Lab 5: Control and Sign Extender

Justin Roessler and Jon Johnston

February 25, 2019

1 Executive Summary

The purpose of this lab was to make a Control module and a Sign Extender module for the Decode stage of our proccessor. The Sign Extender module reads in an instruction and extracts the opcode from it. It compares the given opcode to each case of the casex statement (CB, B, LDUR, and STUR). If the opcode does not match any of the cases, it is an R-type and will not be sign extended. When the opcode matches a particular case, it extracts the MSB of its address field and uses that bit to extend the address field to 64 bits.

The Control module works similar to the Sign Extender module, but inside the casex statement it sets the values on all the control wires depending on the opcode. The Control module will test for all 8 commands the proccessor will be able to execute, instead of the Sign Extender only testing 4, and set their cooresponding control wires. After comparing the Expected Results Table and each module's simulation, the lab was successful.

2 Test Report

To verify operation of these modules, this lab requires 2 test benches.

1. Control Test Bench
2. Sign_Extender Test Bench

Figure 1: Expected Results of the Lab 5.

	Instruction 1	Instruction 2	Instruction 3	Instruction 4	Instruction 5	Instruction 6	Instruction 7	Instruction 8	Instruction 9	Instruction 10
Instruction	LDUR X9, [X22, #64]	ADD X10, X19, X9	SUB X11, X20, X10	STUR X11, [X22, #96]	CBZ X11, -5	CBZ X9, 8	B 64	B -55	ORR X9, X10, X21	AND X9, X22, X10
Machine Instruction (hex)	F84402C9	8B09026A	CB0A028B	F80602CB	B4FFFF6B	B4000109	14000040	17FFFFC9	AA150149	8A0A02C9
opcode (binary)	11111000010	10001011000	11001011000	11111000000	10110100	10110100	000101	000101	10101010000	10001010000
sign_extended_output (hex)	0000000000000040	0000000000000000	0000000000000000	0000000000000060	FFFFFFFFFFFFFFFB	0000000000000008	0000000000000040	FFFFFFFFFFFFFFC9	0000000000000000	0000000000000000
reg2_loc	0	0	0	1	1	1	0	0	0	0
branch	0	0	0	0	1	1	0	0	0	0
mem_read	1	0	0	0	0	0	0	0	0	0
mem_to_reg	1	0	0	0	0	0	0	0	0	0
alu_op	00	10	10	00	01	01	01	01	10	10
mem_write	0	0	0	1	0	0	0	0	0	0
alu_src	1	0	0	1	0	0	0	0	0	0
reg_write	1	1	1	0	0	0	0	0	1	1
uncondbranch	0	0	0	0	0	0	1	1	0	0

Figure 2: Timing diagram for the Control test.

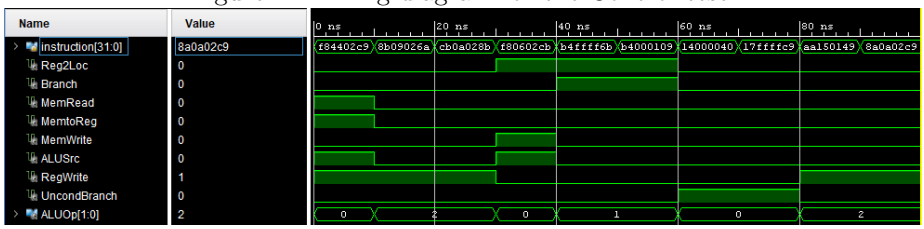
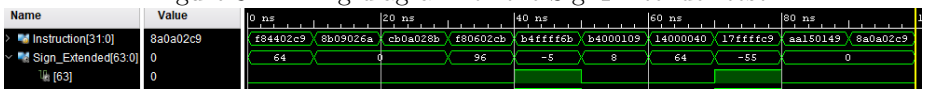


Figure 3: Timing diagram for the Sign_Extender test.



3 Code Appendix

Listing 1: Verilog code for testing the Control Module.

```
`include "definitions.vh"

module control_test;

    reg [`INSTR_LEN-1:0] instruction;
    wire Reg2Loc, Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite, UncondB
    wire [1:0] ALUOp;

    control cont_mod(.instruction(instruction), .Reg2Loc(Reg2Loc), .Branch(Branc
                     .MemWrite(MemWrite), .ALUSrc(ALUSrc), .RegWrite(RegWrite),
initial
    begin
    instruction = 32'hF84402C9; #10;
    instruction = 32'h8B09026A; #10;
    instruction = 32'hCB0A028B; #10;
    instruction = 32'hF80602CB; #10;
    instruction = 32'hB4FFFF6B; #10;
    instruction = 32'hB4000109; #10;
    instruction = 32'h14000040; #10;
    instruction = 32'h17FFFFC9; #10;
    instruction = 32'hAA150149; #10;
    instruction = 32'h8A0A02C9; #10;
    $finish;
    end

endmodule
```

Listing 2: Verilog code for implementing the Control Module.

```
`include "definitions.vh"

module control(
    input [`INSTR_LEN-1:0] instruction,
    output reg Reg2Loc, Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite, U
    output reg [1:0] ALUOp
    );

    reg [10:0] opcode;

    always @(*)
    begin
    opcode <= instruction[31:21];
```

```
        casex (opcode)
            `ADD: begin
                Reg2Loc <= 0;
                Branch <= 0;
                MemRead <= 0;
                MemtoReg <= 0;
                MemWrite <= 0;
                ALUSrc <= 0;
                RegWrite <= 1;
                UncondBranch <= 0;
                ALUOp <= 2'b10;
            end
            `SUB: begin
                Reg2Loc <= 0;
                Branch <= 0;
                MemRead <= 0;
                MemtoReg <= 0;
                MemWrite <= 0;
                ALUSrc <= 0;
                RegWrite <= 1;
                UncondBranch <= 0;
                ALUOp <= 2'b10;
            end
            `CBZ: begin
                Reg2Loc <= 1;
                Branch <= 1;
                MemRead <= 0;
                MemtoReg <= 0;
                MemWrite <= 0;
                ALUSrc <= 0;
                RegWrite <= 0;
                UncondBranch <= 0;
                ALUOp <= 2'b01;
            end
            `B: begin
                Reg2Loc <= 0;
                Branch <= 0;
                MemRead <= 0;
                MemtoReg <= 0;
                MemWrite <= 0;
                ALUSrc <= 0;
                RegWrite <= 0;
                UncondBranch <= 1;
                ALUOp <= 2'b00;
            end
            `LDUR: begin
```

```
                Reg2Loc <= 0;
                Branch <= 0;
                MemRead <= 1;
                MemtoReg <= 1;
                MemWrite <= 0;
                ALUSrc <= 1;
                RegWrite <= 1;
                UncondBranch <= 0;
                ALUOp <= 2'b00;
            end
            `STUR: begin
                Reg2Loc <= 1;
                Branch <= 0;
                MemRead <= 0;
                MemtoReg <= 0;
                MemWrite <= 1;
                ALUSrc <= 1;
                RegWrite <= 0;
                UncondBranch <= 0;
                ALUOp <= 2'b00;
            end
            `ORR: begin
                Reg2Loc <= 0;
                Branch <= 0;
                MemRead <= 0;
                MemtoReg <= 0;
                MemWrite <= 0;
                ALUSrc <= 0;
                RegWrite <= 1;
                UncondBranch <= 0;
                ALUOp <= 2'b10;
            end
            `AND: begin
                Reg2Loc <= 0;
                Branch <= 0;
                MemRead <= 0;
                MemtoReg <= 0;
                MemWrite <= 0;
                ALUSrc <= 0;
                RegWrite <= 1;
                UncondBranch <= 0;
                ALUOp <= 2'b10;
            end
            default: begin
                Reg2Loc <= 0;
                Branch <= 0;
```

```
                    MemRead <= 0;
                    MemtoReg <= 0;
                    MemWrite <= 0;
                    ALUSrc <= 0;
                    RegWrite <= 0;
                    UncondBranch <= 0;
                    ALUOp <= 2'b00;
                end
            endcase
    end
endmodule
```

Listing 3: Verilog code for testing the Sign_Extender Module.

```
`include "definitions.vh"

module sign_extender_test;

    reg [`INSTR_LEN -1:0] Instruction;
    wire [`WORD -1:0] Sign_Extended;

sign_extender Sign_Ext(.Instruction(Instruction), .Sign_Extended(Sign_Extended))

initial
begin
Instruction = 32'hF84402C9; #10;
Instruction = 32'h8B09026A; #10;
Instruction = 32'hCB0A028B; #10;
Instruction = 32'hF80602CB; #10;
Instruction = 32'hB4FFFF6B; #10;
Instruction = 32'hB4000109; #10;
Instruction = 32'h14000040; #10;
Instruction = 32'h17FFFFC9; #10;
Instruction = 32'hAA150149; #10;
Instruction = 32'h8A0A02C9; #10;
$finish;
end

endmodule
```

Listing 4: Verilog code for implementing the Sign_Extender Module.

```
`include "definitions.vh"

module sign_extender(
        input [`INSTR_LEN-1:0] Instruction,
        output reg [`WORD-1:0] Sign_Extended);

        reg [10:0] opcode;
        reg num;



      always @(*) begin
      opcode = Instruction[31:21];
        casex (opcode)
        `LDUR: begin
            num = Instruction[20];
            Sign_Extended = {{55{num}}, Instruction[20:12]};
        end
        `STUR: begin
            num = Instruction[20];
            Sign_Extended = {{55{num}}, Instruction[20:12]};
        end
        `B: begin
            num = Instruction[25];
            Sign_Extended = {{38{num}}, Instruction[25:0]};
        end
        `CBZ: begin
            num = Instruction[23];
            Sign_Extended = {{45{num}}, Instruction[23:5]};
        end
        default: begin
         Sign_Extended = 0;
        end
      endcase
      end

endmodule
```