CprE 381: Computer Organization and Assembly-Level Programming

Project Part 1 Report

Team Members:	Eli von Nordheim
	Sam Forde
Project Teams Grou	up #:Sec B 02

Refer to the highlighted language in the project 1 instruction for the context of the following questions.

[Part 1 (d)] Include your final MIPS processor schematic in your lab report.

[Part 2 (a.i)] Create a spreadsheet detailing the list of *M* instructions to be supported in your project alongside their binary opcodes and funct fields, if applicable. Create a

separate column for each binary bit. Inside this spreadsheet, create a new column for the N control signals needed by your datapath implementation. The end result should be an N*M table where each row corresponds to the output of the control logic module for a given instruction.

Spreadsheet in docs folder

[Part 2 (a.ii)] Implement the control logic module using whatever method and coding style you prefer. Create a testbench to test this module individually, and show that your



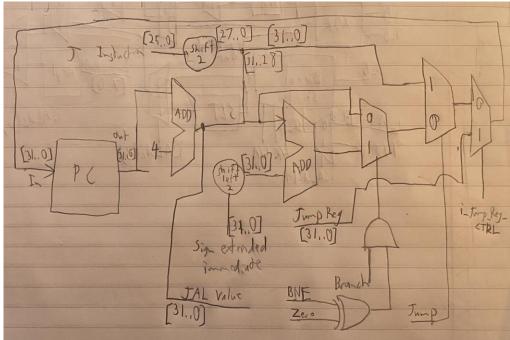
output matches the expected control signals from problem 1(a).

[Part 2 (b.i)] What are the control flow possibilities that your instruction fetch logic must support? Describe these possibilities as a function of the different control flow-related instructions you are required to implement.

The fetch logic must be able to support standard incrementing by 4, jump and jal instructions, branch instructions, and jump register.

[Part 2 (b.ii)] Draw a schematic for the instruction fetch logic and any other datapath modifications needed for control flow instructions. What additional control signals are needed?

Signals needed are JAL, Jump, a bit that knows if the branch instruction is bne or beq, and the zero detect line.



[Part 2 (b.iii)] Implement your new instruction fetch logic using VHDL. Use Modelsim to test your design thoroughly to make sure it is working as expected. Describe how the

execution of the control flow possibilities corresponds to the Modelsim waveforms in your writeup.

	Msgs				
logic/s_pcOUT	-No Data-	00400000	00400604		
logic/s_immed	-No Data-	(00000000)		AF321604	<u> </u>
logic/s_pcWRITE	-No Data-	(00400004	08000004	BD085E18	00403400
:/s_JUMPRETVAL	-No Data-	XXXXXXXX			00403400
logic/s_JALVAL	-No Data-	(00400004	00400608		<u> </u>
logic/s_JADD	-No Data-	(0000000)	2000001		
logic/s_Jump	-No Data-				
logic/s_Mux1s	-No Data-				
logic/s_JUMPRET	-No Data-				

This simulation looks at the different functions of the fetch logic and tests each of them. The first 10 ns test the standard increment. The next 10 tests a jump, where the output is the shifted and augmented input from the j type instruction. The next part tests the branch path, where the immediate value is added to the current instruction counter. The final part is the Jump Register, where a the output from the registers is multiplexed into the fetch logic.

[Part 2 (c.i.1)] Describe the difference between logical (srl) and arithmetic (sra) shifts. Why does MIPS not have a sla instruction?

srl shifts bits to the right filling in 0s as bits are shifted out, sra shifts to the right but fills in the sign bit. There is no sla because sla can be accomplished by adding a number to itself per shift.

[Part 2 (c.i.2)] In your writeup, briefly describe how your VHDL code implements both the arithmetic and logical shifting operations.

The code implements right shifting using a barrel shifter where the 0s are instead connected to a mux that can select between 0 and the sign bit. Left shifting is implemented by inverting both the input and the output of the barrel shifter.

[Part 2 (c.i.3)] In your writeup, explain how the right barrel shifter above can be enhanced to also support left shifting operations.

If both the input and output are inverted, the barrel shifter will shift left instead of right

[Part 2 (c.i.4)] Describe how the execution of the different shifting operations

Wave - Default ===																
<u>←</u>	Msgs															
<u>★</u> ◆ /tb_barrelshifte	r/s 5'h03	00		01	02	03	00		01	02	03	00		01	02	03
/tb_barrelshifte	r/s 32'h12345678	000000	12345678				000000	12345678				00000000	F2345678			
/tb_barrelshifte	r/s 0															
/tb_barrelshifte	r/s 32'h02468ACF	000000	12345678	091A2B	048D15	02468A	000000	12345678	2468AC	48D159	91A2B3	00000000	F23456	F91A2B	FC8D15	FE468A
/tb_barrelshifte	r/s 0															
/tb_barrelshifte	r/c 20 ns	20 ns														
					_											

corresponds to the Modelsim waveforms in your writeup.

The above waveform shows that the 3 shift functions work separately

→ /tb_barrelshifter/s	5'h03	01	02	03	04	05	06	07	08	10
★ /tb_barrelshifter/s	32'h12345678	12345678								
/tb_barrelshifter/s	0									
★ /tb_barrelshifter/s	32'h02468ACF	091A2B3C	048D159E	02468ACF	01234567	0091A2B3	0048D159	002468AC	00123456	00001234
/tb_barrelshifter/s	0									
/tb_barrelshifter/c	20 ns	20 ns								

The above waveform shows that all of the control bits increase the shift amount by the correct amount.

[Part 2 (c.ii.1)] In your writeup, briefly describe your design approach, including any resources you used to choose or implement the design. Include at least one design decision you had to make.

In order to get the ALU up to specification, I added an AND gate array(made out of single bit AND gates), a OR gate array(made out of single bit OR gates), a XOR gate array(made out of single bit xor gates), a SLT unit(a simple dataflow assignment of zeroes and the sign bit of the adder), and a zero detect functional unit(dataflow unit that checks if the ALU output is zero or not). I also built an ALU control unit that takes signals from the control logic and converts them into specific signals for the ALU to use.

[Part 2 (c.ii.2)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.

The ALU control takes the simplified signal from the control logic, and decodes those signals into specific bits to set for the ALU, using funct for the R-type instructions. This testbench cycles through all used configurations that the ALU control unit is designed to receive and send to the ALU.

/tb_alu_control/s_ALUCIN -No Data-	0	1	2	13	4	ļ5	6	7	8	∦F	0												
/tb_alu_control/s_funct -No Data-	(00											02	03	08	20	21	22	23	24	25	26	27	2A
/tb_alu_control/s_o -No Data-	05	01	22	02	0A	(00	OC.	03	15	XX	05	104	06	XX	22	02	2A	ĮΟA	00	Ĭ 01	03	09	0C
/tb_alu_control/s_jr -No Data-																							
/tb_alu_control/s_signed -No Data-																							

The below testbench checks the N-bit AND gate, showing that the AND is bit wise and works as expected.

	IVISUS								
/tb_andgn/s_D0	-No Data-	(000000	00	FFFFF	FF				
/tb_andgn/s_D1	-No Data-	000000	00			000000	01	(FFFFFF	FF X
/tb andgn/s O	-No Data-	000000	00			000000	01	FFFFF	FF (

The below testbench checks the N-bit OR gate, showing that the OR is bit wise and works as expected.

	maga								
/tb_orgn/s_D0	-No Data-	000000	00	FFFFF	FF				ļ(
/tb_orgn/s_D1	-No Data-	000000	00			000000	01	FFFFF	FF (
/tb_orgn/s_O	-No Data-	000000	00	FFFFF					į.

The below testbench checks the N-bit XOR gate, showing that the XOR is bit wise and works as expected.

	Msgs								
/tb_xorgn/s_D0	-No Data-	(000000	00	FFFFF	FF				ļ(
/tb_xorgn/s_D1	-No Data-	000000	00			000000	01	FFFFF	FF X
/tb_xorgn/s_O	-No Data-	000000	00	FFFFF	FF	FFFFF	FE	000000	00

The below testbench checks the possible states of SLT, and shows that only the first bit is dependant on i_D.

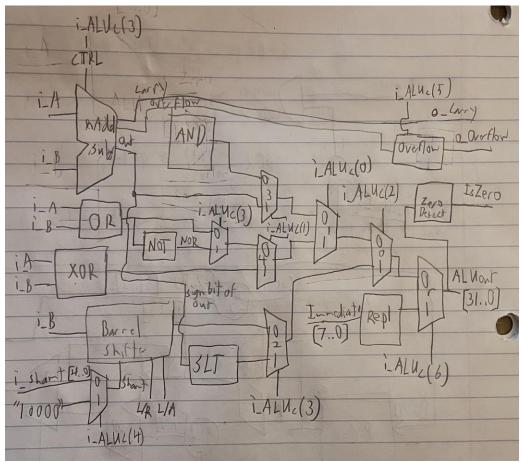
	maga			
/tb_slt/s_D /tb_slt/s_O		(00000000	00000001	

The below testbench checks the zero detect unit

/tb_zerodetect/s_O	-No Data-				
/tb_zerodetect/s_D	-No Data-	00000000	00000001	FFFFFFE	‡c

[Part 2 (c.iii)] Draw a simplified, high-level schematic for the 32-bit ALU. Consider the following questions: how is Overflow calculated? How is Zero calculated? How is slt implemented?

Overflow is a not of the last bit of i_A and a not of the last bit of i_B ANDed with the last bit of the calculated output. This part is then ORed with i_A and i_B and a NOTed output to check if the sign bit of the output differs from the expected output sign. Zero is calculated by seeing if all of the bits of the output of the ALU are zero. SLT simply fills the first bit of the output with the sign bit from the ALU and fills the rest of the bits with



zero.

[Part 2 (c.v)] Describe how the execution of the different operations corresponds to the Modelsim waveforms in your writeup.

	Mogo															
.ı/gCLK_HPER	-No Data-	10 ns														
u/s_A	-No Data-	(00000000)		000000)1	12345678	80000000	00000004	0000000	0						
u/s_B	-No Data-	12345678				00000001	80000000	00000005	0000123	4						
u/s_ALUc	-No Data-	05	04	02		0A	02	0A	01		05	04	06	15	40	
u/s_shamt	-No Data-	02									10			00		
u/s_zero	-No Data-															
.ı/s_c	-No Data-															
/s_o	-No Data-	(48D159E0	048D159	123456	79	12345677	00000000	IEEEEEEE	0000123	4	12340000	00000000		12340000	5555555	
u/s_replimmed	-No Data-	XX													55	
u/s_sign	-No Data-															
.i/cCLK_PER	-No Data-	20 ns														

This testbench checks various possible states of the ALU using spoofed control signals from the ALU control unit, and sample inputs to i_A, i_B, and shamt just to verify the system outputs properly, since all of the individual units have already been tested. The ALUc input 0x5 is a test of sll, where the output is shifted left by two. ALUc input 0x4 is srl is a right shift by the same amount. 0x2 is the add function, and 0xA is subtraction. The next two 0x2 and 0xA's test that the adder has been hooked up correctly, and the zero detect line. After that comes 0x1, which is the or function. After that comes another test of sll, used for comparison to the lui function's shifting. Then another shift right logical to check another case with zero detect, and 0x6 to check sra. Then comes 0x15, which is the load upper immediate test. Finally 0x40 is a test of repl.qb.

[Part 2 (c.viii)] justify why your test plan is comprehensive. Include waveforms that demonstrate your test programs functioning.

Instead of spending time on building an extra test bed, we opted to integrate the ALU into the MIPS processor design, and test with MIPS assembly.

[Part 3] In your writeup, show the Modelsim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

Whatever you say boss

[Part 3 (a)] Create a test application that makes use of every required arithmetic/logical instruction at least once. The application need not perform any particularly useful task, but it should demonstrate the full functionality of the processor (e.g., sequences of many instructions executed sequentially, 1 per cycle while data written into registers can be effectively retrieved and used by later instructions). Name this file Proj1_base_test.s.

	Msgs																						
CLK	0																						
MyMips/CTRLLOGIC/opcode	6'h1F	(00						Ĭ.	03	00							08				09		OA A
//www.dyMips/ALULOGIC/funct	6'h12	20	2:	2	21	X:	23	χ:	28	24	25	26	27		08	X	05		3B				OF .
vlyMips/ALU32b/i_A	32'h00000000	00000	00A (0	000001E	(00000	028)(00000000	F0F0	0F0				004000)40 X	000000	00A			00	00	(0000000A)
vlyMips/ALU32b/i_B	32'h00400040	00000	014 (0	000001E	(00000	00A (000000	14 (00000000	(OFOF	F0F				(00000	000 (000000	005	FFFFF	FFB			(0000000F)
alu_out	32'h1F1F1F1F	00000	01E (0	0000000	(00000	032 (000000	14 (00000000		X FFFFI	FFF	(0000	0000			00000	00F	00000	005	00	FF	00000001

Add Sub Addu Subu Jal And Or Xor Nor Jr Addi Addi Addiu Slti While I ran more instructions in this program than just these, this is most of the standard arithmetic/logical. This passed the MARS comparison, and if one calculates out each equation, they will find it all passes inspection.

[Part 3 (b)] Create and test an application which uses each of the required control-flow instructions and has a call depth of at least 5 (i.e., the number of activation records on the stack is at least 4). Name this file Projl_cf_test.s.



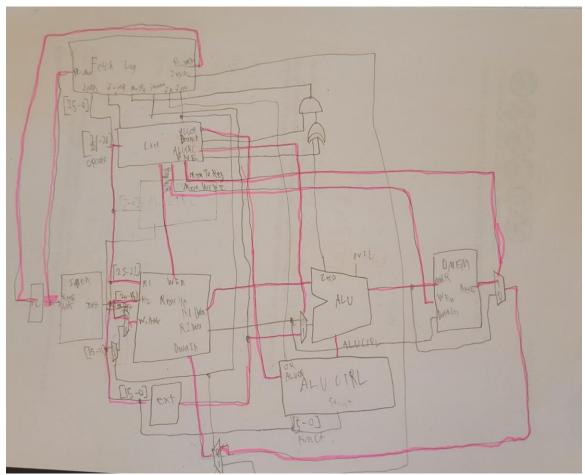
Here is a segment of the 54 instruction control-flow tests. It passed MARS comparison. [Part 3 (c)] Create and test an application that sorts an array with *N* elements using the BubbleSort algorithm (link). Name this file Proj1_bubblesort.s.

				_							_											
	Msgs																					
b/CLK (0								\neg \Box								\neg _					
b/alu_out		(100100) 0000	00 (1001	00 X 000C	00 (1001	.00 0000	00 (1001	00 🕻 000dc	00 (1001	.00 (0000	00 (1001	00 🕻 000	000 🕻 1001	000 (000	00 (1001	00 0000	00 🕻 1001	000 (000	000 (0000	00 (0	. 0	(000000)
b/MyMips/DMem/addr :		(000	001		002	000	003	(002	(004	001	005	(001	1006	(000	007	001	1008	(000	002	(0	. 001	1 (002 (0
b/MyMips/DMem/data 3		00000009	00000003		00000007		00000001		80000008		00000005		00000004		00000002		00000006	(000	0000	(0.	. 0	(00000000

The bubble sort algorithm also passed MARS comparison with no issues.

[Part 4] report the maximum frequency your processor can run at and determine what your critical path is. Draw this critical path on top of your top-level schematics. What components would you focus on to improve the frequency?

The processor's maximum frequency is 23.25 MHz, with a critical path going from the program counter to the imem, which is then decoded by the ctrl logic and ALU logic, through the ALU's adder and into DMEM which then outputs data back to the register file. This is assumed, as the critical path in timing.txt is a load word instruction that goes through the barrel shifter rather than the adder, which is not implemented in any of our



instructions, with the greatest time sink being the DMEM.