EECS 112L Lab #2

Single Cycle MIPS Processor

Steven Tabilisma
16326339

10/6/22

Objective

The objective of this lab was to modify the given single cycle processor to accept 11 new instructions, as well as better familiarize ourselves with the structure of single cycle MIPS Processors. This lab is also a prerequisite to the following lab in which the modified processor will be used.

Procedure

The actual procedure of this lab was very simple and straightforward. By referencing the provided Table 4 in the lab document, I simply modified the control, ALUControl and ALU files in order to have the processor recognize the 11 new instructions. I have included my code for the mentioned files below:

control.v:
```
module control(
   input reset,
   input[5:0] opcode,
   output reg reg_dst, mem_to_reg,
   output reg [1:0] alu_op,
   output reg mem_read, mem_write, alu_src, reg_write, branch, jump
   );

always @(*)
begin
  if(reset == 1'b1)
  begin
    reg_dst = 1'b0;
    mem_to_reg = 1'b0;
    alu_op = 2'b10;
    mem_read = 1'b0;
    mem_write = 1'b0;
    alu_src = 1'b0;
    reg_write = 1'b0;
```

```
end
else
begin
  case(opcode)

  6'b000000:
  begin // add, and, or, sub, nor, slt, xor, mult, div
      reg_dst = 1'b1;
      mem_to_reg = 1'b0;
      alu_op = 2'b10;
      mem_read = 1'b0;
      mem_write = 1'b0;
      alu_src = 1'b0;
      reg_write = 1'b1;
      jump = 1'b0;
      branch = 1'b0;
  end
  6'b001000:
  begin // addi
      reg_dst = 1'b0;
      mem_to_reg = 1'b0;
      alu_op = 2'b00;
      mem_read = 1'b0;
      mem_write = 1'b0;
      alu_src = 1'b1;
      reg_write = 1'b1;
      jump = 1'b0;
      branch = 1'b0;
  end
  6'b100011:
  begin // lw
      reg_dst = 1'b0;
      mem_to_reg = 1'b1;
      alu_op = 2'b00;
      mem_read = 1'b1;
      mem_write = 1'b0;
      alu_src = 1'b1;
      reg_write = 1'b1;
      jump = 1'b0;
      branch = 1'b0;
   end
  6'b101011:
  begin // sw
      reg_dst = 1'b0;
```

```
    mem_to_reg = 1'b0;
    alu_op = 2'b00;
    mem_read = 1'b0;
    mem_write = 1'b1;
    alu_src = 1'b1;
    reg_write = 1'b0;
    jump = 1'b0;
    branch = 1'b0;
end

6'b001100:
begin // andi
    reg_dst = 1'b0;
    mem_to_reg = 1'b0;
    alu_op = 2'b11;
    mem_read = 1'b0;
    mem_write = 1'b0;
    alu_src = 1'b1;
    reg_write = 1'b1;
    jump = 1'b0;
    branch = 1'b0;
end

6'b000111:
begin // beq
    reg_dst = 1'b0;
    mem_to_reg = 1'b0;
    alu_op = 2'b01;
    mem_read = 1'b0;
    mem_write = 1'b0;
    alu_src = 1'b0;
    reg_write = 1'b0;
    jump = 1'b0;
    branch = 1'b1;
end

6'b110000:
begin // sll/srl/sra
    reg_dst = 1'b1;
    mem_to_reg = 1'b0;
    alu_op = 2'b10;
    mem_read = 1'b0;
    mem_write = 1'b0;
    alu_src = 1'b0;
```

```
            reg_write = 1'b1;
            jump = 1'b0;
            branch = 1'b0;
        end

        6'b000010:
        begin // jump
            reg_dst = 1'b0;
            mem_to_reg = 1'b0;
            alu_op = 2'b00;
            mem_read = 1'b0;
            mem_write = 1'b0;
            alu_src = 1'b0;
            reg_write = 1'b0;
            jump = 1'b1;
            branch = 1'b0;
        end


        endcase
      end
  end
  endmodule
```

ALUControl.v:

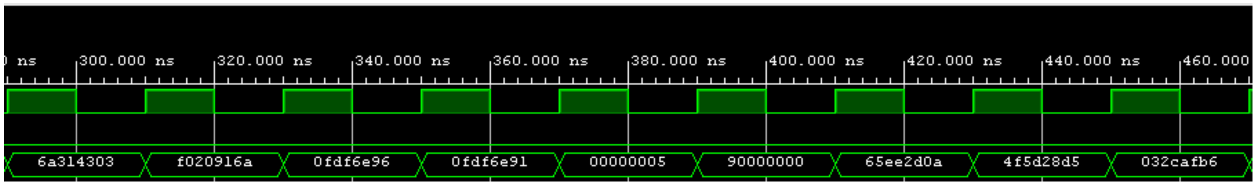
```
module ALUControl(
    input [1:0] ALUOp,
    input [5:0] Function,
    output reg[3:0] ALU_Control);

    wire [7:0] ALUControlIn;

    assign ALUControlIn = {ALUOp,Function};

    always @(ALUControlIn)

    casex (ALUControlIn)
        8'b10100000: ALU_Control=4'b0010; // add
        8'b00xxxxxx: ALU_Control=4'b0010; // addi/lw/sw
        8'b10100100: ALU_Control=4'b0000; // and
        8'b11xxxxxx: ALU_Control=4'b0000; // andi
        8'b01xxxxxx: ALU_Control=4'b0110; // beq
        8'b10100111: ALU_Control=4'b1100; // nor
        8'b10100101: ALU_Control=4'b0001; // or
        8'b10101010: ALU_Control=4'b0111; // slt
        8'b10000000: ALU_Control=4'b1000; // sll
        8'b10000010: ALU_Control=4'b1001; // srl
        8'b10000011: ALU_Control=4'b1010; // sra
        8'b10100010: ALU_Control=4'b0110; // sub
        8'b10100110: ALU_Control=4'b0100; // xor
        8'b10011000: ALU_Control=4'b0101; // mult
        8'b10011010: ALU_Control=4'b1011; // div
        8'bxxxxxxxx: ALU_Control=4'bxxxx; // jump
        default: ALU_Control=4'b0000;
    endcase
endmodule
```

ALU.v:

```
module ALU(
    input [31:0] a,
    input [31:0] b,
    input [3:0] alu_control,
    output zero,
    output reg [31:0] alu_result);

 always @(*)
 begin
      case(alu_control)
      4'b0010: alu_result = a + b; // add/addi/lw/sw // done
      4'b0000: alu_result = a & b; // and/andi // done
      4'b1100: alu_result = ~(a | b); // nor // done
      4'b0001: alu_result = a | b; // or // done
      4'b0111: alu_result = a + b; // slt
      4'b1000: alu_result = a << b; // sll // done
      4'b1001: alu_result = a >> b; // srl // done
      4'b1010: alu_result = a >>> b; // sra // done
      4'b0110: alu_result = a - b; // sub/beq // done
      4'b0100: alu_result = a ^ b; // xor // done
      4'b0101: alu_result = a * b; // mult // done
      4'b1011: alu_result = a / b; // div // done
      4'bxxxx: alu_result = a + b; // jump

      default: alu_result = a + b; // add

      endcase
 end
 assign zero = (alu_result==32'd0) ? 1'b1: 1'b0;

endmodule
```

Simulation Results

Given that we were provided a test bench, verifying that the processor was working was trivial. I will go over each newly implemented instruction.



Above are the results of the first set of tests for 9/11 of the new instructions. This does not cover BEQ and J.

ANDI

ANDI is the operation simply done between the value at the register provided and an immediate value. It is an R type instruction that only requires the signal from the control for writing to register.

NOR

NOR requires a destination register, and access to register writing in order to function, then it simply takes the given values, executes bitwise OR, and then applies NOT to the result and outputs it.

SLT

SLT takes the destination register and permission to write to registers, then it shifts the till to the first open one.

SLL

SLL takes a destination register and permission to write to registers, and then when recognized by ALUControl takes the first provided value and shifts it left logically by the value in the second register given. The result is left in the destination register.

SRL

SRL takes a destination register and permission to write to registers, and then when recognized by ALUControl takes the first provided value and shifts it Right logically by the value in the second register given. The result is left in the destination register.v

SRA

SRL takes a destination register and permission to write to registers, and then when recognized by ALUControl takes the first provided value and shifts it Right arithmetically by the value in the second register given. The result is left in the destination register.v

XOR

XOR is another R type instruction that takes the exact same signal requirements as ADD and AND do. Requires a destination register. It will execute a bitwise XOR between the two values in the given registers.

MULT

MULT is another R type instruction that takes the exact same signal requirements as ADD and AND do. Requires a destination register. It will execute an arithmetic multiplication between the two values in the given registers and then store it to the destination register.

DIV

DIV is another R type instruction that takes the exact same signal requirements as ADD and AND do. Requires a destination register. It will execute an arithmetic division between the two values in the given registers and then store it to the destination register.

For BEQ and J there is no actual result output to be shown so I have no image to include, but the instructions work as described below.

BEQ

BEQ is an I type instruction that will branch in the program to the address provided in the given register, given that the first two registers contain equal values. IF the evaluation is successful, the program counter is simply incremented to the address that was provided in the given register.

J

J is its own type of instruction only requiring an address to be provided. It will immediately jump the execution of the program by changing the program counter to the specified address from the register.