Project 3

Jacob Abel

November 29, 2017

This goal of this project was to successfully disassemble CPU op-codes into assembly and implement several new instructions based on knowledge gained from the initial disassembly. This requires application of basic combinational and sequential logic design skills, knowledge of verilog syntax and style, and a comprehensive understanding of basic computer architecture and instruction formats.

The final deliverables for this project include the modifications necessary to implement the instructions in table 1 and the disassembly of 15 instructions specified in the project specification. The disassembled instructions can be found in table 2.

The newly implemented instructions in table 1 are implemented with minimal changes to the original source to perform the following operations. ADDINC adds the contents of R[A] and R[B], increments the sum, and stores the result in R[D]. SUBI subtracts a hard coded constant OP from R[A] and stores the result in R[D]. NEGI stores the negative of a hard coded constant OP in R[D]. NAND performs a not and operation on R[A] and R[B]. The result is stored in R[D]. The JAL operation stores the address following the current program counter address in R[B] and performs a jump to the location stored in R[A]. The JAL operation allows programs to return to the original location following a jump. One use of this operation is to provide the faculties for function calls by performing a JAL and then later a jump to the saved address at the end of the function. The specification labelled this operation as optional however it is fully implemented and functional.

Instruction	Mnemonic	Format	Description
Add and Increment	ADDINC	RD, RA, RB	$R[DR] \leftarrow R[SA] + R[SB] + 1$
Subtract Immediate	SUBI	RD, RA, OP	$R[DR] \leftarrow R[SA]$ – zf OP
Negate Immediate	NEGI	RD, OP	$R[DR] \leftarrow$ – zf OP
NAND	NAND	RD, RA, RB	$R[DR] \leftarrow \overline{R[SA] \wedge R[SB]}$
Jump and link	JAL	RA, RB	$PC \leftarrow R[SA], R[SB] \leftarrow PC + 1$

Table 1: Instructions to be implemented from the Project 3 Specifications

The initial stage of the project is the disassembly of a series of instructions and the analysis of pre-existing source before implementing the new instructions. The first part of disassembly was to convert all of the hexadecimal instructions to binary. After that all the operator flags(FS, MB, MD, RW, MW, PL, JB, BC) in the instructions were worked out by separating the binary based on the verilog in instr_decoder.v and matching the instructions to their corresponding assembly op-codes based on the operator flags. Once the op-codes had been identified, the instructions were put into their specific formats and the register addresses and hard coded constants were converted from binary to decimal. At this point the instructions were fully disassembled and are available in table 2. Following this, simulations of the design performing the instructions were run in Quartus and recorded. The annotated simulations are available in Appendix A: Simulation Waveforms, table 1. They were then cross-referenced against the table to verify that the disassembled instructions were valid.

Memory Address	Machine Code	Instruction (Values in decimal)
0	1400	XOR R0, R0, R0
1	2100	LD R4, R0
2	8443	ADI R1, R0, 3
3	0480	ADD R2, R0, R0
4	00E0	MOV R3, R4
5	0290	INC R2, R2
6	0ADA	SUB R3, R3, R2
7	0C48	DEC R1, R1
8	C00A	BRZ R0, R1
9	C1C4	BRZ R7, R0
A	02D8	INC R3, R3
B	0C48	DEC R1, R1
C	1A41	SHR R1, R1
D	C20A	BRN R0, R1
E	E000	JMP R0

Table 2: Initial Instruction Disassembly

Going into the design the goal was to provide as minimal changes to the source as possible. Due to this there are a number of lines where rather than introduce a new multiplexer, an existing multiplexer could have been expanded.

The first step in designing the new instructions was to identify the existing capabilities of the hardware. This was largely completed in the previous section however there were some additional steps that were necessary to completely identify the full capabilities of the hardware. By working out what type of operations every FS combination performed, it was discovered that several undocumented op-codes either nearly or already completely implemented the new instructions.

ADDINC only required flipping bit 0 of FS for the ADD instruction to implement. SUBI only required flipping the MB bit of the SUB instruction. NEGI utilised an undocumented instruction that inverted the constant and therefore the only required addition was to add a 2x1 mux for the A operand that toggled between providing A or the constant 1.

NAND was determined to be more complicated as the only FS code that also performed the AND operation in the logic portion of the function unit was a undocumented shift operation. As such additional cases had to be added to the mux for toggling between the logic, arithmetic, and shift units. All that remained was to negate the logic unit output and attach the negated output to the newly added port in the function unit mux.

The JAL operation was an entirely different process to design compared to the other operations as it was a control operation rather than a function operation. This operation required a restructuring of the PC controller arguments list as it required the addition of a new output containing the saved return address. Additionally it required adding to the multiplexer logic for the register input so to connect the new PC controller output to the registers.

For reference all source code modifications are available in Appendix B: Verilog Modification Listings.

Assembly Instruction	HEX	Fields	FS	MB	MD	RW	MW	PL	JB	BC
ADDINC	04xx	RD, RA, RB	0011	0	0	1	0	0	0	1
SUBI	8Axx	RD, RA, OP	0101	1	0	1	0	0	0	1
NEGI	88xx	RD, OP	0100	1	0	1	0	0	0	0
NAND	1Exx	RD, RA, RB	1111	0	0	1	0	0	0	1
JAL	A2xx	RA, RB	0001	1	1	1	0	0	1	1

Table 3: Implemented Operations

To validate the design following completion simulations were run in Quartus and recorded. The annotated simulation is available in Appendix A: Simulation Waveforms table 2. The simulation results were cross-referenced against the expected results of instructions and were proven correct. Additionally the instructions were tested on a development board and worked as intended.

This project was a good exercise in working with designing a CPU. It provides a decently complex task to be completed that doesn't take too much work to do properly. Additionally it provides experience working with pre-existing code which will be the most common situation in the industry. The documentation is clear and concise. There was nothing that was too difficult to do without reading the documentation once or twice.

Given time to rework the project, I would likely try to provide more detailed breakdowns of the disassembly analysis and instruction design process. Additionally there are several lines that could be condensed from multiple muxes into one mux however this is only a minor blemish in the code. Otherwise I believe that the project was a success.

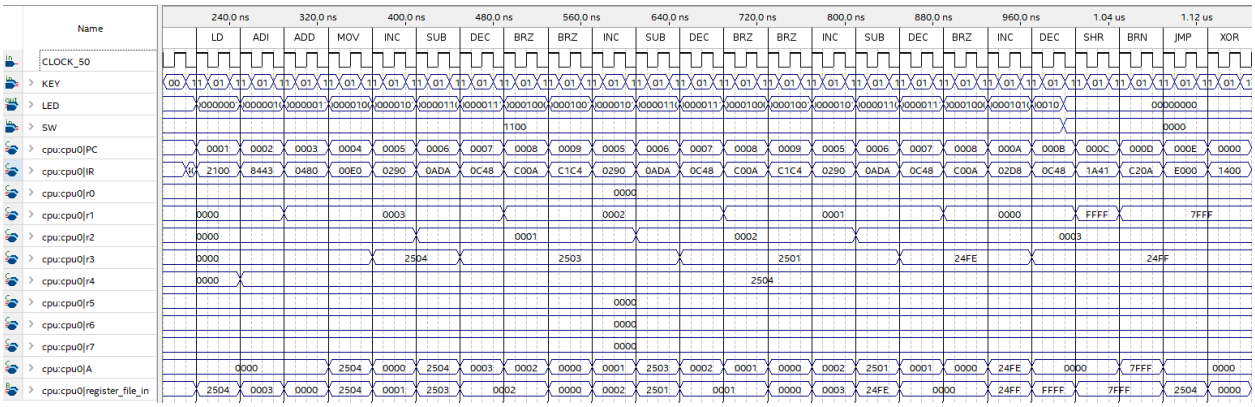

Figure 1: Initial Simulation Waveform

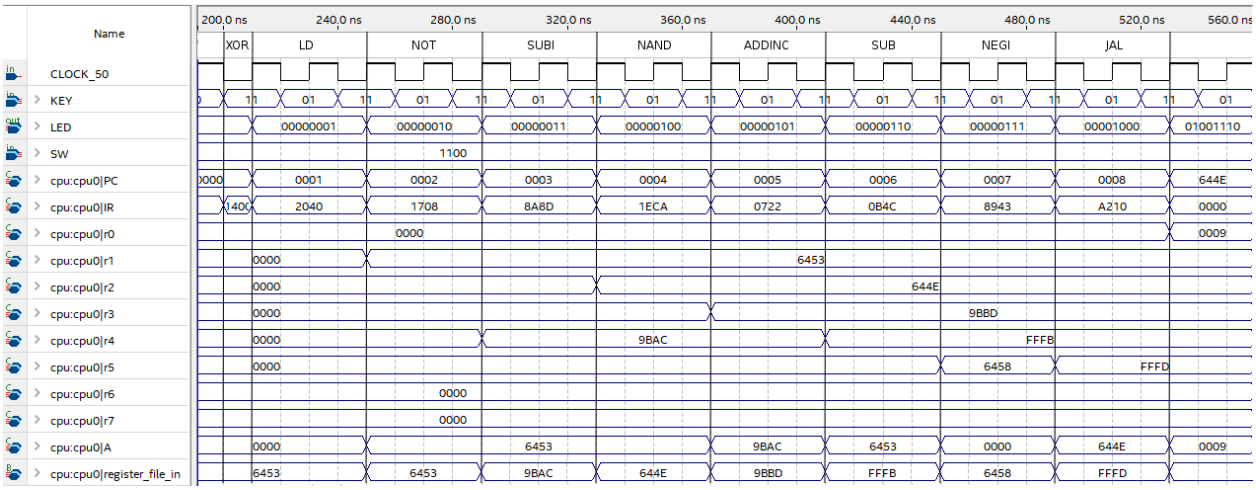

Figure 2: Operator Validation Simulation Waveform

```
-module pc_controller(rst,clk,N,C,V,Z,PL,JB,BC,PC,ld_pc,jp_addr);
+module pc_controller(rst,clk,N,C,V,Z,PL,JB,BC,PC,ld_pc,jp_addr,ret_pc);
     input             rst;
     input             clk;
     input             N, Z, V, C, PL, JB, BC;
     input [15:0]      ld_pc, jp_addr;
     output reg [15:0] PC;
+    output [15:0]     ret_pc;
     wire [3:0]        status;
     wire [3:0]        ctrl_pc;
     reg  [15:0]       next_pc;

     assign ctrl_pc = {rst, PL, JB, BC};  // Concatenate the PC control
         bits
+    assign ret_pc = PC + 1;
```

```
always@(ctrl_pc or jp_addr or N or Z or PC or ld_pc) begin
    casex(ctrl_pc)
        4'b1xxx: next_pc = 16'h0000;  // PC = 0 on reset.
        4'b011x: next_pc = jp_addr;   // For the JUMP instruction, PC =
            jp_addr.
+       4'b0011: next_pc = jp_addr;
   // For the JAL instruction, PC = jp_addr. ret_pc = PC + 1
        4'b0101: begin                // For Branch instructions, look
           at the status bits.
```

Figure 3: pc_controller.v Changes

```
    wire [15:0] data_mem_out          /* synthesis keep */;
    wire [5:0]  ad;                   // Address offset.
    wire [15:0] se_ad;                // Sign-extended address offset.

+   wire [15:0] return_addr_reg       /* synthesis keep */;
    wire [15:0] data_in_bus           /* synthesis keep */;
```

```
    //Instantiate the PC controller.
    pc_controller pc_ctrl
    (
        .rst(rst),
        .clk(clk),
        .N(N),
        .C(C),
        .V(V),
        .Z(Z),
        .PL(PL),
        .JB(JB),
        .BC(BC),
        .PC(PC),
        .ld_pc(se_ad),
-       .jp_addr(A)
+       .jp_addr(A),
+       .ret_pc(return_addr_reg)
    );
```

```
    defparam data_mem.DATA_WIDTH = 16;
    defparam data_mem.ADDR_WIDTH = 8;
    single_port_ram data_mem(.data(mux_b_out), .addr(A[7:0]), .we(MW),
       .clk(clk), .q(data_mem_out));

    // Take the data memory out as an input to multiplexer D.
-   assign data_in_bus = data_mem_out;
+   assign data_in_bus = (JB && BC) ? return_addr_reg : data_mem_out;

    // Instatiate multiplexer D.
    mux mux_d
    (
        .din_1(data_in_bus),
        .din_0(function_unit_out),
        .sel(MD),
        .q(register_file_in)
    );
```

Figure 4: cpu.v Changes

```
31      // This is the select line for the Function Unit's output
           multiplexer.
32  -   wire           MF;
33  +   wire           MF, NM;
```

```
37      // These 17-bit vectors are results that include a carry-out.
38  -   wire [16:0]    arith_out, logic_out, shift_out;
39  +   wire [16:0]    arith_out, logic_out, shift_out, nand_out;
40      wire [16:0]    temp_G, temp_F;
```

```
43      // The adder operands are zero-filled to 17 bits to generate a
           carry-out.
44  -   assign arith_A    = {1'b0, A};
45  +   assign arith_A    = (FS[2:0] == 3'o4 ) ? 17'h0001    : {1'b0, A};
46      assign arith_B    = (FS[2:1] == 2'b00) ? 17'b0       :
```

```
52  // The logic circuit uses FS[1:0] to perform (AND, OR, XOR, Complement)
53  -   assign logic_out = (FS[1:0] == 2'b00) ? {1'b0, A} & {1'b0, B} :
54  +   assign logic_out = ( (FS[1:0] == 2'b00)
55  +                       ||(FS[2] == 1'b1)) ? {1'b0, A} & {1'b0, B} :
56                          (FS[1:0] == 2'b01) ? {1'b0, A} | {1'b0, B} :
57                          (FS[1:0] == 2'b10) ? {1'b0, A} ^ {1'b0, B} :
58                          (FS[1:0] == 2'b11) ? {1'b0, ~A}            : 17'
                               bxxxxxxxxxxxxxxxx;
```

```
60      assign temp_G     = (FS[3] == 1'b0) ? arith_out : logic_out;
61
62  +   assign nand_out   = ~logic_out;
63
64      // The Shifter uses FS[1:0] to perform (B, sr B, sl B).
```

```
69      // The mux that chooses between the ALU and shifter has a select
           equal to (FS[3] & FS[2]).
70      assign MF = FS[3] & FS[2];
71  +   // NAND Operator Mode Flag
72  +   assign NM = FS[1] & FS[0];
73
74      // The mux on the Function Unit output uses MF to perform (ALU,
           Shifter).
75  -   assign temp_F = (MF == 1'b0) ? temp_G : shift_out;
76  +   assign temp_F = (MF == 1'b0) ? temp_G    :
77  +                   (NM == 1'b0) ? shift_out : nand_out;
```

Figure 5: function_unit.v Changes