# ECSE318 Homework 3

Benjamin Scholar

CaseID: bbs27

Email: scholar@case.edu

## Problem 1:

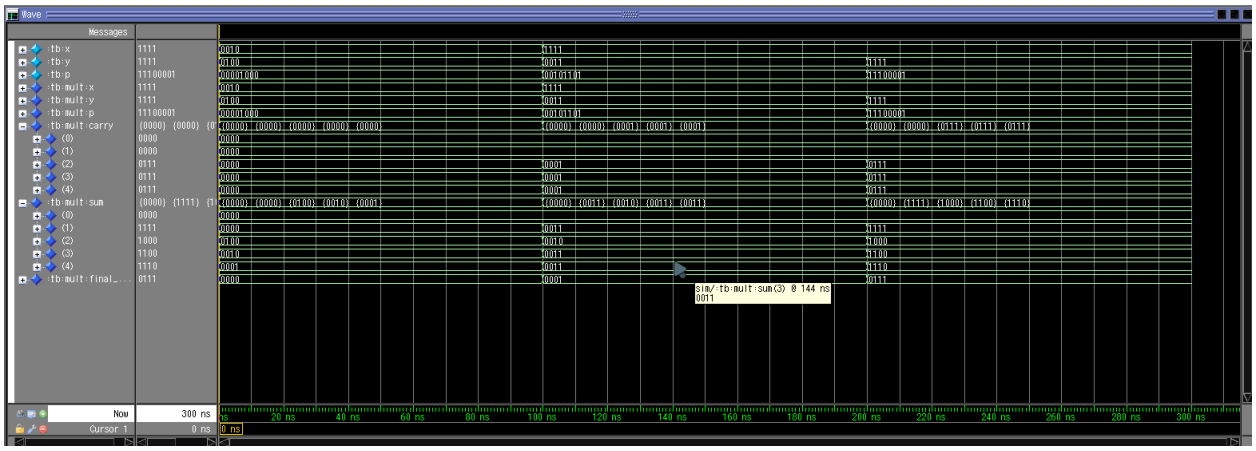
Files:

- Full_adder.vhd - implements a full adder module
- Csa_module.vhd - csa_module, essentially a row of full adders
- Prop_adder.vhd - ripple carry adder, made of full adders
- multiplier .vhd - no-clocked multiplier module
- Tb.v - testbench for the multiplier module

This design is essentially the same as the verilog implementation from homework 1. The multiplier module consists of rows of carry-save adder modules with a final ripple carry adder for the final stage. For each row, the csa_module entity is used. The inputs to each of these rows is generated using a generate loop and a block statement. The block statement allows for each module to have a uniquely named input signal that can be declared within the generate loop. This could probably be done with another array of buses, but I found that this solution is more elegant.

The design was tested with the same test cases as homework 1 (2*4, 15*3,15*15). The printed output of the module and the waveforms generated by the module can be seen below. All outputs of the test bench were found to match the expected behavior.

```
vsim -c work.tb -do "run -all"
Reading /mgc/anacad2009_2c/questasim/v6.5f/tcl/vsim/pref.tcl

# 6.5f

# vsim -do {run -all} -c work.tb
# //  QuestaSim-64 6.5f Jun 16 2010 Linux 5.15.0-52-generic
# //
# //  Copyright 1991-2010 Mentor Graphics Corporation
# //             All Rights Reserved.
# //
# //  THIS WORK CONTAINS TRADE SECRET AND
# //  PROPRIETARY INFORMATION WHICH IS THE PROPERTY
# //  OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS
# //  AND IS SUBJECT TO LICENSE TERMS.
# //
# Loading work.tb
# Loading std.standard
# Loading ieee.std_logic_1164(body)
# Loading ieee.numeric_std(body)
# Loading ieee.std_logic_arith(body)
# Loading ieee.std_logic_unsigned(body)
# Loading work.multiplier(rtl)
# Loading work.csa_module(rtl)
# Loading work.full_adder(rtl)
# Loading work.prop_adder(rtl)
# run -all
# x=2, y=4 ⇒ p=8
# x=15, y=3 ⇒ p=45
# x=15, y=15 ⇒ p=225
```



**Problem 2:**

Files:

- Full_adder.vhd - implements a full adder
- Register.vhd - creates a 1 bit wide adder entity (reg1)
- Shift_register.vhd - variable width, unidirectional shift register
- Serial_adder.vhd - serial adder with variable width shift registers
- Tb.v - testbench for serial_adder entity

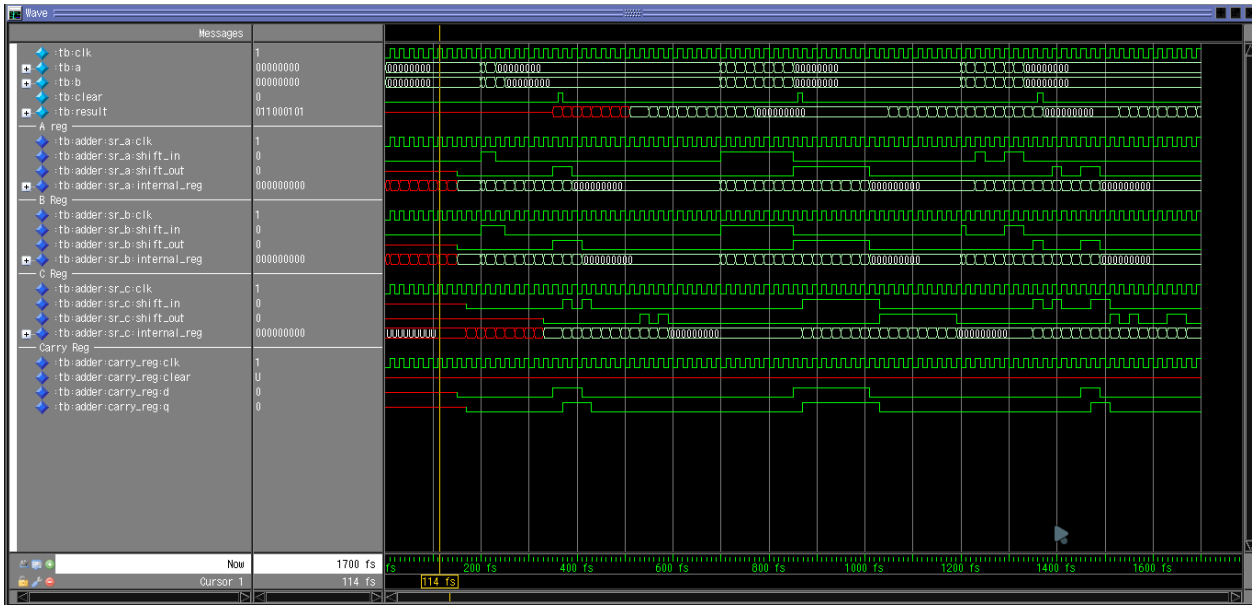
The serial_adder module consists of 3 shift registers, 1 single-bit d-flip-flop, and a full adder. The shift_register module consists of a sequence of the reg1 entity (Defined in the register.vhd file). On each clock cycle, each bit is passed to the next register in the sequence. The carry bit register is set to the low state by the clear input line when addition should be performed.

Some work was needed to get the timing of the testbench working correctly. I ended up using a verilog process to automate the testing sequence. This can be seen in the image below.

```
task test;
  input [7:0] x, y;

  begin
    a = x;
    b = y;
    #160;
    clear = 1'b1;
    #10;
    clear = 1'b0;
    #150;
    #160;
    #20;
    $display("%0d + %0d = %0d, cout=%0b", x, y, result[7:0], result[8]);
  end
endtask
```

The test cases can be seen in the output image below. All the outputs matched the expected behavior of the module. The waveform can also be seen below the image depicting the testbench output.

```
❯ vsim -c work.tb -do "run -all"
Reading /mgc/anacad2009_2c/questasim/v6.5f/tcl/vsim/pref.tcl

# 6.5f

# vsim -do {run -all} -c work.tb
# //  QuestaSim-64 6.5f Jun 16 2010 Linux 5.15.0-52-generic
# //
# //  Copyright 1991-2010 Mentor Graphics Corporation
# //              All Rights Reserved.
# //
# //  THIS WORK CONTAINS TRADE SECRET AND
# //  PROPRIETARY INFORMATION WHICH IS THE PROPERTY
# //  OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS
# //  AND IS SUBJECT TO LICENSE TERMS.
# //
# Loading work.tb
# Loading std.standard
# Loading ieee.std_logic_1164(body)
# Loading ieee.numeric_std(body)
# Loading work.serial_adder(rtl)
# Loading work.shift_register(struct)
# Loading work.reg1(struct)
# Loading work.full_adder(rtl)
# run -all
# 3 + 7 = 10, cout=0
# 255 + 255 = 254, cout=1
# 100 + 97 = 197, cout=0
```



**Problem 3:**

Files:

- Dff.vhd - implementation of a single-bit D-flip-flop
- State_machine.vhd - contains the state machine specified in the homework
    - Contains two architectures: one structural, one behavioral
- Both.vhd - entity that contains an instance of each architecture of the state_machine entity for testing purposes, also generates an output signal which indicates any difference between the two state machines' outputs
- Tb.v - testbench for the both entity

The state machine specified for this portion of the homework was implemented in a near-identical manner to how I did so in homework 1. I chose to write two separate architectures for the same entity declaration. One architecture holds the structural version and the other holds the behavioral version. Both of the versions of the state machine were instantiated in a module, named *both*, mostly because I could not figure out how to select a vhdl entity's architecture from within the verilog test bench file. I am unsure if there's a better way to do this. The both modules also take the XOR of the output of both versions of the module. This output will go to the 1 state if there is any difference between the output of the two modules. The terminal output of this module can be seen below. The behavior matches the behavior of the circuit from homework 1, and matches the desired behavior of the circuit.

```
# //
# Loading work.tb
# Loading std.standard
# Loading ieee.std_logic_1164(body)
# Loading work.both_fsm(rtl)
# Loading work.state_machine(struct)
# Loading work.dff(behav)
# Loading work.state_machine(behav)
# run -all
# w=1, e=1 ⇒ out1=x, out2=x | out_diff=x
# w=1, e=1 ⇒ out1=0, out2=0 | out_diff=0
# w=0, e=0 ⇒ out1=0, out2=0 | out_diff=0
# w=0, e=0 ⇒ out1=1, out2=1 | out_diff=0
# w=0, e=1 ⇒ out1=1, out2=1 | out_diff=0
# w=0, e=1 ⇒ out1=0, out2=0 | out_diff=0
# w=0, e=0 ⇒ out1=0, out2=0 | out_diff=0
# w=0, e=1 ⇒ out1=0, out2=0 | out_diff=0
# w=0, e=0 ⇒ out1=0, out2=0 | out_diff=0
```

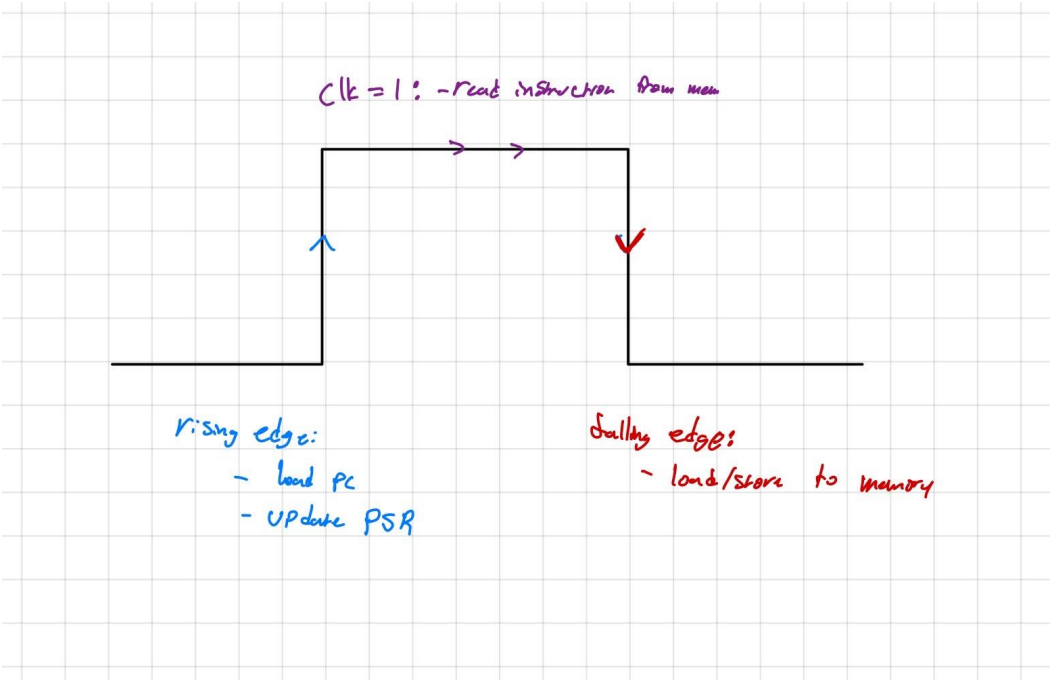
**Problem 4:**

Files:

- Sign_extend.v - sign extension module
- Alu.v - alu implementation
- Barrel_shift.v - variable width, bidirectional barrel shifter
- Barrel_tb.v - testbench for barrel shifter
- Rf.v - register file module
- Mem.v - memory module
- Cpu.v - cpu module
- Soc.v - module that contains clock generator, cpu, memory, and interconnects between these components.

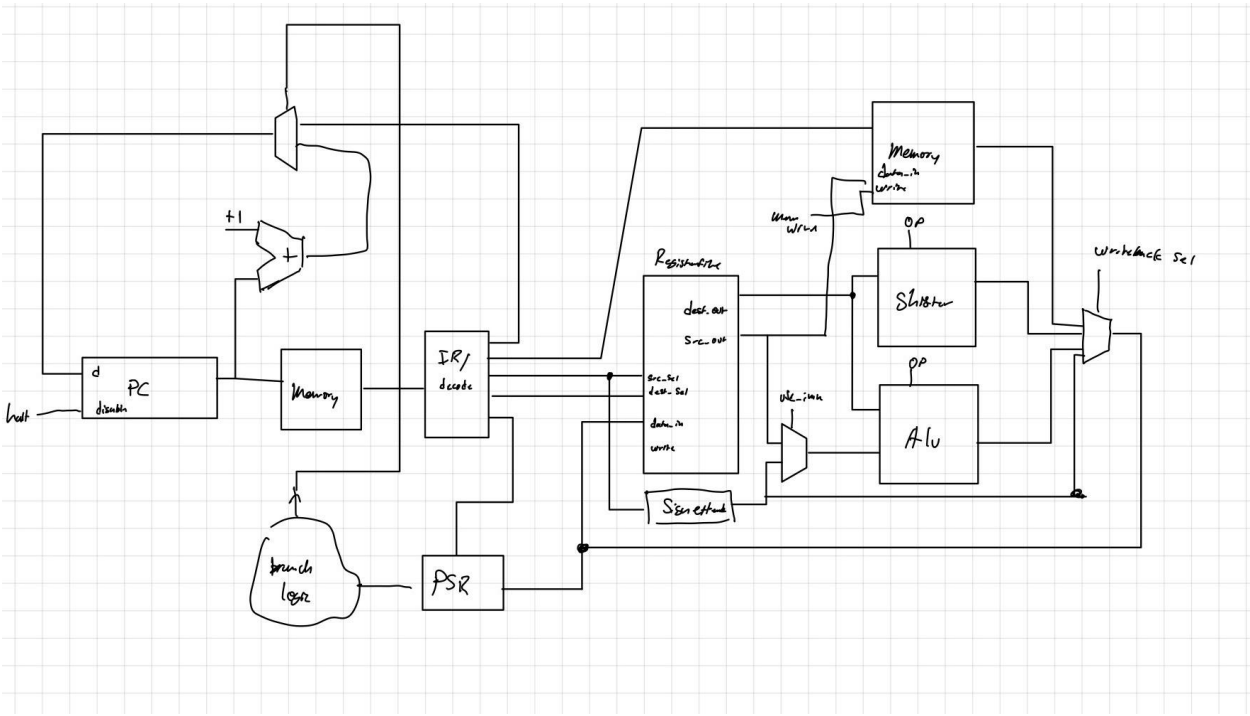
I will go over the general design decisions made for my implementation of the Risc V architecture in this section of the report.

Timing was one of the most important things for me to figure out in this part of the lab, namely when each component updates its values based on the clock signal. I decided that the program counter will read its new value on the positive edge of the clock as a starting point. The memory module must be able to perform an operation on both edges of the clock if we wish to have this implementation run on a single clock cycle per instruction. Thus the memory module was designed to be able to read/write on both edges of the clock. Because the CPU is non-pipelined and single cycle, I used a latch to store the instruction read from memory. When the clock is high, the instruction latch will be able to change state,when the clock is low, the value in the instruction latch cannot be changed. This implementation detail allows for data to be loaded and stored from data on the negative edge of the clock. Finally, the processor status register and register file writeback are both done on the positive edge of the clock as well. Timing diagram:




After each instruction is read from memory, all control lines are to be set accordingly. In my implementation, this is done with a switch statement with the opcode as the selector. In a real implementation it would be better to do this with some sort of microcode implementation but I figured that'd be overkill for simulation. The control lines include selected ALU operations, immediate selection, memory writeback, register file writeback, etc.

A rudimentary diagram of the structure of the CPU can be found below.




For each operation, data from the register file and the sign-extended immediate are used as the inputs into the shifter, ALU, and the memory. The ALU handles the NOP(add with no writeback), XOR,ADD, and CMP instructions. The shifter handles the SHF and ROT instructions. I updated an old implementation of a barrel shifter that I used for a personal project. The shifter is implemented in a structural manner using generate loops. I have modified it to support rotation operations. Additionally, I had to modify the inputs to the shifter in a behavioral block in order to comply with the shifting requirements specified in the homework document (+- 16 instead of +-31 placed for a 32 bit system). The value to be written back to the registerfile is selected from the ALU, shifter, extended immediate value, and the read data from memory using a control line. The selected value is also used to update the PSR register given that the given operation should update/clear the PSR.

Branching functionality is implemented using the values within the PSR register. The CC code from the instruction is to retrieve the corresponding value from the PSR. If that value is logic high, then a mux will be used to select the branch destination address instead of the incremented PC value. The halt instruction is implemented by preventing the program counter from updating, thus resulting in the halt instruction continuously being read.

The program counter starts from a value of 0xF hex in order to be able to run the instruction in memory address 0x10. I decided to implement it this way in order to leave memory addresses 0x0 through 0xF for input data (for the homework assignment). Programs are written in binary in .txt files then loaded into the memory module using the $readmemb function builtin to verilog. This allows for programs to be selected between (commented in and out) in a more convenient way.

The assembly program for computing the two's complement representation of the number stored in address 1 would be as follows. The binary representation of this can be seen in *p4.txt*.

```
ld r0,0
cmp r0,r0
add r0,#1
str 1,r0
hlt
```

Which translates to this in binary.

```
00010000000000000000000000000000 // ld r0,0
10010000000000000000000000000000 // cmp r0,r0
01011000000000000001000000000000 // add r0,#1
00100100000000000000000000000001 // str 1,r0
10000000000000000000000000000000 // hlt
```

This program is loaded into memory as follows, where memory location 0 is set to 0x6.

```
// uncomment for problem 4
$readmemb("p4.txt", data);
data[0] = 32'd6;
```

For ease of determining the answers of each of the following programs, every store operation will be printed to the console.

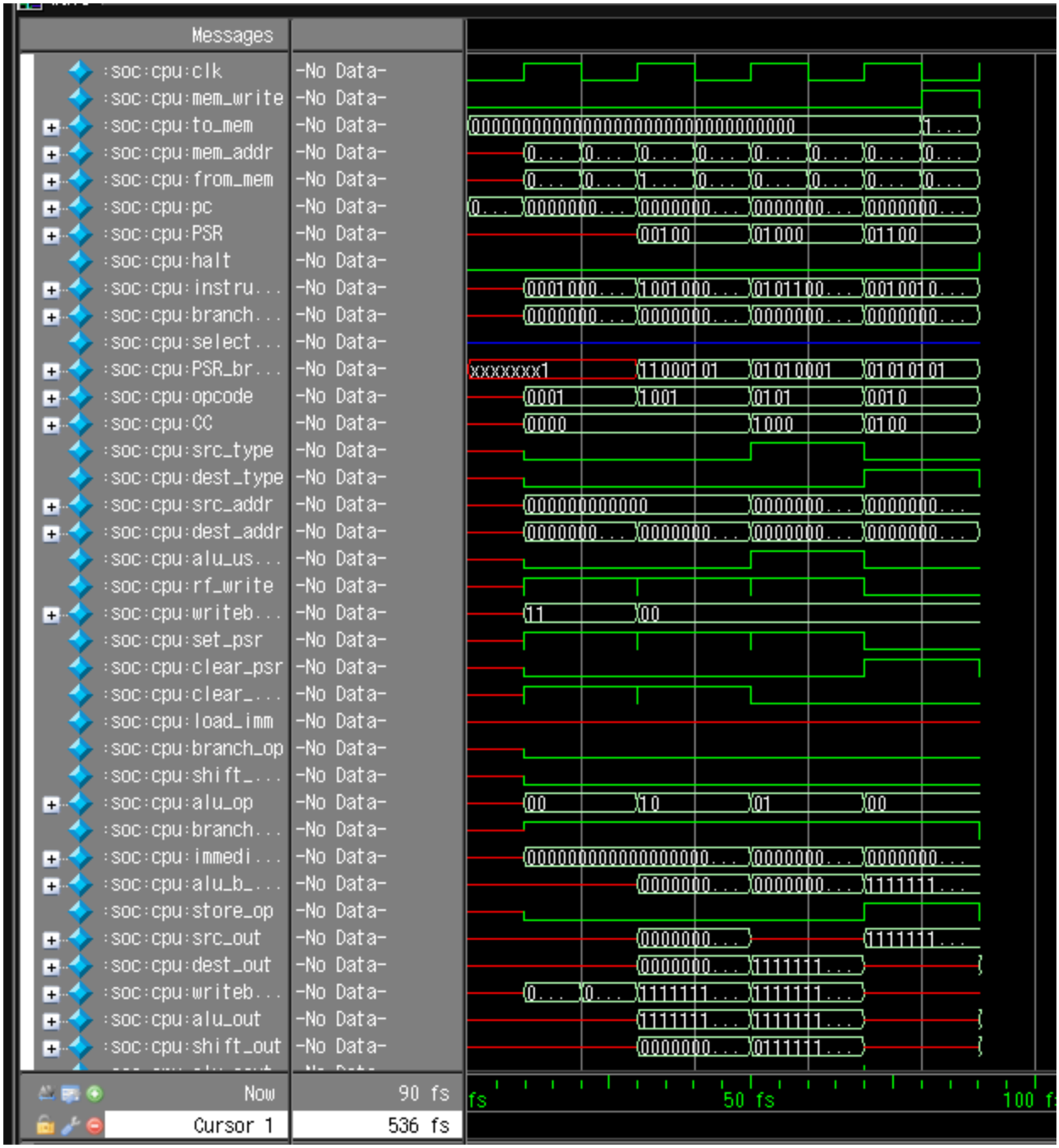
```
always @ (clk) begin
  if(write) begin
    data[addr] <= data_in;
    $display("Wrote %0d to memory location %0d",  $signed(data_in), addr);
  end
end
```

After running the program it can be seen that -6 was written back to memory, which is the correct two's complement of +6.

```
VSIM 24> run -all
# Wrote -6 to memory location 1
# 1
# Break in Module cpu at cpu.v line 359

/SIM 25>
```

Below is an image of memory address 0 and 1, it can be seen that the correct values are present.


Below is the waveform of the program running.


Messages
:soc:cpu:clk -No Data-
:soc:cpu:mem_write -No Data-
:soc:cpu:to_mem -No Data-
:soc:cpu:mem_addr -No Data-
:soc:cpu:from_mem -No Data-
:soc:cpu:pc -No Data-
:soc:cpu:PSR -No Data-
:soc:cpu:halt -No Data-
:soc:cpu:instru... -No Data-
:soc:cpu:branch... -No Data-
:soc:cpu:select... -No Data-
:soc:cpu:PSR_br... -No Data-
:soc:cpu:opcode -No Data-
:soc:cpu:CC -No Data-
:soc:cpu:src_type -No Data-
:soc:cpu:dest_type -No Data-
:soc:cpu:src_addr -No Data-
:soc:cpu:dest_addr -No Data-
:soc:cpu:alu_us... -No Data-
:soc:cpu:rf_write -No Data-
:soc:cpu:writeb... -No Data-
:soc:cpu:set_psr -No Data-
:soc:cpu:clear_psr -No Data-
:soc:cpu:clear_... -No Data-
:soc:cpu:load_imm -No Data-
:soc:cpu:branch_op -No Data-
:soc:cpu:shift_... -No Data-
:soc:cpu:alu_op -No Data-
:soc:cpu:branch... -No Data-
:soc:cpu:immedi... -No Data-
:soc:cpu:alu_b_... -No Data-
:soc:cpu:store_op -No Data-
:soc:cpu:src_out -No Data-
:soc:cpu:dest_out -No Data-
:soc:cpu:writeb... -No Data-
:soc:cpu:alu_out -No Data-
:soc:cpu:shift_out -No Data-
Now
90 fs
Cursor 1
536 fs
fs
50 fs
100 f

**Problem 5:**

Files:

P5.txt

The assembly program for counting the number of ones would be as follows in assembly

```
ld r0,0
ld r1,#0
ld r2,#32
ld r3,#1
cmp r3,r3
add r3,#1
loop:
add r0,#0
bra noadd,E
add r1,#1
noadd:
shf r0,#1
add r2,r3
bra end,Z
bra loop,A
end:
str 1,r1
hlt
```

The program would load the value. Then adding 1 to a counter if the number is odd (lowest bit equal to 1), shifting the number to the right by 1 bit, and finally repeating these operations 31 more times. The binary representation of this program can be seen in *p5.txt*.

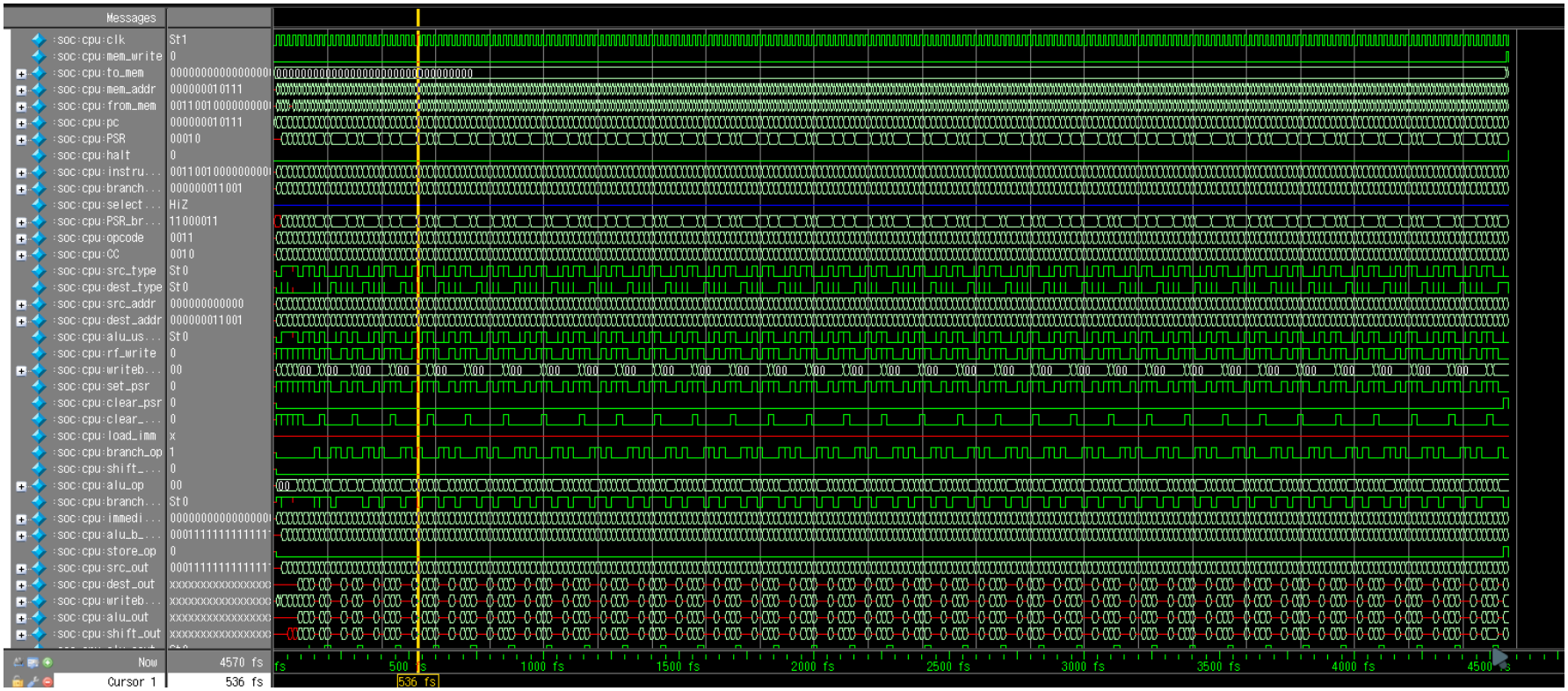
The program is loaded into memory as follows:

```
// uncomment for problem 5
$readmemb("p5.txt", data);
data[0] = {{30{1'b1}}, 2'b0};
```

Since the data in memory location 0 has 30 1s and 2 zeros, the value 30 should be written back to memory location 1. It can be seen in the following screenshots that the observed behavior matches the expected behavior.

```
VSIM 30> run -all
# Wrote 30 to memory location 1
# 1
# Break in Module cpu at cpu.v line 71
```


The wave form can be seen below.


Messages
/soc_cpu/clk St1
/soc_cpu/mem_write 0
/soc_cpu/to_mem 0000000000000000
/soc_cpu/mem_addr 000000010111
/soc_cpu/from_mem 0011001000000000
/soc_cpu/pc 000000010111
/soc_cpu/PSR 00010
/soc_cpu/halt 0
/soc_cpu/instru... 0011001000000000
/soc_cpu/branch... 000000011001
/soc_cpu/select... HiZ
/soc_cpu/PSR_br... 11000011
/soc_cpu/opcode 0011
/soc_cpu/CC 0010
/soc_cpu/src_type St0
/soc_cpu/dest_type St0
/soc_cpu/src_addr 000000000000
/soc_cpu/dest_addr 000000011001
/soc_cpu/alu_us... St0
/soc_cpu/rf_write 0
/soc_cpu/writeb... 00
/soc_cpu/set_psr 0
/soc_cpu/clear_psr 0
/soc_cpu/clear_... 0
/soc_cpu/load_imm x
/soc_cpu/branch_op 1
/soc_cpu/shift_... 0
/soc_cpu/alu_op 00
/soc_cpu/branch... St0
/soc_cpu/immedi... 0000000000000000
/soc_cpu/alu_b_... 0001111111111111
/soc_cpu/store_op 0
/soc_cpu/src_out 0001111111111111
/soc_cpu/dest_out xxxxxxxxxxxxxxxx
/soc_cpu/writeb... xxxxxxxxxxxxxxxx
/soc_cpu/alu_out xxxxxxxxxxxxxxxx
/soc_cpu/shift_out xxxxxxxxxxxxxxxx
Now 4570 fs
Cursor 1 536 fs
fs
500 fs
1000 fs
1500 fs
2000 fs
2500 fs
3000 fs
3500 fs
4000 fs
4500 fs
536 fs

**Problem 6:**

Files:

P6.txt

The 4 bit multiplication program is as follows. See *p6.txt* for the binary representation.

```
ld r0,0
ld r1,1
ld r2,#0
ld r3,#4
ld r4,#1
cmp r4,r4
add r4,#1
loop:
add r0,#0
bra noadd,E
add r2,r1
noadd:
shf r0,#1
shf r2,#-1
add r3,r4
bra end,Z
bra loop,A
end:
str 2,r2
hlt
```

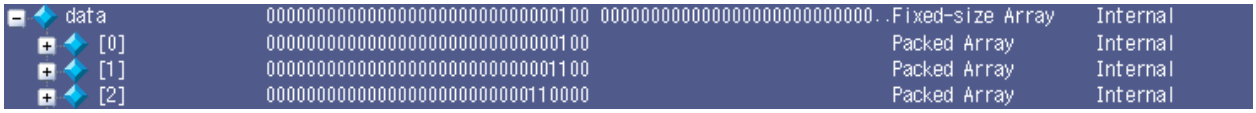
The program is loaded as follows:

```
$readmemb("p6.txt",data);
data[0] = 32'd4;
data[1] = 32'd12;
```

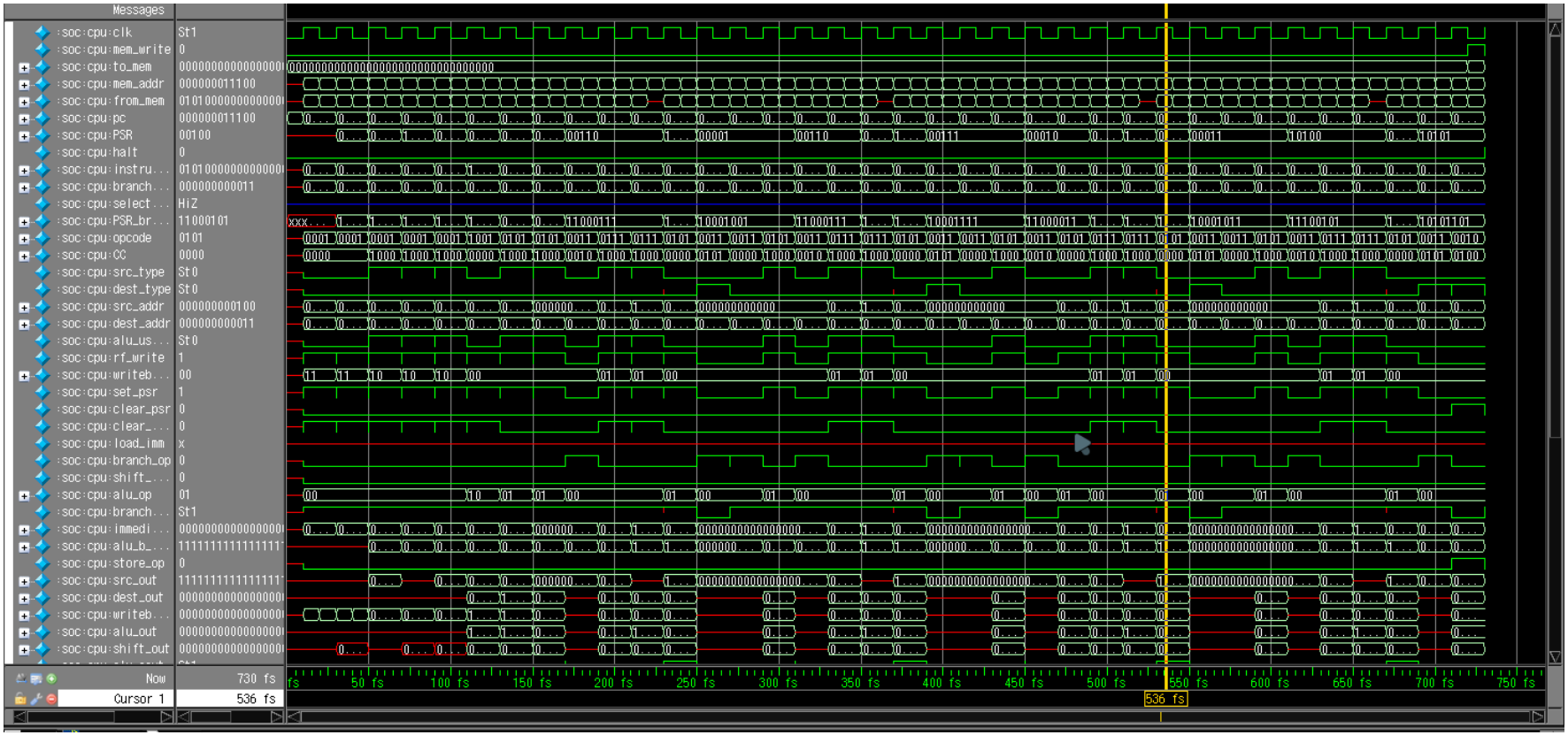
The values of 4 and 12 were chosen to test the algorithm, thus we are expecting an output value of 48. It can be seen in the screenshots of the output below that the expected value is indeed output by the program.

```
VSIM 33> run -all
# Wrote 48 to memory location 2
# 1
# Break in Module cpu at cpu.v line 71
```

Memory addresses 0-2 can be seen below with values 4, 12, and 48 respectively.




The waveforms of this program can be found below:


Messages
:soc:cpu:clk St1
:soc:cpu:mem_write 0
:soc:cpu:to_mem 00000000000000000000000000000000
:soc:cpu:mem_addr 00000011100
:soc:cpu:from_mem 01010000000000000
:soc:cpu:pc 00000011100
:soc:cpu:PSR 00100
:soc:cpu:halt 0
:soc:cpu:instru... 01010000000000000
:soc:cpu:branch... 00000000011
:soc:cpu:select... HiZ
:soc:cpu:PSR_br... 11000101
:soc:cpu:opcode 0101
:soc:cpu:CC 0000
:soc:cpu:src_type St0
:soc:cpu:dest_type St0
:soc:cpu:src_addr 00000000100
:soc:cpu:dest_addr 00000000011
:soc:cpu:alu_us... St0
:soc:cpu:rf_write 1
:soc:cpu:writeb... 00
:soc:cpu:set_psr 1
:soc:cpu:clear_psr 0
:soc:cpu:clear_... 0
:soc:cpu:load_imm x
:soc:cpu:branch_op 0
:soc:cpu:shift_... 0
:soc:cpu:alu_op 01
:soc:cpu:branch... St1
:soc:cpu:immedi... 00000000000000000
:soc:cpu:alu_b_... 11111111111111111
:soc:cpu:store_op 0
:soc:cpu:src_out 11111111111111111
:soc:cpu:dest_out 00000000000000000
:soc:cpu:writeb... 00000000000000000
:soc:cpu:alu_out 00000000000000000
:soc:cpu:shift_out 00000000000000000
Now 730 fs
Cursor 1 536 fs
fs 50 fs 100 fs 150 fs 200 fs 250 fs 300 fs 350 fs 400 fs 450 fs 500 fs 550 fs 600 fs 650 fs 700 fs 750 fs
536 fs