Harvard Architecture, Tri-State Buses - Team I

Souder, Epifano, McGuire, Pranvoku, Drexel
Computer Architecture Spring 2017

July 21, 2017

1 Overview

B.E.S.T. (Binary Electronic System of Transistors) Processor is a 16 bit, Harvard architecture processor. The register file consists of eight 16-bit registers that allow data to be written to one destination register and read from two source registers at a time. The arithmetic logic unit is capable of bitwise logical operations NOT, AND, OR, XOR, NAND, and NOR. This component is also responsible for shifting register bits right and left, as well as the arithmetic operations addition, subtraction and negative.
The processor utilizes a Harvard architecture, meaning it uses separate buses for memory addresses and data, as well as separate memory blocks for program memory and random access memory. This architecture allows for an instruction execution and instruction fetch each clock cycle, making most instructions only take one cycle to complete.
As mentioned, there are two distinct memory blocks. Instruction sets are stored in a 512 word by a 16 bit read only memory (ROM) block. Additional memory which can be used for storage is available in the form of a 512 word by 16 bit random access memory (RAM) block. The first 17 addresses $(000)_{16}$ to $(010)_{16}$ are reserved for special function access. Primarily used for multi-cycle instruction execution, a 256 word by 16 bit hardware stack was implemented with full push and pop functionality.
Instructions are loaded from ROM by the instruction register. The instruction that is loaded is based on the value of the program counter, and after being loaded the instructions are decoded in the control unit from their 16 bit value from ROM into a 46 bit control word.
The max clock frequency attainable by this processor is 110Mhz.

Contents

2 Processor Architecture

The specific architecture utilized in this processor is commonly referred to as “Harvard Architecture”. The main feature of this type of architecture “includes an internal RAM where a portion of that internal RAM (herein called “common space”) can be used as either data space or instruction space. Additionally, the common space can be used as both data space and instruction space by partitioning into two parts.” [1] This results in more effective usage of the internal RAM. “However, because both instructions and data are fetched simultaneously under Harvard Architecture, the common space could receive conflicting data and instruction fetches. . . ” [1]

The other main component of this architecture is the use of tri-state buffers instead of muxes for selection. This reduces the amount of gate delay that it takes each selection signal to propagate. For muxes where we would have more than one bit, like our data bus mux, we had to select between 5 different things. We used 5 bits with 1-hot encoding to achieve the same effect as a mux would.
The clock used in for this processor comes from the 50 MHz crystal on the DE0 board.

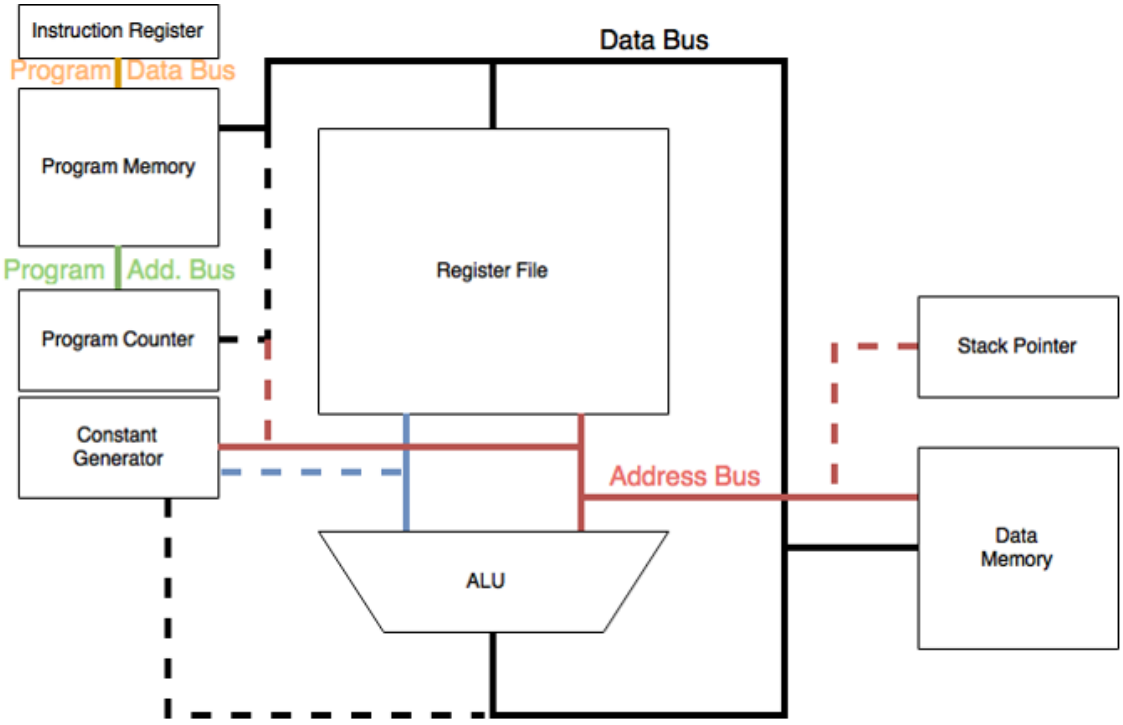

Figure 1: Abstracted View of our architecture

3 Register File

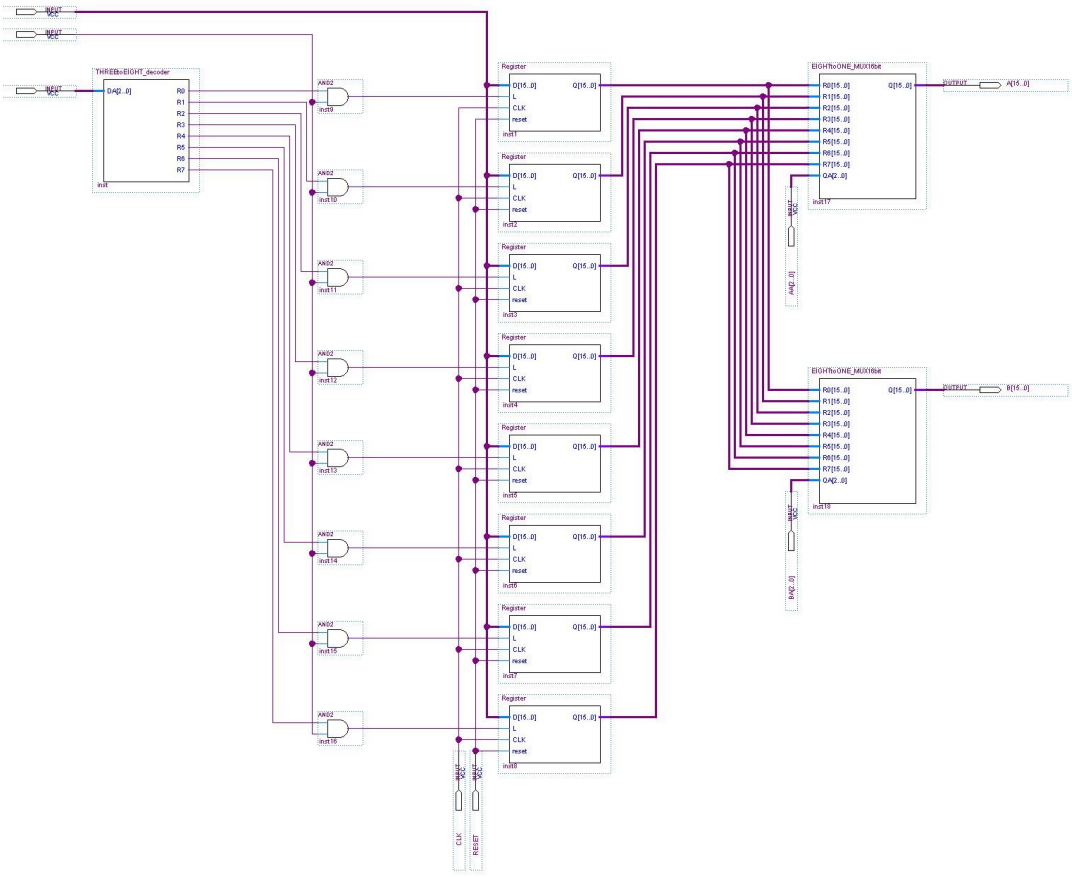

Figure 2: Schematic of Register File

The register file consists of eight 16 bit registers. A 3 to 8 decoder is used select the register to which incoming data will be written. The outputs of the decoder are ANDed with an enable signal (WR) which allows data to be written to the register file only when WR is high. Each register also has an asynchronous reset switch that sets the register to 16'b0 when the reset signal is high. The reset signal is tied to every register in the register file, so it will clear every register in the register file simultaneously. The output of each register is routed to two 8 to 1 multiplexers that allow two registers to be read simultaneously. The registers to be read are selected by 3 bit inputs SA and SB.

The inputs of the register file are clear, clock, write enable, 16 bit data in, 3 bits A,

B, and Destination address. The outputs of the register file are A and B output, all registers R0-R7 are output pins that way one could monitor the status of all registers.

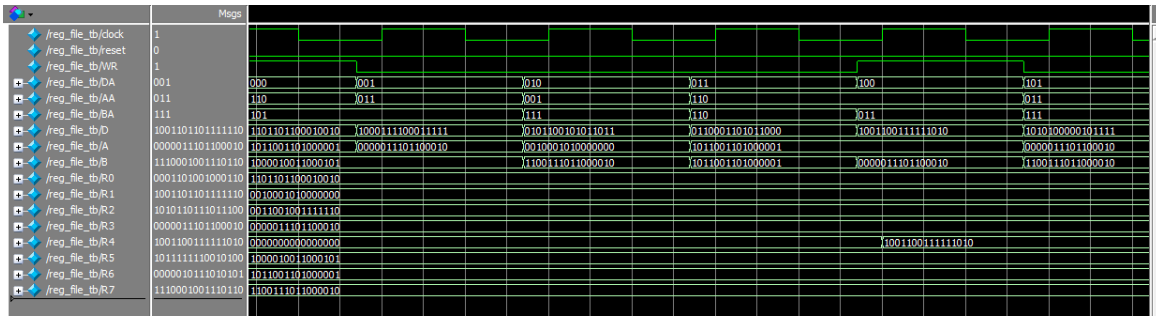

Figure 3: Simulation of Register File

This screen shot in Figure 3 shows that when the reset is low, and write is high the register selected by DR is loaded with the data in D. This can be seen as when write is high the selected registers are always loaded with the whole data set held in D.

4 ALU

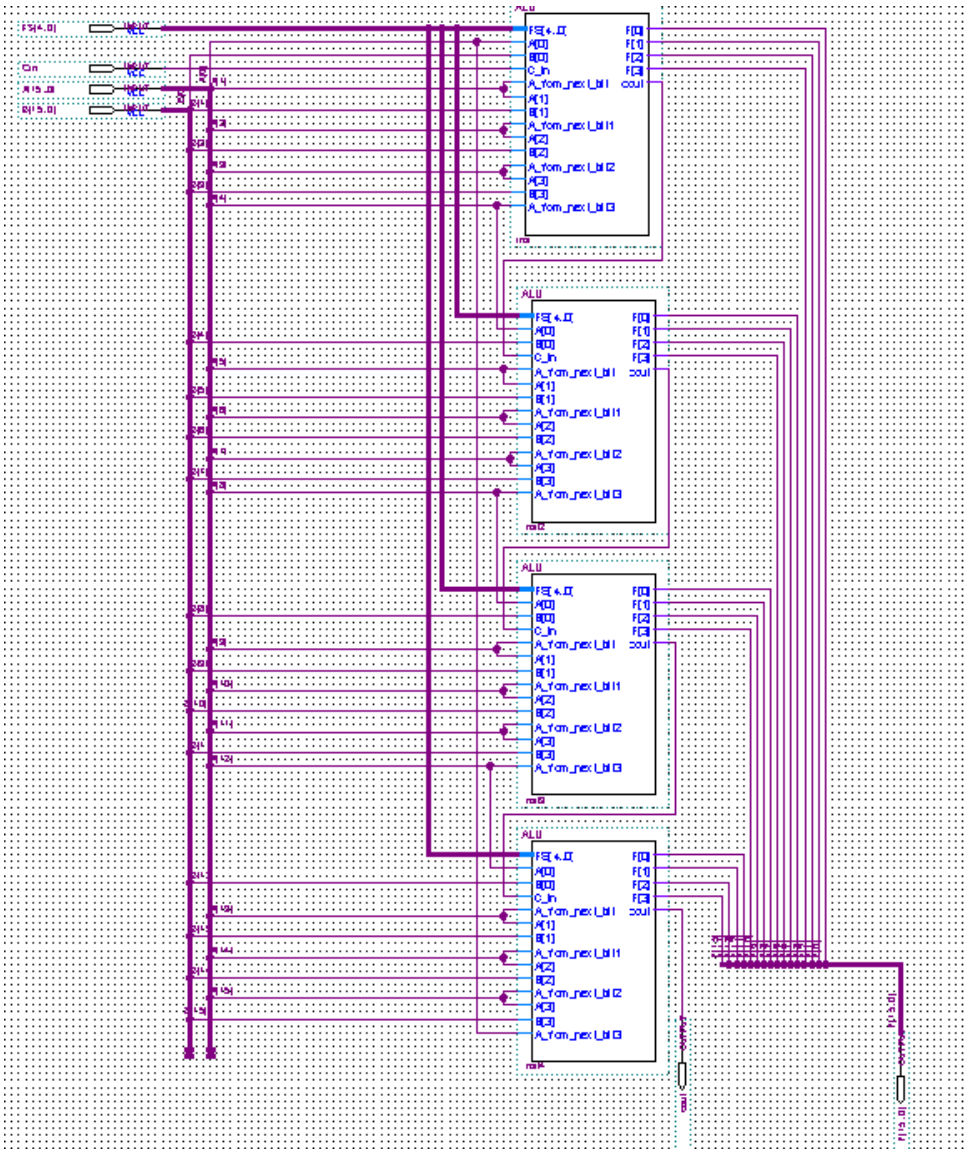

Figure 4: Schematic of the 16 bit ALU block

The Arithmetic Logic Unit has three primary functions- addition/ subtraction, logical operations (NOT, AND, NAND, OR, NOR, XOR), and right-bit-shifting. Right bit shift operations simply shifts the bits one place to the right. This 16 bit ALU utilizes a carry look ahead generator to calculate carried values in arithmetic operations in order to

increase the speed or the adder. The arithmetic unit can also perform subtraction by inverting the subtracted number and adding 1 to it. This operation yields the subtracted number's two's complement. The adder then adds the numbers normally. The operation can be expressed like this: A - B = A + (B' + 1) We used a 4-bit carry look ahead adder to reduce gate delays. The reason we only used 4 carry look aheads is because after this point the number of gates we would have to implement to generate the equations for each Cout will be about equal to or over the number of gate delays in a ripple carry. We then combined 4 in series to be used as a ripple-carry. This was how we chose to optimize our ALU.The individual 4 bit cell schematic can be seen in the appendix in Figure 22. The 1 bit schematic can be seen in the appendix in Figure 21.

4.1 Testing of ALU

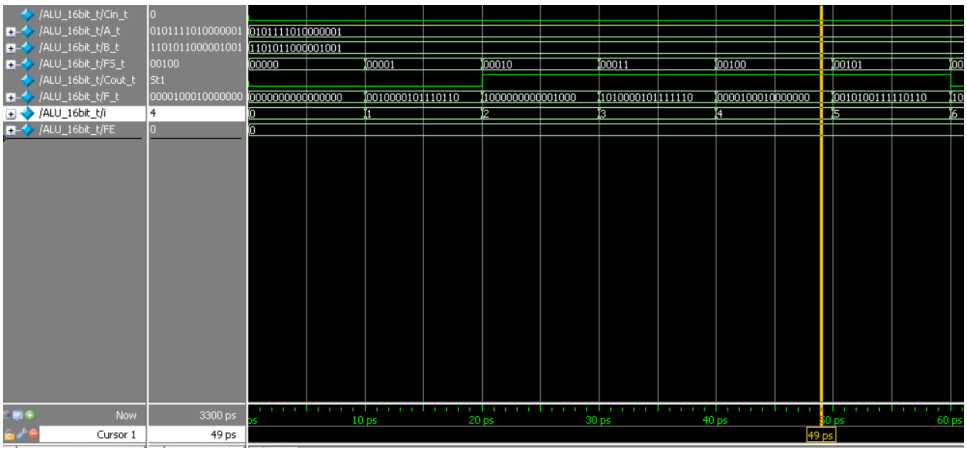

Figure 5: ALU simulation

```
            5'b11xx0: FE    =   (A_t<<1)+Ci
            5'b11xx1: FE    =   A_t>>1;
        endcase
        if (F_t!=FE) begin
            $display("FAILED %b", FS_t);
        end
    end
    initial begin
        #1000 $stop;
    end
endmodule
```

Figure 6: ALU Test bench

To test the ALU a test bench was used. This test bench took in the 5 Function Select Bits, and 16 bit A and B values. The test bench was then simulated with Altera ModelSim and it was found that all of the functions behaved as expected. To make verification even easier, an if statement was added to the test bench to confirm that all of the F outputs match the expected output for the input bits. If the output is not equal to the expected output, then ERROR would be printed to the transcript in ModelSim. As you can see from figure 7 no errors were printed, therefore the ALU preformed as expected.

```
view structure
.main_pane.structure.interior.cs.body.struct
view signals
.main_pane.objects.interior.cs.body.tree
run -all
Break in Module ALU_16bit_t at C:/Users/Jake/Documents/New-Comp-Arch-Processor/ALU_16bit_t.v line 66
Simulation Breakpoint: Break in Module ALU_16bit_t at C:/Users/Jake/Documents/New-Comp-Arch-Processor/ALU_16bit_t.v line 66
MACRO ./Register_File_run_msim_rtl_verilog.do PAUSED at line 21
WARNING: No extended dataflow license exists
```

Figure 7: Transcript for ModelSim simulation of the ALU test bench

4.2 Function Select

Below is the full list of all ALU functions, as well as their respective FS bits and carry in value. Function select bits are decoded in the control unit and for all ALU operations are accompanied by an A address, B address, and destination address.

Function	FS4	FS3	FS2	FS1	FS0	Cin
Move 0	0	0	0	0	0	0
Move 0	0	0	0	0	0	1
NOR	0	0	0	0	1	0
NOR	0	0	0	0	1	1
ÃB	0	0	0	1	0	0
ÃB	0	0	0	1	0	1
MOV Ã	0	0	0	1	1	0
MOV Ã	0	0	0	1	1	1
AB̃	0	0	1	0	0	0
AB̃	0	0	1	0	0	1
Move B̃	0	0	1	0	1	0
Move B̃	0	0	1	0	1	1
XOR	0	0	1	1	0	0
XOR	0	0	1	1	0	1
NAND	0	0	1	1	1	0
NAND	0	0	1	1	1	1
AND	0	1	0	0	0	0
AND	0	1	0	0	0	1
NOT XOR	0	1	0	0	1	0
NOT XOR	0	1	0	0	1	1
Move B	0	1	0	1	0	0
Move B	0	1	0	1	0	1
NOT(AB̃)	0	1	0	1	1	0
NOT(AB̃)	0	1	0	1	1	1
Move A	0	1	1	0	0	0
Move A	0	1	1	0	0	1
NOT (ÃB)	0	1	1	0	1	0
NOT (ÃB)	0	1	1	0	1	1
OR	0	1	1	1	0	0
OR	0	1	1	1	0	1
Move1	0	1	1	1	1	0
Move 1	0	1	1	1	1	1
A	1	0	0	0	0	0
A+1	1	0	0	0	0	1
Ã	1	0	0	0	1	0
-A	1	0	0	0	1	1
A+1	1	0	0	1	0	0
A+2	1	0	0	1	0	1

-A	1	0	0	1	1	0
1-A	1	0	0	1	1	1
A+B	1	0	1	0	0	0
A+B+1	1	0	1	0	0	1
Ã+B	1	0	1	0	1	0
B-A	1	0	1	0	1	1
A+B̃	1	0	1	1	0	0
A-B	1	0	1	1	0	1
Ã+B̃	1	0	1	1	1	0
Ã-B**	1	0	1	1	1	1
Left Shift in 0	1	1	0	0	0	0
Left Shift in 1	1	1	0	0	0	1
Right Shift*	1	1	0	0	1	0
Right Shift*	1	1	0	0	1	1
Left Shift in 0	1	1	0	1	0	0
Left Shift in 1	1	1	0	1	0	1
Right Shift*	1	1	0	1	1	0
Right Shift*	1	1	0	1	1	1
Left Shift in 0	1	1	1	0	0	0
Left Shift in 1	1	1	1	0	0	1
Right Shift*	1	1	1	0	1	0
Right Shift*	1	1	1	0	1	1
Left Shift in 0	1	1	1	1	0	0
Left Shift in 1	1	1	1	1	0	1
Right Shift*	1	1	1	1	1	0
Right Shift*	1	1	1	1	1	1

5 Memory Organization

There are two distinct memory blocks. Instruction sets are stored in a 512x16 read only memory (ROM) block. Other information is stored in a 512x16 bit random access memory (RAM) block. The first 17 addresses $(000)_{16}$ to $(010)_{16}$ are reserved for special function access. Primarily used for multi-cycle instruction execution, a 256 word by 16 bit hardware stack was implemented with full push and pop functionality.
The main RAM was implemented by the Quartus mega function wizard. For 16-bit words the DE0 that we used called for a memory format of 16x512 of M9K RAM module.

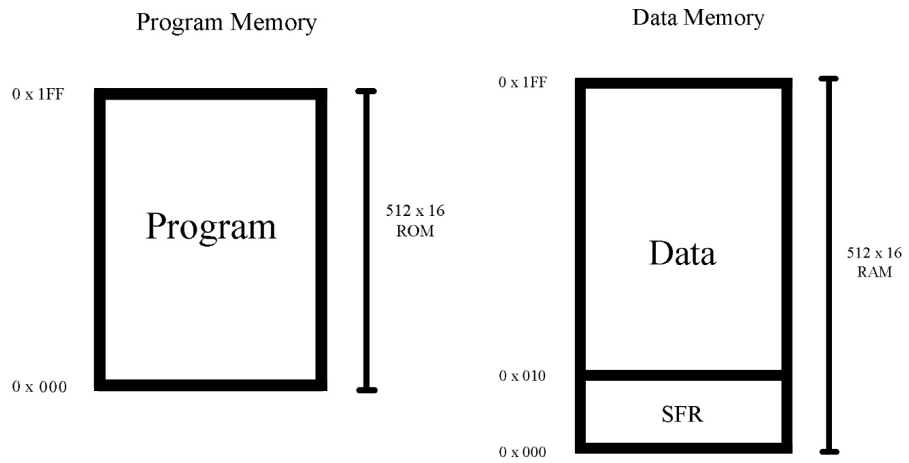

Figure 8: Map of random access and read only memories

5.1 Hardware Stack

The processor utilizes an accessible 256x16 hardware stack capable of “push” and “pop” functionality. This block is separate from the other two memories and is connected directly to the data bus. It is accessed using two control bits that are named Stack Select (SS). See Control Unit for more information regarding control signals.

5.2 Special Function Register Addresses

Our special function registers hold 11 places in memory. 10 are help by our GPIO pins and 1 is used for PS/2 data. In the figure below LED represents LEDs that are wired on a breadboard using the 2.5V from the GPIO pins. Buttons also follow the same method as they are wired off on a breadboard using the 2.5V GPIO pins. For PS/2, the data that is coming out of the module is held in memory address 8. When we want to load this data into the processor we just store load this value into a register using an LDI command.
See section 10 for more information on peripherals.

Address	Use
000	LED
001	LED
002	LED
003	LED
004	LED
005	LED
006	LED
007	LED
008	zf8 bit PS/2 output
009	Button
00A	Button
00B	Button
00C	Button
00D	Button
00E	Button
00F	Button
010	Button

6 Datapath

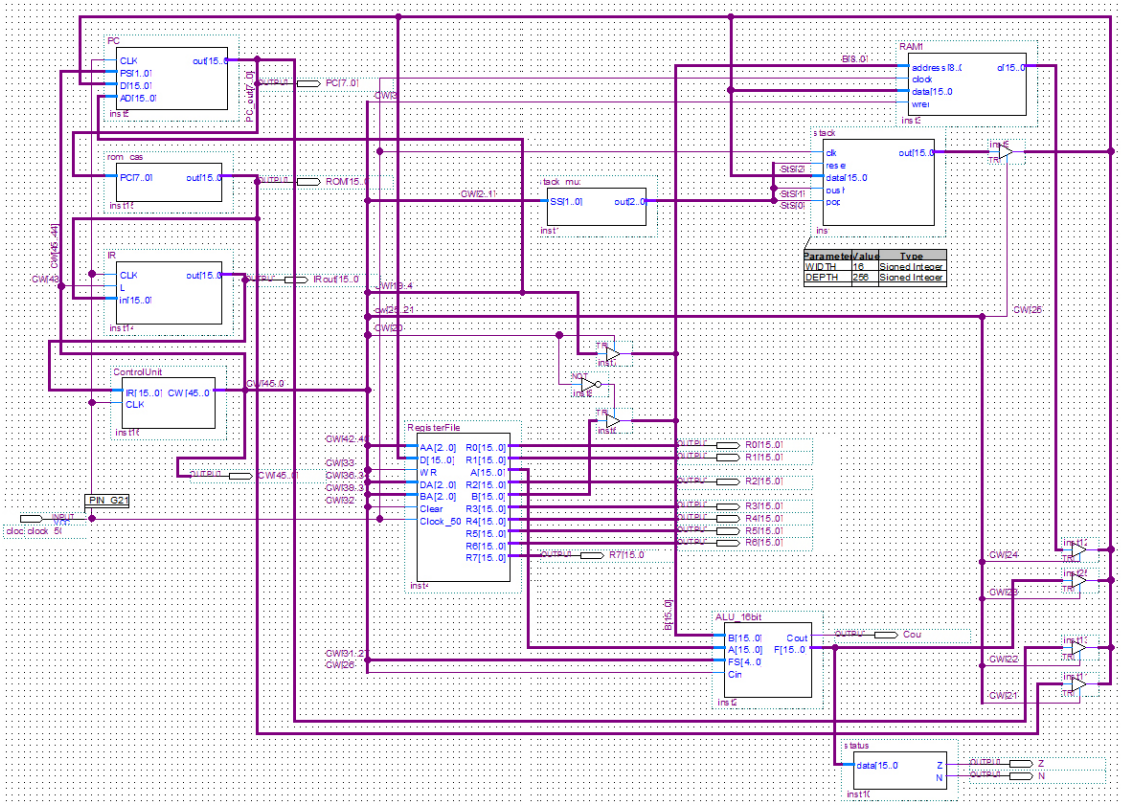

Figure 9: The paths along which data can flow (does not show control signals)

The datapath doesn't use any multiplexers, instead, implementing tristate buffers unseen in the figure. For example, in order to choose between the B address and our constant signal for the B input of the ALU, we have a single bit TriB signal connected to two tri state buffers. One of the buffers has a not gate in front of it. If TriB is on, the constant signal is passed to the databus. If TriB is off, the not gate enables the tri state buffer in front of the B address and passes that onto the data instead. In either case, only one signal is allowed on the data bus.
In order to choose between output from the stack, RAM, ALU, PC, and ROM, we selected between five tri-state buffers, using a 5 bit control signal MuxD. As stated in the architecture section, by using one hot encoding, we ensured that the data bus would only be getting data from one component. For example, if MuxD is 00100, only the tri-state buffer in front of the ALU is turned on, avoiding data collisions. See Control Unit for further detail regarding control signals.

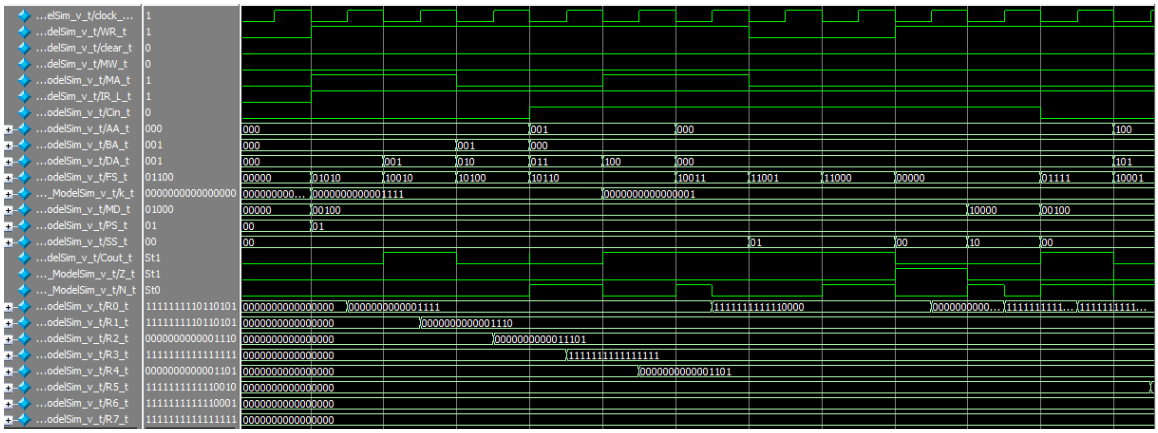

Figure 10: Simulation for Datapath

In this simulation you can see that all of the control variables are given by the test bench. With the full control word given the processor executes each command and the results are written to the registers.
The data path allows the ability to read and write from the register file in one clock cycle. This is utilized through different instructions such as SA <- SA + 1. The data path also allows for the use of the stack. This stack was designed to handle call and return functions.

7 Control Unit

The control unit is where 16 bit instructions are decoded into the 46 bit control word that is then released into data path and control components. The control unit was split into 4 different parts: $IR[15:14] = 00, 01, 10, 11$. This was make the design and debugging of module easier. The four decoders are then tied together into a mux and are selected based on the first two bits of the instruction register output.

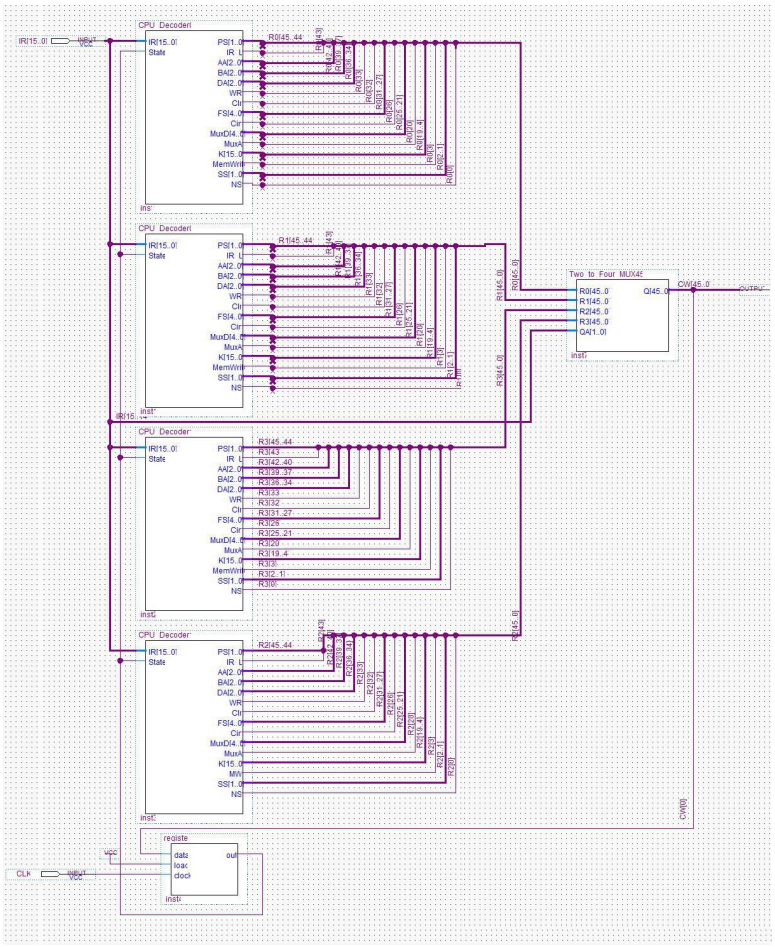

Figure 11: Schematic of the control unit

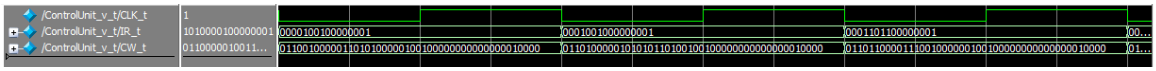

Figure 12: Simulation of the control unit

In figure 12 the first line represents then input which is the output of the instruction register. The second line represents the output of the control unit which is the 46-bit control word. This simulation shows that the IR input is correctly being decoded into a full length control word.
The control unit uses two status bits to make branch decisions such as branch on zero and branch on negative. These status bits come from the output of the ALU.
Below is the 46-bit control word and what each bit represents. PS[1] is the first bit and stack[0] is the last bit in this string of bits.

Signal	Description
PS[2]	Program select controls the program counter. See program counter for details.
IRL[1]	Controls if the instruction register is loaded.
SA[3]	Source register A address.
SB[3]	Source register B address.
DR[3]	Destination register address.
WR[1]	Controls if a register is written to.
Clear[1]	Clears registers.
FS[5]	Controls ALU function. See ALU for details.
C_{in}	Carry bit.
MUXD[4]	Selects from where data is loaded to the data bus. 10000 Stack output 01000 RAM output 00100 ALU output 00010 PC output 00001 ROM output
MUXA[1]	Selects between register B and constant K as ALU input.
K[16]	Constant 16 bit value.
MW[1]	Controls if RAM is being written to.
Stack[2]	Controls hardware stack function. 00 Do nothing 01 Push 10 Pop 11 Reset

7.1 Program Counter

The program counter takes the two-bit input PS (program select) from the control unit.
If PS = 00, program counter output remains the same.
If PS = 01, increment the PC because there is a new instruction.
If PS = 10, the data on the data bus goes to the PC.
If PS = 11, the PC is incremented by a constant value.

See the figure 24 in the appendix to see the verilog description of the program counter.

7.2 State Machine

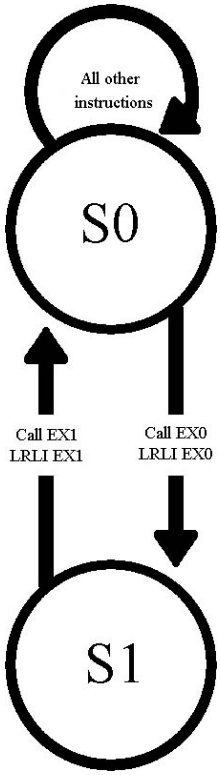

Figure 13: State machine diagram

As the figure shows, the processor only enters the S1 state on its two multi-cycle instructions: LRLI and CALL. All single cycle instructions keep the processor in state S0 and increment the program counter normally. LRLI and CALL take two cycles to complete and they increment the program counter on the first cycle only, as the processor transitions to the S1 state. The program counter is not incremented during the second cycle which transitions the processor back to state S0.

8 Instruction Set

8.1 Control Word

Operation	PS [2]	IR_L [1]	AA [3]	BA [3]	DA [3]	WR [1]	CLR R[1]	FS [5]	Cin [1]	MuxD [4]	MuxA [1]	K [16]	MW[1]	STACK [2]	STATE	NS	OPCODE
NOP	x	x	x	x	x	0	x	x	x	x	x	x	0	0	x	x	x
INC	1	1	IR5:3	X	IR8:6	1	0	10000	1	100	X	X	0	0	0	0	110000
ADD	1	1	IR5:3	IR2:0	IR8:6	1	0	10100	0	10	0	X	0	0	0	0	110100
ADDC	1	1	IR5:3	IR2:0	IR8:6	1	0	10100	1	100	0	X	0	0	0	0	110101
SUB	1	1	IR5:3	IR2:0	IR8:6	1	0	10110	1	100	0	X	0	0	0	0	110110
DEC	1	1	IR5:3	X	IR8:6	1	0	10110	1	100	1	16'b1	0	0	0	0	110010
NEG	1	1	IR5:3	X	IR8:6	1	0	10011	1	100	X	X	0	0	0	0	110001
SHR	1	1	IR5:3	X	IR:8:6	1	0	11001	X	100	X	X	0	0	0	0	111001
SHL	1	1	IR5:3	X	IR8:8	1	0	11000	X	100	X	X	0	0	0	0	111000
CLR	1	1	X	X	IR8:6	1	0	0	X	100	X	X	0	0	0	0	100000
SET	1	1	X	X	IR8:6	1	0	1111	0	100	X	X	0	0	0	0	101111
NOT	1	1	IR5:3	X	IR8:6	1	0	10001	X	100	X	X	0	0	0	0	100011
AND	1	1	IR5:3	IR2:0	IR8:6	1	0	1000	X	100	0	X	0	0	0	0	101000
OR	1	1	IR5:3	IR2:0	IR8:6	1	0	1110	X	100	0	X	0	0	0	0	101110
XOR	1	1	IR5:3	IR2:0	IR8:6	1	0	110	X	100	0	X	0	0	0	0	100110
MOVA	1	1	IR5:3	X	IR8:6	1	0	1100	X	100	X	X	0	0	0	0	101100
MOVB	1	1	X	IR2:0	IR8:6	1	0	1010	X	100	0	X	0	0	0	0	101010
ADDI	1	1	IR10:8	X	IR10:8	1	0	10100	0	100	1	zf(IR7:0)	0	0	0	0	1
SUBI	1	1	IR10:8	X	IR10:8	1	0	10110	1	100	1	zf(IR7:0)	0	0	0	0	10
ANDI	1	1	IR10:8	X	IR10:8	1	0	1000	X	100	1	zf(IR7:0)	0	0	0	0	11
ORI	1	1	IR10:8	X	IR10:8	1	0	1110	X	100	1	zf(IR7:0)	0	0	0	0	101
XORI	1	1	IR10:8	X	IR10:8	1	0	110	X	100	1	zf(IR7:0)	0	0	0	0	110
LRI	1	1	X	X	IR13:11	1	0	1010	X	100	1	zf(IR7:0)	0	0	0	0	11
LRLI (EX0)	1	0	X	X	IR[8:6]	1	0	X	X	1	X	X	0	0	0	1	1000010
LRLI (EX1)	1	1	X	X	IR8:6	0	0	1010	X	100	1	IR15:0	0	0	1	0	1000010
LDI	1	1	X	X	IR10:8	1	0	1010	X	1000	1	zf(IR7:0)	0	0	0	0	10100
STI	1	1	IR10:8	X	X	0	0	1100	X	100	1	zf(IR7:0)	1	0	0	0	10101
PUSH	1	1	IR 5:3	X	X	0	0	1100	X	100	X	X	0	1	0	0	1000000
POP	1	1	X	X	IR8:6	1	0	X	X	10000	X	X	0	10	0	0	1000001
STR	1	1	IR8:6	IR2:0	X	0	0	1100	X	100	0	X	1	0	0	0	1000101
LDR	1	1	X	IR2:0	IR8:6	1	0	X	X	1000	0	X	0	0	0	0	1000100
CALL (EX0)	1	0	X	X	X	0	0	X	X	10	X	X	X	1	0	1	1001110
CALL (EX1)	10	1	X	X	X	0	0	1010	X	100	1	zf(IR8:0)	0	0	1	0	1001110
RET	10	1	X	X	X	0	0	X	X	10000	X	X	0	10	0	0	1001111
BRZ False	1	1	x	x	x	0	x	x	x	x	x	x	0	x	x	x	N/A
BRZ True	11	1	IR10:8	X	X	0	0	1100	X	100	1	IR[7:0]	0	0	0	0	10110
BRN False	1	1	x	x	x	0	x	x	x	x	x	x	x	x	x	x	N/A
BRN True	11	1	IR10:8	X	X	0	0	1100	X	100	1	IR[7:0]	0	0	0	0	10111
jumpr	11	1	X	X	X	0	0	1010	X	100	1	IR[8:0]	0	0	0	0	1001101

Figure 14: Control word table

8.2 Instruction Set Summary

Mnemonic/Operands	Description	Cycles	Op code
NOP/NA	Do nothing	1	0000000
INC/DR, SA	Increments SA by 1 and stores into DR.	1	0110000
ADD/DR, SA, SB	Adds SA and SB, stores in DR.	1	0110100
ADDC/DR, SA, SB	Adds SA and SB with carry, stores in DR.	1	0110101
SUB/DR, SA, SB	Subtracts SB from SA, stores in DR.	1	0110110
DEC/DR, SA	Decrements SA and stores in DR.	1	0110010
NEG/DR, SA	Negate SA and store it in DR.	1	0110001
SHR/DR, SA	Shifts SA right one bit. Store in DR.	1	0111001
SHL/DR, SA	Shifts SA left one bit. Store in DR.	1	0111000
CLR/DR	Clear bits in DR.	1	0100000
SET/DR	Sets bits in DR.	1	0101111
NOT/DR, SA	Logic NOT of SA, store in DR.	1	0100011
AND/DR, SA, SB	Logic AND of SA and SB. Store in DR.	1	0101000
OR/DR, SA, SB	Logic OR of SA and SB. Store in DR.	1	0101110
XOR/DR, SA, SB	Logic XOR of SA and SB. Store in DR.	1	0100110
MOVA/DR, SA	Move SA to DR.	1	0101100
MOVB/DR, SB	Move SB to DR.	1	0101010
ADDI/DR, SA, K	Add SA and K, store in DR.	1	00001
SUBI/DR, SA, K	Subtract K from SA, store in DR.	1	00010
ANDI/DR, SA, K	AND SA and K, store in DR.	1	00011
ORI/DR, SA, K	OR SA and K, store in DR.	1	00101
XORI/DR, SA, K	XOR SA and K, store in DR.	1	00110
LRI/DR, K	Loads K to DR.	1	11
LRLI(EX0)/DR	Execute state 0 of LRLI.	2	1000010
LRLI(EX1)/K	Load long literal K to DR.	2	1000010
LDI/DR, K	Load memory at address K to DR.	1	10100
STI/SA, K	Store SA to memory at address K.	1	10101
PUSH/SA	Push SA on to stack.	1	1000000
POP/DR	POP stack to DR.	1	1000001
STR/DR, SB	Store SB to memory at address DR.	1	1000101
LDR/DR, SB	Load DR with memory at address DR.	1	1000100
CALL(EX0)/ -	Call a subroutine.	2	1001110
CALL(EX1)/K	Second cycle of subroutine call.	2	1001110
RET/-	Return from a subroutine.	1	1001111
BRZ/SA, K	Branch to PC + K if SA is 0.	1	10110
BRN/SA, K	Branch to PC + K if SA is negative.	1	10111
JUMPR	Jump to PC = SA.	1	1001101

8.3 Instruction List

Below is a full list of compatible instructions with descriptions, assembler syntax, a list of operand parameters, and RTL operations.

NOP	No Operation	INC	Increment
Syntax	NOP	Syntax	INC DR, SA
Operands	N/A	Operands	$0 \leq DR, SA \leq 7$
Operation	N/A	Operation	$R[DR] \leftarrow R[SA] + 1$
Description	Do nothing but still increment the program counter.	Description	Add 1 to value in source register A and store it to destination register.
DEC	Decrement	ADD	Addition
Syntax	DEC DR, SA	Syntax	ADD DR, SA, SB
Operands	$0 \leq DR, SA \leq 7$	Operands	$0 \leq DR, SA, SB \leq 7$
Operation	$R[DR] \leftarrow R[SA] - 1$	Operation	$R[DR] \leftarrow R[SA] + R[SB]$
Description	Do nothing but still increment the program counter.	Description	Add values in source registers A and B and store it to destination register.
SUB	Subtraction	ADDC	Addition with Carry
Syntax	DEC DR, SA, SB	Syntax	ADDC DR, SA, SB, C_{in}
Operands	$0 \leq DR, SA, SB \leq 7$	Operands	$0 \leq DR, SA, SB, C_{in} \leq 7$
Operation	$R[DR] \leftarrow R[SA] - R[SB]$	Operation	$R[DR] \leftarrow R[SA] + R[SB]$
Description	Subtract value in source register B from value in source register A. Store in destination register.	Description	Add values in source registers A and B and C_{in}. Store to destination register.
NEG	Negate	SHR	Shift Right

Syntax	NEG DR, SA	Syntax	SHR DR, SA
Operands	$0 \leq DR, SA \leq 7$	Operands	$0 \leq DR, SA \leq 7$
Operation	$R[DR] \leftarrow -R[SA]$	Operation	$R[DR] \leftarrow srR[SA]$
Description	Store the two's compliment of soucre register A's value to destination register.	Description	Shift bits in source register A to the right. Store in destination register.

SHL	Shift Left	CLR	Clear Register
Syntax	SHL DR, SA	Syntax	CLR DR
Operands	$0 \leq DR, SA \leq 7$	Operands	$0 \leq DR \leq 7$
Operation	$R[DR] \leftarrow slR[SA]$	Operation	$R[DR] \leftarrow 0$
Description	Shift bits in source register A to the left, store in destination register.	Description	Set bits in destination register to 0.

SET	Set Register	NOT	Logic NOT
Syntax	SHL DR	Syntax	CLR DR, SA
Operands	$0 \leq DR \leq 7$	Operands	$0 \leq DR, SA \leq 7$
Operation	$R[DR] \leftarrow 1$	Operation	$R[DR] \leftarrow\sim R[SA]$
Description	Sets bits in destination register to 1.	Description	Logic NOT source register A store in destination register.

AND	Logic AND	OR	Logic OR
Syntax	AND DR, SA, SB	Syntax	OR DR, SA, SB
Operands	$0 \leq DR, SA, SB \leq 7$	Operands	$0 \leq DR, SA, SB \leq 7$
Operation	$R[DR] \leftarrow R[SA] \wedge R[SB]$	Operation	$R[DR] \leftarrow R[SA] \vee R[SB]$
Description	Logic AND of source registers A and B store in destination register.	Description	Logic OR of source registers A and B store in destination register.

XOR	Logic Exclusive OR	MOVA	Move SA
Syntax	XOR DR, SA, SB	Syntax	MOVA DR, SA

Operands	$0 \leq DR, SA, SB \leq 7$	Operands	$0 \leq DR, SA \leq 7$
Operation	$R[DR] \leftarrow R[SA] \oplus R[SB]$	Operation	$R[DR] \leftarrow R[SA]$
Description	Logic XOR source registers A and B store in destination register.	Description	Load source register A into destination register.

MOVB	Move SB	ADDI	Add immediate
Syntax	MOVA DR, SB	Syntax	ADDI DR, SA, K
Operands	$0 \leq DR, SB \leq 7$	Operands	$0 \leq DR, SB \leq 7$ $0 \leq K \leq 255$
Operation	$R[DR] \leftarrow R[SB]$	Operation	$R[DR] \leftarrow R[SA] + zf_K$
Description	Load source register B to destination register.	Description	Add source register A with zero-filled literal K, store in destination register.

SUBI	Subtract immediate	ANDI	And immediate
Syntax	SUBI DR, SA, K	Syntax	ANDI DR, SA, K
Operands	$0 \leq DR, SA \leq 7$ $0 \leq K \leq 255$	Operands	$0 \leq DR, SA, SB \leq 7$ $0 \leq K \leq 255$
Operation	$R[DR] \leftarrow R[SA] \min zf_K$	Operation	$R[DR] \leftarrow R[SA] \wedge zf_K$
Description	Subtract source register A with zero-filled literal K, store in destination register.	Description	Logical AND of source register A and zero-filled literal K, store in destination register.

ORI	Or immediate	XORI	Exclusive OR with immediate
Syntax	ORI DR, SA, K	Syntax	XORI DR, SA, K
Operands	$0 \leq DR, SB \leq 7$ $0 \leq K \leq 255$	Operands	$0 \leq DR, SA \leq 7$ $0 \leq K \leq 255$
Operation	$R[DR] \leftarrow R[SA] \vee zfK$	Operation	$R[DR] \leftarrow R[SA] \vee zfK$
Description	Logical bitwise OR of source	Description	Logical bitwise XOR of source

	register A and zero-filled literal K, store in destination register.		register A and zero-filled literal K, store in destination register.
LRI	Load register immediate	LRLI	Load register long immediate
Syntax	LRI DR, K	Syntax	OR DR, SA, SB
Operands	$0 \leq DR \leq 7$ $0 \leq K \leq 255$	Operands	$0 \leq DR \leq 7$ $0 \leq K \leq 65535$
Operation	$R[DR] \leftarrow zf_K$	Operation	$R[DR] \leftarrow K$
Description	Logic zero-filled literal K to destination register.	Description	Load register with unsigned long literal K in the EX0 state.
LDI	Load data at immediate		
Syntax	LDI DR, K		
Operands	$0 \leq DR \leq 7$ $0 \leq K \leq 255$		
Operation	$R[DR] \leftarrow M[K]$		
Description	Load destination register with data from memory at literal address value.		
STI	Store immediate	PUSH	Hardware stack push
Syntax	STI, K, SA	Syntax	PUSH SA
Operands	$0 \leq DR, SA, SB \leq 7$ $0 \leq K \leq 255$	Operands	$0 \leq SA \leq 7$
Operation	$M[k] \leftarrow R[SA]$	Operation	$PUSHR[SA]$
Description	Store data from source register A to memory at immediate address.	Description	Stack operation ”Push” data at source register A to the hardware stack.
POP	Hardware stack pop	STR	Store register to memory
Syntax	POP DR	Syntax	STR DR, SA
Operands	$0 \leq DR \leq 7$	Operands	$0 \leq DR, SA \leq 7$

Operation	$R[DR] \leftarrow POP$	Operation	$M[R[DR]] \leftarrow R[SA]$
Description	Stack operation "Pop" data off the hardware stack to destination register.	Description	Store data from source register A to memory at address in destination register.

LDR	Load from memory to register	CALL	Call
Syntax	LDR DR, SA	Syntax	CALL K
Operands	$0 \leq DR, SA \leq 7$	Operands	$0 \leq K \leq 255$
Operation	$R[DR] \leftarrow M[R[SA]]$	Operation	$PUSHPC + 1$ $PC \leftarrow zf_K$
Description	Load data from memory address stored in source register A and store it in destination register.	Description	Call subroutine, push program counter plus one to the stack. Must execute NOP two instructions after.

JMPR	Jump to address in register.	RET	Return from subroutine
Syntax	JMPR SA	Syntax	RET
Operands	$0 \leq SA \leq 7$	Operands	N/A
Operation	$PC \leftarrow R[SA]$	Operation	$PC \leftarrow POP$
Description	Jump to memory at address stored in source register A.	Description	End subroutine. Pop stack and put it into PC.

BRZ	Branch if zero	BRN	Branch if negative
Syntax	BRZ SA, K	Syntax	BRN SA, K
Operands	$0 \leq SA \leq 7$ $0 \leq K \leq 255$	Operands	$0 \leq SA \leq 7$ $0 \leq K \leq 255$
Operation	$if(R[SA] == 0)$ $PC \leftarrow PC + se_K$	Operation	$if(R[SA] < 0)$ $PC \leftarrow PC + se_K$
Description	The program counter is incremented by a sign extended literal if source register A is 0.	Description	The program counter is incremented by a sign extended literal if source register A is negative.

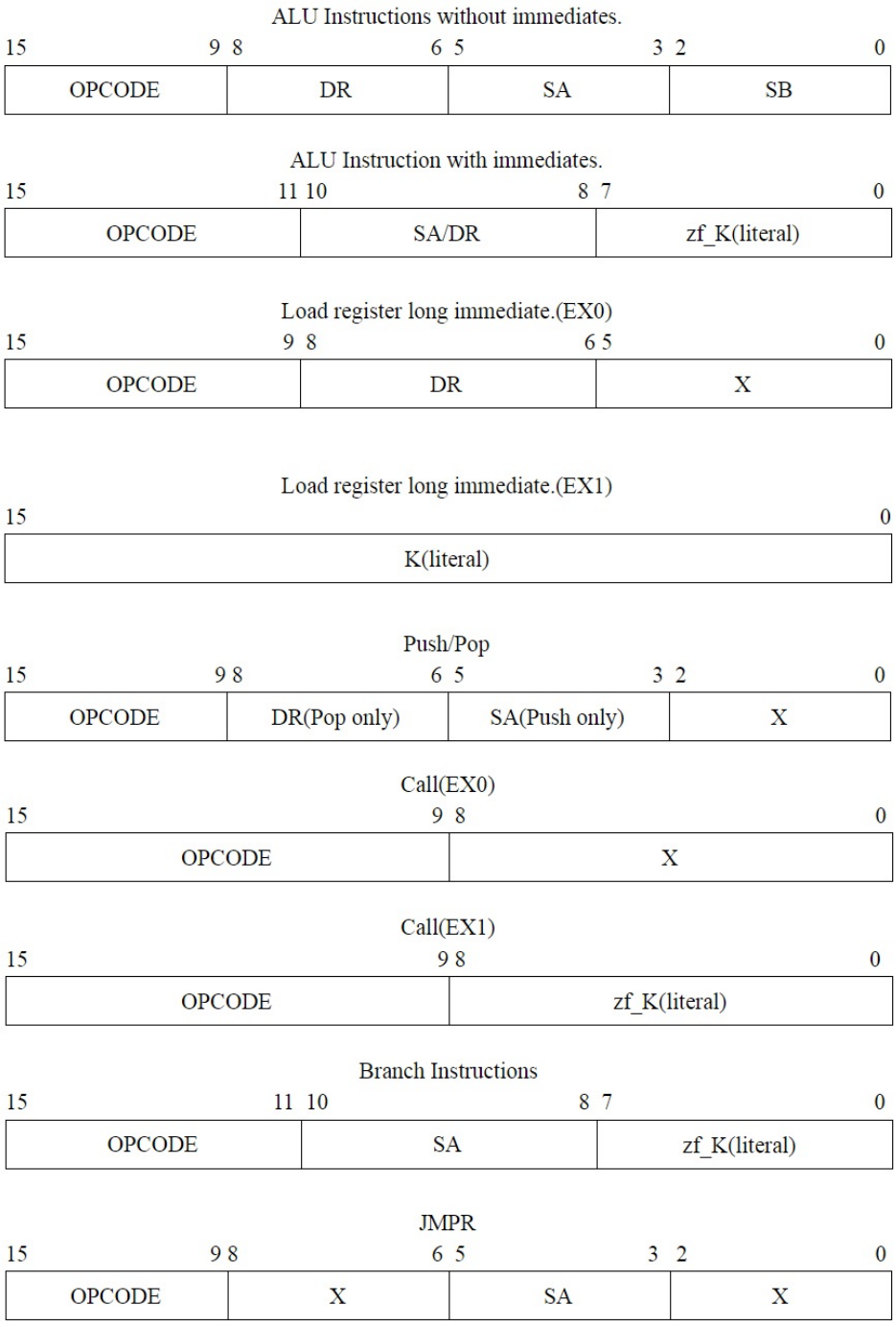

Figure 15: Instruction format per Instruction

9 CPU

9.1 Design and Features

This processor utilizes a number of advanced instructions including ADDC, STR, LDR, PUSH, POP, CALL and Return.
This processor allows an instruction execution and instruction fetch each clock cycle, making most instructions only take one cycle to complete. However, this processor does feature two multi-cycle instructions - Call and LRLI. The hardware stack is used to implement the call and return instructions. The call function pushes the PC + 1 on to the stack. Then when return is called it will be popped back onto the program counter to return from the subroutine.
The processor is also capable of storing data from a register to a memory address specified by another register using the STR instruction, as well as loading in the same way using the LDR instruction.
Instructions are loaded from ROM into the instruction register. The instruction that is loaded is based on the value of the program counter, and after being loaded the instructions are decoded in the control unit from their 16 bit value from ROM into a 46 bit control word. This control word is used to manipulate the data path to perform the actions which the instruction calls for.
The max clock frequency was tested using the PS/2 program. Any clock speed over 110MHz corrupted the output data, and the program would no longer work as intended. See section 10.2 for more information on this program.

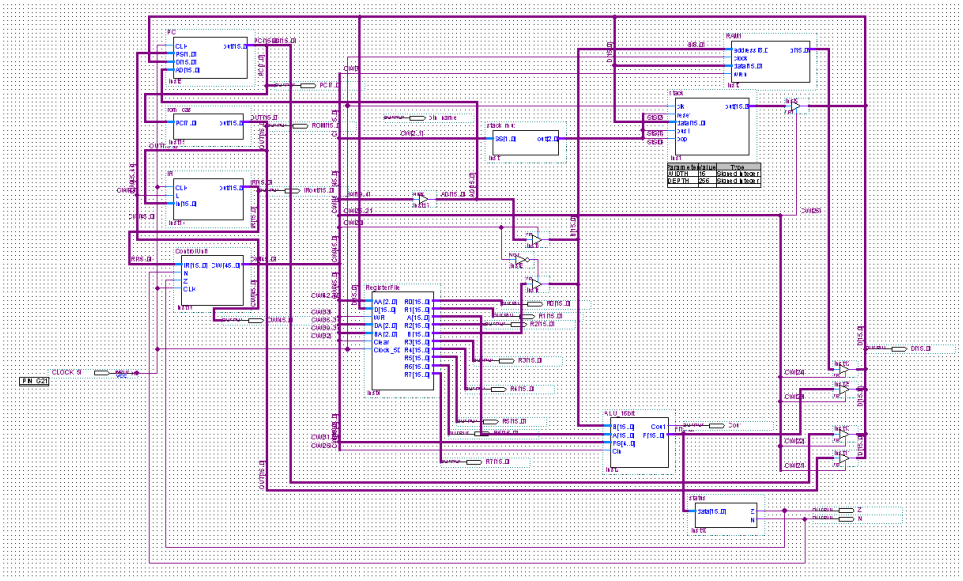

Figure 16: Schematic for Control Unit, Program Counter, ALU, Memory, Register File, and Stack

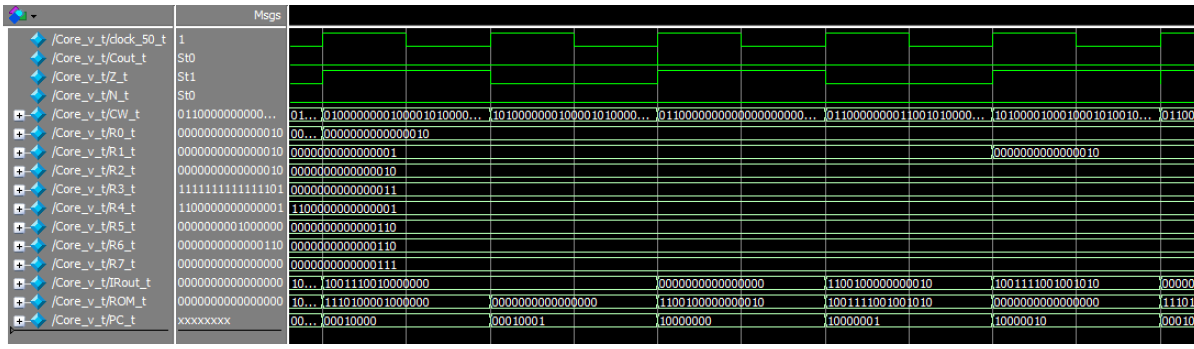

Figure 17: Simulation of Complete CPU

The simulation above is a screen shot of our CPU test bench in which we tested our call and return functions. Here you can see that after PC 10001, PC jumps to 10000000, where the subroutine loads a literal value to R1 and then returns to the previous PC.

10 Peripherals

10.1 General Purpose Input and Output

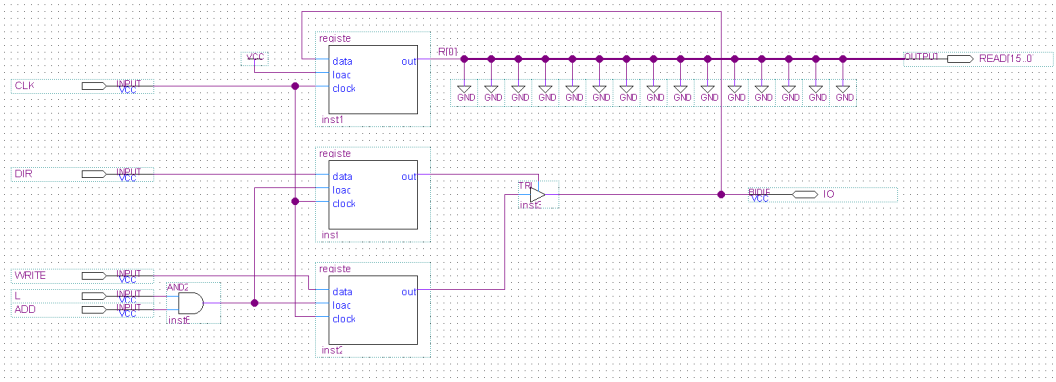

Figure 18: The schematic for a single GPIO pin

The General Purpose Input and Output (GPIO) peripheral is responsible for receiving input and communicating output through pins on the DE0. The schematic of an individual GPIO pin can be seen in Figure 18. These cells are the placed together to make larger blocks, these blocks can be seen in figure 19.

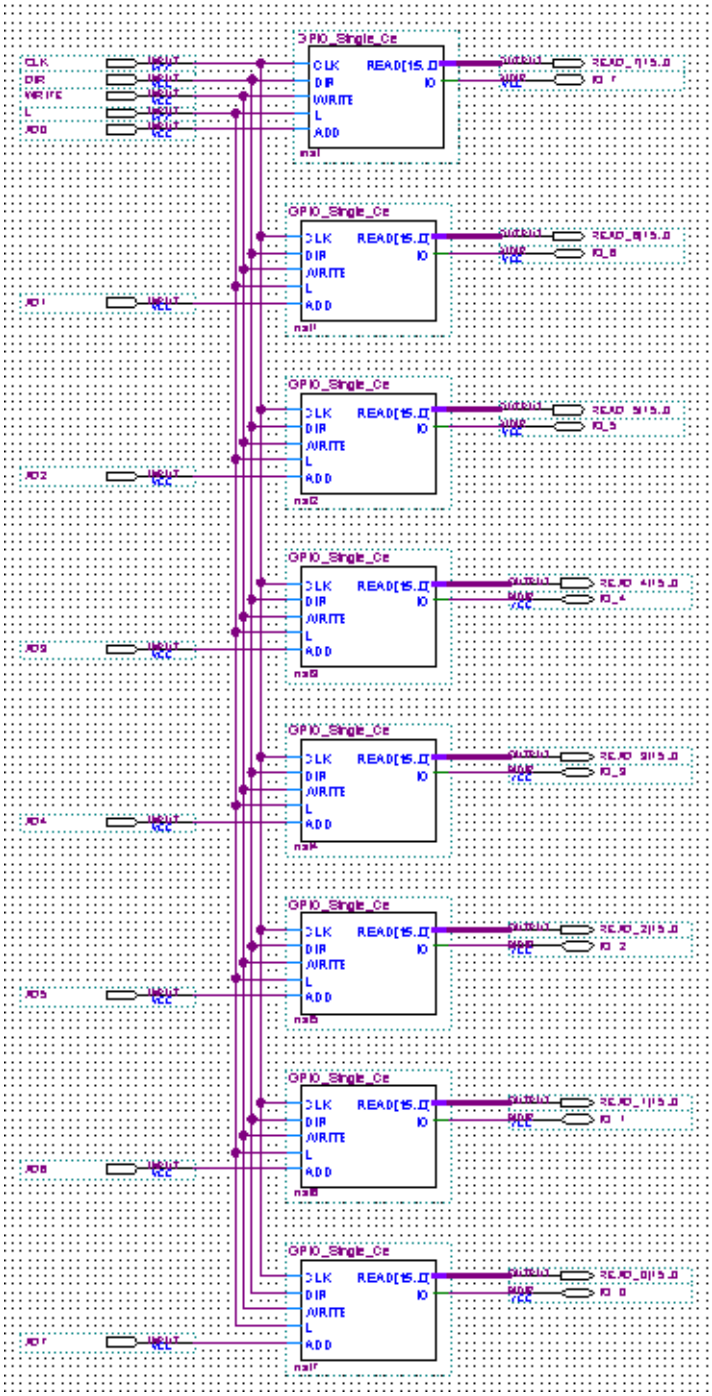

Figure 19: The schematic to manipulate 8 individual GPIO pins

10.2 Personal System/2 Keyboard

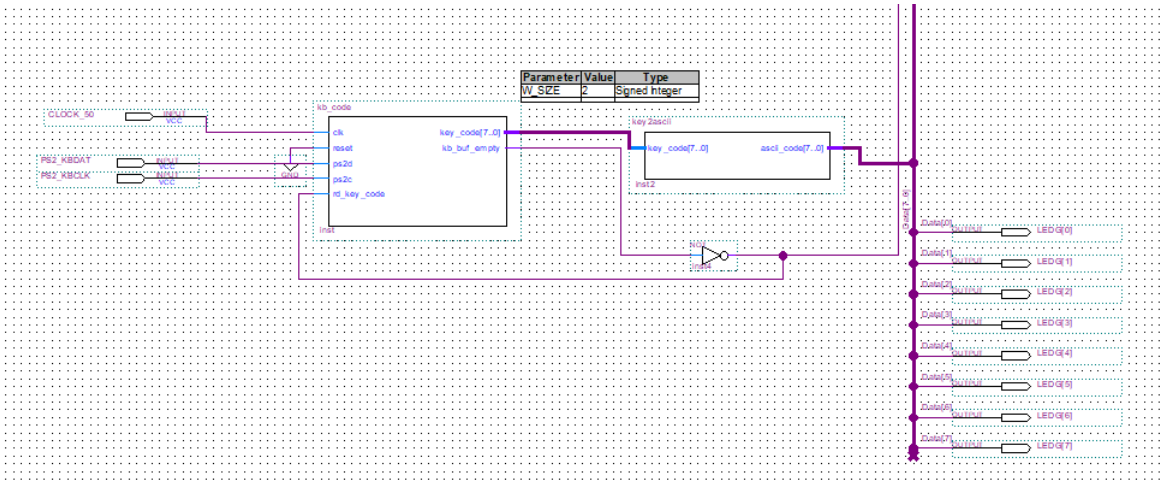

Figure 20: Full schematic of PS/2 module

Figure 20 contains the top level design for the PS/2 peripheral. This consists of a receiver, a first in first out buffer (FIFO), and finite state machine for control (FSM). The basic idea is to use the FSM to keep track of the F0 packet which is the break code. After this break code has been received, the next packet should be the make code of the key that was pressed. This make code is then written into the FIFO buffer. The output of the FIFO buffer is sent to a switch statement that converts the hex code output of the PS/2 to ASCII so that it can be converted to letters, numbers and symbols. We plan to use the ASCII converter to control which LEDs light up. In the configuration in figure 20 the output of the ASCII converter is connected to 8 LEDs which will display the output of the circuit. [2]

11 Example Program

11.1 McGuire

The following is the ROM for a program that flashes selected LEDs. It does this by writing the GPIO pin of each LED with a 1 and then writing a 0 to it shortly after. This is executed in an infinite loop as indicated by the ”JUMPR 0” instruction.

NOT R7, R2
$8'b00000000 : out[15:0] \leftarrow 16'b0100011111010010;$
ORI R3 2
$8'b00000001 : out[15:0] \leftarrow 16'b0010101100000010;$
STI R7, 0
$8'b00000010 : out[15:0] \leftarrow 16'b1010111100000000;$
NOP (skips AD 1)
$8'b00000011 : out[15:0] \leftarrow 16'b0000000000000000;$
STI R7, 2
$8'b00000100 : out[15:0] \leftarrow 16'b1010111100000010;$
NOP (skips AD 3)
$8'b00000101 : out[15:0] \leftarrow 16'b0000000000000000;$
STI R7, 4
$8'b00000110 : out[15:0] \leftarrow 16'b1010111100000100;$
STI R7, 5
$8'b00000111 : out[15:0] \leftarrow 16'b1010111100000101;$
STI R7, 6
$8'b00001000 : out[15:0] \leftarrow 16'b1010111100000110;$
STI R7, 7
$8'b00001001 : out[15:0] \leftarrow 16'b1010111100000111;$
STI R3, 0
$8'b00001010 : out[15:0] \leftarrow 16'b1010101100000000;$
NOP (skips AD 1)
$8'b00001011 : out[15:0] \leftarrow 16'b0000000000000000;$
STI R3, 2
$8'b00001100 : out[15:0] \leftarrow 16'b1010101100000010;$
NOP (skips AD 3)
$8'b00001101 : out[15:0] \leftarrow 16'b0000000000000000;$
STI R3, 4
$8'b00001110 : out[15:0] \leftarrow 16'b1010101100000100;$
STI R3, 5
$8'b00001111 : out[15:0] \leftarrow 16'b1010101100000101;$
STI R3, 6
$8'b00010000 : out[15:0] \leftarrow 16'b1010101100000110;$
JMPR 0
$8'b00010001 : out[15:0] \leftarrow 16'b1001101011101110;$
NOP
$default : out \leftarrow 16'b0000000000000000;$

11.2 Souder

Below is the ROM for the program which turns on the LED's 2 4 5 6 7. This ROM sets the memory addresses for the special function registers associated with the LED's last 2 bits to 11, the first 1 means that we will be writing to the LED and the second 1 is the data to be written so the data written is a 1 which means that the light will turn on.

NOT R7, R2
$8'b00000000 : out[15:0] \leftarrow 16'b0100011111010010;$
LRI R7, 0
$8'b00000001 : out[15:0] \leftarrow 16'b1010111100000000;$
NOP
$8'b00000010 : out[15:0] \leftarrow 16'b0000000000000000;$
LRI R7, 2
$8'b00000011 : out[15:0] \leftarrow 16'b1010111100000010;$
NOP
$8'b00000100 : out[15:0] \leftarrow 16'b0000000000000000;$
STI R7, 4
$8'b00000101 : out[15:0] \leftarrow 16'b1010111100000100;$
STI R7, 5
$8'b00000110 : out[15:0] \leftarrow 16'b1010111100000101;$
STI R7, 6
$8'b00000111 : out[15:0] \leftarrow 16'b1010111100000110;$
STI R7 7
$8'b00001000 : out[15:0] \leftarrow 16'b1010111100000111;$
LDI R0, 0
$8'b00001001 : out[15:0] \leftarrow 16'b1010000000000000;$
LDI R1, 1
$8'b00001010 : out[15:0] \leftarrow 16'b1010000100000001;$
LDI R2, 2
$8'b00001011 : out[15:0] \leftarrow 16'b1010001000000010;$
LDI R3, 3
$8'b00001100 : out[15:0] \leftarrow 16'b1010001100000011;$
LDI R4, 4
$8'b00001101 : out[15:0] \leftarrow 16'b1010010000000100;$
LDI R5, 5
$8'b00001110 : out[15:0] \leftarrow 16'b1010010100000101;$
LDI R6, 6
$8'b00001111 : out[15:0] \leftarrow 16'b1010011000000110;$
LDI R7, 7
$8'b00010000 : out[15:0] \leftarrow 16'b1010011100000111;$
JMPR 0
$8'b00010001 : out[15:0] \leftarrow 16'b1001101011101110;$

NOP
$default : out[15:0] \leftarrow 16'b0000000000000000;$

11.3 Epifano

This program will take input from the keyboard at memory address 8. The program will be constantly storing whatever the keyboard outputs to LED 0. The LED will only turn on if the last two bits of the make key is 11. For this keyboard, K and J meet this requirement. The program will jump back to the first instruction and do it all over again. This program verifies three things: the keyboard sends information to memory, the data that is sent is the correct value per key, and the GPIO is correctly reading the data and responding correctly to the data.

$LDI, R2, 8$
$\mathbf{8'b00000000 : out[15:0] \leftarrow 16'b1010001000001000;}$
$STI, R2, 0$
$\mathbf{8'b00000001 : out[15:0] \leftarrow 16'b1010101000000000;}$
$JUMPR, PC \rightarrow 0$
$\mathbf{8'b00000010 : out[15:0] \leftarrow 16'b1001101011111101;}$

12 Errata

Some things on the processor work a little differently than expected. One example of this is the branch instructions. For the branch instructions to work properly the status signals must be on the data bus one clock cycle before. The easiest way to make sure that the instruction is used correctly include a NOP one instruction before the branch instructions. The JUMPR instruction works by increasing the program counter by the immediate value. This means that the function behaves like an offset. The maximum clock frequency was determined by running the PS/2 peripheral program until it corrupted. This is because the over clocked ROM used branch instructions which are implemented differently.

13 Appendix

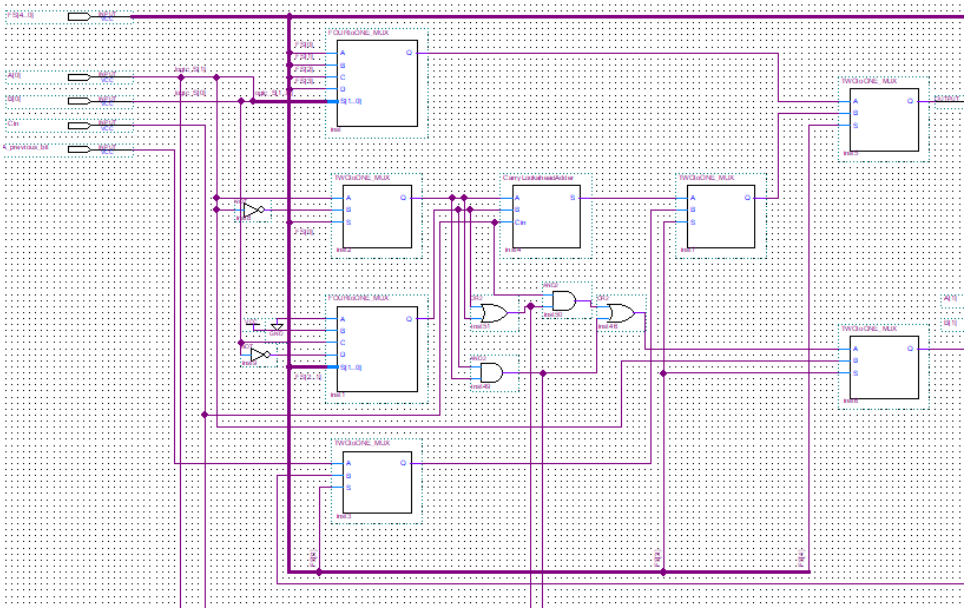

Figure 21: 1-bit ALU schematic

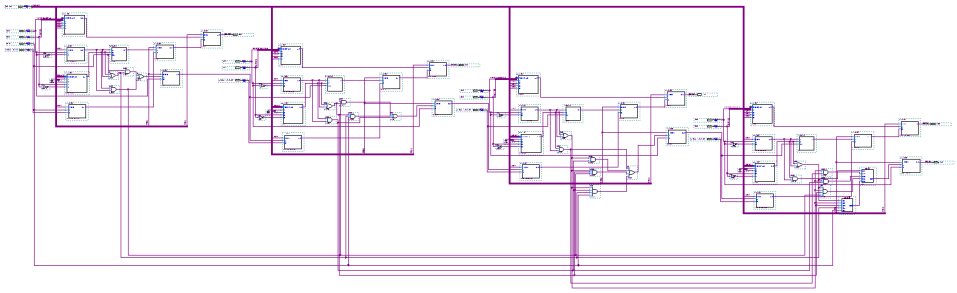

Figure 22: 4-bit ALU schematic

```
1   module IR (CLK, L, in, out);
2       input L;              //Signal
3       input CLK;
4       input [15:0]in;
5       output reg [15:0]out;
6
7       initial begin
8           out <=16'b0110000000000000;
9       end
10      always @(posedge CLK) begin
11          if (L) begin
12              out<=in;
13          end
14      end
15  endmodule
16
```

Figure 23: Verilog Description of Instruction Register

```
module PC (CLK, PS, D, AD, out);
    input [1:0]PS;
    input CLK;
    input [15:0] D;
    input [15:0] AD;
    output  reg [15:0]out;

    initial begin
        out <= 0;
    end

    always @(posedge CLK)begin
        case (PS)
            2'b00: out = out;
            2'b01: out = out+1'b1;
            2'b10: out = D;
            2'b11: out = out+AD;
        endcase
    end
endmodule
```

Figure 24: Verilog Description of Program Counter

References

[1] ”Microprocessor with Harvard Architecture”, 5034887 A, 2017.

[2] P. Chu, FPGA Prototyping By Verilog Examples. Somerset: Wiley, 2011.