Organization of Digital Computer Lab
EECS112L/CSE 132L

Final Project
Pipeline MIPS

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

March 13, 2016

Contents

1 Introduction **3**

2 Processor **3**
2.1 Design Schematic 4
2.2 Simulation 4
2.2.1 Waveform: Using our own MIPS test code. 5

3 The Pipeline Architecture **11**
3.1 Pipelining 11
3.2 Forwarding 11
3.3 Hazard Detection 11

4 Components From Previous Labs **12**
4.1 Program Counter 12
4.2 Instruction Memory 13
4.3 Controller 13
4.4 ALU 14
4.5 Register File 15
4.6 Multiplexer 16
4.7 Sign Extension Unit 17
4.8 Adder 17
4.9 Shifter 18
4.10 RAM 18

5 Synthesis **19**

6 Conclusion **20**

List of Figures

1 Processor Design Schematic 4
2 Program Counter Diagram 13
3 Instruction Memory Diagram 13
4 Controller Diagram 14
5 ALU Diagram 15
6 Register File Diagram 16
7 MUX Diagram 17
8 Sign Extender Diagram 17
9 Adder Diagram 18
10 Shifter Diagram 18
11 RAM Diagram 19

1 Introduction

For this lab, we modified our single-cycle MIPS processor from Lab 3 into a five-stage pipeline processor. In a pipelined processor, the instruction cycle is broken up into stages. Each stage executes a different part of the instruction, which allows the stages to operate concurrently. This means that multiple instructions can be processed in parallel. The stages are instruction fetch, instruction decode, execute, read/write data memory, and write back.

The advantages of a pipeline processor over a non-pipelined processor is that it improves throughput as multiple instructions can be completed in a shorter amount of time. It doesn't improve the speed of completing just one instruction, but if there are many instructions, the completion time is much faster than a single-cycle non-pipelined processor. The downside to pipelining are instances where the next instruction cannot execute, called hazards. In our design, we address data hazards and control hazards by implementing forwarding and stalling. The third type of hazard is structural hazard, when the hardware cannot support the combination of instructions executed in the same clock cycle.

2 Processor

Since there are five stages in the single-cycle pipelined processor, this means that up to five instructions will be executed in a single clock cycle. The datapath is thus split up into five sections with registers between each stage. These registers are used to retain the values of an instruction in order to pass them to the upcoming stages as that stage will then be used to execute a different instruction. The following are the registers that were added:

- Between the fetch and decode stage
- Between the decode and execute stage
- Between the execute and memory stage
- Between the memory and writeback stage

As mentioned above, the downsides of pipelining would be the need to address potential hazards that can occur from executing different instructions at the same time. To address these hazards, specifically data and control hazards, we need to implement forwarding and stalling. This required additional components, including:

- A hazard unit, which is needed to handle the data and control hazards
- A 3-to-1 MUX
- A comparator

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

Specfically, in our implementation of the pipelined processor, we placed each stage in its own .vhd file. This allowed us to work on the assignment simultaneously without changing the behavior of the pipelined processor. We also separated the forwarding unit from the hazard unit to make a distinction between their different purposes.

2.1 Design Schematic

Below is the design schematic for our implementation of the processor, showing the five stages, hazard unit, and forwarding unit.

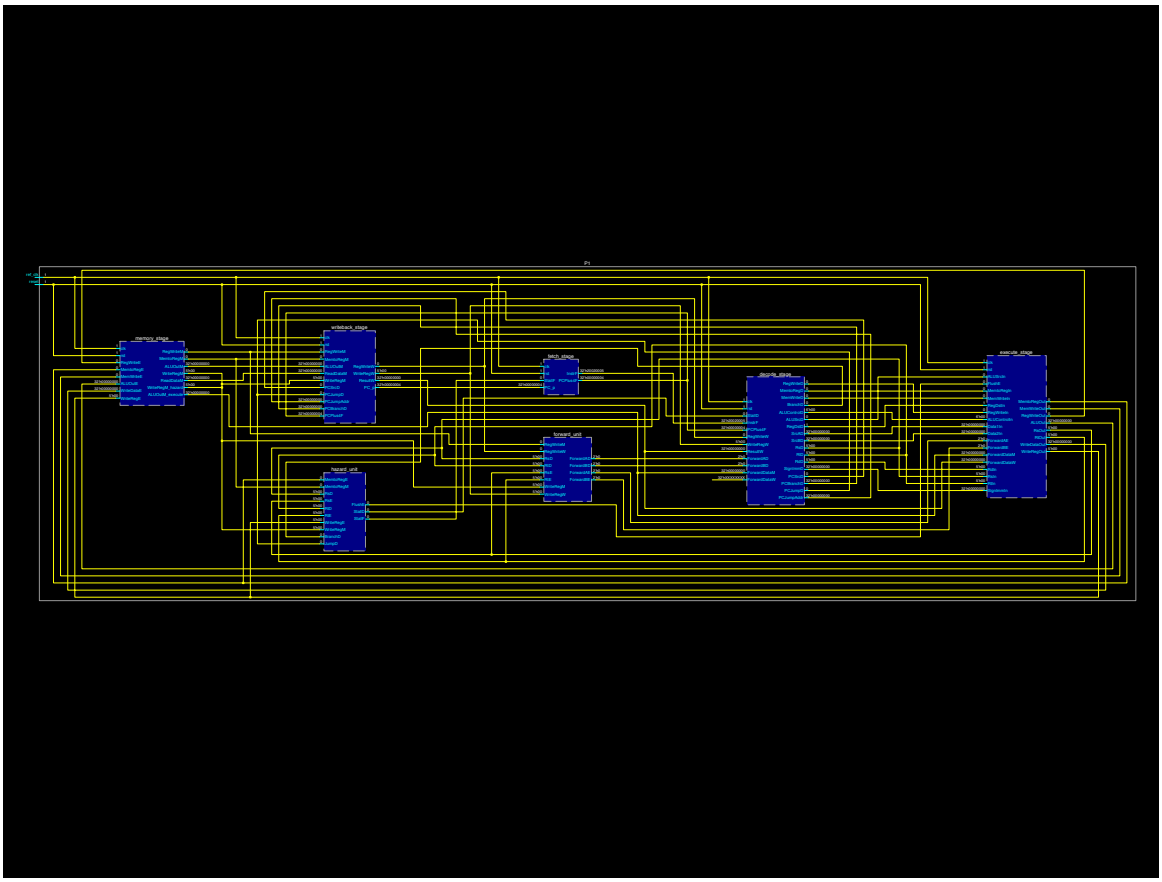


Figure 1: Design Schematic of the Processor generated from QuestaSim.

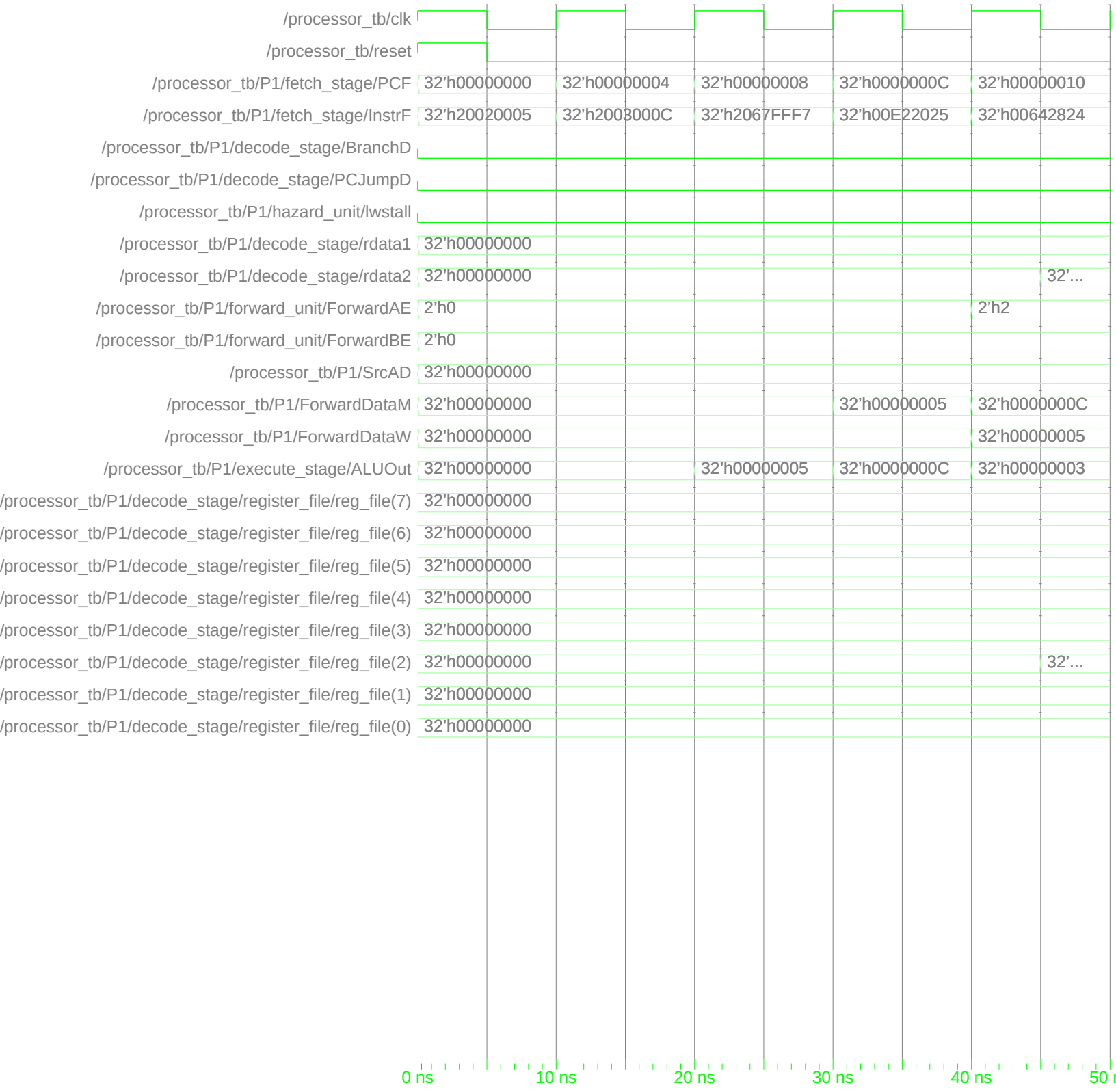
2.2 Simulation

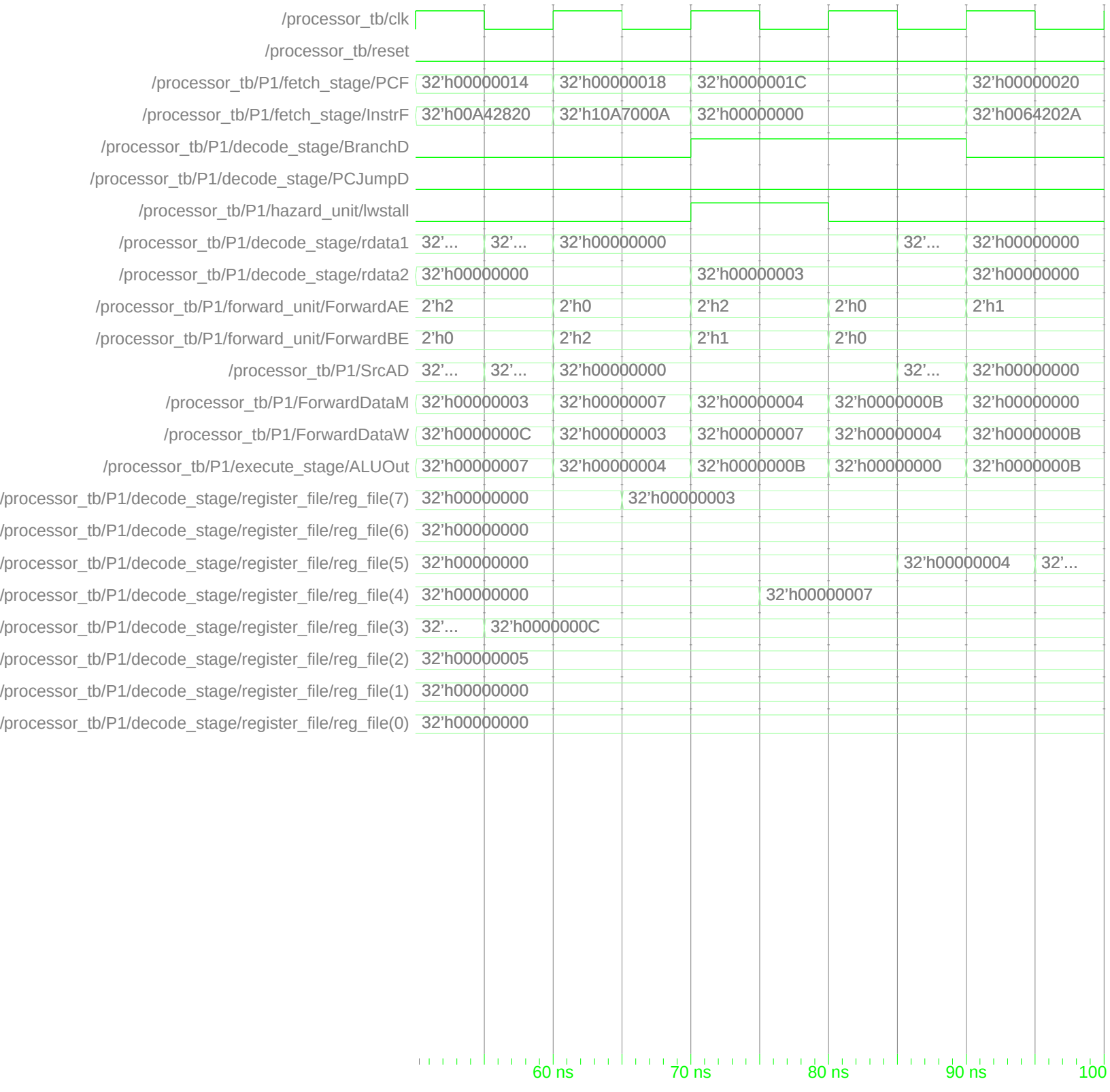
To simulate the processor, we created a testbench and our own test code in MIPS. The code is preloaded into the instruction memory. It tests R-type, I-type and J-type instructions. The following operations are supported:

Supported Instructions			
Arithmetic	Shift	Branch	Memory
ADD	SLLV	BEQ	LW
ADDU	SRLV	JUMP	SW
SUB	SRAV		
SUBU			
AND			
OR			
XOR			
NOR			
SLT			
SLTU			
ADDIU			
ANDI			
ORI			
XORI			
LUI			
SLTI			
SLTIU			

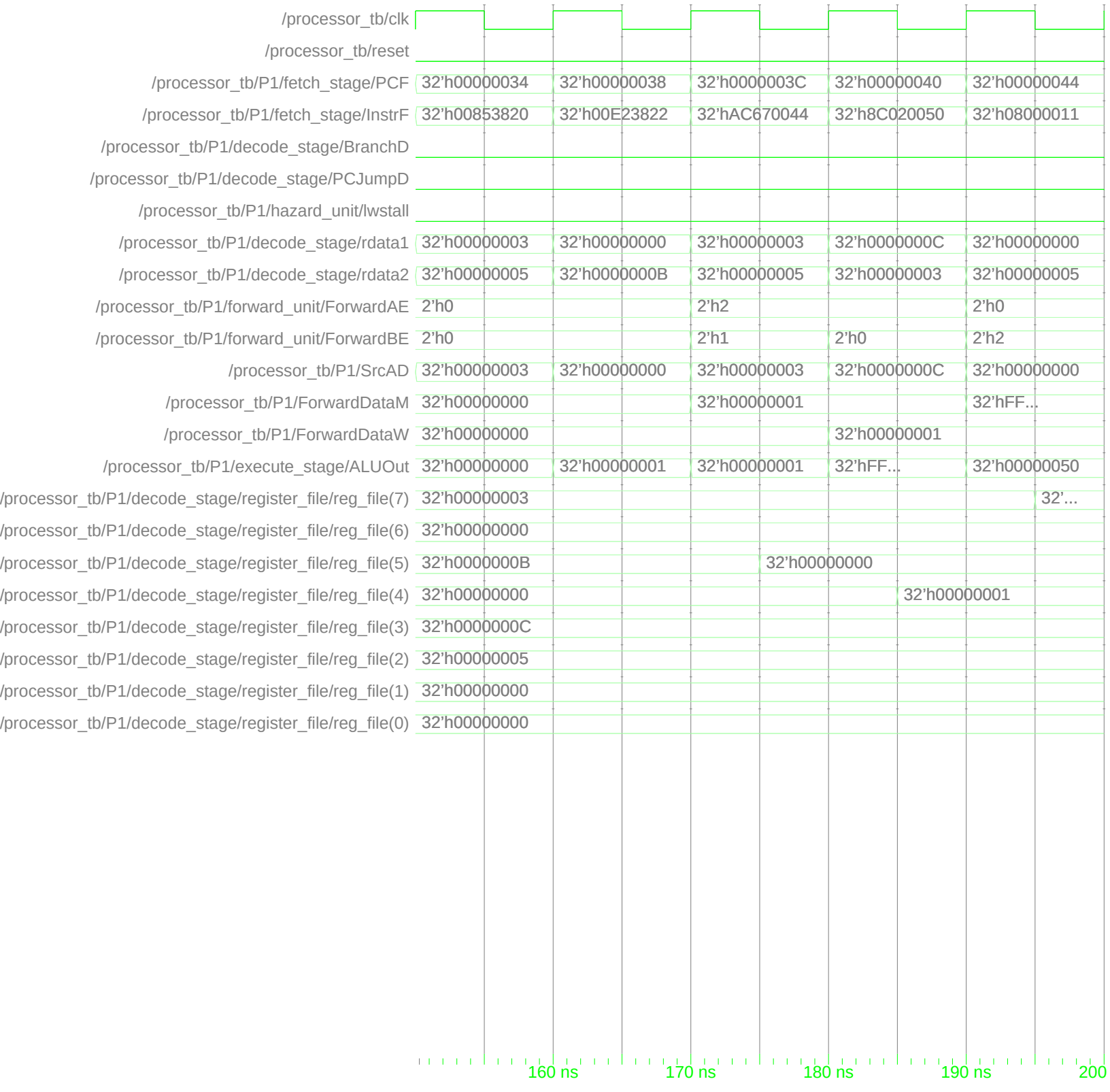
2.2.1 Waveform: Using our own MIPS test code.

A separate file in the test folder called rom3.txt provides the MIPS equivalent of the binary code for this waveform. We also have our other test files we used to verify the design, rom.txt, rom1.txt and rom2.txt.


/processor_tb/clk
/processor_tb/reset
/processor_tb/P1/fetch_stage/PCF
32'h00000000
32'h00000004
32'h00000008
32'h0000000C
32'h00000010
/processor_tb/P1/fetch_stage/InstrF
32'h20020005
32'h2003000C
32'h2067FFF7
32'h00E22025
32'h00642824
/processor_tb/P1/decode_stage/BranchD
/processor_tb/P1/decode_stage/PCJumpD
/processor_tb/P1/hazard_unit/lwstall
/processor_tb/P1/decode_stage/rdata1
32'h00000000
/processor_tb/P1/decode_stage/rdata2
32'h00000000
32'...
/processor_tb/P1/forward_unit/ForwardAE
2'h0
2'h2
/processor_tb/P1/forward_unit/ForwardBE
2'h0
/processor_tb/P1/SrcAD
32'h00000000
/processor_tb/P1/ForwardDataM
32'h00000000
32'h00000005
32'h0000000C
/processor_tb/P1/ForwardDataW
32'h00000000
32'h00000005
/processor_tb/P1/execute_stage/ALUOut
32'h00000000
32'h00000005
32'h0000000C
32'h00000003
/processor_tb/P1/decode_stage/register_file/reg_file(7)
32'h00000000
/processor_tb/P1/decode_stage/register_file/reg_file(6)
32'h00000000
/processor_tb/P1/decode_stage/register_file/reg_file(5)
32'h00000000
/processor_tb/P1/decode_stage/register_file/reg_file(4)
32'h00000000
/processor_tb/P1/decode_stage/register_file/reg_file(3)
32'h00000000
/processor_tb/P1/decode_stage/register_file/reg_file(2)
32'h00000000
32'...
/processor_tb/P1/decode_stage/register_file/reg_file(1)
32'h00000000
/processor_tb/P1/decode_stage/register_file/reg_file(0)
32'h00000000
0 ns
10 ns
20 ns
30 ns
40 ns
50 n


/processor_tb/clk
/processor_tb/reset
/processor_tb/P1/fetch_stage/PCF
32'h00000014
32'h00000018
32'h0000001C
32'h00000020
/processor_tb/P1/fetch_stage/InstrF
32'h00A42820
32'h10A7000A
32'h00000000
32'h0064202A
/processor_tb/P1/decode_stage/BranchD
/processor_tb/P1/decode_stage/PCJumpD
/processor_tb/P1/hazard_unit/lwstall
/processor_tb/P1/decode_stage/rdata1
32'...
32'...
32'h00000000
32'...
32'h00000000
/processor_tb/P1/decode_stage/rdata2
32'h00000000
32'h00000003
32'h00000000
/processor_tb/P1/forward_unit/ForwardAE
2'h2
2'h0
2'h2
2'h0
2'h1
/processor_tb/P1/forward_unit/ForwardBE
2'h0
2'h2
2'h1
2'h0
/processor_tb/P1/SrcAD
32'...
32'...
32'h00000000
32'...
32'h00000000
/processor_tb/P1/ForwardDataM
32'h00000003
32'h00000007
32'h00000004
32'h0000000B
32'h00000000
/processor_tb/P1/ForwardDataW
32'h0000000C
32'h00000003
32'h00000007
32'h00000004
32'h0000000B
/processor_tb/P1/execute_stage/ALUOut
32'h00000007
32'h00000004
32'h0000000B
32'h00000000
32'h0000000B
/processor_tb/P1/decode_stage/register_file/reg_file(7)
32'h00000000
32'h00000003
/processor_tb/P1/decode_stage/register_file/reg_file(6)
32'h00000000
/processor_tb/P1/decode_stage/register_file/reg_file(5)
32'h00000000
32'h00000004
32'...
/processor_tb/P1/decode_stage/register_file/reg_file(4)
32'h00000000
32'h00000007
/processor_tb/P1/decode_stage/register_file/reg_file(3)
32'...
32'h0000000C
/processor_tb/P1/decode_stage/register_file/reg_file(2)
32'h00000005
/processor_tb/P1/decode_stage/register_file/reg_file(1)
32'h00000000
/processor_tb/P1/decode_stage/register_file/reg_file(0)
32'h00000000
60 ns
70 ns
80 ns
90 ns
100


/processor_tb/clk
/processor_tb/reset
/processor_tb/P1/fetch_stage/PCF
32'h00000024
32'h00000028
32'h0000002C
32'h00000030
/processor_tb/P1/fetch_stage/InstrF
32'h10800001
32'h00000000
32'h20050000
32'h00E2202A
/processor_tb/P1/decode_stage/BranchD
/processor_tb/P1/decode_stage/PCJumpD
/processor_tb/P1/hazard_unit/lwstall
/processor_tb/P1/decode_stage/rdata1
32'h0000000C
32'h00000007
32'h00000000
/processor_tb/P1/decode_stage/rdata2
32'h00000007
32'h00000000
32'h0000000B
/processor_tb/P1/forward_unit/ForwardAE
2'h0
2'h1
2'h0
/processor_tb/P1/forward_unit/ForwardBE
2'h0
/processor_tb/P1/SrcAD
32'h0000000C
32'h00000007
32'h00000000
/processor_tb/P1/ForwardDataM
32'h0000000B
32'h00000000
/processor_tb/P1/ForwardDataW
32'h00000000
32'h0000000B
32'h00000000
/processor_tb/P1/execute_stage/ALUOut
32'h00000000
32'h00000000
32'h00000000
/processor_tb/P1/decode_stage/register_file/reg_file(7)
32'h00000003
/processor_tb/P1/decode_stage/register_file/reg_file(6)
32'h00000000
/processor_tb/P1/decode_stage/register_file/reg_file(5)
32'h0000000B
/processor_tb/P1/decode_stage/register_file/reg_file(4)
32'h00000007
32'h00000000
/processor_tb/P1/decode_stage/register_file/reg_file(3)
32'h0000000C
/processor_tb/P1/decode_stage/register_file/reg_file(2)
32'h00000005
/processor_tb/P1/decode_stage/register_file/reg_file(1)
32'h00000000
/processor_tb/P1/decode_stage/register_file/reg_file(0)
32'h00000000
110 ns
120 ns
130 ns
140 ns
150


/processor_tb/clk
/processor_tb/reset
/processor_tb/P1/fetch_stage/PCF
32'h00000034
32'h00000038
32'h0000003C
32'h00000040
32'h00000044
/processor_tb/P1/fetch_stage/InstrF
32'h00853820
32'h00E23822
32'hAC670044
32'h8C020050
32'h08000011
/processor_tb/P1/decode_stage/BranchD
/processor_tb/P1/decode_stage/PCJumpD
/processor_tb/P1/hazard_unit/lwstall
/processor_tb/P1/decode_stage/rdata1
32'h00000003
32'h00000000
32'h00000003
32'h0000000C
32'h00000000
/processor_tb/P1/decode_stage/rdata2
32'h00000005
32'h0000000B
32'h00000005
32'h00000003
32'h00000005
/processor_tb/P1/forward_unit/ForwardAE
2'h0
2'h2
2'h0
/processor_tb/P1/forward_unit/ForwardBE
2'h0
2'h1
2'h0
2'h2
/processor_tb/P1/SrcAD
32'h00000003
32'h00000000
32'h00000003
32'h0000000C
32'h00000000
/processor_tb/P1/ForwardDataM
32'h00000000
32'h00000001
32'hFF...
/processor_tb/P1/ForwardDataW
32'h00000000
32'h00000001
/processor_tb/P1/execute_stage/ALUOut
32'h00000000
32'h00000001
32'h00000001
32'hFF...
32'h00000050
/processor_tb/P1/decode_stage/register_file/reg_file(7)
32'h00000003
32'...
/processor_tb/P1/decode_stage/register_file/reg_file(6)
32'h00000000
/processor_tb/P1/decode_stage/register_file/reg_file(5)
32'h0000000B
32'h00000000
/processor_tb/P1/decode_stage/register_file/reg_file(4)
32'h00000000
32'h00000001
/processor_tb/P1/decode_stage/register_file/reg_file(3)
32'h0000000C
/processor_tb/P1/decode_stage/register_file/reg_file(2)
32'h00000005
/processor_tb/P1/decode_stage/register_file/reg_file(1)
32'h00000000
/processor_tb/P1/decode_stage/register_file/reg_file(0)
32'h00000000
160 ns
170 ns
180 ns
190 ns
200


/processor_tb/clk
/processor_tb/reset
/processor_tb/P1/fetch_stage/PCF
32'h00000048
/processor_tb/P1/fetch_stage/InstrF
32'h20020001
/processor_tb/P1/decode_stage/BranchD
/processor_tb/P1/decode_stage/PCJumpD
/processor_tb/P1/hazard_unit/lwstall
/processor_tb/P1/decode_stage/rdata1
32'h00000000
/processor_tb/P1/decode_stage/rdata2
32'h00000000
/processor_tb/P1/forward_unit/ForwardAE
2'h0
/processor_tb/P1/forward_unit/ForwardBE
2'h0
/processor_tb/P1/SrcAD
32'h00000000
/processor_tb/P1/ForwardDataM
32'h00000050
32'h00000000
/processor_tb/P1/ForwardDataW
32'hFF...
32'h00000050
32'hFF...
32'h00000000
/processor_tb/P1/execute_stage/ALUOut
32'h00000050
32'h00000000
/processor_tb/P1/decode_stage/register_file/reg_file(7)
32'...
32'hFFFFFFFC
/processor_tb/P1/decode_stage/register_file/reg_file(6)
32'h00000000
/processor_tb/P1/decode_stage/register_file/reg_file(5)
32'h00000000
/processor_tb/P1/decode_stage/register_file/reg_file(4)
32'h00000001
/processor_tb/P1/decode_stage/register_file/reg_file(3)
32'h0000000C
/processor_tb/P1/decode_stage/register_file/reg_file(2)
32'h00000005
32'hFFFFFFFC
/processor_tb/P1/decode_stage/register_file/reg_file(1)
32'h00000000
/processor_tb/P1/decode_stage/register_file/reg_file(0)
32'h00000000
210 ns
220 ns
230 ns
240 ns
250

3 The Pipeline Architecture

3.1 Pipelining

Pipelining is a technique that allows multiple instructions to be executed in parallel and thus improves CPU throughput. The instruction cycle is broken up into a sequence of stages. Each stage is dependent on the results from the previous stage. The idea behind pipelining instructions is similar to an assembly line. Once one stage is done processing one part of an instruction, it will save the results in a register and then begin working on another instruction. The goal is to keep each stage busy with an instruction. In this way, the time to complete each instruction is the same as a single-cycle processor, but the time to complete all the instructions is faster as there is concurrent processing.

In our VHDL design for the pipelined processor, each stage was placed in its own file with the corresponding components for that stage as well as the register between itself and the stage before it. There are five stages total: instruction fetch, instruction decode, execute, memory access, and register writeback. Between each of these stages is a pipeline register that is needed to isolate signals between the different stages. All the signals that need to move from one stage to another must pass through the pipeline registers. For example, the controller is located in the Decode stage, however, there are signals that do not get used until the Memory stage such as the MemWrite signal. This signal has to move through the registers between the Decode/Execute stage and the Execute/Memory stage in order to arrive at its destination as the write enable signal to the RAM. As an instruction moves through the stages, the registers are there to save the values and pass them on to the next stage so that the previous stage can begin working on another instruction.

For branch instructions, we encountered an issue where we would need to insert three NOPs after the branch instruction. Our solution was to move components from the Execute stage into the Decode stage and also make a copy of other components. By doing so, we were able to reduce the number of NOPs needed to one. The components that we moved were a shifter and comparator. The components we made copies of were the two MUXs that whose outputs would go as inputs for the ALU. This allowed us to get the result of the branch prediction earlier, so we wouldnt need to wait until after the Execute stage to know whether to branch or not.

3.2 Forwarding

Forwarding is a hardware technique used to avoid Read After Write (RAW) data hazards and reduce stalling. A RAW data hazard occurs when instructions use data that is dependent upon a modification in a different stage of the pipeline.

We use the forwarding technique in the decode stage and the execution stage. The result of the comparator in the decode stage is forwarded to the preceding instructions memory stage or writeback stage. The result of the ALU in the execution stage is forwarded to the preceding instructions memory or writeback stage. If RegWrite signal is high for the memory state or writeback stage, we check if the Rs or Rt signals from the decode and execution are equal to WriteRegM or WriteRegW signal. If one or more of the conditions are true, then forwarding occurs for each of the corresponding stages.

3.3 Hazard Detection

As discussed above, forwarding solves some of the data dependencies between instructions. One dependency that is not handled by the forwarding unit is load-use data hazard. Load-use data hazard occurs when there is a load instruction, and the next instruction needs the data loaded from the memory. There is nothing forwarding can do to address this, because at the time the data is needed, the data has not even become available from the memory yet. Therefore, a one-cycle stall is needed after the load instruction, so that the data can become available and loaded into a register for the next instruction to use.

To detect this condition, a hazard unit needs to check the source and target addresses of an instruction

against the destination register of the previous instruction, and it needs to know whether the previous instruction was a load instruction or not. In our implementation, the hazard unit takes the MemtoRegE signal, the addresses rs and rt from the decode stage, and the address rt from the execute stage. We take source and target addresses from the decode stage to detect this condition as early as possible. That means that the destination register will come from the execute stage. MemtoRegE tells us if the instruction before the current instruction being decoded is a load instruction or not. If it is, and the destination of the load matches either the source or target address of the decode stage, then there will be a stall.

A similar condition also applies to cases where the dependent instruction is a branch instruction. Since we moved the hardware to determine branch outcomes to the decode stage, if the branch operands use the result of the previous instruction, there needs to be a stall, because the output of the ALU will not be stable enough for the forwarding unit to forward to the decode stage. Also, if branch needs data loaded in the previous instruction, there needs to be two stalls, because the required data will be available from the memory to be forwarded only after two cycles have passed.

To address this issue, our hazard unit takes the destination addresses from the execute and memory stages, the source and target addresses and the branch signal from the decode stage, and the MemtoReg signal from the memory stage. We first check if the source and target addresses in the decode stage match the destination address of any R-type or load instruction in the execute stage. If there is a match, then we stall for one cycle. Then, we check the same source and target registers with the destination register of the memory stage. Here, we only stall if there is a match and the instruction in the memory stage is a load instruction.

Stalling is achieved by disabling or clearing the pipeline registers. To stall, we disable the writeback/fetch (program counter) and fetch/decode pipeline registers, thereby preventing the processor from moving the instructions and data already in those stages. We also clear the decode/execute pipeline register, because that register will contain the signals we wanted to stall from the fetch/decode register, and we dont want those signals to continue passing through the datapath. Clearing that register amounts to a nop in the instructions.

4 Components From Previous Labs

These components have not been modified from previous labs, specifically Lab 2 and 3.

4.1 Program Counter

The program counter is a register that stores the address of the instruction being executed. The address in the program counter is passed to the instruction memory. After the instruction is fetched, the counter gets incremented to go to the next address. In our case, the program counter is incremented by 4 using an adder as our design is byte-addressable. Our implementation of the program counter, takes in a clock signal, reset, and input. 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
input	32	IN	incremented value
output	32	OUT	outputs an address

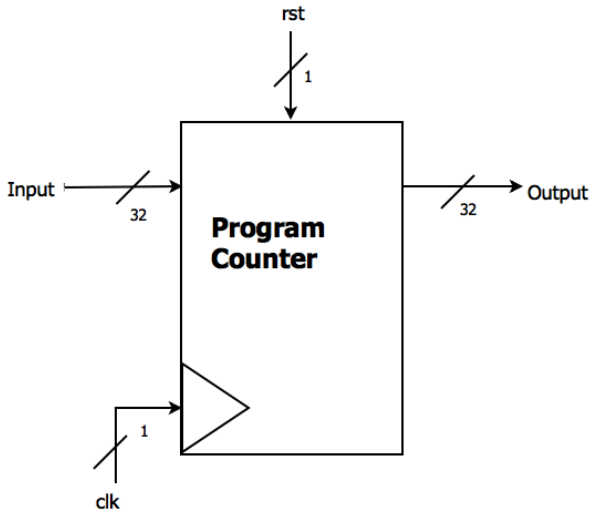


Figure 2: Program Counter Diagram

4.2 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	32	IN	address for the location of instruction, only takes the lower 9-bits
dataIO	32	INOUT	outputs an instruction

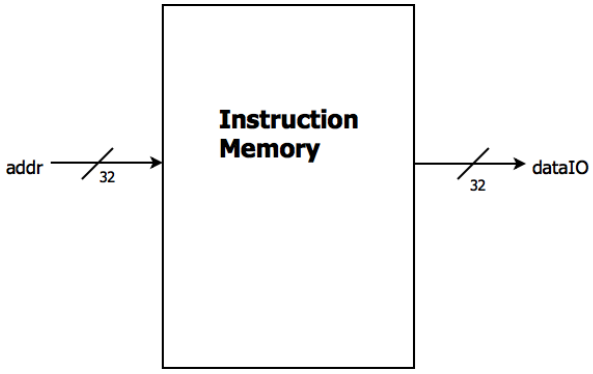


Figure 3: The instruction memory has one 32-bit input and one 32-bit output.

4.3 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.

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
Jump	1	OUT	controls jump instruction

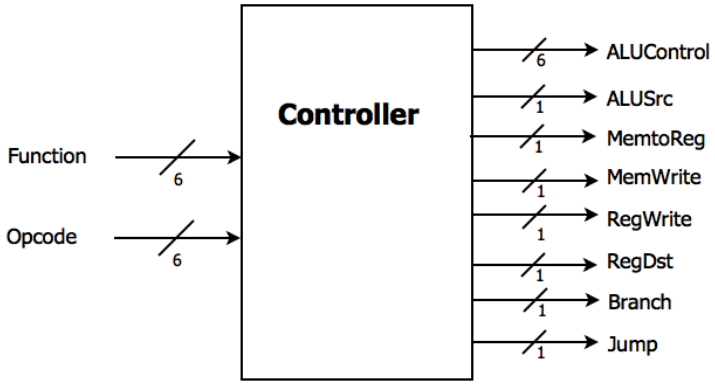


Figure 4: Diagram of the controller

4.4 ALU

The ALU performs arithmetic and logic operations on two 32-bit operands and produces a 32-bit output. It also provides a one-bit value for the branch instruction.

ALU Port Description			
Port name	Port size	Port Type	Description
Func_in	6	IN	opcode from controller
A_in	32	IN	operand A
B_in	32	IN	operand B
O_out	32	OUT	output from ALU
Branch_out	1	OUT	for branch instruction

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
sll	B << A	000X00	
srl	B >> A	000X10	
sra	B >>> A	000X11	
beq	A == B	111100	
nop	nothing	others	Any other function code does nothing

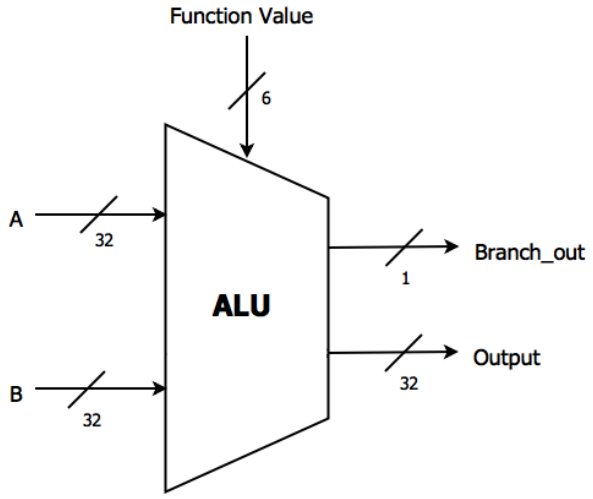


Figure 5: The ALU has 3 inputs and two outputs. There are two 32-bit inputs that act as the operands and a 6-bit input that lets the ALU know which operation to perform. The result is then sent to the output and a value for the branch.

4.5 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

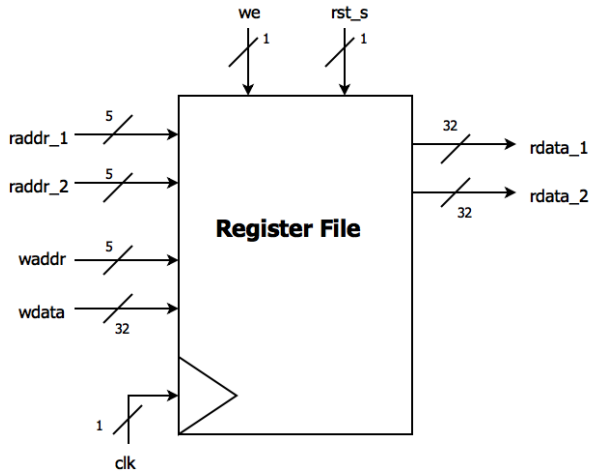


Figure 6: Register file

4.6 Multiplexer

The multiplexer chooses an output from several possible inputs based on the value of the select signal. In our design, we are using 2-to-1 MUX and 3-to-1 MUX.

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

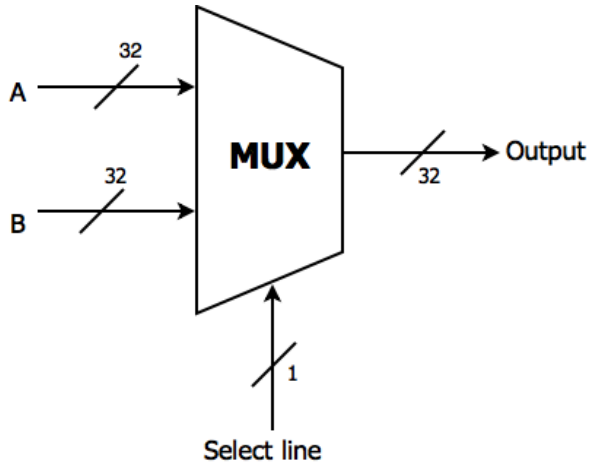


Figure 7: 2-to-1 MUX

4.7 Sign Extension Unit

The sign extension unit performs sign extension on the 16-bit immediate value provided by the controller 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 8: Takes in a 16-bit value and extends it to 32-bits.

4.8 Adder

The adder takes 32-bit inputs and adds them together to create an 32-bit sum. This adder supports signed values.

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

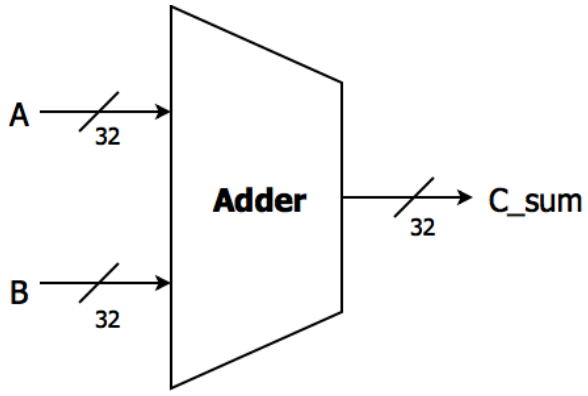


Figure 9: A diagram of an adder with two 32-bit inputs and one 32-bit output.

4.9 Shifter

The shifter shifts the input value 2 places to the left where zeros will be shifted in the least significant bit positions. The size of the input and output are the same.

Shifter Port Description			
Port name	Port size	Port Type	Description
a	32	IN	input value
o	32	OUT	shifted value

Figure 10: A diagram of a shifter that shifts inputs 2 places to the left.

4.10 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

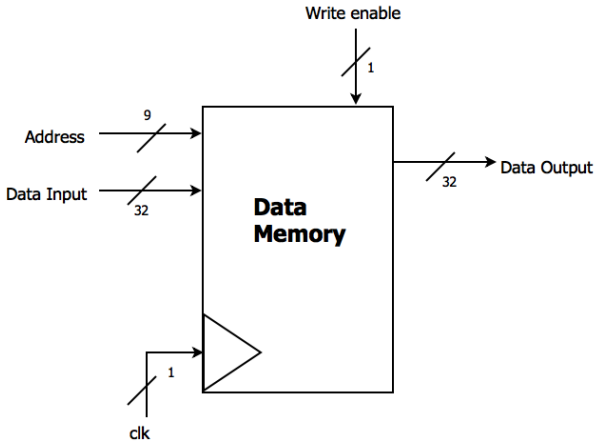


Figure 11: Random Access Memory

5 Synthesis

As with lab 3, we had to make a few changes to our design in order to be able to synthesize our processor. First, we had to use the std_logic_unsigned package instead of the numeric_std_unsigned package in the ROM, and as a result, had to change occurrences of to_integer to conv_integer. Also, we had to remove the preloading mechanism from the ROM, as it would not synthesize. Second, we had to use the built-in SRAM library, through the provided example, instead of our own design for the data memory. Additionally, we had to change the name of the clock in the processor from ref_clk to clk, and we had to hardcode some instructions into the instruction memory, because otherwise, the instruction memory would not be driving any nets.

- Area
  - Total area: 81118 square microns
  - Area for black box regions: 50667 square microns
  - Area for combinational components: 11880 square microns
- Power
  - Total power: 21 mW
  - Leakage power: 16.2 mW
  - Internal power: 4.68 mW
  - Switch power: 94 uW
- Maximum Frequency
  - Minimum clock period: 2.40 ns
  - Maximum frequency: 417 MHz

The maximum frequency of the pipelined processor increased from the maximum frequency of the single-cycle processor we designed earlier, but only slightly. Our single-cycle processors maximum frequency was 275 MHz. One possible reason why the speed didnt increase more could be due to the way we decided to split the pipeline architecture. Having a separate entity for each stage may have introduced some extra overhead into our design, as opposed to having all components instantiated in the processor.

6 Conclusion

After completing this project, we learned about the various challenges of designing a pipelined processor. To make life easier for each group member and to make it possible to have multiple members work on the project, we decided to split the processor so that each stage would reside in a separate entity. The difficult part of the process was maintaining the processor entity and keeping it up-to-date when a component interface changed. Also, we decided to use a naming convention and use that convention consistently throughout all files, in order to make understanding the code easier.

As part of this project, we also learned about the issues arising from allowing multiple instructions to be in the pipeline at the same time. We learned about some techniques to address those issues: determining branch outcomes earlier in the pipeline; forwarding the ALU result from the memory and writeback stages to the decode and execute stages; detecting data hazards not handled by forwarding; and implementing a mechanism to stall the processor.

By synthesizing the design, we were able to see the performance gain as a result of implementing pipelining. The performance gain turned out to be moderate for our design, possibly due to the structure of our implementation. However, with other design decisions, the pipelined processors performance gain should be substantial.