

# Advanced Instruction Re-Ordering: Dynamically Scheduled Pipelines

## Handout 10

October 19, 2004

Shoukat Ali

shoukat@umr.edu



UNIVERSITY OF MISSOURI-ROLLA  
The Name. The Degree. The Difference.

1

## Can HW Be Used To Re-Order Instructions?

- we have seen how pipelining can cause true dependencies to turn into RAW hazards
  - we have seen how RAW hazards can cause incorrect execution
  - we have seen how to avoid RAW hazards by
    - re-ordering the code (done by compiler)
    - “nop” insertions (done by the compiler when re-ordering does not work)
- static scheduling
- pipeline forwarding (done by the HW)
  - stalling the pipeline (done by the HW, specifically by “pipeline interlocking HW”)
- *Can the HW re-order the instructions?*

2

## Why Should HW Be Asked to Re-Order?

- some dependencies cannot be detected at all by a compiler
  - sw followed by a lw
- some dependencies are dealt with very inefficiently by a compiler
  - lw followed by an instruction that uses loaded data
  - compiler does not know how many independent inst are needed after lw (1 for perfect \$, more for imperfect \$, how many more?)
- if compiler does re-ordering, and the pipeline implementation for an ISA changes in future, a new compiler will have to be written, and every single user program will have to be re-compiled
  - but if HW does all re-ordering
    - easy to write a compiler
    - programs easily port from one implementation to another implementation

3

## Simple Pipelines: Review

- fetch an instruction
- decode it, read operands, and send to functional unit (FU)
  - ALU, multiply, divide, or other unit
  - note: instruction is not sent to the FU if there is a hazard (data, control, or structural) that cannot be fixed with forwarding
    - hazard is detected by "pipeline interlocking logic"
- execute the instruction
- access memory
- write back results, if any, to the register file

4

## Simple Pipelines: Problem

- motivating example:

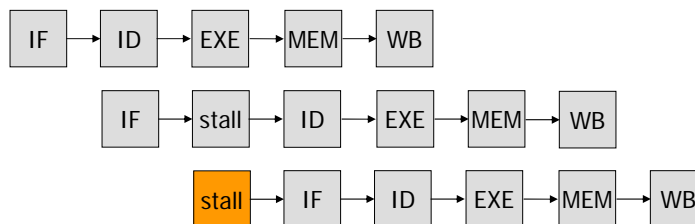
```
ld r2, 0(r1)
add r3,r2,r7
sub r12,r11,r13
```

- as soon as the add instr is decoded, hardware knows r2 has not yet been written, and so add must be stalled for one clock cycle
  - but why does sub have to wait behind add?
- can HW pull sub out from behind add and issue it?

5

## Simple Pipelines: Problem Illustration

- simple in-order pipeline

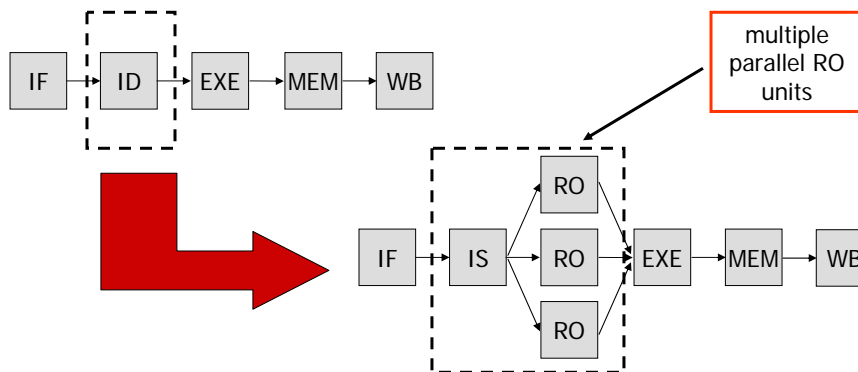


- instruction is stalled in decode (ID) stage
  - this stalls later instruction in IF
- fundamental pipeline change needed
  - stalled dependent instructions must not block later independent instructions

6

## Pipeline Modification: Split the ID Stage

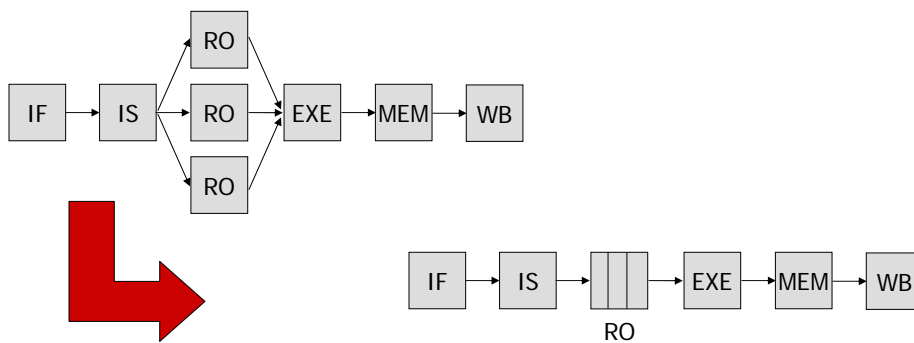
- split ID into two new stages
- “issue”: instruction is decoded, stays in this stage until FU is free
  - at that time instr proceeds to “read operands” stage
- “read operands”: waits in this stage until no RAW hazards and then operands are read and instr proceeds to EXE stage



7

## Effect of Pipeline Modification

- the “read operands” stage buffers a “stuck” instruction until its source operands are available
  - RO stage is like a rest area on a highway
- this allows later independent instructions to make progress
  - “fetch” and “issue” are not stalled

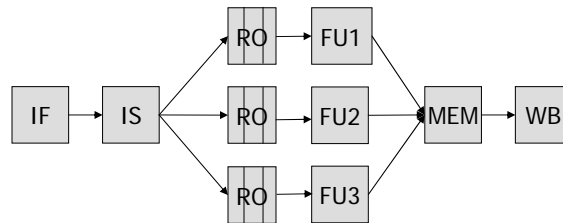


simplified representation

8

## May Ask in Exam: Multiple Functional Units

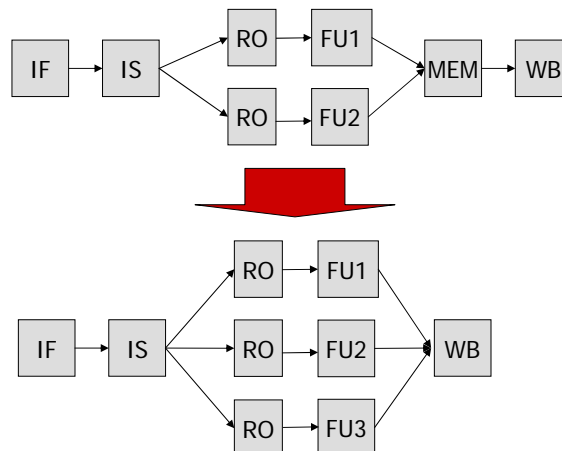
- previous pipeline modification would NOT make sense until:
  - there were multiple functional units, or
  - pipelined functional units, or
  - both
- modern processors have many functional units working in parallel
- MIPS R10K has
  - one each of FP add, FP multiply, FP divide, and FP sqrt units
  - two “integer” units for integer operations, branches
  - one load/store unit for loads and stores



9

## Memory As a Functional Unit

- memory will be treated as a functional unit from now
- also, multiple RO units per FU will be understood until I explicitly say otherwise



10

## Dynamic Scheduling

- definition: a pipelining arrangement in which all instructions pass through the issue stage in order, but can be stalled or bypass each other in the read operands stage and thus enter execution out of order
- brief definition: a HW implementation for in-order issue, out-of-order execution

11

### Dynamically Scheduled Pipelines

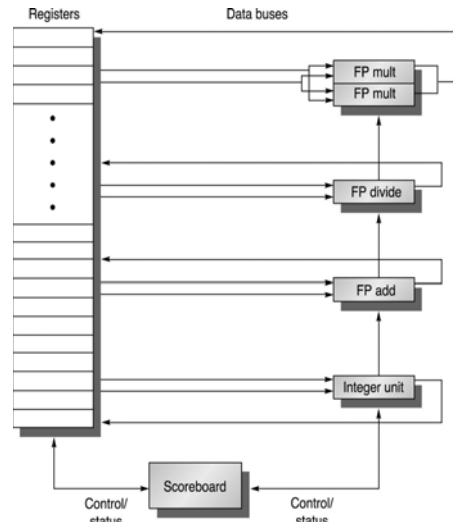
Scheduling Done  
Using Scoreboard

Scheduling Done Using  
Tomasulo's Algorithm

12

## Scoreboard

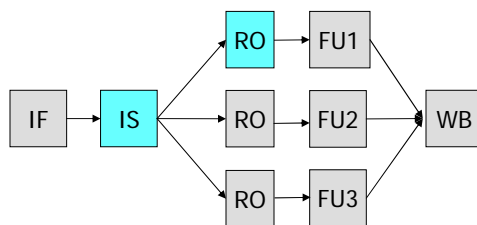
- scoreboard: a data structure that
  - resolves RAW, WAR, WAW hazards at run time AND
  - implements dynamic scheduling
- once an instruction is fetched, the scoreboard takes over its control
  - determines when it will graduate from each pipeline stage
- scoreboard first used in CDC 6600



13

## Scoreboard: Issue Logic Details

- current status: instruction X has been sent from IF to IS



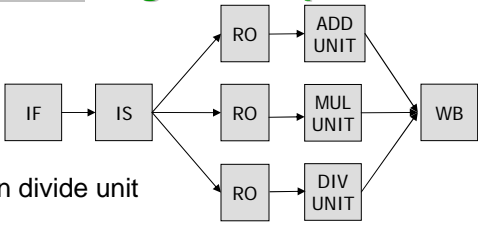
- send X from Issue to Read Operand stage if requested FU (integer unit, mult unit, div unit) needed by X is free AND no other active instruction is yet to write to destination register of X
- note that
  - 1<sup>st</sup> AND condition ensures structural hazards are avoided
  - 2<sup>nd</sup> AND condition ensures that WAW hazards are avoided

14

## Scoreboard: Issue Logic Example

- example
 

div f0, f2, f4  
 add f6, f0, f8  
 mul f12, f10, f14

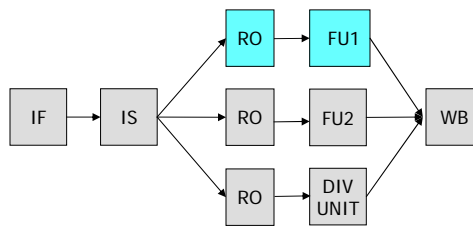

- assume instr div is executing in divide unit
- add will be fetched one CC after div is fetched
  - IS→RO one CC later because add unit is free & dest(add) ≠ dest(div)
  - will be stalled in RO until div gets done
- mul be fetched one CC after add is fetched
  - IS→RO one CC later because mul unit is free & dest(mul) ≠ dest(add,div)
  - will not be stalled in RO, will enter execution, and will write back



15

## Scoreboard: Read Operands Logic Details

- status: instruction X has been sent from IS to RO
- scoreboard monitors the availability of X's operand
  - a given operand is available if no earlier issued active instruction is going to write to it
- when ALL operands are available, scoreboard signals the FU to read the operands from register file; read operand stage finishes
  - this resolves RAW hazards
  - note: operands are always read from the reg file
    - no forwarding



16



### Scoreboard: Execution Logic Details

- status: instruction X has been sent from RO to FU
- the FU begins execution, and notifies the scoreboard when finished executing

17

### Scoreboard: Write Result Logic Details

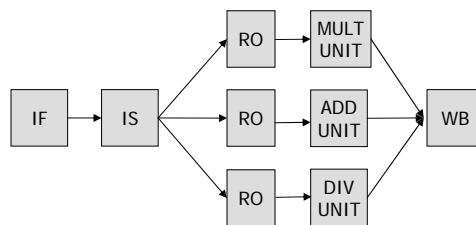
- status:
  - instruction X has been sent from FU to WB
  - result from the execution of instruction X is available but has not been written to the register file yet
- scoreboard examines the source registers of the instructions waiting in the Read Operand stage for **any** FU
  - if X wants to write to any of those registers, X's write back must wait until the prior instructions have read their operands
    - this resolves WAR hazards

18

## Scoreboard: Write Result Logic Example

```
div f0, f2, f4
add f10, f0, f8
mult f8, f7, f14
```

- assume instruction **div** is executing in divide unit
  - will **add** be fetched? be issued? its operands read? be executed? its result written back?
  - will **mult** be fetched, be issued, its operands read? be executed? its result written back?



**add** will be fetched, issued, but will stay in RO stage until f0 is produced by **div**

**mult** will be fetched, issued, its ops read, executed but its result will not be written back until f8 has been read by **add**

19

## Scoreboard Functioning: A Summary

- scoreboard definition: a data structure for resolving RAW, WAR, WAW hazards AND implementing dynamic scheduling
- when instr X is fetched, the scoreboard takes over its control
  - scoreboard constructs a record of X's data dependences
  - scoreboard determines if X can read its operands
    - if yes, instruction is dispatched to the FU it needs
    - if no, instruction is stalled in the RO stage
      - scoreboard constantly monitors X's operands to see when they get ready
  - for the instructions that finish their execution, scoreboard determines if they can write their results back

20