CprE 381: Computer Organization and Assembly-Level Programming

Project Part 2 Report

Team Members: Jake Hafele

Thomas Gaul

[1.a] Come up with a global list of the datapath values and control signals that are required during each pipeline stage.

			Control Signal			
Control Signal	IF (Instruction Fetch)	ID (Instruction Decode)	EX (Execute)	DMEM (Data Memory)	WB (Writeback)	Notes
reg_WE_SEL	NO	s_ID_reg_WE_SEL	s_EX_reg_WE_SEL	NO	NO	
Halt	NO	s_ID_Halt	s_EX_Halt	s_DMEM_Halt	s_WB_Halt	
nZero_Sign	NO	s_ID_nZero_Sign	NO	NO	NO	Sign extender in ID stage
ALUSrc	NO	s_ID_ALUSrc	s_EX_ALUSrc	NO	NO	
overflow_chk	NO	s_ID_overflow_chk	s_EX_overflow_chk	s_DMEM_overflow_chk	s_WB_overflow_chk	Overflow checked in last stage
branch_chk	NO	s_ID_branch_chk	NO	NO	NO	Branch control in ID stage
reg_DST_ADDR_SEL [1:0]	NO	s_ID_reg_DST_ADDR_SEL [1:0]	s_EX_reg_DST_ADDR_SEL [1:0]	NO	NO	Writeback in last stage
reg_DST_DATA_SEL [1:0]	NO	s_ID_reg_DST_DATA_SEL [1:0]	s_EX_reg_DST_DATA_SEL [1:0]	s_DMEM_reg_DST_DATA_SEL [1:0]	s_WB_reg_DST_DATA_SEL [1:0]	Writeback in last stage
PC_SEL[1:0]	NO	s_ID_PC_SEL[1:0]	NO	NO	NO	
reg_WE	NO	s_ID_reg_WE	s_EX_reg_WE	NO	NO	MUXed in ID stage and carried through as regWr
mem_WE	NO	s_ID_mem_WE	s_EX_mem_WE	s_DMEM_mem_WE	NO	
branch_SEL[2:0]	NO	s_ID_branch_SEL[2:0]	NO	NO	NO	
nAdd_Sub	NO	s_ID_nAdd_Sub	s_EX_nAdd_Sub	NO	NO	
shift_SEL[1:0]	NO	s_ID_shift_SEL[1:0]	s_EX_shift_SEL[1:0]	NO	NO	
logic_SEL[1:0]	NO	s_ID_logic_SEL[1:0]	s_EX_logic_SEL[1:0]	NO	NO	
out_SEL[1:0]	NO	s_ID_out_SEL[1:0]	s_EX_out_SEL[1:0]	NO	NO	

SW Pipeline Control Signals

			Our Datapath Signal	\$		
Datapath Signal	IF (Instruction Fetch)	ID (Instruction Decode)	EX (Execute)	DMEM (Data Memory)	WB (Writeback)	Notes
branch	NO	s_ID_branch	s_EX_branch	NO	NO	
PC [31:0]	s_IF_PC [31:0]	NO	NO	NO	NO	Assign to nextInstrAddr
PC_4 [31:0]	s_IF_PC_4 [31:0]	s_ID_PC_4 [31:0]	s_EX_PC_4 [31:0]	s_DMEM_PC_4 [31:0]	s_WB_PC_4 [31:0]	Has to propogate through for writeback for jal to set r[31] = PC + 4
INSTR [31:0]	NO	s_ID_INSTR [31:0]	NO	NO	NO	Use given Instr signal for IF
OPCODE [5:0]	NO	s_ID_OPCODE [5:0]	NO	NO	NO	
FUNCT [5:0]	NO	s_ID_FUNCT [5:0]	NO	NO	NO	
rt_ADDR [4:0]	NO	s_ID_rt_ADDR [4:0]	s_EX_rt_ADDR [4:0]	NO	NO	
rs_ADDR [4:0]	NO	s_ID_rs_ADDR [4:0]	NO	NO	NO	
rd_ADDR [4:0]	NO	s_ID_rd_ADDR [4:0]	s_EX_rd_ADDR [4:0]	NO	NO	Needed for WB
SHAMT [4:0]	NO	s_ID_SHAMT [4:0]	s_EX_SHAMT [4:0]	NO	NO	
IMM [15:0]	NO	s_ID_IMM [15:0]	NO	NO	NO	Not needed since sign extender in ID and turns into sign extended immediate 32 bit
J_ADDR [25:0]	NO	s_ID_J_ADDR [25:0]	NO	NO	NO	
rs_DATA [31:0]	NO	s_ID_rs_DATA [31:0]	s_EX_rs_DATA [31:0]	NO	NO	
rt_DATA [31:0]	NO	s_ID_rt_DATA [31:0]	s_EX_rt_DATA [31:0]	s_DMEM_rt_DATA [31:0]	NO	DMemData used in DMEM stage
ALUOut [31:0]	NO	NO	s_EX_ALUOut [31:0]	s_DMEM_ALUOut [31:0]	s_WB_ALUOut [31:0]	DMemAddr used in DMEM stage. When to assign for output? WB?
rt_DATA_MUX [31:0]	NO	NO	s_EX_rt_DATA_MUX [31:	NO	NO	MUX for second ALU operand between RT DATA and EXT IMM
IMM_EXT [31:0]	NO	s_ID_IMM_EXT [31:0]	s_EX_IMM_EXT [31:0]	NO	NO	
overflow	NO	NO	s_EX_overflow	s_DMEM_overflow	s_WB_overflow	Overflow should propogate to end of final stage (WB)
DMEM_DATA [31:0]	NO	NO	NO	s_DMEM_DMEM_DATA [s_WB_DMEM_DATA [31	Needed between EX/MEM buffer to writeback MUX
			MUX'd signals			
RegWr	NO	NO	s_EX_RegWr	s_DMEM_RegWr	s_WB_RegWr	Final write signal, MUXXED out
RegWrAddr [4:0]	NO	NO	s_EX_RegWrAddr [4:0]	s_DMEM_RegWrAddr [4:0	s_WB_RegWrAddr [4:0]	MUX with rd_ADDR, rt_ADDR, and decimal 31 (\$ra reg)

SW Pipeline Datapath Signals

[1.b.ii] high-level schematic drawing of the interconnection between components.



[1.c.i] include an annotated waveform in your writeup and provide a short discussion of result correctness.

The assembly file ran is titled Proj2SW_cf_test.s



40ns- addi 100ns- add 160ns- addiu

Wave - Default											- t 🖻
🔬 .	Msgs										
General Inputs											
s iclk											
📣 iRST											
Instruction Fetch											
	32'h00094823	254AFFFF	01084021	01294821	014A5021	00084022	00094822	000A5022	00084023	00094823	000
🖸 🔷 s_IF_PC	32'h00400040	00400020	00400024	00400028	0040002C	00400030	00400034	00400038	0040003C	00400040	004
s_IF_PC_4	32'h00400044	00400024	00400028	0040002C	00400030	00400034	00400038	0040003C	00400040	00400044	004
— Instruction Decode —											
■→ s_ID_INSTR	32'h00084023	25290000	254AFFFF	01084021	01294821	014A5021	00084022	00094822	000A5022	00084023	000
s_id_opcode	6'h00	09		00							
s_ID_FUNCT	6'h23	00	3F	21			22			23	
	32'h00400040	00400020	00400024	00400028	0040002C	00400030	00400034	00400038	0040003C	00400040	004
■→ s_ID_rs_ADDR	5'h00	09	0A	08	09	0A	00				
s_ID_rt_ADDR	5'h08	09	0A	08	09	0A	08	09	0A	08	09
s_ID_rd_ADDR	5'h08	00	1F	08	09	0A	08	09	0A	08	09
💽 🤣 s_ID_rs_DATA	32'h00000000	00000000	FFFFFFE	0000003	00000000	FFFFFFD	00000000				
💽 🔷 s_ID_rt_DATA	32'hFFFFFFFA	00000000	FFFFFFE	0000003	00000000	FFFFFFD	0000006	00000000	FFFFFFA	FFFFFFA	000
- Execute											
s_EX_nAdd_Sub		0						1			
	2'h0	0									
s_EX_logic_SEL	2'h0	0									
s_EX_out_SEL	2'h0	0									
S_EX_PC_4	32'h0040003C	0040001C	00400020	00400024	00400028	0040002C	00400030	00400034	00400038	0040003C	004
	5'h0A	08	09	0A	08	09	0A	08	09	0A	08
s_EX_rd_ADDR	5'h0A	00		1F	08	09	0A	08	09	0A	08
s_EX_rs_DATA	32'h00000000	00000002	00000000	FFFFFFE	0000003	00000000	FFFFFFD	00000000			
s_EX_rt_DATA	32'hFFFFFFA	00000002	00000000	FFFFFFE	0000003	0000000	FFFFFFD	0000006	00000000	FFFFFFA	
s_EX_ALUOut	32'h00000006	00000003	00000000	FFFFFFD	0000006	00000000	FFFFFFA	FFFFFFA	00000000	00000006	
DMEM											
s_DMEM_reg_DS		0									
S_DMEM_mem		0									
s_DMEM_PC_4	32'h00400038	00400018	0040001C	00400020	00400024	00400028	0040002C	00400030	00400034	00400038	004
🖘 📥 e DMEM RanWr	5%00	04	08	00	04	08	00	04	08	09	04
🛎 📰 💿 👘 Now		2	200 ns	220 ns	240 ns	260 ns	280 ns	300 ns	320 ns	340 ns	360 ns
🔁 🎤 😑 Cursor 7				220 ns							
🔁 🎤 😑 🛛 Cursor 8	280 ns						280 ns				
Cursor 9	340 ns									340 ns	
	13	4		1 1 1							

220ns- addu 280ns- sub 340ns- subu

Kan Msgs	
- General Innuts	
★ ICLK 1	
Pr☆ s Inst 32/b00085026 3C08FFFF 00000000 3508FFFF 00000000 00085024 310A0000 00085027	00085
B ♦ S IF PC 32/h0040006C 0040004B 0040005C 00400050 00400054 00400058 0040005C 00400064 00400068	00400
■ ♦ s IF PC 4 32760400070 0040005C 00400050 00400054 00400058 0040005C 00400066 00400066 0040006C	00400
s b INSTR 32/h00085027 000A5023 3C08FFF 00000000 3508FFF 00000000 00085024 310A0000	00085
■ ♦ s_ID_OPCODE 6100 00 00 00 00 00 00 00	00
s_0_FUNCT 6h27 2k 3F 00 3F 00 24 00	27
s D PC 4 3210040006C 00400048 0040004C 00400050 00400054 00400058 0040005C 00400064 00400068	00400
s_D_rs_ADDR 5100 00 00 00 00	00
© 🛇 s_ID_rt_ADDR 5h08 _0A _08 _0A _08 _0A	08
G 🛇 s_ID_rd_ADDR 5h0A 0A 1F 00 1F 00 00 0A 00	0A
© ♦ s_ID_rs_DATA 32/h0000000 00000000 FFFFF0000 00000000 FFFFF0000	00000
© ◇ s_D_rt_DATA 32hFFFFFFF 00000000 00000000 00000000 FFFFFF	FFFFF
C s_EX_shit_SEL 2h0 0 2 0	
• ◆ s_EX_logic_SEL 2ħ0 0	
■ S = S = 212 0 1 2 2 1 2 2	
s_EX_PC_4 32100400068 00400044 00400048 0040004C 00400050 00400054 0040005C 00400060 00400064	00400
C S EX. TL ADDR 5 MAA 09 0A 08 00 00 00 00 08	0A
C S EX rd ADDR 5m00 09 0A 1F 00 1F 00 0A 0A	00
	FFFFF
	FFFFF
	00000
400 ns 420 ns 440 ns 460 ns 480 ns 500 ns 520 ns 540 ns 56	ns
40 IIS 20	

400ns- lui 420ns- nop 460ns- ori 480ns- nop 520ns- andi 540ns- andi 560ns- nor

wa Wa	ive - Default 🖂 🔤																			+ 8
8.		Msgs																		
	General Inputs																			
4		1	-																	
4	irst 🛛	0																		
•	s_inst	32'h0109502A	00085026	390A0000	0	00085025		350A0000		20080005		2009FFFB		00000000				0128502A		01095
•	s_IF_PC	32'h00400090	0040006C	00400070	0	00400074		00400078		0040007C		00400080		00400084		00400088		0040008C		00400
.	s_IF_PC_4	32'h00400094	00400070	00400074	0	00400078		0040007C		00400080		00400084		00400088		0040008C		00400090		00400
	s_ID_INSTR	32'h0128502A	00085027	00085026	3	390A0000		00085025		350A0000		20080005		2009FFFB		00000000				01285
.	s_ID_OPCODE	6'h00	00		0)E		00		0D		08				00				
	s_ID_FUNCT	6'h2A	27	26	0	00		25		00		05		3B		00				2A
B -4	s_ID_PC_4	32'h00400090	0040006C	00400070	0	00400074		00400078		0040007C		00400080		00400084		00400088		0040008C		00400
	s_ID_rs_ADDR	5'h09	00		0	98		00		08		00								09
B <	s_ID_rt_ADDR	5'h08	08		0)A		08		0A		08		09		00				08
⊞ -₹	s_ID_rd_ADDR	5'h0A	0A		0	00		0A		00				1F		00				0A
B <	s_ID_rs_DATA	32'hFFFFFFB	00000000		F	FFFFFFF		00000000		FFFFFFF		00000000								FFFFF
	s_ID_rt_DATA	32'h00000005	FEFFFFFF		0	00000000		FFFFFFFF		FFFFFFFF				00000000						00000
	Execute																			
	s_EX_nAdd_Sub	1'n0	0																	
B <	s_EX_shift_SEL	2'n0	0																	
	s_EX_logic_SEL	2'n0	0	1	2	2				3				0						
B -4	s_EX_out_SEL	2'n1	2											0				1		
⊡ ≺	s_EX_PC_4	32'h0040008C	00400068	0040006C	0	00400070		00400074		00400078		0040007C		00400080		00400084		00400088		00400
•	s_EX_rt_ADDR	5'n00	0A	08				0A		08		0A		08		09		00		
	s_EX_rd_ADDR	5'n00	00	0A				00		0A		00				1F		00		
	s_EX_rs_DATA	32'h00000000	FFFFFFFF	00000000				FFFFFFFF		00000000		FFFFFFFF		00000000						
■ <	s_EX_rt_DATA	32'h00000000	FFFFFFA	FFFFFFF				00000000		FFFFFFFF						00000000				
A 4	S EY ALLIOUT	32%0000000	0000000	0000000		CREEPER		FFFFFFFF						00000005		REFERENC		0000000		
24 B		1260 ns	58	0 ns	600 r	ns	62	ns	640	ns	660	ns	680	ns	70	ns	720) ns	740	ns
		580 ns	58	0 ns 2) ns															
		600 ns				1 <mark>5</mark> 2	0 ns													
		620 ns					62) ns2	0 ns											
		640 ns							640	l ns 2	0 ns									
		660 ns									660) ns	20 ns							
		680 ns												Ins 2	10 ns					
	Cursor 24	700 ns) ns		10 ns		
- E /	Cursor 25	740 ns																	740) ns

580ns- xor 600ns- xori 620ns- or 640ns- ori 660ns- andi 700ns- nop 740ns- slt

wave - Default	lt :=====											+ - 2
2.		Msgs										
General Inpi	outs ——											
s iCLK												
irst 🄄												
Instruction F	Fetch											
		32'h0109502A	0109502A	290AFFFB	290A0000	290A0005	292AFFFB	292A0000	292A0005	3C08FFFF	00000000	
s_IF_PC	:	32'h00400090	00400090	00400094	00400098	0040009C	004000A0	004000A4	004000A8	004000AC	004000B0	
	_4	32'h00400094	00400094	00400098	0040009C	004000A0	004000A4	004000A8	004000AC	004000B0	004000B4	
- Instruction D	Decode —											
	STR	32'h0128502A	0128502A	0109502A	290AFFFB	290A0000	290A0005	292AFFFB	292A0000	292A0005	3C08FFFF	
🖪 🎝 s_ID_OP	CODE		00		0A						OF	
s_ID_FUI	JNCT		2A		3B	00	05	3B	00	05	3F	
	2_4	32'h00400090	00400090	00400094	00400098	0040009C	004000A0	004000A4	004000A8	004000AC	004000B0	
📕 🖸 🔶 💁 📕	ADDR	5'h09	09	08				09			00	
📃 🕁 🔷 s_ID_rt_A	ADDR	5'h08	08	09	0A						08	
B	_ADDR		0A		1F	00		1F	00		1F	
🖪 🔄 🔶 🖪 🔄	DATA	32'hFFFFFFB	FFFFFFB	00000005				FFFFFFB			00000000	
	DATA	32'h00000005	00000005	FFFFFFB	FFFFFFF	00000001	00000000				00000005	
- Execute -												
	Add_Sub		0	1								
s_EX_shi	nift_SEL		0									
B- S_EX_log	gic_SEL		0									
	ut_SEL		1	3								
S_EX_PC	C_4	32'h0040008C	0040008C	00400090	00400094	00400098	0040009C	004000A0	004000A4	004000A8	004000AC	
	_ADDR	5'h00	00	08	09	0A						
s_EX_rd_	_ADDR		00	0A		1F	00		1F	00		
■ s_EX_rs_	_DATA	32'h00000000	0000000	FFFFFFB	00000005				FFFFFFB			
	_DATA	32'h00000000	0000000	0000005	FFFFFFB	FFFFFFF	00000001	00000000				
		22%00000000	0000000	0000001	0000000		0000000		0000000	0000001		
- 65 0	NOW	1260 ns		760 ns	780 ns	800 ns	820 ns	840 ns	860 ns	880 ns	900 ns	920
ê 🖉 😑 🕐	Cursor 29	780 ns			780 ns 20 ns							
i 🔁 🖉 😑 👘	Cursor 30	800 ns				800 ns 20 ns						
🖻 🌽 😑 🔹 🦉	Cursor 31	820 ns					820 ns 20 ns					
	Cursor 32	840 ns						840 ns 20 ns				
	Cursor 33	860 ns							860 ns 20 ns			
	Cursor 34	880 ns								880 ns 20 ns		
	Cursor 35	900 ns									900 ns	
	Cursor 36	744 ns	744 ns									
5	2		<									

780ns- slti 900ns- lui

💼 Wave - Default 💳										🛨 🛃
<u> </u>	Msgs									
— General Inputs —										
A ICLK										
irst 🇳										
- Instruction Fetch -										
s_Inst	32'h00084FC2	00000000	3408FFFF	00000000		00084C00	00084FC0	00084C02	00084FC2	00084C03
	32'h004000D0	004000B0 004000B4	004000B8	004000BC	004000C0	004000C4	004000C8	004000CC	004000D0	004000D4
	32'h004000D4	004000B4 004000B8	004000BC	004000C0	004000C4	004000C8	004000CC	004000D0	004000D4	004000D8
— Instruction Decode										
s_iD_INSTR	32'h00084C02	3C08FFFF 00000000		3408FFFF	00000000		00084C00	00084FC0	00084C02	00084FC2
■→ s_ID_OPCODE	6'n00	0F 00		0D	00					
s_ID_FUNCT	6'n02	3F 00		3F	00				02	
s_ID_PC_4	32'h004000D0	004000B0 004000B4	004000B8	004000BC	004000C0	004000C4	004000C8	004000CC	004000D0	004000D4
• s_ID_rs_ADDR	5ħ00	00								
	5ħ08	08 00		08	00		08			
s_ID_rd_ADDR	5'h09	1F 00		11F	00		09			
s_ID_rs_DATA	32'h00000000	0000000								
s_ID_rt_DATA	32'h0000FFFF	00000005 00000000		FFFF0000	00000000		0000FFFF	COOOFFFF		
Execute	48.0									
s EX nAdd Su	1110									
S_EX_Shint_SEL	2 mu	0 2	0							
S_EX_logic_SEL	210	2 1			12	0				
	22%004000000	004000 00400080	00400084	00400089	0040008C	00400000	004000C4	00400000	004000000	00400000
	5108	04 08	00	00400020	08	00	00400004	08	00400000	00400000
	5100	00 1E	00		1E	00		09		
s EX rs DATA	32'h00000000	FFFFFFB 0000000								
s EX rt DATA	32'h0000FFFF	00000000 0000005	00000000		FFFF0000	00000000		0000FFFF		
s EX ALUOUT	32'h80000000	00000001 FFFF0000	00000000		0000FFFF	00000000		FFFF0000	80000000	00000000
DMEM										
	S 2ħ0	0								
	1ħ0	0								
A OMEM DC /	32%00/000008	00400048 0040004C	00400080	00400084	00400088	004000BC	00400000	004000C4	004000C8	004000
🛎 🧱 🖲 🛛 🛛 Ni	w 1260 ns	920 ns	940 ns	960 ns	980 ns	1000 ns	1020 ns	1040 ns	1060 ns	1080 ns
🔂 🎤 🥥 🛛 Cursor	37 920 ns	920 ns	40 ns							
🔁 🎤 😑 🛛 Cursor	38 960 ns			960 ns 20 ns						
🔁 🎤 😑 🦳 Cursor	39 980 ns				980 ns		80 ns			
Cursor	40 1060 ns								1060 ns	

920ns- nop 960ns- ori 980ns - nop 1020ns- sll 1060ns- srl

wave - D	efault :=====								1000								
۵.		Msgs															
- Genera	al Inputs																
A iCLH		1	-														
🔺 iRS		0		7 F													
- Instruct		-		1													
🗉 🔶 s In		32'hXXXXXXX	00084C03	00084FC3	20090	000	3C09FFFF		50000000		XXXXXXX						
🗉 🔶 s IF		32'h004000EC	004000D4	004000D8	00400	DDC	004000E0		004000E4		004000E8		004000EC	004000F0		004000F4	
🖪 🔶 s_IF		32'h004000F0	004000D8	004000DC	00400	DEO	004000E4		004000E8		004000EC		004000F0	004000F4		004000F8	
- Instruct																	
	D_INSTR	32'hXXXXXXXX	00084FC2	00084C03	00084	FC3	20090000		3C09FFFF		50000000		XXXXXXXX				
	D_OPCODE	6'hXX	00				08		0F		14		XX				
🖪 🔶 s_ID	D_FUNCT	6'hXX	02	03			00		3F		00		XX				
🖪 🔶 s_ID	D_PC_4	32'h004000EC	004000D4	004000D8	00400	DC	004000E0		004000E4		004000E8		004000EC	004000F0		004000F4	
🖪 🔶 s_ID		5'hXX	00										XX				
🖪 🔶 s_ID		5'hXX	08				09				00		XX				
🖪 🔶 s_ID		5'hXX	09				00		1F		00		XX				
🖪 🔿 s_ID		32'n00000000	00000000														
🗉 🔶 s_ID		32'h00000000	0000FFFF				00000000										
- Execut	te																
🔄 🕀 🔶 s_E	X_nAdd_Sub	1'h0	0														
💽 🔷 S_E	X_shift_SEL	2'h0	1		3				0		2		0				
🗖 🔷 s_E		2'h0	0														
💽 🔷 S_E	X_out_SEL	2'h0	1						0		1		0				
🖪 🔿 s_E	X_PC_4	32'h004000E8	004000D0	004000D4	00400	DD8	004000DC		004000E0		004000E4		004000E8	004000EC		004000F0	
🗷 🔶 s_E	X_rt_ADDR	5'h00	08						09				00	XX			
	X_rd_ADDR	5'h00	09						00		1F		00	XX			
	X_rs_DATA	32'h00000000	00000000						_								
🖪 🔿 s_E	X_rt_DATA	32'h00000000	0000FFFF						00000000								
	X_ALUOut	32'h00000000	00000000						00000000		FFFF0000		00000000				
- DMEM																	
🖪 🔷 s_D	MEM_reg_DS	2'n0	0														
	MEM mom	110												 			in month
		1260 ns	110	00 ns	1120 ns		40 ns	116	0 ns	118	0 ns	120	0 ns	20 ns	124	0 ns	
	Cursor 42	1100 ns	11	00 ns	40 ns												
	Cursor 43	1140 ns					40 ns	20 ns									
	Cursor 44	1160 ns							0 ns	20 ns							
	Cursor 45	1180 ns									io ns	20 ns					
	Cursor 46	1200 ns										120	<u>O ns</u>				
I (1)		131	5														1 1 1

1100ns- sra 1140ns- addi 1160ns- lui 1180ns- halt [1.c.ii] Include an annotated waveform in your writeup of two iterations or recursions of these programs executing correctly and provide a short discussion of result correctness. In your waveform and annotation, provide 3 different examples (at least one data-flow and one control-flow) of where you did not have to use the maximum number of NOPs.

	0x0000	0x0004	0x0008	0x000c
step	array[0]	array[1]	array[2]	array[3]
null	3105	5	-68	3
0	5	-68	3	3105
1	-68	3	5	3105
2	-68	3	5	3105
3	-68	3	5	3105

The assembly file ran is Proj2SW_bubblesort.s

Each step in the above table represents a full sweep through each array value to compare with the next index, to see if it is larger than the following array index. If it is the case, then the two values swap positions and are rewritten into the data memory with swapped indices. The following five screenshots show when s_DMEM_mem_WE is asserted to 1, indicating a write to the data memory of the array has occurred. We can prove our design works as intended by confirming that the smaller of the two compared array values is written first to the lower array memory address, and in turn getting "sorted" to a lower index. Given the above table, this swap occurs 5 times between the first two loops. Each screenshot below shows one swap at the cursor position, taking two consecutive instructions to write back to the data memory.



Bubble Sort Array Swap 1



Bubble Sort Array Swap 2



Bubble Sort Array Swap 3

	referm	1	_							_						_	_	_	_			
General Inputs																						
A ICLK	1																					
4 iRST	0		<u>من م</u>			ه ک	<u>م الم الم الم الم الم الم الم الم الم ال</u>			_							_		<u>من ال</u>	ه ک		ر عنامه
- Instruction Fetch																						
n de s inst	32508100014	AD	21440001	08100014	00000000	014B602A	00000000		11800015	00000000	00046080	00000000		008C6820	00000000		8DAE00	8DAE0004	00000000		01EEC0	00000000
	22/00/00000	004	00400048	004000	00003000	01400050	00400054	00400059	0040005C	000000000	000000000	00000000	00400060	00800020	000000000	00400079	00400070	00/0700004	000000000	00400099	016600	00000000
	3211004000110	004	00400040	004000000	004000004	00400054	00400050	00400050	004000000	00400084	00400060	004000000	004000070	00400074	00402070	00400070	00400000	00402004	00400000	00400000	00400000	00400004
S_PC_4	32100400080	004	004000746	00406080	00400054	00400004	00406055	00400056	00400000	00406064	00400008	00400000	00405010	00400074	00400016	00400076	00400080	00400004	00400055	00400086	00406030	00400034
	2015-21440001	40	404500	21440001	00100014	00000000	01406034	00000000		11000015	00000000	00045000	00000000		000000000	00000000	کک	004500	00450004	00000000		015500
	32h214A0001	AD	ADAEoo	21440001	08106014	00000000	01486024	000000000		11806015	00000000	000A6060	00000000		00806620	00000000		SUAE00	8DAH0004	00000000		OTEECom
	61108	28		08	02	00				04	00							23		00		
s_ID_FUNCT	6'h01	00	04	01	14	00	2A	00		15	00				20	00			04	00		2A
s_ID_PC_4	32'h004000AC	004	004000A8	004000	004000B0	00400084	00400054	00400058	0040005C	00400060	00400064	00400068	0040006C	00400070	00400074	00400078	0040007C	00400080	00400084	00400088	0040008C	00400090
s_ID_rs_ADDR	5'd10	13	<u>ک ک</u>	10	0		10	0		12	0				4	0	کک	13	<u>ک</u>	0		15
■ s_ID_rt_ADDR	5'd10	15	14	10	16	0	11	0			کک	10	0		12	0	کک	14	15	0		14
s_ID_rd_ADDR	5'd0	0					12	0				12	0		13	0						24
s_ID_rs_DATA	32'd0	26850	0992	0			1	0		1	0				2685009	0		2685009	2685009	0		3
s_ID_rt_DATA	32'd0	-68	5	0			2	0				1	0		4	0		5	-68	0		5
- Execute																						
s_SEX_nAdd_Sub	1110	0					<u>م الم الم الم الم الم الم الم الم الم ال</u>	1	0		1	0								<u>م م</u>		
s_EX_shift_SEL	2110	0															کک					
s EX logic SEL	2110	0															کک					
s EX out SEL	2110	1	0				1	3	1		0	1				0	1		0		1	
🖬 🔶 s EX PC 4	32'h004000A8	004	004000A4	004000A8	004000AC	004000B0	004000B4	00400054	00400058	0040005C	00400060	00400064	00400068	0040006C	00400070	00400074	00400078	0040007C	00400080	00400084	00400088	0040008C
s EX rt ADDR	5'd14	0	15	14	10	16	0	11	0				10	0		12	0		14	15	0	
s EX rd ADDR	5'd0	0						12	0				12	0		13	0					
- s EX is DATA	32'd268500992	0	268500992		0			1	0		1	0				2685009	0		268500996		0	
s EX rt DATA	32'd5	0	-68	5	0			2	0				1	0		4	0		5	-68	0	
E ALLOUT	32/4268500996	0	2685009	2685009	1	0		1	ů.		1	0	4	0		2685009	ů.		2685009	2685010	i i	
- DMEM	52 0200300000		20030	20051		<u> </u>		_								2005000			20030	2003010		
A s DMEM mem WE	1111	0		1		0																
s DMEM DMEM DATA	32/15	5		<u> </u>	-68	-68									-68			-68				-68
S DMEM # DATA	-32168	0		-68	5	0				0				1	0		4	0			-68	
	22%10010000	000	00000000	10010000	10010004	00000001	00000000		00000001	0000000		00000001	00000000	00000004	0000000		10010004	0000000		10010004	10010009	0000000
S DMEM_ALGOOM	22/00/000004	000	000000000	00400044	00400048	00000000	0000000000	00400084	00000001	000000000	00400050	00000001	000000000	000000004	000000000	00400070	00400074	000000000	00400070	00400090	00400084	000000088
White Dark	321100400074	004	004000700	00406684	00406048	004000746	00400000	00400004	00400004	00406056	00400030	00400000	00406004	00400008	00400000	00400070	00406674	00406018	00400076	00400000	00400084	00400088
- Write Back	0																					
	0																					ت کے
s_Halt	0	004	00100000	00100010	00100014	00100010	00100010	00100000	100100001	00100051	00100050	00100050	100100000	00100001	00400000	00100000	100100030	00400074	00100070	00100070	00100000	00100001
s_wb_Pc_4	32'h004000A0	004	00400090	00400050	00400044	004000748	004000AC	00400080	100400084	00400054	00400058	00400050	100400060	00400004	00400068	00400060	00400070	00400074	00400078	00400070	00400080	00400084
s_WB_ALUOUT	32'd0	0		0	26885009	2685009		0						10	4	0		26655009	<u>a</u>		26815009	2685010
s_WB_RegWr	1h1		<u>0</u>	1	0	<u>ک</u>		0	6		کک		10	1						<u>ک</u>		ر کے
•	32'd0	0	1	0	2685009	2685009		0			10			10	4	10		2685009	10			
s_WB_RegWrAddr	5'd0	0			ک		10	0		.12	10				12	10	کک	.13	0		14	415

Bubble Sort Array Swap 4



Bubble Sort Array Swap 5

In the below screenshot, the instruction decoded in the ID stage following the cursor is the ori instruction, described by the s_ID_OPCODE signal being 0x0D. After this instruction, we can see that only one NOP instruction is decoded next as s_ID_INSTR is 0x00000000. This shows that we did not use 5 NOP instructions for every single signal. We updated our register file such that the register read would output the incoming memory that was being written before the clock cycle, which reduced our potential of

NOP's from 3 to 2 for any arithmetic or ALU operation. In other cases, like seen below, it can still help to reduce some NOP's from 2 down to 1, or even 1 to using none!

General Inputs						
siclk 🕺	1					
👍 iRST	0					
Instruction Fetch						
🖅 🔶 s Inst	32'h00000000	34050004	00000000	20080000	20A9FFFF	
	32'h00400018	00400014	00400018	0040001C	00400020	
s IF PC 4	32'h0040001C	00400018	0040001C	00400020	00400024	
Instruction Decode						
🖅 🔷 s ID INSTR	32'h34050004	34240000	34050004	00000000	20080000	
	6'h0D	0D		00	08	
	6'h04	00	04	00		
	32'h00400018	00400014	00400018	0040001C	00400020	
• s_ID_rs_ADDR	5'd0	1	0			
	5'd5	4	5	0	8	
	5'd0	0				
🖃 🔶 s_ID_rs_DATA	32'd0	268500992	0			
🖃 🔶 s_ID_rt_DATA	32'd0	0				
Execute						
🖅 🔶 s_EX_nAdd_Sub	1'h0	0				
🖅 🔶 s_EX_shift_SEL	2'h0	0				
🖅 🔷 s_EX_logic_SEL	2'h3	0	3		0	
• • • s_EX_out_SEL	2'h2	1	2		1	
	32'h00400014	00400010	00400014	00400018	0040001C	
🖅 🔶 s_EX_rt_ADDR	5'd4	0	4	5	0	
• s_EX_rd_ADDR	5'd0	0				
🖃 🔶 s_EX_rs_DATA	32'd268500992	0	268500992	0		
🖃 🔶 s_EX_rt_DATA	32'd0	0				
🖃 🔶 s_EX_ALUOut	32'd268500992	0	268500992	4	0	

Bubble Sort Data Flow NOP Proof

Based on the below screenshot, it can be seen that we are branching because the s_ID_branch_chk control signal is asserted to 1 after the cursor. After the branch instruction, it can be seen that the next decoded instruction is 0x00000000, which is a NOP. After the NOP, the program resumes with a regular instruction that is intended for the Bubble Sort algorithm. Since we moved our branch module from the ALU in the EX stage to the top level in the ID stage, we only need to wait for one cycle with a NOP to determine a branch, and in turn find the next PC.

siclk	1							
👍 iRST	0							
Instruction Fetch								
	32'h00000000	00000000		11800015	00000000	21080001	0810000B	00000000
	32'h00400060	00400054	00400058	0040005C	00400060	004000B4	004000B8	004000BC
	32'h00400064	00400058	0040005C	00400060	00400064	004000B8	004000BC	004000C0
Instruction Decode								
	32'h11800015	014B602A	00000000		11800015	00000000	21080001	0810000B
	6'h04	00			04	00	08	02
	6'h15	2A	00		15	00	01	0B
	32'h00400060	00400054	00400058	0040005C	00400060	00400064	004000B8	004000BC
	5'd12	10	0		12	0	8	0
	5'd0	11	0				8	16
	5'd0	12	0					
🖪 🔶 s_ID_rs_DATA	32'd0	3	0		0			
🖪 🔶 s_ID_rt_DATA	32'd0	3	0					
🖪 🔶 s_ID_branch_chk	1'h1	0			1	0		
🖃 🔶 s_ID_branch	1'h1	1			1			

Bubblesort Control Flow NOP Proof

[1.d] report the maximum frequency your software-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).

Our maximum frequency reported was 55.05 MHz.

- The critical path follows as:
 - 1. Data Memory
 - 2. Reg Address MUX
 - 3. Register File
 - 4. Branch module
 - 5. Prefetch Module to update PC

The longest path occurred in our Instruction Decode stage. The path followed decoding a given register, and passing that along to the branch module which determined if a branch should be made or not. Then, the PC finally got updated. We moved the branch conditional check outside of the ALU and into the ID stage to help prevent less NOP's in our test code and stalling later on. Now, we can see the tradeoff of having to wait for the register file contents for the RS and RT DATA to propagate and output from the register file before checking the branch conditions. For project 3, we could analyze the tradeoffs on reducing the CPI of branch instructions by 1 versus the added propagation delay after the register file in our instruction decode stage. It could be the case that the next slowest propagation is only slightly faster, in which case would lead to a much better benefit in keeping the conditional branch module in the ID phase, to reduce CPI.

[2.a.ii] Draw a simple schematic showing how you could implement stalling and flushing operations given an ideal N-bit register.

The following example depicts an N-bit register that has both stalling and flushing operations. To stall a pipeline register for one cycle, the i_stall input control signal will be asserted to 1. This means that each DFF in the pipeline register will not be written on the next positive edge clock cycle, instead HOLDING it's previous contents. To flush, we will MUX 0's in for every bit so that on the POSITIVE EDGE of the next clock cycle, the pipeline register will be filled with 0's and not affect any registers or memory.



[2.a.iii] Create a testbench that instantiates all four of the registers in a single design. Show that values that are stored in the initial IF/ID register are available as expected four cycles later, and that new values can be inserted into the pipeline every single cycle. Most importantly, this testbench should also test that each pipeline register can be individually stalled or flushed.

The following waveform depicts our testbench that instantiates all four pipeline register buffers in a single testbench. For this test, we only used one sample datapath signal that propagates throughout each pipeline register, which was PC_4. Initially, we began by buffering decimal 1 through the IF_ID_PC_4 pipeline register, and then waiting 4 clock cycles to confirm decimal 1 is propagated through each pipeline register. The order of the pipeline registers follows the stages between them, following IF_ID, ID_EX, EX_DMEM, and finally DMEM_WB. If we write a signal decimal 2 and hold that value through multiple clock cycles, 2's are confirmed to be filled through each pipeline register as expected. When we wrote decimal 4 for IF_PC_4, we drove the stall control for each pipeline register on the clock cycle; it was expected to write a new value for PC_4, indicating that the old contents of 0 were not written over. Finally, we simulated a condition that had two data hazards and a control hazard, which led to two stalls and then a flush. This was very beneficial to test since a sample set of instructions with a read/write data dependency and a flush was how we derived the logic for our data hazard module.

System													
🔷 sim:i_CLK	-No Data-												
♦ sim:i_RST	-No Data-												
— Data Inputs —													
	-No Data-	0 1 0)	2		0 4			0	4 8	12		
Stall Inputs													
sim:i_IF_ID_STALL	-No Data-												
sim:i_ID_EX_STALL	-No Data-												
sim:i_EX_DMEM_STALL	-No Data-												
sim:i_DMEM_WB_STALL	-No Data-												
sim:i_IF_ID_FLUSH	-No Data-												
sim:i_ID_EX_FLUSH	-No Data-												
sim:i_EX_DMEM_FLUSH	-No Data-												
sim:i_DMEM_WB_FLUSH	-No Data-												
Data Outputs													
	-No Data-	0 1	0	2		0	4		0	4 8		12	
• sim:s_EX_PC_4	-No Data-	0	1 0		2	0	4		0	4	0 8	0 12	
• sim:s_DMEM_PC_4	-No Data-	0	(1)()	2	0		4	0		4 0	8 0	12
	-No Data-	0	(1	0	(2	0			0)		(4)(0	8	0 12

Instruction	Format	Decode	Produce Execute	Produce Demem	Produce Writeback
add	R	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
addi	I	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
addiu	I	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
addu	R	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
and	R	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
andi	I	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
lui	I	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
lw	I	-	s_EX_ALUOut	s_DMEM_DMEM_DATA	s_RegWrData
nor	R	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
xor	R	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
xori	I	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
or	R	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
ori	I	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
slt	R	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
slti	I	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
sll	R	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
srl	R	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
sra	R	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
sw	I	-	s_EX_ALUOut	-	-
sub	R	-	s_EX_ALUOut	s_EX_ALUOut	s_RegWrData
subu	R	-	s_EX_ALUOut	s_EX_ALUOut	s_RegWrData
beq	I	s_ID_branch	-	-	-
bne	I	s_ID_branch	-	-	-
j	J	s_ID_IMM_EXT	-	-	-
jal	J	s_ID_IMM_EXT	s_EX_PC_4	s_DMEM_PC_4	s_RegWrData
jr	R	s_ID_IMM_EXT	-	-	-
bgez	I	s_ID_IMM_EXT	-	-	-
bgezal	I	s_ID_branch	s_EX_PC_4	s_DMEM_PC_4	-
bgtz	I	s_ID_branch	-	-	-
blez	I	s_ID_branch	-	-	-
bitzal	I	s_ID_branch	s_EX_PC_4	s_DMEM_PC_4	-
bltz	1	s_ID_branch	-	-	-
halt		s_ID_Halt,	s_EX_Halt,	s_DMEM_Halt,	s_WB_Halt,

[2.b.i] list which instructions produce values, and what signals (i.e., bus names) in the pipeline these correspond to.

[2.b.ii] List which of these same instructions consume values, and what signals in the
pipeline these correspond to.

Instruction	Format	Decode	Consume Decode	Consume Decode
add	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
addi	I	-	s_ID_rs_DATA	-
addiu	I	-	s_ID_rs_DATA	-
addu	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
and	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
andi	I	-	s_ID_rs_DATA	-
lui	I	-	s_ID_rs_DATA	-
lw	I	-	s_ID_rs_DATA	-
nor	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
xor	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
xori	I	-	s_ID_rs_DATA	-
or	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
ori	I	-	s_ID_rs_DATA	-
slt	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
slti	I	-	s_ID_rs_DATA	-
sll	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
srl	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
sra	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
SW	I	-	s_ID_rs_DATA	s_ID_rt_DATA,
sub	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
subu	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
beq	I	s_ID_branch	s_ID_rs_DATA	s_ID_rt_DATA,
bne	I	s_ID_branch	s_ID_rs_DATA	s_ID_rt_DATA,
j	J	s_ID_IMM_EXT	-	-
jal	J	s_ID_IMM_EXT	-	-
jr	R	s_ID_IMM_EXT	s_ID_rs_DATA	-
bgez	I	s_ID_IMM_EXT	s_ID_rs_DATA	-
bgezal	I	s_ID_branch	s_ID_rs_DATA	-
bgtz	I	s_ID_branch	s_ID_rs_DATA	-
blez	I	s_ID_branch	s_ID_rs_DATA	-
bltzal	I	s_ID_branch	s_ID_rs_DATA	-
bltz	I	s_ID_branch	s_ID_rs_DATA	-
halt		s_ID_Halt,	-	-

[2.b.iii] generalized list of potential data dependencies. From this generalized list, select those dependencies that can be forwarded (write down the corresponding pipeline stages that will be forwarding and receiving the data), and those dependencies that will require hazard stalls.

	EX RS ALU INPUT	EX RT ALU INPUT	DMEM DATA INPUT			
Producing Signal	s_EX_RS_DATA_MUX_FWD	s_EX_RT_DATA_MUX_FWD	s_EX_DMEM_DATA_MUX_FWD	Branch	JR	
s_EX_RS_DATA [31:0]	s_EX_RS_MUX_FWD_SEL = 00	NO	NO	NOT ADDRESSED	NOT ADDRESSED	EX
s_EX_RT_DATA_MUX [31:0]	NO	s_EX_RT_MUX_FWD_SEL = 00	s_DMEM_DATA_FWD_SEL = 00	NOT ADDRESSED	NOT ADDRESSED	DMEM
s_DMEM_ALUOut [31:0]	s_EX_RS_MUX_FWD_SEL = 10	s_EX_RT_MUX_FWD_SEL = 10	s_DMEM_DATA_FWD_SEL = 10	NOT ADDRESSED	NOT ADDRESSED	WB
s_RegWrData	s_EX_RS_MUX_FWD_SEL = 01	s_EX_RT_MUX_FWD_SEL = 01	s_DMEM_DATA_FWD_SEL = 01	NOT ADDRESSED	NOT ADDRESSED	
10 ₆₆				Would need to be forwared to the comparator module	Would need to forward both producing signals to jump adder	

[2.b.iv] global list of the datapath values and control signals that are required during each pipeline stage

	Control Signal											
Control Signal	IF (Instruction Fetch)	ID (Instruction Decode)	EX (Execute)	DMEM (Data Memory)	WB (Writeback)	Notes						
reg_WE_SEL	NO	s_ID_reg_WE_SEL	s_EX_reg_WE_SEL	NO	NO			Test				
Halt	NO	s_ID_Halt	s_EX_Halt	s_DMEM_Halt	s_WB_Halt			Top Level				
nZero_Sign	NO	s_ID_nZero_Sign	NO	NO	NO	Sign extender in ID stage		Fetch Logic				
ALUSrc	NO	s_ID_ALUSrc	s_EX_ALUSrc	NO	NO			ALU				
overflow_chk	NO	s_ID_overflow_chk	s_EX_overflow_chk	s_DMEM_overflow_chk	s_WB_overflow_chk	Overflow checked in last stage		Stall				
branch_chk	NO	s_ID_branch_chk	NO	NO	NO	Branch control in ID stage		Flush				
reg_DST_ADDR_SEL [1:0]	NO	s_ID_reg_DST_ADDR_SEL [1:0]	s_EX_reg_DST_ADDR_SEL [1:0]	NO	NO	Writeback in last stage		Forwarding				
reg_DST_DATA_SEL [1:0]	NO	s_ID_reg_DST_DATA_SEL [1:0]	s_EX_reg_DST_DATA_SEL [1:0]	s_DMEM_reg_DST_DATA_SEL [1:0]	s_WB_reg_DST_DATA_SEL [1:0]	Writeback in last stage						
PC_SEL[1:0]	NO	s_ID_PC_SEL[1:0]	NO	NO	NO							
						MUXed in ID stage and carried						
reg_WE	NO	s_ID_reg_WE	s_EX_reg_WE	NO	NO	through as regWr						
mem_WE	NO	s_ID_mem_WE	s_EX_mem_WE	s_DMEM_mem_WE	NO							
branch_SEL[2:0]	NO	s_ID_branch_SEL[2:0]	NO	NO	NO							
nAdd_Sub	NO	s_ID_nAdd_Sub	s_EX_nAdd_Sub	NO	NO		-					
shift_SEL[1:0]	NO	s_ID_shift_SEL[1:0]	s_EX_shift_SEL[1:0]	NO	NO		-					
logic_SEL[1:0]	NO	s_ID_logic_SEL[1:0]	s_EX_logic_SEL[1:0]	NO	NO							
out_SEL[1:0]	NO	s_ID_out_SEL[1:0]	s_EX_out_SEL[1:0]	NO	NO							
			Hazard Detection	1			1					
IF_ID_STALL	s_IF_ID_STALL	NO	NO	NO	NO		-					
ID_EX_STALL	NO	s_ID_EX_STALL	NO	NO	NO		-					
EX_DMEM_STALL	NO	NO	s_EX_DMEM_STALL	NO	NO							
DMEM_WB_STALL	NO	NO	NO	s_DMEM_WB_STALL	NO							
PC_STALL	s_PC_STALL	NO	NO	NO	NO		-					
IF_ID_FLUSH	s_IF_ID_FLUSH	NO	NO	NO	NO							
ID_EX_FLUSH	NO	s_ID_EX_FLUSH	NO	NO	NO							
EX_DMEM_FLUSH	NO	NO	s_EX_DMEM_FLUSH	NO	NO							
DMEM_WB_FLUSH	NO	NO	NO	s_DMEM_WB_FLUSH	NO							
			Forwarding									
EX_RS_DATA_FWD_SEL	NO	s_EX_RS_DATA_FWD_SEL	NO	NO	NO	Select line for ALU RS forwarding						
EX_RT_DATA_FWD_SEL	NO	s_EX_RT_DATA_FWD_SEL	NO	NO	NO	Select line for ALU RT forwarding						
DMEM_DATA_FWD_SEL	NO	s_DMEM_DATA_FWD_SEL	NO	NO	NO	Select line for DMEM Data forwarding						
LW_HAZARD_CHK	NO	s_LW_HAZARD_CHK	s_LW_HAZARD_CHK	NO	NO	indicator for SW in EX stage, indicating stall to hazard detection						

HW Pipeline Control Signals

			Our Datapath Signals				
Datapath Signal	IF (Instruction Fetch)	ID (Instruction Decode)	EX (Execute)	DMEM (Data Memory)	WB (Writeback)	Notes	
branch	NO	s_ID_branch	s_EX_branch	NO	NO		fetch
PC [31:0]	s_IF_PC [31:0]	NO	NO	NO	NO	Assign to nextInstrAddr	Instruction Decode
PC_4 [31:0]	s_IF_PC_4 [31:0]	s_ID_PC_4 [31:0]	s_EX_PC_4 [31:0]	s_DMEM_PC_4 [31:0]	s_WB_PC_4 [31:0]	Has to propogate through for writeback for jal to set r[31] = PC + 4	ALU
INSTR [31:0]	NO	s_ID_INSTR [31:0]	NO	NO	NO	Use given Instr signal for IF	RT DATA MUX
OPCODE [5:0]	NO	s_ID_OPCODE [5:0]	s_EX_OPCODE [5:0]	NO	NO		Sign Extender
FUNCT (5:0)	NO	s_ID_FUNCT [5:0]	s_EX_FUNCT [5:0]	NO	NO		Overflow
rt_ADDR [4:0]	NO	s_ID_rt_ADDR [4:0]	s_EX_rt_ADDR [4:0]	NO	NO	rt_ADDR needed in EX for forwarding	Data Memory
rs_ADDR [4:0]	NO	s_ID_rs_ADDR [4:0]	s_EX_rs_ADDR [4:0]	NO	NO	rs_ADDR needed in EX for forwarding	MUX
rd_ADDR [4:0]	NO	s_ID_rd_ADDR [4:0]	s_EX_rd_ADDR [4:0]	NO	NO	Needed for WB	
SHAMT [4:0]	NO	s_ID_SHAMT [4:0]	s_EX_SHAMT [4:0]	NO	NO		
IMM [15:0]	NO	s_ID_IMM [15:0]	NO	NO	NO	Not needed since sign extender in ID and turns into sign extended immediate 32 bit	
J_ADDR [25:0]	NO	s_ID_J_ADDR [25:0]	NO	NO	NO		
rs_DATA [31:0]	NO	s_ID_rs_DATA [31:0]	s_EX_rs_DATA [31:0]	NO	NO		
rt_DATA [31:0]	NO	s_ID_rt_DATA [31:0]	s_EX_rt_DATA [31:0]	s_DMEM_rt_DATA [31:0]	NO	DMemData used in DMEM stage	
ALUOut [31:0]	NO	NO	s_EX_ALUOut [31:0]	s_DMEM_ALUOut [31:0]	s_WB_ALUOut [31:0]	DMemAddr used in DMEM stage. When to assign for output? WB?	
rt_DATA_MUX [31:0]	NO	NO	s_EX_rt_DATA_MUX [31:0]	NO	NO	MUX for second ALU operand between RT DATA and EXT IMM	
IMM_EXT [31:0]	NO	s_ID_IMM_EXT [31:0]	s_EX_IMM_EXT [31:0]	NO	NO		
overflow	NO	NO	s_EX_overflow	s_DMEM_overflow	s_WB_overflow	Overflow should propogate to end of final stage (WB)	
DMEM_DATA [31:0]	NO	NO	NO	s_DMEM_DMEM_DATA [31:0]	s_WB_DMEM_DATA [31	Needed between EX/MEM buffer to writeback MUX	
			MUX'd signals				
RegWr	NO	NO	s_EX_RegWr	s_DMEM_RegWr	s_WB_RegWr	Final write signal, MUXXED out	
RegWrAddr [4:0]	NO	NO	s_EX_RegWrAddr [4:0]	s_DMEM_RegWrAddr [4:0]	s_WB_RegWrAddr [4:0]	MUX with rd_ADDR, rt_ADDR, and decimal 31 (\$ra reg)	
			MUX'd Forwarding signals				
RS_DATA_FWD_MUX [31:0]	NO	s_ID_RS_DATA_FWD_MUX [3	NO	NO	NO		
RT_DATA_FWD_MUX [31:0]	NO	s_ID_RT_DATA_FWD_MUX [31	NO	NO	NO		
DMEM_DATA_MUX_FWD [31:0]	NO	NO	s_EX_DMEM_DATA_MUX_FWD [31:0]	s_DMEM_DMEM_DATA_MUX_FWD [31:0]	NO		

HW Pipeline Datapath Signals

[2.c.i] list all instructions that may result in a non-sequential PC update and in which pipeline stage that update occurs.

beq- decode bne- decode j- decode jal- decode jr- decode bgez- decode bgezal- decode blez- decode blez- decode bltzal- decode bltz- decode

[2.c.ii] For these instructions, list which stages need to be stalled and which stages need to be squashed/flushed relative to the stage each of these instructions is in.

beq- if branch taken instruction IF_ID flush bne- if branch taken instruction IF_ID flush j- instruction IF_ID flush jal- instruction IF_ID flush bgez- if branch taken instruction IF_ID flush bgezal- if branch taken instruction IF_ID flush bgtz- if branch taken instruction IF_ID flush blez- if branch taken instruction IF_ID flush blez- if branch taken instruction IF_ID flush bltzal- if branch taken instruction IF_ID flush bltzal- if branch taken instruction IF_ID flush

Any time an unconditional or conditional branch is taken, we will flush the IF_ID register ONLY, since we have moved our branch checking to the ID stage. We load PC+4 after the branch and jump instructions, predicting that the branch will NOT be taken, and will NOT flush the instruction if it is not.

[2.d] implement the hardware-scheduled pipeline using only structural VHDL. As with the previous processors that you have implemented, start with a high-level schematic drawing of the interconnection between components.



[2.e.i] Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table. Show the Questasim output for the following test

Α 🔻	В	С	
Instruction	TEST CASE	Reasoning	
add	ALU_RS, ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX	
addi	ALU_RS	Forwarding From DMEM ALU out or WB Mux to EX	
addiu	ALU_RS	Forwarding From DMEM ALU out or WB Mux to EX	
addu	ALU_RS, ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX	
and	ALU_RS, ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX	
andi	ALU_RS	Forwarding From DMEM ALU out or WB Mux to EX	
lui	ALU_RS, ALU_RT	No data dependencies can only cause them	
lw	DMEM	Has delayed producing timing from all other commands	
nor	ALU_RS, ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX	
xor	ALU_RS, ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX	
xori	ALU_RS	Forwarding From DMEM ALU out or WB Mux to EX	
or	ALU_RS, ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX	
ori	ALU_RS	Forwarding From DMEM ALU out or WB Mux to EX	
slt	ALU_RS, ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX	
slti	ALU_RS	Forwarding From DMEM ALU out or WB Mux to EX	
sll	ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX	
srl	ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX	
sra	ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX	
sw	DMEM	Has different consuming points from ALU operations	
sub	ALU_RS, ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX	
subu	ALU_RS, ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX	
beq	Control/Jump	Stall for all data dependencies	
bne	Control/Jump	Stall for all data dependencies	
j	NA	No Data dependies	
jal	ALU_RS, ALU_RT	Can cause data dependencies	
jr	ALU_RS, ALU_RT	Needs stalling from data dependencies	
bgez	Control/Jump	Stall for all data dependencies	
bgezal	Control/Jump	Stall for all data dependencies	
bgtz	Control/Jump	Stall for all data dependencies	
blez	Control/Jump	Stall for all data dependencies	
bltzal	Control/Jump	Stall for all data dependencies	
bltz	Control/Jump	Stall for all data dependencies	
halt	NA	No data dependencies	

Test data forwarding and hazard detection here

Data Hazard Test Cases



ALU_RS Data Hazard Test Cases



ALU_RT Data Hazard Test Cases

🖬 Wave - Default						+ 2
Ms 🚺	gs					
General Inputs						
vsim3:/tb/MvMips/i 1			ihndnnhnr			hodoobodoo
📣 vsim3:/tb/MyMips/i 0					حاصات المتعل والمتع المتع	
- Instruction Fetch						
vsim3:/tb/MvMips/ 32'h20090005	2009 20090005	5 X X 120090005 X X	20090005 X X 20090005	5 1 X I X X08100006 I	I I I X08100006 X X X	1 108100006 X X X X X X X X X X X X X X X X X
vsim3:/tb/MyMips/ 32'hXXXXXXX	(004) 00400010	0 (00400010 ()	00400010 00400010	0 0 00400024	[]] 00400024 () ()	00400024 (0040
sim3:/tb/MyMips/ 32'hXXXXXXX	004	1 X X 100400014 ()	00400014 00400014	4 1 1 1 1 1 00400028	1 1 X 00400028 X X X	100400028 X X X X 0040
- Instruction Decode						
sim3:/tb/MyMips/ 32'hXXXXXXX	(000 1528FFFE	(1528FFFE)	1528FFFE	(11350001	11350001	11350001 (1135
sim3:/tb/MyMips/ 6'hXX	00 08 00 05	00 05 00	05 00 05	08 00 04 02	2 00 04 02 00	04 02 00 04
vsim3:/tb/MyMips/ 6'nXX	00 (05 (01)22 3E	00 22 3E 00 22	2 3E 00 22 3E	05 01 22 01 06	6 00 22 01 06 00 22	101 06 00 22 01
sim3:/tb/MyMips/ 32'hXXXXXXX	000) / / 00400010 / /	00400010 00400010	0 1)) (00400024	I I I (00400024 () ()	00400024))) ((0040
sim3:/tb/MyMips/ 5'dX	0 9	X0 X9 X0 X9	X0 X9	10 19)0 (9 0		0 9 0 9 0 9
vsim3:/tb/MyMips/ 5'dX	0 (9 (8	<u>Xo Xs</u> <u>(o Xs</u>	Xo X8	<u>19)8)0 (21</u> 16	6 0 18 10)21 (16 (0 (8	0 21)16 0 8 0 21
	0 9 31	<u> (0)</u>	31 0 9 31	10 9 21 0	9 21 0 (9	21 0 9 21 0
vsim3:/tb/MyMips/ 32'd0	0 5 5 4	1 (0 (4 (3 (0)3	2 0 2 1	1 0 1 0 5 4 4 0	4 0 4 3 3 0 3	0 3 2 2 0 2 0 2 1
vsim3:/tb/MyMips/ 32'd0	0)1)1		() (0 () 1	<u>11)0</u>	11 10 (1	10 X1 X0
Execute						
	0 1 0	1 0 1 0 1 0	1 0 1 0 1 0	1 0 1 0 1	0 1 0 1 0	1 0 1 0 1 0
vsim3:/tb/MyMips/ 2'hX	10					
sim3:/tb/MyMips/ 2hX	10					
	0 1 0	1 0 1	0 1 0		1 0 1	0 1 0
vsim3:/tb/MyMips/ 32'hXXXXXXX	0000000 000 0000		0000			
vsim3:/tb/MyMips/ 5'dX	10 19 8 10		8 0 8 0 8 0	8 9 8 0 21		8 0 21 16 0 8 0
vsim3:/tb/MyMips/ 5'dX	0 9 0	31 0 9 0 31 0	9 0 31 0 9 0	31 0 9 21 0	9 21 0	9 21 0 9 21 0
vsim3:/tb/MyMips/ 32'dX	10	4 0 4 0 3 0	3 0 2 0 2 0	11 10 11 10 4	0 4 0 13 0	3 0 2 0 2 0
vsim3:/tb/MyMips/ 32'dX	10	1 0 1 0 1 0		1 0	1 0	1 0 1 0
vsim3:/tb/MyMips/ 32'dX						
vsim3:////www.ps/	0					
vsim3:/tb/MyMips/ 1hX	0					
Vsims/w/MyMips/ 32hXXXXXXX		0000		0000		0000
VSIII3/W/MyMips/	10 19 8 19 10		1 0 19 10 131 0 19 10		19 21 0	9 21 0 9 21
VSIIIIS JUIMYMIPS/ 32 dX						
Now Or	s					
G 2 G Cursor 1 0 r	is 100 ns	200 hs 30	00 hs 400 hs	500 ns 600 ns	s 700 hs 80	ons 900 ns 100

Control/Jump Data Hazard Test Cases

Wave - Default	·														33																
<u>ku</u> .	Msgs																														
— General Inputs ——																															
/tb/MyMips/iCLK	1																														
/tb/MyMips/iRST	0																														
— Instruction Fetch ——																															
	32ħ3C011001	3C011	3.	3	8	2	AE080.	. 8	A	8E0	40)	8	8E0C0	. A	E090	86	0C0	0	000000) A		. 2	23FI	=00	0.	XXX	XXXX	X (5	A.	8E1F000)4
	32hXXXXXXXX	00400	. 0	0	0	0	004000		0	0040	000	0	004000		04000	. 00	4000	. 0	004000) (O.		. 0	0040	000	0	0040	00054	0	0.	0040004	14
 ∎- √ /tb/MyMips/s_IF_P	32hXXXXXXXX	00400	. (0.	<u>[0</u>	0	0	004000	0	0	0040	000]	0	004000	00	04000	00	4000	. 0	004000) [0.	. 10.	. (0	0040)00	10.	0040	00058	10	0.	0040004	18
- Instruction Decode -						_					r	_	1	_				-)r			<u> </u>				~	1					
tb/MyMips/s_ID_I	3211XXXXXXXX	100000	3.	<u>13</u> V o D	3	8	200800		8	AEU	90	8	8E0B0.	. 8	-0C0		090	8	0C100			. 8	2210	000	12	03E0	00008	10	5	AE1F000	04
to/mymips/s_ID	0 N X X		Vol	100	UF Vol	23	28	28	23	28	-	23	100		_	128		23	10	100		23	08			100		100	114	128	
to/wywips/s_iD	225777777777	00000		100		20	20		10	0040		00	1004000		-	7 00	4000	Yo	004000			10	0040	000	Yo	0040	0054	100	10	104	4
th/MyMips/s_ID	5/12	1000000	<u>, o</u>	<u>γυ</u> Υ1	10	1	004000	16	0	004	000]	0	1004000		J4000	100	4000	1,0	004000	<u>, , u</u> .	16		004		131	10041	0034	10	10.	16	
+ /th/MyMins/s ID r	5'dX	10	71-	16	Ŷ1	17	8	110	9		Υ	10	111	11	>	Ya		12	16	Yo	31		16		31	10				131	
The second secon	5'dX	10	12	10	2	0																									
+ /tb/MyMips/s ID r	32'd0	0				2)	0	268	50099	92									0		26	850099	92		4	4194	1360	4 0		2685009	996
	32'd0	0														23	5	3	268500)) 0		4	268	500	4	0				4194364	
- Execute																															
	1'hX	0																													
	2'hX	10		2	0	2	0																								
■→ /tb/MyMips/s_EX	2'hX	0			3	0																							_		
	2'hX	0	1		2	1	0														1	0			-				1	0	
• /tb/MyMips/s_EX	32hXXXXXXXX	000000	00	0	0	0	0 0.	. 0	0	0	0	0	0 0.	. 0	0	0.	. (0	0	0 0	0.	. 0.	0	0	0	0.	0	0000	00 0	0.	(0)00	0000
• /tb/MyMips/s_EX	5'dX	10		1	16	1	17 0	8		9	0	9	10 0	1	L 0	12	<u>(0</u>	9	12 0	10	5 0	31		0	16	31	0				
• /tb/MyMips/s_EX	5'dX	0		2	0	2	0							-	1.		v								-				-		
TO/MyMips/s_EX	32'0X	10			_		2 0		268	500	0	268	500 10	2	0	2.		268	500 0	Va		268	500	0	2.	4	0	4	0		
to/mymips/s_EX	3208			10	2	2	2 10		260	00		2	2 10	2	0	12	V.o.	230	3 0	2	. 0	260	4	0	2	4	0	14	10		
	32 UA				2	2	2 0	23:	208	300		2	2 10	2.	U	14		200	301		. 0	200	300	U		4	U				
Th/MyMins/s DM	2°hX	0						0										0				0									
+ /tb/MyMins/s_DM	1'hX	H _o					A. 4			1	0		1 0				-		1 0		-			0							
tb/MyMips/s DM	3277XXXXXXXX	000000	00		0	0	0 0.	. 0	0	0	0	0	0 0.	. 0.	0	0.	. 0	0	0 0	0.	. 0.	. 0	0	0	0	0	0	000000.	. 0.	0 0	0
• /tb/MyMips/s DM	5'dX	0			1	16	1 17	0	8	0	9	0	10) 0	11		12	0	1	2 0	31	0		31	0	16	31	0			
• /tb/MyMips/s DM	32'd1	1					8	1	0	5	235	1	235 1		108	0 1	3	1	3 2	35 1			2	4	1	4	0	1		1	
¢																															

DMEM Data Hazard Test Case

[2.e.ii] Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table. Show the Questasim output for the following test

Create a set of assembly programs that exhaustively tests control hazard avoidance. Minimally include one test program per control flow instruction. Then you should create a set of test programs that activate combinations of these instructions in the pipeline.

Instruction	Assembly File	Branch Taken	Branch Not Taken
j	j_controlHazard.s	Flush IF/ID	N/A
jal	jal_controlHazard.s	Flush IF/ID	N/A
jr	jr_controlHazard.s	Flush IF/ID	N/A
	beq_controlHazard.s	Flush IF/ID	No Action
beq	branch_explosion_controlHazard.s	Flush IF/ID	No Action
bne	bne_controlHazard.s	Flush IF/ID	No Action
bgez	bgez_controlHazard.s	Flush IF/ID	No Action
bgezal	bgezal_controlHazard.s	Flush IF/ID	No Action
bgtz	bgtz_controlHazard.s	Flush IF/ID	No Action
blez	blez_controlHazard.s	Flush IF/ID	No Action
bltzal	bltzal_controlHazard.s	Flush IF/ID	No Action
bltz	bltz_controlHazard.s	Flush IF/ID	No Action

Control Hazard Test Cases

The following waveforms demonstrate that when a control hazard occurs, the IF/ID pipeline register flushes all of its contents. The control hazard occurs when a branch is taken, since we will already be fetching the PC + 4 instruction after the branch or jump instruction. We designed our prediction module to not flush the IF/ID Buffer register when we do not branch, to help improve our CPI. In the following three waveforms, near the cursor, it can be seen that one NOP is propagated ONLY when a branch is taken.



J Instruction Control Hazard Test Case

- Conoral Inputs							
scherarinpus							-
vsim1:iRST 0							
- Instruction Fetch							
	0080009 20080008	10000004	20080009		2008000A	2008000B	
	0400038 00400020	00400024	00400028	00400038	0040003C	00400040	
	040003C 00400024	00400028	0040002C	0040003C	00400040	00400044	
- Instruction Decode -							
	0000000 20080007	20080008	10000004	0000000	20080009	2008000A	
•••• vsim1:s_ID_OPC 6'h00	08		04	00	08		
•••• vsim1:s_ID_FUNCT 6'h00	07	08	04	00	09	0A	
	0000000 00400020	00400024	00400028	0000000	0040003C	00400040	
	0						
	8		0		8		
	0						
• vsim1:s_ID_rs_D 32'd0	0						
	4	5	0		8	8	

BEQ Instruction Control Hazard Test Case

General Inputs							
sim2:iCLK 0							
sim2:iRST 0							
Instruction Fetch							
	20080009 (20080008	04110004	20080009		2008000A	2008000B	
	00400038 00400020	00400024	00400028	00400038	0040003C	00400040	
• vsim2:s_IF_PC_4 32'h0	0040003C (00400024	00400028	0040002C	0040003C	00400040	00400044	
Instruction Decode							
• vsim2:s_ID_INSTR 32'h0	20080007	20080008	04110004	0000000	20080009	2008000A	1
	08		01	00	08		
	07	08	04	00	09	0A	
• vsim2:s_ID_PC_4 32'h0	0000000 00400020	00400024	00400028	0000000	0040003C	00400040	
	0						
	8		17	0	8		
• vsim2:s_ID_rd_A 5'd0	0						
	0						
) (4	5	0		8	8	

BGEZAL Instruction Control Hazard Test Case

In our final test, we created the assembly file branch_explosion_controlHazard.s, which simulates 5 consecutive branch taken conditions in a row. Amazingly, this passed by the end of everything. As with our other validation and waveforms, the instruction that was fetched in the IF stage after the branch instruction would get flushed on every branch taken, as seen below in the following waveforms.

General Inputs																	
s iCLK	1																
IRST 🤣	0																
🖪 🔶 s_inst	32'hXXXXXXXX	10000	002		1000FFFD		10000002		1000FFFD	50000000	10000002	1000FFFD	50000000	XXXXXXXX			
	32'h0040001C	00400	000	00400004	0040000C	00400010	00400004	00400008	00400010	00400014	00400008	0040000C	00400014	00400018	004	0001C	
	32'h00400020	00400	0004	00400008	00400010	00400014	00400008	0040000C	00400014	00400018	0040000C	00400010	00400018	0040001C	004	0020	
— Instruction Decode —																	
	32'hXXXXXXXX	00000	000	10000002	00000000	1000FFFD	00000000	10000002	00000000	1000FFFD	00000000	10000002	00000000	50000000	xxx	XXXXX	
	6'hXX	00		04	00	04	00	04	00	04	00	04	00	14	XХ		
s_ID_FUNCT	6'hXX	00		02	00	3D	00	02	00	3D	00	02	00		xх		
• s_ID_PC_4	32'h0040001C	00000	0000	00400004	00000000	00400010	00000000	00400008	00000000	00400014	00000000	0040000C	00000000	00400018	004	0001C	
	5'dX	0													х		
	5'dX	0													х		
s_ID_rd_ADDR	5'dX	0				31	0			31	0				X		
	32'd0	0															
🖅 🔶 s ID rt DATA	32'd0	0													1		

Multiple Consecutive Branch Instructions Control Hazard Test Case

[2.f] report the maximum frequency your hardware-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).

Maximum frequency WITH forwarding: 48.66 MHz

Critical Path:

- 1. Data Memory (?)
- 2. MUX to ALU
- 3. ALU (Ripple Adder)
- 4. ALU (Ripple SLL)
- 5. EX/DMEM Pipeline Register (MUX to write)

We assumed that the critical path changed from our ID stage to our EX stage due to adding more multiplexers from the pipeline registers to the ALU inputs, leading to the increased timing delay before the ALU processes its operations. Similar to the single cycle, the ALU ripple carry adder led to a very high propagation delay, leading to our critical path again once we added more hardware for forwarding. Despite this, our CPI dropped to an average of 1.50 for our tests, which is a good tradeoff for decreasing the maximum frequency by around 6 MHz from a CPI average of near 2.1.