Final Project: SingleCycleProcessor

ECE 4612

Leo Battalora

2021/12/15

Table of Contents

Objective(s) **5**

Tools/Equipment/Software **5**

Procedure **5**

SingleCycleProcessor 5

Control 8

ProgramCounter 10

InstructionMemory 10

Registers 12

ALUControl 13

ALU 13

DataMemory 15

Adder32bit 17

LeftShifter 19

SignExtender 20

Mux2to1 21

SignedAdder32bit 21

SignedSubtractor32bit 22

SignedMultiplier32bit 23

SignedDivider32bit 25

Test Plan **27**

SingleCycleProcessor 27

lw (load word) 29

sw (store word) 29

add (add word) 29

sub (subtract word) 30

mul muh mulu muhu (multiply integers (with result to GPR)) 31

div mod divu modu (divide integers (with result to GPR)) 32
Control 32
ProgramCounter 33
InstructionMemory 34
Registers 34
ALUControl 35
ALU 36
DataMemory 37
Adder32bit 37
LeftShifter 37
SignExtender 37
Mux2to1 38
SignedAdder32bit 38
SignedMultiplier32bit 38
SignedDivider32bit 39
Results/Observations **39**
SingleCycleProcessor 39
Control 41
ProgramCounter 41
InstructionMemory 41
Registers 42
ALUControl 42
ALU 43
DataMemory 43
Adder32bit 44
LeftShifter 46
SignExtender 47
Mux2to1 48

SignedAdder32bit 48

SignedMultiplier32bit 49

SignedDivider32bit 49

Conclusion **50**

Objective(s)

The objective of this project is to design a MIPS32-based single-cycled processor using Vivado. In the SingleCycleProcessor subsection of Results/Observations a test program is used to validate the top-level design.

The processor supports a subset of the MIPS32 instruction set (revision 6.06). A copy of the instruction set manual can be found here: https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00086-2B-MIPS32BIS-AFP-6.06.pdf

The subset of instructions supported include the following: `lw, sw, beq, j, slt, and, or, add, sub, mul, muh, div, mod`.

Tools/Equipment/Software

Module development and testing was performed using Vivado 2020.2.

Procedure

SingleCycleProcessor

The SingleCycleProcessor module is implemented using all the following modules in the Procedure section. The submodules are wired together according to the diagram in Figure 1 below.

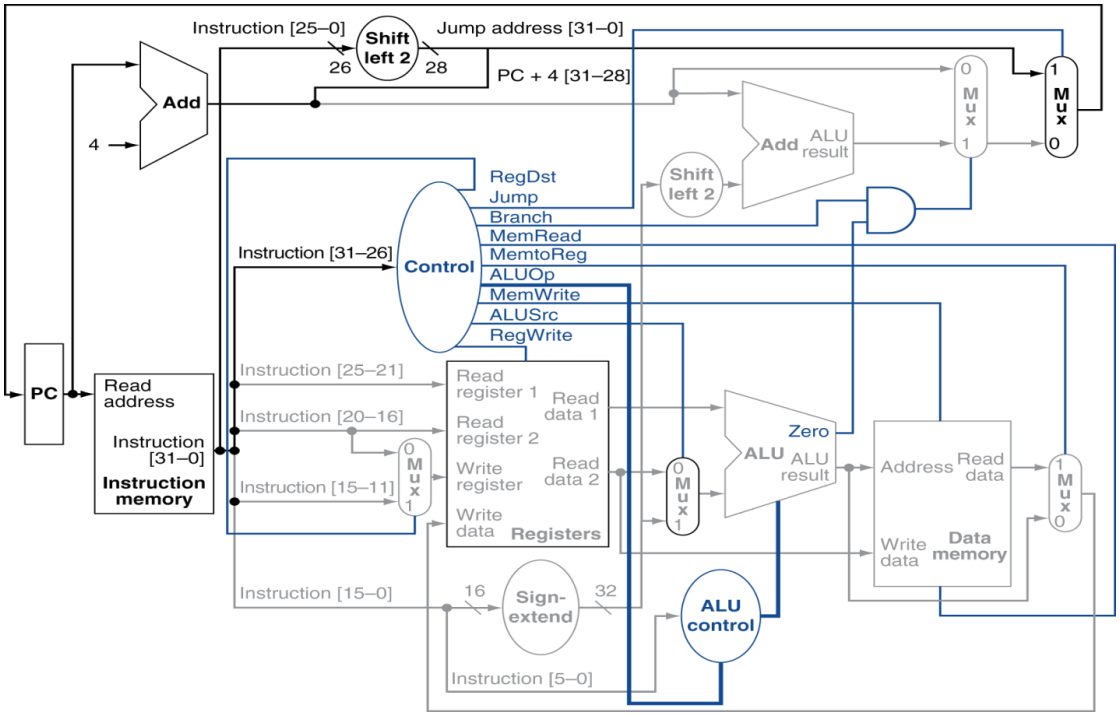


Figure 1. Block diagram of a simplified single-cycled processor

The Verilog definition of the SingleCycleProcessor is shown in Figure 2 below. Notice that it takes in a single input of a clock signal (`clk`).

```
module SingleCycleProcessor(
    input clk
    );

    wire [31:0] pc; // current address
    wire [31:0] pc_plus_1; // current address +4
    wire [31:0] instr_index_shifted; // low 28 bits of target address when jumping
    wire [31:0] pc_jump_addr; // new address calculated from jump
    wire [31:0] pc_branch_calc; // new address calculated if branch occurs
    wire [31:0] pc_branch_res; // new address selected from branch or normal +4
    wire [31:0] immediate; // sign-extended branch offset
    wire [31:0] branch_offset_shifted; // sign-extended branch offset shifted
    wire [31:0] pc_next; // input of ProgramCounter
    wire [31:0] instruction; // output of InstructionMemory
    wire [31:0] rs_data; // source register 1 (alu_input_1)
    wire [31:0] rt_data; // source register 2
    wire [31:0] alu_input_2; // either rt_data or immediate
    wire [31:0] rd_data; // destination register
    wire [31:0] alu_result; // output of ALU
    wire [31:0] mem_read_data; // output of DataMemory
    wire [4:0] write_reg; // destination register to write to
    wire [3:0] alu_ctl; // output of ALUControl to dictate which operation ALU performs
    wire [1:0] alu_op; // output of Control to dictate ALUControl operation selectron
    wire reg_dst_sel; // output of Control
    wire jump_src_sel; // output of Control
    wire branch_src_sel; // output of Control
    wire alu_src_sel; // output of Control
    wire mem_to_reg_sel; // output of Control
    wire reg_write; // output of Control
    wire mem_read; // output of Control
    wire mem_write; // output of Control
    wire branch_src_sel_res; // output from AND of branch_src_sel and alu_zero
    wire alu_zero; // output from ALU indicating a 0 result


    // Control
    Control             M00(.opcode(instruction[31:26]),.alu_op(alu_op),.reg_dst_sel(reg_dst_sel),
                            .jump_src_sel(jump_src_sel),.branch_src_sel(branch_src_sel),
                            .alu_src_sel(alu_src_sel),.mem_to_reg_sel(mem_to_reg_sel),
                            .reg_write(reg_write),.mem_read(mem_read),.mem_write(mem_write));

    // Program Counter and Instruction fetching
    ProgramCounter      M01(.pc_next(pc_next),.pc_current(pc),.clk(clk));
    InstructionMemory   M02(.address(pc),.instruction(instruction));
    Adder32bit          M03(.a(pc),.b(32'd1),.c_in(1'b0),.sum(pc_plus_1));
    LeftShifter         M04(.in(instruction[25:0]),.out(instr_index_shifted));
    assign pc_jump_addr = {pc_plus_1[31:28],instr_index_shifted[27:0]};
    SignExtender        M05(.in(instruction[15:0]),.out(immediate));
    LeftShifter         M06(.in(immediate),.out(branch_offset_shifted));
    Adder32bit          M07(.a(pc_plus_1),.b(branch_offset_shifted),.c_in(1'b0),.sum(pc_branch_calc));
    assign branch_src_sel_res = branch_src_sel & alu_zero;
    Mux2to1             M08(.a(pc_plus_1),.b(pc_branch_calc),.sel(branch_src_sel_res),.out(pc_branch_res));
    Mux2to1             M09(.a(pc_branch_res),.b(pc_jump_addr),.sel(jump_src_sel),.out(pc_next));

    // Registers
    Mux2to1 #(.WIDTH(5)) M10(.a(instruction[20:16]),.b(instruction[15:11]),.sel(reg_dst_sel),.out(write_reg));
    Registers           M11(.read_reg_1(instruction[25:21]),.read_reg_2(instruction[20:16]),
                            .read_data_1(rs_data),.read_data_2(rt_data),
                            .write_reg(write_reg),.write_data(rd_data),.reg_write(reg_write),
                            .clk(clk));

    // ALU and ALU control
    Mux2to1             M12(.a(rt_data),.b(immediate),.sel(alu_src_sel),.out(alu_input_2));
    ALUControl          M13(.alu_op(alu_op),.shamt(instruction[10:6]),.func_code(instruction[5:0]),.alu_ctl(alu_ctl));
    ALU                 M14(.a(rs_data),.b(alu_input_2),.ctl(alu_ctl),.result(alu_result),.zero(alu_zero));

    // Memory
    DataMemory          M15(.address(alu_result),.mem_write(mem_write),.mem_read(mem_read),
                            .write_data(rt_data),.read_data(mem_read_data),.clk(clk));
    Mux2to1             M16(.a(alu_result),.b(mem_read_data),.sel(mem_to_reg_sel),.out(rd_data));
endmodule
```

Figure 2. Verilog definition of SingleCycleProcessor module

The synthesized schematic for the SingleCycleProcessor module shown above is shown in Figure 3 below.

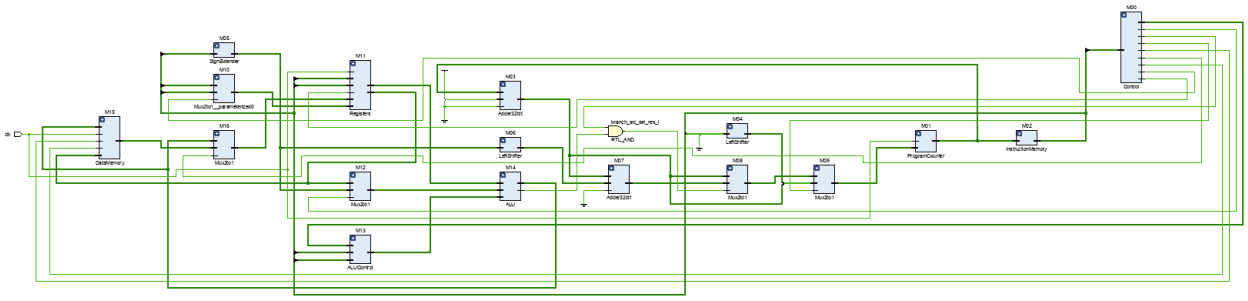


Figure 3. Schematic representation of SingleCycleProcessor module

Control

The Control module is used to change the control signals in the processor based on the opcode of the current instruction. A case statement is used to evaluate the opcode type and specify the control signals for each instruction type (`lw`, `sw`, `beq`, `j`, `slt`, `and`, `or`, `add`, `sub`, `mul`, `muh`, `div`, `mod`).

The Verilog definition of the Control module is shown in Figure 4 below.

```verilog
module Control(
    input [5:0] opcode, // instruction[31:26]
    output reg [1:0] alu_op,
    output reg reg_dst_sel,
    output reg jump_src_sel,
    output reg branch_src_sel,
    output reg alu_src_sel,
    output reg mem_to_reg_sel, // determines if data written to register comes from memory or ALU
    output reg reg_write,
    output reg mem_read,
    output reg mem_write
    );

    always @(opcode) begin
        case (opcode) // instruction[31:26]
            6'b100011: begin // lw (load word) I-format
                alu_op <= 2'b00;
                reg_dst_sel <= 1'b0;
                jump_src_sel <= 1'b0;
                branch_src_sel <= 1'b0;
                alu_src_sel <= 1'b1;
                mem_to_reg_sel <= 1'b1;
                reg_write <= 1'b1;
                mem_read <= 1'b1;
                mem_write <= 1'b0;
            end
            6'b101011: begin // sw (store word) I-format
                alu_op <= 2'b00;
                reg_dst_sel <= 1'bx;
                jump_src_sel <= 1'b0;
                branch_src_sel <= 1'b0;
                alu_src_sel <= 1'b1;
                mem_to_reg_sel <= 1'bx;
                reg_write <= 1'b0;
                mem_read <= 1'b0;
                mem_write <= 1'b1;
            end

            6'b000100: begin // beq (branch on equal) I-format
                alu_op <= 2'b01;
                reg_dst_sel <= 1'bx;
                jump_src_sel <= 1'b0;
                branch_src_sel <= 1'b1;
                alu_src_sel <= 1'b0;
                mem_to_reg_sel <= 1'bx;
                reg_write <= 1'b0;
                mem_read <= 1'b0;
                mem_write <= 1'b0;
            end
            6'b000010: begin // j (jump) J-format
                alu_op <= 2'bxx; // doesn't use ALU
                reg_dst_sel <= 1'bx;
                jump_src_sel <= 1'b1;
                branch_src_sel <= 1'b0;
                alu_src_sel <= 1'b0;
                mem_to_reg_sel <= 1'bx;
                reg_write <= 1'b0;
                mem_read <= 1'b0;
                mem_write <= 1'b0;
            end
            6'b000000: begin // SPECIAL (slt,and,or,add,sub,mul,muh,div,mod) R-format
                alu_op <= 2'b10;
                reg_dst_sel <= 1'b1;
                jump_src_sel <= 0;
                branch_src_sel <= 0;
                alu_src_sel <= 0;
                mem_to_reg_sel <= 1'b0;
                reg_write <= 1'b1;
                mem_read <= 1'b0;
                mem_write <= 1'b0;
            end
        endcase
    end
endmodule
```

Figure 4. Verilog definition of Control module

ProgramCounter

The ProgramCounter module takes in the next program counter value (i.e. next instruction address) and outputs it on each clock cycle.

The Verilog definition of the ProgramCounter module is shown in Figure 5 below.

```
14  module ProgramCounter(
15      input [31:0] pc_next,
16      output reg [31:0] pc_current,
17      input clk
18      );
19
20      // start the program at instruction address 0
21      initial begin
22          pc_current <= 32'h00000000;
23      end
24
25      // output next instruction address on each clock cycle
26      always @(posedge clk) begin
27          pc_current <= pc_next;
28      end
29  endmodule
```

Figure 5. Verilog definition of ProgramCounter module

InstructionMemory

The Instruction Memory module is implemented using a memory space which can be customized using the WIDTH and DEPTH parameters. It defaults to a 32-bit wide memory block with 1024 addresses. It uses a continuous assignment to output the instruction corresponding to the requested instruction memory address.

An initial block is used to initialize the instruction memory from a file. A 'program.txt' file was placed in the root of the project and imported as a simulation source. The contents of the file are read as whitespace separated binary values. Figure 6 below shows a set of 14 32-bit binary codes that would be initialized to the instruction memory.

The Verilog definition of the InstructionMemory module is shown in Figure 7 below.

program.txt

C:/Users/rktli/Documents/GitHub/TU-ECE-4612-Advanced-Processor-Systems/SingleCycleProcessor/program.txt

```
1  10001100000010000000000000000000
2  10001100000010010000000000000001
3  10001100000010100000000000000010
4  10001100000010110000000000000011
5  10001100000011000000000000000100
6  10001100000011010000000000000101
7  00000001000010100100000000100010
8  00000001011010010101100000100010
9  00000001011011000101100010011000
10 00000001000010110100000000100000
11 00000001000011010111000010011010
12 00000001000011010111100011011010
13 10101100000011100000000000000110
14 10101100000011110000000000000111
```

Figure 6. File to hold program instructions

```
module InstructionMemory #(parameter WIDTH = 32, DEPTH = 1024) (
    input [31:0] address,
    output [31:0] instruction
    );

    // memory with 1024 locations [0:1023] (depth) each of 32 bits [31:0] (width).
    reg [WIDTH-1:0] memory [0:DEPTH-1];

    // initialize memory to all 0's, then read program.txt
    initial begin : initialize_memory
        integer i;
        for (i = 0; i < DEPTH; i = i+1) begin
            memory[i] = 32'b0;
        end
        $readmemb("program.txt", memory);
    end

    // output the instruction corresponding to the requested address
    assign instruction = memory[address];
endmodule
```

Figure 7. Verilog definition of InstructionMemory module

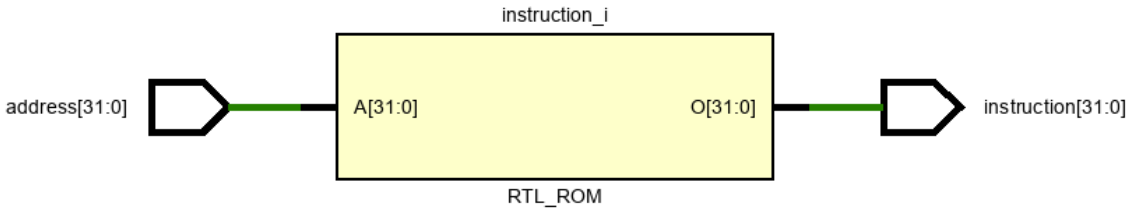


Figure 8. Schematic representation of InstructionMemory module

Registers

The Registers module contains a bank of 32 32-bit registers. It can read from two registers at once, and write to one register at a time. On module initialization, all registers are initialized to 0.

The Verilog definition of the Register module is shown in Figure 9 below.

```
module Registers(
    input [4:0] read_reg_1,
    input [4:0] read_reg_2,
    input [4:0] write_reg,
    input [31:0] write_data,
    input clk,
    input reg_write,
    output [31:0] read_data_1,
    output [31:0] read_data_2
    );

    // register space with 32 locations [0:31] (depth) each of 32 bits [31:0] (width)
    reg [31:0] registers [0:31];

    // initialize registers to all 0's
    initial begin : initialize_registers
    integer i;
        for (i = 0; i < 32; i = i+1) begin
            registers[i] = 32'b0;
        end
    end

    // on each clock cycle, write write_data to write_reg if reg_write is high
    always @(posedge clk) begin
        if (reg_write == 1) begin
            registers[write_reg] = write_data;
        end
    end

    assign read_data_1 = registers[read_reg_1];
    assign read_data_2 = registers[read_reg_2];
endmodule
```

Figure 9. Verilog definition of Registers module

ALUControl

The ALUControl module takes in the alu_op, shamt, and func_code signals to determine the code of the ALU function which should be executed by the ALU module. It is implemented using nested case statements.

The Verilog definition of the ALUControl module is shown in Figure 10 below.

```
module ALUControl(
    input [1:0] alu_op,
    input [4:0] shamt,      // instruction[10:6]
    input [5:0] func_code,  // instruction[5:0]
    output reg [3:0] alu_ctl
    );

    always @(alu_op, shamt, func_code) begin
        case (alu_op)
            2'b00: alu_ctl <= 4'b0010; // lw,sw (add)
            2'b01: alu_ctl <= 4'b0011; // beq (sub)
            2'b10: begin // R-type
                case (func_code) // instruction[5:0]
                    6'b101010: alu_ctl <= 4'b1000; // slt
                    6'b100100: alu_ctl <= 4'b0000; // and
                    6'b100101: alu_ctl <= 4'b0001; // or
                    6'b100000: alu_ctl <= 4'b0010; // add
                    6'b100010: alu_ctl <= 4'b0011; // sub
                    6'b011000: begin
                        case(shamt)
                            5'b00010: alu_ctl <= 4'b0100; // mul shamt = 00010
                            5'b00011: alu_ctl <= 4'b0101; // muh shamt = 00011
                        endcase
                    end
                    6'b011010: begin
                        case(shamt)
                            5'b00010: alu_ctl <= 4'b0110; // div shamt = 00010
                            5'b00011: alu_ctl <= 4'b0111; // mod shamt = 00011
                        endcase
                    end
                endcase
            end
        endcase
    end
endmodule
```

Figure 10. Verilog definition of ALUControl module

ALU

The ALU module performs all arithmetic operations required by the processor. Each time its inputs change (ctrl, a, b), it recalculates all possible results. The control signal is then used to

select which of the results should be output by the ALU. A special zero signal is also output whenever the ALU result is 0.

The Verilog definition of the ALU module is shown in Figure 11 below. A synthesized schematic representation is shown in Figure 12 below.

The adder, subtractor, multiplier, and divider modules are discussed later in this section.

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

    wire [31:0] res_and;
    wire [31:0] res_or;
    wire [31:0] res_slt;
    wire [31:0] res_add; // sum
    wire [31:0] res_sub; // difference
    wire [31:0] res_mul; // product low
    wire [31:0] res_muh; // product high
    wire [31:0] res_div; // quotient
    wire [31:0] res_mod; // remainder (modulo)

    // zero is true if result is 0
    assign zero = (result == 0);

    assign res_and = a & b;
    assign res_or = a | b;
    assign res_slt = (a < b) ? 1 : 0;
    SignedAdder32bit        M1(.a(a),.b(b),.c_in(1'b0),.sum(res_add));
    SignedSubtractor32bit   M2(.a(a),.b(b),.difference(res_sub));
    SignedMultiplier32bit   M3(.a(a),.b(b),.out_hi(res_muh),.out_lo(res_mul));
    SignedDivider32bit      M4(.a(a),.b(b),.out_q(res_div),.out_r(res_mod));

    always @(ctl, res_and, res_or, res_slt, res_add, res_sub, res_mul, res_muh, res_div, res_mod) begin
        case (ctl)
            4'b0000: result <= res_and;
            4'b0001: result <= res_or;
            4'b0010: result <= res_add;
            4'b0011: result <= res_sub;
            4'b0100: result <= res_mul;
            4'b0101: result <= res_muh;
            4'b0110: result <= res_div;
            4'b0111: result <= res_mod;
            4'b1000: result <= res_slt;
        endcase
    end
endmodule
```

Figure 11. Verilog definition of ALU module

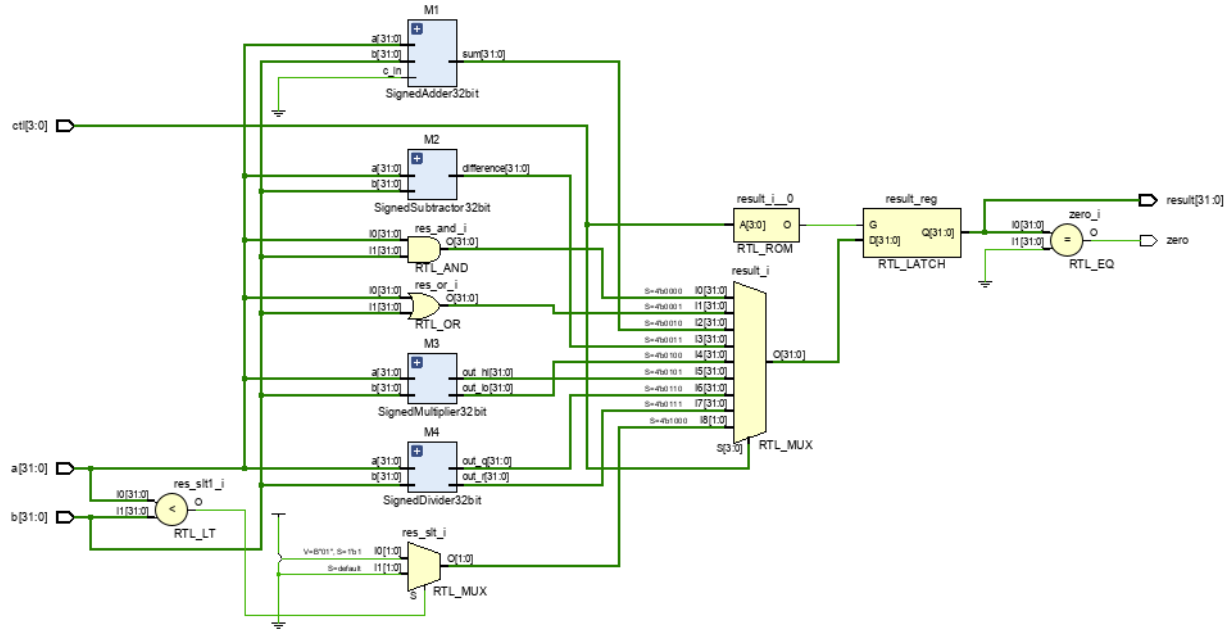


Figure 12. Schematic representation of ALU module

DataMemory

The Data Memory module is implemented using a memory space which can be customized using the WIDTH and DEPTH parameters. It defaults to a 32-bit wide memory block with 1024 addresses. On each clock cycle, if the memory write signal is high, then the new value will be written to the specified address. Whenever the address signal is changed, the specified address will be read.

An initial block is used to initialize the data memory from a file. A 'data.txt' file was placed in the root of the project and imported as a simulation source. The contents of the file are read as whitespace separated binary values. Figure 13 below shows a set of 6 32-bit binary values that would be initialized to the instruction memory.

The Verilog module of the Signed Adder can be seen in Figures 14 below.

data.txt

C:/Users/rktli/Documents/GitHub/TU-ECE-4612-Advanced-Processor-Systems/SingleCycleProcessor/data.txt

```
1 00000000000000000000001111101000
2 00000000000000000000000011001000
3 00000000000000000000000100101100
4 00000000000000000000000110010000
5 00000000000000000000000111110100
6 00000000000000000000000000000011
```

Figure 13. File to hold initial data memory

```
module DataMemory #(parameter WIDTH = 32, DEPTH = 1024) (
    input [WIDTH-1:0] address,
    input [WIDTH-1:0] write_data,
    output reg [WIDTH-1:0] read_data,
    input mem_write,
    input mem_read,
    input clk
    );

    // memory with 1024 locations [0:1023] (depth) each of 32 bits [31:0] (width).
    reg [WIDTH-1:0] memory [0:DEPTH-1];

    // initialize memory to all 0's, then read data.txt
    initial begin : initialize_memory
        integer i;
        for (i = 0; i < DEPTH; i = i+1) begin
            memory[i] = 32'b0;
        end
        $readmemb("data.txt", memory);
    end

    // on each clock cycle, write write_data to address if mem_write is high
    always @(posedge clk) begin
        if (mem_write == 1) begin
            memory[address] = write_data;
        end
    end

    // on each clock cycle, read data from address to read_data
    always @(address) begin
        read_data <= memory[address];
    end
endmodule
```

Figure 14. Verilog definition of DataMemory module

Adder32bit

A hierarchical design was used to build up to a 32-bit adder. First a half adder was designed and tested. Next a full adder was created using two half adders. Next a 4-bit adder was created using four full adders. Next a 16-bit adder was created using four 4-bit adders. And finally the 32-bit adder was created using two 16-bit adders. This hierarchy can be seen in Figure 15 below.

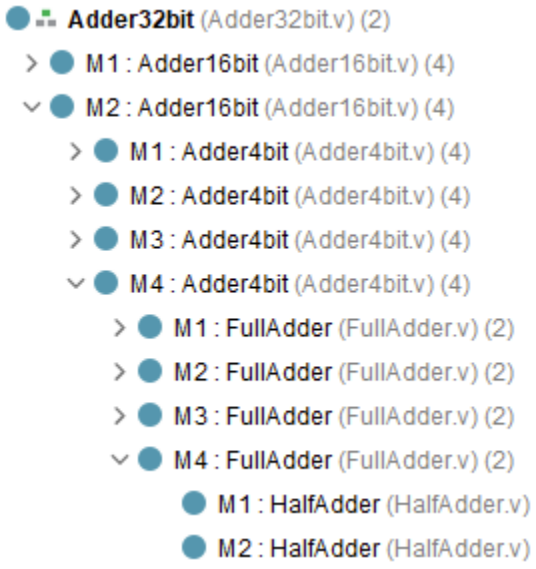
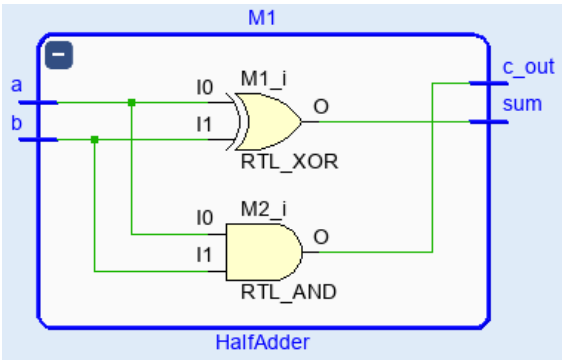


Figure 15. 32-bit adder hierarchical design

The half adder takes in two bits, a and b, as inputs, then outputs a 1-bit sum with a carry bit. The design for a half adder can be seen in Figures 16 & 17 below.

```
module HalfAdder(
    input a,
    input b,
    output sum,
    output c_out
    );

    xor M1(sum, a, b);
    and M2(c_out, a, b);
endmodule
```



Figures 16 & 17. Verilog and schematic representations of a half adder

The full adder takes in three bits, a, b, and a carry, as inputs, then outputs a 1-bit sum with a carry bit. The design for a full adder can be seen in Figures 18 & 19 below.

```
module FullAdder(
    input a,
    input b,
    input c_in,
    output sum,
    output c_out
    );

    wire    w1,w2,w3;

    HalfAdder   M1(.sum(w1),.c_out(w2),.a(a),.b(b));
    HalfAdder   M2(.sum(sum),.c_out(w3),.a(w1),.b(c_in));
    or          M3(c_out,w2,w3);
endmodule
```

Figure 18. Verilog representation of a full adder

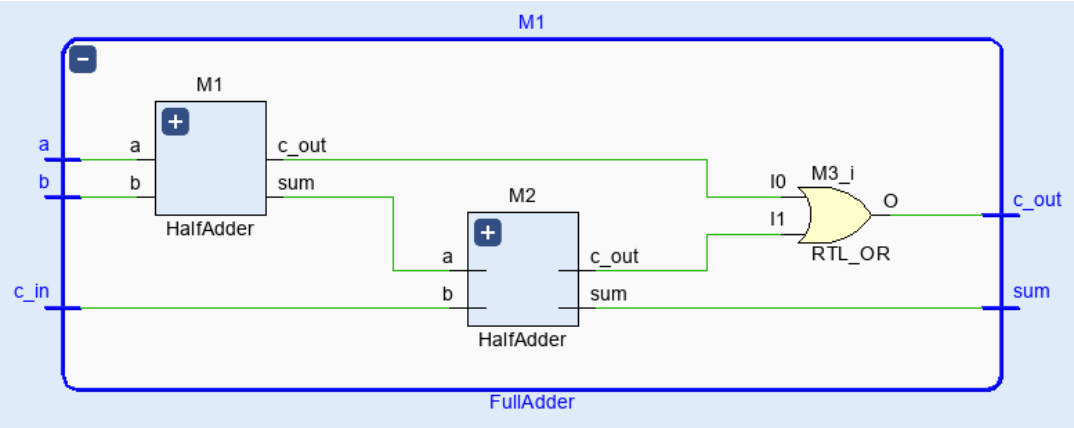


Figure 19. schematic representation of a full adder

The 4-bit adder takes in a carry bit and two 4-bit inputs a and b. It outputs a 4-bit sum with a carry bit. The design schematic for a 4-bit adder can be seen in Figure 20 below.

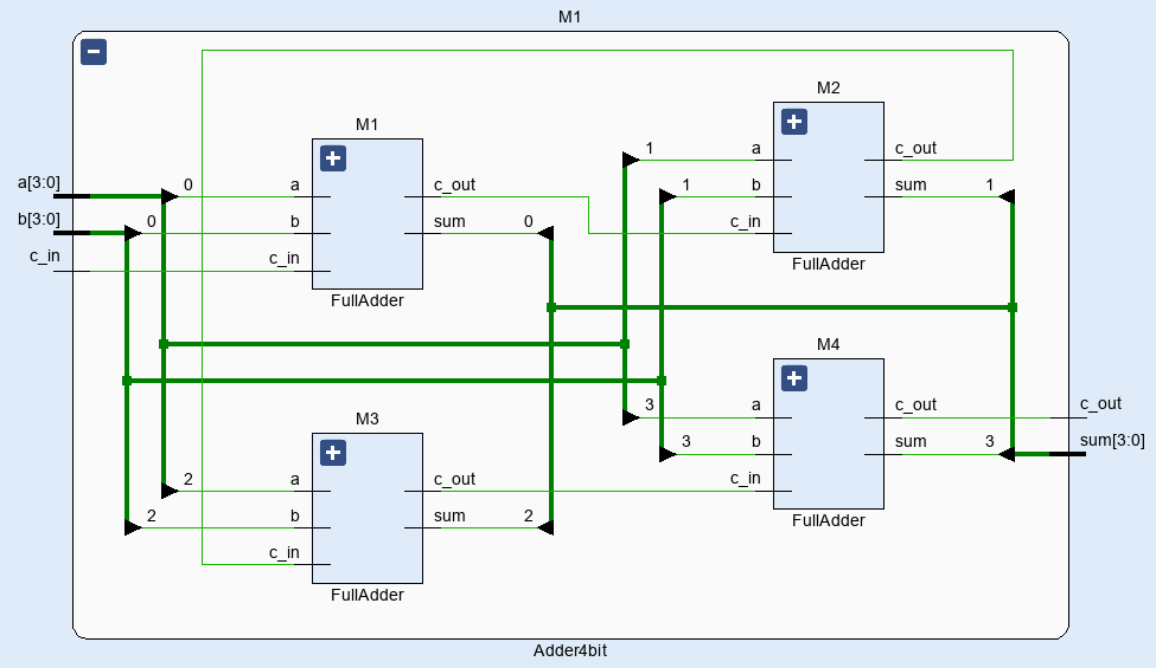


Figure 20. schematic representation of a 4-bit adder

The pattern continues with a 16-bit adder using four 4-bit adders and the 32-bit adder using two 16-bit adders. The design schematic for a 32-bit adder can be seen in Figure 21 below. It takes in a carry bit and two 32-bit inputs a and b. It outputs a 32-bit sum with a carry bit.

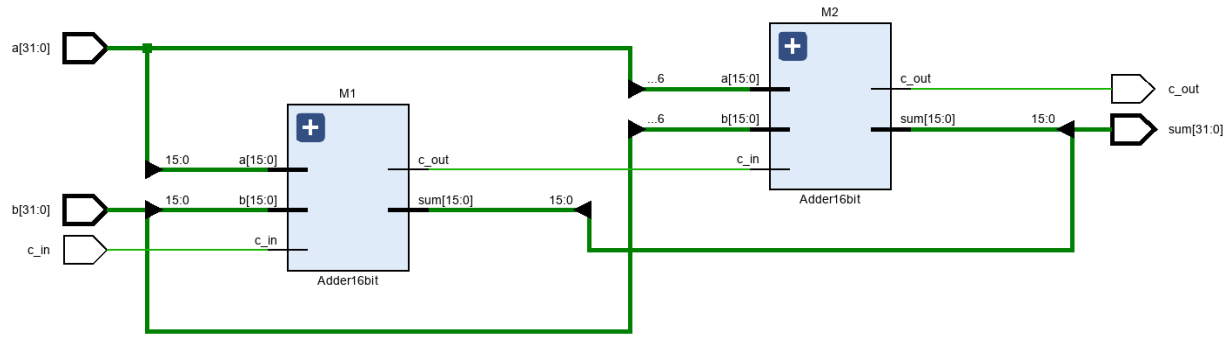


Figure 21. schematic representation of a 32-bit adder

As each module in the hierarchy was developed a corresponding testbench was created and ran to confirm the expected behavior.

LeftShifter

The Left Shifter is implemented using a continuous assignment which changes based on the 32-bit input signal. A left shift operator is used to shift the input signal by two. In the comment

above the assign statement, there is also an implementation using a concatenator and two zero bits. The Verilog module and schematic representations of the Left Shifter can be seen in Figures 22 and 23 below, respectively.

```
module LeftShifter(
    input [31:0] in,
    output [31:0] out
    );

    //assign out = {in[29:0], 2'b00};
    assign out = in << 2;
endmodule
```

Figure 22. Verilog representation of a LeftShifter

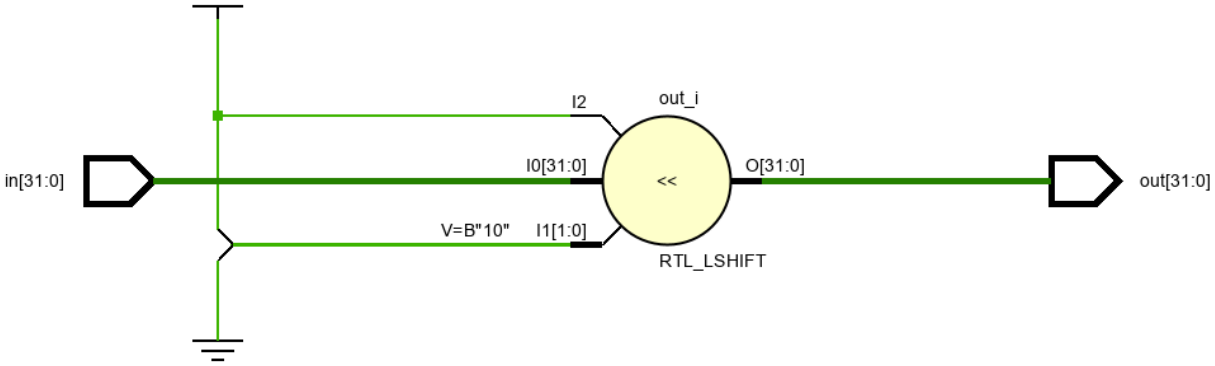


Figure 23. schematic representation of a LeftShifter

SignExtender

The Sign Extender is implemented using a continuous assignment which changes based on the 16-bit input signal. Concatenation and replication operators are used to assign 16 bits of the MSB of the input signal concatenated to the 16-bit input signal. The Verilog module and schematic representations of the Sign Extender can be seen in Figures 24 and 25 below, respectively.

```
module SignExtender(
    input [15:0] in,
    output [31:0] out
    );

    assign out = {{16{in[15]}}, in};
endmodule
```

Figure 24. Verilog representation of a SignExtender

Figure 25. schematic representation of a SignExtender

Mux2to1

The 2-to-1 Multiplexer is implemented using a continuous assignment which changes based on the 1-bit select input signal. A ternary operator is used to assign signal a when sel is 0 and b when sel is 1. The Verilog module and schematic representations of the 2-to-1 Multiplexer can be seen in Figures 26 and 27 below, respectively.

```
module Mux2to1(
    input [31:0] a,
    input [31:0] b,
    input sel,
    output [31:0] out
    );

    // when sel == 0, out = a
    // when sel == 1, out = b
    assign out = sel ? b : a;
endmodule
```

Figure 26. Verilog representation of a 2-to-1 multiplexer

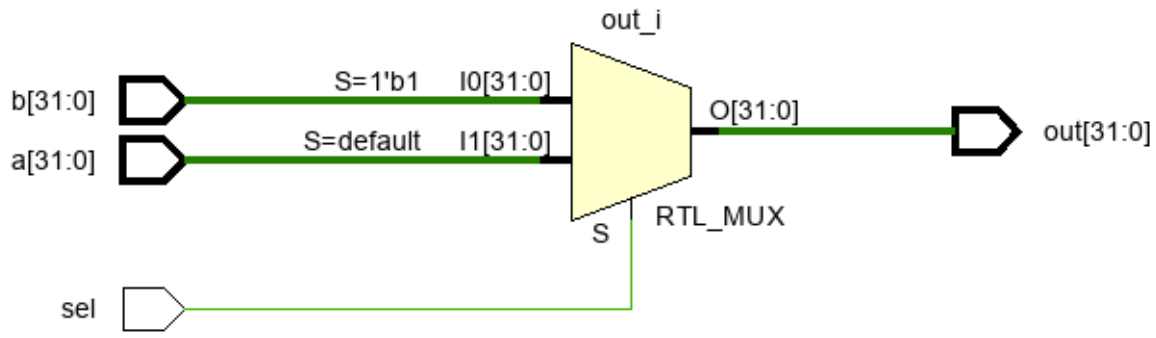


Figure 27. schematic representation of a 2-to-1 multiplexer

SignedAdder32bit

The previously built 32-bit Adder was used to generate the sum and carry out signals. An additional assignment and output signal were added to indicate whether an overflow had occurred as a result of the addition of the 2's complement input integers. The Verilog module

and schematic representations of the Signed Adder can be seen in Figures 28 and 29 below, respectively.

```
module SignedAdder32bit(
    input [31:0] a,
    input [31:0] b,
    input c_in,
    output [31:0] sum,
    output c_out,
    output overflow
    );

    Adder32bit M1(.sum(sum),.c_out(c_out),.a(a),.b(b),.c_in(c_in));

    // rules for detecting overflow:
    //   - If the sum of two positive numbers yields a negative result, the sum has overflowed.
    //   - If the sum of two negative numbers yields a positive result, the sum has overflowed.
    //   - Otherwise, the sum has not overflowed.
    assign overflow = ((a[31] == 0 && b[31] == 0 && sum[31] == 1) ||
                       (a[31] == 1 && b[31] == 1 && sum[31] == 0)) ? 1 : 0;
endmodule
```

Figure 28. Verilog representation of a Signed Adder

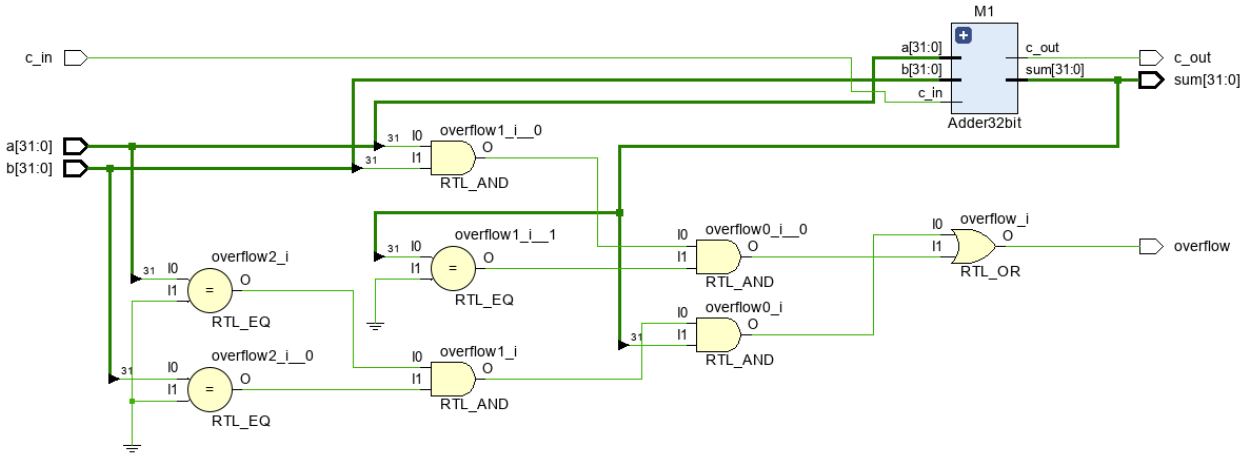


Figure 29. schematic representation of a Signed Shifter

SignedSubtractor32bit

The SignedSubtractor was implemented by taking a SignedAdder module and negating the subtrahend.

The Verilog definition of the SignedSubtractor32bit module is shown in Figure 30 below.

```
module SignedSubtractor32bit(
    input [31:0] a,
    input [31:0] b,
    output [31:0] difference,
    output c_out,
    output overflow
    );

    wire [31:0] negated_b;

    assign negated_b = -b;

    SignedAdder32bit M1(.sum(difference),.c_out(c_out),.overflow(overflow),.a(a),.b(negated_b),.c_in(1'b0));
endmodule
```

Figure 30. Verilog definition of ALU module

SignedMultiplier32bit

The multiplier used in the ALU was designed based on a sequential multiplication algorithm. The hardware design is like what is shown in Figure 31 below.

A flowchart for the sequential multiplication algorithm is shown in Figure 32 below. On each cycle it adds the multiplicand to the product if the LSB of the multiplier is 1. Then it shifts the multiplicand left by 1 bit and shifts the multiplier right by 1 bit. This cycle repeats 32 times.

The module was first implemented as an unsigned 32-bit multiplier. To make it signed, additional logic was embedded before and after to strip the inputs of their sign and reapply it correctly after the multiplication was complete. The Verilog definition of the SignedMultiplier32bit module is shown in Figure 33 below.

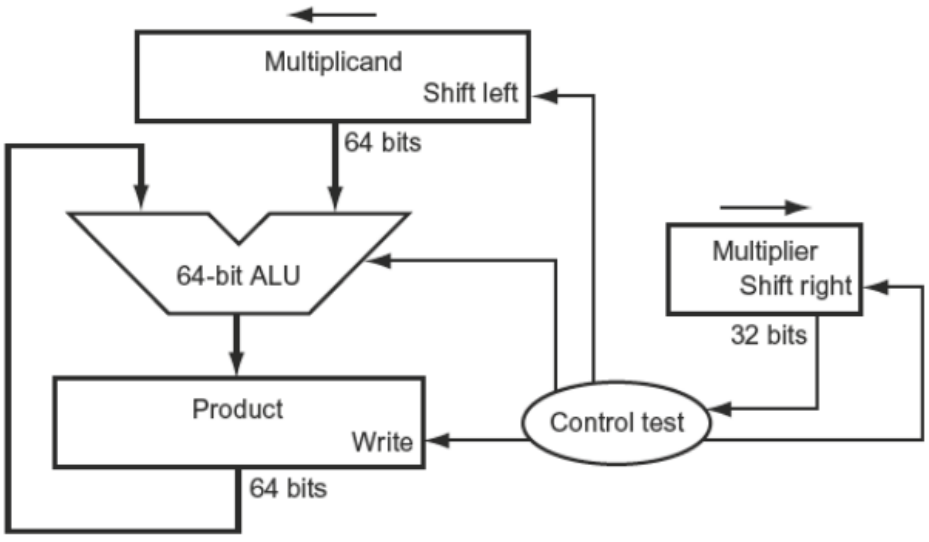


Figure 31. Multiplication hardware representation

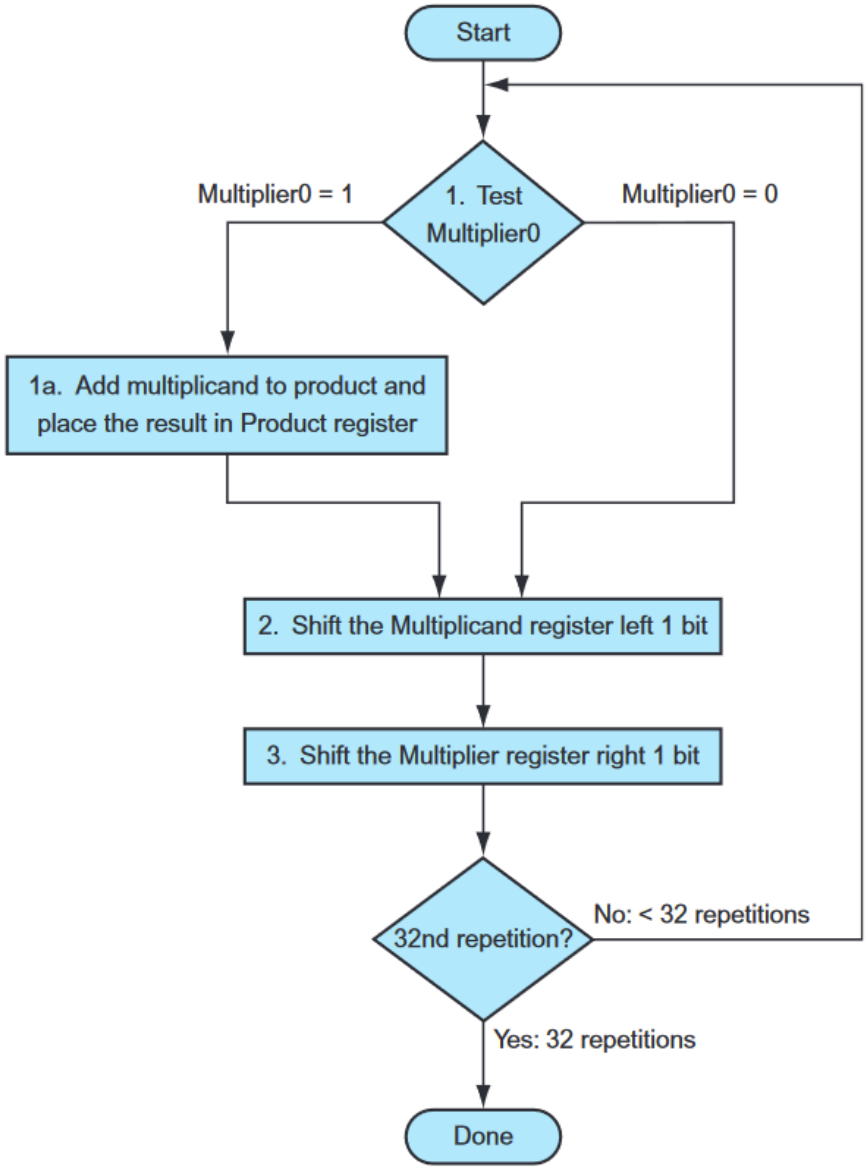


Figure 32. Flowchart for sequential multiplication algorithm

```
module SignedMultiplier32bit(
    input [31:0] a,
    input [31:0] b,
    output reg [31:0] out_hi,
    output reg [31:0] out_lo
    );

    integer i;

    reg [31:0] a_unsigned;
    reg [31:0] b_unsigned;
    reg [63:0] multiplicand;
    reg [31:0] multiplier;
    reg [63:0] product;

    always @ (a or b) begin
        // negate a if negative
        if (a[31] == 1'b1) begin
            a_unsigned = -a;
        end else begin
            a_unsigned = a;
        end

        // negate b if negative
        if (b[31] == 1'b1) begin
            b_unsigned = -b;
        end else begin
            b_unsigned = b;
        end

        multiplicand = {32'b0,a_unsigned};
        multiplier = b_unsigned;
        product = 0;

        // unsigned multiplication logic
        for (i = 0; i < 32; i = i + 1) begin
            // add multiplicand to product if multiplier[i] == 1
            if (multiplier[0] == 1'b1) begin
                product = product + multiplicand;
            end

            // shift multiplicand left 1 bit
            multiplicand = {multiplicand[63:0],1'b0};

            // shift multiplier right 1 bit
            multiplier = {1'b0,multiplier[31:1]};
        end

        // negate product if only a or only b is negative
        if (a[31] ^ b[31] == 1'b1) begin
            product = -product;
        end

        out_hi = product[63:32];
        out_lo = product[31:0];
    end
endmodule
```

Figure 33. Verilog representation of SignedMultiplier32bit

SignedDivider32bit

The divider used in the ALU was designed based on a simple division algorithm. The hardware design is like what is shown in Figure 34 below.

A flowchart for the division algorithm is shown in Figure 35 below. On each cycle it subtracts the division from the remainder. If the remainder stays positive as a result of that operation, the quotient is shifted left by 1 bit, setting the rightmost bit to 1. If the remainder becomes negative, the divisor will be added back to the remainder and the quotient is shifted left, but with the rightmost bit being set to 0 instead. The divisor is shifted right by 1 bit before repeating the cycle. This cycle repeats 33 times.

The module was first implemented as an unsigned 32-bit divider. To make it signed, additional logic was embedded before and after to strip the inputs of their sign and reapply it correctly to the quotient and remainder after the division was complete. The Verilog definition of the SignedDivider32bit module is shown in Figure 36 below.

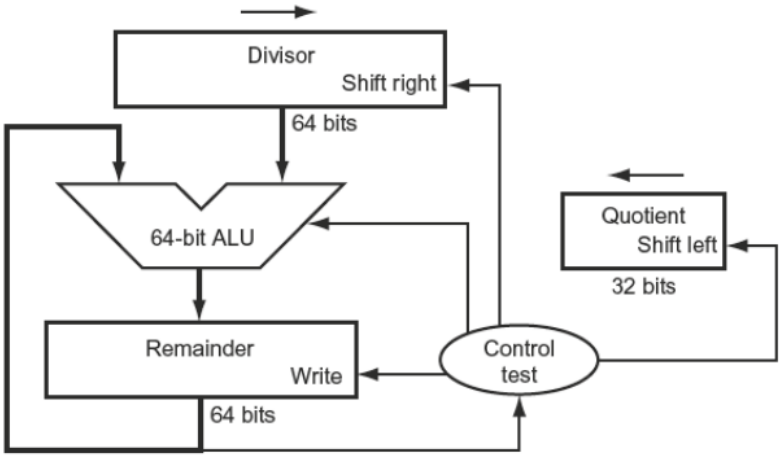


Figure 34. Division hardware representation

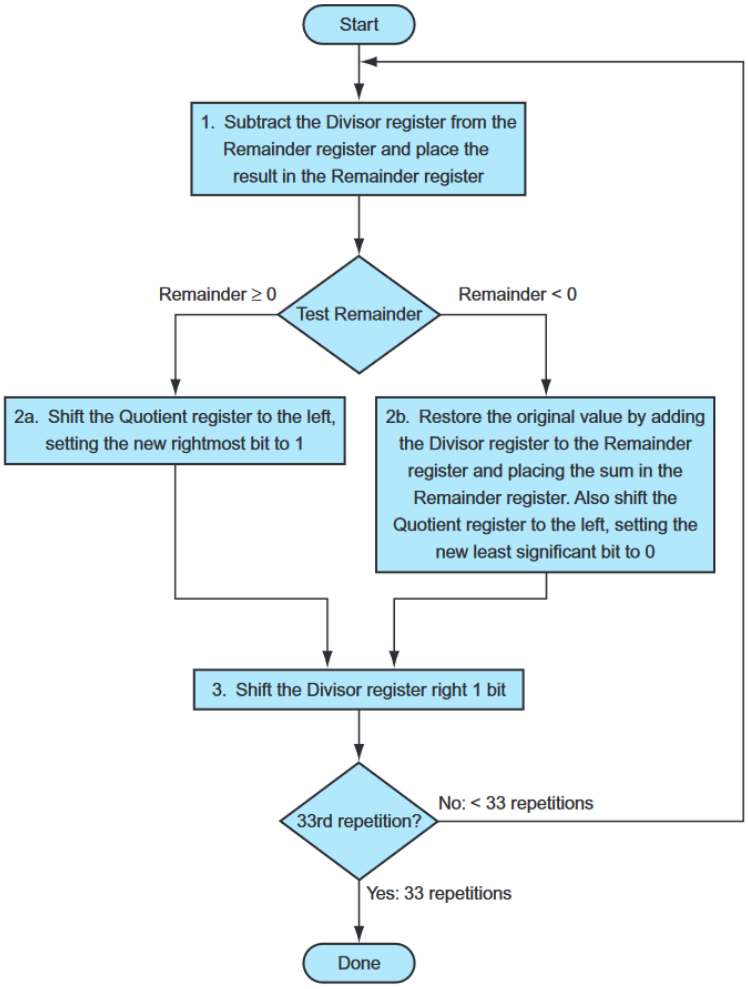


Figure 35. Flowchart for division algorithm

```
module SignedDivider32bit(
    input [31:0] a,
    input [31:0] b,
    output reg [31:0] out_q,
    output reg [31:0] out_r
    );

    integer i;

    reg [31:0] a_unsigned;
    reg [31:0] b_unsigned;
    reg [63:0] divisor;
    reg [31:0] quotient;
    reg [63:0] remainder;

    always @ (a or b) begin
        // negate a if negative
        if (a[31] == 1'b1) begin
            a_unsigned = -a;
        end else begin
            a_unsigned = a;
        end

        // negate b if negative
        if (b[31] == 1'b1) begin
            b_unsigned = -b;
        end else begin
            b_unsigned = b;
        end

        divisor = {b_unsigned,32'b0};
        quotient = 0;
        remainder = {32'b0,a_unsigned};

        // unsigned division logic
        for (i = 0; i < 33; i = i + 1) begin
            // subtrtact divisor from remainder
            remainder = remainder - divisor;

            // check if remainder is >= 0
            if (remainder[63] != 1'b1) begin
                // shift quotient left, setting new rightmost bit to 1
                quotient = {quotient[30:0],1'b1};
            end else begin
                // restore remainder by adding divisor back
                remainder = remainder + divisor;
                // shift quotient left, setting new rightmost bit to 0
                quotient = {quotient[30:0],1'b0};
            end

            // shift divisor right
            divisor = {1'b0,divisor[63:1]};
        end

        // negate quotient if only a or only b is negative
        if (a[31] ^ b[31] == 1'b1) begin
            quotient = -quotient;
        end

        // negate remainder if divident is negative
        if (a[31] == 1'b1) begin
            remainder = -remainder;
        end

        out_q = quotient;
        out_r = remainder[31:0];
    end
endmodule
```

Figure 36. Verilog representation of SignedDivider32bit

Test Plan

Test benches were developed to test the functionality of the SingleCycleProcessor module and all other submodules. The following section - Results/Observations - outlines the test bench used for each module along with their corresponding behavioral results.

SingleCycleProcessor

A test program was designed and run to test the functionality of the SingleCycleProcessor:

$$y = ((a - c) + (d - b) * e) / f$$
with a = 1000, b = 200, c = 300, d = 400, e = 500, and f = 3

The values a through f were stored into data memory at initialization by placing them in the data.txt file described in the DataMemory subsection of the Procedure section. The addresses of a through f correspond to 0 through 5, respectively. The data.txt file can be seen in Figure 13.

With the data set, a program was developed to interact with the memory space to calculate the value of y, storing it back into the data memory on completion. Because the final operation will be divided, both the quotient and remainder will be stored.

The program instructions are as follows:

```
lw $8, 0            # a = 1000
lw $9, 1            # b = 200
lw $10, 2           # c = 300
lw $11, 3           # d = 400
lw $12, 4           # e = 500
lw $13, 5           # f = 3
sub $8, $8, $10     # $8 = a - c = 700
sub $11, $11, $9    # $11 = d - b = 200
mul $11, $11, $12   # $11 = $11 * e = 100000
add $8, $8, $11     # $8 = $8 + $11 = 100700
div $14, $8, $13    # $14 = $8 / f = 33566    = 0x831e
mod $15, $8, $13    # $15 = $8 % f = 2        = 0x0002
sw $14, 6           # y(quotient) = $14
sw $15, 7           # y(remainder) = $15
```

These instructions were manually converted to their binary equivalent according to the MIPS32 revision 6 specification. These instructions were placed in program.txt and loaded into instruction memory on initialization. The contents of program.txt can be seen in Figure 6.

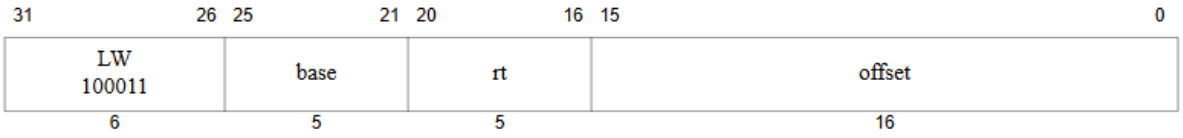
The test bench module for the SingleCycleProcessor is shown below:

```
module SingleCycleProcessor_tb();

    reg clk;

    SingleCycleProcessor M01(.clk(clk));

    initial begin : clock_gen
        integer cycles, i;

        // set number of cycles to run (safe: instructions + 1)
        cycles = 15;

        // first clock cycle
        clk = 0; #20
        cycles = cycles - 1;

        // remaining clock cycles
        for (i = 0; i < cycles; i = i + 1) begin
            clk = 1; #10;
            clk = 0; #10;
        end
        $finish;
    end
endmodule
```

Figure 37. SingleCycleProcessor test bench

The specifications for the instructions listed above are as follows:

lw (load word)



Format: `LW rt, offset(base)` **MIPS32**

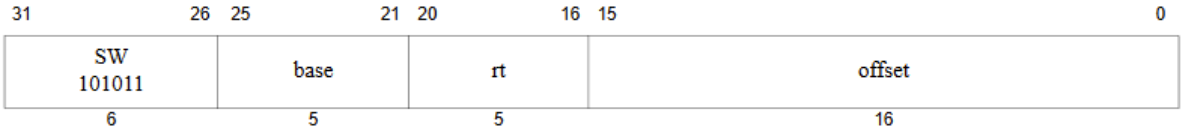
Purpose: Load Word

To load a word from memory as a signed value

Description: `GPR[rt] ← memory[GPR[base] + offset]`

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

sw (store word)



Format: `SW rt, offset(base)` **MIPS32**

Purpose: Store Word

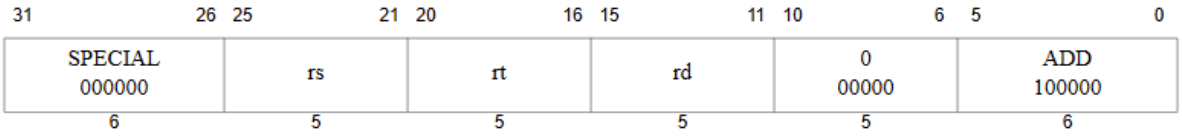
To store a word to memory.

Description: `memory[GPR[base] + offset] ← GPR[rt]`

The least-significant 32-bit word of GPR *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

add (add word)

integer overflow exception: status register = 12



Format: `ADD rd, rs, rt` **MIPS32**

Purpose: Add Word

To add 32-bit integers. If an overflow occurs, then trap.

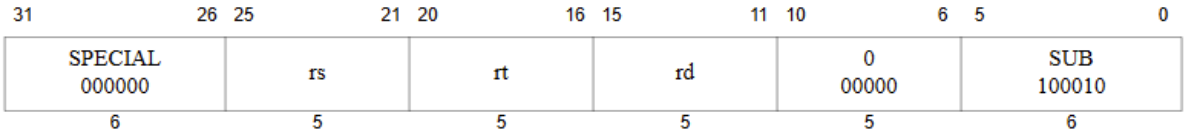
Description: `GPR[rd] ← GPR[rs] + GPR[rt]`

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* to produce a 32-bit result.

- If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs.
- If the addition does not overflow, the 32-bit result is placed into GPR *rd*.

sub (subtract word)

integer overflow exception: status register = 12



Format: `SUB rd, rs, rt` **MIPS32**

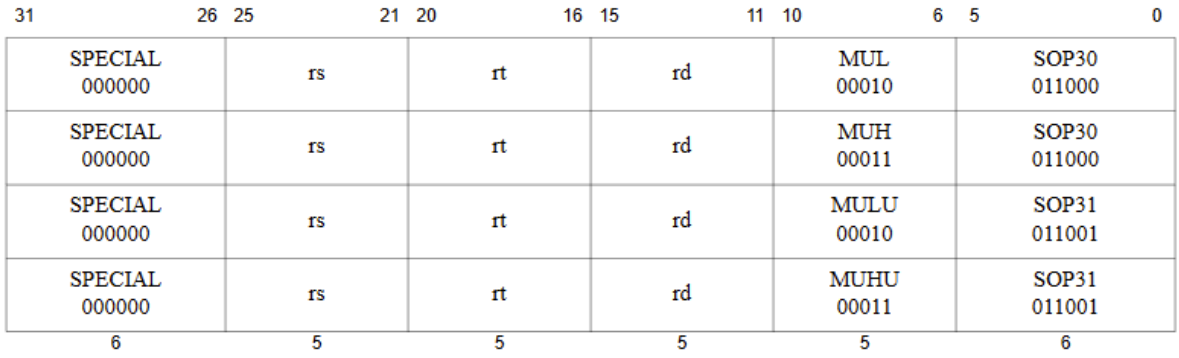
Purpose: Subtract Word

To subtract 32-bit integers. If overflow occurs, then trap.

Description: `GPR[rd] ← GPR[rs] – GPR[rt]`

The 32-bit word value in GPR *rt* is subtracted from the 32-bit value in GPR *rs* to produce a 32-bit result. If the subtraction results in 32-bit 2's complement arithmetic overflow, then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR *rd*.

mul muh mulu muhu (multiply integers (with result to GPR))



31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	MUL 00010	SOP30 011000
SPECIAL 000000	rs	rt	rd	MUH 00011	SOP30 011000
SPECIAL 000000	rs	rt	rd	MULU 00010	SOP31 011001
SPECIAL 000000	rs	rt	rd	MUHU 00011	SOP31 011001
6	5	5	5	5	6

Format: `MUL MUH MULU MUHU`

```
MUL rd,rs,rt                    MIPS32 Release 6
MUH rd,rs,rt                    MIPS32 Release 6
MULU rd,rs,rt                   MIPS32 Release 6
MUHU rd,rs,rt                   MIPS32 Release 6
```

Purpose: Multiply Integers (with result to GPR)

MUL: Multiply Words Signed, Low Word
MUH: Multiply Words Signed, High Word
MULU: Multiply Words Unsigned, Low Word
MUHU: Multiply Words Unsigned, High Word

Description:

```
MUL:   GPR[rd] ← lo_word( multiply.signed( GPR[rs] × GPR[rt] ) )
MUH:   GPR[rd] ← hi_word( multiply.signed( GPR[rs] × GPR[rt] ) )
MULU:  GPR[rd] ← lo_word( multiply.unsigned( GPR[rs] ×GPR[rt] ) )
MUHU:  GPR[rd] ← hi_word( multiply.unsigned( GPR[rs] ×GPR[rt] ) )
```

div mod divu modu (divide integers (with result to GPR))

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	DIV 00010	SOP32 011010
SPECIAL 000000	rs	rt	rd	MOD 00011	SOP32 011010
SPECIAL 000000	rs	rt	rd	DIVU 00010	SOP33 011011
SPECIAL 000000	rs	rt	rd	MODU 00011	SOP33 011011
6	5	5	5	5	6

Format: `DIV MOD DIVU MODU`

`DIV rd,rs,rt` **MIPS32 Release 6**
`MOD rd,rs,rt` **MIPS32 Release 6**
`DIVU rd,rs,rt` **MIPS32 Release 6**
`MODU rd,rs,rt` **MIPS32 Release 6**

Purpose: Divide Integers (with result to GPR)

DIV: Divide Words Signed
MOD: Modulo Words Signed
DIVU: Divide Words Unsigned
MODU: Modulo Words Unsigned

Description:

```
DIV:   GPR[rd] ← ( divide.signed( GPR[rs], GPR[rt] )
MOD:   GPR[rd] ← ( modulo.signed( GPR[rs], GPR[rt] )
DIVU:  GPR[rd] ← ( divide.unsigned( GPR[rs], GPR[rt] )
MODU:  GPR[rd] ← ( modulo.unsigned( GPR[rs], GPR[rt] )
```

Control

The Control module is a simple case statement, which switches based on the passed opcode. The control test bench inputs each of the possible opcodes one at a time.

```
module Control_tb();

    reg [5:0] opcode;
    wire [1:0] alu_op;
    wire reg_dst_sel;
    wire jump_src_sel;
    wire branch_src_sel;
    wire alu_src_sel;
    wire mem_to_reg_sel;
    wire reg_write;
    wire mem_read;
    wire mem_write;

    Control M01(.opcode(opcode),.alu_op(alu_op),.reg_dst_sel(reg_dst_sel),
                .jump_src_sel(jump_src_sel),.branch_src_sel(branch_src_sel),
                .alu_src_sel(alu_src_sel),.mem_to_reg_sel(mem_to_reg_sel),
                .reg_write(reg_write),.mem_read(mem_read),.mem_write(mem_write));

    initial begin
        opcode = 6'b100011; #10 // lw (load word) I-format
        opcode = 6'b101011; #10 // sw (store word) I-format
        opcode = 6'b000100; #10 // beq (branch on equal) I-format
        opcode = 6'b000010; #10 // j (jump) J-format
        opcode = 6'b000000; #10 // SPECIAL (slt,and,or,add,sub,mul,muh,div,mod) R-format
        $finish;
    end
endmodule
```

Figure 40. Control test bench

ProgramCounter

A test bench was developed to test the functionality of the ProgramCounter's initialization to 0 and update function on each clock cycle.

```
module ProgramCounter_tb();

    reg [31:0] pc_next;
    wire [31:0] pc_current;
    reg clk;

    ProgramCounter  M01(.pc_next(pc_next),.pc_current(pc_current),.clk(clk));

    initial begin
        clk = 0; #10 pc_next = 2; #10
        clk = 1; #10 clk = 0; pc_next = 3; #10
        clk = 1; #10 clk = 0; pc_next = 4; #10
        $finish;
    end
endmodule
```

Figure 41. ProgramCounter test bench

InstructionMemory

A test bench was developed to test the functionality of reading from the instruction memory. In 10ns intervals the addresses 0 through 4 were switched between.

```
module InstructionMemory_tb();

    reg [31:0] address;
    wire [31:0] instruction;

    InstructionMemory M1(.address(address), .instruction(instruction));

    initial begin
        #10 address = 0;
        #10 address = 1;
        #10 address = 2;
        #10 address = 3;
        #10 address = 4;
        #10 $finish;
    end
endmodule
```

Figure 42. InstructionMemory test bench

Registers

The Registers test bench reads from two registers, then writes data to one of them. The register should be written to on the first positive edge of a clock cycle after the reg_write signal is set high.

```
module Registers_tb();

    reg [4:0] read_reg_1;
    reg [4:0] read_reg_2;
    reg [4:0] write_reg;
    reg [31:0] write_data;
    reg reg_write;
    reg clk;
    wire [31:0] read_data_1;
    wire [31:0] read_data_2;

    Registers   M01(read_reg_1,read_reg_2,write_reg,write_data,clk,reg_write,read_data_1,read_data_2);

    initial begin
        read_reg_1 = 0;
        read_reg_2 = 1;
        write_reg = 0;
        write_data = 'habcd0123;
        reg_write = 0;
        clk = 0; // 20ns period

        #10 clk = 1;                // read_data should be 0x00000000
        #10 clk = 0;                // read_data should be 0x00000000
        #10 clk = 1;                // read_data should be 0xffffffff
        #10 clk = 0; reg_write = 1; // read_data should be 0xffffffff
        #10 clk = 1;                // read_data should be 0xffffffff
        #10 clk = 0; reg_write = 0; // read_data should be 0xffffffff
        #10 clk = 1;                // read_data should be 0xabcd0123
        #10 clk = 0;                // read_data should be 0xabcd0123
        #10 $finish;
    end
endmodule
```

Figure 43. Registers test bench

ALUControl

The ALUControl test bench tested every possibility of alu_op, func_code, and shamt that would be possible for the instruction subset: `lw, sw, beq, j, slt, and, or, add, sub, mul, muh, div, mod`.

```
module ALUControl_tb();

    reg [1:0] alu_op;
    reg [4:0] shamt;
    reg [5:0] func_code;
    wire [3:0] alu_ctl;

    ALUControl  M1(.alu_op(alu_op),.shamt(shamt),.func_code(func_code),.alu_ctl(alu_ctl));

    initial begin
        alu_op = 2'b00; #10                          // 4'b0010 lw,sw (add)
        alu_op = 2'b01; #10                          // 4'b0011 beq (sub)
        alu_op = 2'b10; func_code = 6'b101010; #10   // 4'b1000 slt
        alu_op = 2'b10; func_code = 6'b100100; #10   // 4'b0000 and
        alu_op = 2'b10; func_code = 6'b100101; #10   // 4'b0001 or
        alu_op = 2'b10; func_code = 6'b100000; #10   // 4'b0010 add
        alu_op = 2'b10; func_code = 6'b100010; #10   // 4'b0011 sub
        alu_op = 2'b10; func_code = 6'b011000; shamt = 5'b00010; #10 // 4'b0100 mul
        alu_op = 2'b10; func_code = 6'b011000; shamt = 5'b00011; #10 // 4'b0101 muh
        alu_op = 2'b10; func_code = 6'b011010; shamt = 5'b00010; #10 // 4'b0110 div
        alu_op = 2'b10; func_code = 6'b011010; shamt = 5'b00011; #10 // 4'b0111 mod
        $finish;
    end
endmodule
```

Figure 44. ALUControl test bench

ALU

The ALU test bench tested sequentially switching between several control and data inputs. The outputs were checked for accuracy.

```
module ALU_tb();

    reg [3:0] ctl;
    reg [31:0] a;
    reg [31:0] b;
    wire [31:0] result;
    wire zero;

    ALU M1(.ctl(ctl),.a(a),.b(b),.result(result),.zero(zero));

    initial begin
        ctl = 4'b0010; a = 3; b = 2; #10 // 5
        ctl = 4'b0011; a = 5; b = 5; #10 // 0
        ctl = 4'b0100; a = -3; b = 7; #10 // -27
        $finish;
    end
endmodule
```

Figure 45. ALU test bench

DataMemory

A test bench was developed to test the functionality of reading and writing to the memory space. For this test bench, a clock was simulated with a 20ns period. First, the value at address 0 is checked for the correct output (0x00000000). Before a positive clock edge, the write signal is set to high, and on the next positive clock edge, the write_data signal is written to memory address 0. This value is read out correctly as 0xabcd1234. Finally, the address is changed to 1, which correctly shows the initialized value of 0x00000000.

Adder32bit

Test benches were developed to test various input possibilities to the adder modules. The inputs are organized into initial blocks which change values in 10ns intervals. The expected results were calculated and commented next to each line which can be matched up to the output signals during simulation.

As the adders increased in bit width, it became infeasible to simulate and verify the full range of possible inputs. In these cases, a few key input scenarios were chosen - usually comprising of the smallest cases, maximum values, and a couple in-between which may or may not require carries.

LeftShifter

A test bench was developed to test various input possibilities to the extender. The inputs are organized into initial blocks which change values in 10ns intervals. The expected results were calculated and commented next to each line which can be matched up to the output signals during simulation.

Only one test case is necessary to determine the functionality of the left shifter: an input of 1. That would show whether the shift and zero padding was functioning correctly.

SignExtender

A test bench was developed to test various input possibilities to the extender. The inputs are organized into initial blocks which change values in 10ns intervals. The expected results were calculated and commented next to each line which can be matched up to the output signals during simulation.

Only the MSB of the input signal should determine the most significant 16 bits of the output. Therefore only two test cases were run - one for a MSB of 0 and one for a MSB of 1.

Mux2to1

A test bench was developed to test various input possibilities to the mux. The inputs are organized into initial blocks which change values in 10ns intervals. The expected results were calculated and commented next to each line which can be matched up to the output signals during simulation.

Because there is no data processing on the input signals, only one value for each of the input buses was tested. To prove functionality, the output should match either a or b entirely, depending on the input sel signal.

SignedAdder32bit

A test bench was developed to test various input possibilities to the adder. The inputs are organized into initial blocks which change values in 10ns intervals. The expected results were calculated and commented next to each line which can be matched up to the output signals during simulation.

The three important cases which needed to be tested to validate functionality of the overflow detector were:

- addition of two positives or two negatives which do not overflow.
- addition of two positives which result in a negative. overflow occurs.
- addition of two negatives which result in a positive. overflow occurs.

SignedMultiplier32bit

Two test benches were created: one for the unsigned multiplier and one for the signed multiplier. The signed one built off the unsigned test bench. Each operation was checked for accuracy.

```
module SignedMultiplier32bit_tb();

    reg[31:0]   a,b;
    wire[63:0]  product;

    SignedMultiplier32bit    M1(.a(a),.b(b),.out_hi(product[63:32]),.out_lo(product[31:0]));

    initial begin
        a = 1; b = 1; #10 // 1 * 1 = 1                                          = 0x 00000000 00000001
        a = 1; b = 1000; #10 // 1 * 1000 = 1000                                 = 0x 00000000 000003e8
        a = 8; b = 3; #10 // 8 * 3 = 24                                         = 0x 00000000 00000018
        a = 40000; b = 1600000000; #10 // 40000 * 1600000000 = 64,000,000,000,000   = 0x 00003A35 29440000
        a = 999; b = 0; #10 // 999 * 0 = 0                                      = 0x 00000000 00000000
        a = -3; b = 7; #10 // -3 * 7 = -21                                      = 0x FFFFFFFF FFFFFFEB (will not work as expected)
        $finish;
    end
endmodule
```

Figure 46. SignedMultiplier test bench

SignedDivider32bit

Two test benches were created: one for the unsigned divider and one for the signed divider. The signed one built off the unsigned test bench so only the signed test bench is shown here. Each operation was checked for accuracy. When evaluating signs, it was important to check not only the quotient, but also the remainder.

```
module SignedDivider32bit_tb();

    reg [31:0]  a,b;
    wire[31:0]  quotient;
    wire[31:0]  remainder;

    SignedDivider32bit    M1(.a(a),.b(b),.out_q(quotient),.out_r(remainder));

    initial begin
        a = 20; b = 20; #10 // 20 / 20 = 1 R 0          = 0x00000001 R 0x00000000
        a = 1; b = 10; #10 // 1 / 10 = 0 R 1            = 0x00000000 R 0x00000001
        a = 8; b = 3; #10 // 8 / 3 = 2 R 2              = 0x00000002 R 0x00000002
        a = 'heeaaaa; b = 'hdd; #10 // 0xEEAAAA / 0xDD  = 0x00011476 R 0x000000CC
        a = 999; b = 0; #10 // 999 / 0 = ???           = 0xFFFFFFFF R 0x000003E7 (undefined behavior)
        a = 21; b = -3; #10 // 21 / -3 = -7 R 0         = 0xFFFFFFF9 R 0x00000000
        a = 14; b = 4; #10 // 14 / 4 = 3 R 2           = 0x00000003 R 0x00000002
        a = 14; b = -4; #10 // 14 / -4 = -3 R 2         = 0xFFFFFFFD R 0x00000002
        a = -14; b = 4; #10 // -14 / 4 = -3 R -2        = 0xFFFFFFFD R 0xFFFFFFFE
        a = -14; b = -4; #10 // -14 / -4 = 3 R -2       = 0x00000003 R 0xFFFFFFFE
        $finish;
    end
endmodule
```

Figure 47. SignedDivider test bench

Results/Observations

SingleCycleProcessor

At the beginning of simulation, the data memory contains the correct data entries as specified in data.txt.

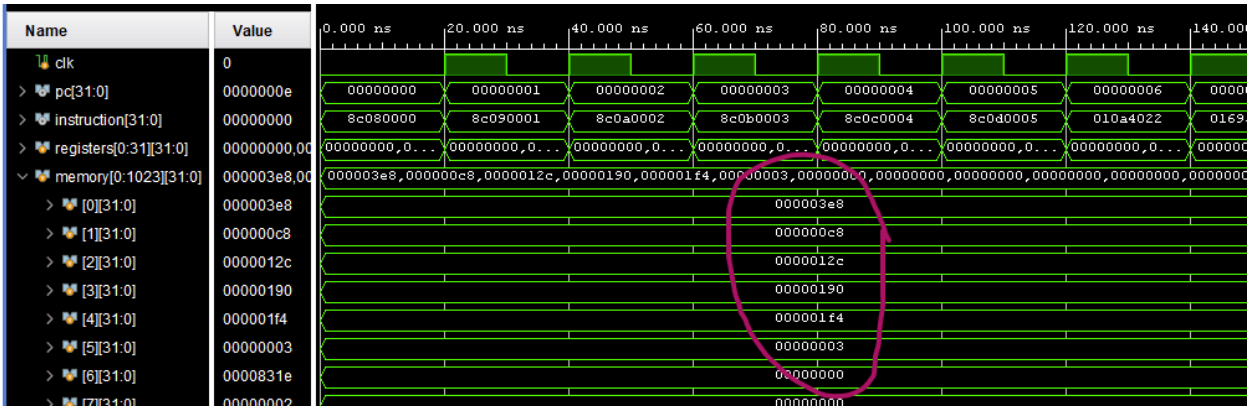


Figure 48. Data memory

As the program executes, one can see that the program counter (instruction address) and instruction progress correctly (according to the contents of program.txt).

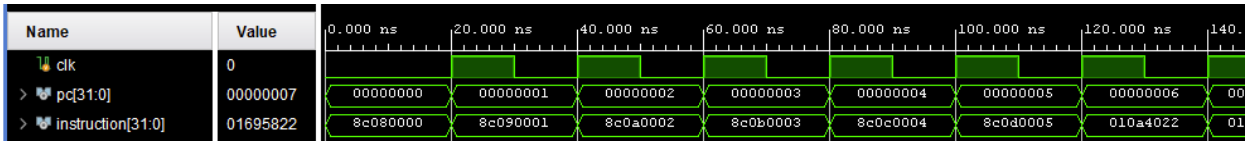


Figure 49. PC and instructions

As the various memory loads and register writes occur, one can see the register fields update as expected. They each happen on the positive clock edge after each instruction is executed.

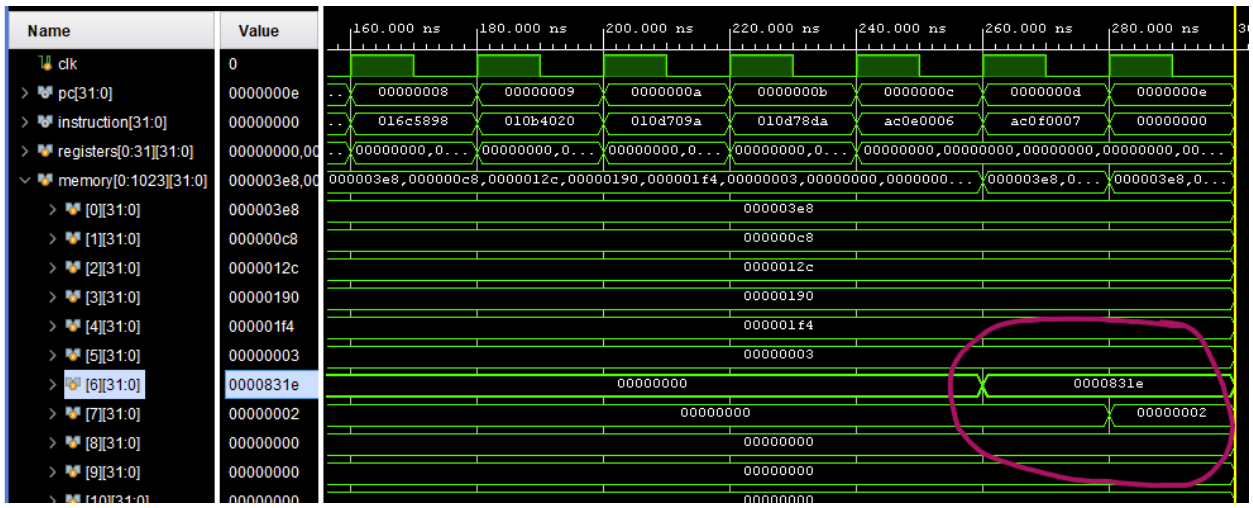


Figure 50. Final data memory stores

At the end of program execution, one can see that the correct value of y: 33566 (0x831e) with remainder 2, is written to data memory locations 6 and 7, respectively. This indicates that the

SingleCycleProcessor is functioning as expected for the test program outlined in the previous section.

Control

The Control testbench outputs match the values in the Control case statement exactly.

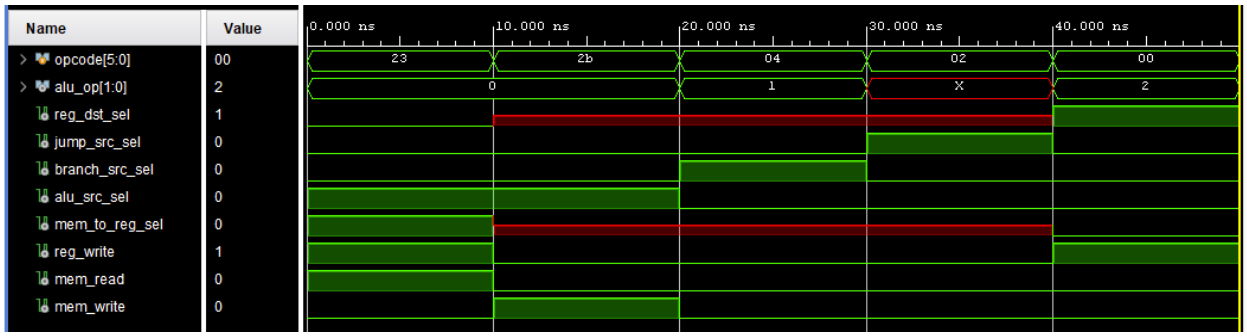


Figure 51. Control test bench simulation

ProgramCounter

The ProgramCounter testbench simulation demonstrated successfully that the program counter begins at 0x00000000 and updates to the value of pc_next on each clock cycle.

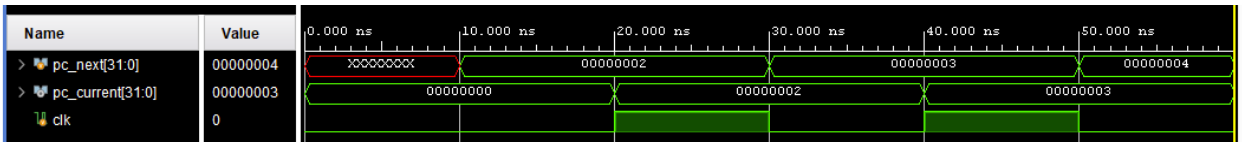


Figure 52. ProgramCounter test bench simulation

InstructionMemory

The test bench snippet for the instruction memory is shown in Figure 53 below. The test simulation is shown in the following figure. The expected and simulated outputs match with no error (the instruction bus in Figure 5 matches the contents of program.txt).

```
InstructionMemory M1(.address(address), .instruction(instruction));

initial begin
    #10 address = 0;
    #10 address = 1;
    #10 address = 2;
    #10 address = 3;
    #10 address = 4;
    #10 $finish;
end
```

Figure 53. Instruction Memory test bench

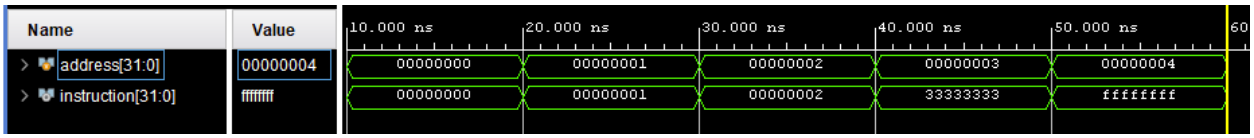


Figure 54. Instruction Memory test bench simulation

The Instruction Memory module simulated with no detectable errors - indicating that it is suitable for integration into the full single cycle processor.

Registers

The Register test bench simulation showed that registers can be read from and written to - indicating that it is suitable for integration into the full single cycle processor.

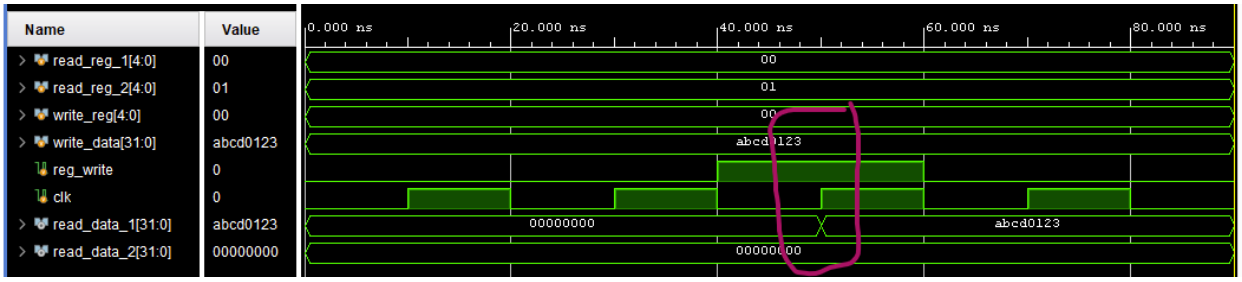


Figure 55. Registers test bench simulation

ALUControl

The ALUControl test bench simulation alu_ctl output selected the correct ALU operation for each possible input combination (all specified instructions).

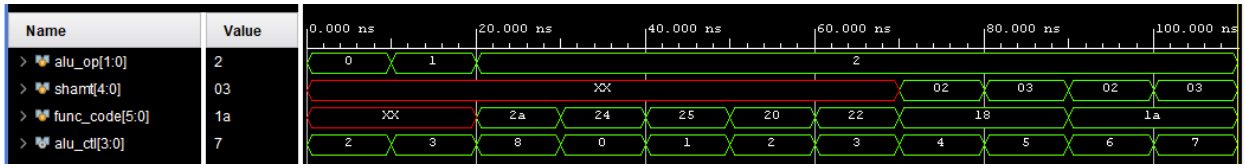


Figure 56. ALUControl test bench simulation

ALU

The ALU test bench simulation result output generated the correct value for each function tested. The primary objectives of this test were to validate the functionality of the zero signal, and validate the operations were occurring as normal.

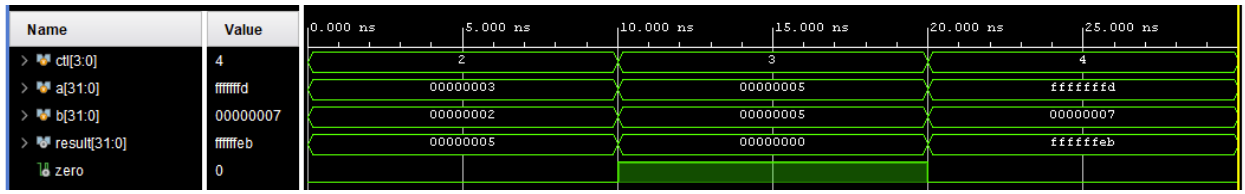


Figure 57. ALU bench simulation

DataMemory

The test bench snippet for the data memory is shown in Figure 58 below. The test simulation is shown in the following figure. The expected (comments in test bench) and simulated (out) outputs match with no error.

```
module DataMemory_tb();

    reg [31:0] address;
    reg [31:0] write_data;
    reg mem_write;
    reg mem_read;
    reg clk;
    wire [31:0] read_data;

    DataMemory M1(.address(address), .write_data(write_data), .read_data(read_data), .mem_write(mem_write), .mem_read(mem_read), .clk(clk));

    initial begin
        address = 0;
        write_data = 'habcd0123;
        mem_write = 0;
        mem_read = 0;
        clk = 0; // 20ns period

        #10 clk = 1;                // read_data should be 0xffffffff
        #10 clk = 0;                // read_data should be 0xffffffff
        #10 clk = 1; mem_read = 1;  // read_data should be 0x00000000
        #10 clk = 0;                // read_data should be 0x00000000
        #10 clk = 1; mem_read = 0;  // read_data should be 0xffffffff
        #10 clk = 0;                // read_data should be 0xffffffff
        #10 clk = 1;                // read_data should be 0xffffffff
        #10 clk = 0; mem_write = 1; // read_data should be 0xffffffff
        #10 clk = 1;                // read_data should be 0xffffffff
        #10 clk = 0; mem_write = 0; // read_data should be 0xffffffff
        #10 clk = 1; mem_read = 1;  // read_data should be 0xabcd0123
        #10 clk = 0;                // read_data should be 0xabcd0123
        #10 clk = 1; address = 1;   // read_data should be 0x00000000
        #10 clk = 0;                // read_data should be 0x00000000
        #10 $finish;
    end
endmodule
```

Figure 58. Data Memory test bench

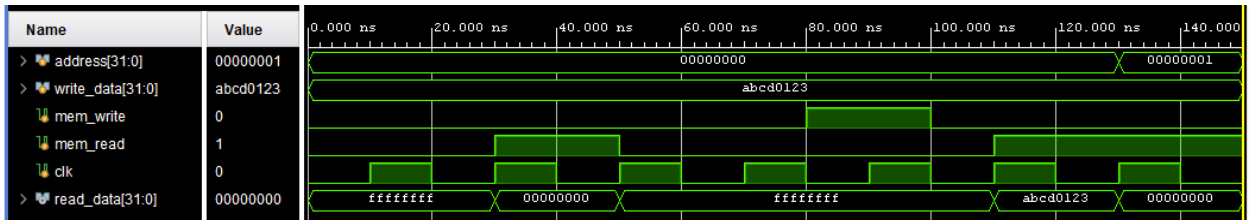


Figure 59. Data Memory test bench simulation

The Data Memory module simulated with no detectable errors - indicating that it is suitable for integration into the full single cycle processor.

Adder32bit

All modules in the 32-bit adder hierarchy were tested with their own test benches. For brevity purposes, the simulation results for only the half, 4-bit, and 32-bit adders are shown below.

The test bench snippet for the half adder is shown in Figure 60 below. The test simulation is shown in the following figure. The expected (comments in test bench) and simulated (sum and c_out lines in simulation) outputs match with no error.

```
HalfAdder   M1(.sum(sum),.c_out(c_out),.a(a),.b(b));

initial
    begin
        #10 a = 0; b = 0;   // 0b 0 0
        #10 a = 0; b = 1;   // 0b 0 1
        #10 a = 1; b = 0;   // 0b 0 1
        #10 a = 1; b = 1;   // 0b 1 0
        #10 $finish;
    end
```

Figure 60. half adder test bench

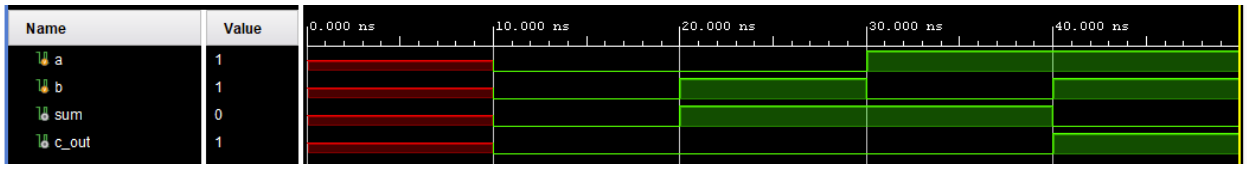


Figure 61. half adder test bench simulation

The test bench snippet for the 4-bit adder is shown in Figure 62 below. The test simulation is shown in the following figure. The expected (comments in test bench) and simulated (sum and c_out lines in simulation) outputs match with no error.

```
Adder4bit   M1(.sum(sum),.c_out(c_out),.a(a),.b(b),.c_in(c_in));

initial begin
    #10 a = 0; b = 0; c_in = 0;     // 0b 0 0000
    #10 a = 0; b = 0; c_in = 1;     // 0b 0 0001
    #10 a = 0; b = 1; c_in = 0;     // 0b 0 0001
    #10 a = 0; b = 1; c_in = 1;     // 0b 0 0010
    #10 a = 0; b = 2; c_in = 0;     // 0b 0 0010
    #10 a = 0; b = 2; c_in = 1;     // 0b 0 0011

    #10 a = 15; b = 15; c_in = 0;   // 0b 1 1110
    #10 a = 15; b = 15; c_in = 1;   // 0b 1 1111

    #10 a = 7; b = 7; c_in = 1;     // 0b 0 1111
    #10 a = 7; b = 8; c_in = 1;     // 0b 1 0000
    #10 a = 5; b = 5; c_in = 1;     // 0b 0 1011
    #10 $finish;
end
```

Figure 62. 4-bit adder test bench

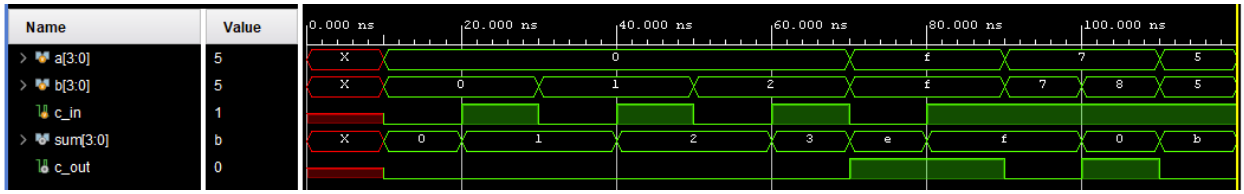


Figure 63. 4-bit adder test bench simulation

The test bench snippet for the 32-bit adder is shown in Figure 64 below. The test simulation is shown in the following figure. The expected (comments in test bench) and simulated (sum and c_out lines in simulation) outputs match with no error.

```
Adder32bit   M1(.sum(sum),.c_out(c_out),.a(a),.b(b),.c_in(c_in));

initial begin
    #10 a = 0; b = 0; c_in = 0;      // 0x 0 0000 0000
    #10 a = 0; b = 0; c_in = 1;      // 0x 0 0000 0001
    #10 a = 0; b = 2; c_in = 0;      // 0x 0 0000 0002
    #10 a = 0; b = 2; c_in = 1;      // 0x 0 0000 0003

    #10 a = 'hFFFFFFFF; b = 'hFFFFFFFF; c_in = 0;    // 0x 1 FFFF FFFE
    #10 a = 'hFFFFFFFF; b = 'hFFFFFFFF; c_in = 1;    // 0x 0 FFFF FFFF

    #10 a = 'h7FFF; b = 'h7FFF; c_in = 1;    // 0x 0 0000 FFFF
    #10 a = 'h7FFF; b = 'h8000; c_in = 1;    // 0x 0 0001 0000
    #10 $finish;
end
```

Figure 64. 32-bit adder test bench

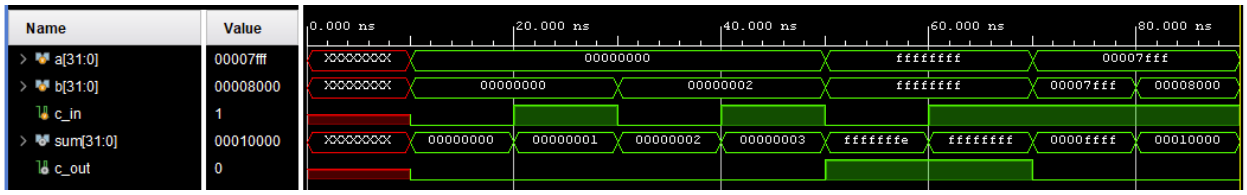


Figure 65. 32-bit adder test bench simulation

The 32-bit adder module simulated with no detectable errors - indicating that it is suitable for integration into the full single cycle processor.

LeftShifter

The test bench snippet for the Left Shifter is shown in Figure 66 below. The test simulation is shown in the following figure. The expected (comments in test bench) and simulated (out) outputs match with no error.

```
LeftShifter M1(.out(out),.in(in));

initial begin
    #10 in = 32'b1111;      // 0x 003C
    #10 in = 32'b0000;      // 0x 0000
    #10 $finish;
end
```

Figure 66. Left Shifter test bench

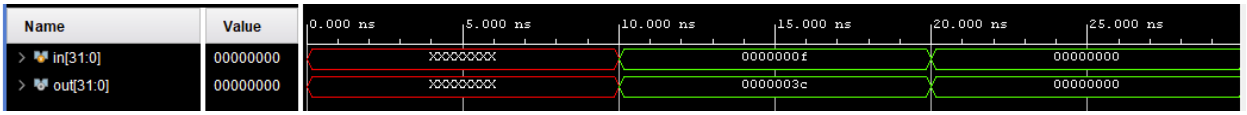


Figure 67. Left Shifter test bench simulation

The sign extender module simulated with no detectable errors - indicating that it is suitable for integration into the full single cycle processor.

SignExtender

The test bench snippet for the Sign Extender is shown in Figure 68 below. The test simulation is shown in the following figure. The expected (comments in test bench) and simulated (out) outputs match with no error.

```
SignExtender M1(.out(out),.in(in));

initial begin
    #10 in = 'h00FF;      // 0x 0000 00FF
    #10 in = 'hF0FF;      // 0x FFFF F0FF
    #10 $finish;
end
```

Figure 68. Sign Extender test bench

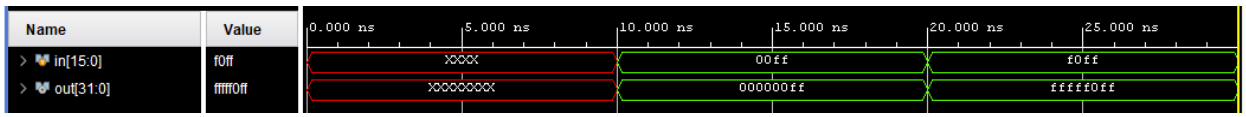


Figure 69. Sign Extender test bench simulation

The sign extender module simulated with no detectable errors - indicating that it is suitable for integration into the full single cycle processor.

Mux2to1

The test bench snippet for the 2-to-1 Multiplexer is shown in Figure 70 below. The test simulation is shown in the following figure. The expected (comments in test bench) and simulated (out) outputs match with no error.

```
Mux2to1 M1(.out(out),.a(a),.b(b),.sel(sel));

initial begin
    #10 a = 'hFFFFFFFF; b = 'h00000000; sel = 'b0;      // 0x FFFF FFFF
    #10 a = 'hFFFFFFFF; b = 'h00000000; sel = 'b1;      // 0x 0000 0000
    #10 $finish;
end
```

Figure 70. half adder test bench

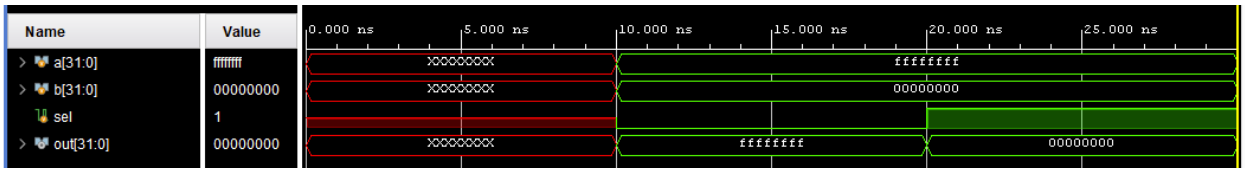


Figure 71. half adder test bench simulation

The 2-to-1 multiplexer module simulated with no detectable errors - indicating that it is suitable for integration into the full single cycle processor.

SignedAdder32bit

The test bench snippet for the adder is shown in Figure 72 below. The test simulation is shown in the following figure. The expected (comments in test bench) and simulated (out) outputs match with no error.

```
SignedAdder32bit    M1(.sum(sum),.c_out(c_out),.a(a),.b(b),.c_in(c_in),.overflow(overflow));

initial begin
    // signed 32-bit integer range: [-2147483648 to 2147483647]

    // addition of two positive or two negatives which do not overflow.
    #10 a = 1; b = 1; c_in = 0;     // sum =  2, overflow = 0, c_out = 0;
    #10 a = -1; b = -1; c_in = 0;   // sum = -2, overflow = 0, c_out = 0;

    // addition of two positives which result in a negative. overflow occurs.
    #10 a = 2147483647; b = 1; c_in = 0;   // sum = -2147483648, overflow = 1, c_out = 0;

    // addition of two negatives which result in a positive. overflow occurs.
    #10 a = -2147483648; b = -1; c_in = 0;   // sum = 2147483647, overflow = 1, c_out = 1;
    #10 $finish;
end
```

Figure 72. Signed Adder test bench

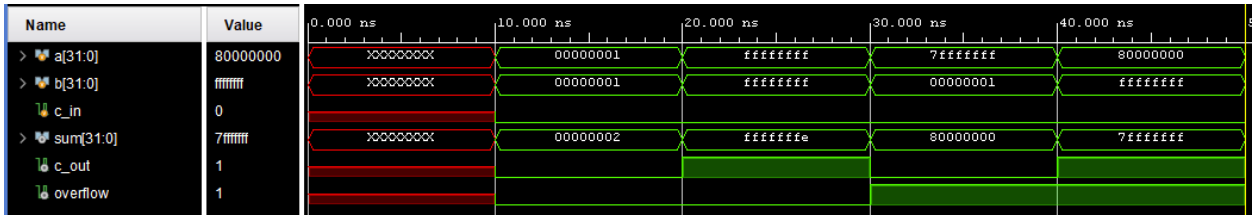


Figure 73. Signed Adder test bench simulation

The Signed 32-bit Integer Adder module simulated with no detectable errors - indicating that it is suitable for integration into the full single cycle processor.

SignedMultiplier32bit

The signed multiplier test bench simulated without errors. Each numerical result and sign matched the inputs exactly.

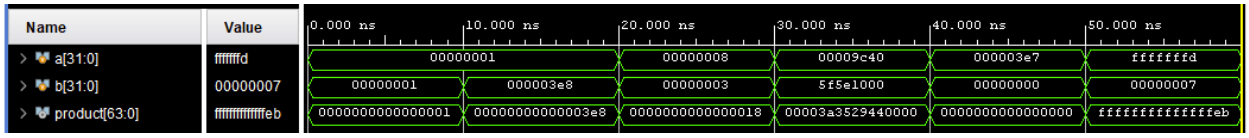


Figure 74. Signed Multiplier test bench simulation

SignedDivider32bit

The signed divider test bench simulated without errors. Each numerical result and sign matched the inputs exactly. Particular care was taken to add a test for every quotient/remainder sign combination.

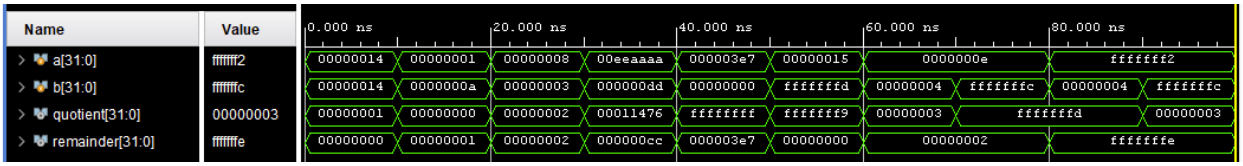


Figure 75. Signed Divider test bench simulation

Conclusion

Due to the hierarchical design, the top level processor module contains almost no control logic. This project was effective at teaching how processors are designed and controlled.

This implementation follows the revision 6 specification for MIPS32 closely with few exceptions. Integer overflow trapping was not implemented for the add and sub instructions, and the zero register is writable.

The SingleCycleProcessor executed the test program without errors.