

Northeastern University

Department of Electrical and Computer
Engineering

EECE2323: Digital Systems Design Lab

Lecturer: **Dr. Emad Aboelela**

TAs:

Ke Chen

Linbin Chen

**Lab # 6 - 8: Adding Instruction Decoding to the
Datapath, Adding Instruction Memory and
Program Counter to Your Computer, Adding
Branch Logic to the Datapath to Complete Your
Computer**

Group # 14:

Jin Hyeong Kim,
Timothy VanSlyke

Semester: Spring 2018

Date: April 16, 2018

Lab Session: Tuesday, 1:00PM

Lab Location: 9 Hayden Hall, Northeastern University,
Boston, MA 02115

Contents

1	Introduction	2
2	Design Approach	3
2.1	Instruction Set Architecture	3
2.2	Implementing the ISA in Hardware	4
2.2.1	Instruction Decoder	4
2.2.2	Program Counter with Conditional Branching	5
3	Results and Analysis	6
3.1	Design Simulation	6
3.2	Hardware Testing	6
4	Conclusions	10
	Appendices	11
A	Instruction Decoder Verilog Module	11
B	Software Multiply Implementation	15

1 Introduction

In these experiments, we will investigate the problem of introducing an instruction dispatch system to our existing computer. We will implement a minimal instruction set architecture (ISA) with a MIPS-inspired encoding and mnemonics. We will equip our computer with an instruction decoder, a persistent instruction memory component, and facilities for conditional instruction jumps. The instruction decoder, instruction memory + program counter, and branching facilities will be implemented in Verilog and merged into our existing design.

Our experiments will demonstrate the feasibility of our system by showing hardware testing of manual single-instruction dispatch, and automatic, sequential, multiple-instruction dispatch from instruction memory. Additionally, we will show that our computer is capable of executing arbitrary programs written in our MIPS-like assembly language.

2 Design Approach

2.1 Instruction Set Architecture

Our ISA encoding follows the conventions of MIPS with R-Type, I-Type, and J-Type instruction categories. Since our computer cannot correctly implement the MIPS computer model, our encoding diverges slightly from the traditional MIPS encoding.

We choose to encode our instructions in a 16-bit double word. Since our computer has only 4 registers, all encodings of source and destination registers use 2-bit slices of the 16-bit encoding. Our limited instruction set allows us to encode all ALU op-codes in only 4-bit slices. The R-Type instruction encoding concatenates (from MSB to LSB) a 4-bit op-code followed by three 2-bit register encodings. The first encoded register is the first source register, the second encoded register is the second source register, and the last register is the destination register in which the ALU output is stored. The last six unused padding bits are zeroed in all R-Type encodings.

Opcode	RS	RT	RD	Padding
4 Bits	2 Bits	2 Bits	2 Bits	6 Bits

Figure 1: Encoding of an R-Type instruction for our ISA.

Both I-Type and J-Type encodings are identical in their format. A 4-bit opcode precedes the source and destination register encodings. The last 8 bits are used to encode a single-word immediate which is interpreted as an 8-bit two's complement signed integer. For I-Type instructions, this immediate is a numeric literal used by the ALU to perform its computation, while for J-Type instructions, the immediate is the signed distance to the jump destination.

Opcode	RS	RT	Immediate
4 Bits	2 Bits	2 Bits	8 Bits

Figure 2: Encoding of an I-Type instruction for our ISA.

Opcode	RS	RT	Immediate
4 Bits	2 Bits	2 Bits	8 Bits

Figure 3: Encoding of a J-Type instruction for our ISA.

2.2 Implementing the ISA in Hardware

2.2.1 Instruction Decoder

Our instruction decoder implementation uses a simple case statement to switch between the following instruction opcodes:

- LW = 0b0000
- SW = 0b0001
- ADD = 0b0010
- ADDI = 0b0011
- INV = 0b0100
- AND = 0b0101
- ANDI = 0b0110
- OR = 0b0111
- ORI = 0b1000
- SRA = 0b1001
- SLL = 0b1010
- BEQ = 0b1011
- BNE = 0b1100
- CLR = 0b1101

Each case has a hard-coded set of values that are fed to the ALUOp, RegDst, RegWrite, ALUSrc1, ALUSrc2, MemWrite, and MemToReg outputs. Additionally a straight verilog assign statement hardwires slices of the encoded instruction's bit-sequence to the opcode, RS address, RT address, RD address, and immediate outputs. The module's implementation is exceptionally simple but long-winded.

2.2.2 Program Counter with Conditional Branching

The program counter implementation is shown inline here:

```
module PC_Logic (  
    input clk,  
    input rst,  
    input [7:0] ofs,  
    input take_branch,  
    output reg [7 : 0] counter  
);  
    always @( posedge clk)  
        counter <= (rst ? 0 : counter + (take_branch ? ofs : 1));  
endmodule
```

The program counter implementation is rather simple. The module unconditionally writes to the counter output. In the event that the `rst` input is on, the ternary operator selects 0 to be written to the counter, otherwise if the `take_branch` input is on a second ternary operator selects the sum of counter and `ofs` to be written. In the event that neither `rst` nor `take_branch` are on, counter is simply assigned its previous value plus one.

3 Results and Analysis

3.1 Design Simulation

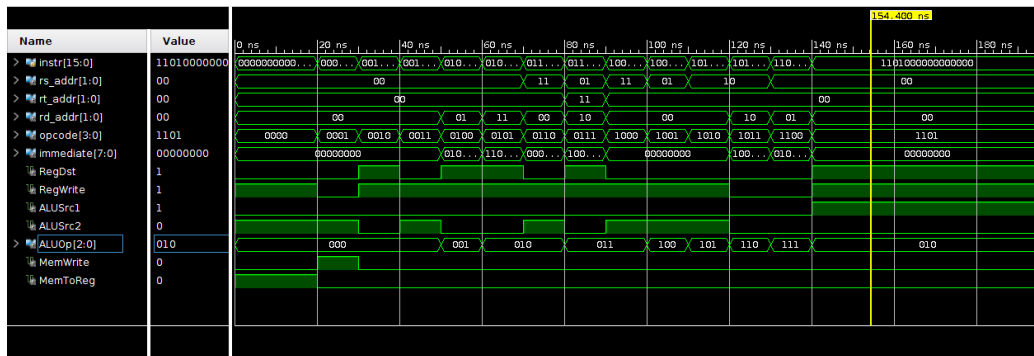
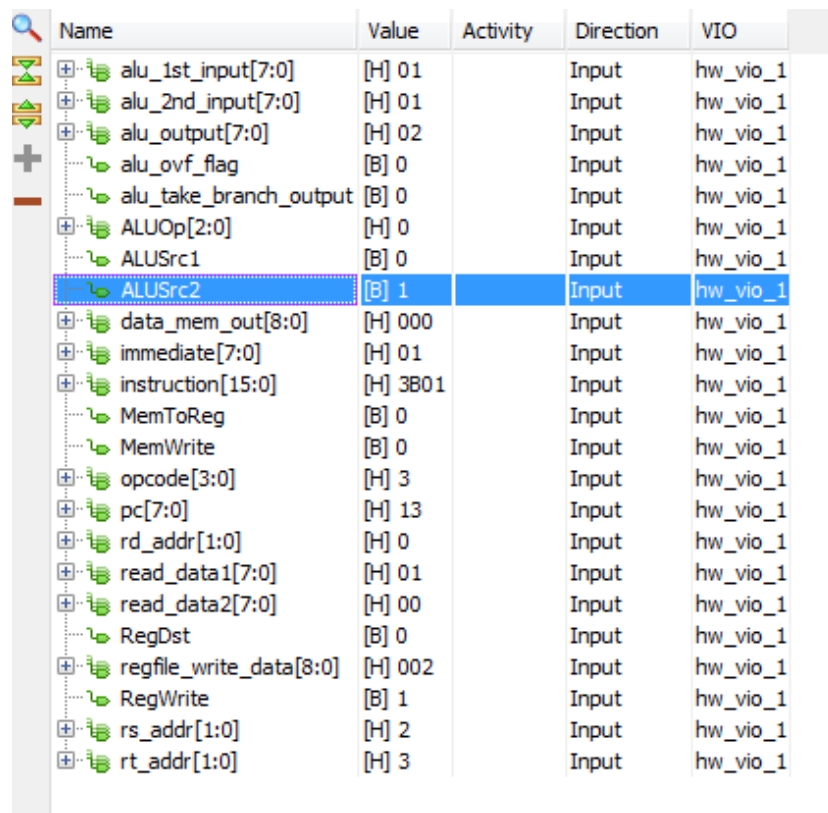


Figure 4: Simulation/test of our computer during the sixth experiment. The test shows that the Verilog implementation of our instruction decoder has the correct behavior.

3.2 Hardware Testing

Name	Value	Activity	Direction	VIO
alu_1st_input[7:0]	[H] FE		Input	hw_vio_1
alu_2nd_input[7:0]	[H] F0		Input	hw_vio_1
alu_output[7:0]	[H] FE		Input	hw_vio_1
alu_ovf_flag	[B] 0		Input	hw_vio_1
alu_take_branch_output	[B] 0		Input	hw_vio_1
data_mem_out[8:0]	[H] 000		Input	hw_vio_1
instruction[15:0]	[H] 8AF0		Output	hw_vio_1
opcode[3:0]	[H] 8		Input	hw_vio_1
read_data1[7:0]	[H] FE		Input	hw_vio_1
read_data2[7:0]	[H] FE		Input	hw_vio_1
regfile_write_data[8:0]	[H] 0FE		Input	hw_vio_1

Figure 5: Hardware VIO testing results for the instruction decoder (lab 6).



Name	Value	Activity	Direction	VIO
alu_1st_input[7:0]	[H] 01		Input	hw_vio_1
alu_2nd_input[7:0]	[H] 01		Input	hw_vio_1
alu_output[7:0]	[H] 02		Input	hw_vio_1
alu_ovf_flag	[B] 0		Input	hw_vio_1
alu_take_branch_output	[B] 0		Input	hw_vio_1
ALUOp[2:0]	[H] 0		Input	hw_vio_1
ALUSrc1	[B] 0		Input	hw_vio_1
ALUSrc2	[B] 1		Input	hw_vio_1
data_mem_out[8:0]	[H] 000		Input	hw_vio_1
immediate[7:0]	[H] 01		Input	hw_vio_1
instruction[15:0]	[H] 3B01		Input	hw_vio_1
MemToReg	[B] 0		Input	hw_vio_1
MemWrite	[B] 0		Input	hw_vio_1
opcode[3:0]	[H] 3		Input	hw_vio_1
pc[7:0]	[H] 13		Input	hw_vio_1
rd_addr[1:0]	[H] 0		Input	hw_vio_1
read_data1[7:0]	[H] 01		Input	hw_vio_1
read_data2[7:0]	[H] 00		Input	hw_vio_1
RegDst	[B] 0		Input	hw_vio_1
regfile_write_data[8:0]	[H] 002		Input	hw_vio_1
RegWrite	[B] 1		Input	hw_vio_1
rs_addr[1:0]	[H] 2		Input	hw_vio_1
rt_addr[1:0]	[H] 3		Input	hw_vio_1

Figure 6: Hardware VIO testing results for our instruction memory implementation (lab 7). The VIO shows the end result of running a simple program written in our custom MIPS-like assembly language.

Name	Value	Activity	Direction	VIO
alu_1st_input[7:0]	[H] DE		Input	hw_vio_1
alu_2nd_input[7:0]	[H] 00		Input	hw_vio_1
alu_output[7:0]	[H] DE		Input	hw_vio_1
alu_ovf_flag	[B] 0		Input	hw_vio_1
alu_take_branch_output	[B] 0		Input	hw_vio_1
ALUOp[2:0]	[H] 0		Input	hw_vio_1
ALUSrc1	[B] 0		Input	hw_vio_1
ALUSrc2	[B] 1		Input	hw_vio_1
data_mem_out[8:0]	[H] 000		Input	hw_vio_1
immediate[7:0]	[H] 00		Input	hw_vio_1
instruction[15:0]	[H] 3F00		Input	hw_vio_1
MemToReg	[B] 0		Input	hw_vio_1
MemWrite	[B] 0		Input	hw_vio_1
opcode[3:0]	[H] 3		Input	hw_vio_1
pc[7:0]	[H] 29		Input	hw_vio_1
rd_addr[1:0]	[H] 0		Input	hw_vio_1
read_data1[7:0]	[H] DE		Input	hw_vio_1
read_data2[7:0]	[H] DE		Input	hw_vio_1
RegDst	[B] 0		Input	hw_vio_1
regfile_write_data[8:0]	[H] 0DE		Input	hw_vio_1
RegWrite	[B] 1		Input	hw_vio_1
rs_addr[1:0]	[H] 3		Input	hw_vio_1
rt_addr[1:0]	[H] 3		Input	hw_vio_1

Figure 7: Hardware VIO testing results for the final computer design (lab 8).

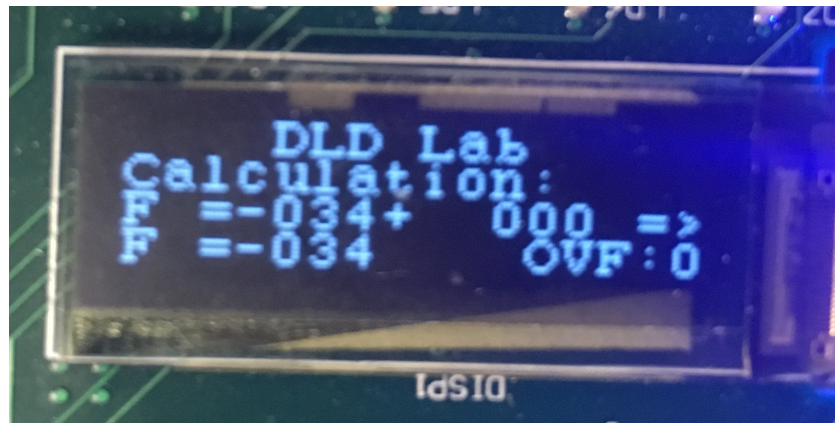


Figure 8: Hardware test with OLED screen showing software multiply implementation giving the result -34.

4 Conclusions

Appendices

A Instruction Decoder Verilog Module

```
module Decoder(  
    input  [15:0]instr,  
    output [3:0]opcode,  
    output [1:0]rs_addr,  
    output [1:0]rt_addr,  
    output [1:0]rd_addr,  
    output [7:0]immediate,  
    output reg RegDst,  
    output reg RegWrite,  
    output reg ALUSrc1,  
    output reg ALUSrc2,  
    output reg [2:0]ALUOp,  
    output reg MemWrite,  
    output reg MemToReg  
);  
    localparam LW    = 4'b0000;  
    localparam SW    = 4'b0001;  
    localparam ADD   = 4'b0010;  
    localparam ADDI  = 4'b0011;  
    localparam INV   = 4'b0100;  
    localparam AND   = 4'b0101;  
    localparam ANDI  = 4'b0110;  
    localparam OR    = 4'b0111;  
    localparam ORI   = 4'b1000;  
    localparam SRA   = 4'b1001;  
    localparam SLL   = 4'b1010;  
    localparam BEQ   = 4'b1011;  
    localparam BNE   = 4'b1100;  
    localparam CLR   = 4'b1101;  
    // unconditional  
    assign opcode = instr[15:12];  
    assign rs_addr = instr[11:10];  
    assign rt_addr = instr[9:8];  
    assign rd_addr = instr[7:6];
```

```
assign immediate = instr[7:0];
```

```
always @(*) begin
  case(opcode)
    LW : begin
      ALUOp    <= 3'b000;
      RegDst   <= 1'b0;
      RegWrite <= 1'b1;
      ALUSrc1  <= 1'b0;
      ALUSrc2  <= 1'b1;
      MemWrite <= 1'b0;
      MemToReg <= 1'b1;
    end
    SW : begin
      ALUOp <= 3'b000;
      RegDst <= 1'b0;
      RegWrite <= 1'b0;
      ALUSrc1 <= 1'b0;
      ALUSrc2 <= 1'b1;
      MemWrite <= 1'b1;
      MemToReg <= 1'b0;
    end
    ADD : begin
      ALUOp <= 3'b000;
      RegDst <= 1'b1;
      RegWrite <= 1'b1;
      ALUSrc1 <= 1'b0;
      ALUSrc2 <= 1'b0;
      MemWrite <= 1'b0;
      MemToReg <= 1'b0;
    end
    ADDI: begin
      ALUOp <= 3'b000;
      RegDst <= 1'b0;
      RegWrite <= 1'b1;
      ALUSrc1 <= 1'b0;
      ALUSrc2 <= 1'b1;
      MemWrite <= 1'b0;
      MemToReg <= 1'b0;
    end
  end
end
```

```

INV : begin
    ALUOp <= 3'b001;
    RegDst <= 1'b1;
    RegWrite <= 1'b1;
    //ALUSrc1 <= 1'b0;
    ALUSrc2 <= 1'b0;
    MemWrite <= 1'b0;
    MemToReg <= 1'b0;
end
AND : begin
    ALUOp <= 3'b010;
    RegDst <= 1'b1;
    RegWrite <= 1'b1;
    ALUSrc1 <= 1'b0;
    ALUSrc2 <= 1'b0;
    MemWrite <= 1'b0;
    MemToReg <= 1'b0;
end
ANDI: begin
    ALUOp <= 3'b010;
    RegDst <= 1'b0;
    RegWrite <= 1'b1;
    ALUSrc1 <= 1'b0;
    ALUSrc2 <= 1'b1;
    MemWrite <= 1'b0;
    MemToReg <= 1'b0;
end
OR  : begin
    ALUOp <= 3'b011;
    RegDst <= 1'b1;
    RegWrite <= 1'b1;
    ALUSrc1 <= 1'b0;
    ALUSrc2 <= 1'b0;
    MemWrite <= 1'b0;
    MemToReg <= 1'b0;
end
ORI : begin
    ALUOp <= 3'b011;
    RegDst <= 1'b0;
    RegWrite <= 1'b1;

```

```

    ALUSrc1 <= 1'b0;
    ALUSrc2 <= 1'b1;
    MemWrite <= 1'b0;
    MemToReg <= 1'b0;
end
SRA : begin
    ALUOp <= 3'b100;
    RegDst <= 1'b0;
    RegWrite <= 1'b1;
    ALUSrc1 <= 1'b0;
    //ALUSrc2 <= 1'b1;
    MemWrite <= 1'b0;
    MemToReg <= 1'b0;
end
SLL : begin
    ALUOp <= 3'b101;
    RegDst <= 1'b0;
    RegWrite <= 1'b1;
    ALUSrc1 <= 1'b0;
    //ALUSrc2 <= 1'b0;
    MemWrite <= 1'b0;
    MemToReg <= 1'b0;
end
BEQ : begin
    ALUOp <= 3'b110;
    RegDst <= 1'b0;
    RegWrite <= 1'b0;
    ALUSrc1 <= 1'b0;
    ALUSrc2 <= 1'b0;
    MemWrite <= 1'b0;
    MemToReg <= 1'b0;
end
BNE : begin
    ALUOp <= 3'b111;
    RegDst <= 1'b0;
    RegWrite <= 1'b0;
    ALUSrc1 <= 1'b0;
    ALUSrc2 <= 1'b0;
    MemWrite <= 1'b0;
    MemToReg <= 1'b0;

```

```

end
CLR : begin
    ALUOp <= 3'b010;
    RegDst <= 1'b1;
    RegWrite <= 1'b1;
    ALUSrc1 <= 1'b1;
    ALUSrc2 <= 1'b0;
    MemWrite <= 1'b0;
    MemToReg <= 1'b0;
end
endcase
end
endmodule

```

B Software Multiply Implementation

```

clr $0
clr $1
clr $2
clr $3
sw $3, 0x3($3)
addi $0, $0, 0xFE
addi $1, $1, 0x11
sw $0, 0x1($2)
sw $1, 0x2($2)
andi $2, $0, 0x80
beq $2, $3, CHK_SG
inv $0, $0
addi $0, $0, 0x1

CHK_SG:
andi $2, $1, 0x80
beq $2, $3, mult_loop
inv $1, $1
addi $1, $1, 0x1

mult_loop:
andi $2, $1, 0x1

```



```

beq $2, $3, bit_clear
clr $2
lw $3, 0x3($2)
add $3, $3, $0
sw $3, 0x3($2)
clr $3

```

```

bit_clear:
sll $0, $0, 0x1
sra $1, $1, 0x1
bne $1, $2, mult_loop
lw $0, 0x1($2)
lw $1, 0x2($2)

```

```

andi $0, $0, 0x80
andi $1, $1, 0x80
inv $2, $0
and $2, $1, $2
inv $3, $1
and $3, $0, $3
or $2, $2, $3
clr $0
lw $3, 0x3($0)
beq $2, $0, DONE
inv $3, $3
addi $3, $3, 0x1

```

```

DONE:
addi $3, $3, 0x0

```