A Report On

MIPS Pipelined Processor with Hardware Solutions for Data and Control Hazards

BY

ABDULTAIYEB VASANWALA	2019AAPS0279G
DHRITIMAN SINHA	2019AAPS0005G
PRIYANSH PARIKH	2019A3PS0288G

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

TABLE OF CONTENTS

SIMULATION WINDOW WITH HAZARDS POINTED OUT

1) addi $1,$0,10

addi $2,$0,20

add $3,$1,$2

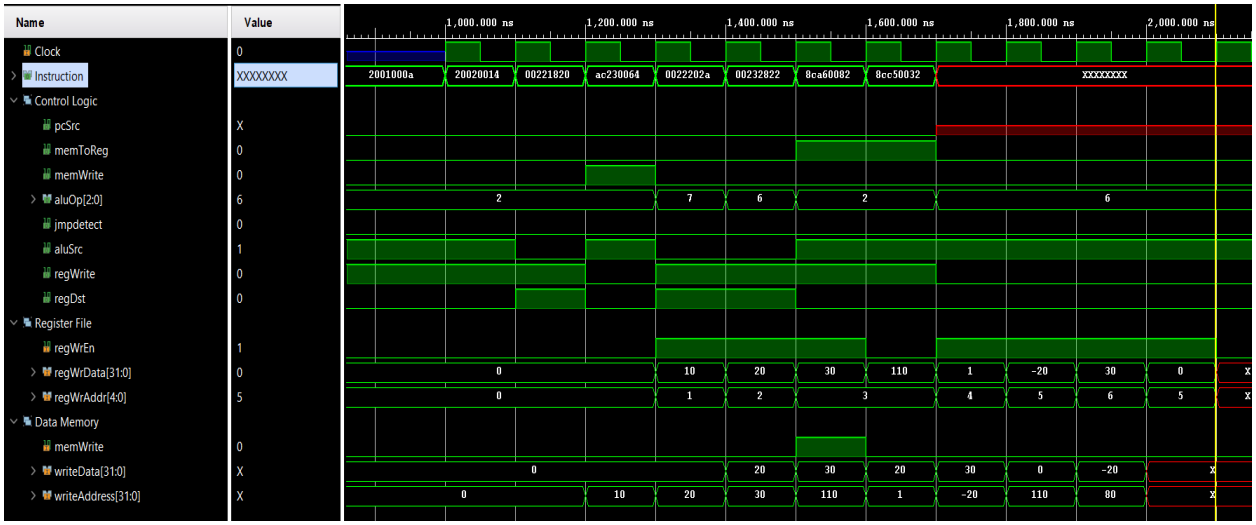
sw $3,100($1)

slt $4,$1,$2

sub $5,$1,$3

lw $6,130($5)

lw $5,50($6)



2) addi $1,$0,20

branch1: addi $2,$2,20

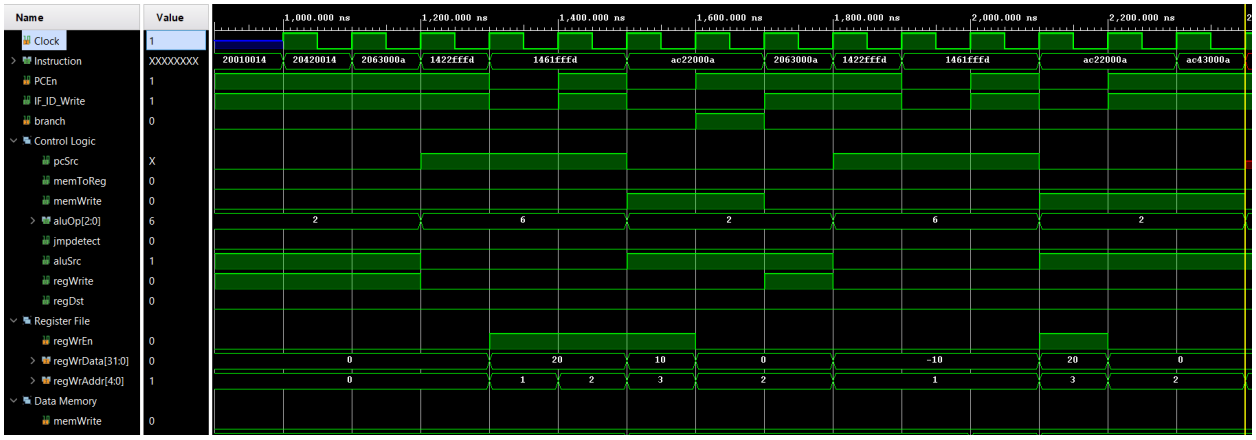
branch2: addi $3,$3,10

bne $1,$2,branch1

bne $3,$1,branch2

sw $2,10($1)

sw $3,10($2)



We have used the above 2 programs to demonstrate how we have resolved different types of data hazards and also the control hazard.

DATA HAZARD DETECTION

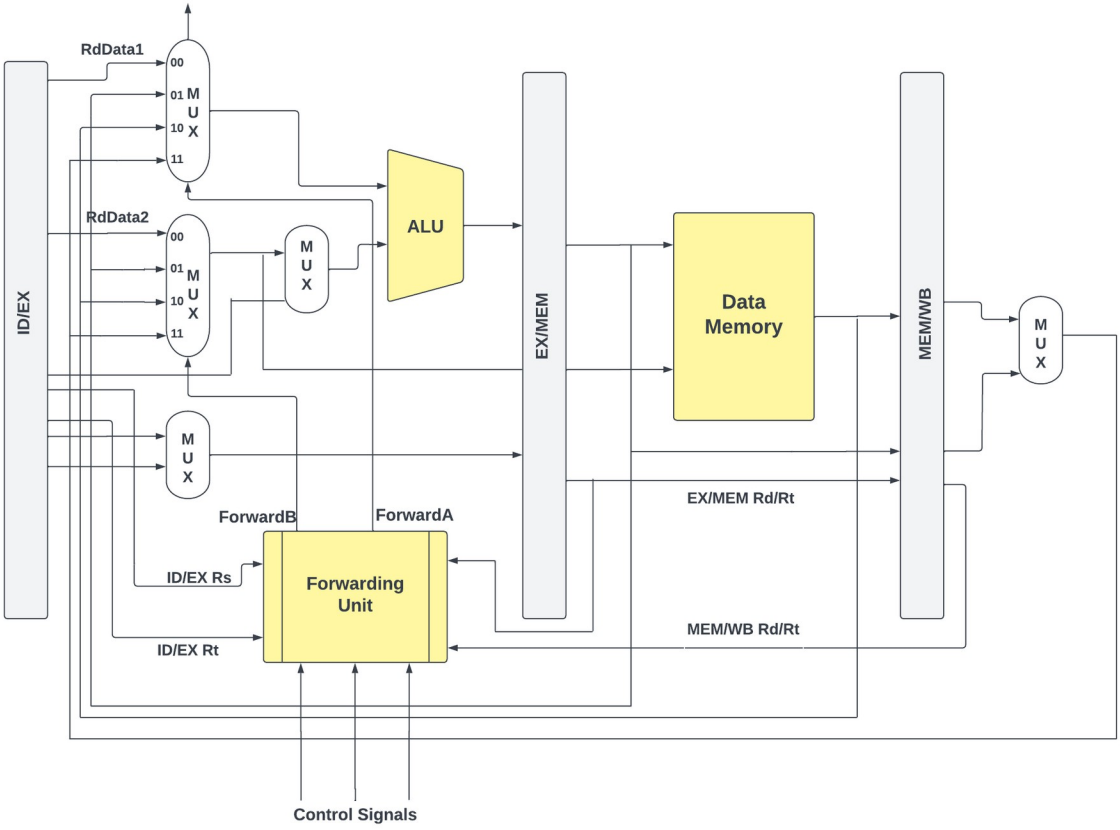


Fig.1

We have categorized the data hazards into 4 different groups. Description about all the four pages are given below

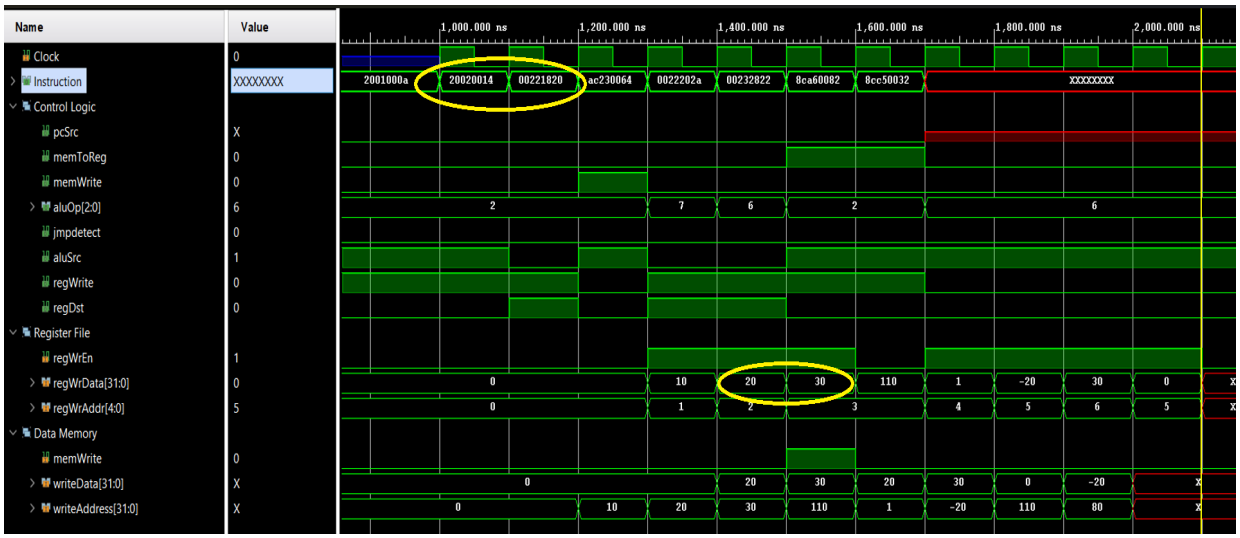
1. *EX Hazard Type 1*

This type of data hazard occurs when an R-type instruction is followed by an R-type or an I-type instruction.

For example,

```
addi $2,$0,20        Instruction 1
add $3,$1,$2         Instruction 2
```

In both cases the 2nd instruction requires the updated value of t1. Hence, we need data forwarding in this case. The ALU result is forwarded from the MEM stage to the inputs of the ALU as shown in Fig.1. The forwarding unit takes the inputs, ID/EX Rs, ID/EX Rt, EX/MEM Rd, EX/MEM RegWrite and accordingly sets the select bits of the two 4:1 muxes. (ForwardA and ForwardB are both set to 01 in this case).



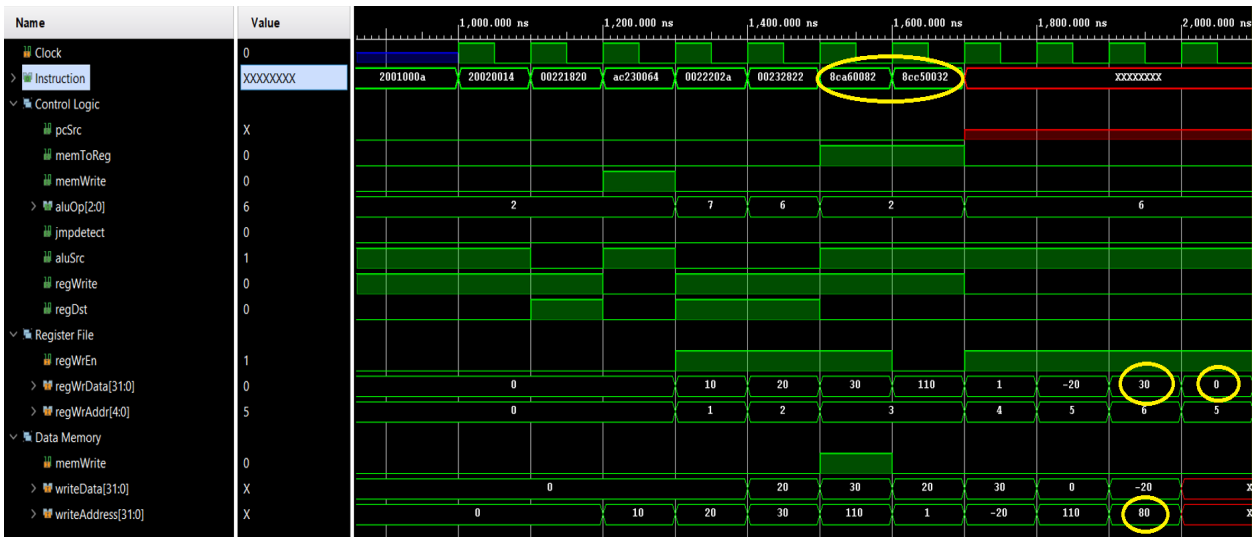
2. EX Hazard Type 2

This type of data hazard occurs when a Load instruction is followed by an R-type or an I-type instruction.

For example,

```
lw $6,130($5)        Instruction 1
lw $5,50($6)         Instruction 2
```

Here the 2nd instruction requires the data that was read from the data memory in the previous instruction. Hence, the data read from the memory is forwarded to the inputs of the ALU as shown in Fig.1. The forwarding unit takes the inputs, ID/EX Rs, ID/EX Rt, EX/MEM Rd, EX/MEM RegWrite, EX/MEM MemToReg and accordingly sets the select bits of the two 4:1 muxes. (ForwardA and ForwardB are both set to 10 in this case).



3. *MEM Hazard*

This type of data hazard occurs when there is a data dependency between the 3rd instruction and the 1st instruction.

For example,

```
addi $1,$0,10          Instruction 1
addi  $2,$0,20   (instruction having no dependency on the 1st instruction)
add $3,$1,$2           Instruction 3
```

In this case the data is forwarded from the Write Back stage to the Execute stage. The forwarding unit takes the inputs, ID/EX Rs, ID/EX Rt, MEM/WB Rd, MEM/WB RegWrite and accordingly sets the select bits of the two 4:1 muxes. (ForwardA and ForwardB are both set to 11 in this case).

4. Register File Forwarding

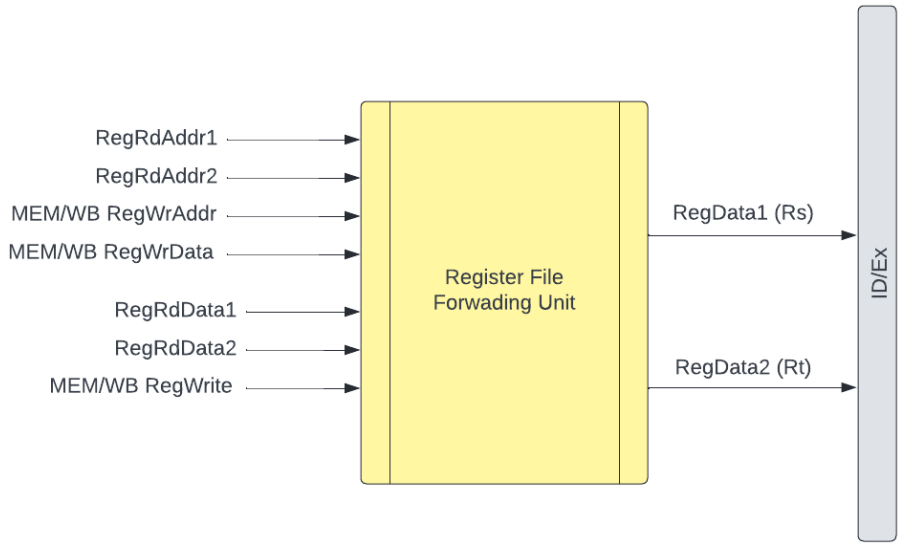


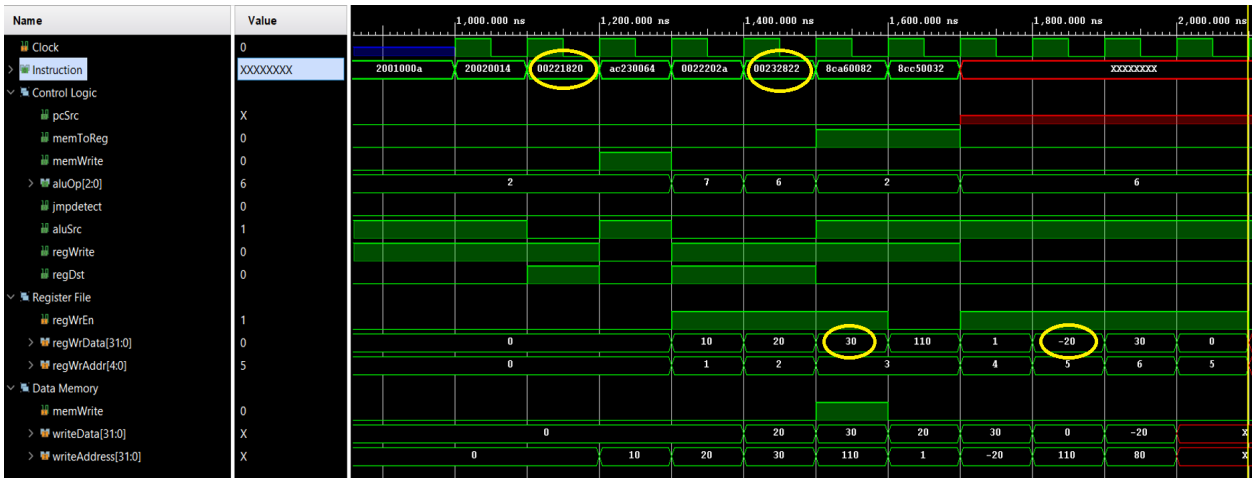
Fig.2

This data hazard occurs due to the clock synchronization of the register file. Even though the data is written onto the WriteData line of the register file, it is actually written into the register on the next clock cycle.

For Example,

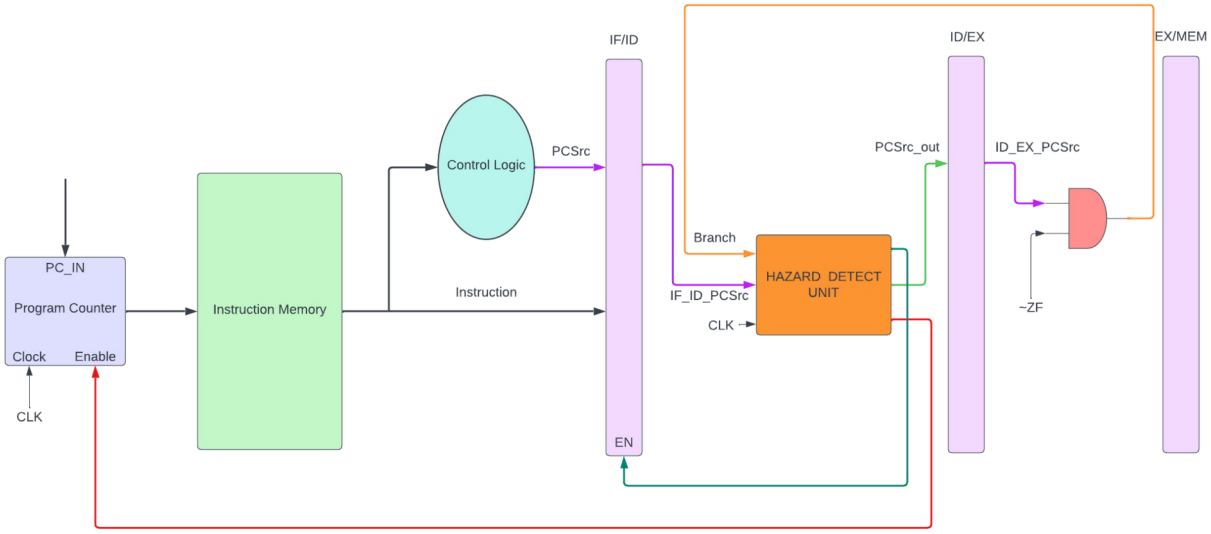
```
add $3,$1,$2
```
– (instruction not dependent on the value of $3)
– (instruction not dependent on the value of $3)
```
sub $5,$1,$3
```

Hence, the data is forwarded onto the Rs and Rt data read from the register file. The inputs and outputs of the Register File Forwarding Unit are shown in Fig.2.



Hence, the Data Hazards are resolved by appending the Forwarding 4:1 muxes and the Register File forwarding Unit to the pipelined des MIPS processor.

CONTROL HAZARD DETECTION

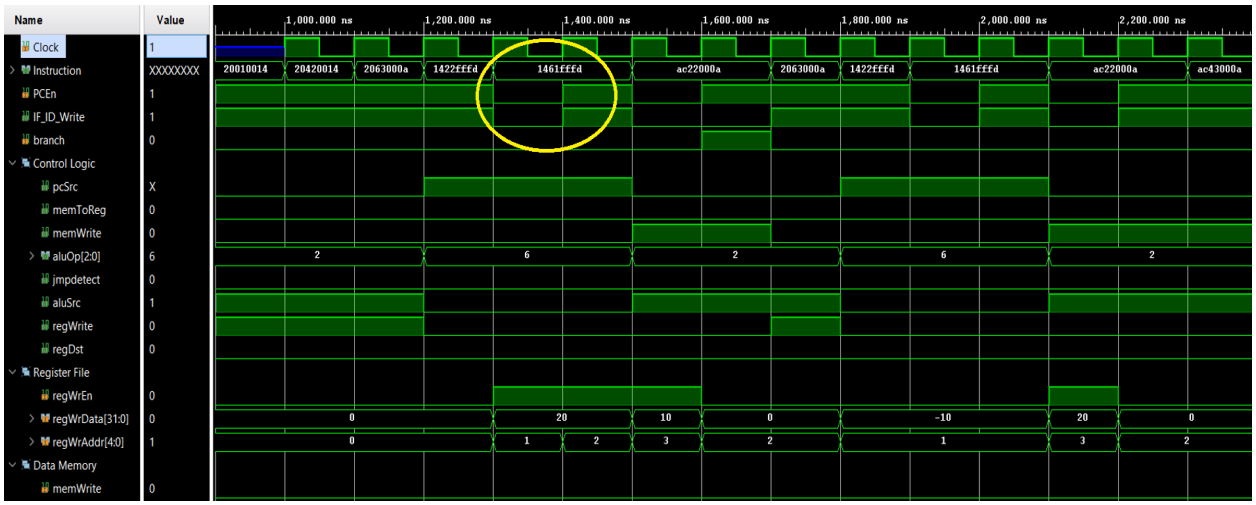


How this technique works in tackling control hazard due to bne instr

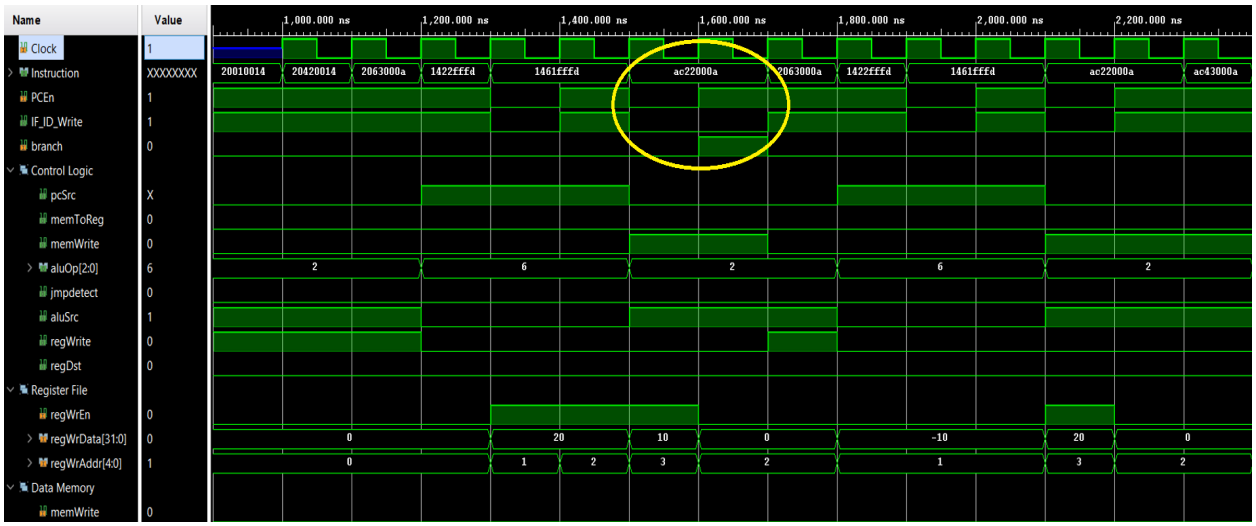
1) The bne instruction is detected by the Control Logic at t = 0 for eg. The PCSrc signal now asserts because of branch instruction detection.
2) At the next clock t = T_{CLK} bne instruction enters the decode stage. IF_ID_PCSrc now is high because it is pipelined PCSrc entering the decode stage. The instruction now at the input of the IF/ID pipeline is the instruction just after branch instruction. The PCSrc signal at the pipeline input becomes low if the next instruction isnt branch
3) There is an internal 2 bit counter in the HDU unit which helps in the enabling and disabling of the program counter and the IF/ID pipeline.
4) At this stage Branch signal is low, IF_ID_PCSrc is high and count in the HDU is 0 with the counter enabled by the IF_ID_PCSrc signal. Thus, the PCSrcOut signal is high now.
5) The HDU disables the program counter using the invert of the PCSrcOut signal. It also disables the IF/ID pipeline as PCSrcOut is high.
6) At the next clock t=2*T_{CLK} the PCSrcOut enters the execute stage as ID_EX_PCSrc. The counter in the HDU increments to value 1. This causes the program counter to again get enabled as PCSrcOut is low now.
7) The ID_EX_PCSrc is already high now because PCSrcOut signal was high in the previous clock cycle. In the execute stage the ALU checks if the values of the two operands are the same. If yes then branch isn't taken and

the normal execution of the program continues. If not then the program needs to branch to the appropriate instruction address as given by the user. This is checked using the “AND” gate in the execute stage. If the zero flag is low and ID_EX_PCSrc is high then branch is to be taken. This controls the MUX (just before the program counter) which supplies the correct address of the instruction which has to be executed.

8) If the branch is not taken then before the next clock approaches the HDU enables the pipeline so that the instruction just after branch can enter the pipeline.
9) If the branch is taken then the pipeline remains disabled for another clock cycle so that the branched instruction can reach the instruction input. This is done because the program counter is clock synchronized.



The above image shows how PCEn and IF_IDEn are made low for one clock cycle in case when a branch is not taken.

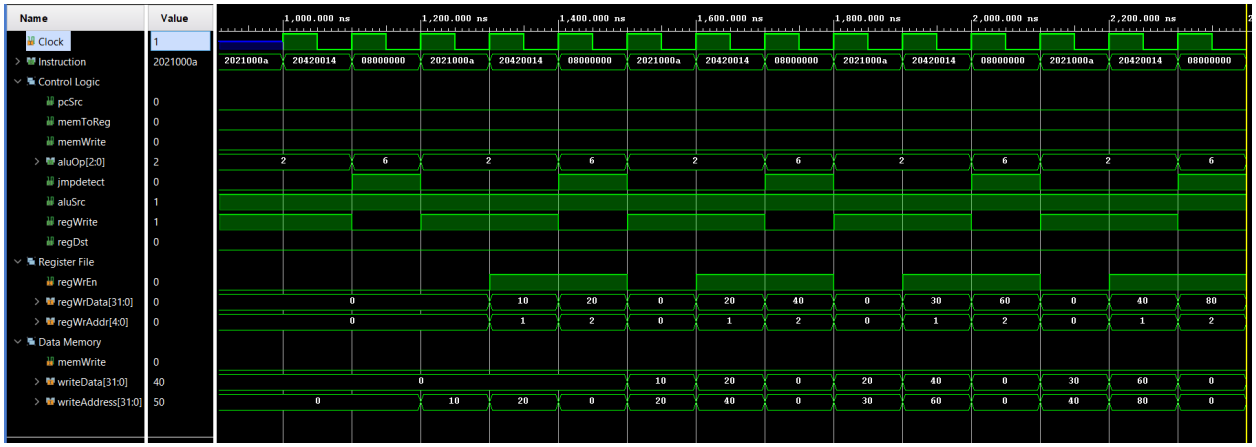


The above image shows how PCEn and IF_IDEn are made low for 1 and 2 clock cycles respectively. This is when the program branches to the required instruction. As the program counter is clock synchronized, an extra clock cycle stall is required in this case.

Thus this method effectively involves at most two clock cycle stalls.

One thing to note here is that control hazards may also be created by unconditional jump instruction. However, the given design addresses this issue without flagging it as another hazard. This is due to the fact that the decoding for the jump address takes place entirely in the IF stage without the involvement of any pipelines. As soon as the jump instruction is loaded by the PC, it loads the effective jump address onto the input of the PC immediately, ready to be loaded on the next clock cycle. Thus no additional circuitry was required for the hazard detection for the jump instruction. This can be shown through a simple example

```
loop: addi $1,$1,10
addi $2,$2,20
j loop
```



****