BITS PILANI K.K. BIRLA GOA CAMPUS

MIPS Pipelining and Hazard Detection

Towards the Fulfilment of Computer Architecture

Manav Agarwal 2018B4A30816G

Report Contains:

Logic behind the solution for hazard (All the diagrams are taken from the book)

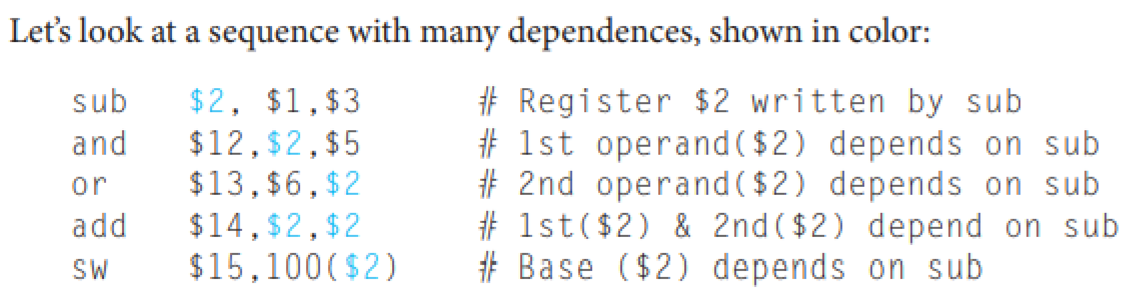
Simulation runs from Vivado Design Suit

Proof Simulation for Support of Jump instruction

Result from Synthesis (Timing information)

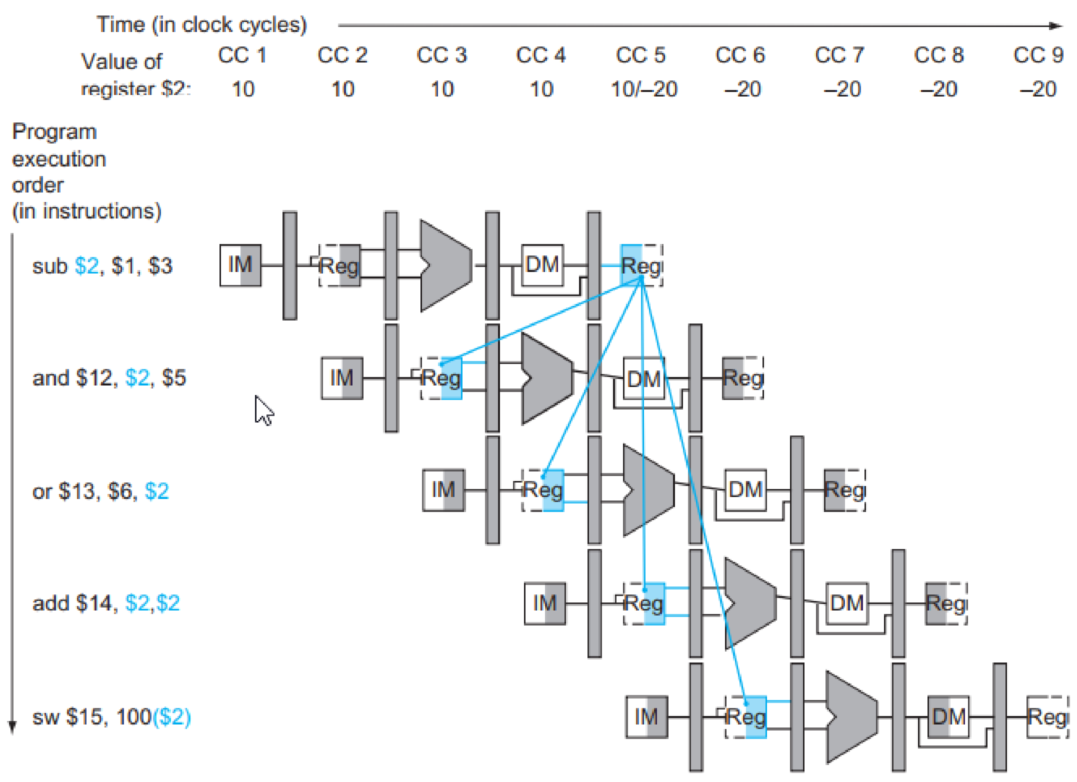
Data Hazard and Solution

Following screenshots are taken from Computer Organization and Architecture by Hennessy and Patterson.



Let's look at a sequence with many dependences, shown in color:

```
sub   $2, $1,$3      # Register $2 written by sub
and   $12,$2,$5      # 1st operand($2) depends on sub
or    $13,$6,$2      # 2nd operand($2) depends on sub
add   $14,$2,$2      # 1st($2) & 2nd($2) depend on sub
sw    $15,100($2)    # Base ($2) depends on sub
```



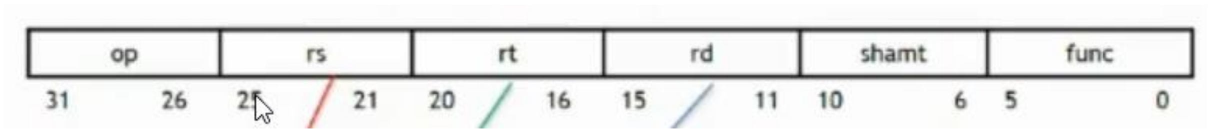
As it can be seen that in normal pipeline with no hazard mitigations we need to time, when to use a value in register because if an operation is performed on it, then it will take couple of clock cycles to be updated in register.

Naïve Solution: Detect hazard and include nops or independent instructions to include delay.

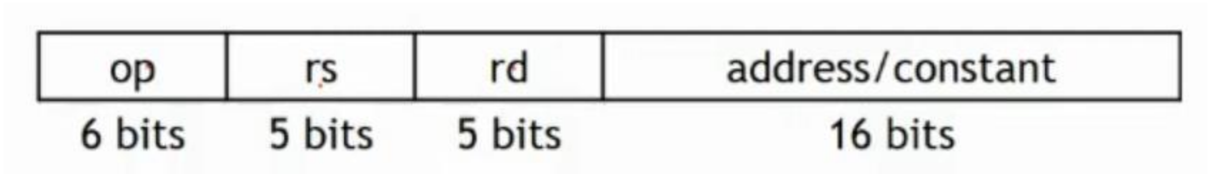
Preferred Solution: Problem is that the updated values are in the system but they are not yet written back to register file. But if we can design a circuit that works on top of pipeline which have access to multiple stages of pipeline then it can help us to take whatever value is needed, to wherever it is needed.

We deal with this problem in 2 parts: A) and B)

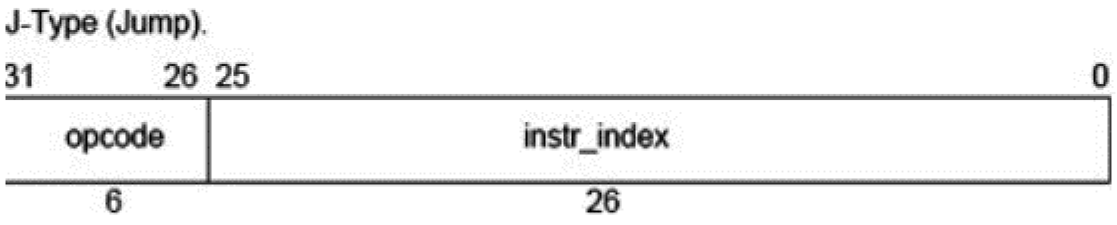
R-TYPE



I-TYPE



J-TYPE



A) Forwarding unit:

Detection Conditions ->

Part 1

- EX_MEM Destination is a register which is needed as operand in instruction currently in ID_EX

  For R-type 2 operands so, need to check both of them

  EX/MEM.RegisterRd = ID/EX.RegisterRs

  EX/MEM.RegisterRd = ID/EX.RegisterRt

- We can have same thing for any instruction in MEM_WB stage

  MEM/WB.RegisterRd = ID/EX.RegisterRs

  MEM/WB.RegisterRd = ID/EX.RegisterRt

Part 2

We need to make sure that instruction ahead in pipeline on whom succeeding instruction depends actually will write something back to register file else forwarding will happen when it is not needed.

So need to check RegWrite Signal

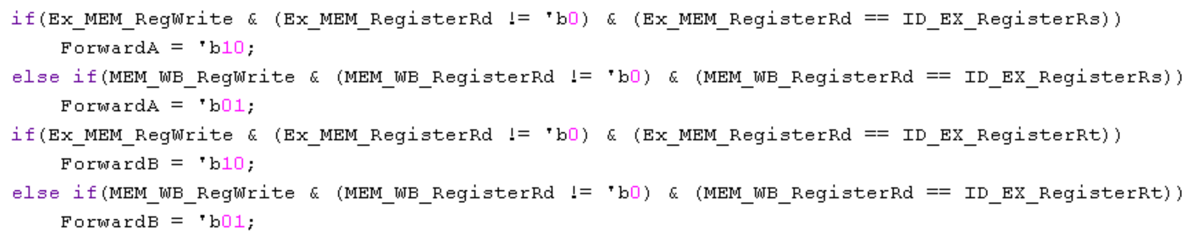
Part 3

We don’t want to forward when a write address is register $0

We want to avoid forwarding non zero result as in register file we have programmed that it is always 0 no matter what but we can end up violating this in instruction by forwarding.

So, Register Rd should not be $0

Final Condition=>



```
if(Ex_MEM_RegWrite & (Ex_MEM_RegisterRd != 'b0) & (Ex_MEM_RegisterRd == ID_EX_RegisterRs))
    ForwardA = 'b10;
else if(MEM_WB_RegWrite & (MEM_WB_RegisterRd != 'b0) & (MEM_WB_RegisterRd == ID_EX_RegisterRs))
    ForwardA = 'b01;
if(Ex_MEM_RegWrite & (Ex_MEM_RegisterRd != 'b0) & (Ex_MEM_RegisterRd == ID_EX_RegisterRt))
    ForwardB = 'b10;
else if(MEM_WB_RegWrite & (MEM_WB_RegisterRd != 'b0) & (MEM_WB_RegisterRd == ID_EX_RegisterRt))
    ForwardB = 'b01;
```

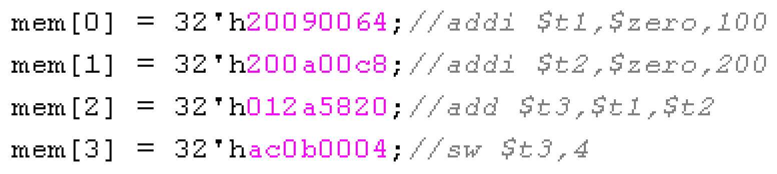
Important Note: Reason for if else in the dealing with hazard source from memory and writeback

If we have multiple instruction acting on same data then we want the latest copy of the data instead of one before that because then we miss one instruction in operation and that is wrong.

So, that’s why we only check for hazard in subsequent stage ie; MEM_WB which has older instruction when we know that the instruction just before doesn’t offer any hazard source.

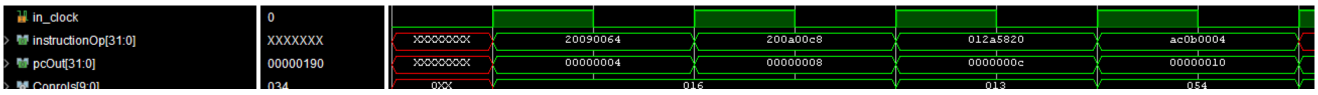
This is achieved by simple if else condition in Verilog code.

Instruction Tested: Data Hazard Present



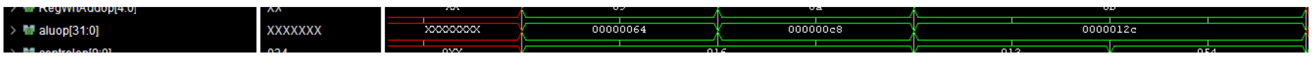
```
mem[0] = 32'h20090064;//addi $t1,$zero,100
mem[1] = 32'h200a00c8;//addi $t2,$zero,200
mem[2] = 32'h012a5820;//add $t3,$t1,$t2
mem[3] = 32'hac0b0004;//sw $t3,4
```

InstructionOp per 4 clock cycles from Fetch to decode:



```
in_clock             0
instructionOp[31:0]  XXXXXXX   XXXXXXXX  20090064  200a00c8  012a5820  ac0b0004
pcOut[31:0]          00000190  XXXXXXXX  00000004  00000008  0000000c  00000010
Conrols[9:0]         034       0XX       016                 013       054
```

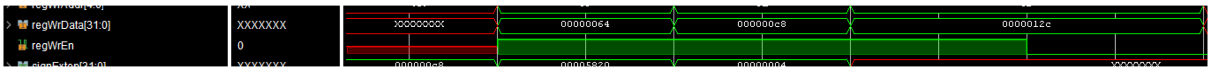
AluOp from execute stage which is as required 0x64, 0xc8, 0x12c



```
aluop[31:0]          XXXXXXX   XXXXXXXX  00000064  000000c8  0000012c
```

Note: we have hazard here as the second and first instruction values are not read from register file but our forwarding unit supplies it

RegWrtData and RegWrite Enable at Writeback stage

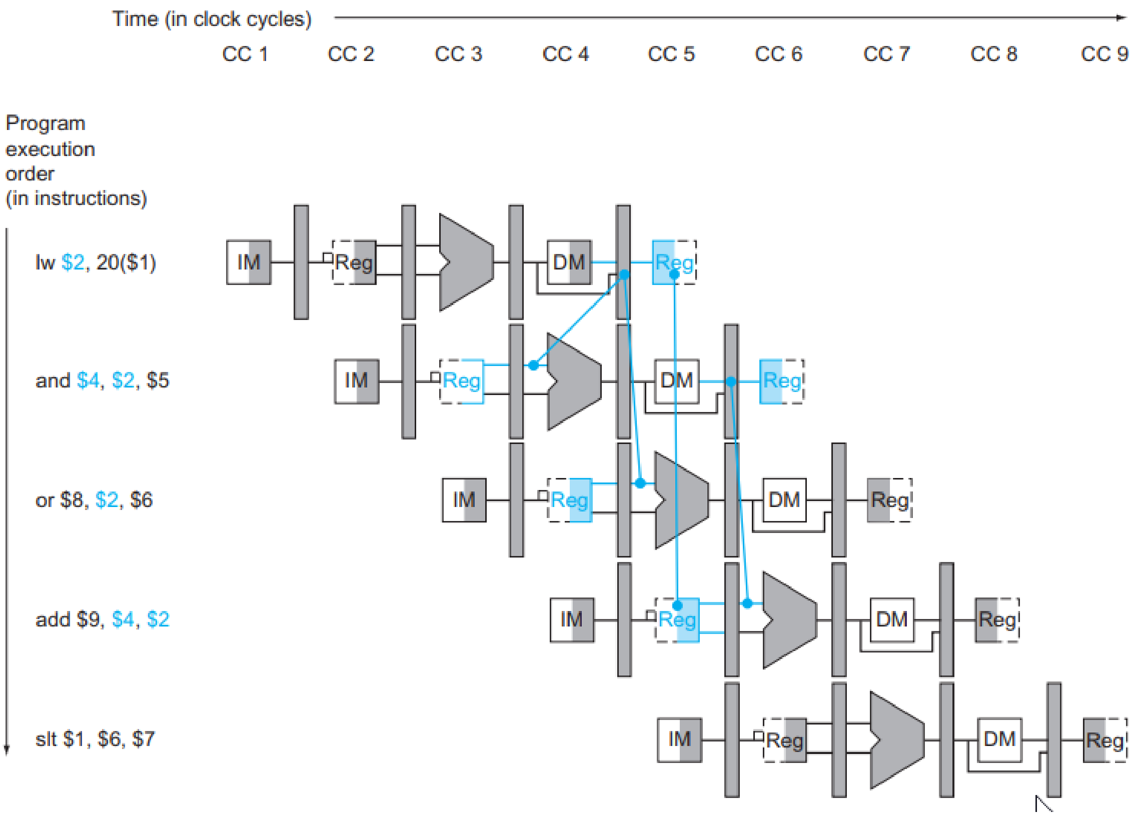


```
regWrData[31:0]      XXXXXXX   XXXXXXXX  00000064  000000c8  0000012c
regWrEn              0
```

B) No Forwarding Path

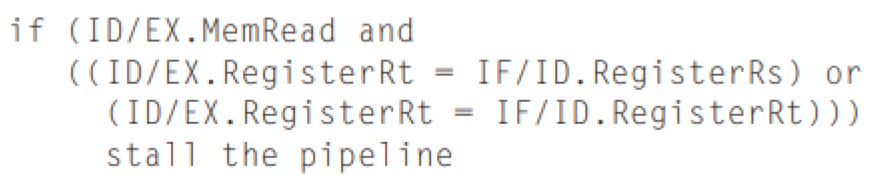
Forwarding works when data is already available with some other stage at time of need ie; destination is after source in time.

It wont work in other case. Like:



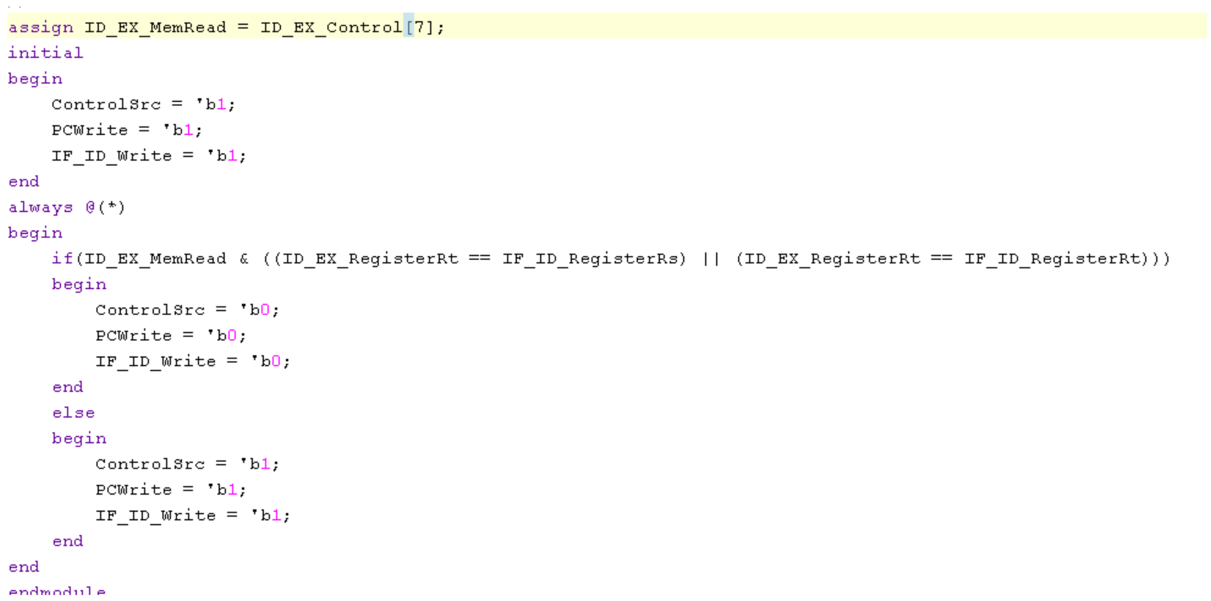
Here only thing we can do is put stalls. So, for that we have a separate circuit named HazardDetectionUnit in design.

LW has longest critical path so, any instruction following that need a nop or stall in between. For that enable added to PC named as PCWrite and ID_EX ie; ID_EX_Write. So when we detect lw we stall pipeline for 1 clk by latching values of PC and ID_EX. And setting controls to ‘b0.



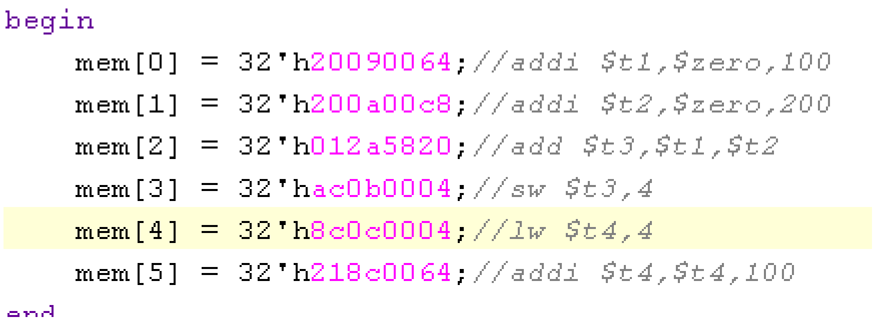
```
if (ID/EX.MemRead and
   ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
     (ID/EX.RegisterRt = IF/ID.RegisterRt))
     stall the pipeline
```

This condition is used to stall.

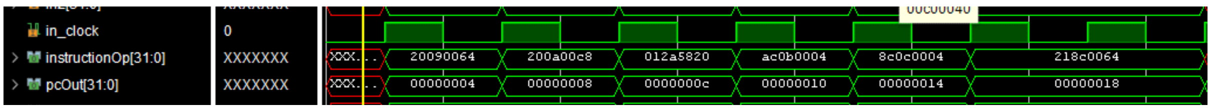


```
assign ID_EX_MemRead = ID_EX_Control[7];
initial
begin
    ControlSrc = 'b1;
    PCWrite = 'b1;
    IF_ID_Write = 'b1;
end
always @(*)
begin
    if(ID_EX_MemRead & ((ID_EX_RegisterRt == IF_ID_RegisterRs) || (ID_EX_RegisterRt == IF_ID_RegisterRt)))
    begin
        ControlSrc = 'b0;
        PCWrite = 'b0;
        IF_ID_Write = 'b0;
    end
    else
    begin
        ControlSrc = 'b1;
        PCWrite = 'b1;
        IF_ID_Write = 'b1;
    end
end
endmodule
```

This is how, it looks like in Verilog code. For lw we have MemToReg signal in control unit. We use that to detect.

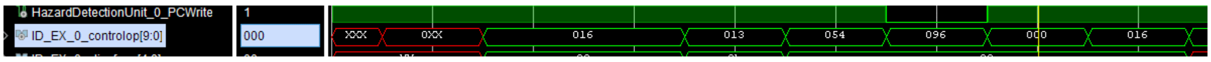


```
begin
    mem[0] = 32'h20090064;//addi $t1,$zero,100
    mem[1] = 32'h200a00c8;//addi $t2,$zero,200
    mem[2] = 32'h012a5820;//add $t3,$t1,$t2
    mem[3] = 32'hac0b0004;//sw $t3,4
    mem[4] = 32'h8c0c0004;//lw $t4,4
    mem[5] = 32'h218c0064;//addi $t4,$t4,100
end
```



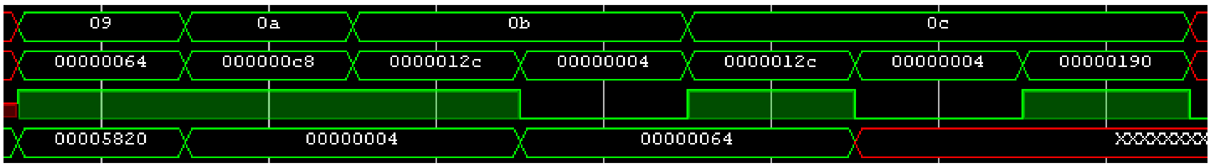
We can see upon addition of 2 instructions which led to the hazard explained above included stall in pipeline ahead so, for 2 clocks the Program Counter and IF_ID remain same.

This stalling can be seen at output of control unit



As soon as this instruction can the control became ‘b0

Now in the writeback stage we check input to RegWrtData input and RegWrtEn



It can be seen that it stores data to $t3, then with hazard again $t4 lw is being called and then with hazard addi is being called whose working is ensured by stall.

Note: Expected output 0x120 + 0x64 = 0x190

<u>Control Hazard and Solutions</u>

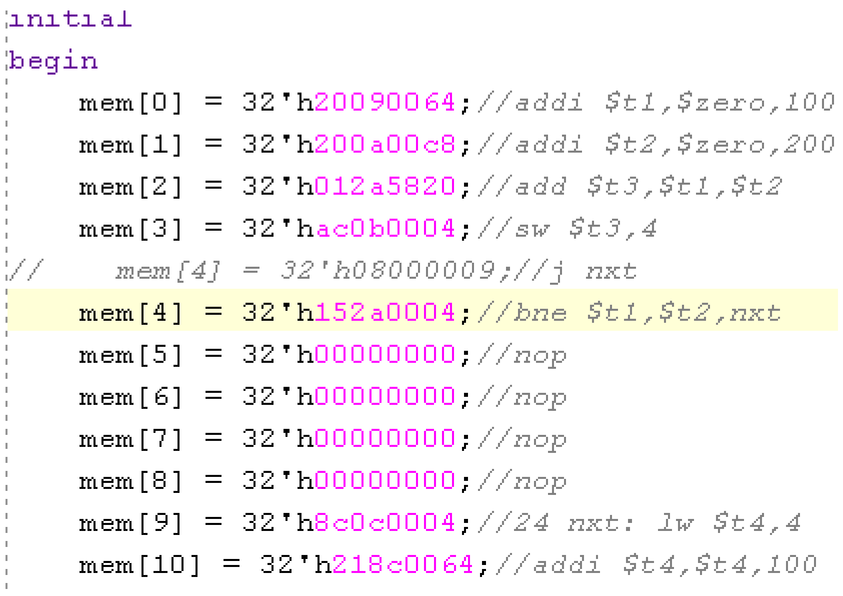
Control Hazard arise when we have make a decision like in bne when we have to decide whether to jump or not to other address.

Problem is that decision is being made later in pipeline so, some instruction might have entered in previous stage which shouldn’t have depending on whether we jump or not.

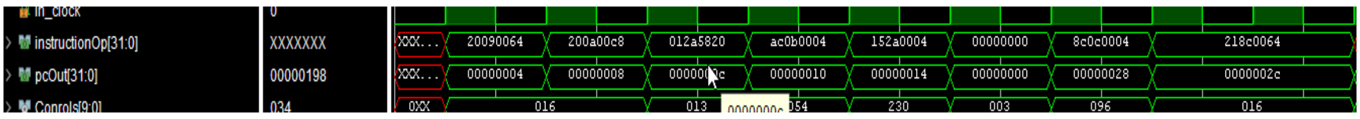
Naïve way: Stalling

Better and Implemented way: Stalling but for 1 clk cycle

By having separate hardware for making decision rather then using ALU we can decide branch and everything in Instruction Decode stage and send the new address to PC from there in combinational way and pass a nop in between. In 1 clk cycle delay we will have the new instruction and no unnecessary instruction will have entered the pipeline.



```
initial
begin
    mem[0] = 32'h20090064;//addi $t1,$zero,100
    mem[1] = 32'h200a00c8;//addi $t2,$zero,200
    mem[2] = 32'h012a5820;//add $t3,$t1,$t2
    mem[3] = 32'hac0b0004;//sw $t3,4
//    mem[4] = 32'h08000009;//j nxt
    mem[4] = 32'h152a0004;//bne $t1,$t2,nxt
    mem[5] = 32'h00000000;//nop
    mem[6] = 32'h00000000;//nop
    mem[7] = 32'h00000000;//nop
    mem[8] = 32'h00000000;//nop
    mem[9] = 32'h8c0c0004;//24 nxt: lw $t4,4
    mem[10] = 32'h218c0064;//addi $t4,$t4,100
```

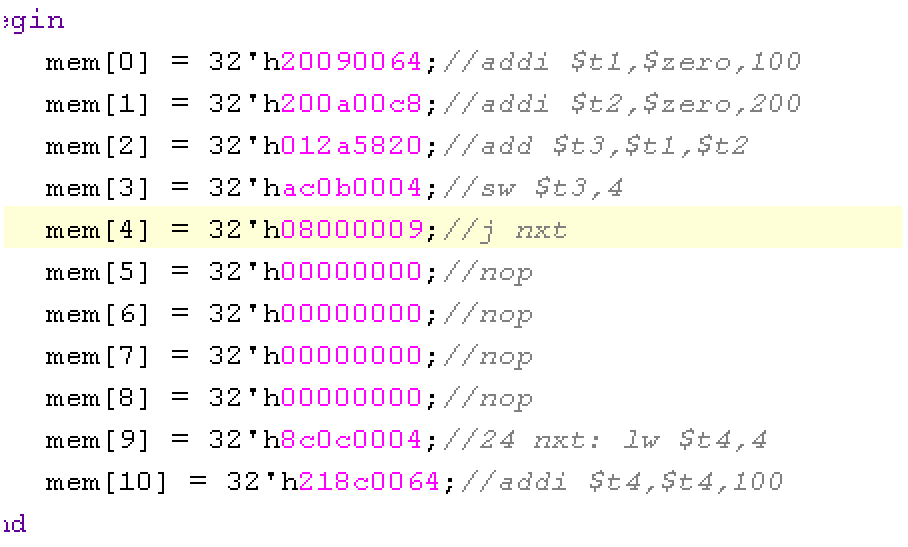


As we can see with 1 nop we jumped to new address.

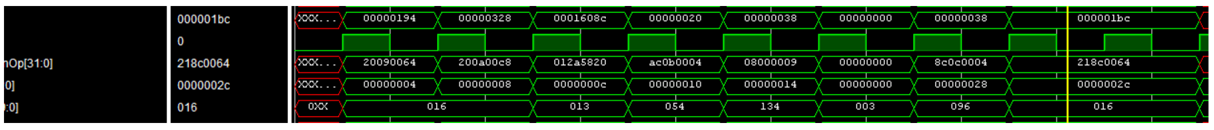
Other solutions involve branch prediction which are not explored in this assignment but will be implemented after this.

Support for Jmp Instruction

Following test run is to show the added support of jump instruction which jumps to new address with 1 clk delay to avoid unnecessary instruction from populating the pipeline.

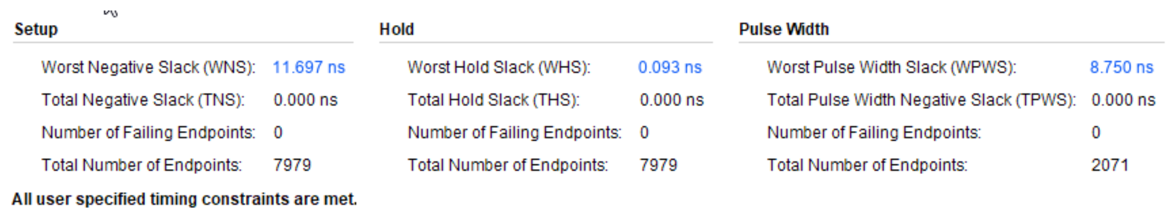


```
egin
   mem[0] = 32'h20090064;//addi $t1,$zero,100
   mem[1] = 32'h200a00c8;//addi $t2,$zero,200
   mem[2] = 32'h012a5820;//add $t3,$t1,$t2
   mem[3] = 32'hac0b0004;//sw $t3,4
   mem[4] = 32'h08000009;//j nxt
   mem[5] = 32'h00000000;//nop
   mem[6] = 32'h00000000;//nop
   mem[7] = 32'h00000000;//nop
   mem[8] = 32'h00000000;//nop
   mem[9] = 32'h8c0c0004;//24 nxt: lw $t4,4
   mem[10] = 32'h218c0064;//addi $t4,$t4,100
nd
```



Synthesis and Timing Results

Entered Clk Constraint = 20ns



Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	11.697 ns	Worst Hold Slack (WHS):	0.093 ns	Worst Pulse Width Slack (WPWS):	8.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	7979	Total Number of Endpoints:	7979	Total Number of Endpoints:	2071

All user specified timing constraints are met.

So, we can run circuit at approx. (20-11.697) ns = 8.303 ns

Ie; 120.4384 Mhz Frequency.