

Done By: Chris Kelley
 Fahmida Jyoti
 Souparni Agnihotri

Project C Lab Report: Pipeline Processor

Team Member Participation:

Souparni Agnihotri: 33.3% Forwarding unit, Schematic, IF stage, ID stage, Testing
 Fahmida Jyoti: 33.3% Hazard control, TB, ID stage, Testing
 Chris Kelley: 33.3% Stage Registers, Remaining Stages, Testing

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

Stage: IF/ID	Datapath Values	Why?	Control Signals	Why?
1.	Reset [1]	Required for reg file and others	PCSrc	For mux before PC
2.	Clock [1]	Required for reg file and others	IF Flush	Enabling flush
3.	Write Enable [1]	Required for reg file and others		
4.	Instruction from Memory[32]	Required for reg file and others		
5.	PC value[32]/ val from adder	PC+4 value		
ID/EX				
1.	Read Data 1[32]	Output from reg file	RegDst	
2.	Read Data 2 [32]	Output from	isLoad	

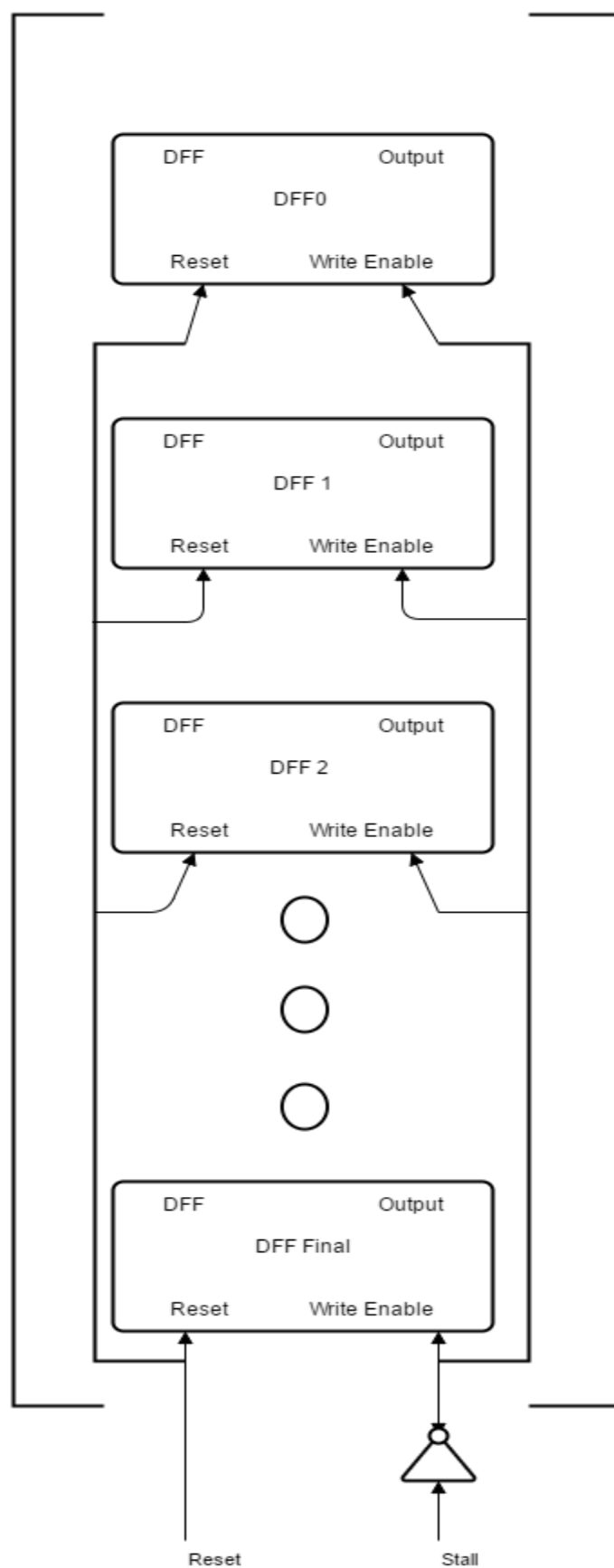
		reg file		
3.	rs(address) (output)	To feed into the forwarding unit	ALU_OP	
4.	rt(address) (output)	To feed into the forwarding unit	Reg_w_en	
5.	Jump/Branch (input)	From mux_9	isLinkALU 1	
6.	Or_branchoutp ut (input)	From branch and jump logic	UpperImm ?1	
7.	zeroOrSignExt (input)	From mux_12	IsTypeSel	
8.	immInstruction [15..0] (input)			
9.	Reset [1]			
10.	Clock [1]			
11.	Write Enable [1]			
EX/MEM				
1.	ALU output (input)		RegDst	
2.	Mux_6 output (input)		isLoad	
	Signals from -->	ID/EX		
3.	PC value[32]/ val from adder		Reg_w_en	

4.	Instruction from Memory[32]		isLinkALU ?1	
5.	immInstruction [15..0] (input)		UpperImm ?1	
6.			IsTypeSel	
7.	Reset [1]			
8.	Clock [1]			
9.	Write Enable [1]			
MEM/WB				
1.			isLoad	
	Signals from -->	EX/MEM		
2.	PC value[32]/ val from adder		isLinkALU	
3.	Instruction from Memory[32]		Reg_w_en	
4.	PC value[32]/ val from adder		UpperImm	
7.	Reset [1]			
8.	Clock [1]			
9.	Write Enable [1]			

Part 1: Pipelined Registers

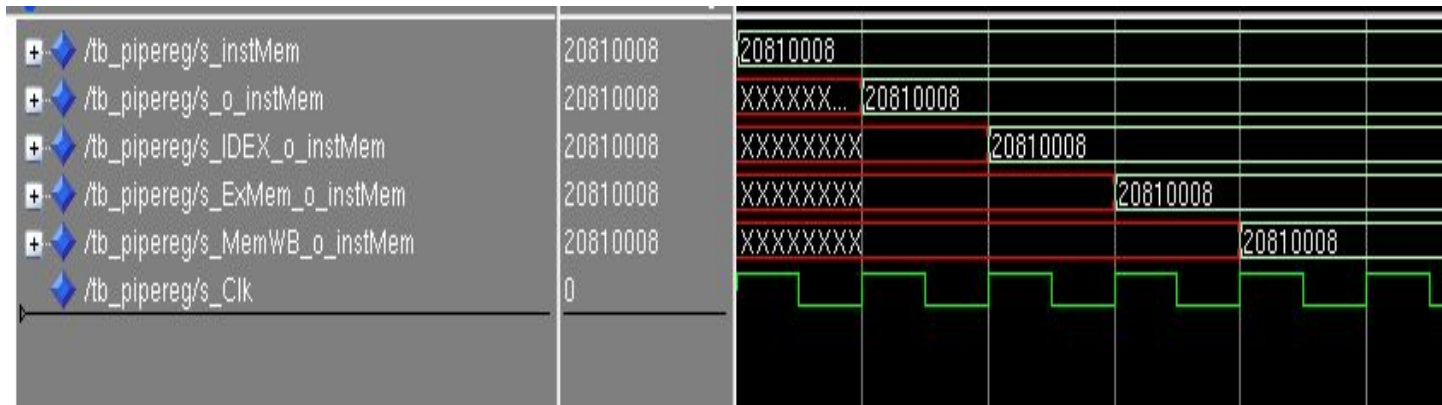
- A) Draw a simple schematic showing how you could implement stalling and flushing operations given an ideal N-bit register.

Pipeline Register



- B)** 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.

We have the addi instruction feeding into the IF stage every cycle and it propagates through all the other stages being delayed by a single clock cycle it travels through the registers.



2) Data Dependencies:

- A)** List which instructions produce values, and what signals in the pipeline these correspond to.
- B)** List which of these same instructions consume values, and what signals in the pipeline these correspond to.

instruction	Produces or not	Consumed or not	Stage Produced	Stage Consumed
add	Yes	Yes	Ex	Ex
addi	Yes	Yes	Ex	Ex
addiu	Yes	Yes	Ex	Ex
addu	Yes	Yes	Ex	Ex
and	Yes	Yes	Ex	Ex
andi	Yes	Yes	Ex	Ex
mul	Yes	Yes	Ex	Ex
nor	Yes	Yes	Ex	Ex
or	Yes	Yes	Ex	Ex
ori	Yes	Yes	Ex	Ex

sll	Yes	Yes	Ex	Ex
sllv	Yes	Yes	Ex	Ex
slt	Yes	Yes	Ex	Ex
slti	Yes	Yes	Ex	Ex
sltiu	Yes	Yes	Ex	Ex
sltu	Yes	Yes	Ex	Ex
sra	Yes	Yes	Ex	Ex
srav	Yes	Yes	Ex	Ex
srl	Yes	Yes	Ex	Ex
srlv	Yes	Yes	Ex	Ex
sub	Yes	Yes	Ex	Ex
subu	Yes	Yes	Ex	Ex
xor	Yes	Yes	Ex	Ex
xori	Yes	Yes	Ex	Ex
bgezal	Yes	Yes	ID	ID
bltzal	Yes	Yes	ID	ID
jalr	Yes	Yes	ID	WB
lui	Yes	Yes	MEM	ID
lb	Yes	Yes	MEM	Ex
lbu	Yes	Yes	MEM	Ex
lh	Yes	Yes	MEM	Ex
lhu	Yes	Yes	MEM	Ex
lw	Yes	Yes	MEM	Ex
sb	Yes	Yes	MEM	Ex
sh	Yes	Yes	MEM	Ex
sw	Yes	Yes	MEM	Ex
jal	Yes	Yes	WB	
beq	No	No		ID
bgez	No	No		ID
bgtz	No	No		ID
blez	No	No		ID
bltz	No	No		ID
bne	No	No		ID
jr	No	No		ID

j	No	No		ID
---	----	----	--	----

C) Come up with a generalized list of potential data dependencies. From this generalized list, select those dependencies that will require forwarding (write down the corresponding pipeline stages that will be forwarding and receiving the data), and those dependencies that will require hazard stalls.

Potential data dependencies would include: Read after write (**RAW**), write after read (**WAR**), and read after read (**RAR**).

Hazard

Both WAR and RAR would include Hazards as a load instruction always requires a stall.

Also any taken branch and jump instruction would require hazard detection. Both require a flush operation.

IDEX_rt, IFID_rs, IFID_rt, IDEX_jump, IDEX_branchAnd, IDEX_branch, IDEX_isload are the signals passed to Hazard detection.

Forwarding

In forwarding we look at the RAW dependencies primarily. Everything that is being written to the register file will have to be forwarded.

For handling the ALU forwarding, we forward the signals and outputs coming from the EX stage, the MEM stage and the WB stage as inputs to the multiplexers preceding the register file.

For example, if we have two add instructions and their registers are dependent on each other, we forward the output (rd) value of the first instruction from the EX stage, into the ID stage. The specific equation for this is :

```
if(s_EXMEM_reg_w_en = '1' and s_EXMEM_RegDest = '0' and (s_EXMEM_rd =
s_IDEX_rs) and (not(s_EXMEM_rd = "00000") and s_isLoad = '0')) {
    forwardA_ALU <= "001";
}
```

This equation exactly forwards the rd value from the EX/MEM stage to the ALU.

Another example of forwarding from WB stage to the EX stage:

This happens when there is a rd value that needs to be forwarded to an instruction which is the third one to be executed.

Specifically:

```
add $1, $2, $3
```

```
nop
```

```
add $4, $1, $5
```

The equation for this is:

```
if(s_MEMWB_reg_w_en = '1' and s_MEMWB_RegDest = '0' and (s_MEMWB_rd =  
s_IDEX_rs) and (not(s_MEMWB_rd = "00000")) and s_isLoad = '1') {
```

```
forwardA_ALU <= "010";
```

```
}
```

So, we are forwarding the value obtained from the MEM/WB stage into the ALU mux so that the value is available at the right time.

For handling the **branch forwarding**, we pass in the outputs from the EX, ID and IF stage as the branch logic is in the ID stage.

The values being forwarded will be from a stage behind because the Branch logic is a stage behind. So for the same sort of scenarios as previously shown for the ALU forwarding, we will have equations that look like this:

```
if(s_IDEX_reg_w_en = '1' and s_IDEX_RegDest = '0' and (s_IDEX_rd = s_IFID_rs)  
and (not(s_IDEX_rd = "00000") and s_isLoad = '0')){
```

```
forwardA_Branch <= "001";
```

```
}
```

This will be for two add instructions sharing the same register.

3) Data Forwarding and Hazard Detection Logic

Write a more generalized series of data forwarding and hazard detection logic based on results from part 2.

Forwarding Equations:

Scenarios:

add \$1, \$2, \$3

add \$4, \$1, \$5

RAW hazard and need to forward EXMEM Aluvalue

```
if((EX/MEM.reg_w_wen and EX/MEM.RegDest = '0' and (EX/MEM.rd == ID/EX.rs) and  
(EX/MEM.rd != 0) and isLoad = 0)
```

```
{  
    forwardA_ALU = "001"  
}
```

-- addi \$1, \$2, 3

-- add \$4, \$1, \$5

-- RAW hazard

```
elseif(EX/MEM.reg_w_wen and EX/MEM.RegDest = '0' and (EX/MEM.rt == ID/EX.rs) and  
(EX/MEM.rt != 0) and isLoad = 0)
```

```
{  
    forwardA_ALU = "001"  
}
```

```
elseif(EXMEM.lui == 1 and MEMWB.reg_w_en == 1 and (IDEX.rs == EXMEM.rd )){
```

```
    forwardA_ALU = "100"  
}
```

```
elseif(MEM/WB.reg_w_wen and MEM/WB.RegDest = '0' and (MEM/WB.rd == ID/EX.rs) and  
(MEM/WB.rd != 0) and isLoad = 1))
```

```
{  
    forwardA_ALU = "010"  
}
```

```
else if (MEM/WB.reg_w_wen and MEM/WB.RegDest = '0' and (MEM/WB.rt == ID/EX.rs) and  
(MEM/WB.rt != 0) and isLoad = 1))
```

```

{
    forwardA_ALU = "010"
}

-- If it is a jr instruction then forward the MEMWB register value
else if((MEM/WB.isLink = '1' or s_MEMWB_isLinkALU = '1') and MEM/WB.reg_w_en = '1' and
(s_IDEX_rs = s_MEMWB_rd) and s_isLoad = '1')
{
    forwardA_ALU <= "011"
}

--- isUpperlmmm instruction being forwarded from MEMWB
elseif(s_MEMWB_lui = '1' and s_MEMWB_reg_w_en = '1' and (s_IDEX_rs = s_MEMWB_rd)
and s_isLoad = '1') then
{
    forwardA_ALU <= "101";
}
else
{
    forwardA_ALU = "000"
}

```

For the second mux to the ALU: (Same as above but with rt values)

```

if((EX/MEM.reg_w_wen = '1' and EX/MEM.RegDest= '0' and (EX/MEM.rd == ID/EX.rt) and
(EX/MEM.rd != "00000") and isLoad = 0)
{
    forwardB_ALU = "001"
}
Else if((EX/MEM.reg_w_wen = '1' and EX/MEM.RegDest = '0' and (EX/MEM.rt == ID/EX.rt) and
(EX/MEM.rt != "00000") and isLoad = 0)
{
    forwardB_ALU = "001"
}
elseif(s_EXMEM_lui = '1' and s_MEMWB_reg_w_en = '1' and (s_IDEX_rs = s_EXMEM_rt)){

    forwardA_ALU <= "100";
}

```

```

}
elseif(MEM/WB.reg_w_wen and MEM/WB.RegDest = '0' and (MEM/WB.rd == ID/EX.rt) and
(MEM/WB.rd != "00000") and s_isLoad = '1')
{
    forwardB_ALU = "010"
}
else if (MEM/WB.reg_w_wen and MEM/WB.RegDest = '0' and (MEM/WB.rt == ID/EX.rt) and
(MEM/WB.rt != "00000") and s_isLoad = '1')
{
    forwardB_ALU = "010"
}

elseif((s_MEMWB_isLink = '1' or s_MEMWB_isLinkALU = '1') and s_MEMWB_reg_w_en = '1'
and s_IDEX_rt = s_MEMWB_rt)
{
    forwardB_ALU <= "011";
}

elseif((s_MEMWB_lui = '1' and s_MEMWB_reg_w_en = '1' and (s_IDEX_rs = s_MEMWB_rt)
and s_isLoad = '1'))
{
    forwardB_ALU <= "101";
}
Else
{
    forwardB_ALU = "00"
}

```

For branches:

```

if((ID/EX.reg_w_wen and ID/EX.RegDest = '0' and (ID/EX.rd == IF/ID.rs) and (ID/EX.rd != 0)
and isLoad = 0)
{
    forwardA_Branch = "001"
}

```

```

elseif(ID/EX.reg_w_wen and ID/EX.RegDest = '0' and (ID/EX.rt == IF/ID.rs) and (ID/EX.rt != 0)
and isLoad = 0)
{
    forwardA_Branch = "001"
}
elseif(ID/EX.lui == 1 and EX/MEM.reg_w_en == 1 and (IFID.rs == IDEX.rd )){

    forwardA_Branch = "100"
}
elseif(EX/MEM.reg_w_wen and EX/MEM.RegDest = '0' and (EXMEM.rd == IF/ID.rs) and
(EXMEM.rd != 0) and isLoad = 1))
{
    forwardA_Branch = "010"
}
else if (EX/MEM.reg_w_wen and EX/MEM.RegDest = '0' and (EXMEM.rt == IF/ID.rs) and
(EX/MEM.rt != 0) and isLoad = 1))
{
    forwardA_Branch = "010"
}
else if((EX/MEM.isLink = '1' or s_EX/MEM_isLinkALU = '1') and EX/MEM.reg_w_en = '1' and
(s_IFID_rs = s_EXMEM_rd) and s_isLoad = '1')
{
    forwardA_Branch <= "011"
}

elseif(s_EXMEM_lui = '1' and s_EXMEM_reg_w_en = '1' and (s_IFID_rs = s_EXMEM_rd) and
s_isLoad = '1') then
{
    forwardA_Branch <= "101";
}
else
{
    forwardA_Branch = "000"
}

```

For the second branch mux

```
if((IDEX.reg_w_wen = '1' and IDEX.RegDest= '0' and (IDEX.rd == IFID.rt) and (IDEX.rd !=
"00000") and isLoad = 0)
{
    forwardB_Branch = "001"
}
Else if((IDEX.reg_w_wen = '1' and IDEX.RegDest = '0' and (IDEX.rt == IFID.rt) and (IDEX.rt !=
"00000") and isLoad = 0)
{
    forwardB_Branch = "001"
}
elseif(s_IDEX_lui = '1' and s_EXMEM_reg_w_en = '1' and (IFID_rs = s_IDEX_rt)){

    forwardB_Branch <= "100";
}
elseif(EX/MEM.reg_w_wen and EXMEM.RegDest = '0' and (EXMEM.rd == IFID.rt) and
(EXMEM.rd != "00000") and s_isLoad = '1')
{
    forwardB_Branch = "010"
}
else if (EXMEM.reg_w_wen and EXMEM.RegDest = '0' and (EXMEM.rt == IFID.rt) and
(EXMEM.rt != "00000") and s_isLoad = '1')
{
    forwardB_Branch = "010"
}

elseif((EXMEM_isLink = '1' or EXMEM_isLinkALU = '1') and EXMEM_reg_w_en = '1' and IFID_rt
= EXMEM_rt)
{
    forwardB_Branch <= "011";
}

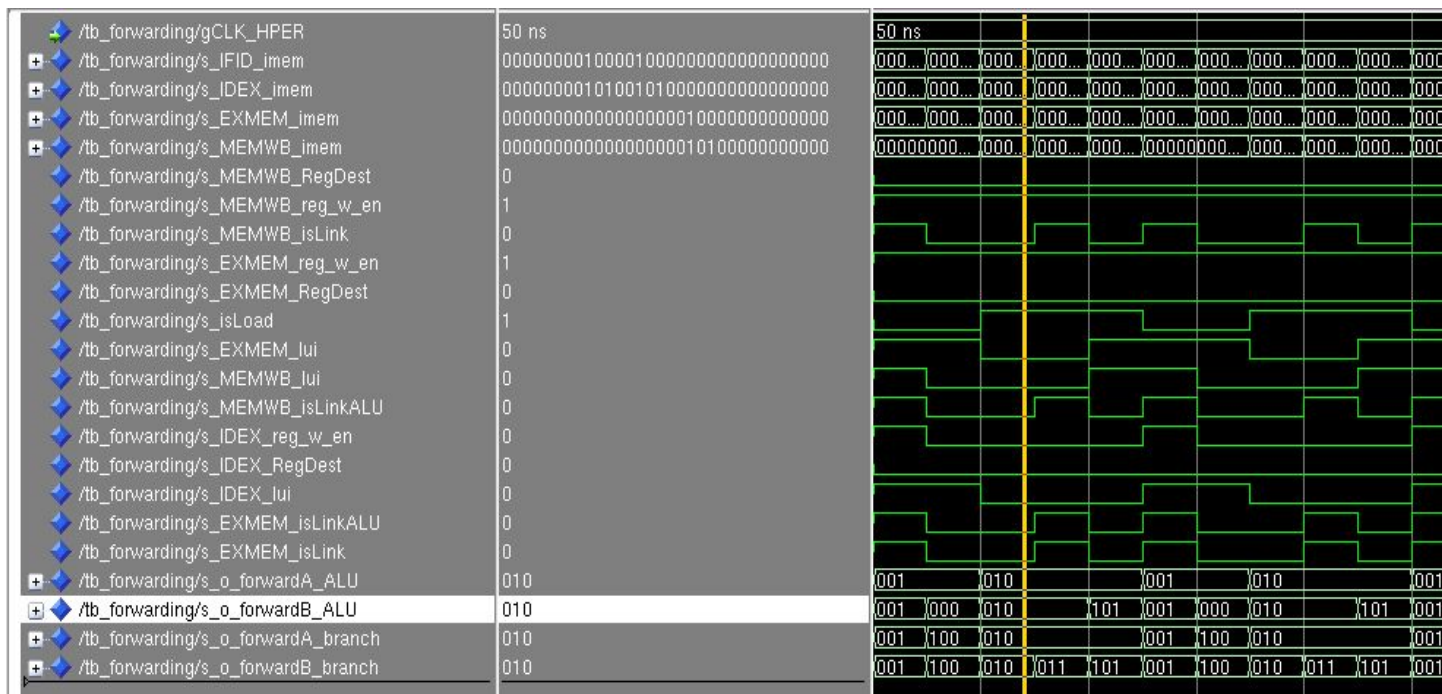
elseif(EXMEM_lui = '1' and EXMEM_reg_w_en = '1' and (IFID_rs = EXMEM_rt) and s_isLoad =
'1'))
{
```

```

        forwardB_Branch <= "101";
    }
    Else
    {
        forwardB_Branch = "000"
    }
}

```

Forwarding unit Testbench:



This is the forwarding unit testbench. According to our equations, when the signals carry the values shown above and we pass in the right bits to the instructions, we are getting the right outputs for our ALU and branch muxes.

The values here are being forwarded from the MEM/WB stage to the EX stage.

The specific type of hazard this deals with is:

addi \$1, \$2, 3

nop

add \$4, \$1. \$5 #\$1 RAW hazard

The equation we are using to check this is:

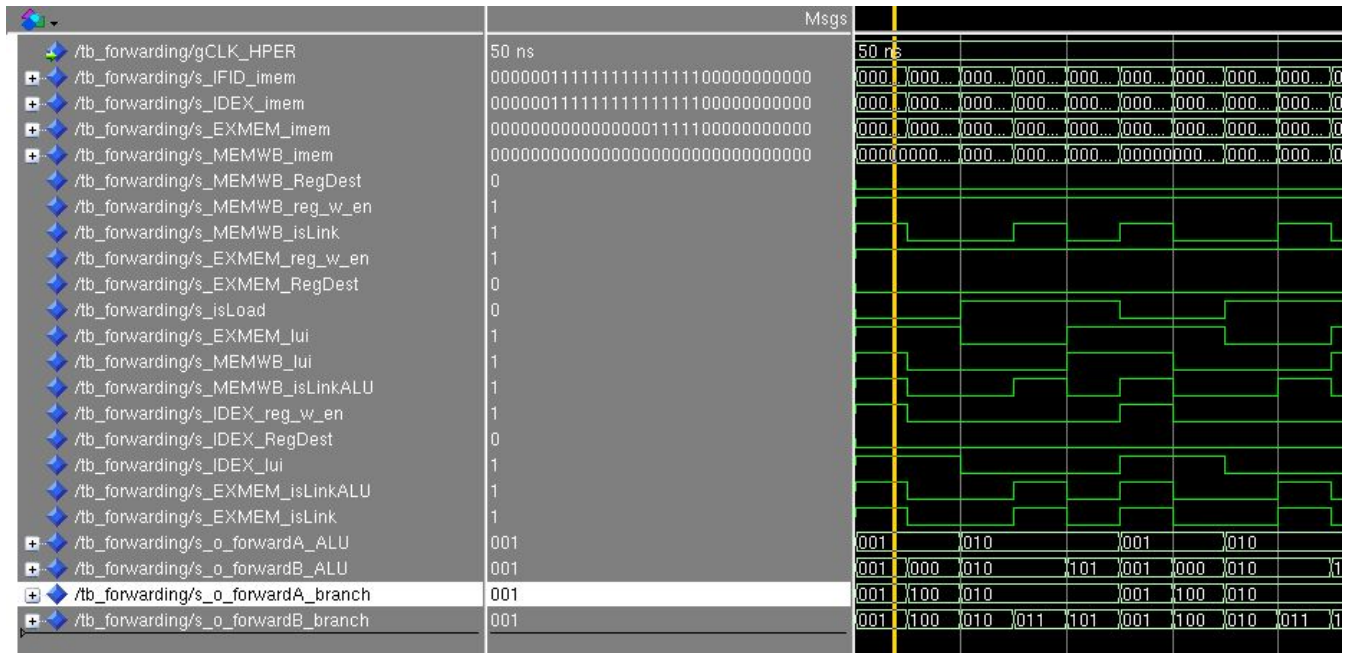
$s_MEMWB_reg_w_en = '1'$ and $s_MEMWB_RegDest = '0'$ and $(s_MEMWB_rt = s_IDEX_rs)$ and $(not(s_MEMWB_rt = "00000"))$ and $s_isLoad = '1'$

That is the ForwardA_ALU output. The Forward B ALU output is the same but with s_IDEX_rt.

The equation that gives the forwardA_branch is:

(s_EXMEM_reg_w_en = '1' and s_EXMEM_RegDest = '0' and (s_EXMEM_rd = s_IFID_rs) and (not(s_EXMEM_rd = "00000"))) and s_isLoad = '1')

And for the forwardB_branch output, the s_IFID_rs is replaced with s_IFID_rt.



This is another picture describing the instructions of type:

add \$1, \$2, \$3

add \$4, \$1, \$5

I.e. instructions that do not have a nop between them. These must be forwarded directly from EX/MEM register outputs to the multiplexers.

```
If (s_IDEX_reg_w_en = '1' and s_IDEX_RegDest = '0' and (s_IDEX_rd = s_IFID_rs) and  
(not(s_IDEX_rd = "00000") and s_isLoad = '0')){  
    forwardA_ALU = "001";  
}
```

Hazarding Equations:

Load

//Load instruction is in the Ex stage

// Add instruction is in the ID stage

Example instruction:

Lw \$t5, 100

Add \$t3, \$t5, \$t4

if ((s_IDEX_isLoad = '1')

and (s_IDEX_rt = s_IFID_rs or s_IDEX_rt = s_IFID_rt) --load hazard

```

and not (s_IDEX_rt = "00000") ) then
s_IDEX_WriteEnable <= '0';
s_IFID_WriteEnable <= '0';
s_IFID_flush <= '1';
s_IDEX_flush <= '0';//We want to load to continue

```

So the load is in Ex/Mem stage while the add instruction which depends on the \$t5 register is in the ID stage so we want to stall the IDEX and IFID register by setting the write enable to be equal to zero and we want to flush the IFID register as we do not want to fetch the add instruction to be fetched again but want the load to continue to execute.

Branch taken

```

// branch is taken so we want to flush the next instruction
//Branch And checks if a branch is taken and IDEX branch detects if it is a branch instruction
Beq $t4,$zero ,L1
Slt $t0, $t2,$t3// we want this to become a nop
L1: mul $t0, $t1, $t2

```

```

elseif (s_IDEX_branchAnd = '1' and s_IDEX_branch = '1') then

```

```

s_IDEX_WriteEnable <= '0';
s_IFID_WriteEnable <= '0';

s_IFID_flush <= '1';
s_IDEX_flush <= '1';

```

In our processor the branchAnd signal (the output of our branch control unit and the branch signal from the control unit which is being passed through the an And gate) detects if a branch is being taken . So we are checking if a branch was taken in the ex stage by checking the value of IDEX_branchAnd signal and the branch signal from the control(If the instruction decoded in the ID stage was a branch). Hence we are flushing the stalling the IDEX and IFID registers. Then we are flushing the IDEX register so that branch instruction is not fetched again and we are flushing the IFID register to flush the instruction after the branch that was fetched.

Jump

```

Example instruction:
J L1 //
Add $t5, $t7,$t6 //this should become a nop
L1 :
Lui $t9 0x0064

```


We detect if there is a jump instruction in the EX stage and we want to flush the next instruction that is fetched. After decoding the jump instruction we want to flush. In this case we are flushing the instruction after jump and we don't want jump instruction to be fetched again after we are done stalling the pipeline. Hence we are flushing the contents of both IDto Ex register and IF to ID register.

```
elseif (( s_IDEX_jump = '1')) then -- jump instruction flush the next instruction
```

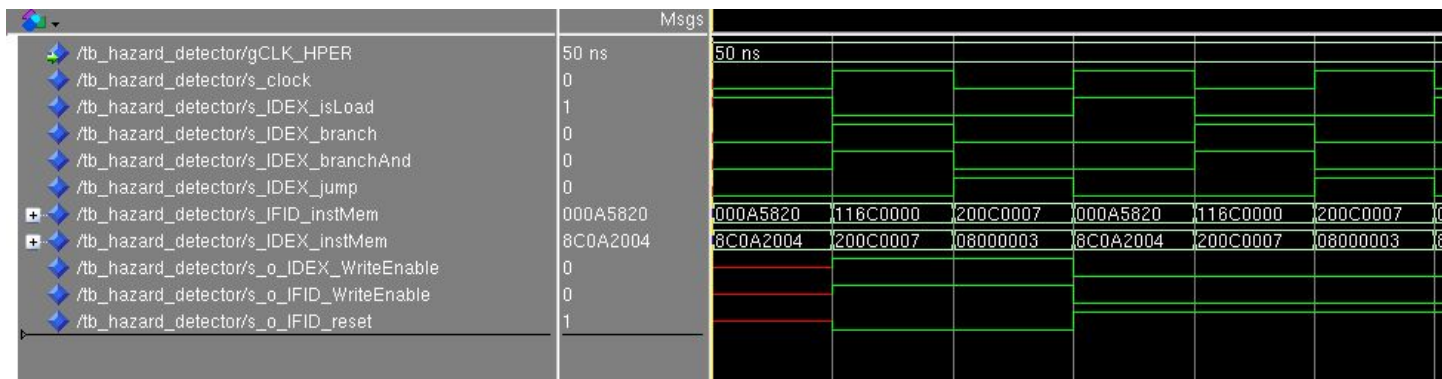
```
s_IDEX_WriteEnable <= '0';
```

```
s_IFID_WriteEnable <= '0';
```

```
s_IFID_flush <= '1';
```

```
s_IDEX_flush <= '1';
```

After we are done stalling



Hazard testbench outputs

4) MIPS Pipelined Processor Implementation

Give a high level schematic drawing of the interconnection between components.

We produced an electronic version of our schematic. You can access it here:

[https://www.draw.io/?state={%22ids%22:\[%220B7flvOYpQehNanNMNkpjNkRXZjQ%22\],%22action%22:%22open%22,%22userId%22:%22103117925243548646793%22}#G0B7flvOYpQehNanNMNkpjNkRXZjQ](https://www.draw.io/?state={%22ids%22:[%220B7flvOYpQehNanNMNkpjNkRXZjQ%22],%22action%22:%22open%22,%22userId%22:%22103117925243548646793%22}#G0B7flvOYpQehNanNMNkpjNkRXZjQ)

5) Testing

A) Create and test an application that makes use of every single instruction.

Registers	Coproc 1	Coproc 0
Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000001
\$t1	9	0x00000000
\$t2	10	0x00000005
\$t3	11	0x00000004
\$t4	12	0x00000007
\$t5	13	0x00000009
\$t6	14	0xffffffff
\$t7	15	0x00000009
\$s0	16	0x00000069
\$s1	17	0x00000009
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x0000000c
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000014
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000

This is the expected register values for our 45 instructions from MARS

+ /mips_pipelined_processor/ID_stage_1/Register_file/i_A	00000000	00000000
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_B	00000000	00000000
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_C	00000000	00000000
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_D	00000000	00000000
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_E	00000000	00000000
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_F	00000000	00000000
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_G	00000000	00000000
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_H	00000000	00000000
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_I	00000001	00000001
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_J	00000000	00000000
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_K	00000005	00000005
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_L	00000004	00000004
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_M	00000007	00000007
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_N	00000000	00000000
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_O	FFFFFFFF	FFFFFFFF
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_P	00000009	00000009
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_Q	00000069	00000069
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_R	00000009	00000009
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_S	00000000	00000000
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_T	00000000	00000000
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_U	0000000C	0000000C
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_V	00000000	00000000
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_W	00000000	00000000
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_X	00000000	00000000
+ /mips_pipelined_processor/ID_stage_1/Register_file/i_Y	00000014	00000014

Expected register values for our 45 MIPS instructions. As you can see the only part that does not match is the LUI. As discussed in class earlier we were unable to figure out why our DFF would not work on the LUI value passed in.

- B) Create a similar application as part a), except select a set of instructions that try to exhaustively test the forwarding and hazard detection capabilities of your pipeline.

In our project we have two test benches, one for each forwarding and hazard detection. Forwarding and hazard is covered above in the report.

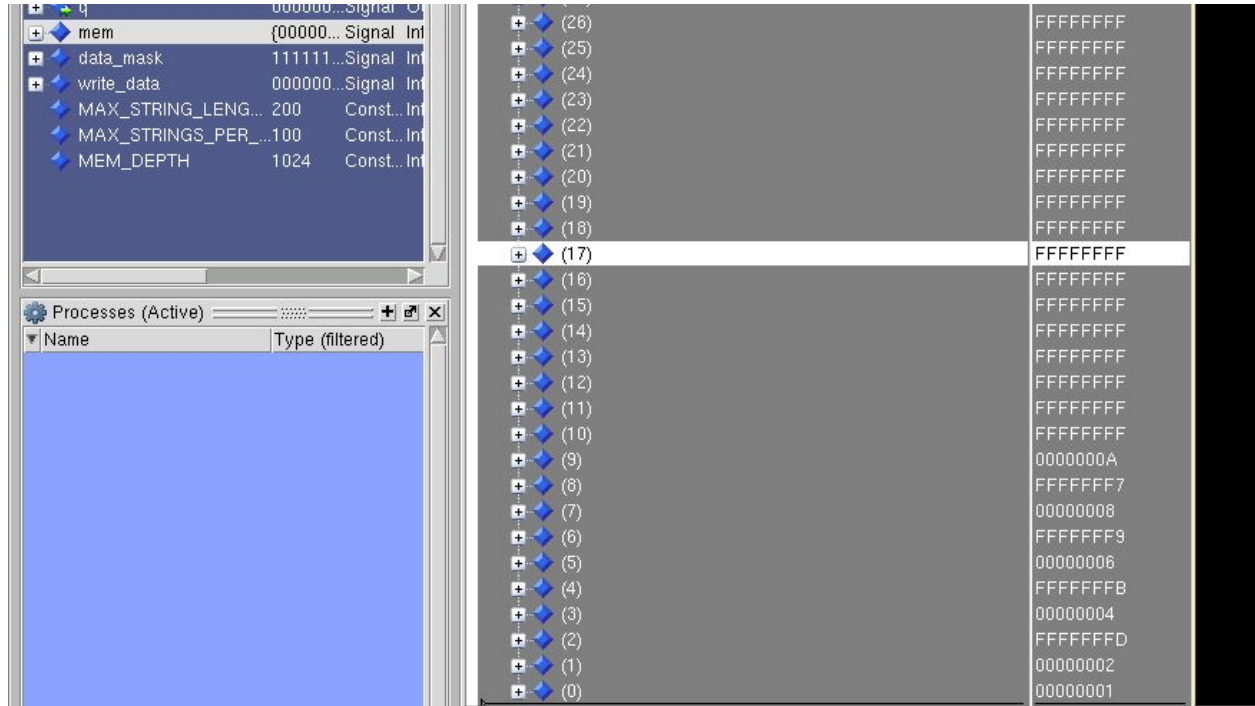
- C) Test the same two sorting algorithms from Project Part B.

In the attached files is a folder called **sorting pictures**. In them we have the registers and memory values expected from MARS and our actual waveform results. In all cases when we tried to access the memory files, they would always be all zeros.

In both sorting cases we used the sorting code given to us by the TAs. We added nops after each instruction. On viewing the pictures you will see that neither sorting algorithm worked completely, values in memory were evaluated and sorted. However, the values expected in the sorting test did not match what we found in memory. This leads us to believe there is an issue

with one of our branches (sorting but incorrect values). On closer look we found that our bgtz was not working properly and causing incorrect values to be stored. The store architecture was given to us by the TAs.

Bubble Sort:



Bubble sort, Mem values at the end

Data Segment					
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Val
0x00000000	0x0000000a	0x0000000f	0x00000014	0x00000024	Val
0x00000020	0x00000054	0x00000055	0x00000062	0x00000000	
0x00000040	0x00000000	0x00000000	0x00000000	0x00000000	
0x00000060	0x00000000	0x00000000	0x00000000	0x00000000	
0x00000080	0x00000000	0x00000000	0x00000000	0x00000000	
0x000000a0	0x00000000	0x00000000	0x00000000	0x00000000	
0x000000c0	0x00000000	0x00000000	0x00000000	0x00000000	
0x000000e0	0x00000000	0x00000000	0x00000000	0x00000000	
0x00000100	0x00000000	0x00000000	0x00000000	0x00000000	
0x00000120	0x00000000	0x00000000	0x00000000	0x00000000	
0x00000140	0x00000000	0x00000000	0x00000000	0x00000000	
0x00000160	0x00000000	0x00000000	0x00000000	0x00000000	
0x00000180	0x00000000	0x00000000	0x00000000	0x00000000	
0x000001a0	0x00000000	0x00000000	0x00000000	0x00000000	
0x000001c0	0x00000000	0x00000000	0x00000000	0x00000000	
0x000001e0	0x00000000	0x00000000	0x00000000	0x00000000	

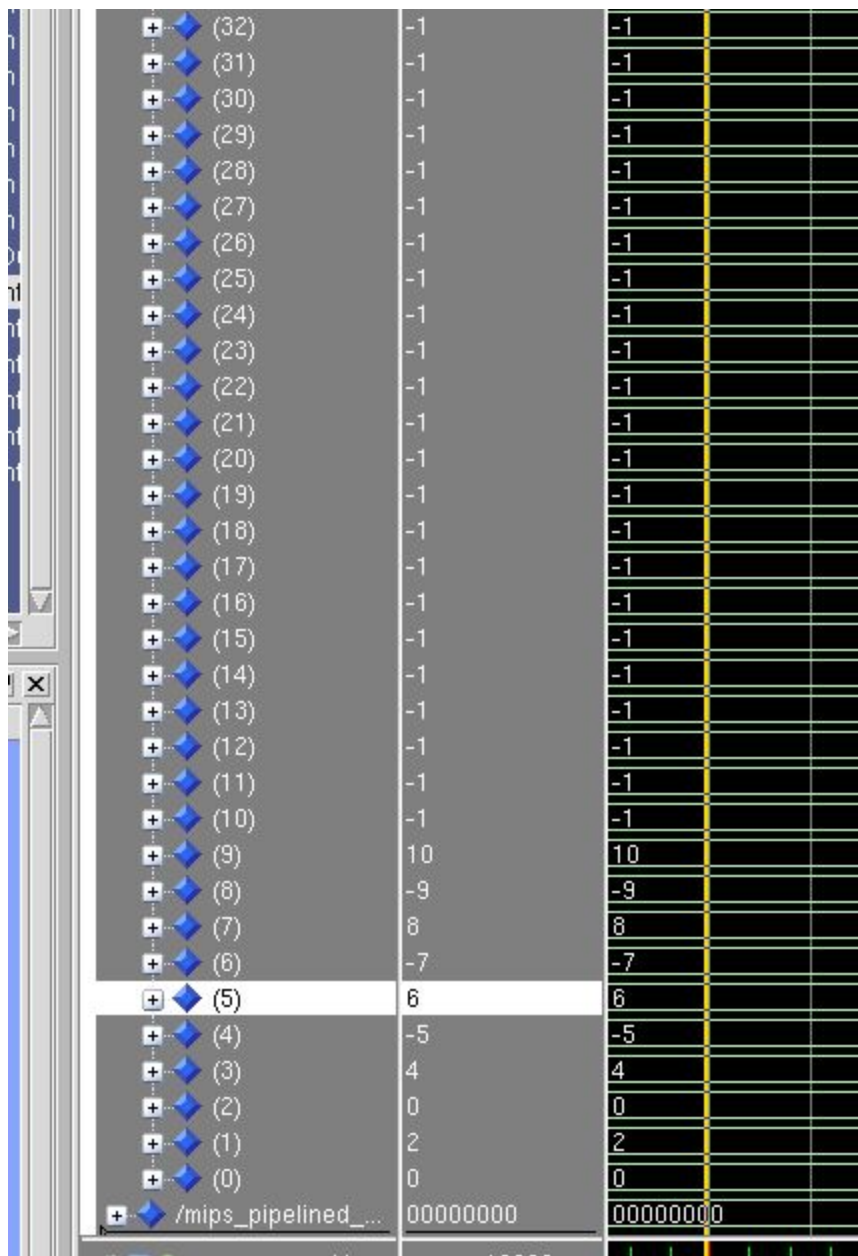
Mars Bubble sort memory output

Registers		
Coproc 1		Coproc 0
Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x0000000a
\$t1	9	0x00000000
\$t2	10	0x00000001
\$t3	11	0x00000004
\$t4	12	0x00000014
\$t5	13	0x0000000f
\$t6	14	0x00000001
\$t7	15	0x00000008
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x00001800
\$sp	29	0x00002ffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x000032bc
hi		0x00000000
lo		0x00000000

Mars Bubble sort registers output

For our bubble sorts we noticed that the values were actually being sorted and the swaps were happening properly but the checking for “less than” and “greater than” was not being done properly. It is mostly because our bgez instruction was not working as it should.

Merge Sort:



(32)	-1	-1
(31)	-1	-1
(30)	-1	-1
(29)	-1	-1
(28)	-1	-1
(27)	-1	-1
(26)	-1	-1
(25)	-1	-1
(24)	-1	-1
(23)	-1	-1
(22)	-1	-1
(21)	-1	-1
(20)	-1	-1
(19)	-1	-1
(18)	-1	-1
(17)	-1	-1
(16)	-1	-1
(15)	-1	-1
(14)	-1	-1
(13)	-1	-1
(12)	-1	-1
(11)	-1	-1
(10)	-1	-1
(9)	10	10
(8)	-9	-9
(7)	8	8
(6)	-7	-7
(5)	6	6
(4)	-5	-5
(3)	4	4
(2)	0	0
(1)	2	2
(0)	0	0
/mips_pipelined_...	00000000	00000000

Mips memory values AFTER merge sort

+	(21)	-1	-1
+	(20)	-1	-1
+	(19)	-1	-1
+	(18)	-1	-1
+	(17)	-1	-1
+	(16)	-1	-1
+	(15)	-1	-1
+	(14)	-1	-1
+	(13)	-1	-1
+	(12)	-1	-1
+	(11)	-1	-1
+	(10)	-1	-1
+	(9)	10	-1
+	(8)	-9	8
+	(7)	8	-9
+	(6)	-7	6
+	(5)	6	-7
+	(4)	-5	4
+	(3)	4	-5
+	(2)	-3	2
+	(1)	2	-3
+	(0)	1	1

Mips memory values BEFORE merge sort

Data Segment				
Address	Value (+0)	Value (+4)	Value (+8)	
0x10010000	0x0000000a	0x00000054	0x0000000f	
0x10010020	0x00000043	0x00000044	0x00000055	
0x10010040	0x00000036	0x00000043	0x00000044	
0x10010060	0x00000000	0x00000000	0x00000000	
0x10010080	0x00000000	0x00000000	0x00000000	
0x100100a0	0x00000000	0x00000000	0x00000000	
0x100100c0	0x00000000	0x00000000	0x00000000	
0x100100e0	0x00000000	0x00000000	0x00000000	
0x10010100	0x00000000	0x00000000	0x00000000	
0x10010120	0x00000000	0x00000000	0x00000000	
0x10010140	0x00000000	0x00000000	0x00000000	
0x10010160	0x00000000	0x00000000	0x00000000	
0x10010180	0x00000000	0x00000000	0x00000000	
0x100101a0	0x00000000	0x00000000	0x00000000	
0x100101c0	0x00000000	0x00000000	0x00000000	
0x100101e0	0x00000000	0x00000000	0x00000000	

Mars memory outputs for merge sort

Registers			Coproc 1	Coproc 0
Name	Number	Value		
\$zero	0	0x00000000		
\$at	1	0x00000000		
\$v0	2	0x00000000		
\$v1	3	0x00000000		
\$a0	4	0x10010004		
\$a1	5	0x1001002c		
\$a2	6	0x00000000		
\$a3	7	0x00000000		
\$t0	8	0x0000000a		
\$t1	9	0x00000010		
\$t2	10	0x00000000		
\$t3	11	0x0000000a		
\$t4	12	0x00000024		
\$t5	13	0x10010028		
\$t6	14	0x00000000		
\$t7	15	0x00000062		
\$s0	16	0x0000000a		
\$s1	17	0x00000000		
\$s2	18	0x00000000		
\$s3	19	0x00000000		
\$s4	20	0x10010050		
\$s5	21	0x00000062		
\$s6	22	0x00000000		
\$s7	23	0x10010030		
\$t8	24	0x0000000a		
\$t9	25	0x00000001		
\$k0	26	0x00000000		
\$k1	27	0x00000000		
\$gp	28	0x10008000		
\$sp	29	0x7fffeffc		
\$fp	30	0x00000000		
\$ra	31	0x00000000		
pc		0x004002e4		
hi		0x00000000		
lo		0x00000000		

Mars register outputs for merge sort

We witnessed the same sorting action with the merge sort as well. But we also saw the same discrepancy in expected and actual memory values as in bubble sort. This can be explained by the same branch error in bgtz.