Organization of Digital Computer Lab
EECS112L/CSE 132L

Assignment 3
Single-cycle MIPS - Complete

prepared by: Team Stressed
Student name:
Mansi Tyagi
Student ID:
23334840

Student name:
Erik Henriquez
Student ID:
57374677

Student name:
Kevin Chau
Student ID:
76934313

Student name:
Steven Chow
Student ID:
70916812

Student name:
Paul Dao
Student ID:
30658761

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

February 21, 2016

1 Summary of Processor Design

1.1 What We Learned

After implementing a subset of MIPS R-type and I-type instructions in Lab 2, we implemented additional I-type instructions and added support for J-type instructions. After reviewing material we had previously learned in CSE 132, we were able to implement a complete single-cycle processor that met this assignment's specifications. For the synthesis portion of this assignment, we learned to use Synopsis for elaboration and to determine the power consumption, area requirements, and clock frequency necessary to build our processor design. Although the use of Synopsis was initially very troublesome, we appreciated the fact that we were able to use tools that well-known corporations, like Intel and AMD, use to design and fabricate their own products.

1.2 Description

For our complete single-cycle processor, we designed ten components: a Program Counter (PC), Controller, Instruction Memory (two versions), Register File, Arithmetic Logic Unit (including a Comparator), Data Memory (RAM), Adders, and the Processor entity. While most of these components existed in our previous processor design, they were all improved and/or modified to work with our new datapath design. The PC no longer increments an initialized variable every clock cycle; instead, it takes in an inputted address and outputs it every clock cycle, basically acting as a register. Our controller has a Branch port (that is being used) and a new Jump port, to enable the use of Branch and J-type instructions, respectively. During simulation, we use the same instruction memory (ROM) component, but we also wrote a ROM for synthesis that has instructions preloaded into its memory array. We changed the design of our Register File so it no longer used WAIT statements, which cannot be synthesized. Our ALU is roughly the same as before, with the exception of the missing Jump port that was moved to the Controller. Our Data Memory was also modified for synthesis, although we chose to use Professor Yaghini's SRAM component for our synthesis results. We created an Adder component to use for our Branch/Jump datapath. Finally, we updated the design of the Processor entity to include all our components and used combinational logic to mimic the multiplexors and logic gates needed along the datapath.

The processor now supports the following branch and J-type instructions: beq, bne, bltz, bgez, blez, bgtz, jump, jr, jal and jalr. To support these instructions, we created a separate Branch/Jump datapath. Composed of two Adder components and combinational logic that acts as the required Shifters and Multiplexors, the new datapath runs alongside our old one. For branch instructions, the Controller and ALU must send branch signals before the PC can be changed appropriately. To do this, the Controller must recognize the instruction as a Branch instruction from the OpCode and the ALU must check if the branch condition (register equivalency, register difference, etc.) is met. For J-type instructions, only the Controller's Jump signal is needed to allow the modification to the PC.

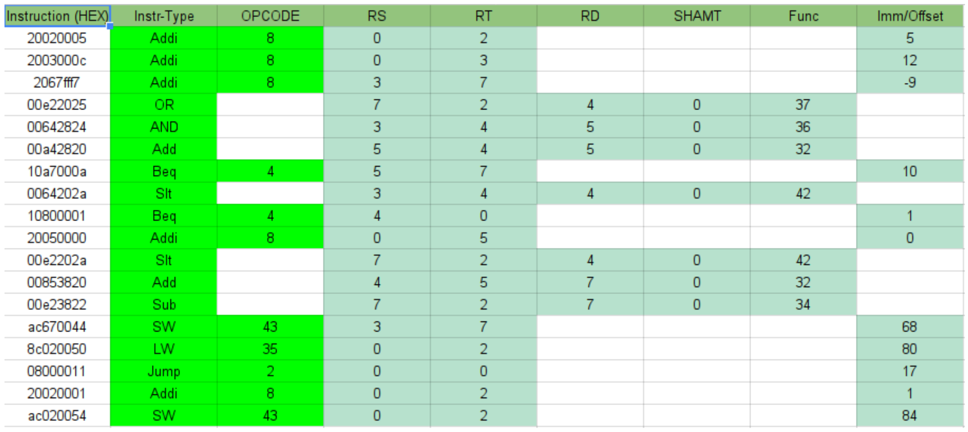
1.3 Testbench Architecture

For the most part, our testbench is the same as the one that we used for our Lab 2 assignment. We added more cycles to accommodate for the additional instructions in our Instruction Memory. For example, we run two beq instructions (one fails and the other succeeds) and a jump instruction that goes into uninitialized memory.

2 Sample Program

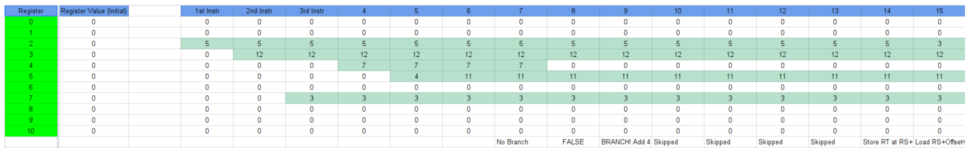
2.1 Instructions

We were given a list of 18 32-bit MIPS instructions, written in hexadecimal form. Before proceeding, we decoded these instructions into a human-readable format.

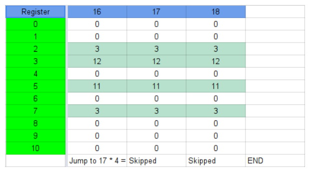


Instruction (HEX)	Instr-Type	OPCODE	RS	RT	RD	SHAMT	Func	Imm/Offset
20020005	Addi	8	0	2				5
2003000c	Addi	8	0	3				12
2067fff7	Addi	8	3	7				-9
00e22025	OR		7	2	4	0	37	
00642824	AND		3	4	5	0	36	
00a42820	Add		5	4	5	0	32	
10a7000a	Beq	4	5	7				10
0064202a	Slt		3	4	4	0	42	
10800001	Beq	4	4	0				1
20050000	Addi	8	0	5				0
00e2202a	Slt		7	2	4	0	42	
00853820	Add		4	5	7	0	32	
00e23822	Sub		7	2	7	0	34	
ac670044	SW	43	3	7				68
8c020050	LW	35	0	2				80
08000011	Jump	2	0	0				17
20020001	Addi	8	0	2				1
ac020054	SW	43	0	2				84

After decoding the instructions, we worked out the theoretical state of the Register File after every instruction. Of the two branch instructions in our sample program, one is successful and results in four instructions being skipped. There is also a jump instruction near the end of the program, which skips the final two instructions and moves the program counter into undefined memory, resulting in a "garbage" instruction.



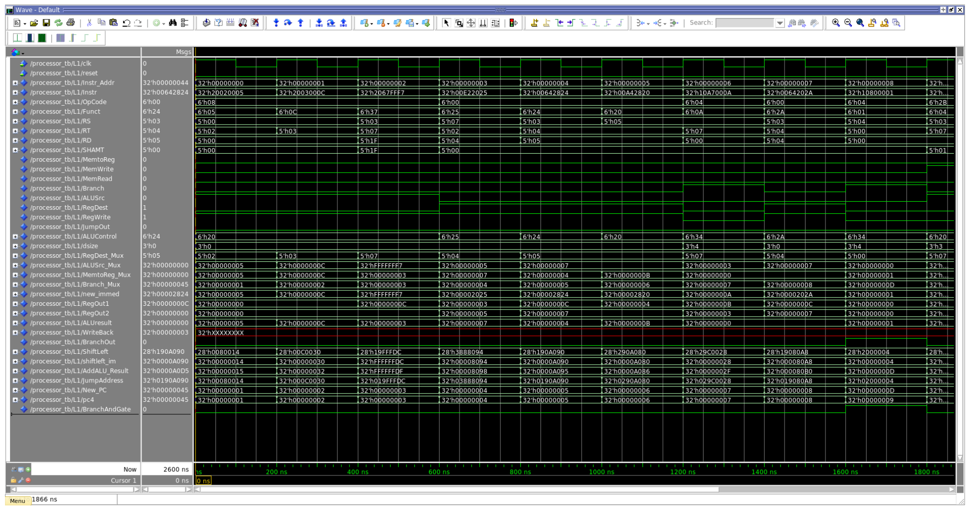
Register	Register Value (Initial)		1st Instr	2nd Instr	3rd Instr	4	5	6	7	8	9	10	11	12	13	14	15
0	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0		5	5	5	5	5	5	5	5	5	5	5	5	5	5	3
3	0		0	12	12	12	12	12	12	12	12	12	12	12	12	12	12
4	0		0	0	0	7	7	7	7	0	0	0	0	0	0	0	0
5	0		0	0	0	0	4	11	11	11	11	11	11	11	11	11	11
6	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0		0	0	3	3	3	3	3	3	3	3	3	3	3	3	3
8	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
									No Branch	FALSE	BRANCH! Add 4	Skipped	Skipped	Skipped	Skipped	Store RT at RS+	Load RS+Offset=



Register	16	17	18	
0	0	0	0	
1	0	0	0	
2	3	3	3	
3	12	12	12	
4	0	0	0	
5	11	11	11	
6	0	0	0	
7	3	3	3	
8	0	0	0	
9	0	0	0	
10	0	0	0	
	Jump to 17 * 4 =	Skipped	Skipped	END

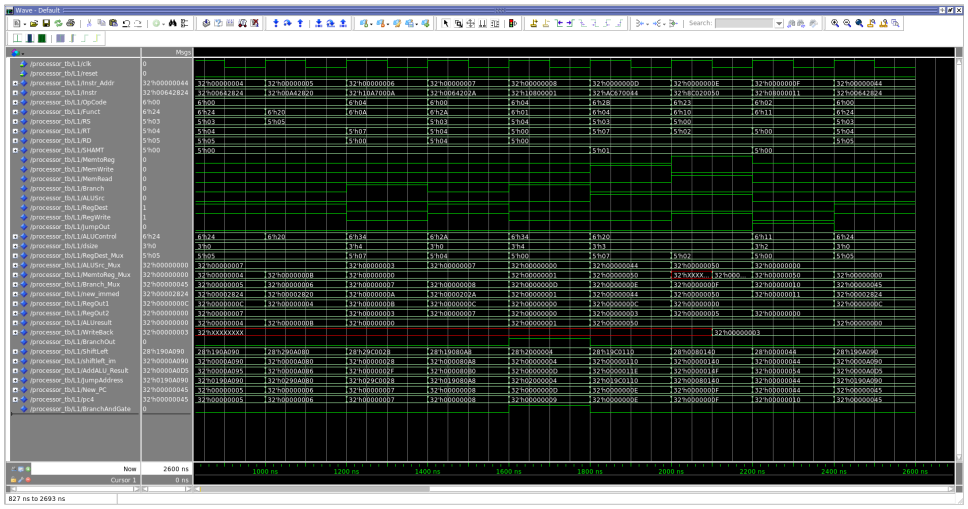
2.2 Simulation Waveform

After designing and testing our Processor, we generated the following waveforms:



	Msgs
/processor_tb/L1/clk	0
/processor_tb/L1/reset	0
/processor_tb/L1/Instr_Addr	32'h00000044
/processor_tb/L1/Instr	32'h00642824
/processor_tb/L1/OpCode	6'h00
/processor_tb/L1/Funct	6'h24
/processor_tb/L1/RS	5'h03
/processor_tb/L1/RT	5'h04
/processor_tb/L1/RD	5'h05
/processor_tb/L1/SHAMT	5'h00
/processor_tb/L1/MemtoReg	0
/processor_tb/L1/MemWrite	0
/processor_tb/L1/MemRead	0
/processor_tb/L1/Branch	0
/processor_tb/L1/ALUSrc	0
/processor_tb/L1/RegDest	1
/processor_tb/L1/RegWrite	1
/processor_tb/L1/JumpOut	0
/processor_tb/L1/ALUControl	6'h24
/processor_tb/L1/dsize	3'h0
/processor_tb/L1/RegDest_Mux	5'h05
/processor_tb/L1/ALUSrc_Mux	32'h00000000
/processor_tb/L1/MemtoReg_Mux	32'h00000000
/processor_tb/L1/Branch_Mux	32'h00000045
/processor_tb/L1/new_immed	32'h00002824
/processor_tb/L1/RegOut1	32'h0000000C
/processor_tb/L1/RegOut2	32'h00000000
/processor_tb/L1/ALUresult	32'h00000000
/processor_tb/L1/WriteBack	32'h00000003
/processor_tb/L1/BranchOut	0
/processor_tb/L1/ShiftLeft	28'h190A090
/processor_tb/L1/shiftleft_im	32'h0000A090
/processor_tb/L1/AddALU_Result	32'h0000A0D5
/processor_tb/L1/JumpAddress	32'h0190A090
/processor_tb/L1/New_PC	32'h00000045
/processor_tb/L1/pc4	32'h00000045
/processor_tb/L1/BranchAndGate	0
Now	2600 ns
Cursor 1	0 ns

1866 ns



As you can see, our branch and jump instructions worked as expected. Of the 18 instruction in the example program, only 12 were actually executed. We included a 13th cycle in our waveform to display the "garbage" instruction after our jump instruction.

3 Synthesis

Through synthesis, we were able to determine the power consumption, area requirements, and frequency of our processor design. Synopsis, a software suite provided by our university, was used to perform synthesis on our design. After adjusting the analysis, elaboration, and synthesis files and scripts, we synthesized our processor design and successfully generated various reports on our processor's performance.

3.1 Power Consumption

Our processor's total power consumption came out to 20.3 mW. From the power hierarchy we generated, approximately 74% of the power consumption is due to our register file component, with another 19.9% being used by the ALU. All other components use three orders of magnitude less power.

3.2 Area

After reading the generated area reports, we could not determine the units being used in the report. However, Synopsys reported that our Design area would be 82228.100300, with a Combinational Area of approximately 16868. We achieved these results using Professor Yaghini's SRAM. However, when using our Data Memory component (written in VHDL), our Combinational Area was over twice as large.

3.3 Required Frequency

According to the generated timing report, our Critical Path Length was 3.18 ns. In order to accommodate this and eliminate negative slack, we raised our clock period from 2.0 ns to 3.2 ns. This results in a frequency of approximately 312.5 MHz.

The majority of our timing delay comes from our PC adder, accounting for approximately 2.25 ns of the critical path length. We expect to see a reduction in the critical path in future labs, where our Adder components will not be required.

4 Known Issues

Due to our unfamiliarity with SystemVerilog, we chose not to use the testbench to preload instruction memory. For the Questasim simulation, we read the example program instructions from a file. For synthesis, we created a second version of instruction memory that has instructions preloaded into memory in order to synthesize the Processor properly.

5 Conclusion

As a complete single-cycle processor, our project now supports a larger subset of the full MIPS instruction set. It is capable of all three types of MIPS instructions: R-type, I-type, and J-type. Through synthesis, we were able to determine the area, power and frequency properties our processor design would have if fabricated. Finally, with this complete processor, we should be well prepared to implement pipelining in our future assignment.