PIPELINE IMPLEMENTATION DETAILS

(phase 2 code is in the file pipeline.cpp)

***This simulator implements MIPS PIPELINE and this report gives some details about the implementation.**

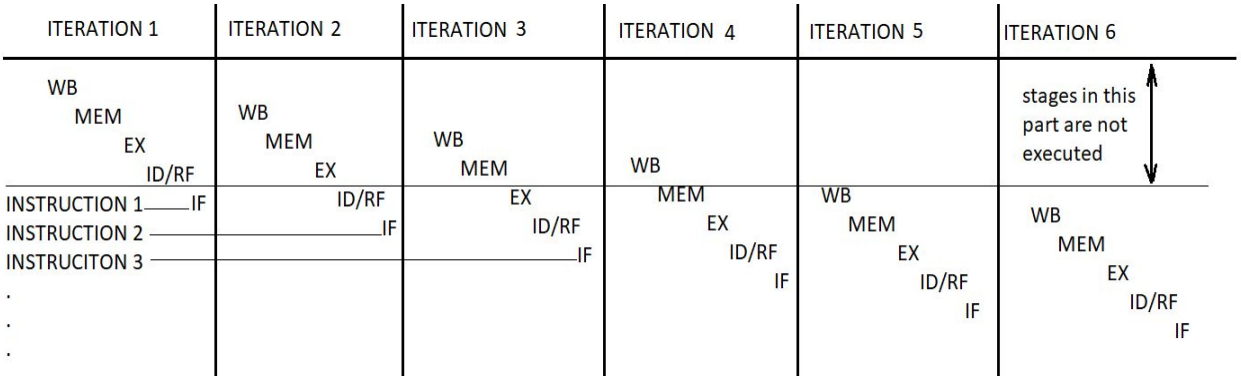
class processor

- Contains fetch(), idrf(), execute() ,memrw(), wb() methods
- Each of the methods simulate the corresponding mips pipeline stages
  To implement pipeline :
- In the processor object : id_rf = false, ex = false, mem = false, wbr = false

```
while(){
    if(wbr){ wb(); }
    if(mem){ memrw(); } // memrw() makes wbr = true
    if(ex){ execute(); } //execute() makes wbr = true
    if(id_rf){ idrf(); } // idrf() makes ex = true
    fetch() // makes id_rf = true
}
```

For rest of the execution id_rf,ex,mem,wbr are true forever

Conditional checks are done to make sure that the first idrf() happens only after the first fetch() , first execute() after the first idrf() and so on



When a method starts executing, first it reads all the necessary values from/ updated by the previous stage, since most of the values are collected in the id/rf stage, These are passed onto the next stages in this way.

fetch() - does the instruction fetch from the text array using the pc
idrf() - decodes the instruction, Checks for RAW dependencies
exdr - pointer that stores the destination register address in EX stage
memdr - pointer that stores the destination register address in MEM stage
If rs is the source register

```
if(rs==exdr){
        depends = true;
        //data forwarding is done
        rs = result (// result is the EX/MEM latch)
}
else if(rs==memdr){
        depends = true;
        //data forwarding is done
        rs = memreg(// memreg is the MEM/WB latch)
}
```

If forwarding does not resolve the hazard the stalls are used

Since all the methods are actually executing in a sequential manner , which is not the case when actual pipeline is implemented (stages execute in parallel) , and thus even though some of the stalls don't make any sense for this simulator they are included keeping in view the hardware implementation of the real MIPS pipeline

If there is a load instruction,load lock is enabled (ld_lock = true)
And If the next instruction has a RAW dependency on the load then all the next pipeline stages are locked for this instruction and the same instruction is again fetched so that when idrf is again done for this instruction the

correct register values will be available in the memreg (latch of the MEM stage)

Comparisons on registers are done on the registers so that in case of branches the decision is done in idrf stage itself and Branch target address is also calculated

But since Branch prediction is not done branch lock is enabled so that the next instruction gets stalled by one cycle

(Actually in the code nothing is done to stall for a control hazard because all methods get executed sequentially as shown in the figure and hence the fetch() of next instruction is done only after idrf() of the branch)

execute() - does all the arithmetic and logical operations
memrw() - memory (dataseg array)access
wb() - writes to the destination registers, terminates the program if jr $ra is seen

STALLS :

- Structural hazards are ignored.
- Branch prediction is not done in this simulator
- Data forwarding is done where ever possible but still, the following sequence of instructions will need stalls

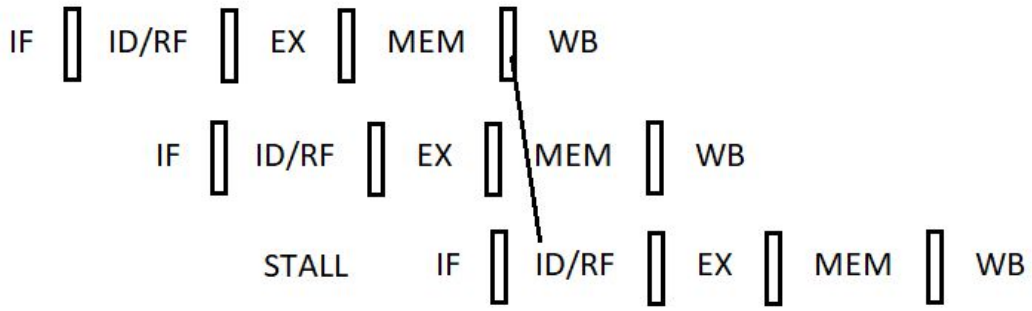
1. **A delayed load** - instruction having RAW with immediately preceding load should stall by 1 cycle. (except : lw - sw RAW can be handled with data forwarding from MEM stage latch of lw to MEM stage of sw and hence no stall is needed)

2. **Data dependencies in branch instruction** - this is because the register comparison operation is shifted to the id/rf stage (no branch prediction)

   Explanation with examples :

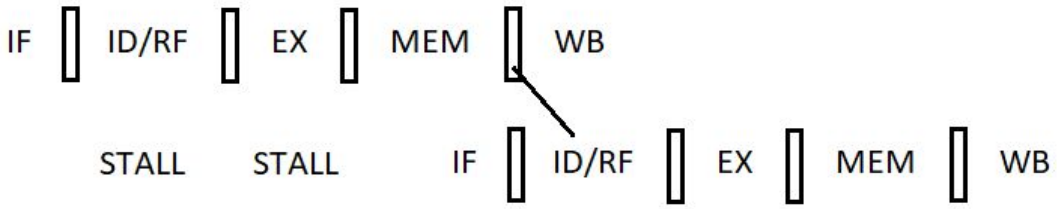
   A. If a comparison register is a destination of 2nd preceding load instruction, 1 stall needed

      lw $t1,0($s0)
      add $t2,$t3,$t4
      beq $t1,$0, target



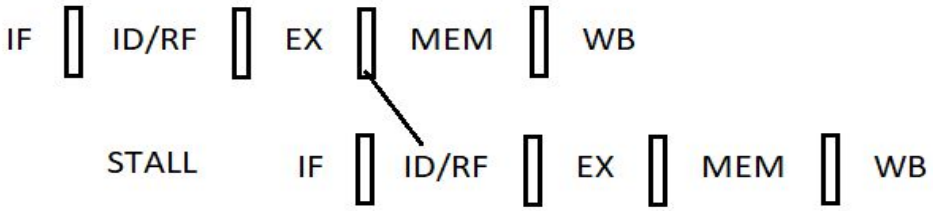
   B. If a comparison register is a destination of immediately preceding load instruction, 2 stalls needed

      lw $t1,0($s0)
      beq $t1,$0, target



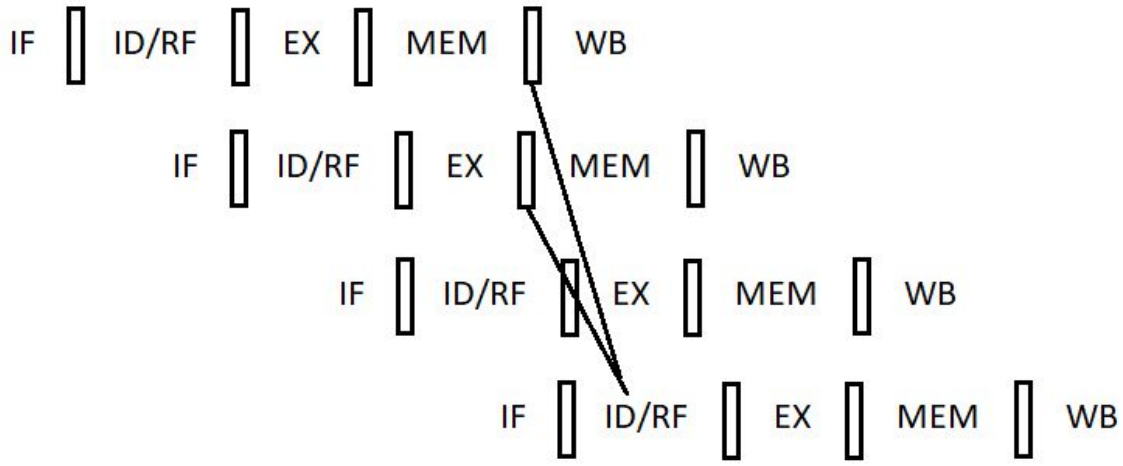
C. If a comparison register is a destination of immediately preceding ALU instruction, 1 stall needed

```
add $t1,$t3,$t4
beq $t1,$0, target
```



D. If a comparison register is a destination of 2 nd or 3rd preceding ALU instruction
In this case stall is not needed data forwarding will help

```
add $t1,$t2,$t3
add $t4,$t5,$t6
add $t2,$t3,$t5
beq $t1,$t4, target
```



In Spite of the above cases it is justified to have register comparison in id/rf itself because, if it is done in the EX stage then also 2 stalls are always necessary after branch (without branch prediction) but when done in ID/RF stage only if there is a data dependency stalls occur before branches

And in such cases the number of stalls occuring are the same in either of the implementations . The following is an example showing this fact

1. Register comparison in EX stage

```
lw $t1,0($s0)
        STALL
beq $t1,$0, target
        STALL
        STALL
```

2. Register comparison in ID/RF stage

```
lw $t1,0($s0)
        STALL
        STALL
beq $t1,$0, target
```

STALL

As an added advantage if the branch does not have any dependencies then one stall is being avoided. Further if branch prediction is done this 1 stall also can be removed depending on the accuracy of predictions

3. **Control hazards** - all branches and jumps have a stall after them because only after the completion of the id/rf stage the branch target address along with the branch decision(in case beq , bne) would be ready and the next instruction fetch should be done after this id/rf stage.