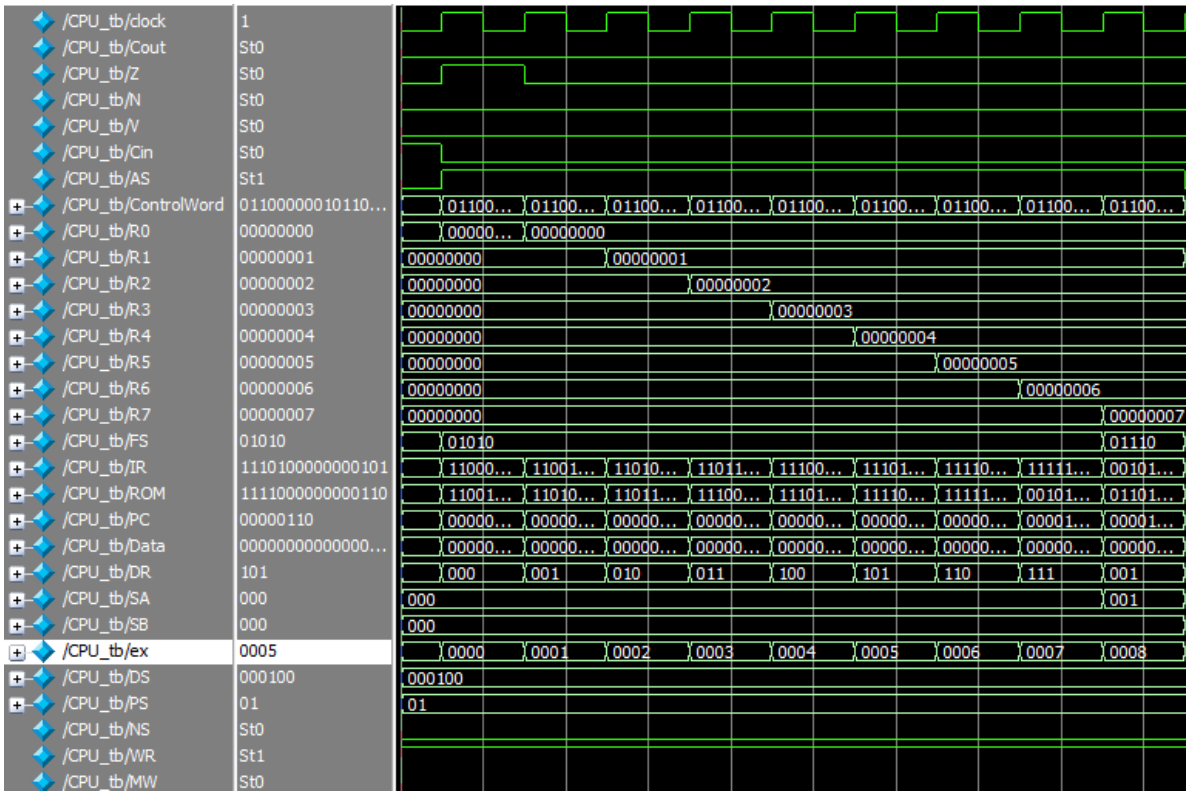
Figure 10-1: Complete Processor Datapath



Note: Smaller snippets of this complete datapath can be found in the **APPENDIX.**

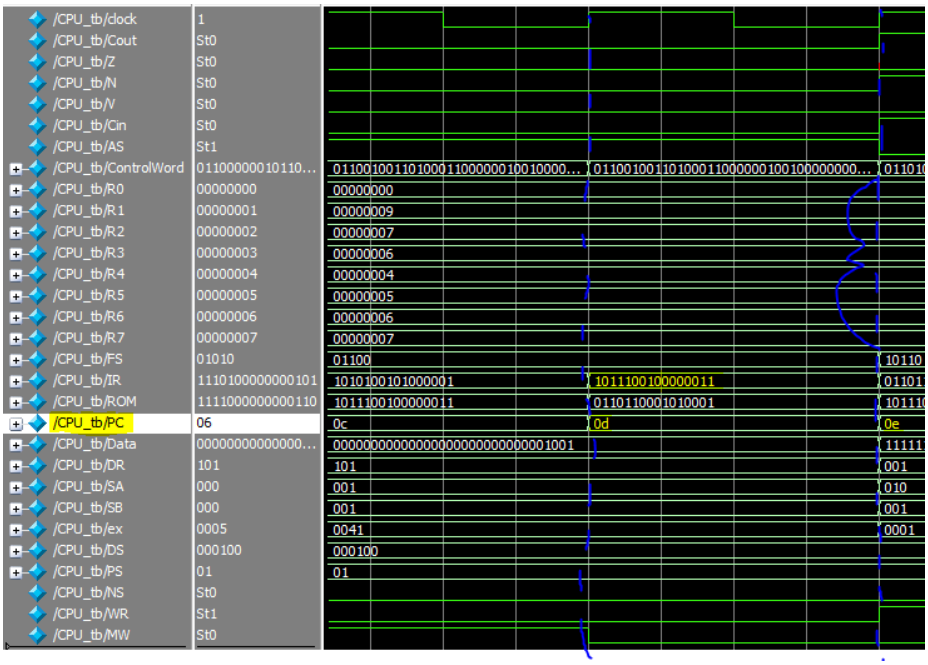
The testbench for our complete CPU was fairly simple. Outputs of the control word were wired for monitoring purposes and the clock was connected as an input. Once that was done, the processor did everything else as far as generating control words via instructions in the ROM. In the following testbench figures, we tested a few of the required instructions, loading to each register to make sure everything was working correctly. Again, there will be specific sections for individual optional instructions included and validating their functionality.

Figure 10-2: Testing LRI Instruction



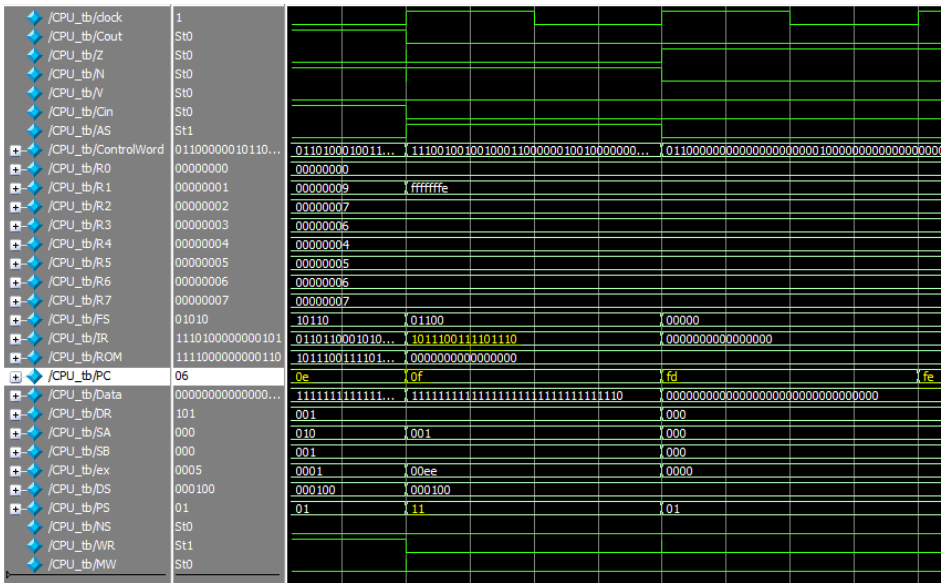
Here, the LRI instruction is being tested, so a value based on the literal value ex is passed onto the desired register. In this case the instruction called for a zero to be loaded into R0, a one to be loaded in R1, and so on. You can see that before a one is loaded into R1, the source was 001, and the literal value ex was also 0001. The instruction value can be seen in the IR signal, and the next instruction to be loaded can be seen in the ROM signal.

Figure 10-3: Testing BRN Instruction (1 of 2)



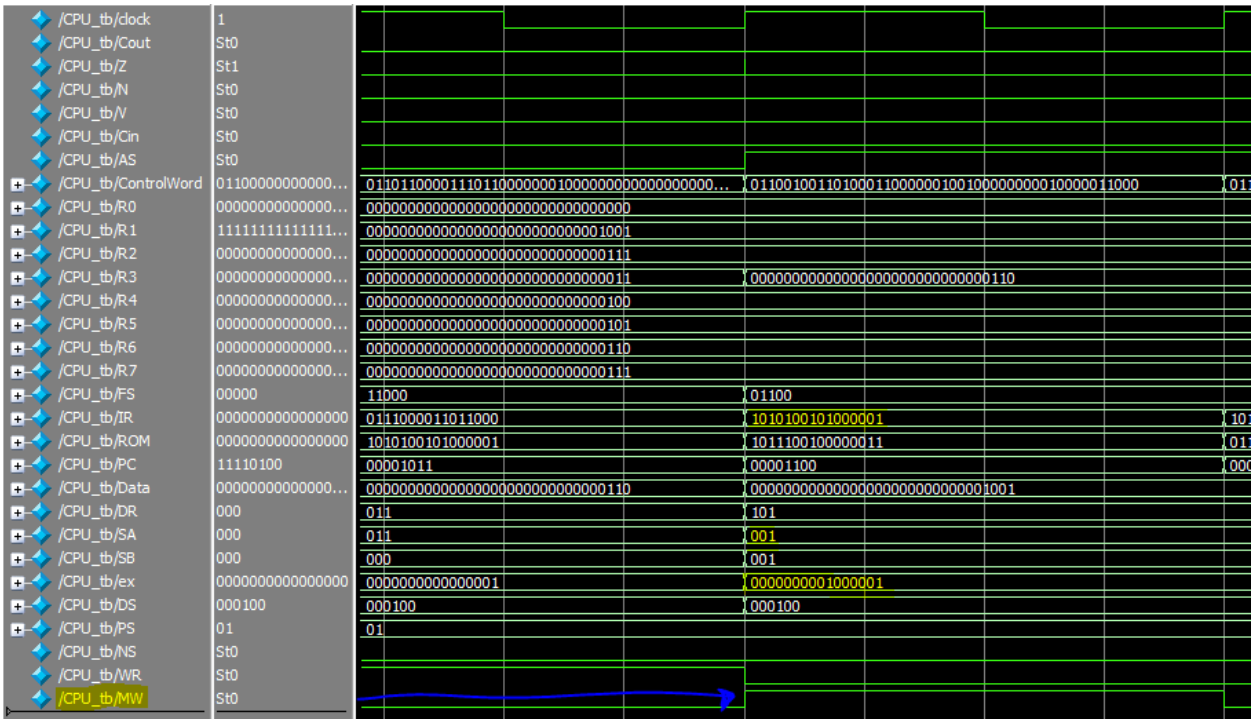
This is an example of the branch not being taken. The program counter continues to increment even though a branch instruction is being executed. This happens because this specific instruction is checking if R1 is negative, and in this case it's not. Therefore, the program counter continues its normal process.

Figure 10-4: Testing BRN Instruction (2 of 2)



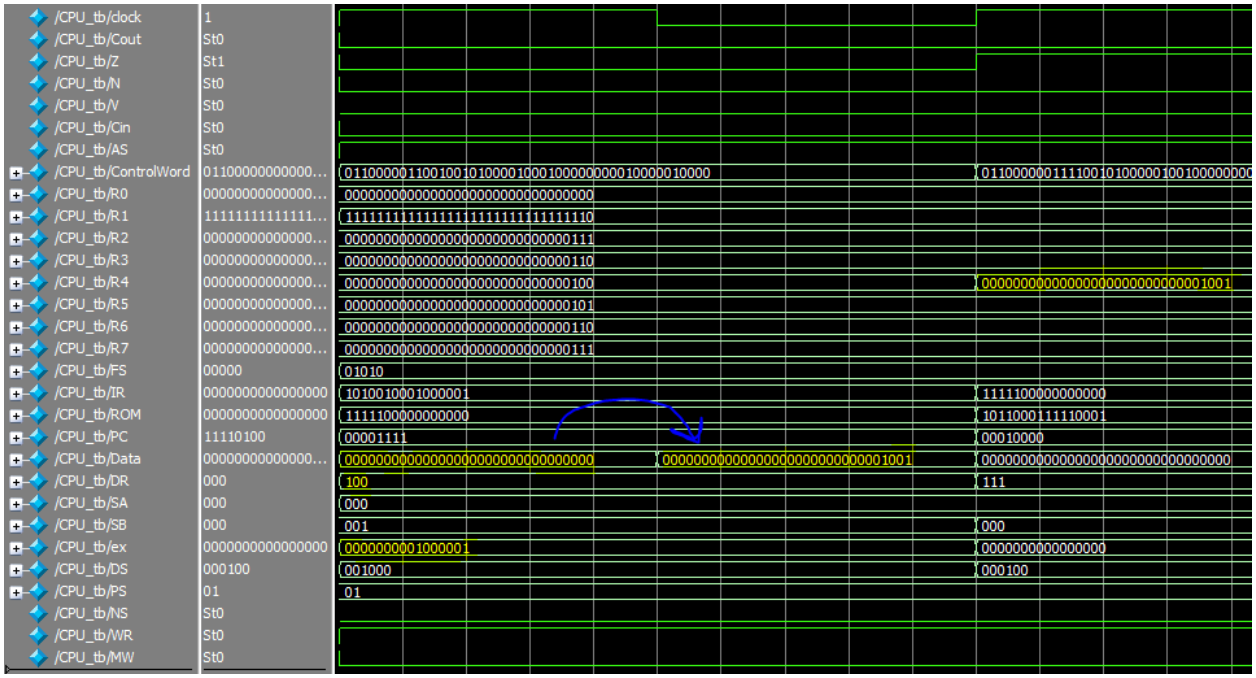
Here, the branch is taken because R1 is now negative. This sets the PS bits to 11 which will increment the program counter by an offset (in this case -15). This takes the PC to -3 and afterwards it returns to incrementing (where it will eventually get back to 0 and restart).

Figure 10-4: Testing STI Instruction



For the storing at an immediate address the corresponding bits are marked in yellow. Source A is one so it will store the data in R1 to address ex which is 41 in hex. Another thing to note is that MW (write to memory) is on and it is.

Figure 10-5: Testing STI Instruction



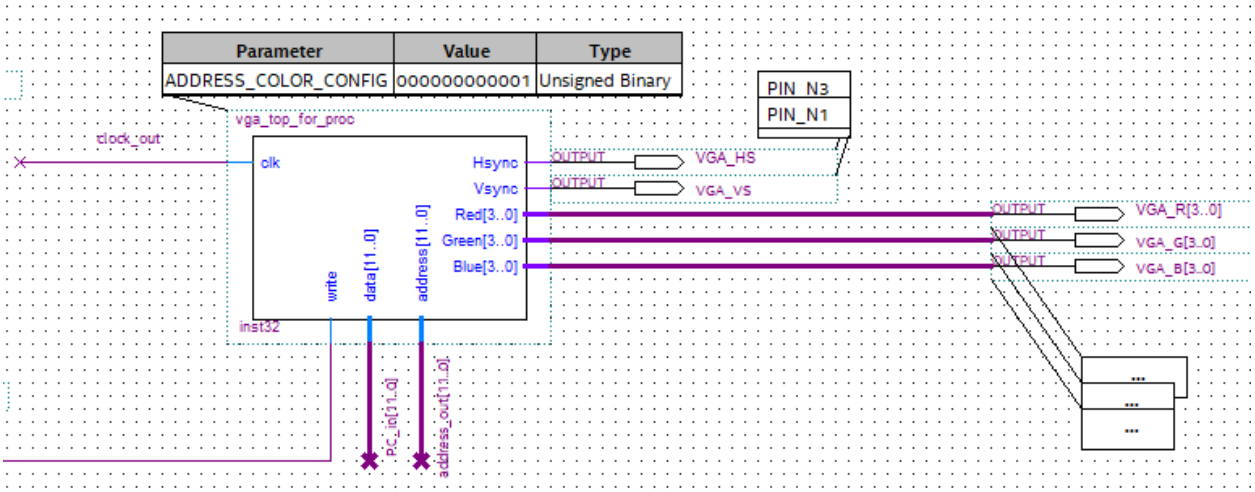
Here we are loading data (LDI). The address is 41 in hex, the data is available in time for the next clock cycle, and is loaded in DR 100 or R4. It is important to note that memory write is off and write to register is on.

The above tests go into detail about some of the required instructions, optional instructions (if supported) will be shown in individual sections.

Section 11: The Peripherals

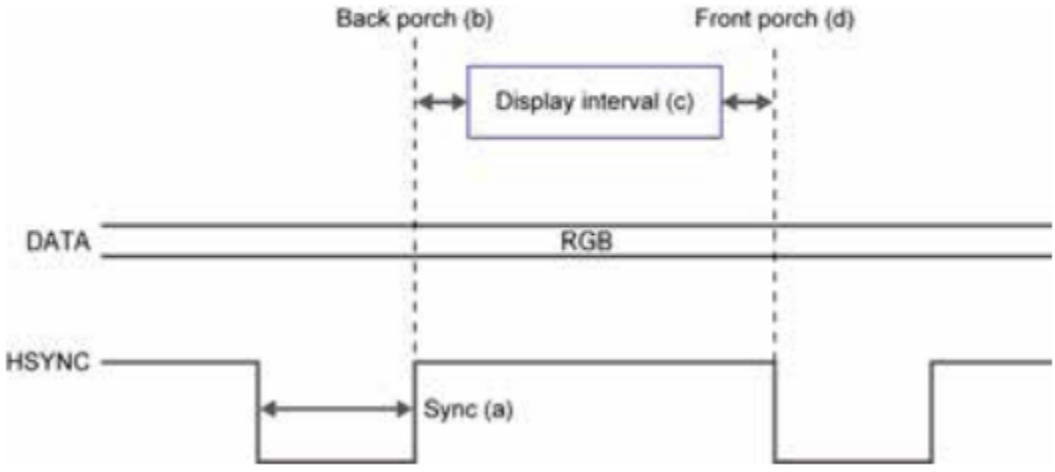
VGA (Jeremy)

Figure 11-1: VGA Block Connected to Processor



At its core, my VGA peripheral is really just a VGA controller with a few tweaks so it can act as a peripheral. In order to make the controller, I referred to the DE-10 Lite Manual which had timing specifications for general VGA purposes. It required a 25Mhz clock and I chose to implement counters that would sync the vertical and horizontal components.

Figure 11-2: Horizontal Timing For VGA



Looking at the horizontal timing, there is a display interval when the horizontal counter reaches a specific number, which will then trigger the vertical counter to start counting which will eventually enable the vertical sync signal. In Verilog, it wasn't extremely difficult to assign the RGB colors to that display interval, where they could then be driven by another signal when true, or zero when false. This is where the special function registers come in. There were several ways to implement the special function registers, but I ended up using only one. This was named 'color_config'. The SFR only received data if it's address and write were enabled. An address selector was used to output data based on the input of an address. Address zero would output a 12-bit 1. The address itself would go into the address input of the VGA block and the first bit of the address would be ANDed with the write to memory signal. Therefore, 'color_config' is technically mapped to 12'h001. Another signal was used to reserve the first address for the SFR, as it would only allow the output of RAM to pass through if none of the SFR addresses were being selected (using a tri-state buffer). The Verilog code for the horizontal sync and vertical sync, along with a clock divider can be found in the **APPENDIX**

Section 12: Instruction Set Architecture

Table 12-1: Opcode Field Descriptions

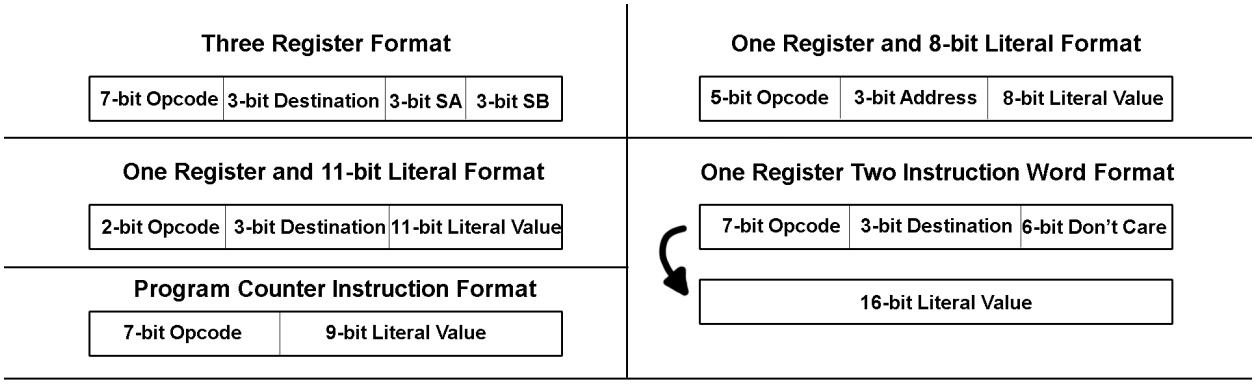
DR	Destination Register Address for Register File
SA	Source A Address for Register File
SB	Source B Address for Register File
AD	Literal Address Value
lit	Literal Data Value
PC	Program Counter (Will increment unless told otherwise)
zf	Zero Fill, extends literal with zeroes
se	Sign Extend, extends literal with sign bit
R[address]	Register File Register at Address
M[address]	Data Memory at Address

Figure 12-1: Instruction Set Summary

Mnemonic	Description	Opcode	RTL	
NOP	No Operation	0000000	N/A	
INC	Increment A by 1	0110000	R[DR] <= R[SA] + 1	Arithmetic and Shift Instructions
ADD	Add A + B	0110100	R[DR] <= R[SA] + R[SB]	
SUB	Subtract A - B	0110110	R[DR] <= R[SA] - R[SB]	
DEC	Decrement A by 1	0110010	R[DR] <= R[SA] - 1	
MUL	Multiply A and B	0110111	R[DR] <= R[SA] * R[SB]	
ADDC	Add with carry A + B + C	0110101	R[DR] <= R[SA] + R[SB] + C	
NEG	Negative -A	0110001	R[DR] <= ~R[SA] + 1	
SHR	Shift A right by 1	0111001	R[DR] <= sr R[SA]	
SHL	Shift A left by 1	0111000	R[DR] <= sl R[SA]	
CLR	Clear A to zero	0100000	R[DR] <= 0	Logic Instructions
SET	Set all A bits to 1	0101111	R[DR] <= FFFF	
NOT	Invert all A bits	0100011	R[DR] <= ~R[SA]	
AND	AND A & B	0101000	R[DR] <= R[SA] & R[SB]	
OR	OR A | B	0101110	R[DR] <= R[SA] | R[SB]	
XOR	XOR A ^ B	0100110	R[DR] <= R[SA] XOR R[SB]	
MOVA	Transfer A	0101100	R[DR] <= R[SA]	
MOVB	Transfer B	0101010	R[DR] <= R[SB]	
ADI	Add A + Literal	00001	R[DR] <= R[SA] + se-lit	Arithmetic and Logic with Immediate Operand
SBI	Subtract A - Literal	00010	R[DR] <= R[SA] - se-lit	
ANI	AND A & Literal	00011	R[DR] <= R[SA] & zf-lit	
ORI	OR A | Literal	00101	R[DR] <= R[SA] | zf-lit	
XRI	XOR A ^ Literal	00110	R[DR] <= R[SA] XOR zf-lit	
LRI	Load literal value to register	11	R[DR] <= se-lit	Loading a literal value
LRLI	Load register with LI literal	1000010	R[DR] <= lit	
LDI	Load data from memory at immediate address	10100	R[DR] <= M[zf-AD]	Memory Access Instructions
STI	Store data to memory at immediate address	10101	M[zf-AD] <= R[SA]	
LDR	Load data from memoey at register address	1000100	R[DR] <= M[R[SA]]	
STR	Store data to memort at register address	1000101	M[R[DR]] <= R[SA]	
PUSH	Push register value on stack	1000000	PUSH R[SA]	Stack Instructions
POP	Pop value from stack on register	1000001	R[DR] <= POP	
CALL	Call a subroutine	1001110	PUSH PC + 1, PC <= zf-AD	Program Counter Instructions
RET	Return from a subroutine	1001111	PC <= POP	
JUMPR	Jump to address in register	1001101	PC <= R[SA]	
BRZ	Branch to address is register is zero	10110	if (R[SA] == 0) PC <= PC + se-AD	Program Control Instructions
BRN	Branch to addres if register is negative	10111	if (R[SA] < 0) PC <= PC + se-AD	

In our instruction set architecture, there are a total of 36 instructions. The most significant bits of each instruction will always be the opcode. There are several different formats used.
. .

Figure 12-2: Instruction Set Formats



The 'Three Register Format', located in the top left corner of figure 7-2 will be used for:

- NOP
- Arithmetic Instructions (INC, ADD, SUB, DEC, MUL, ADDC, NEG)
- Shift Instructions (SHR, SHL)
- Logic Instructions (CLR, SET, NOT, AND, OR, XOR, MOVA, MOVB)
- Jump to Address (JMPR)
- Stack Instructions (PUSH, POP)
- Memory Access Instructions (LDR, STR)

The 'One Register and 8-bit Literal Format', located in the top right corner of figure 7-2 will be used for:

- Arithmetic Instructions with Immediate Operand (ADI, SBI)
- Logic Instructions with Immediate Operand (ANI, ORI, XRI)
- Memory Access Instructions (LDI, STI)
- Program Control Instructions (BRZ, BRN)

The 3-bit address is used as a destination register, source A register, or both destination and source A register.

The 'One Register Two Instruction Word Format' is only used for loading an immediate long literal value to a register (LRLI), while the 'One Register and 11-bit literal Format' will only be used for loading an immediate value to a register (LRI).

The 'Program Counter Instruction Format' will be used for program counter instructions RET and CALL.

Table 12-2: Table of Operations and Control Words
(Opcodes starting with 01)
(1 of 2)

. . .	CLR	SET	NOT	AND	OR	XOR	MOVA	MOVB
OPCODE	0100000	0101111	0100011	0101000	001110	0100110	0101100	0101010
	C	*O*	*N*	*T*	*R*	*O*	*L*	
PS[46:45]	2'b01	2'b01	2'b01	2'b01	2'b01	2'b01	2'b01	2'b01
IRL[44]	1'b1	1'b1	1'b1	1'b1	1'b1	1'b1	1'b1	1'b1
SA[43:41]	x x x	x x x	IR[5:3]	IR[5:3]	IR[5:3]	IR[5:3]	IR[5:3]	x x x
SB[40:38]	x x x	x x x	x x x	IR[2:0]	IR[2:0]	IR[2:0]	x x x	IR[2:0]
DR[37:35]	IR[8:6]	IR[8:6]	IR[8:6]	IR[8:6]	IR[8:6]	IR[8:6]	IR[8:6]	IR[8:6]
WR[34]	1'b1	1'b1	1'b1	1'b1	1'b1	1'b1	1'b1	1'b1
CLR[33]	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0
FS[32:28]	5'b00000	5'b01111	5'b10001	5'b01000	5'b01110	5'b00110	5'b01100	5'b01010
Cin[27]	x	0	x	x	x	x	x	x
DS[26:21]	5'b00100	5'b00100	5'b00100	5'b00100	5'b00100	5'b00100	5'b00100	5'b00100
AS[20]	x	x	x	1'b0	1'b0	1'b0	x	1'b0
ex[19:4]	xxxxxxxxxx	xxxxxxxxxx	xxxxxxxxxx	xxxxxxxxxx	xxxxxxxxxx	xxxxxxxxxx	xxxxxxxxxx	xxxxxxxxxx
MW[3]	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0
SS[2:1]	2'b00	2'b00	2'b00	2'b00	2'b00	2'b00	2'b00	2'b00
NS[0]	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0
S	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0

Table 12-3: Table of Operations and Control Words
(Opcodes starting with 01)
(2 of 2)

INC	ADD	ADDC	SUB	DEC	NEG	SHR	SHL
0110000	0 110100	0110101	0110110	0 110010	0110001	010000	0111000
			W	*O*	*R*	*D*	
2'b01	2'b01	2'b01	2'b01	2'b01	2'b01	2'b01	2'b01
1'b1	1'b1	1'b1	1'b1	1'b1	1'b1	1'b1	1'b1
IR[5:3]	IR[5:3]	IR[5:3]	IR[5:3]	IR[5:3]	IR[5:3]	IR[5:3]	IR[5:3]
x x x	IR[2:0]	IR[2:0]	IR[2:0]	x x x	x x x	x x x	x x x
IR[8:6]	IR[8:6]	IR[8:6]	IR[8:6]	IR[8:6]	IR[8:6]	IR[8:6]	IR[8:6]
1'b1	1'b1	1'b1	1'b1	1'b1	1'b1	1'b1	1'b1
1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0
5'b10000	5'b10100	5'b10100	5'b10110	5'b10110	5'b10011	5'b11001	5'b11000
1'b1	1'b0	1'b1	1'b1	1'b1	x	1'b1	x
5'b00100	5'b00100	5'b00100	5'b00100	5'b00100	5'b00100	5'b00100	5'b00100
x	1'b0	1'b0	1'b0	1'b1	x	x	x
xxxxxxxxxx	xxxxxxxxxx	xxxxxxxxxx	xxxxxxxxxx	16'b1	xxxxxxxxxx	xxxxxxxxxx	xxxxxxxxxx
1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0
2'b00	2'b00	2'b00	2'b00	2'b00	2'b00	2'b00	2'b00
1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0
1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0

Table 12-4: Table of Operations and Control Words
(Opcodes starting with 10)
(1 of 2)

. . .	LRLI	< < < < <	LDI	STI	STR	LDR	PUSH	POP
OPCODE	1000010	1000010	10100	10101	1000101	1000100	1000000	1000001
	C	*O*	*N*	*T*	*L*			
PS[46:45]	2'b01	2'b01	2'b01	2'b01	2'b01	2'b01	2'b01	2'b01
IRL[44]	1'b0	1'b1	1'b1	1'b1	1'b1	1'b1	1'b1	1'b1
SA[43:41]	x x x	x x x	x x x	IR[10:8]	IR[8:6]	x x x	IR[5:3]	x x x
SB[40:38]	x x x	x x x	x x x	x x x	IR[2:0]	IR[2:0]	x x x	x x x
DR[37:35]	IR[8:6]	IR[8:6]	IR[10:8]	x x x	x x x	IR[8:6]	x x x	IR[8:6]
WR[34]	1'b1	1'b0	1'b1	1'b0	1'b0	1'b1	1'b0	1'b1
CLR[33]	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0
FS[32:28]	5'b00000	5'b01111	5'b10001	5'b01000	5'b01100	5'b01010	5'b01110	5'b00110
Cin[27]	x	0	x	x	x	x	x	x
DS[26:21]	5'b000100	5'b000100	5'b000100	5'b000100	5'b000100	5'b000100	5'b000100	5'b000100
AS[20]	x	1'b1	1'b1	1'b1	1'b0	1'b0	x	x
ex[19:4]	xxxxxxxxxx	IR[15:0]	zf(IR[7:0]	zf(IR[7:0]	xxxxxxxxxx	xxxxxxxxxx	xxxxxxxxxx	xxxxxxxxxx
MW[3]	1'b0	1'b0	1'b0	1'b1	1'b1	1'b0	1'b0	1'b0
SS[2:1]	2'b00	2'b00	2'b00	2'b00	2'b00	2'b00	2'b01	2'b10
NS[0]	1'b1	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0
S	1'b0	1'b1	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0

Table 12-5: Table of Operations and Control Words
(Opcodes starting with 10)
(2 of 2)

CALL	< < < < <	RET	BRZ (take)	BRZ (don't)	BRN (take)	BRN (don't)	JMPR
1001110	1001110	1001111	10110		10111		1001101
			W	*O*	*R*	*D*	
2'b10	2'b01	2'b10	2'b11	2'b01	2'b11	2'b01	2'b11
1'b0	1'b1	1'b1	1'b1	1'b1	1'b1	1'b1	1'b1
x x x	x x x	x x x	IR[10:8]	x x x	IR[10:8]	x x x	x x x
x x x	x x x	x x x	x x x	x x x	x x x	x x x	x x x
x x x	x x x	x x x	x x x	x x x	x x x	x x x	x x x
1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0
1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0
5'b10000	5'b10100	5'b10100	5'b10110	5'b10110	5'b10011	5'b11001	5'b11000
1'b1	1'b0	1'b1	1'b1	1'b1	x	1'b1	x
5'b000100	5'b000100	5'b000100	5'b000100	x x x x x	5'b000100	x x x x x	5'b000100
x	1'b1	1'b1	1'b1	x	1'b1	x	1'b1
xxxxxxxxxx	zf(IR[8:0])	xxxxxxxxxx	zf(IR[7:0]	xxxxxxxxxx	zf(IR[7:0]	xxxxxxxxxx	zf(IR[8:0])
1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0
2'b01	2'b00	2'b10	2'b00	2'b00	2'b00	2'b00	2'b00
1'b1	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0
1'b0	1'b1	1'b0	1'b0	1'b0	1'b0	1'b0	1'b0

Table 12-6: Table of Operations and Control Words
(Opcodes starting with 00)

. . .	ADI	SBI	ANI	ORI	XRI	LRI
OPCODE	**00001**	**00010**	**00011**	**00101**	**00110**	**00011**
		C		***W***		
PS[46:45]	2'b**01**	2'b**01**	2'b**01**	2'b**01**	2'b**01**	2'b**01**
IRL[44]	1'b**1**	1'b**1**	1'b**1**	1'b**1**	1'b**1**	1'b**1**
SA[43:41]	IR[10:8]	IR[10:8]	IR[10:8]	IR[10:8]	IR[10:8]	x x x
SB[40:38]	x x x	x x x	x x x	x x x	x x x	x x x
DR[37:35]	IR[10:8]	IR[10:8]	IR[10:8]	IR[10:8]	IR[10:8]	IR[13:11]
WR[34]	1'b**1**	1'b**1**	1'b**1**	1'b**1**	1'b**1**	1'b**1**
CLR[33]	1'b**0**	1'b**0**	1'b**0**	1'b**0**	1'b**0**	1'b**0**
FS[32:28]	5'b**10100**	5'b**10110**	5'b**01000**	5'b**01110**	5'b**00110**	5'b**01010**
Cin[27]	x	1'b**0**	x	x	x	x
DS[26:21]	5'b**000100**	5'b**000100**	5'b**000100**	5'b**000100**	5'b**000100**	5'b**000100**
AS[20]	1'b**1**	1'b**1**	1'b**1**	1'b**1**	1'b**1**	1'b**1**
ex[19:4]	zf(IR[7:0])	zf(IR[7:0])	zf(IR[7:0])	zf(IR[7:0])	zf(IR[7:0])	zf(IR[7:0])
MW[3]	1'b**0**	1'b**0**	1'b**0**	1'b**0**	1'b**0**	1'b**0**
SS[2:1]	2'b**00**	2'b**00**	2'b**00**	2'b**00**	2'b**00**	2'b**00**
NS[0]	1'b**0**	1'b**0**	1'b**0**	1'b**0**	1'b**0**	1'b**0**
S	1'b**0**	1'b**0**	1'b**0**	1'b**0**	1'b**0**	1'b**0**

As previously stated in the control unit section, the opcodes were broken down and sorted by their two most significant bits, and four decoders were made for 00, 01, 10, and 11. So, in these past figures (7-5 through 7-8) the operations are sorted by that system instead of being grouped by their function.

The following figures will describe each operation in detail, including a description of the operation, the operands, the syntax, and the exact operation they perform

Figure 12-3: Detailed Instruction Set List with Descriptions

(1 of 5)

NOP — No Operation

Syntax: NOP

Operands: None

Operations: None

Description:

No Operation is Executed

INC — Increment

Syntax: INC DR, SA

Operands: 0 ≤ DR ≤ 7
0 ≤ SA ≤ 7

Operations:
R[DR] ← R[SA] + 1

Description:

Increment SA register value by 1,store value in DR

ADD — Add

Syntax: ADD DR, SA, SB

Operands: 0 ≤ DR ≤ 7
0 ≤ SB ≤ 7 0 ≤ SA ≤ 7

Operations:
R[DR] ← R[SA] + R[SB]

Description:

Add values in registers SA and SB, store in DR

SUB — Subtract

Syntax: SUB DR, SA, SB

Operands: 0 ≤ DR ≤ 7
0 ≤ SB ≤ 7 0 ≤ SA ≤ 7

Operations:
R[DR] ← R[SA] - R[SB]

Description:

Subtract values in registers SA and SB, store in DR

DEC — Decrement

Syntax: DEC DR, SA

Operands: 0 ≤ DR ≤ 7
0 ≤ SA ≤ 7

Operations:
R[DR] ← R[SA] - 1

Description:

Decrement SA register value by 1, store value in DR

MUL — Multiply

Syntax: MUL DR, SA, SB

Operands: 0 ≤ DR ≤ 7
0 ≤ SB ≤ 7 0 ≤ SA ≤ 7

Operations:
R[DR] ← R[SA] * R[SB]

Description:

Multiply values in registers SA and SB, store in DR

ADDC — Add w/ Carry

Syntax:
ADDC DR, SA, SB, C

Operands: 0 ≤ DR ≤ 7
0 ≤ SB ≤ 7 0 ≤ SA ≤ 7

Operations:
R[DR] ← R[SA] + R[SB] + C

Description:

Add values in registers SA, SB, and carry value, store in DR

NEG — Negative

Syntax: NEG DR, SA

Operands: 0 ≤ DR ≤ 7
0 ≤ SA ≤ 7

Operations:
R[DR] ← ~R[SA] + 1

Description:

Negate value in register SA, store in DR

Figure 12-4: Detailed Instruction Set List with Descriptions

(2 of 5)

SHR — Shift Right

Syntax: SHR DR, SA

Operands: 0 ≤ DR ≤ 7
0 ≤ SA ≤ 7

Operations:
R[DR] ← sr R[SA]

Description:

Shift value in register SA to the right, store in DR

SHL — Shift Left

Syntax: SHL DR, SA

Operands: 0 ≤ DR ≤ 7
0 ≤ SA ≤ 7

Operations:
R[DR] ← sl R[SA]

Description:

Shift value in register SA to the left, store in DR

CLR — Clear Register

Syntax: CLR DR

Operands: 0 ≤ DR ≤ 7
0 ≤ SB ≤ 7 0 ≤ SA ≤ 7

Operations:
R[DR] ← 0

Description:

Clear bits in register DR

SET — Set

Syntax: SET DR

Operands: 0 ≤ DR ≤ 7
0 ≤ SB ≤ 7 0 ≤ SA ≤ 7

Operations:
R[DR] ← FFFF

Description:

Set bits in register DR to 1

NOT — Logical NOT

Syntax: NOT DR, SA

Operands: 0 ≤ DR ≤ 7
0 ≤ SA ≤ 7

Operations:
R[DR] ← ~R[SA]

Description:

NOT value of register SA, store in DR

AND — Logical AND

Syntax: AND DR, SA, SB

Operands: 0 ≤ DR ≤ 7
0 ≤ SB ≤ 7 0 ≤ SA ≤ 7

Operations:
R[DR] ← R[SA] & R[SB]

Description:

AND values in registers SA and SB, store in DR

OR — Logical OR

Syntax: OR DR, SA, SB

Operands: 0 ≤ DR ≤ 7
0 ≤ SB ≤ 7 0 ≤ SA ≤ 7

Operations:
R[DR] ← R[SA] | R[SB]

Description:

OR values in registers SA and SB, store in DR

XOR — Logical XOR

Syntax: XOR DR, SA, SB

Operands: 0 ≤ DR ≤ 7
0 ≤ SB ≤ 7 0 ≤ SA ≤ 7

Operations:
R[DR] ← R[SA] ^| R[SB]

Description:

XOR values in registers SA and SB, store in DR

Figure 12-5: Detailed Instruction Set List with Descriptions

(3 of 5)

MOVA Transfer A

Syntax: MOVA DR, SA
Operands: 0 ≤ DR ≤ 7
0 ≤ SA ≤ 7
Operations:
R[DR] ← R[SA]
Description:
Move value in SA to DR

MOVB Transfer B

Syntax: MOVB DR, SB
Operands: 0 ≤ DR ≤ 7
0 ≤ SB ≤ 7
Operations:
R[DR] ← R[SB]
Description:
Move value in SB to DR

ADI Add Immediate

Syntax: ADI DR, SA, ex
Operands: 0 ≤ DR ≤ 7
0 ≤ ex ≤ 255 0 ≤ SA ≤ 7
Operations:
R[DR] ← R[SA] + zf-lit
Description:
Add value in register SA with zf literal, store in DR

SBI Subtract Immediate

Syntax: SBI DR, SA, ex
Operands: 0 ≤ DR ≤ 7
0 ≤ ex ≤ 255 0 ≤ SA ≤ 7
Operations:
R[DR] ← R[SA] - zf-lit
Description:
Subtract value in register SA with zf literal, store in DR

ANI AND Immediate

Syntax: ANI DR, SA ex
Operands: 0 ≤ DR ≤ 7
0 ≤ SA ≤ 7
Operations:
R[DR] ← R[SA] & zf-lit
Description:
AND value in register SA with zf literal, store in DR

ORI OR Immediate

Syntax: ORI DR, SA, ex
Operands: 0 ≤ DR ≤ 7
0 ≤ ex ≤ 255 0 ≤ SA ≤ 7
Operations:
R[DR] ← R[SA] | zf-lit
Description:
OR value in register SA with zf literal, store in DR

XRI XOR Immediate

Syntax: XRI DR, SA, ex
Operands: 0 ≤ DR ≤ 7
0 ≤ ex ≤ 255 0 ≤ SA ≤ 7
Operations:
R[DR] ← R[SA] ^| zf-lit
Description:
XOR value in register SA with zf literal, store in DR

LRI Load Immediate

Syntax: LRI DR, ex
Operands: 0 ≤ DR ≤ 7
0 ≤ ex ≤ 255 0 ≤ SA ≤ 7
Operations:
R[DR] ← zf-lit
Description:
Load zf literal value into register DR

Figure 12-6: Detailed Instruction Set List with Descriptions

(4 of 5)

LRLI Load Long Immediate

Syntax: LRLI DR, ex
Operands: 0 ≤ DR ≤ 7
0 ≤ ex ≤ 65535
Operations:
R[DR] ← lit
Description:
Load Long literal value ex to register DR in state 0

LDI Load from Immediate Address

Syntax: LDI DR, ex
Operands: 0 ≤ DR ≤ 7
0 ≤ ex ≤ 255
Operations:
R[DR] ← M[zf-AD]
Description:
Load data from memory to DR at address ex

STI Store to Immediate Address

Syntax: STI ex, SA
Operands: 0 ≤ SA ≤ 7
0 ≤ ex ≤ 255
Operations:
M[zf-AD] ← R[SA]
Description:
Store data from SA to memory address ex

LDR Load from Register Address

Syntax: SBI DR, SA
Operands: 0 ≤ SA ≤ 7
0 ≤ ex ≤ 255
Operations:
R[DR] ← M[R[SA]]
Description:
Load data from memory to DR at address SA

STR Store to Register Address

Syntax: STR DR, SA
Operands: 0 ≤ DR ≤ 7
0 ≤ SA ≤ 7
Operations:
M[R[DR]] ← R[SA]
Description:
Store data to DR from memory address SA

PUSH Push Register Value

Syntax: PUSH SA
Operands: 0 ≤ SA ≤ 7
Operations:
PUSH R[SA]
Description:
Push register value SA onto stack

POP Pop register value

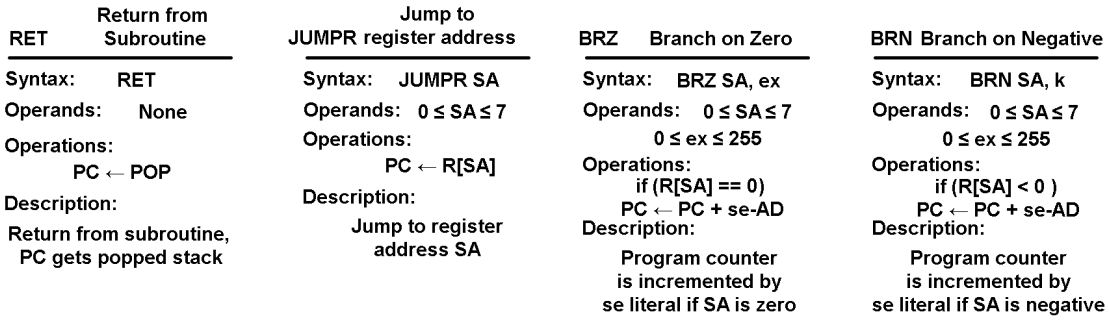
Syntax: POP DR
Operands: 0 ≤ DR ≤ 7
Operations:
R[DR] ← POP
Description:
Pop value from stack on register DR

CALL Call subroutine

Syntax: CALL ex
Operands: 0 ≤ ex ≤ 255
Operations:
PUSH PC + 1, PC ← zf-AD
Description:
Call a subroutine
Push PC + 1 onto stack,
PC gets zf ex address

Figure 12-7: Detailed Instruction Set List with Descriptions

(5 of 5)



Section 13: Example Program

Figure 13-1: Example Program Code (Jeremy)

```
module rom_case_rainbow(out, address); //10
    output reg[15:0] out;
    input [7:0] address; //address- 8 deep memory

    // This program will cycle through 12 bits of RGB color by loading select bits into
    // different register file registers and storing them in address 0 (color_config register)
    // an address selector will (in this case) always chose address zero which outputs
    // 12'b000000000001 which is ANDed with MW (mem write). If that is true, write is enabled
    // Although the address stored into is 0, the address selector takes that zero and outputs a 1
    // meaning the SFR color_config is actually mapped to a 12-bit 1.If both address and write are
    // 1, color_config <= data. For the color to actually be seen a clock divider is required
    // or else you'll just see vertical / diagonal colored lines.

    // Order of color: White -- Red -- Orange -- Yellow -- Green -- Blue -- Indigo -- Violet

    // This program could be easily modified to show combinations of colors or simply a white screen.

    always @(address) begin
        case(address)
      8'h00:  out = 16'b1111000011111111; // LRI R6, 255 // Setting bits to Multiply
      8'h01:  out = 16'b0111000110110000; // SHL R6, R6  // Shifting to the left
      8'h02:  out = 16'b0011011011111111; // XRI R6, 1   // XOR with lit to get 100000001
      8'h03:  out = 16'b1111010011111111; // LRI R5, 255 // Load 11111111 into R5
      8'h04:  out = 16'b0110111111110101; // MUL R7, R6, R5 // Multiply R6 with R5 to get all 1's

      8'h05:  out = 16'b1100000000001111; // LRI R0, 15 = Red
      8'h06:  out = 16'b1100100000101111; // LRI R1, 47 = Orange
      8'h07:  out = 16'b1101000000111111; // LRI R2, 63 = Yellow
      8'h08:  out = 16'b1101100011110000; // LRI R3, 240 = Green
      8'h09:  out = 16'b0100011100101101; // NOT R4, R5   // Setting bits for Blue
      8'h0A:  out = 16'b0101100101100010; // MOVA R5, R4  // Setting Indigo
      8'h0B:  out = 16'b0110000101101000; // INC R5, R5   // ^^^^^^^^^^^^^^^^^^^^
      8'h0C:  out = 16'b0110000110101000; // INC R6, R5   // Setting Violet
      8'h0D:  out = 16'b0110000110110000; // INC R6, R6   // ^^^^^^^^^^^^^^^^^^^^

      8'h0E:  out = 16'b1010100000000000; // STI R0, 0  // Storing Red
      8'h0F:  out = 16'b1010100100000000; // STI R1, 0  // Storing Orange
      8'h10:  out = 16'b1010101000000000; // STI R2, 0  // Storing Yellow
      8'h11:  out = 16'b1010101100000000; // STI R3, 0  // Storing Green
      8'h12:  out = 16'b1010110000000000; // STI R4, 0  // Storing Blue
      8'h13:  out = 16'b1010111010000000; // STI R5, 0  // Storing Indigo
      8'h14:  out = 16'b1010111000000000; // STI R6, 0  // Storing Violet
      8'h15:  out = 16'b1010111100000000; // STI R7, 0  // Storing White
      8'h16:  out = 16'b1010111100000000; // STI R7, 0

      8'h17:  out = 16'b1011111111110110; // BRN R7, -10 // Loop back to storing values if
                                                        // R7 is negative, always true


            default: out = 16'b0000000000000000; //NOP
        endcase
    end
endmodule // rom_case
```

Above is my complete example program using my VGA peripheral. Comments on what the program does and how it works are included before the 'always' block. A general summary is that it changes the RGB colors in a looped 'rainbow' sequence. The program also makes use of the advanced multiply instruction to set register 7 to 65535, meaning all colors will be active and the screen will be white.

Section 14: Performance

Unfortunately for me (Jeremy), I was unable to get the overclock ROM to function properly with my processor. Trying to debug was a nightmare as compile times were now in the 10-15 minute range and because of the delay in the program it was hard to see everything in ModelSim. I think the issue may have been caused by my registers being 32 bits, making the delay hundreds of billions of clock cycles. While I tried to make a VGA overclock test, compile times were just as long and I didn't have the time to get everything straightened out.

I'm confident that in the right circumstances my processor could be tested for max frequency, and in that case I could shine some light on how it could be improved further.

While I have no current idea on how I would do this for my datapath, a pipeline would boost performance a decent bit. Would take a lot of careful thought during the process and to get the full benefit there would be a system to negate hazards.

Changing the instruction set architecture could also boost performance. I debated making the instructions 32 bits, initially with more room for literal values, but it could also be implemented to incorporate multiple instructions in one.

I feel like those two additions / changes would be the right initial steps in making a higher - performance processor.

(Ali Elhamawi)
I was unable to get the ROM to work correctly and was not able to test the overclocking. I am unsure of the issue as of now. I will hopefully keep attempting to figure out why it is now working.
I also had issues with my DE0 board and it has caused complications in some testing.

Section 15: Personal Enhancements and Features (Jeremy)

This is a section intended to showcase the optional / advanced features of my processor. Some of the features included in here may overlap with those in the overall processor, but I aim to give more of a detailed description in this section.

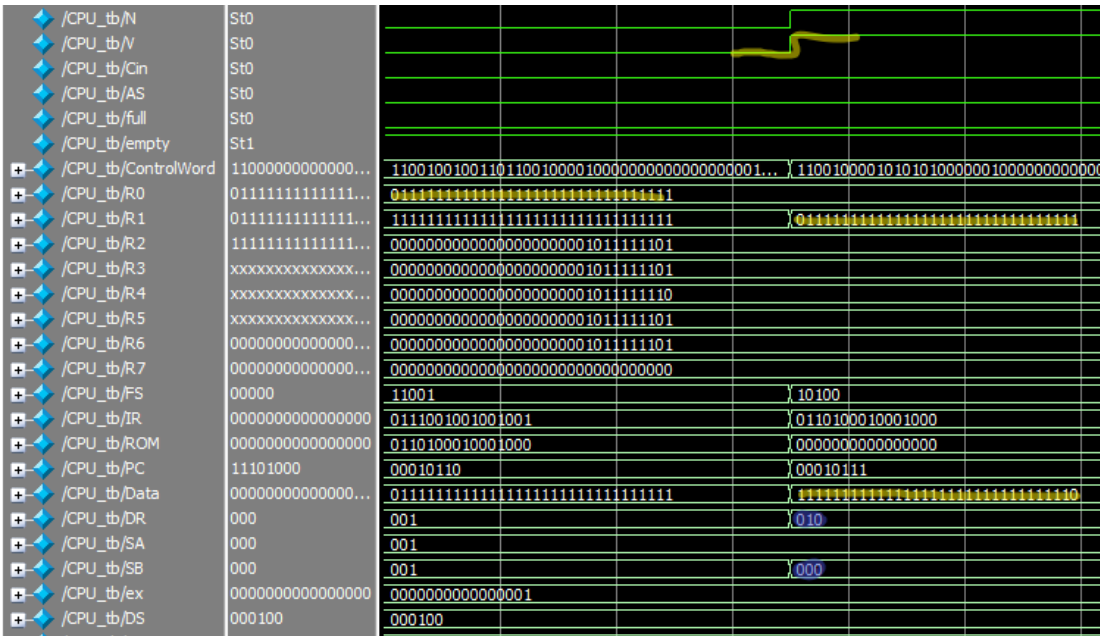
ALU: My ALU is quite similar to the one in the design guide, it can add with carry, has every status flag, but cannot do arithmetic shifts.

Figure 15-1: ALU Flags

```
assign N = ALU_Result[31];
assign Z = ~|ALU_Result;
assign F = ALU_Result;
```

```
assign V = (C ^ Inner_Carry[31]) ? 1'b1 : 1'b0;
```

Figure 15-2: Overflow Example

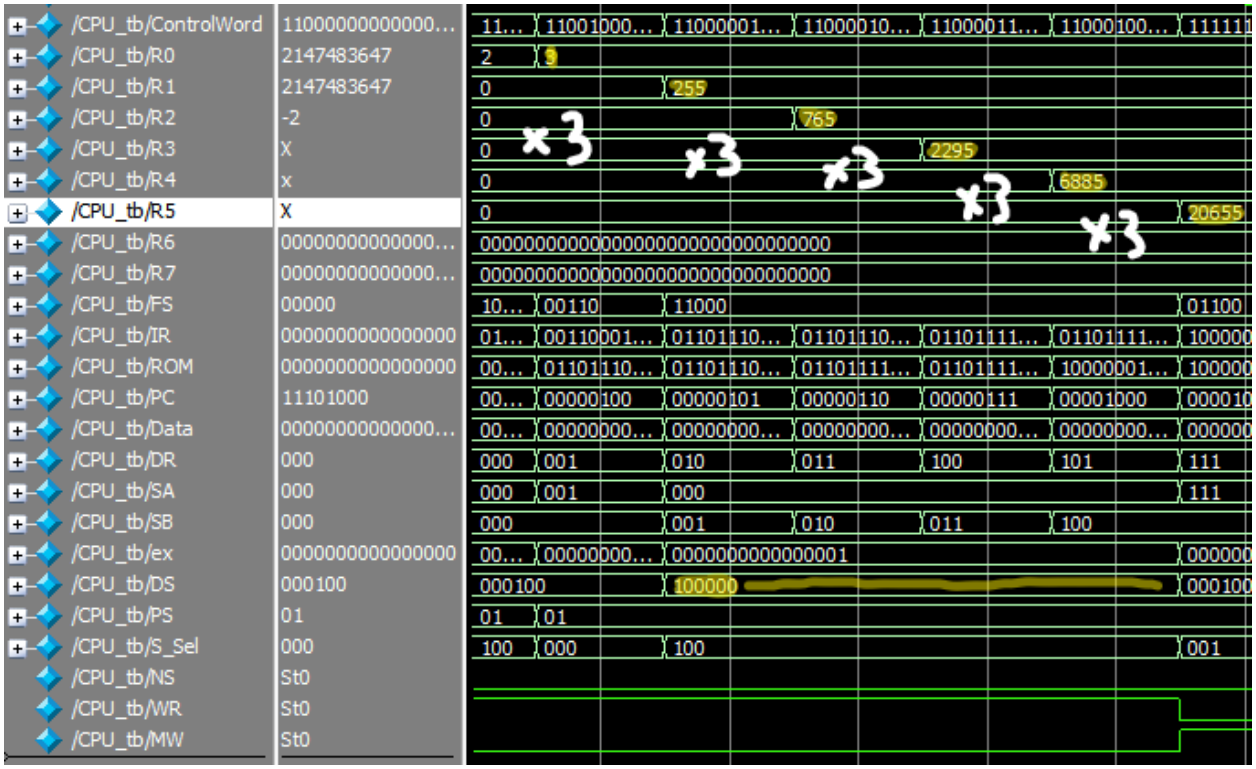


Adding two positive numbers, and the result is negative. This causes the overflow flag to be set.

Multiplier: My processor includes a separate hardware multiplier capable of multiplying two 16-bit numbers. However, this could output a number larger than 16 bits, up to 32-bits. Because of this, the registers in the register file were increased to 32 bits to accommodate.

This issue could have been avoided by implementing an 8x8 multiplier instead of a 16x16 multiplier. The multiplier was built as a schematic using AND gates, half-adders, and full-adders. In the datapath, it has a separate output select apart from the ALU. Because of the 16-bit increase in size, I either concatenated or increased the different signals connected to the data bus in order for everything to match up width wise.

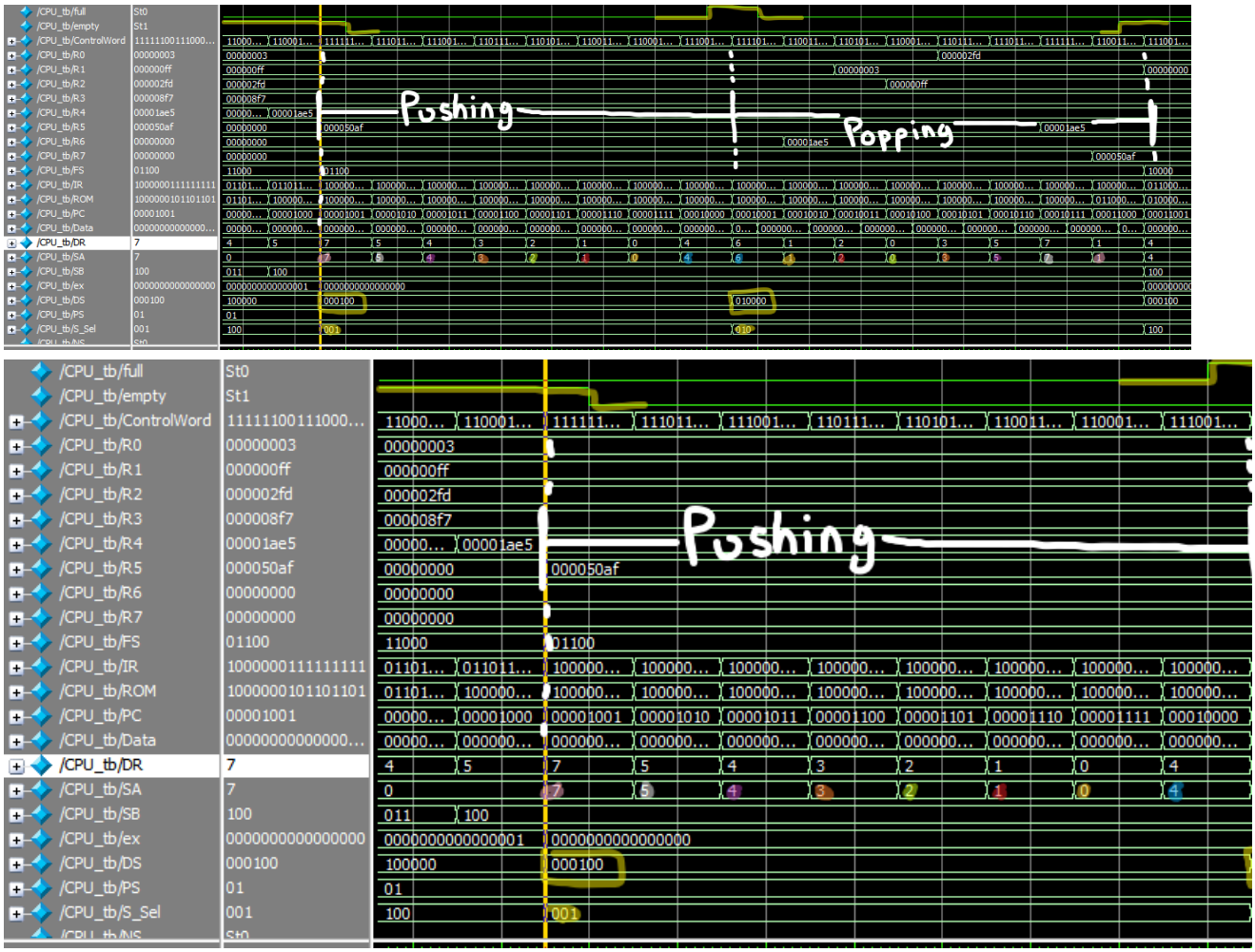
Figure 15-3: Multiplier Test



Here, 3 and 255 (decimal) are multiplied together and the result is then multiplied by 3 and so on. The DS signal is selecting the output of the multiplier and everything is working properly. The schematic for the multiplier can be found **HERE**.

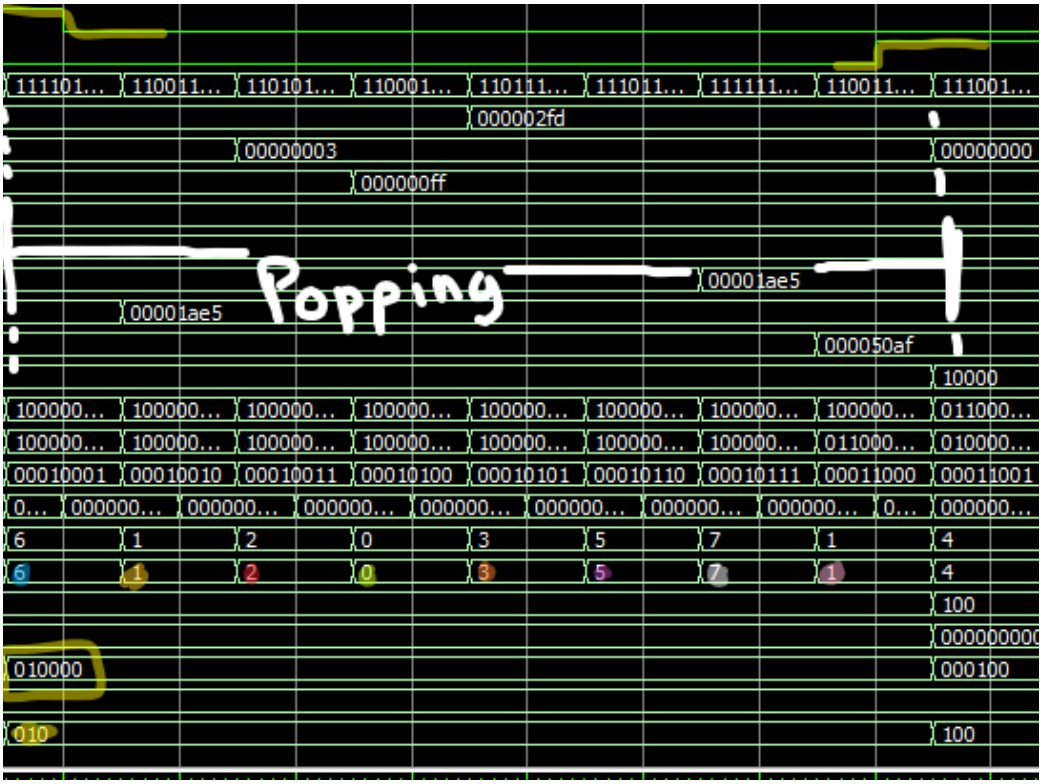
Stack: I managed to implement a hardware stack for my processor. The stack will be utilized for the advanced instructions PUSH, POP, CALL, and RET. It is controlled by a 3-bit signal inside of the control word. 001 will PUSH, 010 will POP, and 100 will reset / clear the stack. Inside the stack is a stack pointer that is able to increment or decrement depending on the instruction. Data is read and written based on the address of the stack pointer. It also has flags for when the stack is full and when it is empty, which also depends on the value of the stack pointer. The stack can be several different sizes (which also changes the size of the stack pointer), but for demonstration purposes it will be set to a smaller size (i.e 8 deep) to showcase the full and empty signals. Similarly to the RAM, the clock needed to be inverted in order for data to be available in time (single cycle). The stack in the finished processor is 128 deep, resulting in a 7-bit stack pointer. The Verilog code for the stack can be found **HERE**.

Figure 15-4: Stack Test



(Pushing zoomed in)

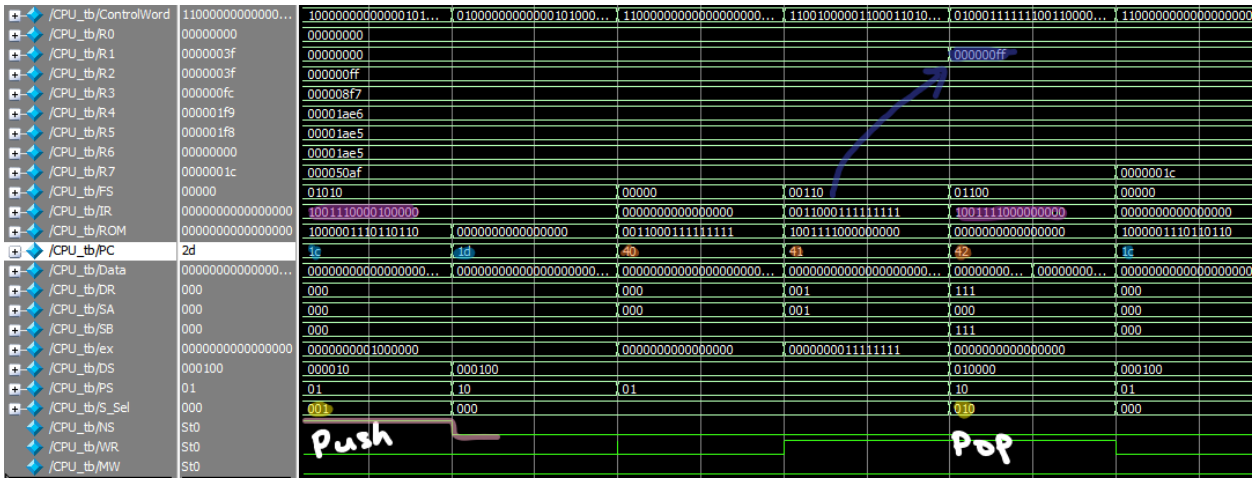
The stack is currently set to 8 deep, meaning that it can hold 8 values in 8 different places in the stack with a 3-bit stack pointer. First, we can see that the empty signal is high at first before going low when data starts to be pushed onto the stack. In the tail end of this zoomed view you can see the full signal go high once 8 values have been pushed. SA determines what value is getting pushed, and SA for each instruction is highlighted in a different color which will match its corresponding destination once popped in the next zoomed view. Whatever is pushed onto the stack last is also the first to come off (last in first out). The final thing to note here is that stack select is 001, signifying that the stack is pushing.

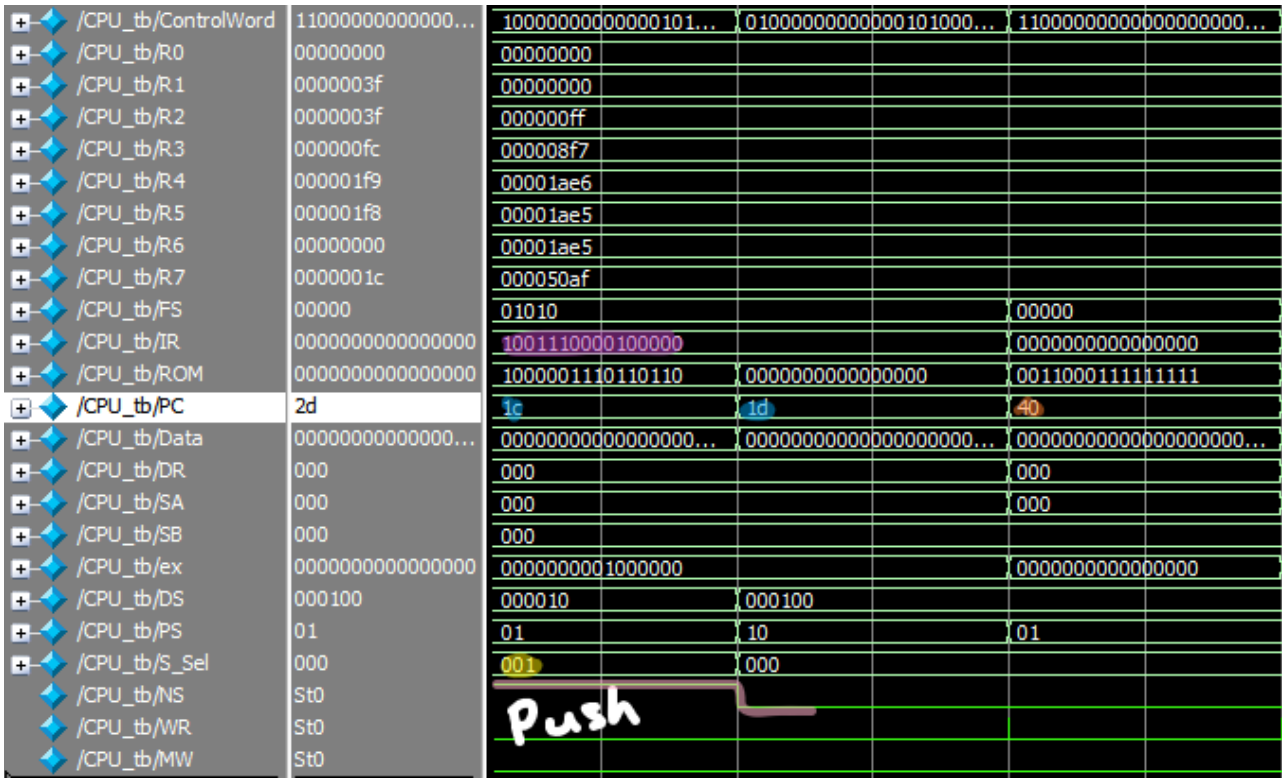


(Popping zoomed in)

There are a few key things happening here. Stack select is now 010, meaning the stack will be popping data. Although not visible here, write is now set to 1, and DS is selecting the output of the stack instead of the ALU. The DR signal determines where the data is being popped to, which is color coordinated with the corresponding data that was pushed. The top right shows the empty signal going high again after the stack pops all data off. Looking at the actual data, the last value that was pushed onto the stack was from R4 (1ae5 hex), that is the first value that popped from the stack, which is written to DR R6. The rest of the simulation follows the same rule, and the most recent pushed data pops off first.

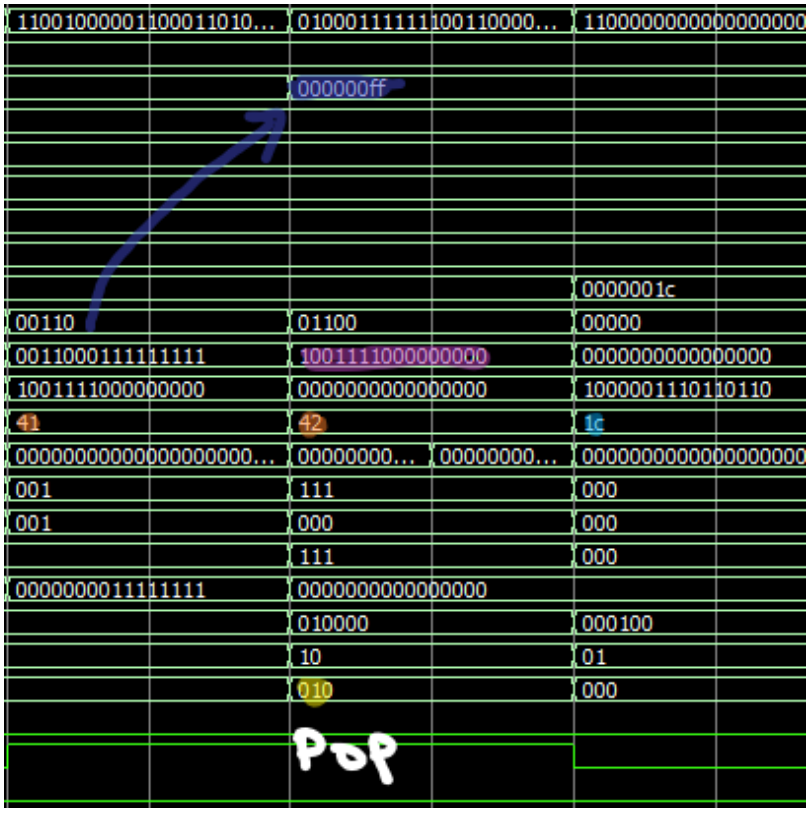
Figure 15-5: Call and Return





(Left Side Zoom)

This program tests the functionality of the CALL and RET instructions. Here, I'm calling a subroutine at address 40 (hex), where a literal value will be XOR'ed with a register value. After that, the program counter should jump back to where it was prior to the subroutine. It does so by pushing the program counter value onto the stack (with PC output selected). For my implementation, CALL is a two cycle instruction.



(Right Side Zoom)

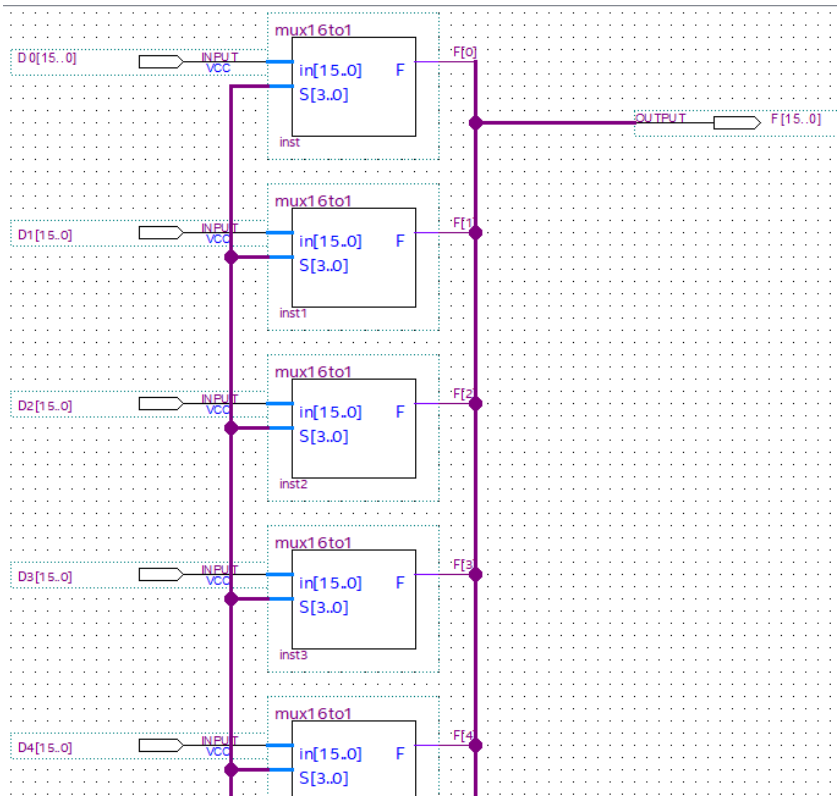
On the left, the XRI instruction happens as the PC counts up from 40. At address 42, the instruction register is executing the RET (return) instruction, which pops the previously stored program counter value onto the data bus. The program counter selects it set to 10, meaning it will take whatever is on the data bus as its own value. That's how the jump from 42 back to 1c happens.

Section 16: Appendix

Figure 16-1: Register File Decoder

```
module decoder3to8(in0, in1, in2, out0, out1, out2, out3, ou

input in0, in1, in2;
output reg out0, out1, out2, out3, out4, out5, out6, out7;


always @ (in2,in1,in0)
begin
   out0=0;
   out1=0;
   out2=0;
   out3=0;
   out4=0;
   out5=0;
   out6=0;
   out7=0;
 begin
 case ({in2,in1,in0})
   3'b000: out0=1;
   3'b001: out1=1;
   3'b010: out2=1;
   3'b011: out3=1;
   3'b100: out4=1;
   3'b101: out5=1;
   3'b110: out6=1;
   3'b111: out7=1;
 default: begin
   out0=0;
   out1=0;
   out2=0;
   out3=0;
   out4=0;
   out5=0;
   out6=0;
   out7=0;
  end
  endcase
 end
end
endmodule
```

Figure 16-2: Register File Mux



(there are 16 16 to 1's)

Figure 16-3: ALU Cell

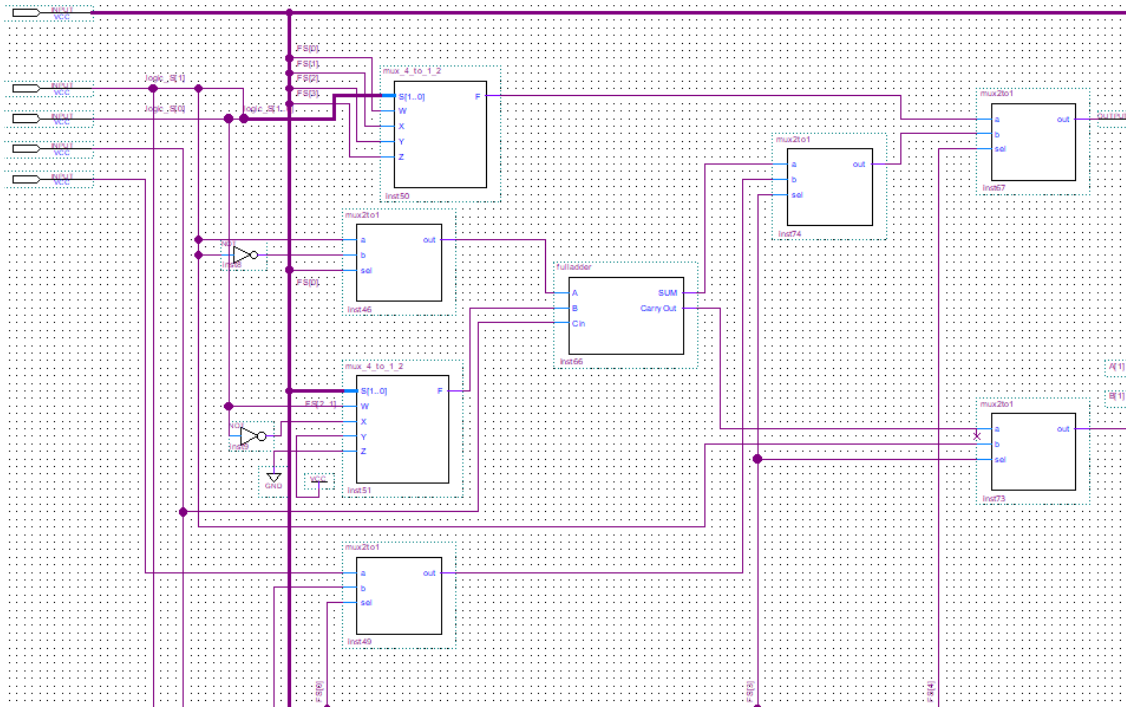


Figure 16-4: Instruction Register Verilog

```
module IR (clock, load, D_in, D_out);

    parameter n = 16;

    input clock, load;
    input [n-1:0]D_in;

    output reg [n-1:0]D_out;

    always @ (posedge clock) begin

          if(load) begin
          D_out <= D_in;
          end
       end
    end
endmodule
```

(Basically normal register)

Figure 16-5: Program Counter Verilog

```
module PC (clock, PS, PC_in, PC_offset, PC);

    input clock;
    input [15:0] PC_in;
    input [15:0] PC_offset;
    input [1:0]PS;

    output reg [15:0]PC;

    always @(posedge clock)begin
       case (PS)
          2'b00: PC = PC; // do nothing
          2'b01: PC = PC+1'b1; // increment
          2'b10: PC = PC_in; // program counter gets data from data bus
          2'b11: PC = PC+PC_offset; // program counter gets pc + offset
       endcase
    end
endmodule
```

Figure 16-6: Better View of Datapath (Jeremy)

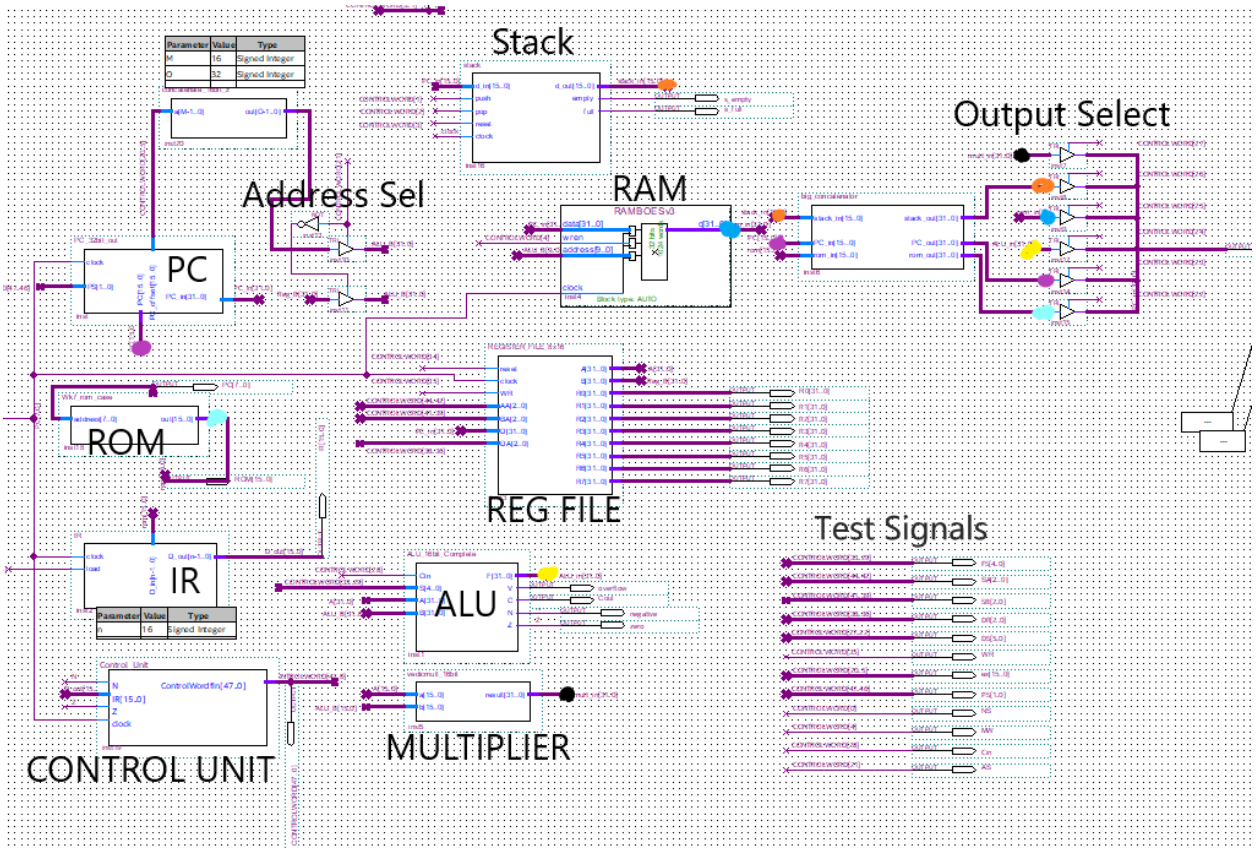


Figure 16-7: Clock Divider

```
module clock_divider(
    input wire clk,
    output reg div_clk
    );

    parameter div_value = 0; //div_value = 50Mhz / 2*desried freq) -1

    integer counter_value = 0;

    always@ (posedge clk)
    begin
        if (counter_value == div_value)
            counter_value <= 0;
        else
            counter_value <= counter_value + 1;

    end

    always@ (posedge clk)
    begin
        if (counter_value == div_value)
            div_clk <= ~div_clk;
        else
            div_clk <= div_clk;
        end
endmodule
```

Figure 16-8: VGA Horizontal Counter

```
module horizontal_counter(
   input clk_25,
   output reg enable_V_Counter =0,
   output reg [15:0] H_Count_value
   );

   always @(posedge clk_25) begin
      if (H_Count_value < 799) begin
          H_Count_value <= H_Count_value + 1;
          enable_V_Counter <= 0;
      end

      else begin
         H_Count_value <= 0;
         enable_V_Counter <=1;
      end
   end

endmodule
```

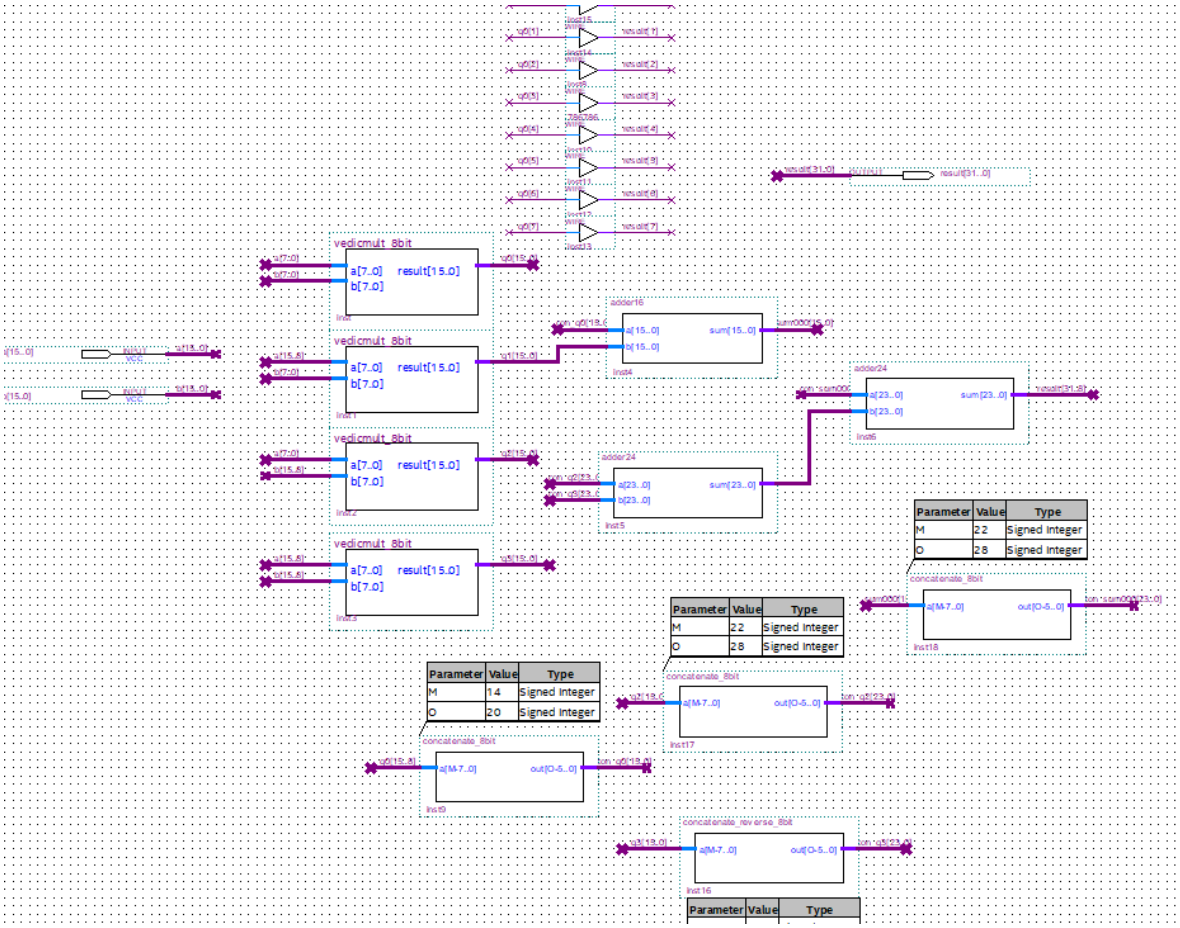
Figure 16-9: VGA Vertical Counter

```
module vertical_counter(
   input clk_25,
   input enable_V_Counter,
   output reg [15:0] V_Count_value =0
   );

   always @(posedge clk_25) begin
      if (enable_V_Counter == 1'b1) begin
         if (V_Count_value < 524)

            V_Count_value <= V_Count_value + 1;

         else V_Count_value <= 0;
      end

   end

endmodule
```

Figure 16-10: VGA Top

```
module vga_top(
    input clk,
    input [9:0]SW,
    input [1:0]KEY,
    output Hsync,
    output Vsync,
    output [3:0] Red,
    output [3:0] Green,
    output [3:0] Blue
    );

wire clk_25;
wire enable_V_Counter;
wire [15:0] H_Count_value;
wire [15:0] V_Count_value;
wire red_switch0, red_switch1, red_switch2, red_switch3;
wire green_switch0, green_switch1, green_switch2, green_switch3;
wire blue_switch0, blue_switch1, blue_button0, blue_button1;
wire [3:0] red_switches, green_switches, blue_switches;

clock_divider VGA_clock_gen (clk, clk_25);
horizontal_counter VGA_H (clk_25, enable_V_Counter, H_Count_value);
vertical_counter VGA_V (clk_25, enable_V_Counter, V_Count_value);

assign red_switch0 = SW[0];
assign red_switch1 = SW[1];
assign red_switch2 = SW[2];
assign red_switch3 = SW[3];

assign green_switch0 = SW[4];
assign green_switch1 = SW[5];
assign green_switch2 = SW[6];
assign green_switch3 = SW[7];

assign blue_switch0 = SW[8];
assign blue_switch1 = SW[9];
assign blue_button0 = ~KEY[0];
assign blue_button1 = ~KEY[1];

assign red_switches = {red_switch0, red_switch1, red_switch2, red_switch3};
assign green_switches = {green_switch0, green_switch1, green_switch2, green_switch3};
assign blue_switches = {blue_switch0, blue_switch1, blue_button0, blue_button1};

assign Hsync = (H_Count_value < 96) ? 1'b1: 1'b0;
assign Vsync = (V_Count_value < 2) ? 1'b1: 1'b0;

assign Red = (H_Count_value < 784 && H_Count_value > 143 && V_Count_value < 515 && V_Count_value > 34) ? red_switches : 4'h0;
assign Green = (H_Count_value < 784 && H_Count_value > 143 && V_Count_value < 515 && V_Count_value > 34) ? green_switches : 4'h0;
assign Blue = (H_Count_value < 784 && H_Count_value > 143 && V_Count_value < 515 && V_Count_value > 34) ? blue_switches : 4'h0;

endmodule
```

Figure 16-11: Multiplier Schematic



(Started with 2 bit multiplier, worked up to 16 bit using different sized adders)

Figure 16-12: Stack Verilog (1)

```
module stack(d_in, d_out, push, pop, reset, empty, full, clock);
input [15:0]  d_in;
input push, pop, reset, clock;
output reg empty, full; // signals that determine if the stack is empty or full
output reg [15:0] d_out;
reg [15:0] stack_mem[0:127]; // 128 deep, can be modified but must also change width of SP
reg  [6:0] SP; // stack pointer max value of 1111111 = 127
integer i;
always @ (posedge clock)
begin
    if (reset==1) begin
         SP       = 7'b1000000; // reset will set SP to this value
         empty    = SP[6]; // reset will empty stack so empty <= 1
         d_out  = 4'h0;
      for (i=0;i<128;i=i+1) begin
         stack_mem[i]= 0; // clearing stack
      end
     end
  else if (reset==0) begin
         full = SP ? 0:1;
```

Figure 16-13: Stack Verilog (2)

```
         empty  = SP[6];
         d_out = 16'h0000;
    if (full == 1'b0 && push == 1'b1) begin // not full and pushing
       SP             = SP-1'b1; // stack pointer decrements
       full           = SP ? 0:1; // check if full
       empty          = SP[6];
       stack_mem[SP] = d_in; // getting data based on SP address
     end
   else if (empty == 1'b0 && pop == 1'b1) begin // not full and popping
       d_out  = stack_mem[SP]; // outputs gets stack at SP address
       stack_mem[SP] = 0; // fill address with zero
       SP  = SP+1; // stack pointer increments
       full  = SP? 0:1; // check full
       empty  = SP[6]; // check empty
    end
   else;
```

: