ECE2504 Design Project 3 Cover Sheet

Student Name: _Joshua Chung____

SIGNATURE

Honor Code Pledge:
I have neither given nor received unauthorized assistance on this assignment (sign in the box to acknowledge this):

Project Item	Value	Points
Project Questions:		
● Table showing the opcode values that were chosen	5	
● Step 1 simulation results.	5	
● Completed table of dissembled instructions (Table 3)	5	
● Compare the results of the simulation to what was expected given the instructions in Table 3	5	
● Clearly documented changes made to the design	5	
● Annotated simulation results in your report showing the correct behavior of the augmented Step 2 instruction	10	
● A table, similar to Table 4, that contains the information necessary to understand the simulation results for each instruction	5	
Report:		
● Cover sheet	5	
● Organization: Clear, concise presentation of content; Use of appropriate, well-organized sections	10	
● Design approach, decisions, observations and conclusions	5	
● Implementation discussion: Choice of methods for implementing instructions, opcode binary assignments for instructions, changes to the design.	10	
● Supporting files: instruction.txt, data.txt, and modified Verilog files	5	
Validation		
● All steps function correctly	25	
● **TOTAL POINTS**	■	
Extra Credit: Correctly implementing optional instruction	10	

Joshua Chung
ECE 2504 (MWF 1:25 – 2:15)
Project 1 writing report

PROJECT OVERVIEW

The objective of project 3 was to implement a simple computer given to us through Verilog and then implement new instructions (specifically four) to perform certain 'operations.' To fully understand the Simple computer I had to understand how to use Registers, Register Transfer Language (RTL) and addresses to create new opcodes. The Verilog code given to us already contained the Arithmetic Logic Unit (ALU) and a Memory Unit so what we had to do was create a new set of instruction codes in Hexadecimal to correctly implement the new instructions.

The project report will go through the design approach, decisions, observations and conclusions.

Design Approach

Step 1 of this project was to understand the operations of the Simple Computer by running Qsim after converting the "Maching Code Value" into the actual instruction that the code is doing. To understand these operations we were given a chart (*refer to table 1*) that referenced our instruction.txt file and the instructions inside the file. This is Table 3 in the project specification.

Instruction Memory Address	Machine code Value	Binary Value	Instruction (values in decimal)
0	1400	|0001010|000|000|000	XOR R0, R0, R0
1	20C0	|0010000|001|100|000	LD R1, R4
2	8444	|1000010|001|000|100	ADI R1, R0, 4
3	0480	|0000010|010|000|000	ADD R2, R0, R0
4	0290	|0000001|010|010|000	INC R2, R2
5	0C48	|0000110|001|001|000	DEC R1, R1
6	C00A	|1100000|000|001|010	BRZ R1, 2 ****
7	C1C5	|1100000|111|000|101	BRZ R0, -3****
8	0158	|0000000|101|011|000	MOVA R5, R3
9	0BAA	|0000101|110|101|010	SUB R6, R5, R2
10	0368	|0000001|101|101|000	INC R5, R5
11	0C48	|0000110|001|001|000	DEC R1, R1
12	1A41	|0001101|001|000|001	SHR R1, R1
13	C20A	|1100001|000|001|010	BRN R1, 2
14	E000	|1110000|000|000|000	JMP R0

Table 1: The table showing all of the original machine code converted to the instruction each code should undergo.

Below is the simulation needed to finish Step 2 in the project specification.

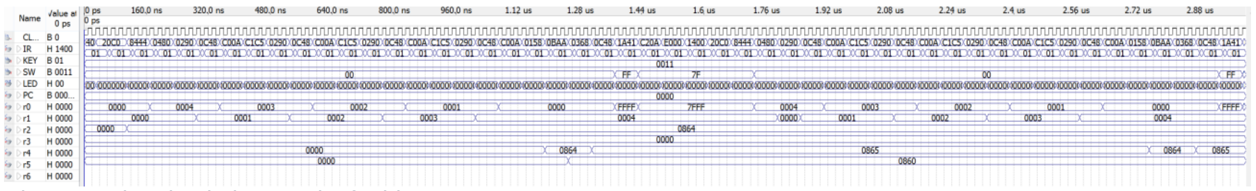


Figure 1: The simulation result of table 1

Looking at the table the first instruction memory address 0 has the machine code value 1400 in hexadecimal. Then the instruction says that the 1400 will run the operation of an XOR between R0 (register 0) and R0 and store the result into R0. In RTL the operation would look like this: R0 $\leftarrow$ R0 XOR R0. Address 2 the machine code value is 8444 which is converted to perform the instruction ADI R1, R0, 4; R1 $\leftarrow$R0 + 4. So for the first step in my Design approach I converted all of the Machine code value into the instruction that it was supposed to do and then referred to the image of the simple computer's architecture (*refer to figure 2*).

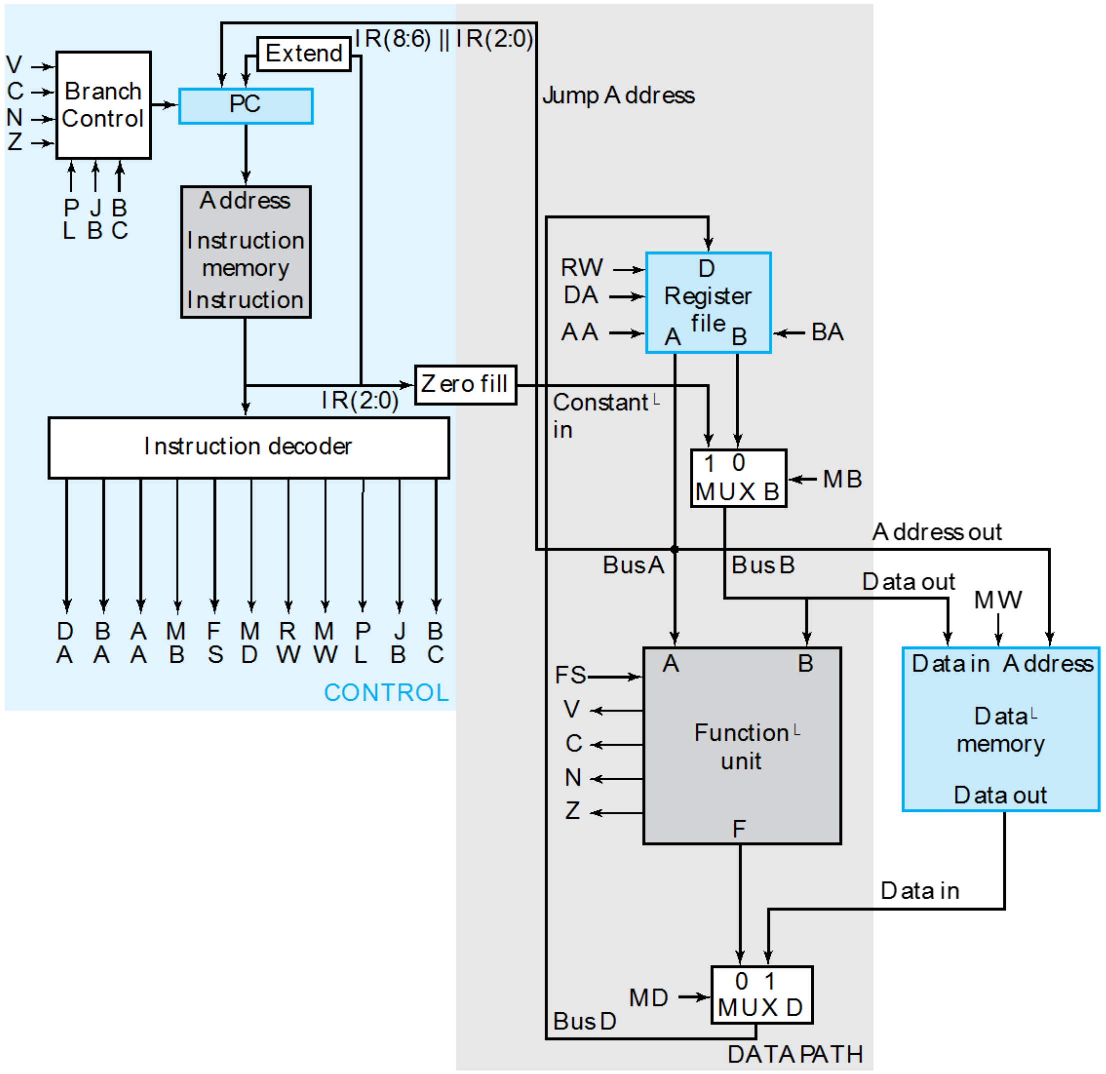


Figure 2: The architecture of the simple computer given to us in Chapter 9.

Referring to Address 2 in table 1 the machine code or also known as the instruction 8444 converted to binary is 1000010001000100. The first seven most significant bits 1000010 is known as the opcode. Within the opcode the last four bits 0010 represent the function (FS) that will be operated in the Function unit (*refer to Figure* 2) in this case the function Add immediate will be operated. The next three bits are 001 which is the destination register R1. The three bits after that are 000 which represent the register R0 and the last three bits 100 represents a constant value of 4. Putting this all together the opcode tells us that the instruction is an Add Immediate instruction where the operation would be R0 + 4 stored into R1. However, this is not always the condition where the last three bits represent a constant value. Depending on the function/opcode the instruction of the code changes. For example if the opcode was 0000001 then the instruction is to Add Register A to Register B and store it in a destination Register (*refer to figure 3*). Therefore, for this opcode there is no constant value but rather there is a register that is called. However, the destination register bits will always represent a register.

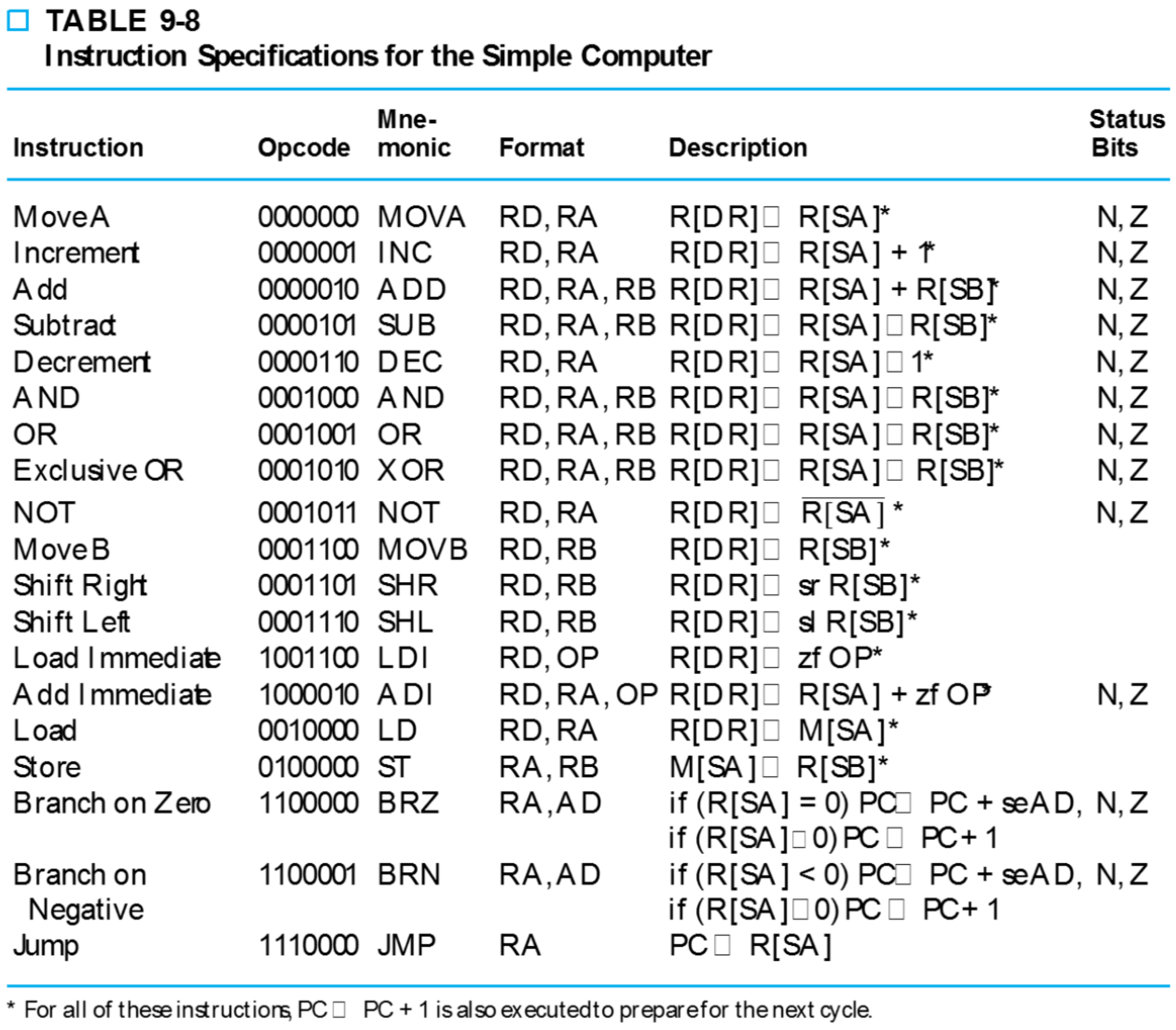


TABLE 9-8
Instruction Specifications for the Simple Computer

Instruction	Opcode	Mne-monic	Format	Description	Status Bits
Move A	0000000	MOVA	RD, RA	R[DR] □ R[SA]*	N, Z
Increment	0000001	INC	RD, RA	R[DR] □ R[SA] + 1*	N, Z
Add	0000010	ADD	RD, RA, RB	R[DR] □ R[SA] + R[SB]*	N, Z
Subtract	0000101	SUB	RD, RA, RB	R[DR] □ R[SA] □ R[SB]*	N, Z
Decrement	0000110	DEC	RD, RA	R[DR] □ R[SA] □ 1*	N, Z
AND	0001000	AND	RD, RA, RB	R[DR] □ R[SA] □ R[SB]*	N, Z
OR	0001001	OR	RD, RA, RB	R[DR] □ R[SA] □ R[SB]*	N, Z
Exclusive OR	0001010	XOR	RD, RA, RB	R[DR] □ R[SA] □ R[SB]*	N, Z
NOT	0001011	NOT	RD, RA	R[DR] □ $\overline{\text{R[SA]}}$ *	N, Z
Move B	0001100	MOVB	RD, RB	R[DR] □ R[SB]*	
Shift Right	0001101	SHR	RD, RB	R[DR] □ sr R[SB]*	
Shift Left	0001110	SHL	RD, RB	R[DR] □ sl R[SB]*	
Load Immediate	1001100	LDI	RD, OP	R[DR] □ zf OP*	
Add Immediate	1000010	ADI	RD, RA, OP	R[DR] □ R[SA] + zf OP*	N, Z
Load	0010000	LD	RD, RA	R[DR] □ M[SA]*	
Store	0100000	ST	RA, RB	M[SA] □ R[SB]*	
Branch on Zero	1100000	BRZ	RA, AD	if (R[SA] = 0) PC □ PC + se AD, if (R[SA] □ 0) PC □ PC + 1	N, Z
Branch on Negative	1100001	BRN	RA, AD	if (R[SA] < 0) PC □ PC + se AD, if (R[SA] □ 0) PC □ PC + 1	N, Z
Jump	1110000	JMP	RA	PC □ R[SA]	

* For all of these instructions, PC □ PC + 1 is also executed to prepare for the next cycle.

Figure 3: the opcodes, operations, and descriptions for all of the already implemented functions inside the Function Unit

Referring back to the instruction code 1000010001000100 I can decipher this instruction into different control words using an "Instruction Decoder" (*figure 4*). The control words are then used throughout the simple computer's architecture to determine the select lines for multiplexers or what function to perform and much more. The conversion of the instruction code 1000010001000100 to control words can be seen through figure 4.

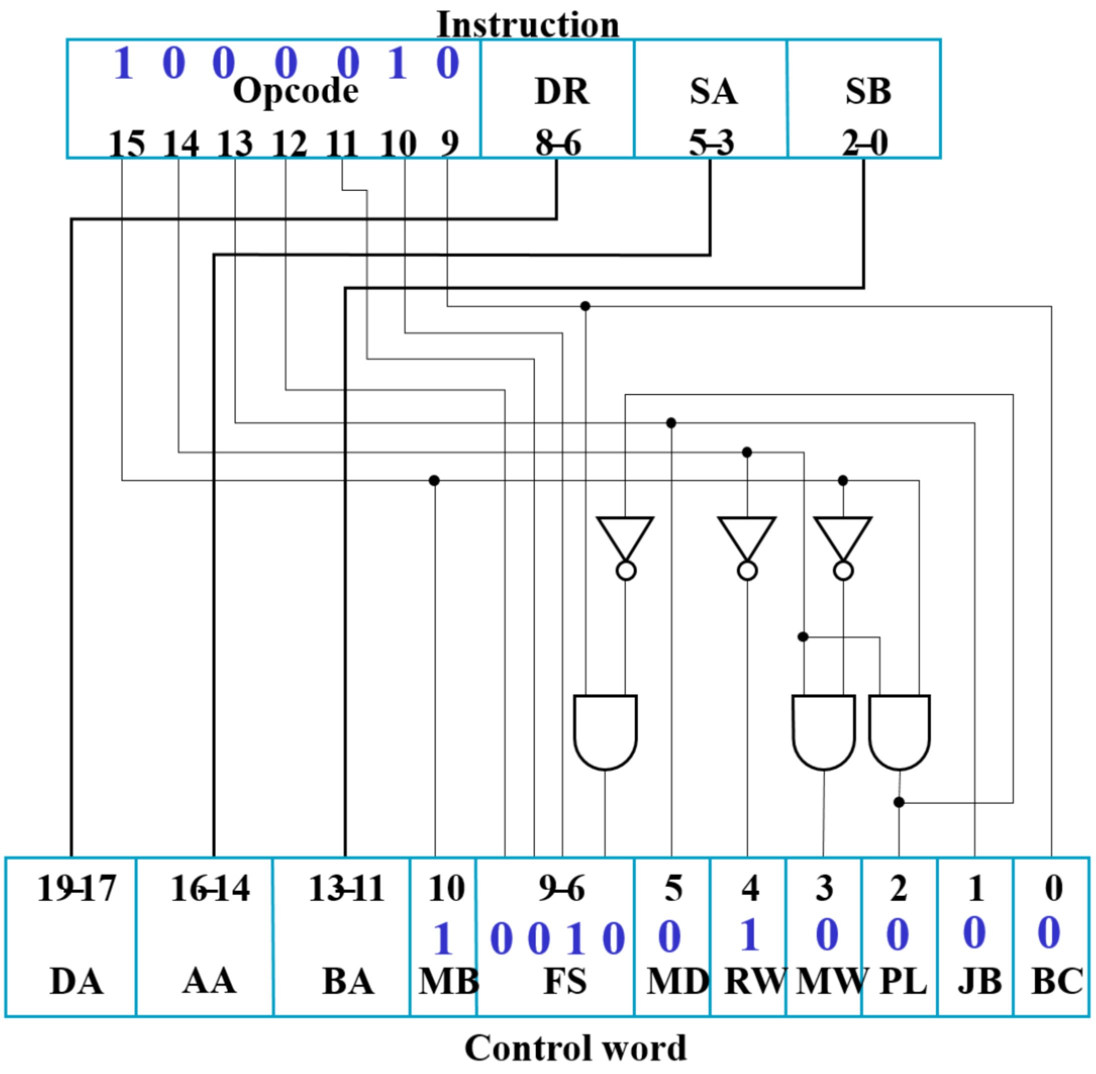


Figure 4: The deciphering of the ADI instruction code -- completed by professor Thweatt

Once converted the control words such as MB or FS can be used to follow the architecture and execution process of the simple computer. For example, looking at MUXB in figure 2 of the architecture while referring to the control word value of MB, which is one it is easy to see that the register BA will be coming out of MUXB. Therefore, MB is the select line or control value for the MUXB which determines whether a register or constant value will be used for the Function Unit or Data Memory. Whenever the most significant bit (bit-15) is 1 then the MUXB will send a constant through which means that the operation in the function unit will deal with a constant value. After the MB control word is the FS control word which represents the function select for the function unit to determine which arithmetic or logic operation to perform. In this case the FS is 0010 which is the function select for the ADD operation. Depending on the most significant bit (MSB) the add operation can be performing the operation between two registers or between a register and a constant. Next is the MD control word which is the select line for MUXD which determines whether or a value from the function unit or a value from the data memory is sent through. In the current case for the 1000010 opcode, the value assigned to MD is 0 which means that the output from the function unit will be sent through MUXD into the Register file. The last two are RW and MW. RW determines whether the output from MUXD will be written into a destination register or not (1 for write, 0 for no-write). MW determines whether the data coming from Bus B will be written into the address coming in from register A (1 for write, 0 for no-write). PL, JB, and BC are used for jumps and branches. For the most part if PL is 0 then JB and BC are don't-cares. The complete sequence of execution in the architecture can be seen through figure 5 below.

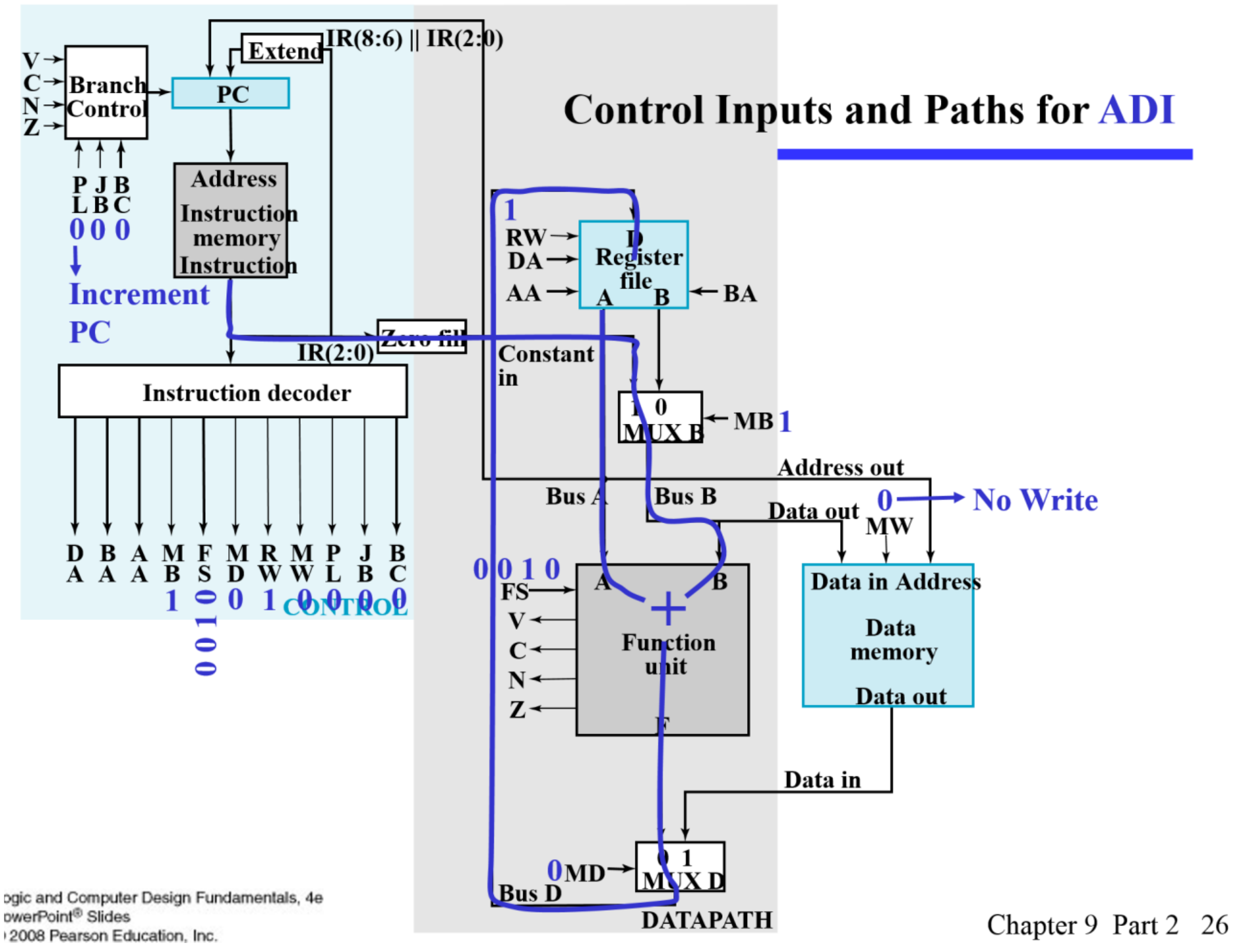


Figure 5: The complete sequence of execution after the opcode has been converted into the control words.

To create new instructions I had to use all of the information just talked about above and put it together to create my own instruction code.

DECISIONS

Assembly instruction	Opcode	Other fields of instruction (e.g., DR)	FS	MB	MD	RW	MW	PL	JB	BC
ADDINC	0000011	011 001 010	0011	0	0	1	0	0	x	x
ANDI	1001000	100 001 101	1000	1	0	1	0	0	x	x
SUBI	1000101	101 001 111	0101	1	0	1	0	0	x	x
REV	1000111	011 010 000	0111	1	0	1	0	0	x	x
JAL	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table 2: The new instruction code for the four new instructions excluding JAL

After understanding how an instruction code is used and how each bit is deciphered into control words next I went into creating new instruction code for the four new instructions shown in the table below (*table 2)*. In the project specification this is Table 4.

Recognizing that the given assembly instructions are ADDINC, ANDI, SUBI, REV, JAL(optional), I re-compiled the code and re-ran the simulation file. The results were as expected and were comprised of a set of unique values that were dependent on the PID inputted in location 0 of data.txt. The PID – stored in R1 - , is then incremented, and reversed in R4 to show that the code is working. This proves that the simulation file incorporates the logic implemented in table 2 above.

I next concluded that the new set of instruction codes are to be values that I will use to replace the existing instructions.txt file. The entire list of instructions that I had to implement are all in the table below (table 3).

Instruction Memory Address	**Instruction**
0	XOR R0, R0, R0
1	LD R1, R0
2	INC R2,R1
3	REV R3, R2
4	ADDINC R3, R1, R2
5	ANDI R4, R1, 5
6	SUBI R5, R1, 7
7	JAL R2, R0
8	MOVA R0, R0

Table 3: The full implementation program

Since these all represent a set of 16-bit instruction codes that include a 7bit-opcode and 3-bits each for the destination register, register A, and register B, I derived the new instruction code using the instructions in Table 3 (above).

I did this by using logic functions seen in figure 4 above while utilizing specific bits of the opcode to represent a control function. To illustrate the decisions that went into producing an acceptable output, I will go through the decision steps of one of the instructions provided. For instruction memory address 0, we have an instruction XOR R0, R0, R0 given to us. From the template provided in figure 4, I know that the first R0 register (bits 8-6) in the set of three is the destination register, where the second R0 is register A (bits 5-3), and the final R0 being the register B (bits 2-0).

The FS code for XOR was 1010 (looking at last four bits of opcode) seen from figure 3. The front 3 bits of the opcode before the FS, correlate to the MUXB, MUXD, RW, and MW. These values are all 0. Since MUXB has to be 0 in order for the Register B (R0) to go through into the function unit. MUXD is 1 so that the output from the functional unit gets sent to the Register File. RW has to be 1 to perform a read and write (which is a logical NOT of the value 0 which makes it 1).

ADDINC

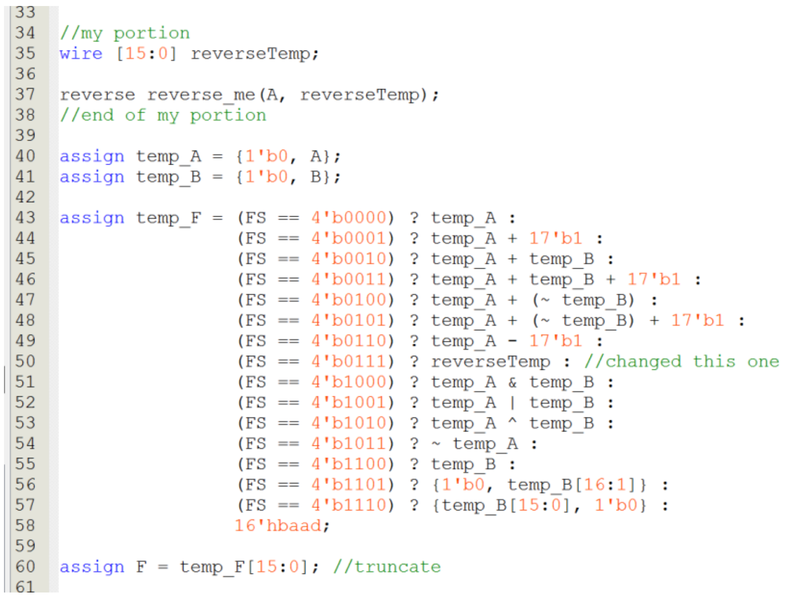
To briefly talk about my ADDINC opcode I used the ADDINC function already provided in the function unit (*line 46 in figure 6*).

ANDI

For my ANDI or ANDIMMEDIATE opcode I passed in a constant value (5) through MUXB and performed the AND operation already provided in the function unit (*line 51 in figure 6*). To do this I made sure that I set my MSB (bit-15) to 1.

SUBI

For my SUBI or SUBIMMEDIATE opcode I passed in a constant value (7) through MUXB and performed the SUB operation already provided in the function unit (*line 48 in figure 6)*. Since this was also an immediate where I pass in a constant I made sure that my MSB was 1.

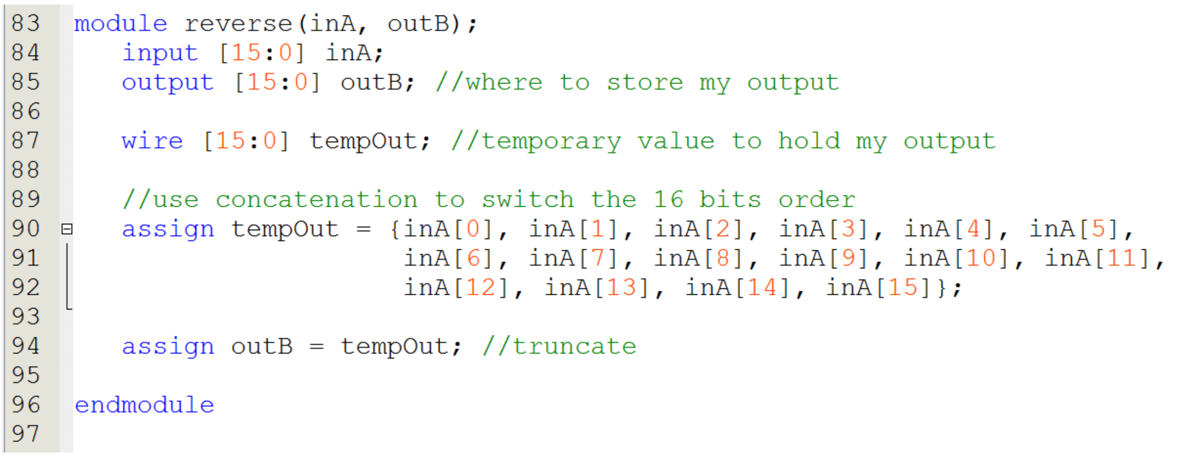


```
33
34  //my portion
35  wire [15:0] reverseTemp;
36
37  reverse reverse_me(A, reverseTemp);
38  //end of my portion
39
40  assign temp_A = {1'b0, A};
41  assign temp_B = {1'b0, B};
42
43  assign temp_F = (FS == 4'b0000) ? temp_A :
44                  (FS == 4'b0001) ? temp_A + 17'b1 :
45                  (FS == 4'b0010) ? temp_A + temp_B :
46                  (FS == 4'b0011) ? temp_A + temp_B + 17'b1 :
47                  (FS == 4'b0100) ? temp_A + (~ temp_B) :
48                  (FS == 4'b0101) ? temp_A + (~ temp_B) + 17'b1 :
49                  (FS == 4'b0110) ? temp_A - 17'b1 :
50                  (FS == 4'b0111) ? reverseTemp : //changed this one
51                  (FS == 4'b1000) ? temp_A & temp_B :
52                  (FS == 4'b1001) ? temp_A | temp_B :
53                  (FS == 4'b1010) ? temp_A ^ temp_B :
54                  (FS == 4'b1011) ? ~ temp_A :
55                  (FS == 4'b1100) ? temp_B :
56                  (FS == 4'b1101) ? {1'b0, temp_B[16:1]} :
57                  (FS == 4'b1110) ? {temp_B[15:0], 1'b0} :
58                 16'hbaad;
59
60  assign F = temp_F[15:0]; //truncate
61
```

Figure 6: The FS codes in my function_unit.v file

REV

For my REV or reverse opcode I decided to implement my function_unit Verilog file. I created a reverse module where I passed in the contents of Register A and outputted the reverse of Register A's content. To do this I concatenated the contents of Register A in reverse order. For example I used the concatenation operation '{}' and I went ahead and assigned each bit position the reversed bits positions of Register A (RA) by assigning "{RA[0], RA[1], …. , RA[15]}" to a temporary wire. By doing this the new wire has the reversed bits of RA since bit-15 in the new wire contains the content of bit-0 from RA and so on. My code is below (*figure 7*). Then I went ahead and called the module up in the main function unit module and replaced the FS 4'b0111 with the new reversebit and assigned my opcode accordingly (*refer to line 50 in figure 6 above)*.



```
module reverse(inA, outB);
   input [15:0] inA;
   output [15:0] outB; //where to store my output

   wire [15:0] tempOut; //temporary value to hold my output

   //use concatenation to switch the 16 bits order
   assign tempOut = {inA[0], inA[1], inA[2], inA[3], inA[4], inA[5],
                     inA[6], inA[7], inA[8], inA[9], inA[10], inA[11],
                     inA[12], inA[13], inA[14], inA[15]};

   assign outB = tempOut; //truncate

endmodule
```

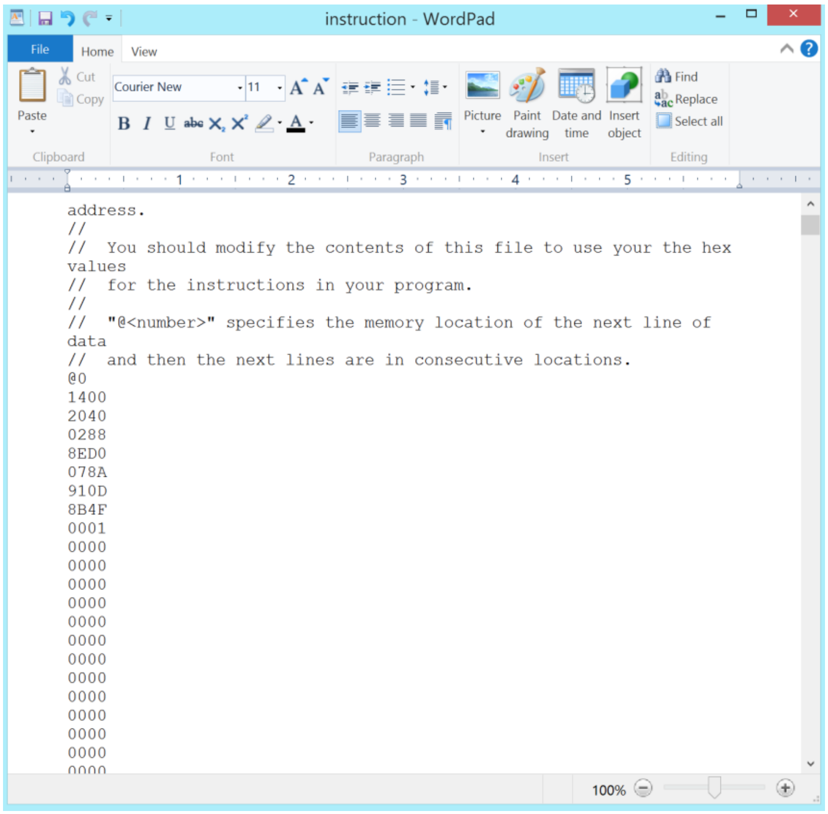
Figure 7: My reversed module where I used the method of concatenation to reverse my bits

After determining and then assigning all the instruction codes the complete Table 3 should look like Table 4 below.

Instruction Memory Address	**Instruction**	**Instruction Code**
0	XOR R0, R0, R0	1400
1	LD R1, R0	2040
2	INC R2,R1	0288
3	REV R3, R2	8ED0
4	ADDINC R3, R1, R2	078A
5	ANDI R4, R1, 5	910D
6	SUBI R5, R1, 7	8B4F
7	MOVA R0, R0	0001
8	MOVA R0, R0	------

Table 3: The completed instruction code for all the instructions needed for validation

The instruction text file would look like figure 8 below. Remembering that the order that the instruction hex code is in direct correlation to the memory address that it is stored in. For example 1400 is the instruction code for XOR R0, R0, R0 stored in address 0.



```
address.
//
//  You should modify the contents of this file to use your the hex
values
//  for the instructions in your program.
//
//  "@<number>" specifies the memory location of the next line of
data
//  and then the next lines are in consecutive locations.
@0
1400
2040
0288
8ED0
078A
910D
8B4F
0001
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
0000
```

Figure 9: The instruction.txt file for the validation

Finally, after compiling Verilog I ran my simulation and my results came out correct as shown below in figure 10. R1 contains my last four digits of my PID (0864). R2 is the incremented value of my PID (0865). R3 will contain my last four digits reversed (A610) and also the value of R1 plus R2 incremented (10CA). R4 contains the value of R1 AND'ed with the constant 5 (0004). R5 should contain R1 minus constant value 7 (085D). The simulation can be seen below in Figure 10.

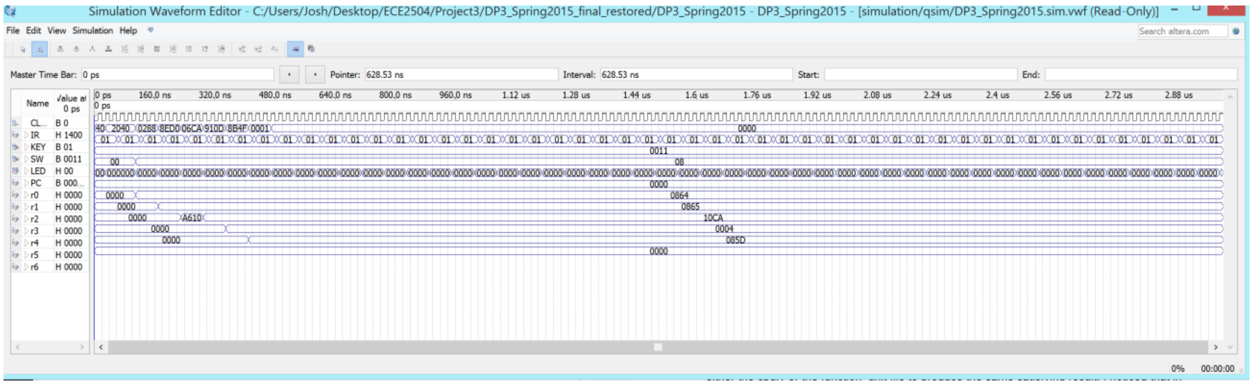


Figure 2: My simulation for my validation and instruction.txt

OBSERVATIONS

It is to be observed that the project required an understanding of the Verilog and operations it produced to get a correct simulated output. The Verilog given to us had included nearly all of the code required to complete the project besides the ‘reversal’. The observation I made while undertaking the task of figuring out the code was that it was possible to modify either the cpu.v or the function_unit file to produce the same satisfying result. I noticed that in the function_unit Verilog file, that the FS codes provided had a duplicate that we did not require. This was in FS code 0000 and 0111 where they were both equivalent to temp_A. This was a waste of an opcode possibility and since we were using the FS code 0000 as the main one, I decided to utilize the remainder FS 0111 to be my new reversal module. The reverse module was written in dataflow format and taken to be a simple assignment of the i-th value (starting at 0) of the chain to be the 15 – ith (read as 15 minus ith bit) value which thus created this reversal that will be performed on the incremented value of my PID stored in R2.

The most interesting observation made by me however was the engineering finesse that went into the operational code and how the instruction code was eventually deciphered to produce control words.

Conclusion

The project was an overall simple project after understanding the architecture of the simple computer given to us. The hardest part about this project was to understand the instruction codes and how the codes get deciphered to particular control words. The reverse instruction was fairly simple when using concatenation. Another challenging part of the project was getting the simulation to work because by changing the keys in the simulation. A lot of the project was given to us completed and what we had to do was go in and find the parts that suited our needs for the project. I ended up looking at the function_unit.v file multiple times to look for operations I can use for my needs by changing certain opcode bit-values.