State University of New York at New Paltz

Department of Electrical and Computer Engineering

Intro to Computer Architecture
EGE-442

Design Of A MIPS Like Processor

Group Members	Department	Major Contribution
Paul Boston	CE	Theory, Design
Mitchell Wagner	CE	Theory, Design

Course Professor: Baback A. Izadi

Due: 12/5/14
FALL 2014

Abstract

Machines take up virtually all parts of our lives and most machines are built of processors. This allows the machine to function sending data to all parts of the device as well as allow data to be received to and from by the user.

Table of Contents:

Introduction

Any processor has several key parts: PC, Instruction Memory, Register File, ALU, and Data Memory.

The Instruction Memory will fetch the instruction that the Program Counter (PC) is presenting pointing at. Instruction is the built of Opcode, Registers, Branch Commands, address, number and Function depending upon what type of instruction it is. When the instruction is taken the PC is also feed into an adder with 4 added to it because each instruction is a word. This will allow for the next clock cycle the next instruction be taken.

The Register File is built of Flip-Flops and Mux's to take the address of the register it is passed and return the data that is associated with that register. The Register File also has the ability to write to a register by sending in both the register address and data that will be store in the register.

The ALU performs arithmetic and logical operations ADD, SUB, OR, AND. These operations are chosen by the being feed from the control bus into the ALU control to be able to determine what operation is needed to be executed. The ALU also will have different flags from each operation such as Zero, Carryout, Sign, and Overflow. The flags are good when using branches by seeing what the current flags are whether or not to branch.

The Data Memory is a device that takes in an address and from that address in memory data can either be read or written to. This device is often used with both the load and store instructions.

There are different instructions formats for every instruction type. The main instruction Types are: R, I, and J also there is a separate Brach instruction type, each instruction is 32 bits.

Table 1 Instruction Type:

	Machine Code					
Mem Loc	0-5	6-10	11-15	16-20	21-23	24-31
R-Type	Op	Rd	Rs	Rt	xxx	Func
I-Type	Op	Rd	Rs	Address		
J-Type	Op	Address				
BC-Type	Op	BI	Address			

R-Type instructions are broken up into: opcode (0-5), rd(6-10), rs(11-15) rt(16-20) and func(24-31). The Function will determine what type of instruction is, whether to and, add, or, sub.

I-Type instruction are of the form opcode(0-5), rd(6-10), rs(11-15) and addr(16-31). In I type the last 16 bits are data of the value or address in LW and SW instructions.

J-Type instructions are of the form opcode(0-5) and addr(6-31) this address is representing the location to go in the case of a jump.

Branch Conditional are of the form opcode(0-5), BI(6-9) and addr(10-31). The branch instruction is calling the BI control this control will tell what type of branch is desired. Once this is found then the branch will look at what the current flag setup is for the instruction.

Design Machine Code

The processor was create similar to a MIPS like machine code. In order to do this a list of all the functions needed to be compiled. Then using the required bit sizes of the opcode and function fields our own data for those fields, different data per instruction were created. The branch instructions needed to have a branch control, this control will decide which branch action will be executed. Table 2 shows the machine code for each intructions of our processor

Table 2: Instruction Code

Type	Instruction		Op	Rd	Rs	Rt	xxx	Func
R and Immediate	ADD	add	000000				----	0000000
		addi	000001					
	SUB	sub	000000				----	0000010
		subi	000101					
	AND	and	000000				----	0000100
		andi	001001					
	OR	or	000000				----	0000101
		ori	001101					

		Op	Rd	Rs	Address
I	lw	100000			
	sw	100001			

		Op	Address
J	Jump	100011	

		Op	BI	Address
BC	BEQ	110000	0001	
	BNE	110000	0010	
	BLT	110000	0011	
	BGT	110000	0100	
	BP	110000	0101	
	BN	110000	0110	
	BO	110000	0111	

Design

ALU Design

The processor design called for an ALU. Inside the ALU there is a full adder, to add or subtract,   an ‘AND’ and ‘OR’ gate. This is the same for all 32 bits for the ALU.

Table 3: Adder Truth Table

Input			Output	
A	B	Cin	Cout	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 4: Multiplexer

Selector's			Instruction Type
S2	S1	S0	
0	0	0	AND
0	0	1	OR
0	1	0	----
0	1	1	ADD
1	0	0	----
1	0	1	----
1	1	0	----
1	1	1	SUB

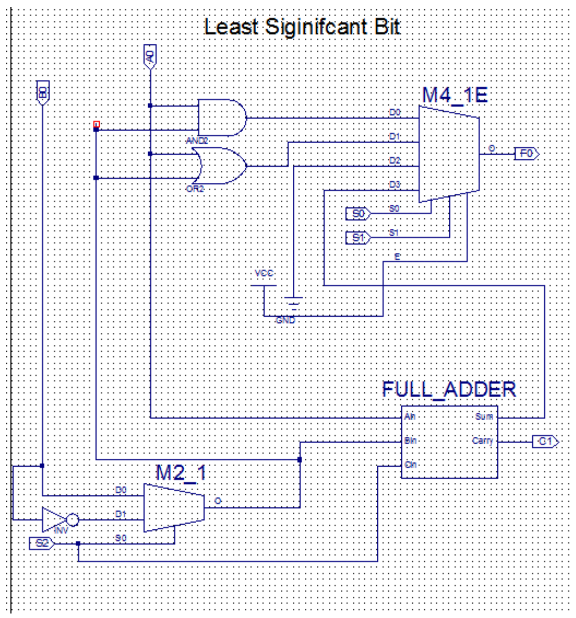


Figure 1: Least Significant Bit

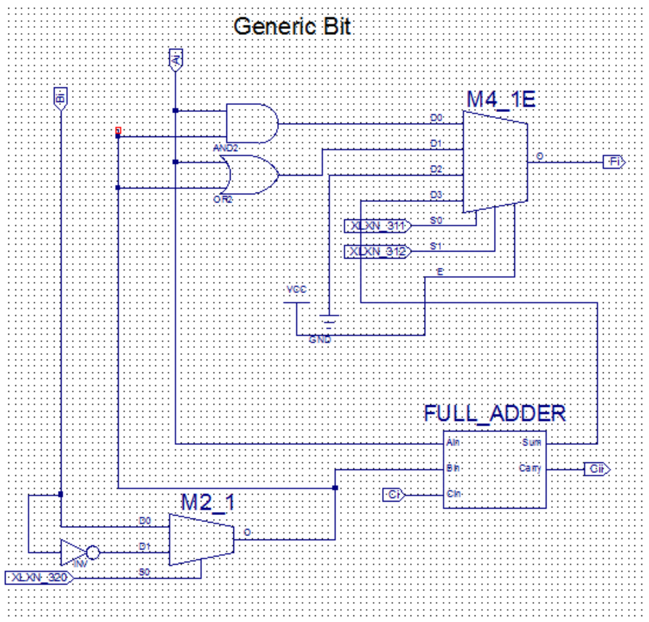


Figure 2: Generic Bit

Figures 1 and 2 show the Least Significant Bit (LSB) and Generic Bit. The difference between these two bits are that the carry in for the LSB is dependent on selector 2 (S2). The Generic Bit’s carry in is taken from the previous bit’s carry out.

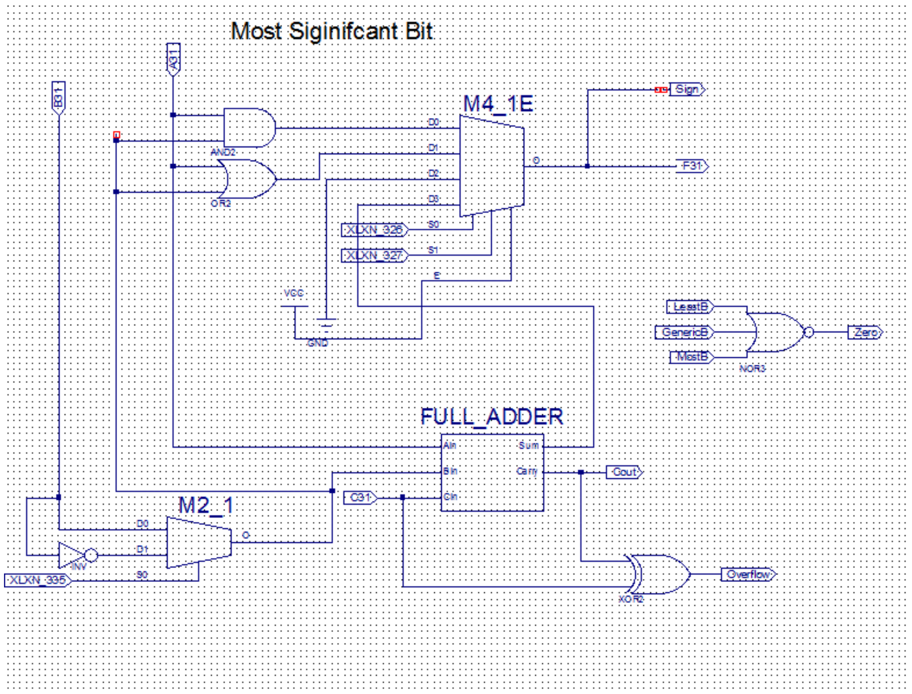


Figure 3: Most Significant Bit

Figure 3 shows the Most Significant Bit (MSB) which also has all the flags that will be used. The Sign is the MSB, Overflow is the XOR of the MSB carry in and carry out, and the zero flag test is the NOR of all the bits solutions.

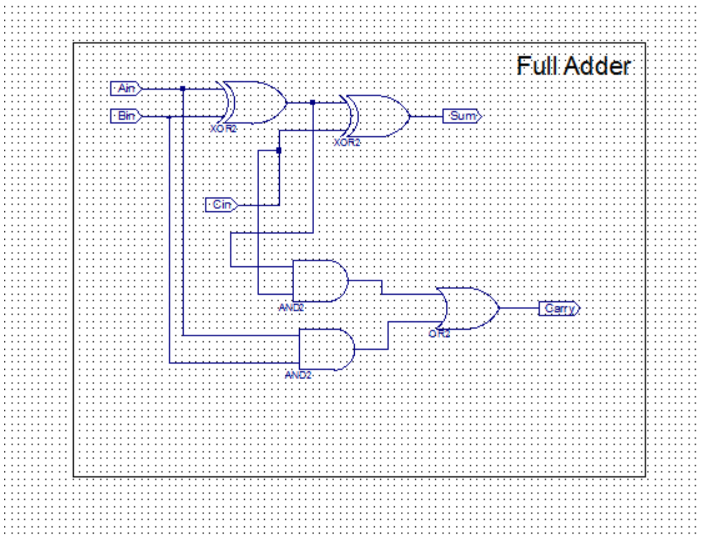


Figure 4: Full Adder

The full adder schematic is seen in Figure 4 the truth table for this schematic is seen in Table 3.

Design Processor Data Path

The data path required for this processor was standard for you normal MIPS processor. The processor could be broken down into 5 different sections, Instruction Memory, Register Read, ALU, Data Memory, and Register Write. Together these functions combined with adders, 2x1 mux's, shifters, and sign extenders the full processor can be constructed. The data path for our processor are separated into 3 Figures.

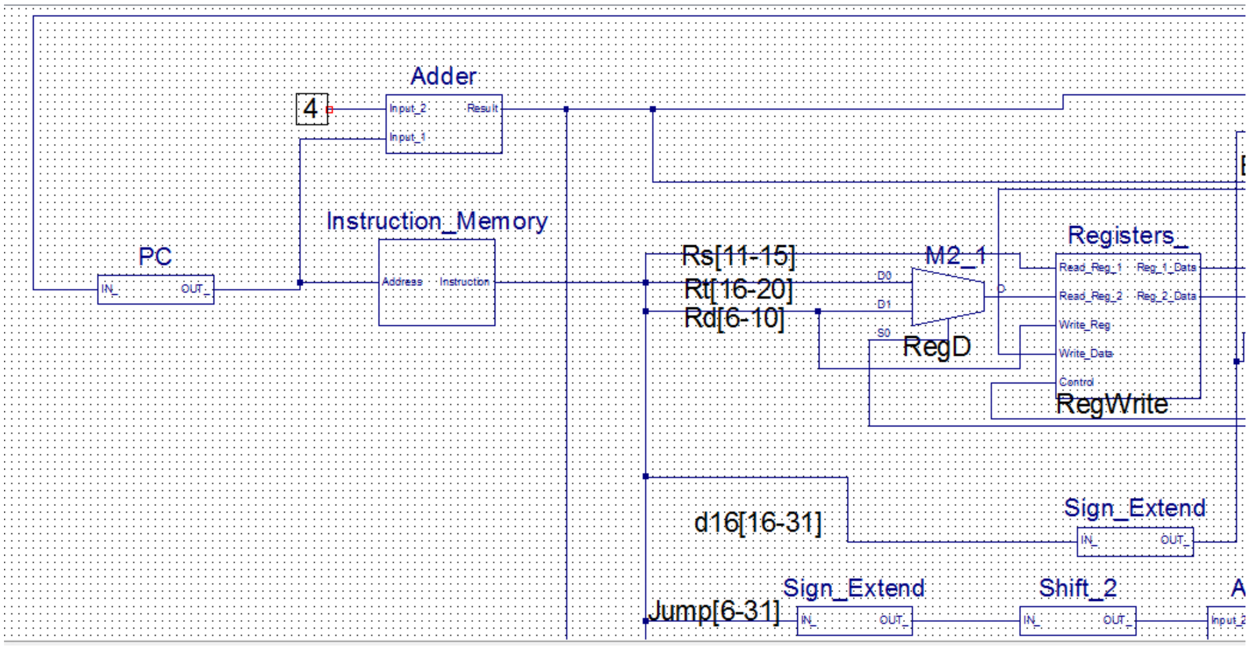


Figure 5: Processor Part 1

Figure 5 shows the first part of the Processor. This comprises of the PC counter, Instruction Memory, and the Register File. The PC counter is being added to 4 to be able to take the next instruction. The instructions going into the Registers File are Rs, Rt, and Rd. Depending on the instruction Rd read in. At the end of the process if data needed to be written back into a register the data would travel back to the Register File and the desired register would have that data stored in it. This Figure also shows d16 getting sign extended so that incase it is a negative that will still be passed through the processor. Jump is also observed to be sign extended and shifted by 2 to make the instruction a word.

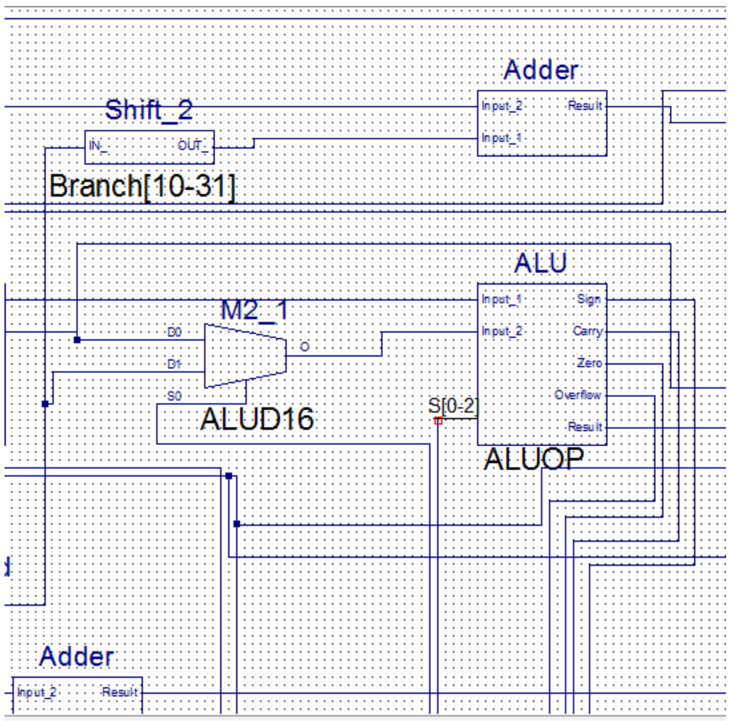


Figure 6: Processor Part 2

Figure 6 shows part 2 of the Processor. This shows both ALU. The second input to the ALU is being fed from a Mux. This Mux will determine whether the value will be from the register or the d16.

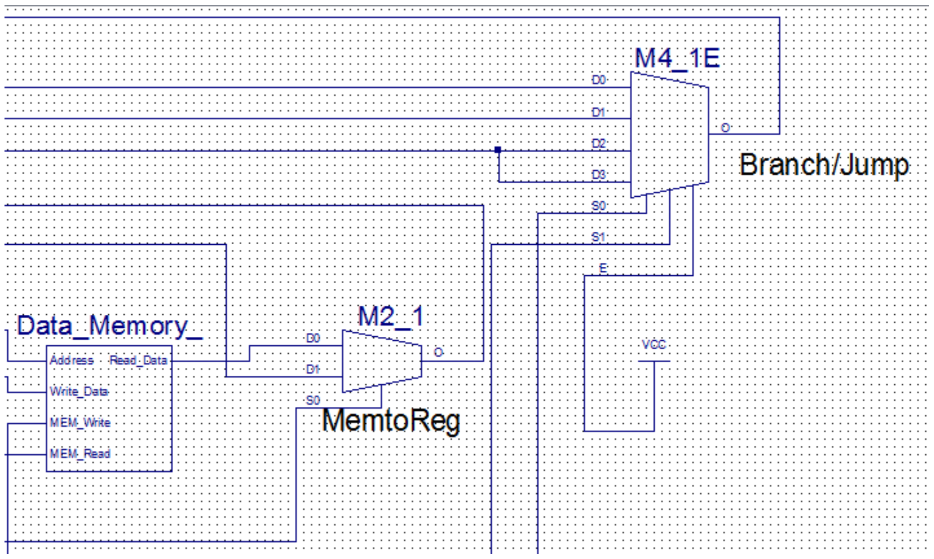
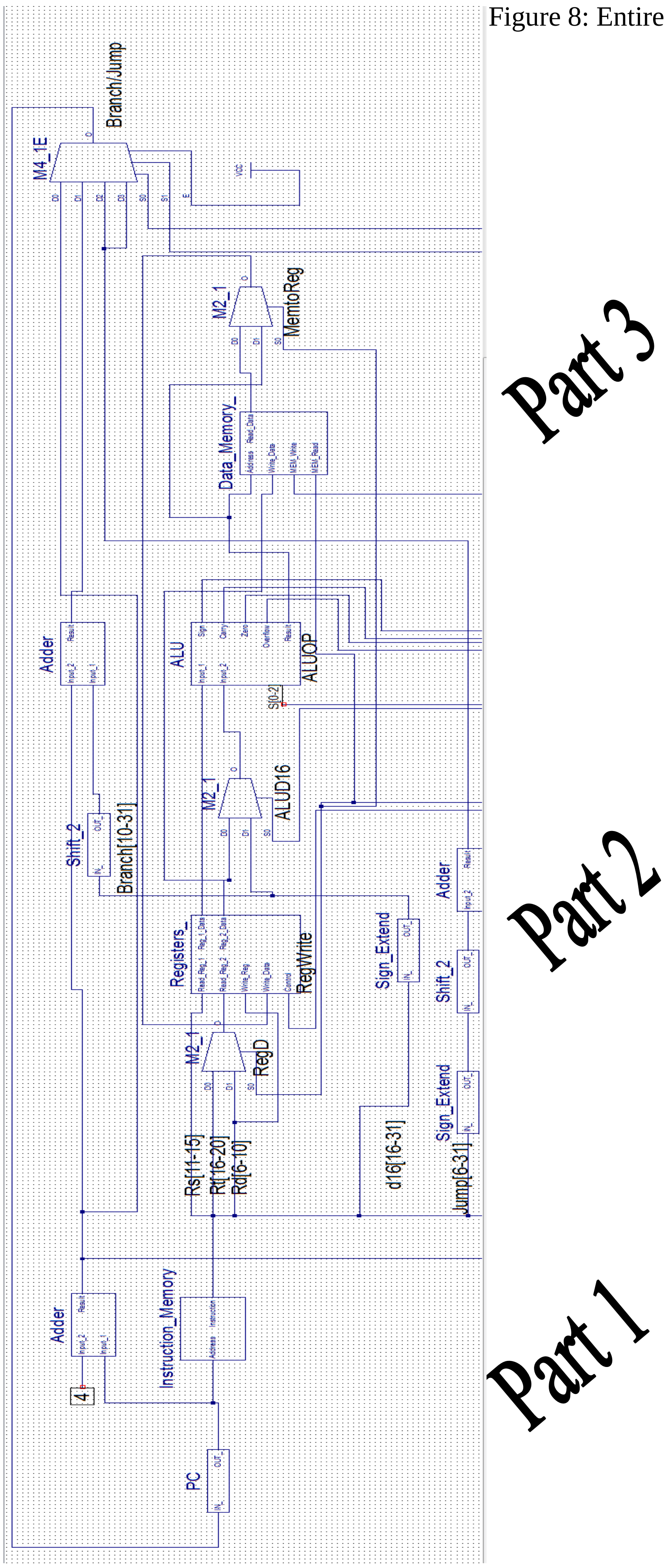


Figure 7: Processor Part 3

Figure 7 shows the part 3 of the Processor. This has the Data Memory which is fed from the ALU and the Mux afterwards will determine if the processor will write the data back to a register. This Figure also show a mux which has PC+4, the Branch address and the Jump address and is determined by the Control and BI buses. All the parts of the Processor can be seen in Figure 8.

Figure 8: Entire Processor



Design ALU Control Design

The ALU control bus would control the select lines of the mux in the ALU to decide which function gets done, so that the Opcode and function bits can control the ALU. The ALU control is comprised of the ALU op1, which is the result of the control bus, function bits [29-31] and Opcode [2, 3, 5]. The output of the ALU control bus would determine what the function would result in.

Table 5: ALU Control

OPCODE 2	OPCODE 3	OPCODE 5	ALU op1	F2	F1	F0	S[2-0]	Type
0	0	0	0	x	x	x	011	add: lw
0	0	1	0	x	x	x	011	add: sw
0	0	0	1	0	0	0	011	add
0	0	0	1	0	1	0	111	sub
0	0	0	1	1	0	0	000	and
0	0	0	1	1	0	1	001	or
0	0	1	1	x	x	x	011	add
0	1	1	1	x	x	x	111	sub
1	0	1	1	x	x	x	000	and
1	1	1	1	x	x	x	001	or

Table 6: ALU Control Logic

S2	Solution: Y = (!A!B!CD!EF!G)+(!ABCD)
S1	Solution: Y = (!A!B!CD!E!G)+(!A!B!D)+(!ACD)
S0	Solution: Y = (!A!B!D!E!FG)+(!A!B!DE!G)+(!A!B!DFG)+(!A!BCD!E!FG)+(!A!B!E!G)+(!A!BE!FG)+(!A!BCDE!G)+(!A!BCDFG)+(BCD)

A=Opcode2 B=Opcode3 C=Opcode5 D=ALU E=F2 F=F1 G=F0

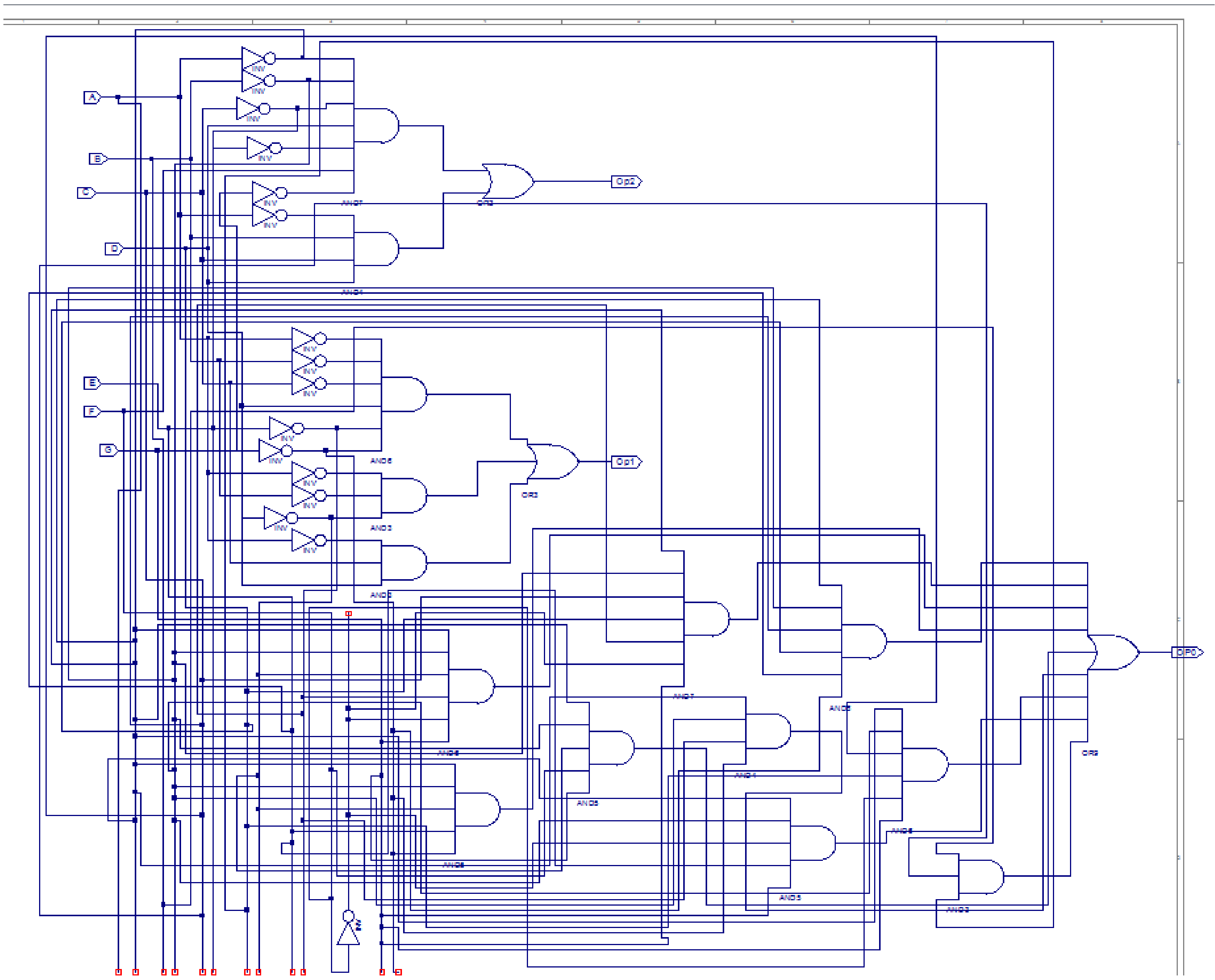


Figure 9: ALU Control Bus Schematic

Figure 9 depicts the schematic of the ALU control bus. All the inputs can be seen what they refer to on Table 5 and the logic equation solved for can be seen on Table 6.

The design of our processor called for branch instructions (BI). These branch instructions relied on the previous ALU flags. The instruction for branching has another control section in addition to the opcode. The BI will allow the logic gates to know exactly what flags to look at and then based on this the system will know to branch or not. This can be seen in Table 7 and Figure 10.

Table 7: BI Bus

BI Bus		RC			
Instruction	BI op	S (Sign)	CY (Carry)	EQ (Zero)	OV (OverFlow)
xxx	0000	x	x	x	x
BEQ	0001	x	x	1=Y	x
BNE	0010	x	x	0=Y	x
BLT	0011	x	1=Y	x	x
BGT	0100	x	0=Y	x	x
BP	0101	0=Y	x	x	x
BN	0110	1=Y	x	x	x
BO	0111	x	x	x	1=Y

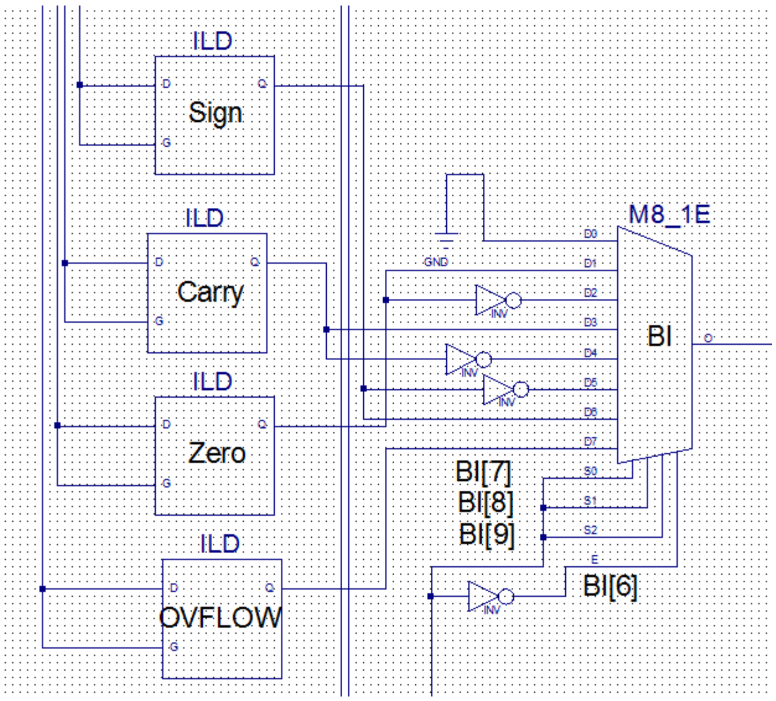


Figure 10: BI Control Bus Schematic

Figure 10 shows that after the ALU all the flags are fed into its own latch. This will allow for the next cycle the previous flags be viewed.

After the main data path and all the other controls are in place, the last control that needs to be created is the main control unit. This control feature will take the opcode from the instruction and figure out how to control all the mux's, the write and read lines. When we created our opcode we made it simple so that our logic could be easily derived from it, because of the features that would need to be active at the time that instruction was taking place. Tables 8 and 9 contain the logic for the control unit:

Table 8: Control Bus

Instruction		OpCode	RegD	RegWrite	ALUop1	JumpSel	MemToReg	MemRead	MemWrite	ALUd16
add	add	000000	0	1	1	0	0	x	0	0
	addi	000001	0	1	1	0	0	x	0	1
sub	sub	000000	0	1	1	0	0	x	0	0
	subi	000101	0	1	1	0	0	x	0	1
AND	and	000000	0	1	1	0	0	x	0	0
	andi	001001	0	1	1	0	0	x	0	1
OR	or	000000	0	1	1	0	0	x	0	0
	ori	001101	0	1	1	0	0	x	0	1
lw		100000	0	1	0	0	1	1	0	1
sw		100001	1	0	0	0	x	0	1	1
Jump		100011	x	0	x	1	x	x	0	x
BEQ		110000	x	0	x	0	x	x	0	x
BNE		110000	x	0	x	0	x	x	0	x
BLT		110000	x	0	x	0	x	x	0	x
BGT		110000	x	0	x	0	x	x	0	x
BP		110000	x	0	x	0	x	x	0	x
BN		110000	x	0	x	0	x	x	0	x
BO		110000	x	0	x	0	x	x	0	x

Table 9: Control Bus Logic

RegD	O0 ~O1 ~O4 ~O5
RegWrite	~O1 ~O4 ~O5 +~O0 ~O1 ~O4
ALU	~O0 ~O1 ~O4
JumpSel	O0 ~O1 O4 O5
MemtoReg	O0 ~O1 ~O4 ~O5
MemRead	O0 ~O1 ~O4 ~O5
MemWrite	O0 ~O1 ~O4 O5
ALUd16	~O1 ~O4 O5 +O0 ~O1 ~O4

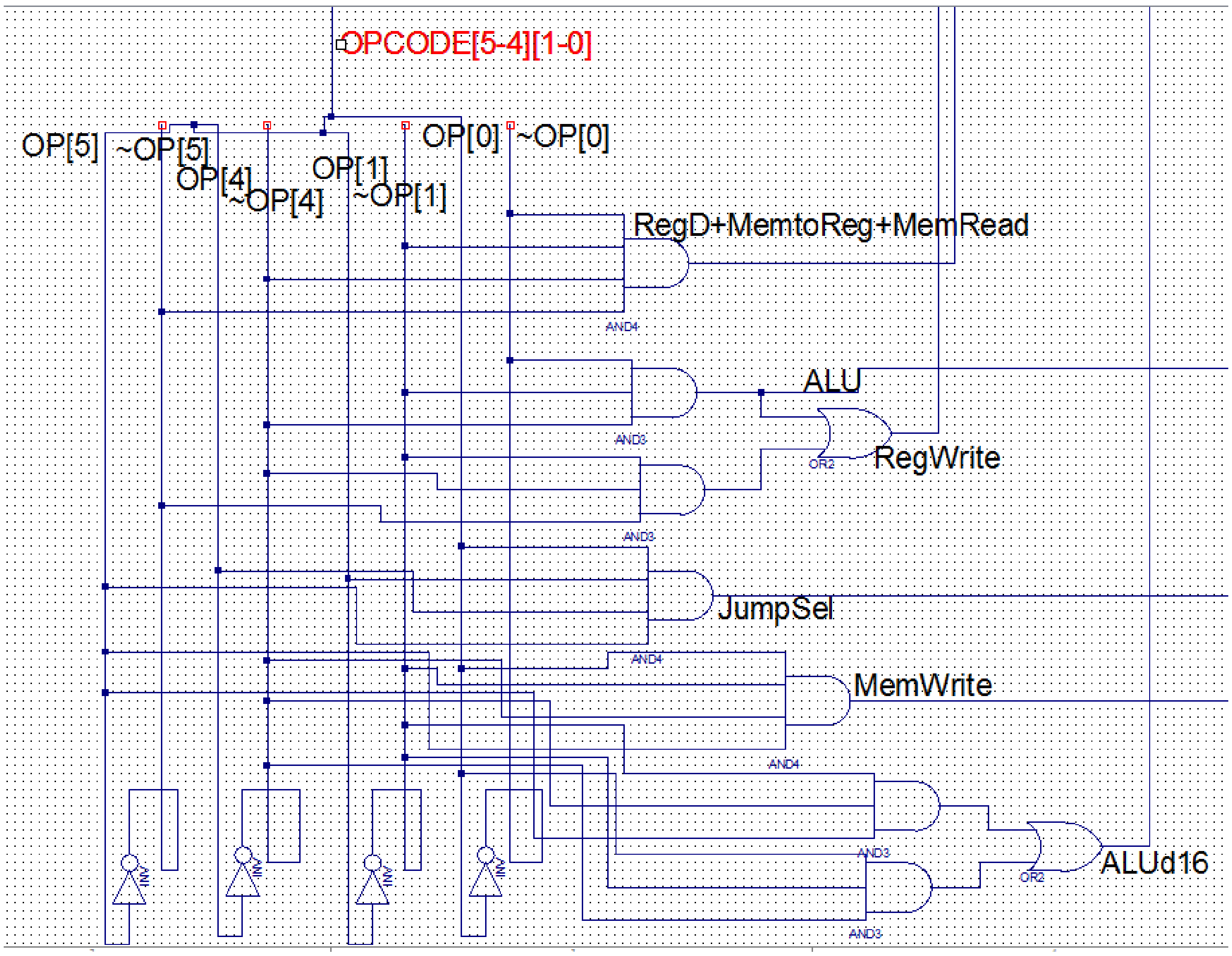


Figure 11: Control Bus Schematic

Figure 11 is designed form the logic of the Control Bus which is seen in Tables 8 and 9. Each of these outputs are then the selectors for all the Mux's, ALU control, Registers and Data Memory.

Conclusion

Through the performance of this project was done to create a processor to handle different type of instructions. These instructions included arithmetic, logical, loading, storing, and branch instructions. As well as all the flags needed to be designed and stored. The key parts of the design included the ALU construction, the Control Bus, ALU control and Branch Control.

The first area of confusion was in understanding of using the branch condition. This involved the flags being sent to a group of latches in with the branch would be looking at whatever the past cycles flags were instead of calculating between two registers and then looking at the flags. Another area of slight confusion was the instruction formats were listed different than the standard MIPS instruction with Rd being the first register and the least significant bit first.

The main problem encounter for this project was found when any of the immediate instructions were going to be executed the processor had no way of understanding the difference between them. So in order to fix this problem instead of starting over the opcode was changed then in the ALU control bus opcodes [2, 3, 5] would be passed in as inputs. This allowed for the correct selection between any immediate instructions. Things that would be done to make this better would be to check that the machine code is accurate before implementation.

Key parts to this lab were that the machine code structure of each type of instruction. Combining the R-type and Branch instructions to select form either the function or BI memory locations. Originally the opcodes were taken from the MIPS instruction than as it was seen that all the instructions formats were different and that the opcodes needed to be changed for branch instructions as well as changing both the function and creating a branch condition our own format was created.