

ECE2504 Design Project 3 Cover Sheet

Student Name: _____Boweï Zhao_____

Honor Code Pledge:

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

SIGNATURE


Project Item	Value	Points
Project Questions:		
• Table showing the opcode values that were chosen	5	
• Step 1 simulation results.	5	
• Completed table of disassembled 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	

Abstract

The project was to implement operations through an updated instruction set that we must make. Given to us, was a single-cycle central processing unit with most of the operational capability pre-written and ready for us. This ‘simple computer’ had functionality right off that bat that included the likes of adding, subtracting, and even loading for it. For the completion of the project, it was determined that there would have to be an additional component that I must build into modifiable files. These five additional instructions (with one being optional) made up the bulk of the code that we would have to write, produce, and or modify opcodes for in the execution of the project. We would therefore use a form of the Register Transfer Language and a standardized 16-bit instruction to produce a result that our FPGA DE0 Nano boards would then display.

Purpose

The purpose of the project was to create a working understanding of instruction sets and how they interface with these ‘simple computers’. This was done through giving us a fundamental approach towards instruction codes that were a set bit-length and contained independent values corresponding to an operation. This project will allow for hands-on experience in developing strategies for implementing Verilog code into modules to write good ‘software’ for good ‘hardware’. This, bundle with a reverse engineering mindset towards the instruction set, will allow one to problem solve engineering issues more efficiently and create a different outlook in the field of computer engineering.

Technical Approach

With regards to approaching the simple computer and its instruction set, it was imperative that one understood opcode operations and what was going on inside the computer. To start with, we were given a chart with instruction memory addresses and machine code that was incomplete. To get the rest of these values, I went into the Quartus file to find any files with the values in it. What I found was the instructions.txt document that contained the entirety of the list shown below in Table 1. These values were inputted into the table at the right memory address location up to Hex code E000. This ‘machine code’, which is in Hex, was then approached from an angle that it must represent a string of binary values that will then produce a 16-bit instruction that we can utilize.

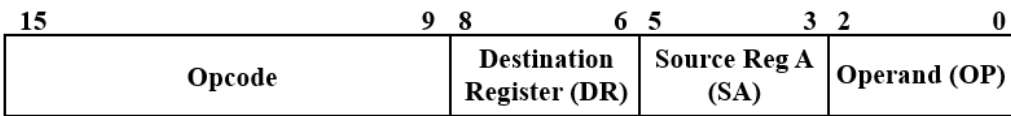
Table 1: Instruction and Machine Code

Instruction Memory Address	Machine code Value	Instruction (values in decimal)
0	1400	XOR R0, R0, R0
1	20C0	Load, R3, R0, R0
2	8444	ADI R1, R0, R4
3	0480	ADD R2, R0, R0
4	0290	INC R2, R2, R0
5	0C48	DEC R1, R1, R0
6	C00A	BRZ R0 R1, R2
7	C1C5	BRZ IR, R0 R5
8	0158	MOVA R5, R3, R0
9	0BAA	SUB PC, R5, R2

10	0368	INC, R5, R5, R0
11	0C48	DEC R1, R1, R0
12	1A41	SHR R1, R0, R1
13	C20A	BRN R0, R1, R2
14	E000	JUMP R0, R0, R0

From this machine code, we split up the binary values into the given instruction standard so that we can produce a meaningful outlook on what they truly represent. We see this instruction standard in figure 1 below, where the split in bits on the diagram relate to where we split the binary values we received from the hexadecimal machine code.

Figure 1: Instruction Standard

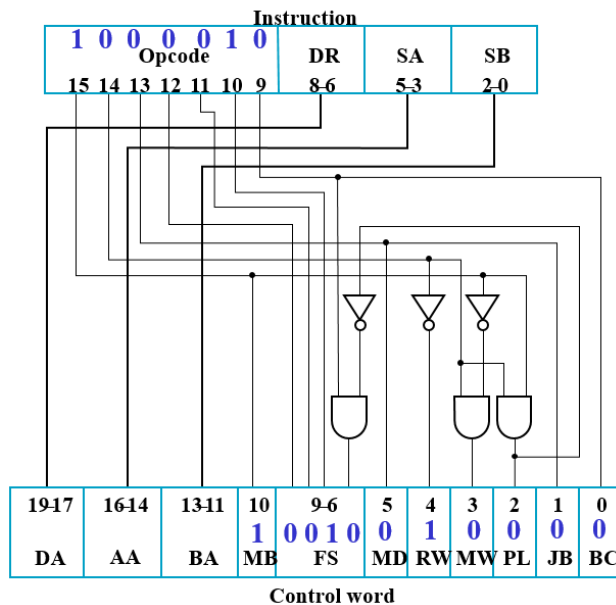


(b) Immediate

The first 7 bits (starting from the left) represent the opcode while the next three bits represent the Destination Register, then followed by another 3-bits for the Source Register A, and depending on the situation, it will be Operand or Operand B for either immediate or register use.

We may further decode this 16-bit instruction into 11 groups of control words that each represent an individual function seen in Figure 2 below. This can be promoted to the idea that each bit of that opcode, and subsequent registers correlate to a set of control words that initiate a response in the circuit.

Figure 2: Control Words to Instruction



Following through this diagram and control word schematic, we were able to finally produce the entirety of the contents in Table 1 where we, when given an hexadecimal machine

code, can convert it to binary to represent a set of functions and registers that produce a result on the FPGA when ran.

The values that we used were determined by a mix of Table 2 and Figure 3 below. The value of the three sets of 3-bit register values were determined through the provided table that allows us to conclude that binary values 0-5 represent the same value Register value whereas binary 6 is the program counter and 7 is the Instruction Register. The 9 combined bits of the register addresses were produced from this idea.

Table 2: Register Values

SW[3:1]	Value displayed on LEDs SW[0] selects between MSB (1) and LSB (0)
000	R0
001	R1
010	R2
011	R3
100	R4
101	R5
110	Program Counter (PC)
111	Instruction Register (IR)

To figure out the actual instructions (the first 7 bits), of the binary operations given, we used the document provided that gave us a set of instructions and their correlated opcodes and description of operations. Wherein we may now lay out the actual operation in Table 1 by comparing the first 7 (starting from left) bits of the binary converted hex to the opcode in Figure 3, and produce a relationship that puts the same values to be equal to each other. The only exception to this figure or rule is that there would be some operations that had a need for ‘don’t cares’, where it didn’t matter what operand A or operand B was. For the table, we would therefore have some source registers that have values that are not significant to the operation of the code. These will be random registers or codes, but will exist in the form that the binary converted hex gave to us and not changed in this case. They will only be changed later for when we implement the actual assembly instructions in Step 2.

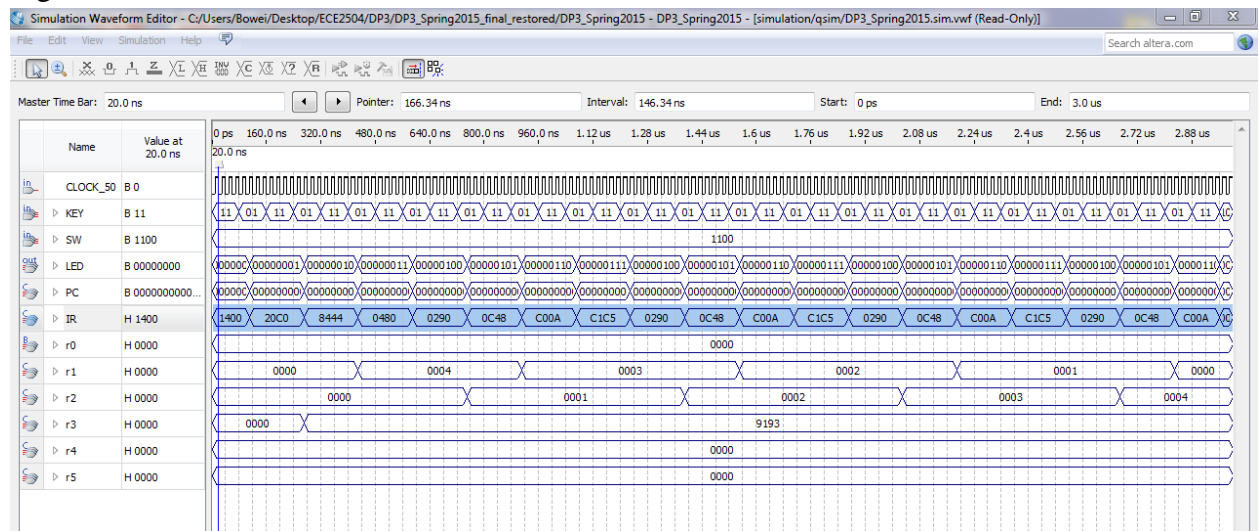
Figure 3: Instruction Opcode and Specifications

Instruction Specifications for the Simple Computer					
Instruction	Opcode	Mnemonic	Format	Description	Status Bits
MoveA	0000000	MOVA	RD, RA	$R[DR] \leftarrow R[SA]^*$	N, Z
Increment	0000001	INC	RD, RA	$R[DR] \leftarrow R[SA] + 1$	N, Z
Add	0000010	ADD	RD, RA, RB	$R[DR] \leftarrow R[SA] + R[SB]^*$	N, Z
Subtract	0000101	SUB	RD, RA, RB	$R[DR] \leftarrow R[SA] - R[SB]^*$	N, Z
Decrement	0000110	DEC	RD, RA	$R[DR] \leftarrow R[SA] - 1$	N, Z
AND	0001000	AND	RD, RA, RB	$R[DR] \leftarrow R[SA] \wedge R[SB]^*$	N, Z
OR	0001001	OR	RD, RA, RB	$R[DR] \leftarrow R[SA] \vee R[SB]^*$	N, Z
Exclusive OR	0001010	XOR	RD, RA, RB	$R[DR] \leftarrow R[SA] \oplus R[SB]^*$	N, Z
NOT	0001011	NOT	RD, RA	$R[DR] \leftarrow \neg R[SA]^*$	N, Z
MoveB	0001100	MOVB	RD, RB	$R[DR] \leftarrow R[SB]^*$	
Shift Right	0001101	SHR	RD, RB	$R[DR] \leftarrow sr\ R[SB]^*$	
Shift Left	0001110	SHL	RD, RB	$R[DR] \leftarrow sl\ R[SB]^*$	
Load Immediate	1001100	LDI	RD, OP	$R[DR] \leftarrow zf\ OP^*$	
Add Immediate	1000010	ADI	RD, RA, OP	$R[DR] \leftarrow R[SA] + zf\ OP$	N, Z
Load	0010000	LD	RD, RA	$R[DR] \leftarrow M[SA]^*$	
Store	0100000	ST	RA, RB	$M[SA] \leftarrow R[SB]^*$	
Branch on Zero	1100000	BRZ	RA, AD	if $(R[SA] = 0) PC \leftarrow PC + seA\ D$, N, Z	
Branch on Negative	1100001	BRN	RA, AD	if $(R[SA] < 0) PC \leftarrow PC + 1$	
Jump	1110000	JMP	RA	if $(R[SA] < 0) PC \leftarrow PC + seA\ D$, N, Z	
				$PC \leftarrow R[SA]$	

Upon the completion of this table, it was imperative that we run a simulation in Quartus to determine that our binary conversion was correct and that what we were meant to do was happening in the right order. To get the waveform working correctly, I determined that I would need to add registers 0, 3, 4, and 5 as these were the only register values missing from the sample waveform provided to us. This was approached and determined as the Table above only includes operations that go from register 0 to a register 5 and no more or less. So to create a meaningful and complete simulation, we needed to include all of those. Before the simulation was run, I realized that leaving the register values and Program Counter in binary form would be counter intuitive as it would be hard to compare to the table. I converted the formats from binary in Quartus to hexadecimal which will result in a much more observable simulation run.

The simulation result is seen below in Figure 4, where it is representative of a successful run of my code. This concludes to me that my binary conversion and table creation was correct and that I may move onto the implementation of the real project parts.

Figure 4: Initial Simulation Run



The second half of the project involves creating a set of new opcodes for custom instructions given to us. These new instructions, seen in Table 3, represent a set of operations that do not come pre-finished on the chart in Figure 3 and thus will require analysis of the functionality of our simple computer and the circuits behind it to decrypt.

Table 3: New Instruction Implementation

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

Whereas some of these operations are fairly standard, the majority are not, and so to approach this, I started looking at the Verilog code to find out what it was that I had to create. The search was over when I found the functional unit and the code within seen in Figure 5 below. This code shows a 16 to 1 multiplexer that will, given a 4-bit FS code (part of the 7-bit total opcode), perform a certain operation that it specified in the conditional statement of the code.

Figure 5: Function Unit Verilog Code

```
module function_unit
(
    input [3:0] FS,
    input [15:0] A,B,
    output V,C,N,Z,
    output [15:0] F
);

wire [16:0] temp_F; // 17 bits for zero extension
wire C_0, C_1;
wire [16:0] temp_A, temp_B; //zero extended

assign temp_A = {1'b0, A};
assign temp_B = {1'b0, B};

assign temp_F = (FS == 4'b0000) ? temp_A :
(FS == 4'b0001) ? temp_A + 17'b1 :
(FS == 4'b0010) ? temp_A + temp_B :
(FS == 4'b0011) ? temp_A + temp_B + 17'b1 :
(FS == 4'b0100) ? temp_A + (~ temp_B) :
(FS == 4'b0101) ? temp_A + (~ temp_B) + 17'b1 :
(FS == 4'b0110) ? temp_A - 17'b1 :
(FS == 4'b0111) ? temp_A :
(FS == 4'b1000) ? temp_A & temp_B :
(FS == 4'b1001) ? temp_A | temp_B :
(FS == 4'b1010) ? temp_A ^ temp_B :
(FS == 4'b1011) ? ~ temp_A :
(FS == 4'b1100) ? temp_B :
(FS == 4'b1101) ? {1'b0, temp_B[16:1]} :
(FS == 4'b1110) ? {temp_B[15:0], 1'b0} :
16'hbaad;
```

The Verilog code is the answer for the implementation for the few new instructions that we have below. Where all of them (except reverse), are a conjunction of one of these opcodes (or two for some), that will then allow us to just use the FS code given here but with a different 3-bit combinational head to produce that output.

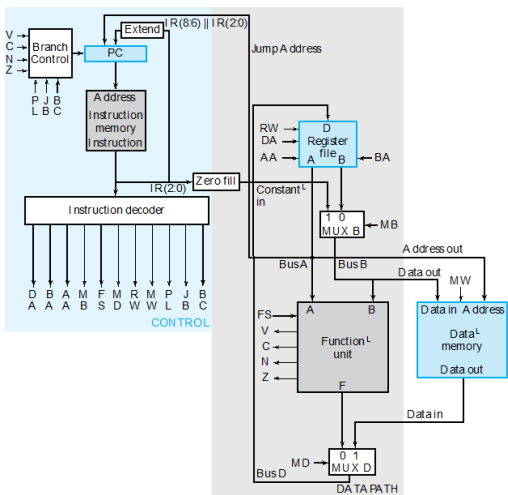
An example of this taken from table 3 would be that for instruction address 4, given ADDINC (Add Increment), there is no specific instruction in our pre-defined chart. But a closer look at the functionality of Add Increment results in where we need it to add and then add the value of 1 to it. This is present in FS code 0011 where operand A and B are added together along with a binary bit of 1 into it.

The production of all the FS codes in table 3 were created from this ideology where they all must come from a standard root in the FS code and where we can then make it different by changing the values of the first 3 bits which correlate to a few of the control group functions seen in figure 2.

The 3 bits in the front starting from the left can be determined through an analysis of the circuit in Figure 6 below and Figure 2 above. You may notice that Mux B is related to bit 15 which is the first of the 3 bits, Mux D to bit 13 which is the third of the 3 bits, and the rest being logical operations utilizing those same 3 bits. It uses a logical operation with gates and combinational circuits to reduce the schematic. While it is possible to use a 20-bit Instruction due to the fact that the entirety of the control word groups is 20-bits, it is inefficient. This is because you can give an instruction to a different group through using just three extra bits in front of the


FS and reducing it through logical combinational circuits shown in figure 2. In Figure 6, we see that the Mux's control whether we have values like constants coming in, whether it uses a register, if it is accessing the external memory unit, and which of the units we want to access. The three bits in the 16-bit instruction are imperative for the function of these control paths. It can thus be concluded that the final opcodes for the new instructions in table 3 are nearley the FS codes in the Verilog along with a introduction of slightly different front 3-bits to control the muxes and innate operation of the circuit.

Figure 6: Circuit Path Diagram



The final table I created seen in Table 4 below shows the values of opcodes and FS units that went into implementing this chart. The opcodes, destination, and source registers and their respective HEX codes are thus given and then loaded into the instructions.txt file in a new update.

Table 4: Full Instruction Implementation

Instruction Memory Address	Instruction	OPCODE	DR	SA	SB	HEX
0	XOR R0, R0, R0	0001010	000	000	000	1400
1	LD R1, R0	0010000	001	000	000	2040
2	INC R2, R1	0000001	010	001	000	0288
3	REV R3, R2	0001111	011	010	000	1ED0
4	ADDINC R3, R1, 	0000011	011	001	010	06CA
5	ANDI R4, R1, 5	1001000	100	001	101	910D
6	SUBI R5, R1, 7	1000010	101	001	111	854F
7	MOVA R0, R0	0000000	000	000	000	0000

The only thing remaining that we have/are not able to implement from the standard code is the reversal. From what I saw, I deduced that it was possible to edit either the cpu.v file or the function_unit.v file to get a reversal going. Based on time constraints, I decided that going through it from the cpu.v with a new module is the best way to do. Seen in figure 7, my Verilog code shows the path I used to implement this function.

Figure 7: Reverse Module

```
//mux_d
mux_mux_d
(
    .din_1(data_in_bus),
    .din_0(d2),
    .sel(MD),
    .q(register_file_in)
);

// Make the instruction register value visible outside the CPU.
assign IR = instr;

reversemodule revname(A, OUTtemp);
wire [15:0] OUTtemp;
wire [15:0] d2;
assign d2 = (instr[15:9] == 7'b0001111) ? OUTtemp : function_unit_out;

endmodule
// End of CPU module

module reversemodule (INr, tempOut);
input [15:0] INr;
output [15:0] tempOut;
wire [15:0] OUTr;

assign OUTr[0] = INr[15];
assign OUTr[1] = INr[14];
assign OUTr[2] = INr[13];
assign OUTr[3] = INr[12];
assign OUTr[4] = INr[11];
assign OUTr[5] = INr[10];
assign OUTr[6] = INr[9];
assign OUTr[7] = INr[8];
assign OUTr[8] = INr[7];
assign OUTr[9] = INr[6];
assign OUTr[10] = INr[5];
assign OUTr[11] = INr[4];
assign OUTr[12] = INr[3];
assign OUTr[13] = INr[2];
assign OUTr[14] = INr[1];
assign OUTr[15] = INr[0];

assign tempOut = OUTr;

endmodule
```

I implemented this by first making a reverse module that took in a 16-bit input and produced a 16-bit output and reversed the order of the bits. I then modified Mux D above in the form of the 0 input of it so that when I call the FS opcode of my new reverse module which is 1111 (an empty opcode that Function Unit did not use), then it will run my reverse code.

Finally, for validation, we are required to only validate and fully implement a few of the total instructions that they gave us anew in step 2. These were ADDINC, ANDI, SUBI, REV and the optional JAL that I did not do.

The final chart that was required of us to create to understand the implementation is below in Table 5. Where we were able to find the FS, MB, MD, RW, MW, PL, JB, and BC thanks to figure 6 and 2 where we go down the pathways to find the values that each operation requires for the Muxes and branches.

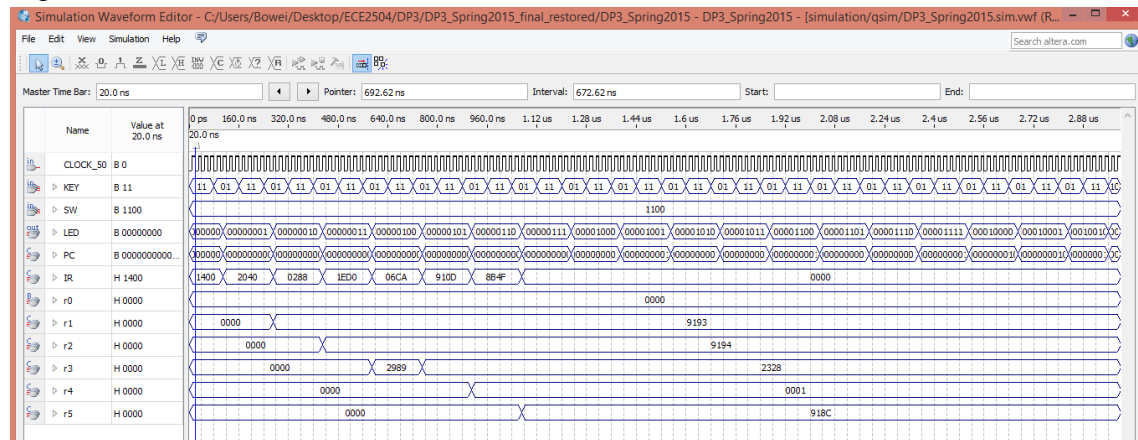
Table 5: Implemented Set of Instructions

Assembly Instruction	PC HEX	OPCODE HEX	Other Instruction	FS	MB	MD	RW	MW	PL	JB	BC
ADDINC		0000011	06CA	011 001 010	0011	0	0	1	0	0	1
ANDI		1001000	910D	100 001 101	1000	1	0	1	0	0	0
SUBI		1000010	854F	101 001 111	0010	1	0	1	0	0	0
REV		0001111	1ED0	011 010 000	1111	0	0	1	0	0	1

With this fully in place, the final simulation run to simulate what the FPGA board can produce was created and run with good results seen in Figure 8. This simulation shows that my PID is displaying, incrementing, reversing, and being stored correctly in the registers after following the instructions. It also shows two sets of values for R3 at different PC intervals which is also correct.

The Technical Approach ends at this point with all simulations and charts run, and a meticulous understanding of the intricacies that the system undergoes when the code runs and my logical processes that I underwent.

Figure 8: Final Simulation Run



Results

Upon seeing the results in Figure 4 and 8 representing the two simulation runs, it was decided that these were successful modelsim outputs given what the FPGA was taking in from data and instructions.txt. This is important as it shows an execution and understanding of how FS codes are applied to the system and the way multiplexers will operate with these units to execute.

I was able to get a result with the FPGA Nano board ultimately by using Key 1 and 0 presses on the board and by following the instructions to read through different registers and program counter intervals.

All the lights on the DE0 board light up as I would expect with the right bits in the right places performing in an order I so set defined in my declaration. This was interesting in that it showed that my logic was correct, and that my operations were executed correctly.

Conclusions

It can therefore be concluded that the project represented an exercise in computing that allowed students like myself to implement an instruction set and its ultimate operation on an FPGA. This was of great value to my education as it made use of the skills I have acquired over the year in the class in a near finale project.

I feel that the project allowed for an application of skills into an real world situation and project that I may one day have to do. What I received from the project was an important proficiency in the Verilog HDL and an acknowledgement of the importance of instruction sets in computing.

A final observation of mine would be that it was interesting how many different files this project used in conjunction with each other. There were a plethora of Verilog complimentary files that we were given and that were used together. This was interesting as it increased processing and compilation time of the code from mere seconds to minutes on my systems.

Overall, this project was done well with good results. It led to an increase in my understanding of the topic and will ultimately affect my future career.