Organization of Digital Computer Lab
EECS112L/CSE 132L

Assignment 2
Single-cycle MIPS Datapath and Control

prepared by: Team CART

Student name	**Student ID**
Raquel Fallman	26316814
Arash Nabili	37684183
Christine Srun	15386050
Tyler Stevens	40051117

EECS Department
Henry Samueli School of Engineering
University of California, Irvine

January, 24, 2016

Contents

List of Figures

1 Introduction

For this lab, we built a simple single-cycle MIPS processor using VHDL for the design and SystemVerilog for the testbenches. Single-cycle means that an entire instruction will execute in just one cycle. For the instruction cycle, the processor first fetches and reads an instruction from the instruction memory. The instruction will be decoded and then executed. Depending on the instruction, it will load or store to/from the memory. Finally, the result of the execution will be written back to the register file and the process repeats.

2 Processor

This processor implementation is divided into two major units: the controller and datapath. Our approach to this design was to break down each of the units into basic functional blocks. We tested each block with its own testbench to verify that the design was correct and working properly. After testing each of the blocks, we connected the blocks to create a unit. In this case, the controller is a standalone unit and the datapath is comprised of many various blocks. There are two blocks that sit outside of the controller and datapath, the program counter and instruction memory.

In our processor design, the program counter, instruction memory, controller, and datapath are instantiated and linked up. The processor sends a clock signal to the program counter and datapath to begin instruction. The clock signal defines when a signal can be read or written. The reset signal is used to bring the components to a initial/normal state. The processor ports are shown below.

Processor Port Description			
Port name	Port size	Port Type	Description
ref_clk	1	IN	clock signal
reset	1	IN	reset to normal state

CPU operation types		
op	Operation Type	Comments
000000	R-type instruction	The funct field will be sent to the ALU
001XXX	I-type instruction	The controller will determine the ALU function code
100011	Load Word	ALU operation will be addi
101011	Store Word	ALU operation will be addi
Others	nop	ALU function code will be set to nop

2.1 Design Schematic

This is our design schematic of the processor. The program counter passes an address to the instruction memory and gets incremented to go to the next address. Usually, an adder would increment the program counter, but as shown in our design schematic, we did not use an adder block in our case. The program counter is implicitly incremented. However, we did create an adder block and will probably use it in upcoming labs. After the program counter passes the address into the instruction memory, the instruction memory fetches the instruction and sends it into the datapath. The wires on the controller are labeled to show where they are connected in the diagram.

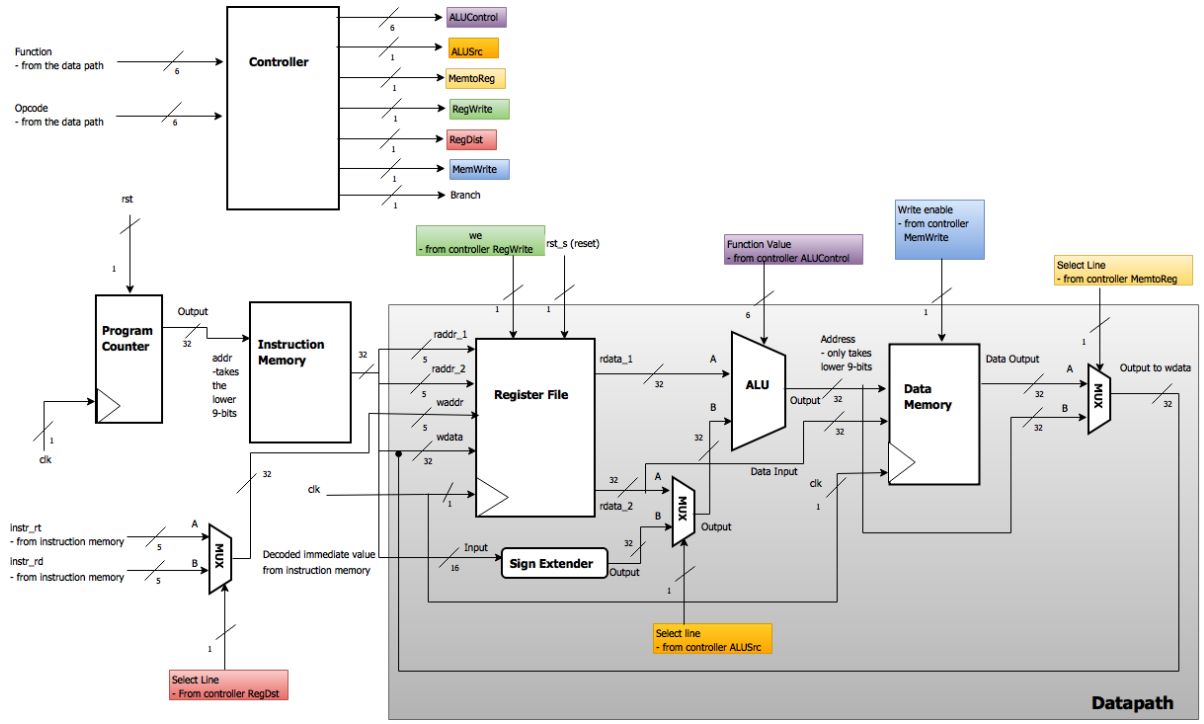

Figure 1: Design Schematic of the Processor: Note that the adder is not included as the program counter is implicity incremented.

2.2 Simulation

To simulate the processor, we created a testbench and an input file "Lab2Program.txt" using MIPS code to verify the design. This tests I-type and R-type instructions. For I-type instructions, the following operations were tested: addi, slti, andi, ori, xori, lw, and sw. For R-type instructions, the following operations were tested: add, sub, and, nor, xor, or, and slt. We had previously tested the smaller components to verify that their design was correct and properly working. Testbenches were created for the instruction memory, controller, ALU, multiplexer, register file, data memory, and sign extension unit.

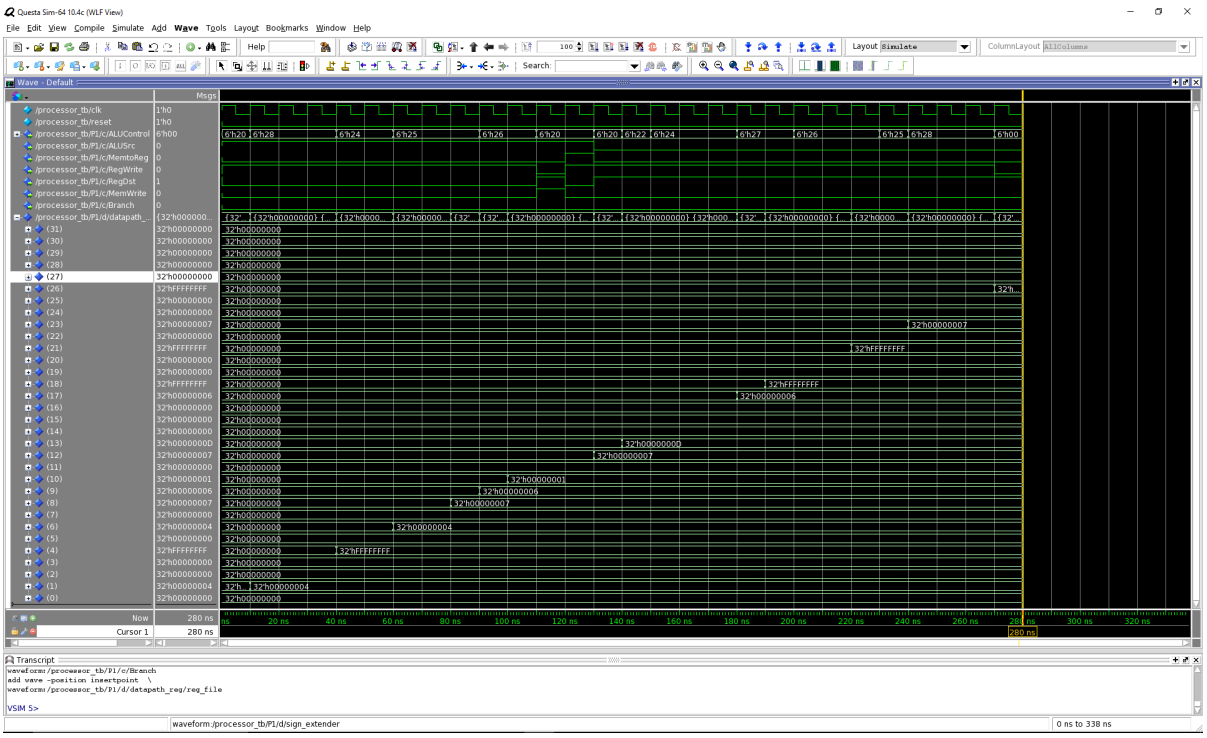

Figure 2: Processor Waveform: Verification that processor was working properly.

2.3 Program Counter

The program counter is a register that contains the addresses of the instructions to be executed. An address gets fetched from the program counter and passed to the instruction memory. After the instruction is fetched, the counter gets incremented to go to the next address. Our implementation of the program counter, takes in a clock signal and reset. It outputs a 32-bit address.

Program Counter Port Description			
Port name	Port size	Port Type	Description
clk	1	IN	clock signal
rst	1	IN	reset bit
output	32	OUT	outputs an address

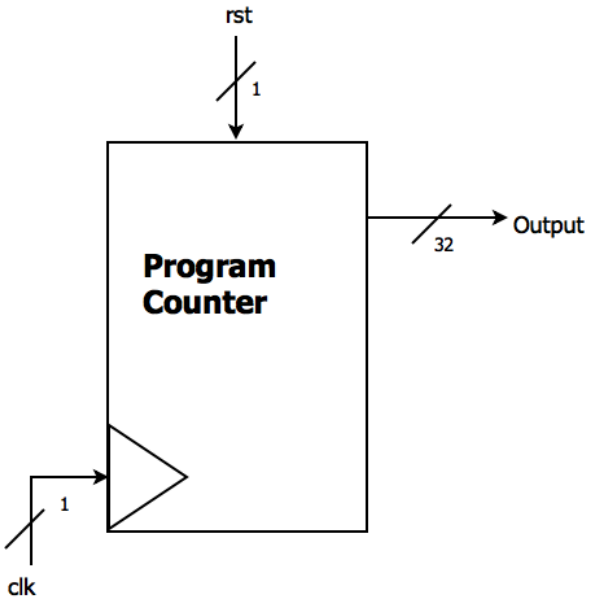

Figure 3: Program Counter Diagram

2.4 Instruction Memory

The instruction memory for our design is implemented as a ROM (read-only memory). It is preloaded with instructions provided in the rom.dat file. The address is sent from the program counter as 32-bits, but the instruction memory will only take the lower 9-bits. This makes our instruction memory size $2^9 - 1$ and each line is 32-bits long. Initially, we set the size to $2^{32} - 1$. However, we encountered errors where QuestaSim and Cadence would not allow an array size greater than $2^{30} - 1$. Since we were not going to use that many registers, we shrunk the size.

Instruction Memory Port Description			
Port name	Port size	Port Type	Description
addr	9	IN	address for the location of instruction
dataIO	32	INOUT	outputs an instruction

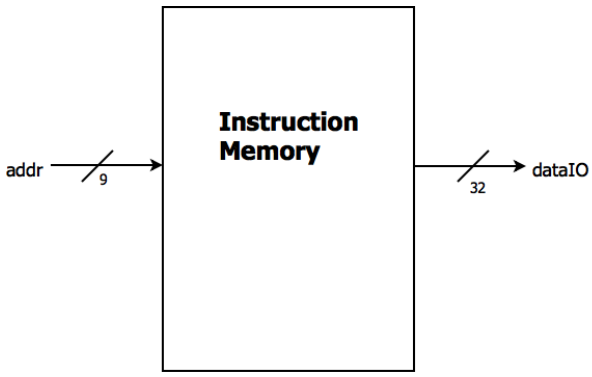

Figure 4: The instruction memory has one 9-bit input and one 32-bit output.

The instruction memory is preloaded with instructions and takes an address input. The following waveforms loads the instructions into the corresponding addresses.

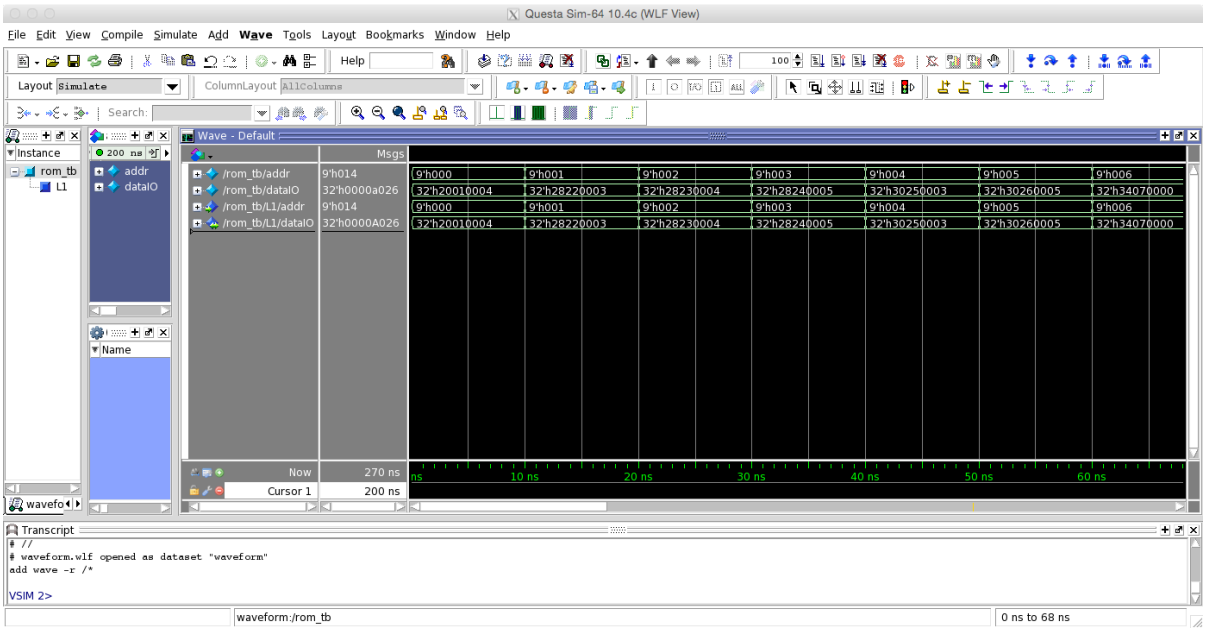

Figure 5: ROM Waveform: 0ns to 70ns

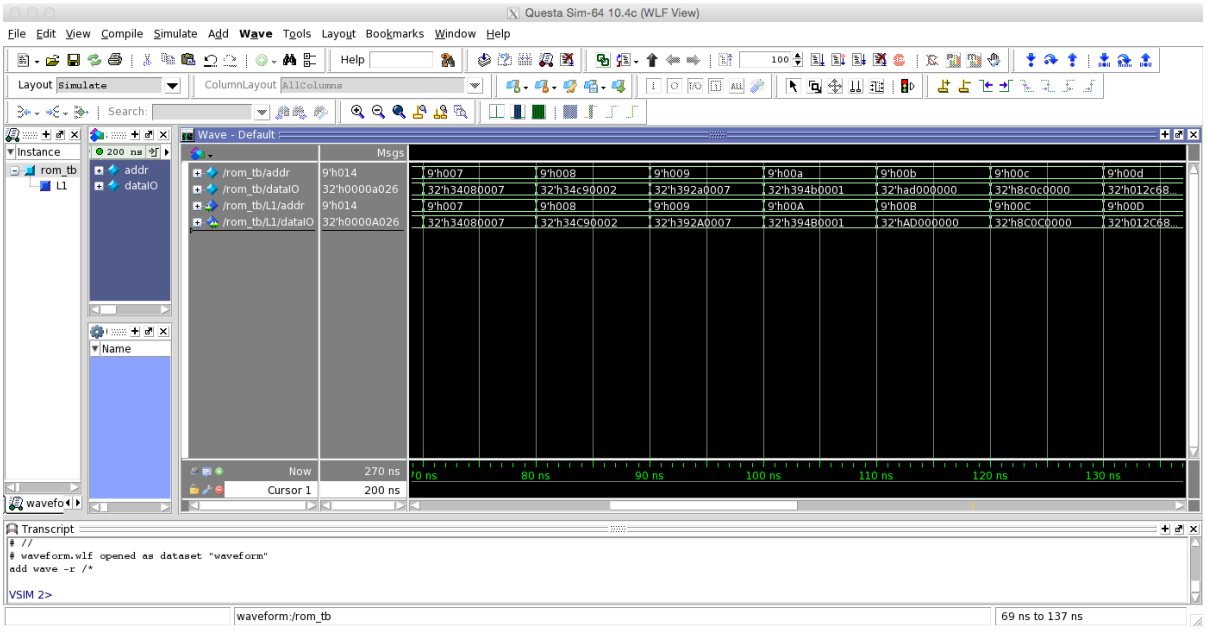

Figure 6: ROM Waveform: 70ns to 140ns

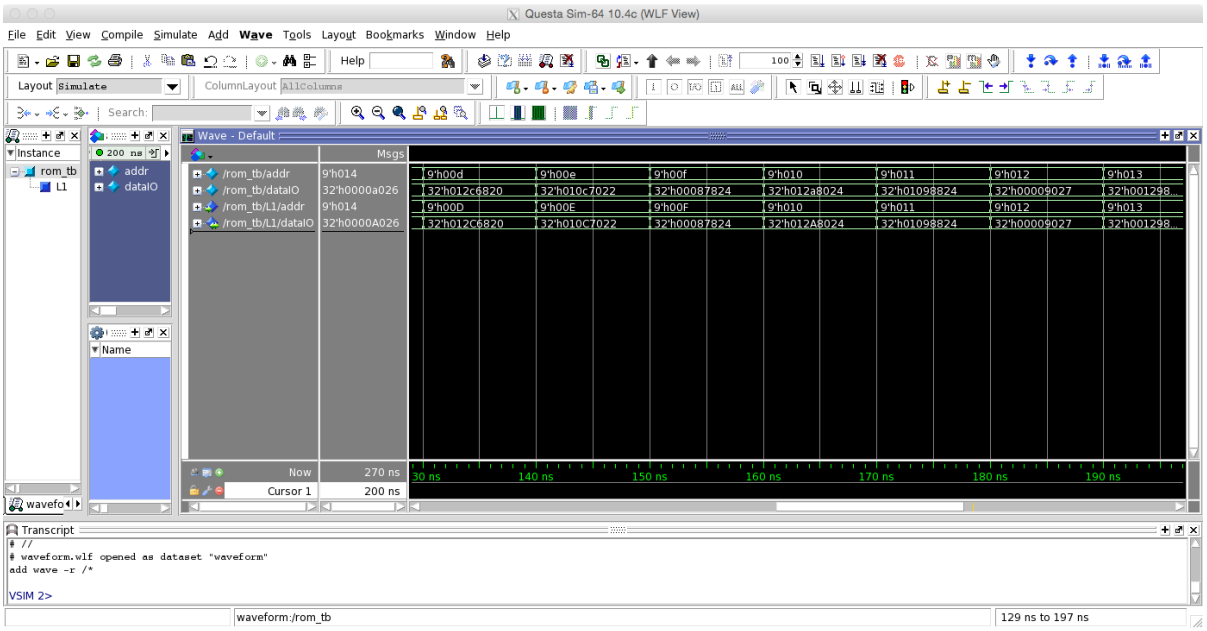

Figure 7: ROM Waveform: 130ns to 200ns

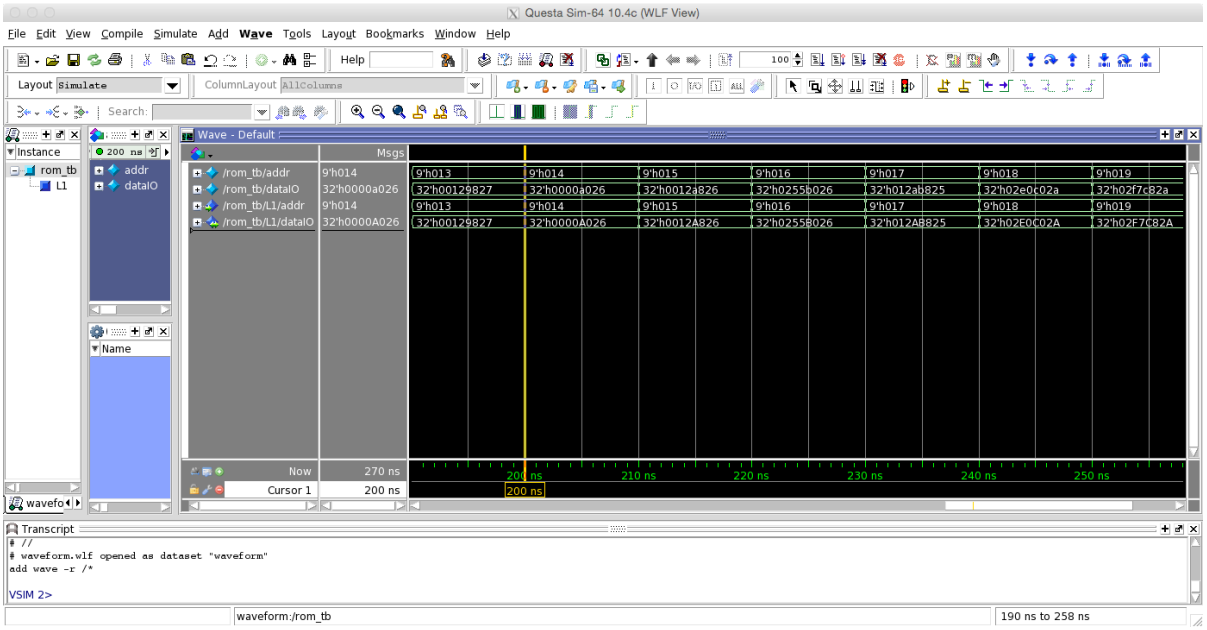

Figure 8: ROM Waveform: 190ns to 260ns

2.5 Adder

We did not use the adder in our design, but we had already created one. It takes two 32-bit inputs and adds them together and gets a 32-bit sum. If there is an overflow, the most significant bit is lost.

Adder Port Description			
Port name	Port size	Port Type	Description
a	32	IN	addend a
b	32	IN	addend b
c	32	OUT	sum

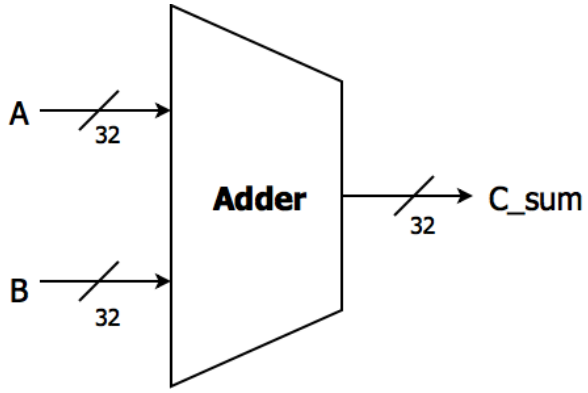

Figure 9: A diagram of an adder with two 32-bit inputs and one 32-bit output.

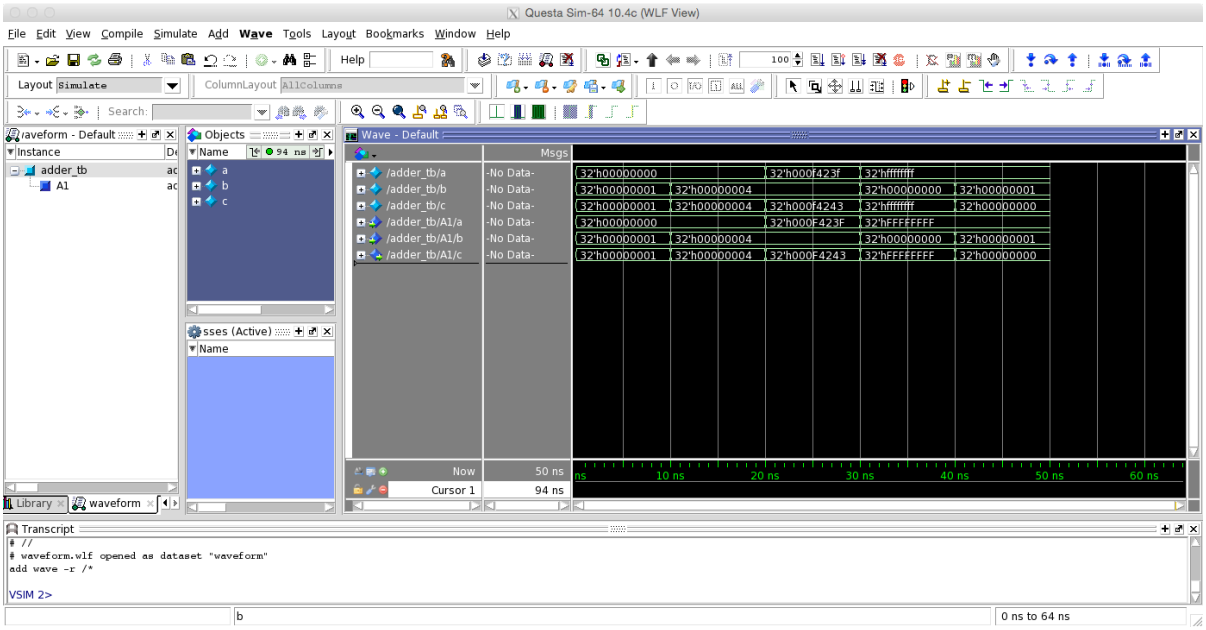

Figure 10: Adder waveform generated from a testbench to verify the design.

3 Controller

The controller is one of the most important blocks in ensuring that the processor works correctly, as it is the part of the processor that decides whether the register file or the data memory should be written to or not. To ensure that data does not accidentally get overwritten, the controller must be very strict in the conditions under which it allows the register file or the data memory to be written. For example, if the instruction is not an R-type, I-type, or load instruction, the register file's write enable line should be set to 0, and if the instruction is not a store instruction, the data memory's write enable line should be set to 0. The correct selection for the multiplexers is also important, as it could allow the incorrect

value to be written to the register file or data memory. It also determines whether to enable the write function of the data memory and the register file, and sends the correct function code to the ALU, based on the instruction.

In our design, the controller determines the correct select value for three multiplexers using the op-code and funct field of the instruction. First, the multiplexer that selects the address for the second register in the register file (Rt for I-type and load instructions; Rd for all other instructions). Second, the multiplexer that chooses the second operand for the ALU (output of register files second read channel for R-type instructions; sign-extended immediate value for all other instructions). Third, the multiplexer that chooses the data to write to the register file (output of data memory for load instructions; output of ALU for all other instructions).

Controller Port Description			
Port name	Port size	Port Type	Description
Funct	6	IN	function
op	6	IN	opcode
ALUControl	6	OUT	goes to the ALU
ALUSrc	1	OUT	goes to the ALU
MemtoReg	1	OUT	memory to the register
RegWrite	1	OUT	controls register to write
RegDis	1	OUT	register
MemWrite	1	OUT	write memory
Branch	1	OUT	controls branch instruction

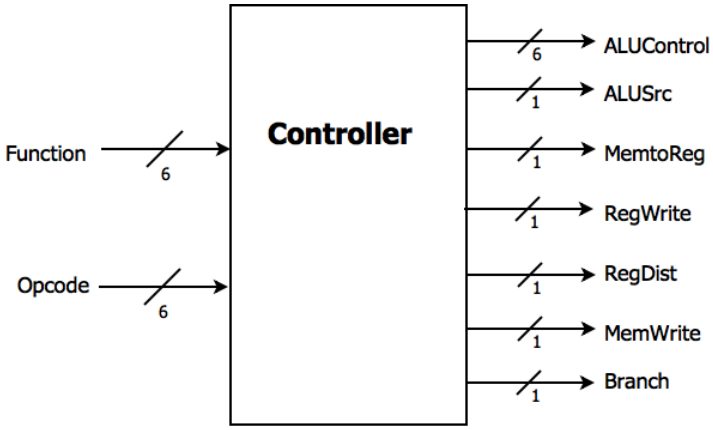

Figure 11: Diagram of the controller

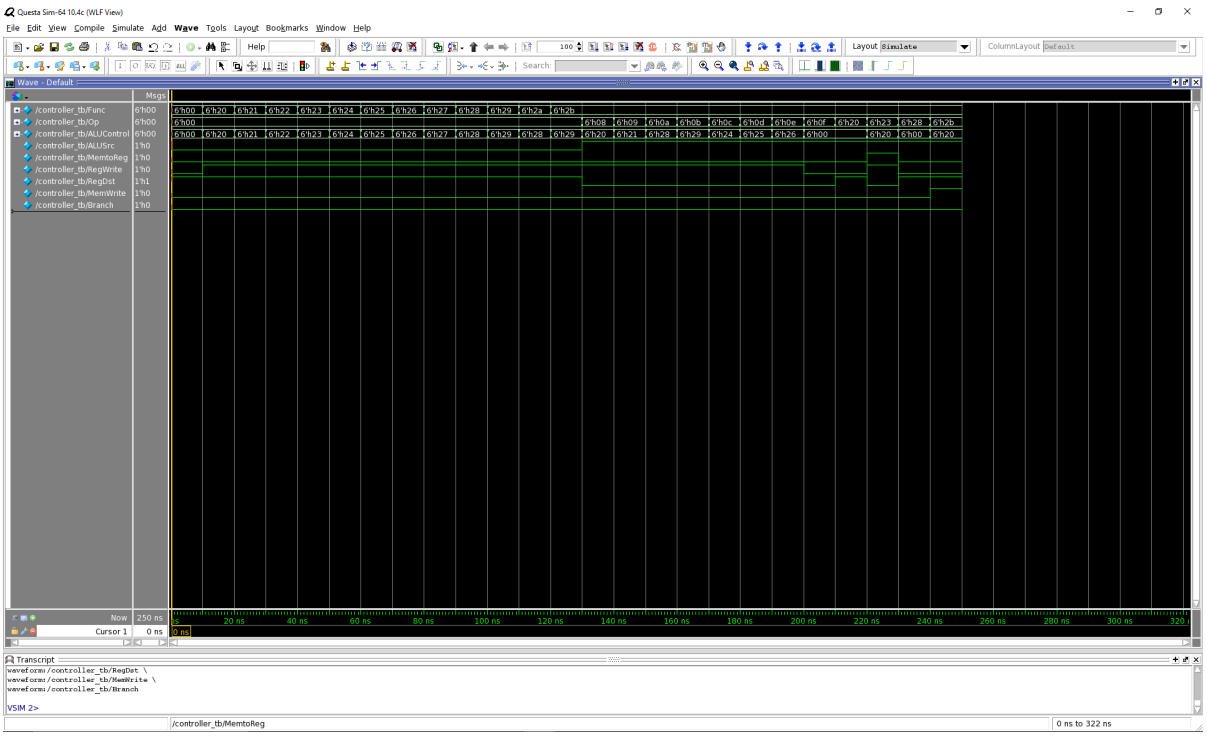

Figure 12: Controller Waveform verified through a SystemVerilog testbench.

4 Datapath

The datapath houses the register file, ALU, and data memory, as well as the multiplexers that choose the address of the register file's second read port, the ALU's second operand, and the data to be written to the register file. It accepts the current instruction, and performs bit slicing to set the correct values for the following: the address of the the register file's first read port, the addresses connected to the multiplexer for the register file's second read port, and the input of the sign extension unit. The datapath also sends the opcode and funct fields of the instruction to the controller, and receives the output of the controller.

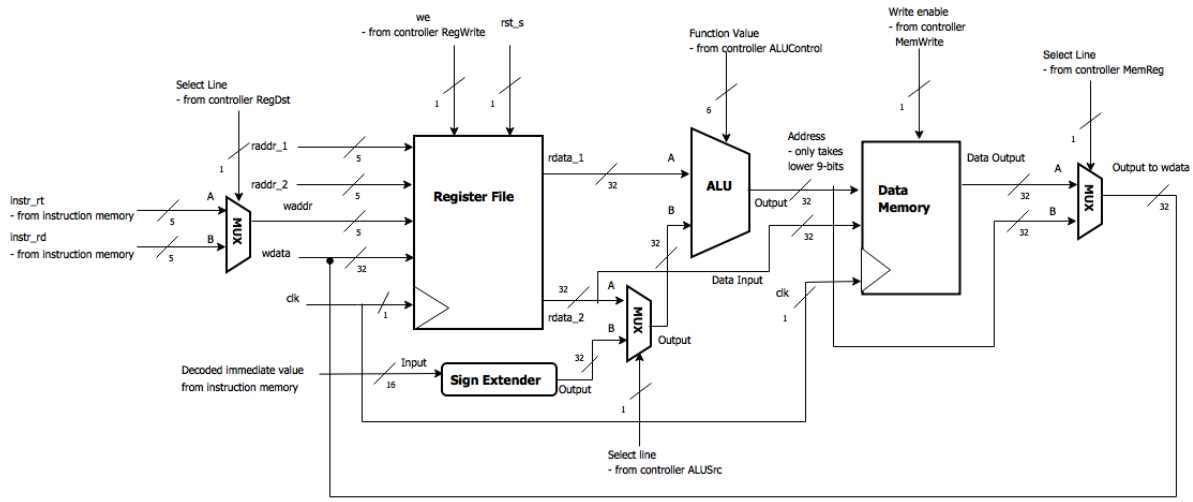

Figure 13: Datapath with ALU, 3 multiplexers, RAM, register file, and sign extension unit.

4.1 ALU

ALU Port Description			
Port name	Port size	Port Type	Description
Func_in	6	IN	opcode
A_in	32	IN	operand A
B_in	32	IN	operand B
O_out	32	OUT	output from ALU
Branch_out	1	OUT	set to 0, not used
Jump_out	1	OUT	set to 0, not used

ALU Function Description			
Instruction	Description	Function Code	Comments
nop	nothing	"000000"	
add/addi	rd ← rs + rt / rd ← rs + immediate	"100000"	
addu/addiu	rd ← rs + rt / rd ← rs + immediate	"100001"	
sub	rd ← rs - rt / rd ← rs - immediate	"100010"	
subu	rd ← rs - rt / rd ← rs - immediate	"100011"	
and/andi	rd ← rs AND rt / rd ← rs AND immediate	"100100"	
or/ori	rd ← rs OR rt / rd ← rs OR immediate	"100101"	
xor/xori	rd ← rs XOR rt / rd ← rs XOR immediate	"100110"	
nor	rd ← rs XOR rt / rd ← rs NOR immediate	"100111"	
slt/slti	Set rd if rs < rt / set rd if rs < immediate	"101000"	If the condition is satisfied set else reset destination
sltu/sltiu	Set rd if rs < rt / set rd if rs < immediate	"101001"	If the condition is satisfied set else reset destination
nop	nothing	others	Any other function code does nothing

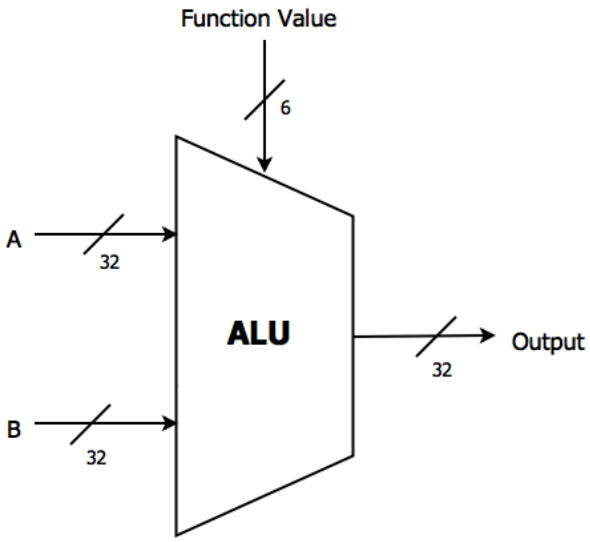

Figure 14: The ALU has 3 inputs and one output. There are two 32-bit inputs that act as the operands and a 6-bit input that lets the ALU know which operation to perform. The result is then sent to the output.

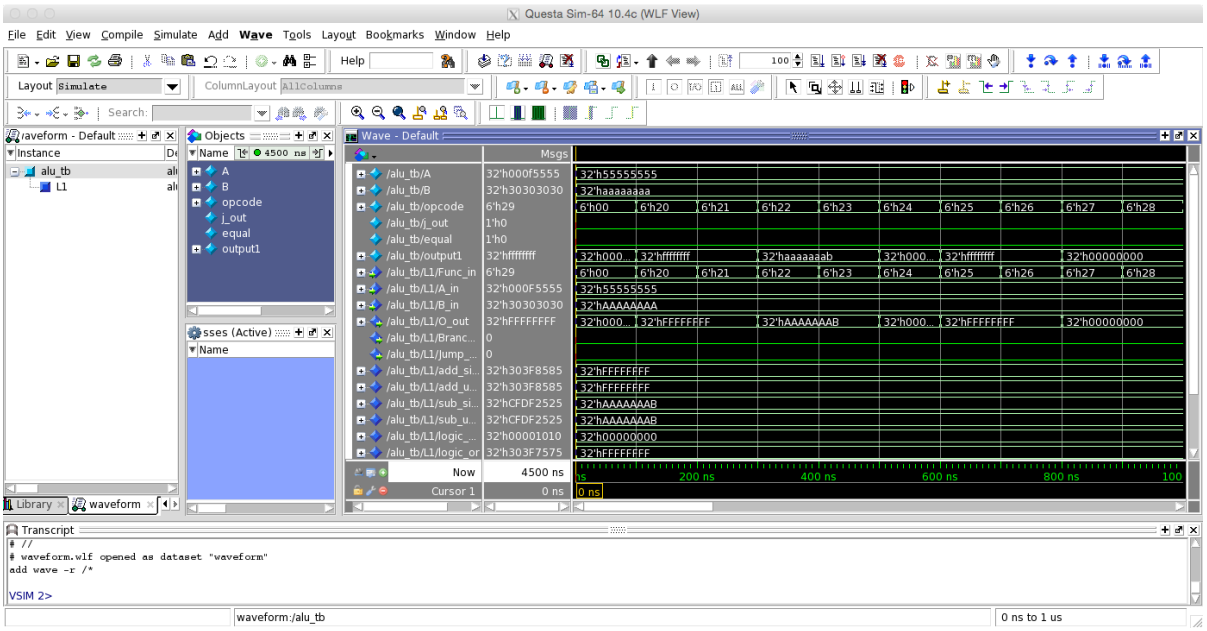

Figure 15: Operations shown in the waveform: ADD, SUB, AND, OR, XOR, NOR, SLTS, SLTU

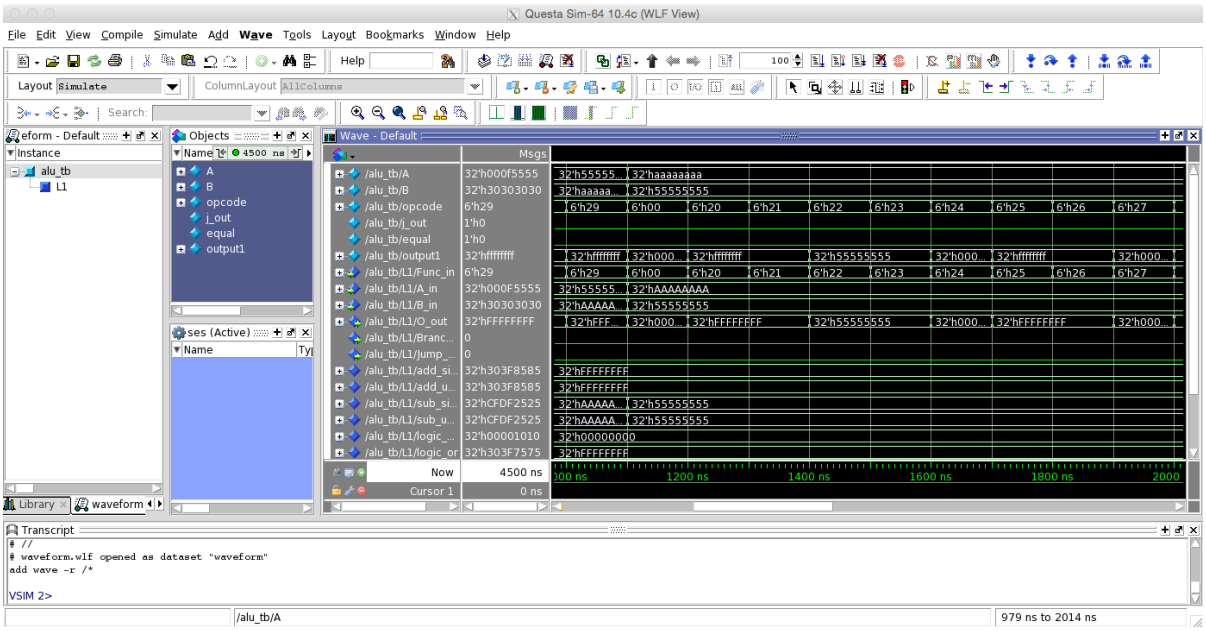

Figure 16: Operations shown in the waveform: ADD, SUB, AND, OR, XOR, NOR, SLTS, SLTU

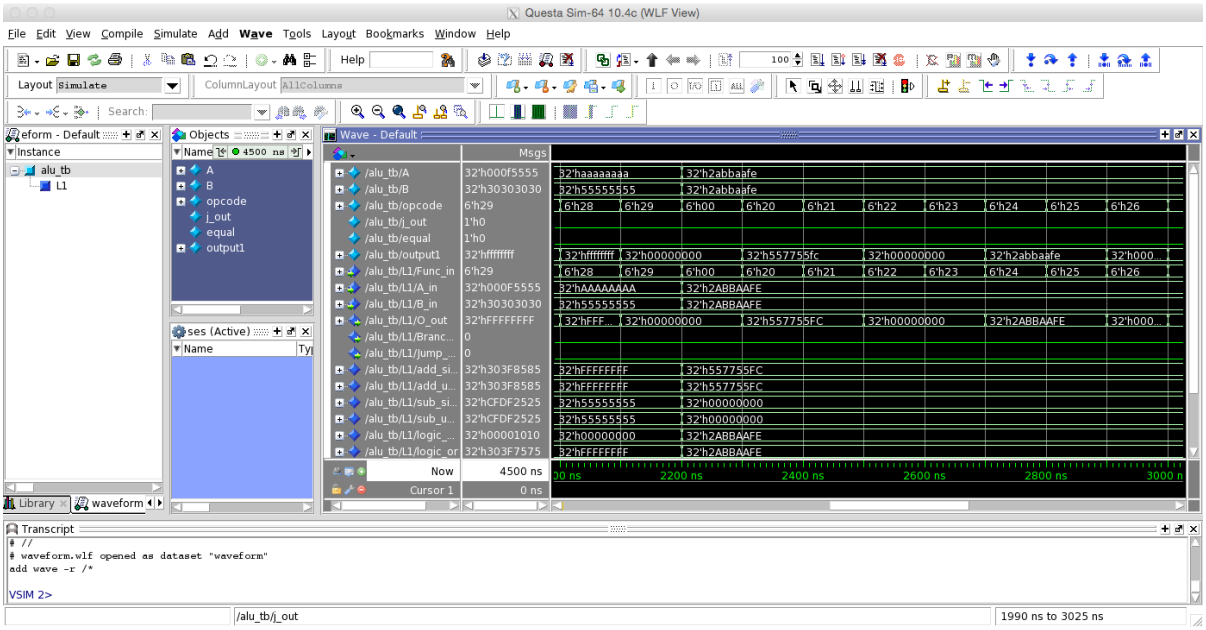

Figure 17: Operations shown in the waveform: ADD, SUB, AND, OR, XOR, NOR, SLTS, SLTU

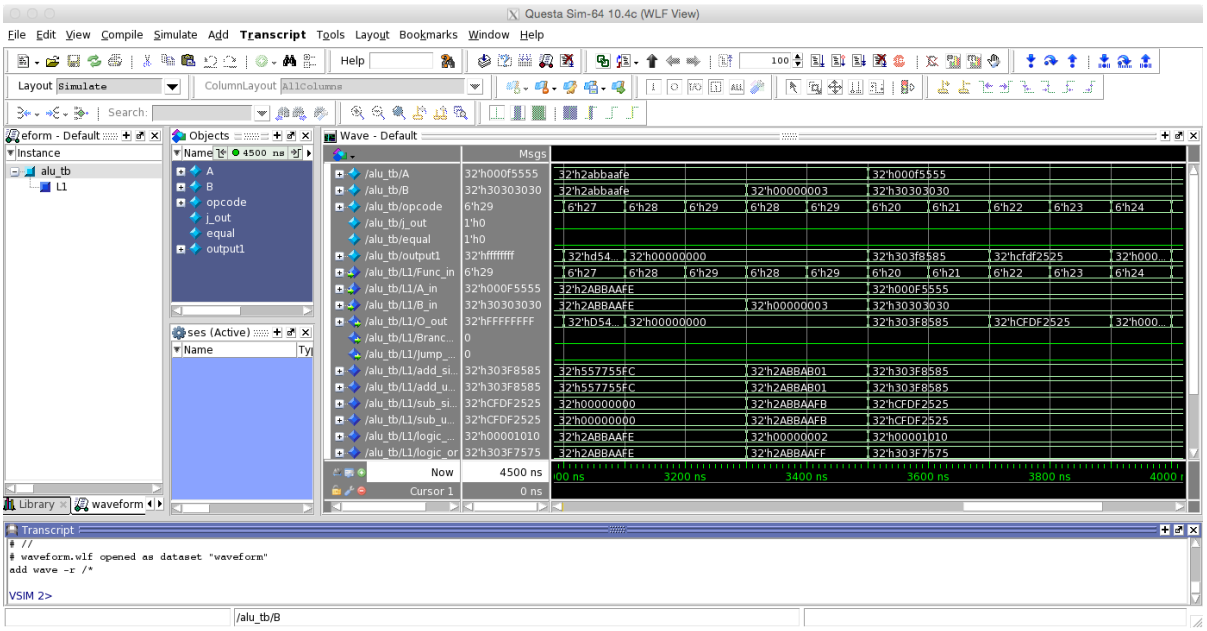

Figure 18: Operations shown in the waveform: OR, XOR, NOR, SLTS, SLTU

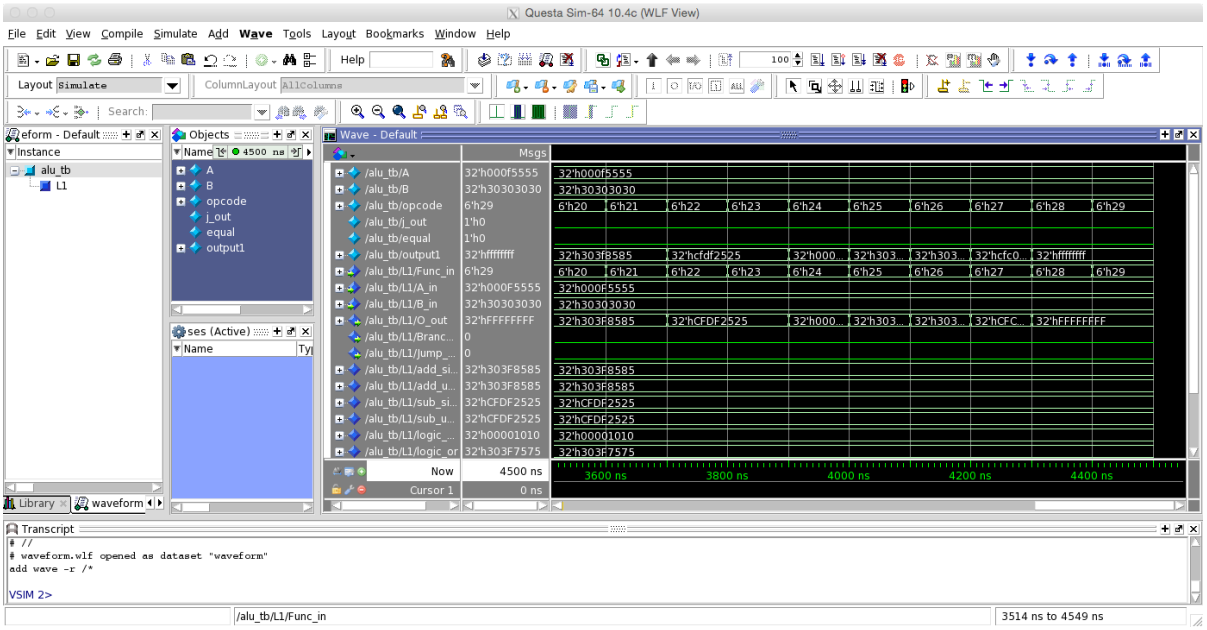

Figure 19: Operations shown in the waveform:

4.2 Multiplexer

In general, the multiplexer chooses an output from several possible inputs based on the value of the select signal. Our design uses three 2:1 multiplexers whose select signal inputs come from the controller block: ci_sel_dt, ci_sel_ri, and ci_sel_am. The ci_sel_dt chooses whether to use rt for R-type instructions or rd for the lw instruction. The ci_sel_ri chooses from either the register file input or sign immediate input as the output for source B. The ci_sel_am chooses between the alu result input for R-type instructions or read data input from data memory for the lw instruction.

Multiplexer Port Description			
Port name	Port size	Port Type	Description
s	1	IN	select line
a	32	IN	data 1
b	32	IN	data 2
o	32	OUT	selected data

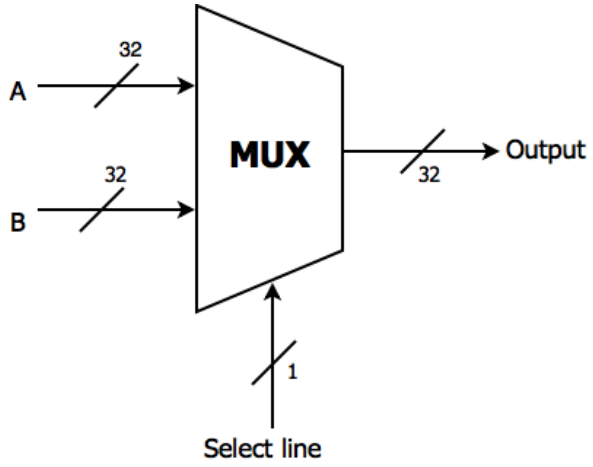

Figure 20: 2-to-1 MUX

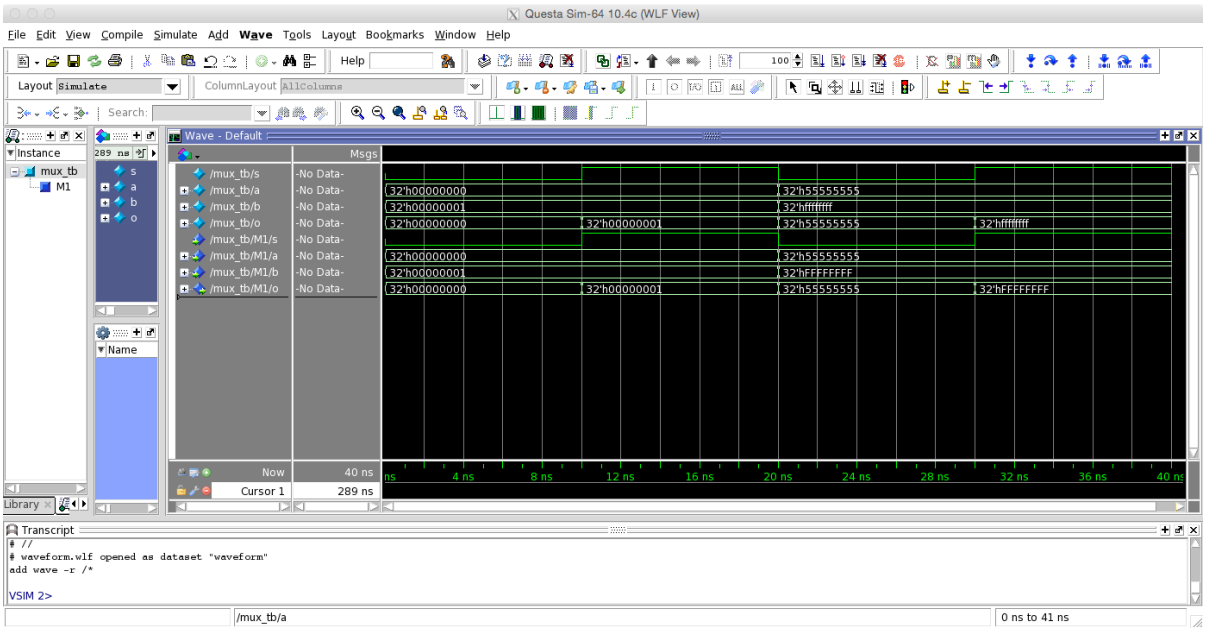

Figure 21: The waveform tests select line for 0 and 1.

4.3 RAM

The data memory is 512 lines with one word per line. It has a single read/write port. On the rising edge of the clock, if write enable is 1, it writes data into the input address. If the write enable is 0, it reads the data from the input address.

RAM Port Description			
Port name	Port size	Port Type	Description
clk	1	IN	clock signal
we	1	IN	write enable
addr	9	IN	address
dataI	32	IN	input data
dataO	32	OUT	output data

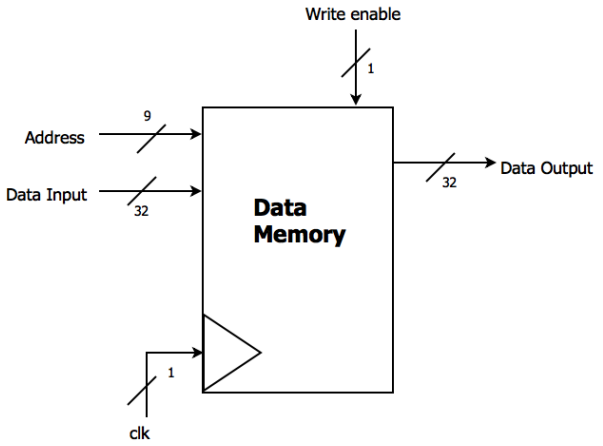

Figure 22: Random Access Memory

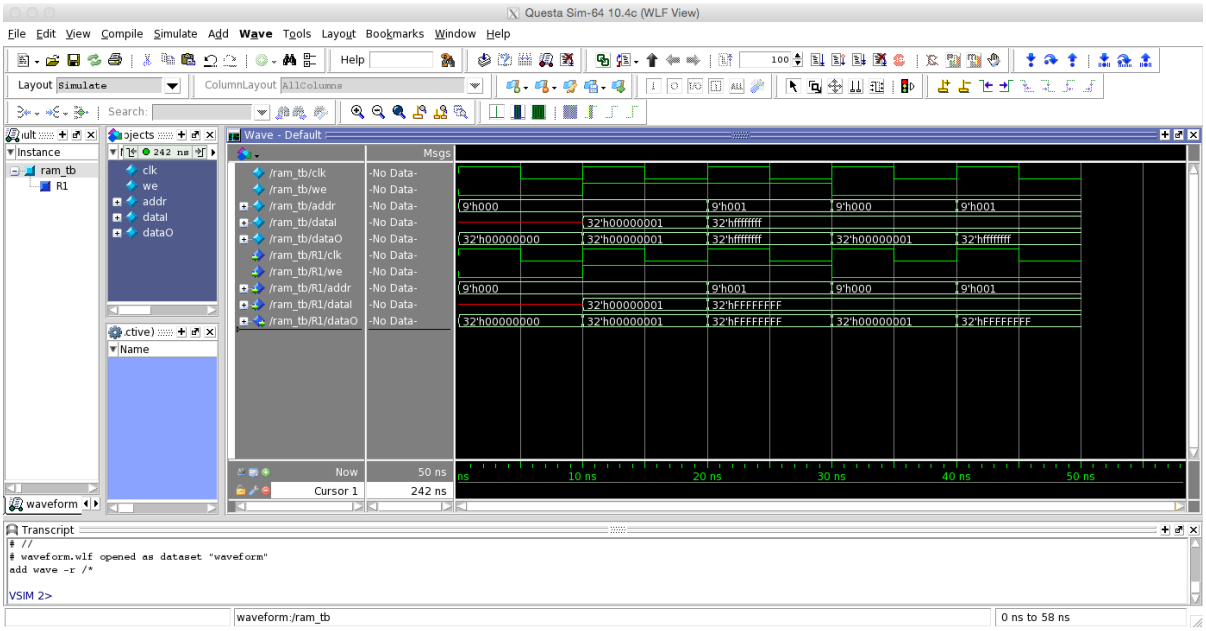

Figure 23: Waveform verifies the RAM design.

4.4 Register File

The register file has 32 registers, each 32 bits wide. There are 2 read ports and one write port. The read ports are asynchronous and the write port is synchronous. The register file has a synchronous reset signal and a write enable signal.

Register Port Description			
Port name	Port size	Port Type	Description
clk	1	IN	clock signal
rst_s	1	IN	synchronous reset
we	1	IN	write enable
raddr_1	5	IN	read address 1
raddr_2	5	IN	read address 2
waddr	5	IN	write address
rdata_1	32	OUT	read data 1
rdata_2	32	OUT	read data 2
wdata	32	IN	write data

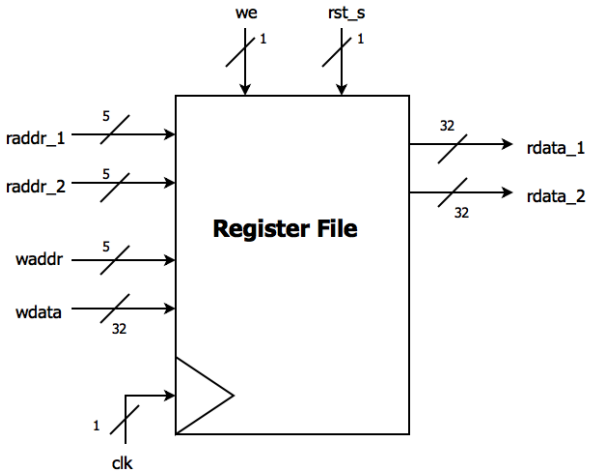

Figure 24: Register file

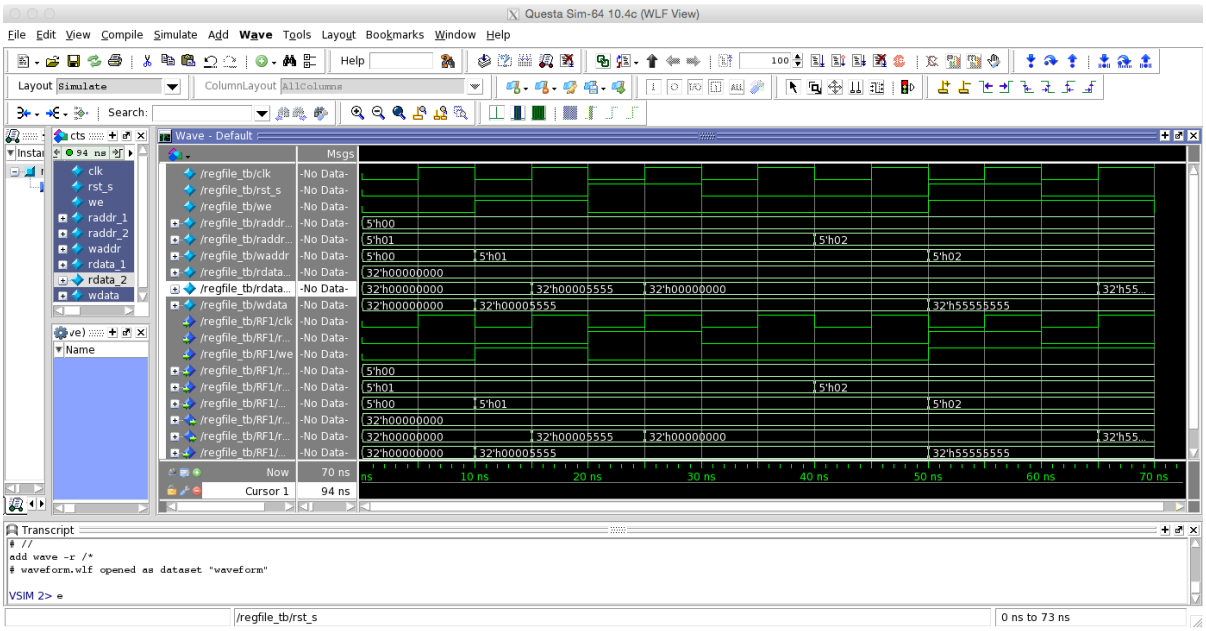

Figure 25: Waveform generated to verify register file design.

4.5 Sign Extension Unit

The sign extension unit performs sign extension on the 16-bit immediate value to make it 32-bits wide, so that it can be used in as a second operand in the ALU.

Sign Extender Port Description			
Port name	Port size	Port Type	Description
input	16	IN	input value
output	32	OUT	extended value to 32-bits

Figure 26: Takes in a 16-bit value and extends it to 32-bits.

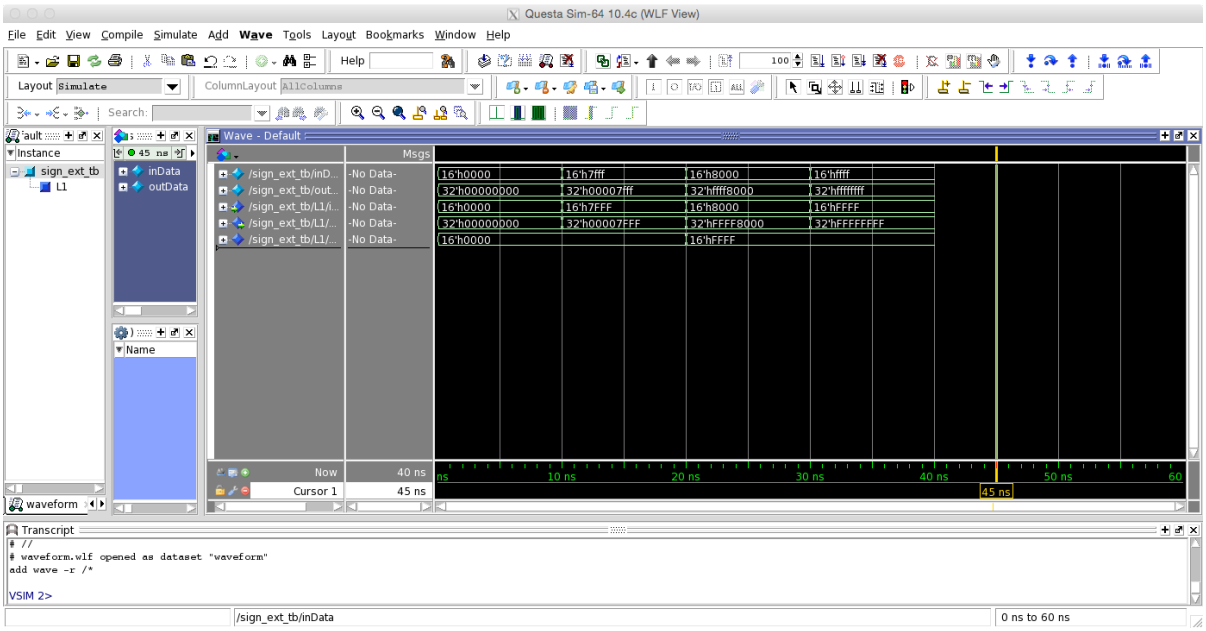

Figure 27: Waveform for sign extension unit shows values getting extended.

5 Conclusion

When designing a processor, we discovered how difficult it is to combine the components together in such a way that they are able to work together even though each component was individually tested and worked well on its own. We encountered problems when linking up the components. For example, the data memory had a delay of one cycle when loading and storing. To fix the problem, we made the read port asynchronous. After completing this lab, we have a better understanding of how a basic processor works. We also gained more insight to each of the functional blocks that make up the processor.