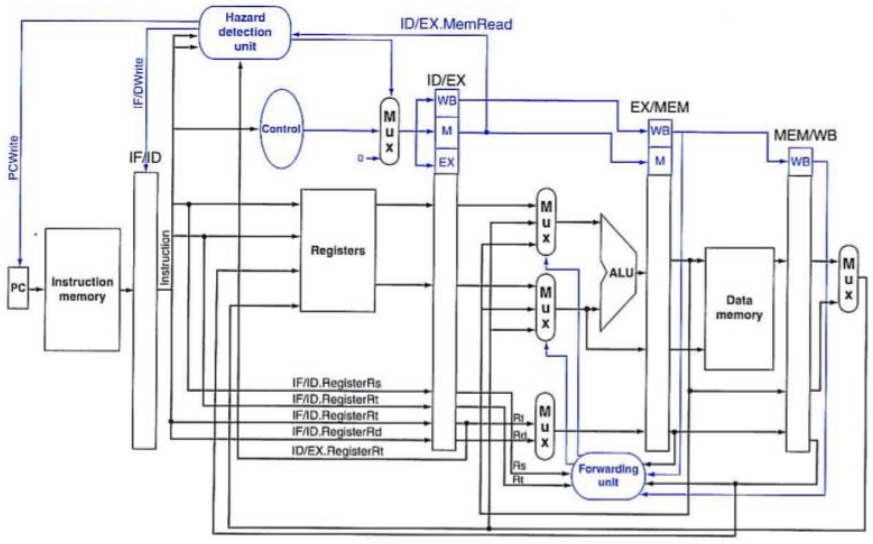
Ain Shams University

Faculty of engineering

Computer and System Engineering Department

Computer Organisation II

Report on: mips pipeline processor



Team members:

Eslam Alaa Zaki	section 1
Fady Faragallah Khalifa	section 2
Mark Remon Ebrahim	section 2
Kirollos Sherif Henry	section 2
Bishoy George Michael	section 1

- **Overview:**

this is an implementation of the pipelined mips processor that takes a textfile as an input for the instruction memory , data memory and register file. also an assembler to support conversion from assembly to machine code.

- **the processor supports:**

add	**sub**	**and**	**or**	**nor**	**sll**	**srl**	**sra**
slt	**addi**	**andi**	**ori**	**slti**	**lui**	**beq**	**bne**
lw	**sw**	**j**	**jr**	**jal**			

- **also the pipeline processor supports forwarding from alu to alu and from memory to alu**
- **a hazard detection unit to hold any operation that reads a register which the lw instruction writes**
- **the processor determines the branch in the execution stage (static branch prediction) so if the branch is taken, the fetched and decoded instruction are flushed.**

- **processor modules:**

  1. pc_counter module:

  - **input: clk, pc_id_exe, pc_if_id, after_sign_ext_id_exe, taken, jump, jal, jr, inst_if_id, jr_cont, hold.**
  - **output: pc**
  - **function:**

    this module should determine the next value of the pc based on the inputs signals (hold,j,jal,taken,jr)

signal	value	pc value
hold	1	pc is the same
j | jal	1	({pc_jump[31:28]},{inst_if_id[25:0]},2'b0}
taken	1	pc_id_exe + 4 + (after_sign_ext_id_exe << 2)
jr	1	jr_cont
hold | j| jal| taken| jr	0	pc + 4

justification:

- **hold(hazard detection unit o/p): means to enter the same instruction again**
- **j | jal : means there's a jump instruction in the decoding stage so the pc will be {{pc_jump[31:28]},inst_if_id[25:0],2'b0}}**

  pc_jump=pc_if_id+4

  inst_if_id is the instruction from the if/id pipeline register which is in this case jump also for pc_if_id is the pc of the instruction in if/id pipeline register

- taken: means the pc will branch to the label so
  pc=pc_id_exe+4+after_sign_ext_id_exe*2

pc_id_exe is the pc of the instruction in the id/ex pipeline register which is in this case beq or bne

after_sign_ext_id_exe is the immediate field after sign extension of the instruction in the id/ex pipeline register which is in this case is beq or bne

taken=((branch&zero_flag) | (branch_not_eq& ~(zero_flag)))

- jr: means it's a jump register instruction so
  pc=jr_cont

jr_cont is the value of read_data1 o/p of the register file

- if none of the previous signals were 1 so the pc = pc+4

---

2. pipeline_registers modules:

a) if_id_reg module:

- input: inst, pc, taken, hold, j, jal, clk.

- output: inst_out, pc_out.

- function: should provide the instruction for the decoding stage

signal	value	pc_out	inst_out
taken| j| jal	1	32'b0 //any value it doesn't matter	32'b0
hold	1	same previous value	same previous value
taken | j| jal | hold	0	pc	inst

justification:

- **taken | j | jal : means that the fetched instruction is wrong so it'll output zero instead.**
- **hold : means to enter the same instruction again**
- **if none of the previous signals were 1 so pc_out=pc of the instruction in the fetching stage and inst_out=the instruction in the fetching stage**

b) id_ex_reg module:

- input: ctrl_signals, pc, read_data1, read_data2, after_sign_ext, rd, rt, rs, flush, taken, clk.

- output: ctrl_signals_out, pc_out, read_data1_out, read_data2_out, after_sign_ext_out, rd_out, rt_out, rs_out.

- function: should provide the control signals (11 signals) and the registers values needed in the execution stage

signal	value	ctrl_signals_out	anything else
flush | taken	1	11'b0	the i/p values //any value it doesn't matter
flush | taken	0	same previous value	same previous value

justification:

- **taken | flush : means that the decoded instruction is wrong so it'll output zero instead.**
- **if none of the previous signals were 1 so the o/p values = i/p values**

c) ex_mem_reg module:

- input: ctrl_signals,alu_out,read_data2,reg_dst_out, clk.
- output: ctrl_signals_out ,alu_out_out ,read_data2_out, reg_dst_out_out.
- function: should provide the control signals (4 signals) and the alu o/p and read_data2 and the reg_dst o/p which is the value of the register to be written in the WB stage.

d) mem_wb_reg module:

- input: ctrl_signals,reg_dst_out,read_data,alu_out, clk.
- output: ctrl_signals_out, reg_dst_out_out, read_data_out, alu_out_out.
- function: should provide the control signals (2 signals) and the alu o/p and the reg_dst o/p which is the value of the register to be written in the WB stage.

---

3. ctrl_unit module:

- input: opcode, function_bits.
- output: 14 bits control signals (mem_write, reg_dst, jump, branch, mem_to_reg, mem_read, alu_src, reg_write, aluop, branch_not_eq, jal, jr) // aluop is 3 bits
- function: should provide the control signals which is determined by the opcode and the function bits

opcode in decimal	ctrl_signals values mem_write,reg_dst,jump ,branch,branch_not_eq,mem_to_reg,mem_read,alu_src,reg_write,aluop, jal, jr	
0	function_bits = 8 // jr instruction	0x000x0x0xxx01
	any other R instruction	01000000101000
2	0x100x0x0xxx00	
3	0x000x0x1xxx10	
35	00000111100000	
43	1x000x01000000	
4	0x010x00000100	
5	0x001x00000100	
8	00000001100000	
13	00000001110000	
12	00000001101100	
10	00000001111000	
15	00000001110100	

- there are 14 signals : 3 of them used in the decoding stage (j , jal ,jr)

  7 of them used in the execution stage (reg_dst, branch, branch_not_eq, alu_src, aluop)

  2 of them used in the memory stage (mem_write, mem_read)

  2 of them used in the write back stage (reg_write, mem_to_reg)

4. alu_control module:

- input: aluop, function_bits.
- output: control_bits.
- function: should provide the control bits to the alu to do operation based on these bits.

aluop	function_bits	control_bits
000	xxxxxx	0010
001	xxxxxx	0110
010 // so the alu control unit will give an output based on the function bits of the instruction	100000	0010
	100010	0110
	100100	0000
	100101	0001
	101010	0111
	100111	1100
	000000	1101
	000010	0011
	000011	0100
011	xxxxxx	0000
100	xxxxxx	0001
101	xxxxxx	1010
110	xxxxx	0101

5. real_alu module:

- input: in1, in2, shamt, op.
- output: out, zero_flag.
- function: do operation based on the control_bits o/p from the alu control unit.

op	operation	
0000	in1 & in2	used in and, andi, lw, sw
0001	in1 | in2	used in or , ori
0010	in1 + in2	used in add , addi
0110	in1 - in2	used in sub , beq , bne
0111 0101	32'b1 & ((in1-in2)>>31)	used in slt used in slti
1100	~ (in1 | in2)	used in nor
1101	in2 << shamt	used in sll
0011	in2 >> shamt	used in srl
0100	$signed(in2) >>> shamt	used in sra
1010	in2 << 16	used in lui

6. HDU module:

- input: id_ex_mem_read ,id_ex_rt ,if_id_rs ,if_id_rt.
- output: hold.
- function: check the following condition if true it will send signal hold to pc_counter ,if_id_reg to keep the same content and id_exe_reg to flush the instruction by giving its control signals zero value:

id_ex_mem_read & ((id_ex_rt==if_id_rs) | (id_ex_rt==if_id_rt))

to prevent data hazard due to lw followed by any instruction depend on its result in the rt register

id_ex_mem_read: indicates that the instruction in the execution stage is lw

id_ex_rt: the rt register of the instruction in the execution stage

if_id_rs: the rs register of the instruction in the decoding stage

if_id_rt: the rt register of the instruction in the decoding stage

---

7. forwarding_unit_module:

- input: id_ex_rs , id_ex_rt, ex_mem_rd , mem_wb_rd , ex_mem_regwrite , mem_wb_regwrite.
- output: forwarding_a , forwarding_b.
- function: check the dependency between the instruction in execution stage and the instructions in both the memory stage and write back stage and forward the correct data to the alu.

condition	value	forwarding signal
(ex_mem_regwrite) && (ex_mem_rd!=0) && (ex_mem_rd==id_ex_rs)	1	10
(ex_mem_regwrite) && (ex_mem_rd!=0) && (ex_mem_rd==id_ex_rs)	0	01
(mem_wb_regwrite) && (mem_wb_rd !=0) && (mem_wb_rd==id_ex_rs)	1	
(ex_mem_regwrite) && (ex_mem_rd!=0) && (ex_mem_rd==id_ex_rs)	0	00
(mem_wb_regwrite) && (mem_wb_rd !=0) && (mem_wb_rd==id_ex_rs)	0	

- same conditions for id_ex_rt.

the periority to the data in memory stage if it's needed by the instruction in the execution stage the forwarding unit will forward it, then if the data in write back stage is needed by the instruction in the execution stage the forwarding unit will forward it.

ex_mem_regwrite: indicates that the instruction in memory stage is R type that write in a register.

mem_wb_regwrite: indicates that the instruction in write back stage is R type that write in a register.

ex_mem_rd: the value of rd register of the instruction in memory stage.

mem_wb_rd: the value of rd register of the instruction in write back stage.

id_ex_rs: the value of rs register of the instruction in execution stage.

id_ex_rt: the value of rt register of the instruction in execution stage.

8. mux modules:

a)mux2to1 module:

- input: in1, in2, sel.
- output: out.
- function: takes 32 bits 2-inputs and outputs one of them based on the sel signal.

sel signal	out
0	in1
1	in2

b)mux5 2to1 module:

- input: in1, in2, sel.
- output: out.
- function: takes 5 bits 2-inputs and outputs one of them based on the sel signal.

sel signal	out
0	in1
1	in2

c)mux3to1 module:

- input: in1, in2, in3, sel.
- output: out.
- function: takes 32 bits 3-inputs and outputs one of them based on the sel signal.

sel signal	out
00	in1
01	in2
10	in3

---

9. adder module:

- input: in1, in2.
- output: out.
- function: adding the 2 inputs.

---

10. sign ext module:

- input: in.
- output: out.
- function: extending the 16^{th} bit of the input signal and outputs 32 bits output.

---

11. **inst_mem_module:**

- input: read_address.
- output: instruction.
- function: initialized from a text file by the instructions to be executed, every clock cycle a an instruction is fetched from it based on the value of read_address.

---

12. **data_mem_module:**

- input: address, write_data , memRead ,memWrite ,clk.
- output: read_data.
- function: initialized from a text file by the values to be filled in it. if memWrite = 1 then it'll store the value of write_data in the provided address.

---

13. **reg_f_module:** (contains 32 registers)

- input: read_reg1,read_reg2,write_reg,write_data,reg_write, jal, pcplus4, clk.
- output: read_data1,read_data2 .
- function: initialized from a text file by the values to be filled in it

jal =1	store pc+4 in $ra register
reg_write =1 // from the mem/wb pipeline register	store write_data in the register that has the address in write_reg

14. **pipeline_processor module:**

- input: clk.

- function: this is an integration of all modules in the pipeline mips processor and connecting them together by internal wires to enable the processor to do its functionality.

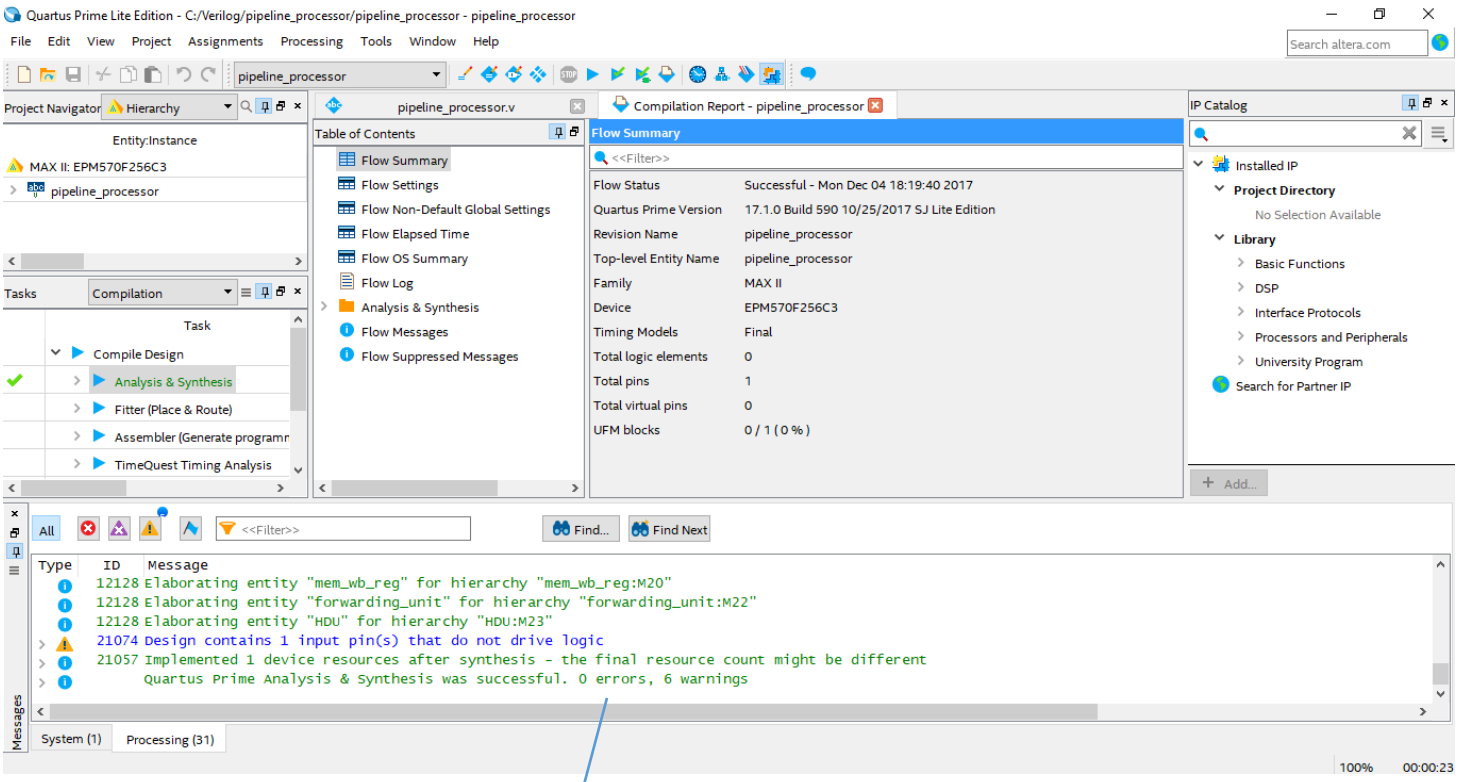
---

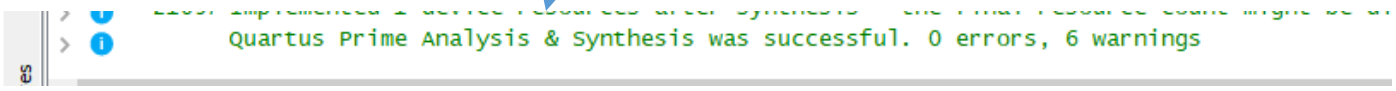
15. **pro_pipe_test module:**

- function: just a test bench to generate the clock needed by the pipeline processor.

- **synthesis check:**

  synthesizability was checked using Quartus prime tool and synthesis was successful





the warnings aren't related to our code, just to the unitialized parts of the memory and number of processors that hasn't been set during installation.

- **also we have tested it using xilnix and generate a mapping file to calculate the number of elements needed in the design and it was successfully synthesized and here ‘s the number of elements coming out of the report**

```
---------------------------------------------------------------------------
HDL Synthesis Report

Macro Statistics
# RAMs                                                 : 1
 7x32-bit single-port RAM                              : 1
# Adders/Subtractors                                   : 7
 32-bit adder                                          : 6
 32-bit subtractor                                     : 1
# Registers                                            : 51
 11-bit register                                       : 1
 2-bit register                                        : 1
 32-bit register                                       : 43
 4-bit register                                        : 1
 5-bit register                                        : 5
# Comparators                                          : 7
 32-bit comparator equal                               : 1
 5-bit comparator equal                                : 6
# Multiplexers                                         : 5
 32-bit 32-to-1 multiplexer                            : 2
 32-bit 4-to-1 multiplexer                             : 2
 4-bit 8-to-1 multiplexer                              : 1
# Logic shifters                                       : 3
 32-bit shifter arithmetic right                       : 1
 32-bit shifter logical left                           : 1
 32-bit shifter logical right                          : 1

---------------------------------------------------------------------------
===========================================================================
Advanced HDL Synthesis Report

Macro Statistics
# RAMs                                                 : 1
 7x32-bit single-port distributed RAM                  : 1
# Adders/Subtractors                                   : 7
 32-bit adder                                          : 6
 32-bit subtractor                                     : 1
# Registers                                            : 1418
 Flip-Flops                                            : 1418
# Comparators                                          : 7
 32-bit comparator equal                               : 1
 5-bit comparator equal                                : 6
# Multiplexers                                         : 67
 1-bit 32-to-1 multiplexer                             : 64
 32-bit 4-to-1 multiplexer                             : 2
 4-bit 8-to-1 multiplexer                              : 1
# Logic shifters                                       : 3
 32-bit shifter arithmetic right                       : 1
 32-bit shifter logical left                           : 1
 32-bit shifter logical right                          : 1

===========================================================================
```

- **how to use the code:**

  the processor reads from 3 text files “inst_mem.txt” for the instruction memory. “data_mem.txt” for the data memory. “reg_f.txt” for the register file.

- **you must change the files path in the code depending on their path on your pc.**
- **after filling the files with the binary values and running simulation the results will be displayed in the memory list in modelsim simulator with the last values of both memories and the register file.**

- **test cases:**
  - **how to use test cases?**

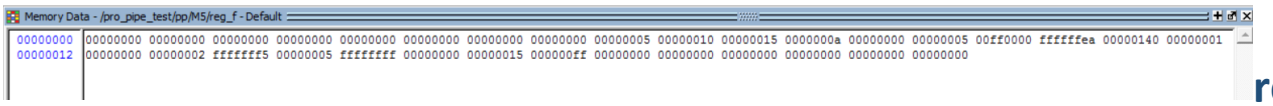
change the path in $memreadb() in each inst_mem module, data_mem module and reg_f module to the path of each test case contents "inst_mem.txt , data_mem.txt and reg_f.txt" .

test case (1):

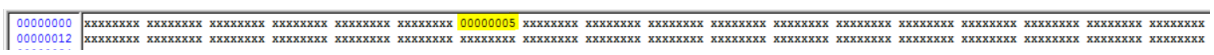
assembly	machine code
add $t2,$t0,$t1	00000001000010010101000000100000
addi $t3,$t0,5	00100001000010110000000000000101
and $t4,$t0,$t1	00000001000010010110000000100100
andi $t5,$t0,15	00110001000011010000000000001111
lui &t6,255	00111100000011100000000011111111
nor $t7,$t0,$t1	00000001000010010111100000100111
or $t8,$t0,$t1	00000001000010011100000000100101
ori $t9,$t0,255	00110101000110010000000011111111
sll $s0,$t0,6	00000000000010001000000110000000
slt $s1,$t0,$t1	00000001000010011000100000101010
slti $s2,$t0,3	00101001000100100000000000000011
srl $s3,$t0,1	00000000000010001001100001000010
sub $s4,$t0,$t1	00000001000010011010000000100010
sw $t0,8($t1)	10101101001010000000000000001000
lw $s5,8($t1)	10001101001101010000000000001000
sra $s6,$t7,15	00000000000011111011001111000011

initial values	expected output
$t0=0x00000005 => index[8] $t1=0x00000010 => index[9] and other registers =0x00000000	$t0=0x00000005 => index[8] $t1=0x00000010 => index[9] $t2=0x00000015 => index[10] $t3=0x0000000a => index[11] $t4=0x00000000 => index[12] $t5=0x00000005 => index[13] $t6=0x00ff0000 => index[14] $t7=0xffffffea => index[15] $t8=0x00000015 => index[24] $t9=0x000000ff => index[25] $s0=0x00000140 => index[16] $s1=0x00000001 => index[17] $s2=0x00000000 => index[18] $s3=0x00000002 => index[19] $s4=0xfffffff5 => index[20] $s5=0x00000005 => index[21] $s6=0xffffffff => index[22] and other registers =0x00000000 data_memory[6]=0x00000005

output:



reg_f



data_mem

- **why using this test case?**

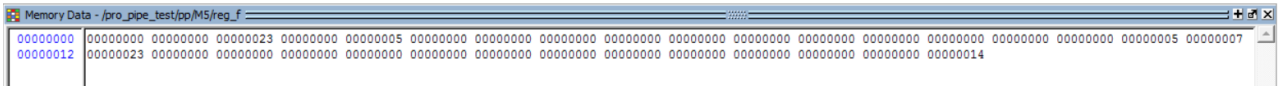
for testing instructions: add addi and andi lui nor or ori sll slt slti srl sub sw lw sra "without any hazard case"

test case (2):

assembly	machine code
Main: addi $s0,$zero,5	00100000000100000000000000000101
addi $s1,$zero,7	00100000000100010000000000000111
add $a0,$s0,$zero	00000010000000000010000000100000
add $a1,$s1,$zero	00000010001000000010100000100000
jal MulPositive	00001100000000000000000000000111
add $s2,$v0,$zero	00000000010000001001000000100000
j Exit	00001000000000000000000000001101
MulPositive: add $v0,$zero,$zero	00000000000000000001000000100000
L: beq $a1,$zero,FunExit	00010000101000000000000000000011
add $v0,$v0,$a0	00000000010001000001000000100000
addi $a1,$a1,-1	00100000101001011111111111111111
j L	00001000000000000000000000001000
FunExit: jr $ra	00000011111000000000000000001000
Exit: sll $zero,$zero,0	00000000000000000000000000000000

initial values	expected output
all registers=0x00000000	$s0=0x00000005 => index[16] $s1=0x00000007 => index[17] $s2=0x00000023 => index[18] $a0=0x00000005 => index[4] $a1=0x00000000 => index[5] $v0=0x00000023 => index[2] $ra=0x00000014 => index[31] and other registers=0x00000000

output:



reg_f

- **<u>why using this test case?</u>**

for testing beq & j & jal & jr instructions

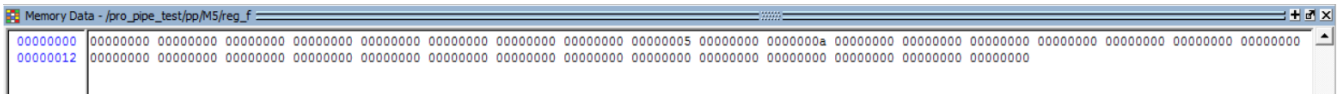
"without any hazard case"

Hint : you must increase simulation time in this test case for example 250 ns at least.

test case (3):

assembly	machine code
lw $t0,8($t1) add $t2,$t0,$t0	10001101001010000000000000001000 00000001000010000101000000100000
initial values	**expected output**
all registers =0x00000000 data memory[0]=data memory [1] =0x00000000 data memory [2]=0x00000005	no change in data memory $t0=0x00000005 => index[8] $t2=0x0000000a => index[10] and other registers =0x00000000

output:



reg_f

- **why using this test case?**

for testing data hazard due to add after lw "control hazard"

and testing forwading from memory to alu

test case (4):

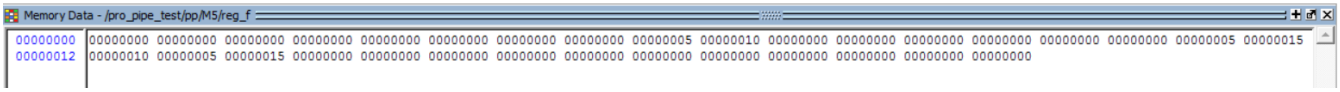
assembly	machine code

sw $t0,8($t1)
lw $s0,8($t1)
add $s1,$s0,$t1
sub $s2,$s0,$t1
	10101101001010000000000000001000
	10001101001100000000000000001000
	00000010000010011000100000100000
	00000010000010011001000000100010

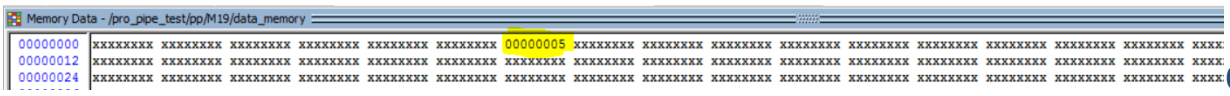
add $s3,$s0,$s0
sub $s3,$s3,$s0
sub $s2,$s3,$s2
add $s4,$s3,$s2
	00000010000100001001100000100000
	00000010011100001001100000100010
	00000010011100101001000000100010
	00000010011100101010000000100000

initial values	expected output
$t0=0x00000005 => index[8] $t1=0x00000010 => index[9] and other registers=0x00000000	$s0=0x00000005 => index[16] $s1=0x00000015 => index[17] $s2=0x00000010 => index[18] $s3=0x00000005 => index[19] $s4=0x00000015 => index[20] $t0=0x00000005 => index[8] $t1=0x00000010 => index[9] and other registers=0x00000000 data memory[6]=0x00000005

output:



reg_f



data_mem

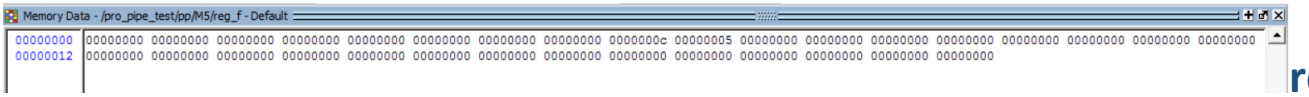
- <u>why using this test case?</u>

for testing data hazard and testing forwarding from alu to alu and memory to alu to avoid read after write hazard and testing control hazard

test case (5):

assembly	machine code
addi $t0,$zero,7	00100000000010000000000000000111
add $t1,$zero,$zero	00000000000000000100100000100000
L: slti $t2,$t1,5	00101001001010100000000000000101
beq $t2,$zero,Exit	00010001010000000000000000000011
addi $t0,$t0,1	00100001000010000000000000000001
addi $t1,$t1,1	00100001001010010000000000000001
j L	00001000000000000000000000000010
Exit: sll $zero,$zero,0	00000000000000000000000000000000
initial values	**expected output**
all registers=0x00000000	$t0=0x0000000c => index[8] $t1=0x00000005 => index[9] and other registers=0x00000000

output:



reg_f

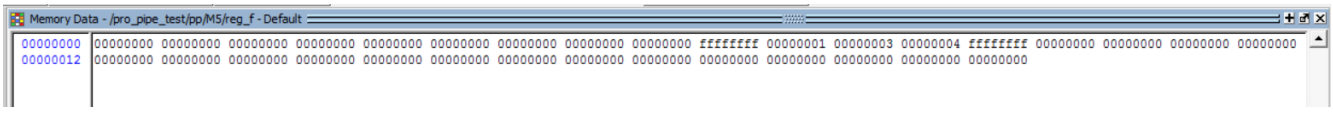
- **why using this test case?**

for testing simple program "for loop".

test case (6):

assembly	machine code
addi $t1,$zero,-11	00100000000010011111111111110101
addi $t1,$t1,10	00100001001010010000000000001010
slti $t2,$t1,0	00101001001010100000000000000000
beq $t2,$zero,L //not taken	00010001010000000000000000000010
addi $t3,$zero,3	00100000000010110000000000000011
addi $t4,$t3,1	00100001011011000000000000000001
L: add $t5,$t2,$t2	00000001010010100110100000100000
sra $t5,$t1,4	00000000000010010110100100000011
initial values	**expected output**
all registers=0x00000000	$t1=0xFFFFFFFF => index[9] $t2=0x00000001 => index[10] $t3=0x00000003 => index[11] $t4=0x00000004 => index[12] $t5=0xffffffff => index[13] and other registers=0x00000000

output:



reg_f

- **why using this test case?**

for testing data hazard and forwarding control hazard with beq instruction.

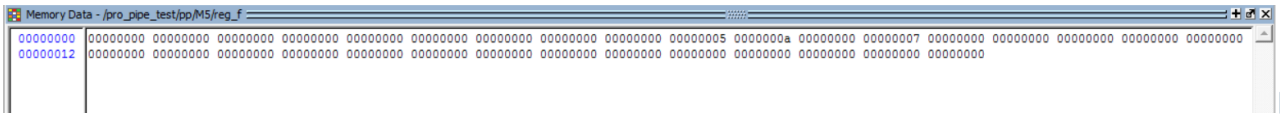
test case (7):

assembly	machine code

lw $t2,4($zero)
beq $t2,$t1,L
addi $t2,$zero,15
nor $t1,$t2,$zero
L: add $t2,$t2,$t2
slt $t3,$t1,$zero
ori $t4,$t1,3
	10001100000010100000000000000100
	00010001010010010000000000000010
	00100000000010100000000000001111
	00000001010000000100100000100111
	00000001010010100101000000100000
	00000001001000000101100000101010
	00110101001011000000000000000011

initial values	**expected output**
$t1=0x00000005 => index[9] data_memory[1] =0x00000005 and other registers=0x00000000	$t1=0x00000005 => index[9] $t2=0x0000000a => index[10] $t3=0x00000000 => index[11] $t4=0x00000007 => index[12] and other registers=0x00000000

output:



reg_f

- **why using this test case?**

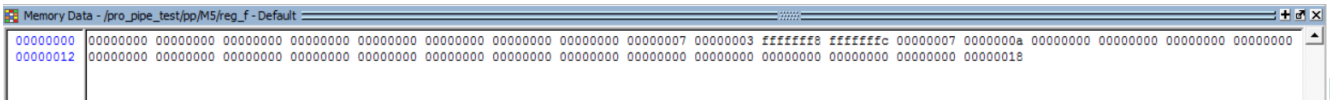
for testing hazard due to beq after lw instruction "control hazard" .

test case (8):

assembly	machine code
nor $t2,$t0,$t1	00000001000010010101000000100111
sub $t3,$t1,$t0	00000001001010000101100000100010
bne $t1,$t0,L	00010101001010000000000000000010
sll $zero,$zero,0	00000000000000000000000000000000
sll $zero,$zero,0	00000000000000000000000000000000
L: jal M	00001100000000000000000000000111
sll $zero,$zero,0	00000000000000000000000000000000
M: or $t4,$t0,$zero	00000001000000000110000000100101
j N	00001000000000000000000000001010
sll $zero,$zero,0	00000000000000000000000000000000
N: add $t5,$t0,$t1	00000001000010010110100000100000
initial values	**expected output**
$t0=0x00000007 => index[8]	$t0=0x00000007 => index[8]
$t1=0x00000003 => index[9]	$t1=0x00000003 => index[9]
and other registers=0x00000000	$t2=0xfffffff8 => index[10]
	$t3=0xfffffffc => index[11]

	$t4=0x00000007 => index[12] $t5=0x0000000a => index[13] $ra=0x00000018 => index[31] and other registers=0x00000000

output:



reg_f

- why using this test case?

for testing jal & j instruction "without hazard "

test case (9):

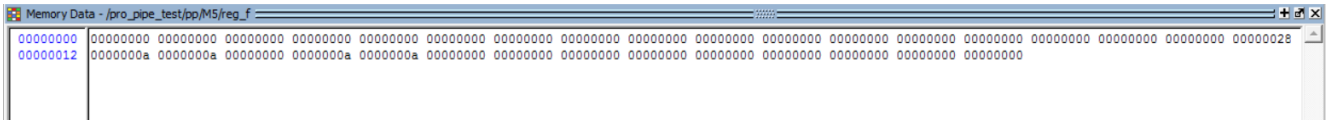
assembly	machine code

lw $s1,0($s2)
sw $s3,0($s1)
add $s2,$zero,$s3
sub $s4,$s2,$s2
or $s5,$s4,$s2
andi $s6,$s2,11
	10001110010100010000000000000000
	10101110001100110000000000000000
	00000000000100111001000000100000
	00000010010100101010000000100010
	00000010100100101010100000100101
	00110010010101100000000000001011

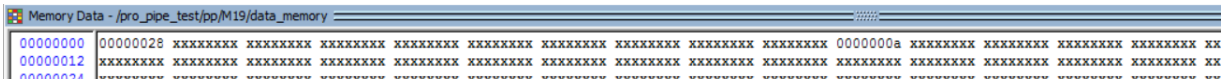
initial values	expected output
$s3=0x0000000a => index[19] and other registers=0x00000000 data_memory[0]=0x00000028	$s1=0x00000028 => index[17] $s2=0x0000000a => index[18] $s3=0x0000000a => index[19]

	$s4=0x00000000 => index[20] $s5=0x0000000a => index[21] $s6=0x0000000a => index[22] and other registers=0x00000000 data_memory[0] =0x00000028 data_memory[10]=0x0000000a

output:



reg_f



data_mem

- **<u>why using this test case?</u>**

for testing all hazard cases