Everett Salley 4/18/2012 EEL4713 Assignment #5

# **Introduction:**

In this lab, we took the single-cycle MIPS processor we had designed in the previous labs and pipelined it. The main reason for pipelining is to improve the overall efficiency of the processor by allowing it to perform different stages of multiple instructions at the same time as opposed to one instruction per clock cycle. By sectioning off the execution of an instruction into multiple registered stages, the amount of time required to execute each stage is smaller than the overall time required to execute an instruction. As a result, one can theoretically increase the overall clock speed of the processor by evenly dividing the delay amongst the different pipeline stages. The tradeoff is that this method introduces numerous hazards and properly dealing with said hazards increases the overall design complexity. Below is a simplified picture showing the 5 pipeline stages in our MIPS processor:



The Instruction Fetch (IF) stage as its name implies fetches the instruction from the instruction memory using the program counter as a pointer to the address. The Instruction Decode (ID) stage decodes the instruction to generate the proper control signals and outputs the requested register data from the register file. The Execute (EX) stage is where the actual arithmetic required by the instruction is performed. The Memory Access (MEM) stage is where data is either written to or read from the data memory. Finally the Write Back (WB) stage is where data is written back to the register file.

As was mentioned before, the introduction of a pipeline also introduces numerous hazards that unless dealt with will break the operation of a program. The problem arises from the fact that there is now a latency from when an instruction is first fetched to when it is fully executed and results are stored. A hazard occurs whenever the program needs data that was a result of a previous operation, but has not yet been stored in either data memory or the register file. Since this data propagates throughout the pipeline registers, the most efficient way of

handling such a hazard is to simply forward it (ie don't wait until it is written to the register file) to the requested location. However, there are some instances where the data is simply not available to be forwarded (fetching data from memory for instance) and the only possible course of action is to "stall" the pipeline until the data is available. Stalling basically consists of accessing the same instruction repeatedly while asserting the control signals to zero for the redundant instructions. The result is that the processor continues executing the instructions before the stall, but performs no operation (nops) until the stall is over. The combination of forwarding and stalling makes the pipelined processor viable. The Hazard Table on the following page shows the various hazards encountered and how to deal with them.

My hazard table differed slightly from the one that was provided to use in class. For instance, my register file is written to on the falling clock edge, which means that the same data can be written to and read from a particular register in the same clock cycle. This completely eliminates the need for the forwarding muxes in the ifid stage. As a result, I only have to check for hazards over 2 stages and my hazard table is significantly reduced in many areas.

	R	AI	L	S	JR	LUI	В	JAL
R	1a: exmem.rd=idex.rs(1) =idex.rt(2) 2a: memwb.rd=idex.rs(3) =idex.rt(4)	1a: exmem.rd=idex.rs(1) 2a: memwb.rd=idex.rs(3)	1a: exmem.rd=idex.rs(1) 2a: memwb.rd=idex.rs(3)	1a: exmem.rd=idex.rs(1) =idex.rt(2) 2a: memwb.rd=idex.rs(3) =idex.rt(4)	1a: idex.rd=ifid.rs(stall) 2a: exmem.rd=ifid.rs(8)		1a: idex.rd=ifid.rs(stall) =ifid.rt(stall) 2a: exmem.rd=ifid.rs(8) =ifid.rt(10)	
AI	1a: exmem.rt=idex.rs(1) =idex.rt(2) 2a: memwb.rt=idex.rs(3) =idex.rt(4)	1a: exmem.rt=idex.rs(1) 2a: memwb.rt=idex.rs(3)	1a: exmem.rt=idex.rs(1) 2a: memwb.rt=idex.rs(3)	1a: exmem.rt=idex.rs(1) =idex.rt(2) 2a: memwb.rt=idex.rs(3) =idex.rt(4)	1a: idex.rt=ifid.rs(stall) 2a: exmem.rt=ifid.rs(8)		1a: idex.rt=ifid.rs(stall) =ifid.rt(stall) 2a: exmem.rt=ifid.rs(8) =ifid.rt(10)	
L	1a: ifid.rt=pre.rs(stall) =pre.rt(stall) 2a: memwb.rt=idex.rs(3) =idex.rt(4)	1a: ifid.rt=pre.rs(stall) 2a: memwb.rt=idex.rs(3)	1a: ifid.rt=pre.rs(stall) 2a: mernwb.rt=idex.rs(3)	1a: ifid.rt=pre.rs(stall) =pre.rt(stall) 2a: memwb.rt=idex.rs(3) =idex.rt(4)	1a: ifid.rt=pre.rs(stall) 2a: idex.rt=pre.rs(stall)		1a: ifid.rt=pre.rs(stall) =pre.rt(stall) 2a: idex.rt=pre.rs(stall) =pre.rt(stall)	
S								
JR								
LUI	1a: exmem.rt=idex.rs(11) =idex.rt(12) 2a: memwb.rt=idex.rs(13) =idex.rt(14)	1a: exmem.rt=idex.rs(11) 2a: memwb.rt=idex.rs(13)	1a: exmem.rt=idex.rs(11) 2a: memwb.rt=idex.rs(13)	1a: exmem.rt=idex.rs(11) =idex.rt(12) 2a: memwb.rt=idex.rs(13) =idex.rt(14)	1a: idex.rt=ifid.rs(17) 2a: exmem.rt=ifid.rs(18)		1a: idex.rt=ifid.rs(17) =ifid.rt(19) 2a: exmem.rt=ifid.rs(18) =ifid.rt(20)	
В								
JAL	1a: exmem.31=idex.rs(21) =idex.rt(22) 2a: memwb.31=idex.rs(23) =idex.rt(24)	1a: exmem.31=idex.rs(21) 2a: memwb.31=idex.rs(23)	1a: exmem.31=idex.rs(21) 2a: memwb.31=idex.rs(23)	1a: exmem.31=idex.rs(21) =idex.rt(22) 2a: memwb.31=idex.rs(23) =idex.rt(24)	1a: idex.31=ifid.rs(27) 2a: exmem.31=ifid.rs(28)		1a: idex.31=ifid.rs(27) =ifid.rt(29) 2a: exmem.31=ifid.rs(28) =ifid.rt(30)	

forward exmem.aludata to rs of alu						
forward exmem.aludata to rt of alu						
forward WBdata to rs of alu						
forward WBdata to rt of alu						
forward WBdata to rs of idex reg						
forward WBdata to rt of idex reg						
forward exmem.aludata to rs of idex reg						
forward exmem.aludata to rt of idex reg						
forward ui from exmem to rs of alu						
forward ui from exmem to rt of alu						
forward ui from memwb to rs of alu						
forward ui from memwb to rt of alu						
forward ui from memwb to rs of idex reg						
forward ui from memwb to rt of idex reg						

(17).	forward ui from idex to rs of idex reg
(18).	forward ui from exmem to rs of idex reg
(19).	forward ui from idex to rt of idex reg
(20).	forward ui from exmem to rt of idex reg
(21).	forward PC+4 from exmem to rs of alu
(22).	forward PC+4 from exmem to rt of alu
(23).	forward PC+4 from memwb to rs of alu
(24).	forward PC+4 from memwb to rt of alu
(25).	forward PC+4 from memwb to rs of idex reg
(26).	forward PC+4 from memwb to rt of idex reg
(27).	forward PC+4 from idex to rs of idex reg
(28).	forward PC+4 from exmem to rs of idex reg
(29).	forward PC+4 from idex to rt of idex reg
(30).	forward PC+4 from exmem to rt of idex reg

# **Book Questions: (USING THE BLUE EDITION)**

# 4.12.1)

-pipeline speed is derived by taking the largest latency found in any of the 5 stages -non-pipelined speed is equivalent to the sum of all latencies

a) pipelined:	500ps	b) pipelined:	200ps
non-pipelined:	1650ps	non-pipelined	: 800ps

# 4.12.2)

-for a non-pipelined processor, it takes 1 cycle to fully execute an instruction, therefore the instruction latency is simply the same as the clock time -for a pipelined processor it takes 5 cycles to fully execute any given instruction, therefore the total instruction latency is 5 times the clock cycle time

a) pipelined: $5*500 = 2$	.500ps	b) pipelined:	5*200 = 1000ps
non-pipelined: 1650ps		non-pipelined	: 800ps

# 4.12.3)

-To increase the clock speed one would ideally split up the stage with the highest latency.

a) MEM stage;	new clock cycle = $400$ ps
b) IF stage;	new clock cycle = $190$ ps

#### 4.20.1)

There are three possible data dependencies here

i) (RAW) Instr needs to read data from register file before it is written

- ii) (WAW) Instr writes to reg that was previously written to
- iii) (WAR) Instr writws to reg that was previously read from
- a) Instr1 & Instr2 (WAR) Instr3 & Instr4 (WAR) Instr1 & Instr3 (RAW) Instr2 & Instr3 (RAW) Instr2 & Instr4 (RAW) Instr3 & Instr4 (RAW) Instr1 & Instr3 (WAW)
- b) Instr1 & Instr2 (WAR) Instr1 & Instr2 (RAW) Instr1 & Instr3 (WAR) Instr1 & Instr3 (WAR) Instr2 & Instr3 (WAR) Instr2 & Instr4 (WAR) Instr3 & Instr4 (WAR) Instr3 & Instr4 (RAW)

### 4.20.2)

For the 5 stage pipeline we only have to worry about read after write. Most cases can be avoided by using forwarding but some instances like a load followed by a read are unavoidable hazards. Therefore, part a has no hazards to worry about (assuming the reg file is falling edge triggered. part b has one data hazard between instr 3 and 4.

### 4.24.1)

-accuracy is determined by the number of correct branches taken divided by the total number of branch decisions

a) always take:75% (3/4) always nt: 25% (1/4)
b) always take:60% (3/5) always nt: 40% (2/5)

# 4.24.2)

-starts off in predict branch not taken state and moves around state machine depending on correctness of decisions. Considering only the first 4 branch decisions...

a) 0% (all predictions are incorrect)b) 25% (3rd prediction is correct)

#### 4.24.3)

-eventually a steady state is reached in the prediction pattern such that it begins every iteration in the same state

a) eventually it settles in the top left state and the accuracy is 75% pattern is T,T,NT,T -> correct, correct, incorrect, correct

b) eventually it settles in the bottom right state and the accuracy is 40% pattern is T,T,T,NT,NT -> incorrect, correct, incorrect, incorrect, incorrect

### **Demonstration of Instructions:**

Many changes were made to the original MIPS processor to make it pipelined so it is important to show that it still functions as expected. This section is dedicated to showing the complete functionality of all 29 of the base MIPS instructions by individually testing each one (in some cases more than once) with custom test programs. The first such test program is designed to examine all of the logical instructions in the MIPS instruction set and the code is given below:

	\$1, \$0, 0x2568
ori	\$1, \$1, 0xABEF
lui	\$2, \$0, 0x3A97
ori	\$2, \$2, 0x5BE0
andi	\$3, \$1, 0x5BE0
nor	\$4, \$1, \$2
or	\$5, \$2, \$1
and	\$6, \$1, \$2
sll	\$7, \$1, 8
sll	\$7, \$1, 16
srl	\$8, \$1, 8
srl	\$8, \$1, 16

The first instructions to be examined are the load upper immediate, OR immediate, and the AND immediate instructions because they are so widely used for initializing registers. The annotated waveform provided on the next page shows the execution of the first five instructions in this program which initializes the registers. Now would be a good time to point out the differences that the pipeline introduces to interpreting the waveform results. In the non-pipelined version the results of any instruction was available on the clock cycle immediately following the instruction. Here that is not the case as there is a multicycle delay from when the instruction is fetched to when it is fully executed. We can examine the first instruction in more detail to see how this works.

In clock cycle 1, the first instruction is being fetched from the instruction memory (instruction fetch stage). In the next clock cycle, the instruction signal (located in the instruction decode stage) now displays the instruction that was previously fetched which is then decoded and sent to the register file. Clock cycle 3 is the execute stage, where the data fetched from the register file is usually (not in this case) sent through the ALU for some sort of computation. Clock cycle 4 is the memory access stage where data is either written to, or read from data memory. Finally clock cycle 5 is the write back stage where the data is written to the destination register in the register file. Notice that in my design I used a falling edge triggered register file, so the data is stored to the register file, and thus available for reads BEFORE the next rising edge. Thus, whenever an instruction appears in the instruction field in a waveform, its results will be written to the register file after 2 rising clock edges.



The next waveform shows the operation of the basic logic instructions AND, OR, and NOR. All three cases use the input values 0x2568ABEF and 0x3A975BE0.

NOR \$4, \$1	, \$2	OR \$5	, \$2, \$1	AND \$	6, \$1, \$2				
< dk	1								
🔶 memclk	1								hп
-🔷 pc_addr	004000	0.0400028	0040002C	00400030	00400034	00400038	0040003C	00400040	004000
-🔶 mif_address	13	DA	ОВ	0C	0D	0E	0F	10	11
-🔶 instruction		00222027	00412825	00223024	FFFFFFF				00013/
-🔷 reg_out(0)		00000000							
-🔷 reg_out(1)	2568AB	2568ABEF							
-🔷 reg_out(2)	3A975B	3A975BE0							
		00000BE0							
-🔷 reg_out(4)	C00004	0000000			C0000410				
-🔷 reg_out(5)	<b>3FFFFBI</b>	0000000				SEFFFBER			
-🔷 reg_out(6)		0000000					(20000BE0	)	
		0000000							
-🔷 reg_out(8)		00000000							
-🔷 dmem_out	FFFFAA	FFFFAAAA			00000000			FFFFAAAA	
-🔷 alu_result	ABEF00	0000000	C0000410	3FFFFBEF	20000BE0	00000000			
			esult of NOR operation itten to register		Result of OR operation written to register			It of ANE ation writ	

The next waveform shows the operation of the shift left logical left and right instructions. In every case it operates on the value 0x2568ABEF. Shifts of 8 and 16 are performed in both directions to be thorough.



The next program investigates every iteration of the set less than instruction. It basically works by testing the unsigned and signed version of the set less than instruction with the input -1 and 1. In the case of the signed operation, the -1 is interpreted as being less than 1 which is true and thus results in the destination register getting set to 1. In the case of the unsigned operation, the -1 is interpreted as approximately 4 billion which of course is not less than 1. Thus for the unsigned operation, the result is false and the destination register is set to 0.

The code for this test and waveform is provided below:

lui	\$1, \$0, 0xFFFF
ori	\$1, \$1, 0xFFFF
ori	\$2, \$0, 1
slt	\$3, \$1, \$2
sltu	\$3, \$1, \$2
slti	\$3, \$1, 1
sltiu	\$3, \$1, 1

load 0xFFF load 1 to reg	FFFFF	to reg 1	sit \$3, \$1, ests if -1 ⊲ rue	\$2 tes	u \$3, \$1, sts if billion < se	slti	\$3, \$1, 1 s if -1 < 1	tests	llion < 1		
🔶 dk	1										
memclk	0		ĻΠΠΓ			ĮΠΠΓ			ĮΠΠΓ	ļΠΠΓ	
-🔷 pc_addr		00 20400004	00400008	<u>0040000</u>	100400010	Ú.		<u>004000 C</u>	00400020	00400024	00400028 00
-🔷 mif_address	11	00 01	02	03	04	05		07	08	09	0A 0B
-🔶 instruction	0000000	XX.<13C01FFFF	3421FFFF	34020001	0022182A	0022182B	28230001	2C230001	00000000	Lunci.	an od otl
-🔷 reg_out(0)	0000000	00000000						ci	gned stl,		gned stl,
-🔷 reg_out(1)	FFFFFF	00000000			FFFF0000	FFFFFFF				resu	lt is false 📲
-🔷 reg_out(2)	0000000	00000000					000000001		sult is tru		
-🔶 reg_out(3)	0000000	00000000		V	alue is 1			00000001	00000000	0000000	100000000
	FFFFAA	00000000	FFFFAAAA			00000000	FFFFAAAA				
-🔶 alu_ia	0000000	xx 00000000		FFFF0000	00000000	FFFFFFFF	FFFFFFF			00000000	
-🔶 alu_ib	0000000	xx 00000000		0000FFFF	00000001	00000001				00000000	
-🔶 alu_result	0000000	00000000		FFFFFFFF	00000001	00000001	00000000	0000001	00000000	00000000	
		unsigned = signed = -1	~4 billion		signed result is	stl,	unsigne result is	d stl,			

The next test will demonstrate all of the branch instructions functioning correctly. For the sake of thoroughness, it shows both when a branch is taken and when it is not. It basically works by first loading values to be compared into registers \$t0, \$t1 and \$t2. It then proceeds by showing a case of BEQ not branching, followed by a case where it does branch. The same is done for the BNE instruction. It finally ends by looping infinitely. The code and waveform are provided below:

main:	ori	\$t0, \$0, 50
	ori	\$t1, \$0, 100
	ori	\$t2, \$0, 50
	beq	\$t0, \$t1, beq1
	beq	\$t0, \$t2, beq2
beq1:	ori	\$t3, \$0, 0x00FF
beq2:	ori	\$t3, \$0, 0x00AA
	bne	\$t0, \$t2, bne1
	bne	\$t0, \$t1, loop
bne1:	ori	\$t3, \$0, 0x00FF
loop:	bne	\$t0, \$t1, loop

Load Valu registers fo		beq \$t0, \$t2, beq2 condition is true and branch should be taken	bne \$t0, \$t2, bne1 condition is false, branch should not be taken	condition is true,		
<ul> <li>dk</li> <li>memclk</li> <li>pc_addr</li> <li>mif_address</li> <li>instruction</li> </ul>	1 1 0040000 00	04 05 06		07 08 09	400024 00400028 0040002C 9 0A 08 5090001 0340800FF 7.509FFFF	00400028 0040002C 00400030 0 00400028 0050002C 00400030 0 004 008 00C 00 005000000 11509FFFF 00000000
<ul> <li>reg_out(0)</li> <li>reg_out(1)</li> <li>reg_out(2)</li> <li>reg_out(3)</li> <li>reg_out(4)</li> <li>reg_out(5)</li> </ul>	000000 0000000 000000 0000000 000000 000000	If correct Bra taken, none these instruc should exec	ranch is e of lf cor uctions is tak	rrect branch ken, should	If correct branch is taken this instruction will	infinite loop
<ul> <li>reg_out(6)</li> <li>reg_out(7)</li> <li>reg_out(8)</li> <li>reg_out(9)</li> <li>reg_out(10)</li> <li>reg_out(11)</li> </ul>	BABBBB         0000000         BABBBB         0000000         BABBBB         BABBBB         0000000         BABBBB         BABBBB         0000000         BABBBB         BABBBB         0000000         BABBBB         BABBBB         BABBBB         BABBBB         BABBBB         BABBBB         BABBBB         BABBBBB         BABBBB         BABBBBB         BABBBBB         BABBBB         BABBBB <td< td=""><td>10000032 )00000064 )00000064</td><td>)00000032</td><td></td><td></td><td></td></td<>	10000032 )00000064 )00000064	)00000032			

The next test examines all three of the jump instructions. I utilize ori instructions as a way of keeping track of where the program is. Correctly functioning code should write a sequence of A, B, and C to register 1 with a value of FFFF indicating an error occurred. It starts by calling the jump instruction which skips to a JAL. The JAL correctly writes the return address to \$ra but instead of using \$ra, I use my own return address in register \$2 equal to the "loop" label. The instruction correctly executes and the program proceeds to loop infinitely. The code and waveform are provided below:

```
main:
       ori $1, $0, 0xAAAA
              target
       i
       ori $1, $0, 0xFFFF
       ori $1, $0, 0xFFFF
       ori $1, $0, 0xFFFF
       ori $1, $0, 0xFFFF
target: ori $1, $0, 0xBBBB
       jal dest
       ori $1, $0, 0xDDDD
       ori $1, $0, 0xFFFF
       ori $1, $0, 0xFFFF
loop: beq $0, $0, loop
dest:
      lui $2, $0, 0x1000
       ori $2, $2, 0x002C
       ori $1, $0, 0xCCCC
       jr $2
```

Load value reg 1 as a n	AAAA arker	into	la bunc		uctions	BBE	d value 3B into 1 as a ker	Load va 0x1000 into reg	002C	Load value CCCC into 1 as marke	reg 🕻	IR using 2 as add				
<ul> <li>dk</li> <li>memclk</li> <li>pc_addr</li> <li>mf_address</li> <li>instruction</li> <li>reg_out(0)</li> <li>reg_out(1)</li> </ul>	344200: 000000	00 XX	3401AAAA 000		06)	07	0C10000C		0D	200400038 Vo= 3442002C	0F 3401CCCC	10	00400044 (11 00000000		10000030 0C 1000FFFF	10000034 0D 3C021000
	100000 004000	00000	000							000400020		1000000	0 (1000002			
- dmem_out - du_ia - alu_ib - alu_result		хх хх	00000000 00000000	FFFFAAAA 00000AAAA 00000AAAA		00000000		00000000			10000000 0000002C 1000002C			00000000	FFFFAAAA	
<pre> ifid_ld     pc_write</pre>	1						next a	addr 0C. ddr value 0040002	(addr		shifte	00002C v d approp s addr of	riately	and reg	n Loops \$1 neve /ritten to	r had

The next test demonstrates all of the arithmetic operations. It basically loads registers 1-4 with values and uses them throughout the rest of the code to perform operations. It should be noted that functionally there is no difference between ADD and ADDU, or SUB and SUBU. The only difference is that the signed versions can set the overflow flag, while the unsigned versions cannot. We do not deal with the overflow flag in this class however, so these differences are of no consequence to us. The code and waveform are given below:

ori	\$1, \$0, 1
ori	\$2, \$0, 2
ori	\$3, \$0, 3
ori	\$4, \$0, 4
add	\$5, \$2, \$3
addu	\$6, \$3, \$3
addi	\$7, \$1, 6
addiu	\$8, \$3, 5
sub	\$9, \$4, \$1
subu	\$10, \$1, \$4

#### ORI \$4, \$0, 4 ADDU \$6, \$3, \$3 ADDIU \$8, \$3, 5 SUBU \$10, \$1, \$4 ORI \$2, \$0, 2 ORI \$3, \$0, 3 ADD \$5, \$2, \$3 ADDI \$7, \$1, 6 SUB \$9, \$4, \$1 ORI \$1, \$0, 1 ΠГ 040000C\_00400010\_00400014\_00440018\_004001C\_00400202\_00400024\_00400025\_0040002C\_00400030\_00400034 00400004 00400008 004000 00 01 02 03 04 05 06 07 08 09 0A 08 XX...(34010001)34020002 34030003 (34040004) 00432820 00633021 120270006 124680005 100814822 00245023 00000000 loc structio eg\_out(0) 000000000 00000001 00000002 eg\_out(2) eg\_out(3) 00000003 eg\_out(4) (00000004) eg\_out(5) 00000005 00000006 eg\_out(6) eg\_out(7) 2 + 3 = 500000000 (00000007) 3 + 3 = 6eg\_out(8) 00000008 1 + 6 = 700000003 000000000 3+5=8 4 - 1 = 3FFFFAAAA JOOOCOOO 000 000000000 FFFFAAAA 0000004 0000001 000000 1 - 4 = -3 alu\_ia 000000 XX... 00000000 00000002 0000003 00000001 0000003 000000 xx... 00000000 0000001 00000002 00000003 00000004 00000001 00000004 00000000 alu\_ib 00000003 00000005 00000003 00000006 00000001 00000002 00000003 00000004 00000005 0000006 0000007 0000008 0000003 FFFFFFD 00000000

10D

FFFFFFD

The final test examines the load and store operations. Stores are tested by using store word, store half word, and store byte to store the quantity 0xABCD1234 to data memory. They are then recovered using the load word instruction. Loads are tested by first storing the quantity 0xBAD2BEEF to memory. The value is then recovered using the load word, load half word, and load byte instructions. The code is provided below:

lui	\$1, \$0, 0xABCD
ori	\$1, \$1, 0x1234
lui	\$2, \$0, 0xBAD2
ori	\$2, \$2, 0xBEEF
lui	\$3, \$0, 0x1000
SW	\$1, 0(\$3)
sh	\$1, 4(\$3)
sb	\$1, 8(\$3)
lw	\$4, 0(\$3)
lw	\$4, 4(\$3)
lw	\$4, 8(\$3)
SW	\$2, 0(\$3)
lw	\$4, 0(\$3)
lhu	\$4, 0(\$3)
lbu	\$4, 0(\$3)

# Arithmetic test

The waveform below shows the test for the store operations:

Store	uard				-	-					
Store v Store F		rd									
		te using	IWO	location		) locatior		@ locatio	n =		
				as sent		as sent		vas sent			
	01234	4	300 000	as sem	SITW	as sem	30 1	vas sem			
clk	0										
pc_addr	004000!	00400018	0040001C	00400020	0400024	00400028	0040002C	00400030	00400034	00400038	004
mif_address	14	06	17	108	09	0A	ОВ	0C	0D	)OE	OF
instruction				A0610008	8C640000	8C640004	8C640008	AC620000	8C640000	94640000	906
reg_out(1)	ABCD12	ABCD1234									
reg_out(2)	BAD2BE	BAD20000	BAD2BEEF								
reg_out(3)	100000(	00000000		10000000							
reg_out(4)	000000E	00000000						JABCD1234	4 100001234	10000003	4D)
alu_rs_sel	0	0	3	4	<u>10</u>						
alu_rt_sel	0	0					1		<u> </u>	2	1
alusrc_ex	0						1				
alu_ia	000000(	00000000	10000000	10000000	10000000						
alu_ib	000000(	00000000		00000004	00000008	0000000	00000004	80000008	00000000	/	
alu_result	000000(	00000000	10000000	10000004	10000508	10000000	1000000+	10000008	10000000		
exmem_flush	0										
idex_flush	0										
ifid_ld	1	Entire w	ord reco	vered	Half of	word red	covered	Single	byte rec	overed	
		from me				emory;			emory;		
				e word		that half	word			ile hvte	
		shows that whole word was stored			was st		nord	shows that single byte was stored			
		1143 3101	ou -		1143 30	orou		Wd5 Storeu			

The waveform below shows the test for the load operations:

store enti 0xBAD28 memory			wor	d entire d from nory		from	fro	ad byte om emory			
dk	1						1				
pc_addr	0040006	0040	0030	004 <mark>10034</mark>	00400038	3 0040	0030	00400040	0040004	4 00400048	0040004
mif_address	1B	JOC		OD \	JOE /	(OF		10	11	12	13
instruction	000000(	AC62	0000	8C640000	9464000	9064	0000	00000000			
reg_out(1)	ABCD12	ABCD12	34								
reg_out(2)	BAD2BE										
reg_out(3)	100000(										
reg_out(4)	000000E	0)AB	CD 1234	000012	34 000000	34		BAD2BEE	1-10000B	EEF 000000E	
alu_rs_sel	0	0									
alu_rt_sel	0	1		0	2	1	_/				
alusrc_ex	0										
alu_ia	000000(	100000	00				<u> </u>		1000000	0	
alu_ib	000000(	00000	8000	00000000					000000	0	
alu_result	000000(	10000	8000	10000000					<mark>/0000000</mark>	0	
exmem_flush	0					<u> </u>		/			
idex_flush	0									_	
ifid_ld	1			l	entire wo baded sh .W works	nows	lo	alf word aded sh HU work		single by loaded s LBU wo	hows

# **Hazard Detection and Forwarding:**

As was discussed previously, without forwarding and stalling the pipelined processor would not function as expected. Therefore it is very important that the processor hold up to rigorous testing of its hazard detection and forwarding capabilities. With this in mind a test program was provided which examines many of the instances in which hazards occur. The decoded assembly is provided below:

1	lui \$5, 0x0302
2	ori \$5, \$5, 0x0100
3	ori \$6, \$0, 2312
4	lui \$29, 0x1000
5	sw \$5, 0(\$29)
6	sh \$6, 4(\$29)
7	ori \$6, \$0, 2826
8	sh \$6, 6(\$29)
9	lui \$30, 4096
10	ori \$30, \$30, 8
11	add \$15, \$30, \$0
12 notmain:	lbu \$18, 0(\$29)
13	lbu \$17, 4(\$29)
14	add \$16, \$17, \$18
15	slti \$10, \$16, 11
16	beq \$10, \$0, case2
17	add \$17, \$16, \$18
18	j case3
19 case2:	sub \$17, \$16, \$18
20 case3:	slti \$10, \$17, 11
21	beq \$10, \$0, case4
22	add \$18, \$17, \$16
23	j end
24 case4:	sub \$18, \$16, \$17
25 end:	sb \$18, 0(\$fp)
26	addi \$30, \$30, 1
27	addi \$29, \$29, 1
28	bne \$29, \$15, notmain
29	ori \$19, \$0, 8
30	subu \$29, \$29, \$19
31	lw \$1, 0(\$29)
32	lw \$2, 4(\$29)
33	lw \$3, 8(\$29)
34	lw \$4, 12(\$29)
35 here:	j here

The rest of the report will be annotated waveform simulations of this program which point out all of the major data hazards and how they are handled.





Instructions 7 - 11 contains some more instructions where forwarding solves the issue

								<u> </u>			
	> dk	0									
	pc_addr	004000!	00400018	0040001C	00400020	00400024	00400028	0040002C	00400030	00400034	<u>100</u>
	mif_address	15	06	07	08	09	0A	0B	0C	0D	)OE
	instruction	114000(	A7A60004	34060B0A	A7A60006	3C1E1000 <	37DE0008	03C07820	93B20000	93B10004	02
	reg_out(1)	000000(	00000000								
	reg_out(2)	0000000	00000000	ori stor	ed to \$6		UI to \$30		ori store t	o \$30	
-	reg_out(3)	0000000	00000000	followe		fo	llowed b		followed		
	reg_out(4)		00000000		that valu		ri using \$	· · · · · · · · · · · · · · · · · · ·	using \$3(		
-4	reg_out(5)		03020100	to men			r aonig v		using \$50		
-4	reg_out(6)		00000908	to men			00000B0A				Ē
-4	reg_out(10)		00000000								F
	reg_out(15)		00000000								Б
_	rea_out(16)	10000000	00000000						fonward	data	
	reg_out(16)		00000000						forward		
	reg_out(17)	0000000	00000000		forwar	d = fo	rward da	ta =	from me	m	
	reg_out(17)	000000( 000000(	00000000 00000000		forwar data fr		rward da om mem		from me stage to	m	
	reg_out(17) reg_out(18) reg_out(19)	000000( 000000( 000000(	00000000 00000000 00000000	¥10000000	data fr	om fro		stage	from me	m	
	reg_out(17) reg_out(18) reg_out(19) reg_out(29)	000000( 000000( 000000( 100000(	00000000 00000000 00000000 00000000	<u>×10000000</u>	data fr	om fro	om mem	stage	from me stage to ALU	em Rs of	
	reg_out(17) reg_out(18) reg_out(19) reg_out(29) reg_out(30)	000000( 000000( 000000( 100000(	00000000 00000000 00000000 00000000 0000		data fro mem s	om fro	om mem Rs of Al	stage _U	from me stage to ALU	em Rs of (10000008	
	reg_out(17) reg_out(18) reg_out(19) reg_out(29) reg_out(30) alu_rs_sel	000000( 000000( 000000( 100000(	00000000 00000000 00000000 00000000 0000	4	data fr	om fro tage to	om mem Rs of Al	stage _U	from me stage to ALU	em Rs of	
	reg_out(17) reg_out(18) reg_out(19) reg_out(29) reg_out(30) alu_rs_sel alu_rt_sel	000000( 000000( 000000( 100000(	00000000 00000000 00000000 00000000 0000		data fro mem s	om fro tage to	om mem Rs of Al	stage _U	from me stage to ALU	em Rs of (10000008	
	reg_out(17) reg_out(18) reg_out(19) reg_out(29) reg_out(30) alu_rs_sel alu_rt_sel alusrc_ex	0000000 0000000 1000000 1000000 0 0 1 1	00000000 00000000 00000000 00000000 3 0	4	data fro mem s	om fro tage to	om mem Rs of Al	stage _U 3	from me stage to ALU	em 9 Rs of <u>)10000008</u> 0	
	reg_out(17) reg_out(18) reg_out(19) reg_out(29) reg_out(30) alu_rs_sel alu_rt_sel	0000000 0000000 1000000 1000000 0 0 1 1	00000000 00000000 00000000 00000000 0000	4	data fro mem s	om fro tage to	om mem Rs of Al	stage _U	from me stage to ALU	m Rs of (10000008	
	reg_out(17) reg_out(18) reg_out(19) reg_out(29) reg_out(30) alu_rs_sel alu_rt_sel alusrc_ex	0000000 0000000 1000000 1000000 0 0 1 000000	00000000 00000000 00000000 00000000 3 0 0 10000000	4 0 10000000	data fro mem s	om fro tage to	om mem Rs of Al	stage U 3 10000000	from me stage to ALU	em 9 Rs of <u>)10000008</u> 0	
	reg_out(17) reg_out(18) reg_out(19) reg_out(29) reg_out(30) alu_rs_sel alu_rt_sel alusrc_ex alu_ia	0000000 0000000 1000000 1000000 0 0 1 000000	00000000 00000000 00000000 00000000 3 0 0 10000000	4 0 10000000	data fro mem s o	om fro tage to 1 1 10000000	om mem Rs of Al 0 0 00000000	stage U 3 10000000	from me stage to ALU 10000000	em 9 Rs of <u>)10000008</u> 0	

Instructions 13 - 16 contains the first hazard that requires a stall to fix. Note that some forwarding is ignored due to the fact that the stall flushes the control lines.

Load to \$	617 foll	owed	- 🔏 🥵 (		∣ ∰ <mark>sl</mark> th	ti result to en use \$ anch cor	o \$10, 10 for			
🐤 dk	0									
🔶 pc_addr	0040006	00400030	00400034 🤇	00400038		1-0400/3C	00400040	00400044	00400048	0040004C
🔶 mif_address	1A	loc 🔪	0D	I0E		IOE	10	111	12	13
instruction	23DE00	93B20000 🤇	193B10004	02328020		[2A0A000B	11400002	02128820	08100013	0000000 1
reg_out(1)	0000000	00000000								
reg_out(2)	0000000	00000000								
reg_out(3)		00000000								
🔶 reg_out(4)	0000000	00000000	stall the		add to :	\$16 then				
reg_out(5)	0302010	03020100	pipeline		read from	m \$16				
🔶 reg_out(6)		00000B0A	1 cycle							
🔶 reg_out(10)	0000000	00000000	1 0/010						0000000	1
reg_out(15)	1000000	00000000		(1000000	08					
🔶 reg_out(16)	0000000	00000000						0000000	3	
🔶 reg_out(17)	0000000	00000000		1		0000000	8	fo	rward da	to from
reg_out(18)		00000000				forward	data from			
🔶 reg_out(19)	0000000	00000000			ignored	mem sta	age to Rs		em stage	IORS
🔶 reg_out(29)	1000000	10000000				ALU in	2	A	LU in	
reg_out(30)	1000000	(10000000	(10000008	3	$\checkmark$					
🔶 alu_rs_sel	0	1	<u>lo</u>			2	1		0	
🔶 alu_rt_sel	0	0			2	Į0				
🔶 alusrc_ex	1							7		
🔶 alu_ia	1000000	10000008	10000000		10000004	00000008	00000008	00000001	00000008	00000000
🔶 alu_ib	0000000	00000000		0000004	00000000		0000000В	00000000		00000008
🔶 alu_result	1000000	10000008	10000000	100 00004	00000000	00000008	00000001	00000001	80000008	00000000
🔶 exmem_flush	0									
🔶 idex_flush	0									

Instruction 20 & 21 contains another hazard solved by forwarding.

I dk	0												
-🔶 pc_addr	004000:	00400	)0	00400	050	00400	)054	00400	058	00400	)05C	00400	060
-🔶 mif_address	0E	13		14		15		16		17		18	
-🔶 instruction	023280:	00000	000	2A2A0	)00B	11400	0002	02309	020	08100	018	00000	000 🕻
-🔶 reg_out(1)	000000(	00000	0000			_							
-🔶 reg_out(2)	000000(	00000	0000										
-🔶 reg_out(3)	000000(	00000	0000			S	lt res	sult ir	n \$10	)			
-🔶 reg_out(4)	0000000	00000	0000			💼 fo	ollow	red b	y				
-🔶 reg_out(5)	030201(	03020	100			E	Brand	ch co	mpa	re 🛛			
-🔶 reg_out(6)	00000B(	00000	)B0A				ising		÷.,				
-🔶 reg_out(10)	0000000	00000	0001										
-🔶 reg_out(15)	100000(	10000	0008										
-🔶 reg_out(16)	0000000	00000	8000					fo	rwar	d da	ta		
-🔶 reg_out(17)	0000000	00000	8000										
	0000000	00000	0000								stage		
-🔷 reg_out(19)	0000000	00000	0000						RS	DI AL	_U in		
-🔶 reg_out(29)	100000(	10000	0000										
-🔶 reg_out(30)	1000000	10000	8000				_						
-🔷 alu_rs_sel	1	0					_ (	1)		0			
-🔷 alu_rt_sel	2	0											
🔷 alusrc_ex	0												

Instruction 27 & 28 contains another instance of forwarding, but also shows off a branch for the first time. My design branches in the mem stage which means that some instructions accumulate in the pipeline that should not execute when a branch occurs. To solve this I simply let the instructions proceed normally unless a branch occurs, in which case I flush all control lines for the three instructions that follow the branch. In effect this turns those instructions into nops.

clk	1									
pc_addr	004000!	00400064	00400068	0040006C	00400070	00400074	00400078	0040002C	00400030	00
mif_address	14	19	1A	1B	1C	(1D (	11E	ЮВ	0C	<b>DD</b>
instruction	2A2A00	A3D20000	23DE0001 🤆	23BD0001	17AFFFEF	34130008	03B3E823	00000000	93820000	93
reg_out(1)	000000(	0000000								
reg_out(2)	0000000	0000000			to \$20		ranch is t	ruo Drog	iram	
reg_out(3)	0000000	0000000		dd result						
reg_out(4)	0000000	0000000		en use \$			ill repeat			
reg_out(5)	030201(	03020100	br	anch cor	npare	Sa	ame instr	ucuons n	iow _	
reg_out(6)	00000B(	00000B0A								
reg_out(10)	0000000	0000001								
reg_out(15)	100000(	10000008								
reg_out(16)	0000000	0000008	E	orward da	ata from					
reg_out(17)		0000008	m	em stage						
reg_out(18)	0000000	00000010		ALU	. 10 1 13 111					
reg_out(19)		0000000		ALO				since p	rogram	
reg_out(29)	1000000	10000000					10000001	branche		
reg_out(30)	100000(	10000008				1000000			e must be	
alu_rs_sel	0	0				1	<u>io</u>	cleared		
alu_rt_sel	0	0								
alusrc_ex	0							/		
alu_ia	0000000	0000000	10000008		10000000	10000001	00000000	10000001	00000000	10
alu_ib	000000(	00000000		00000001		10000008	0000008	00000000		
alu_result	000000(	00000000	10000008	10000009	10000001	FFFFFFF9	0000000	00000000		10
exmem_flush	0									
idex_flush	0									
ifid Id	1									

From here the program begins to loop many times wherein it simply repeats section of code that I have already covered. Therefore, I will move on to the end of the program where more hazards await.

Instructions 29 - 35 contain the last hazards within the program. After forwarding takes care of the issues the program proceeds to loop forever.

; ᡄᢣ ᢛ		then (	sult to \$ used in lation		⊁∣∰]	\$29 use	ult to \$29 d 2 instru w operati	ictions	Ioop	forever p	with		-
clk	1												
pc_addr	0040008 00	0400 0	040006C	00400070	00400074	00400073	0040007C	0400080	00400084	00400088	0040008C	00400088	0040008C
mif_address	23 <u>1</u> 4	1	В	1C	20	Ŭ1F	ΪIE	20	21	22	23	22	23
instruction	081000: 28	3DE0 2	3BD0001	17AFFFEF	34130008	03B3E823	8FA10000	(8FA20004	3FA30008	8FA40000	08100022	0000000	08100022
reg_out(1)	030201(00	000000								03020100			
reg_out(2)	0B0A09	000000									0B0A0908		
reg_out(3)		000000			subu	resustt to						0316FF10	
reg_out(4)		000000			\$29 t	nen \$29	used 📃						110A0908
reg_out(5)		3020100			in lw	operatio	n 📃						
reg_out(6)		0000B0A											
reg_out(10)		0000001				f		f	orward da	to from			
reg_out(15)	100000( 10				_		data fron	·	VB stage				
reg_out(16)		00000E	foru	vard dat	a from		age to Rs		of ALU	10 5 11			
reg_out(17)		000003				in of AL	U		ALU				
reg_out(18)		0000011		n stage f ALU									
reg_out(19)	000000(			IALU				<u>)00000008</u>					
reg_out(29)	100000( 10					1000008			10000000				
reg_out(30)	100000: 10	00000F			10000010			1		0			
alu_rs_sel alu_rt_sel	0 0				1				2	10			
alu_rt_sei alusrc_ex	0 0						<u> </u>	.0				-	
alusrc_ex alu ia	000000( 10	00000F		10000007	10000008	00000000	10000008	10000000		10000000		00000000	
alu_ia alu_ib		000000F	0000001	,10000007	10000008	00000000	10000008	10000000	00000004	10000000	0000000C	100000000E	00000000
alu_io alu_result			0000001	10000008	10000008	00000008	110000000	100000000	10000004	10000008	11000000C	10000000E	00000000 , 100000000
exmem flush	000000		0000010	10000008	00000000	0000008	1000000	,10000000	1000004	1000008	1000000C	00000000	

🔷 dk	1						
	0040008	00400088	0040008C	00400088	0040008C	00400088	0040008C
	22	22	23	22	23	22	23
	0000000	00000000	08100022	00000000	08100022	00000000	08100022
-🔶 reg_out(1)		03020100					
🛶 reg_out(2)	0B0A0	0B0A0908					
	0316F <mark>-</mark> :	0316FF10					
🛶 reg_out(4)	110A 9	110A0908					
🛶 reg_out(5)	0302010	03020100					
🛶 reg_out(6)	0000 <mark>0</mark> B(	00000B0A		Fina	l Registe	r Values	
	000(000	00000001					
	100(000	10000008					
	000(000	0000000E					
	0000,000	0000003					
	0000000	00000011					
	00000000	0000008					
		10000000					
	100000	10000010					
🛶 alu_rs_sel	0						
-🔶 alu_rt_sel	0			0		0	
📥 alusrc ex	0						

The waveform below shows the final values of the program.

These final values are in agreement with the values obtained using MIPSim which leads me to conclude that my design passed the hazard stress test.