CpE 313: Microprocessor Systems Design

Exam 2 Review

November 04, 2004

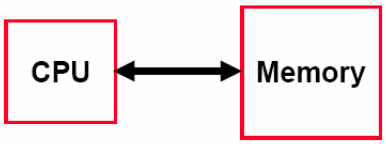
Shoukat Ali

shoukat@umr.edu

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

1

CPU-Memory Bottleneck



Performance of high-speed computers is usually limited by memory *bandwidth* & *latency*

- ***Latency (time for a single access)***
  Memory access time >> Processor cycle time
- ***Bandwidth (number of accesses per unit time)***
  if fraction ***m*** of instructions access memory,
  ⇒ 1+***m*** memory references / instruction
  ⇒ CPI = 1 requires 1+***m*** memory refs / cycle

2

Principle of Locality

- Principle of locality: Programs access a relatively small portion of their address space at any instant of time
  - Temporal locality (time):
  - Spatial locality (space):

Caches are a mechanism to hide memory latency based on the empirical observation that the stream of memory references made by a processor exhibits locality

	PC
...	96
loop: ADD r2, r1, r1	100
SUBI r3, r3, #1	104
BNEZ r3, loop	108
...	112

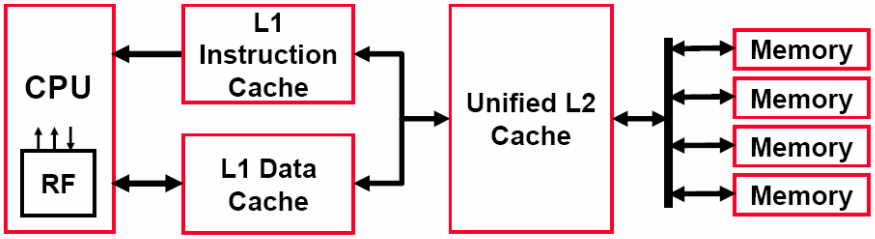
What is the pattern of instruction memory addresses?

3

Split Caches: Why? ... Split Memory: Why?

Split instruction & data primary caches (on-chip SRAM)

Multiple interleaved memory banks (DRAM)



Multiported register file (part of CPU)

Large unified secondary cache (on-chip or off-chip SRAM)

4

4 Qs for Cache Designers

- Q1: On a miss, when a new block is brought from memory, where can the block be placed in the cache?
  (Block placement)
- Q2: On a cache access, how does the HW know if the requested block is in the cache?
  (Block identification)
- Q3: On a miss, which block should be replaced to make room for the new block?
  (Block replacement)
- *Q4: What happens on a write?*
  (Write strategy)

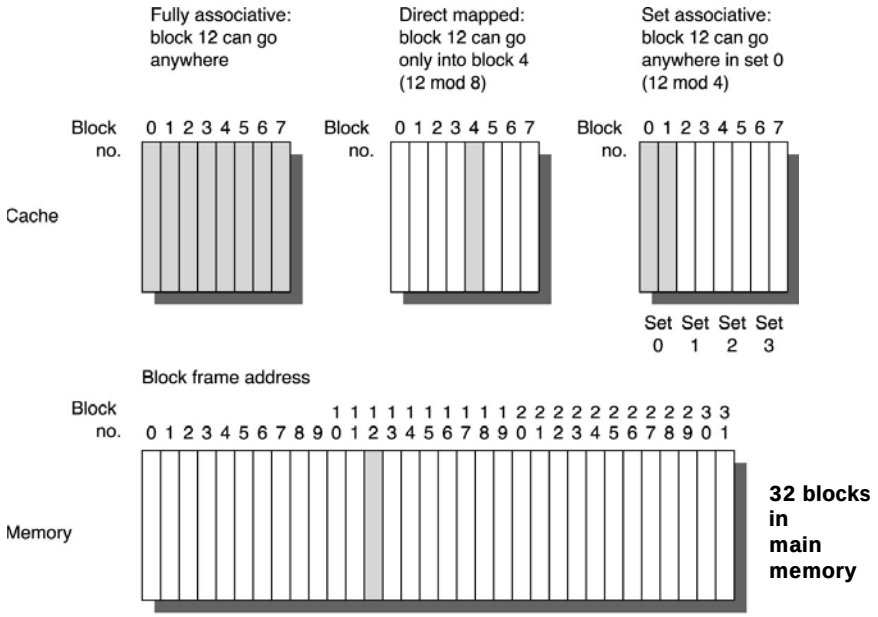
5

Q1: Where Can a Block Be Placed?

- direct mapped: each block has only one frame where it can reside
  - frame no. = (Block address) mod (# of frames in cache)
- fully associative: each block can be placed in any frame in cache
  - frame number = <<no frame number>>
- set associative: each block can be placed in a restricted set of frames in the cache
  - a block is first mapped to a set, and then placed anywhere in set
    - set number = (Block address) mod (# of sets in cache)
  - n-way set associative cache → there are *n* frames in one set
  - cache associativity = # of frames per set

6

Example of Set Associativity in Caches

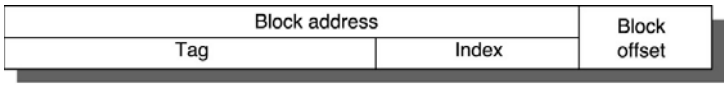


Cache Index

- purpose of cache index: points to that set in the cache which might have the block or where the block might be placed
- size of cache index = $\log_2$(# of sets in cache)
  - # of sets in cache = # of frames in cache / # of frames per set
    = # of frames in cache / associativity
- value of cache index = remainder of (block address) / (# of sets in cache)
  - trick: index = lower *n* bits of block addr; *n* = cache index size

Tag Part of an Address

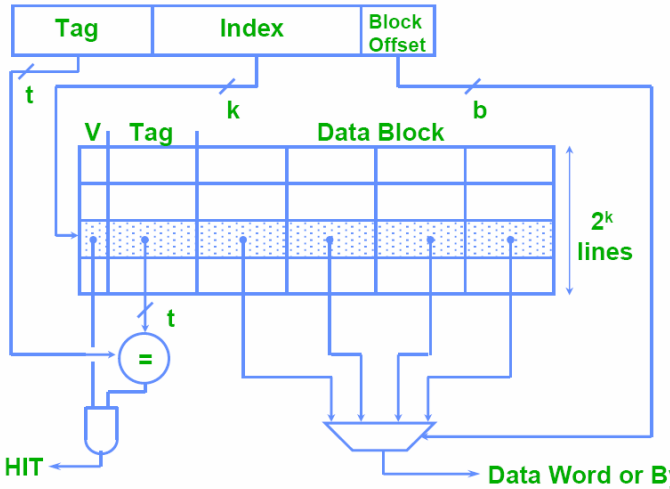
- A = address generated by the CPU
- block addr = A without lower $\log_2$(block size) bits
- index bits = lower I bits of block addr
  - I = $\log_2$(# of sets in cache)
- tag bits = A without index bits and without block offset bits



9

DMC: How Is A Word Found If It Is In Cache?

cache set at $index_A$ is accessed then
[(tag_A is compared with tag in the set, valid bit is checked) and in parallel
(word is transferred to CPU)]



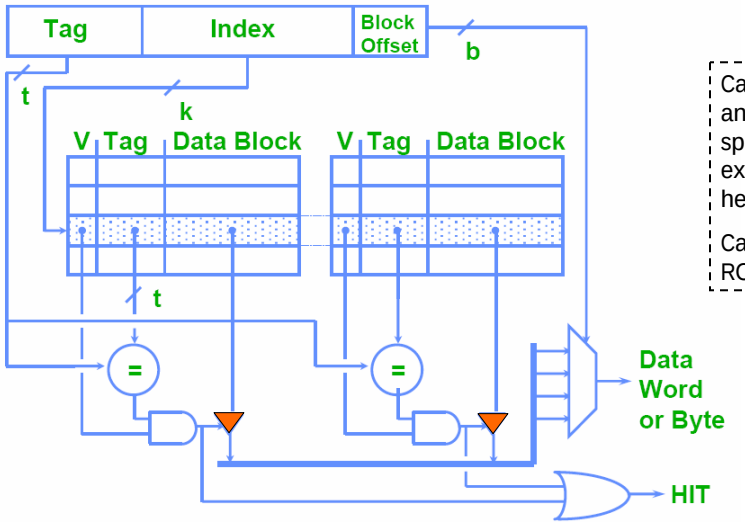
Can we do any form speculative execution here?

Can we use ROB here?

10

2-Way Cache: How Is A Word Found ...

cache set at index$_A$ is accessed then
[(tag$_A$ is compared with **tags** in the set, valid **bits** are checked) and **then**
(word is transferred to CPU)]



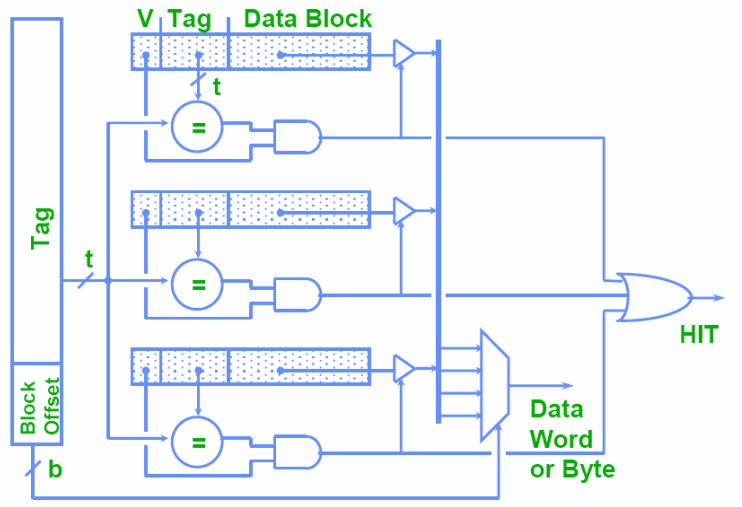
Can we do any form speculative execution here?

Can we use ROB here?

11

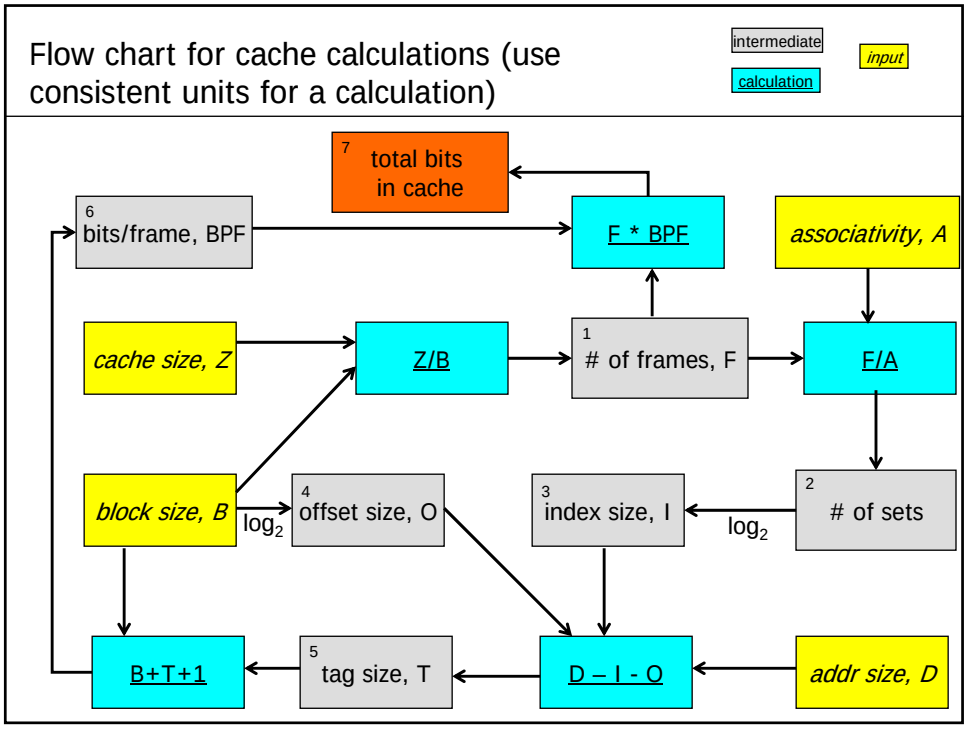
Fully Assoc. Caches: How Is A Word Found ...

(tag$_A$ is compared with **all** tags in the **cache**, valid bits are checked) and then
(word is transferred to CPU)



12

Flow chart for cache calculations (use consistent units for a calculation)



Measures of Cache Performance

- Miss rate
  - fraction of memmory accesses that do not hit in cache
- Average memory-access time (AMAT)
  - Hit time + Miss rate x Miss penalty (ns or clocks)
- CPU time
  - total time needed to execute the program

"miss penalty"

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{exec}} + \frac{\text{memory accesses}}{\text{instruction}} \times \text{miss rate} \times \text{miss penalty} \right) \times \text{cycle time}$$

$$= \text{IC} \times \left[\left(\text{CPI}_{\text{exec}} \times \text{cycle time} \right) + \left(\frac{\text{memory accesses}}{\text{instruction}} \times \text{miss rate} \times \text{miss penalty} \times \text{cycle time} \right) \right]$$

14

Effect of Associativity on AMAT

AMAT (in cycles) for a D-cache system on a DECstation 5000

Cache Size	Associativity			
(KB)	1-way	2-way	4-way	8-way
1	7.65	6.60	6.22	5.44
2	5.90	4.90	4.62	4.09
4	4.60	3.95	3.57	3.19
8	3.30	3.00	2.87	2.59
16	2.45	2.20	2.12	2.04
32	2.00	1.80	1.77	1.79
64	1.70	1.60	1.57	1.59
128	1.50	1.45	1.42	1.44

Numbers in red indicate that higher associativity resulted in increased AMAT.

AMAT = Hit time + Miss rate x Miss penalty

This is due to the fact that higher associativity results in increased hit time
→ speed of CPU is tied directly to the speed of a cache hit
→ increased clock cycle

what about IBM PowerPC and IBM 3033?

15

Split Versus Unified Cache

- on a unified cache, a load or store hit takes extra cycles for data access
  - because there is only one cache port for instructions and data
- a unified cache has a lower miss rate than split caches
- AMAT = %instr x (instr hit time + instr miss rate x instr miss penalty) +
  %data x (data hit time + data miss rate x data miss penalty)

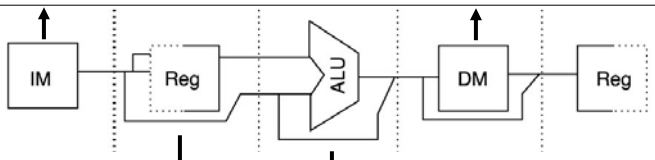
these two are equal for a split cache

these two are equal for a unified cache

16

Who Can Raise an Interrupt?

memory raises an interrupt if:
1) the requested block is not in memory, called a “page fault”
2) memory access is misaligned
3) access is to a protected area of memory



decoding logic raises an interrupt if instruction is unimplemented, i.e., if instruction opcode (or funct) are not recognized

breakpoint, requested by user for debugging

ALU unit raises an interrupt if the result overflows

17

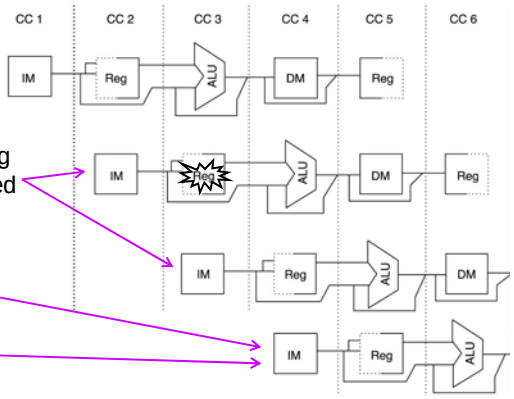
What to Do When An Interrupt is Raised?

4. let all instructions before offending instruction complete

3. turn off all writes for the offending instruction and the instructions issued after it

1. stop fetching further instructions from the program

2. force a “trap” instruction into the pipeline on the next IF

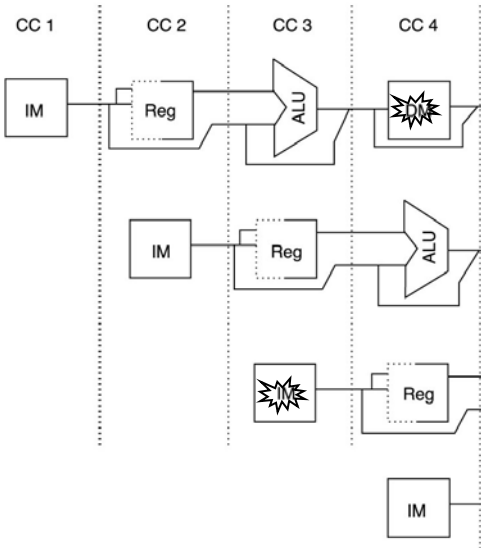


- precise interrupt
  - an interrupt that can be handled in such a manner that the processor state after the interrupt service is exactly as it was before the offending instruction

18

Pipelining Makes Interrupts Servicing Harder

a later-in-program order instruction can raise an interrupt sooner than an earlier-in-program order instruction



19

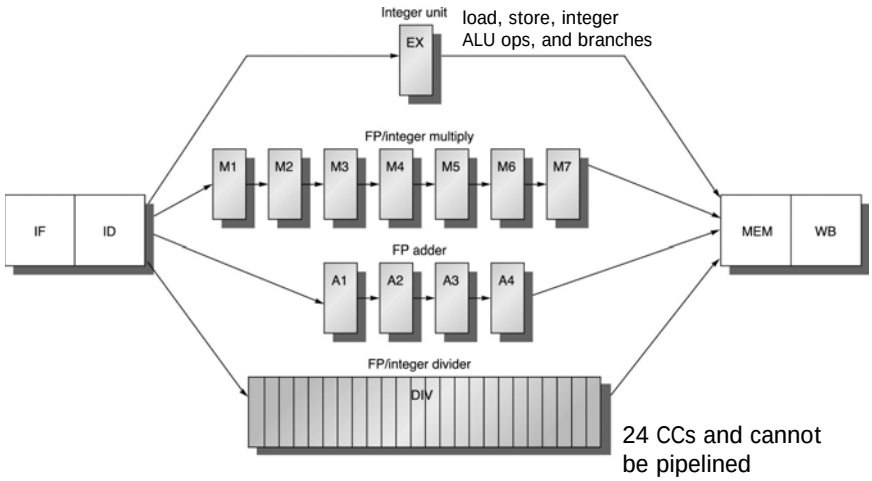
Ensuring Precise Interrupt

- interrupts must be serviced in the un-pipelined order
  - *all exceptions on inst x MUST be serviced before any exception on instr (x+1) is serviced*
- implementations of *"earliest instruction interrupt first"*
  - MIPS integer pipeline: post interrupts anytime, service in WB
  - MIPS floating point pipeline: post interrupts anytime, service in commit stage using a re-order buffer

20

MIPS Floating Point Pipeline

- key fact: the ***execution stage*** for a FP operation is longer than that for an integer operation



24 CCs and cannot be pipelined

21

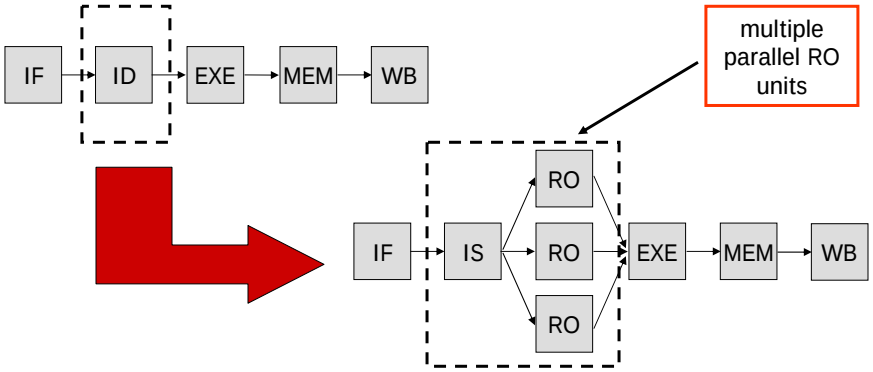
Why Should HW Be Asked to Re-Order?

- 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

22

Pipeline Modification: Split the ID Stage

- the “read operands” stage buffers a “stuck” instruction until its source operands are available



23

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

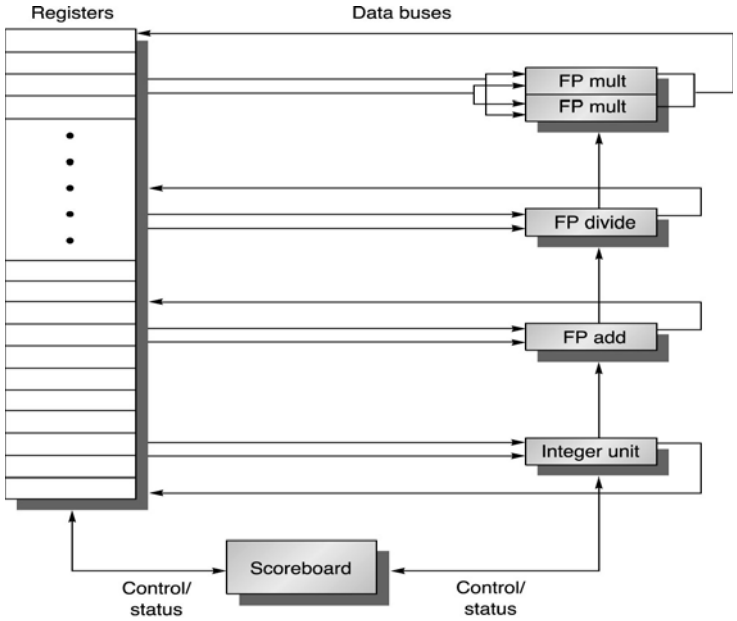
24

Dynamically Scheduled Pipelines

- **Scheduling Done Using Scoreboard**
  - avoids RAW, WAW, WAR and structural hazards albeit inefficiently
- **Scheduling Done Using Tomasulo's Algorithm**
  - efficiently avoids RAW, WAW, WAR and structural hazards

25

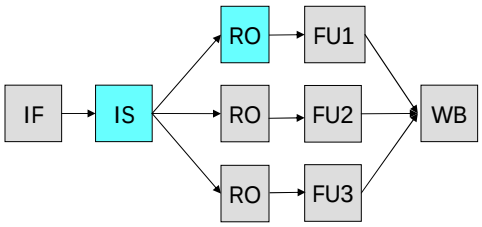
Scoreboard



26

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
  - 1st AND condition ensures structural hazards are avoided
  - 2nd AND condition ensures that WAW hazards are avoided

27

WAR Problems With Scoreboard

- Scoreboard requires that ***all*** operands be read at the same time
  - even those that are available sooner!!

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

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

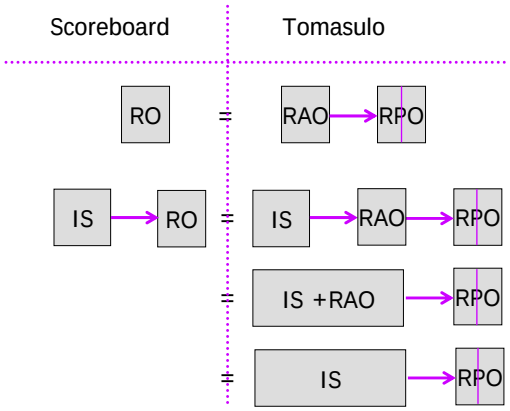
mult will be fetched, issued, its ops read, executed but its result will not be written back until f8 has been read by **add**

if f8 is read as soon as it is available, mult could write its result as soon as it is generated

28

Fixing Scoreboard's WAR Problems

- Scoreboard requires that ***all*** operands be read at the same time
- Tomasulo reads all available operands and then waits for pending operands
- RO stage should be split into "read available operand" and "read pending operand"



This makes scoreboard's Issue stage different from Tom's Issue stage.

29

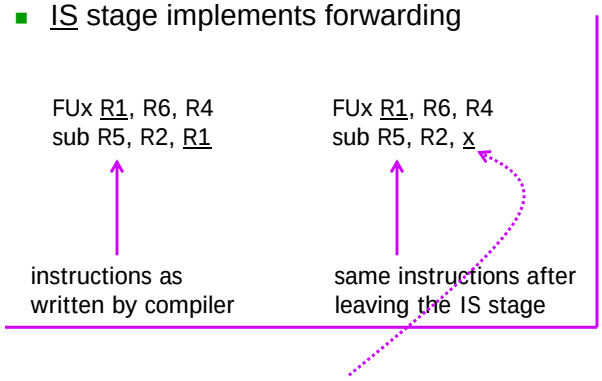
Scoreboard's Forwarding Problems

- Scoreboard requires that every source operand must be read from the register file
- Tomasulo re-instituted forwarding

30

How Is Forwarding Implemented in Tom's Alg?

- IS stage implements forwarding



In the IS stage, an instruction:

1) reads its available operands

2) for any unavailable operand, its register name is replaced with the tag of the FU that will forward the value

tag of the unavailable operand is simply the functional unit number of the producer instruction

register renaming

31

Reservation Stations

- name for the buffer associated with the RPO stage
- any waiting required in RPO stage happens in a "reservation station"
- a reservation station does the following:
  - buffers an instruction
  - buffers the available operand(s)
  - for any pending operands, RS buffers *tags* of the pending operands

32

WAW Problems With Scoreboard

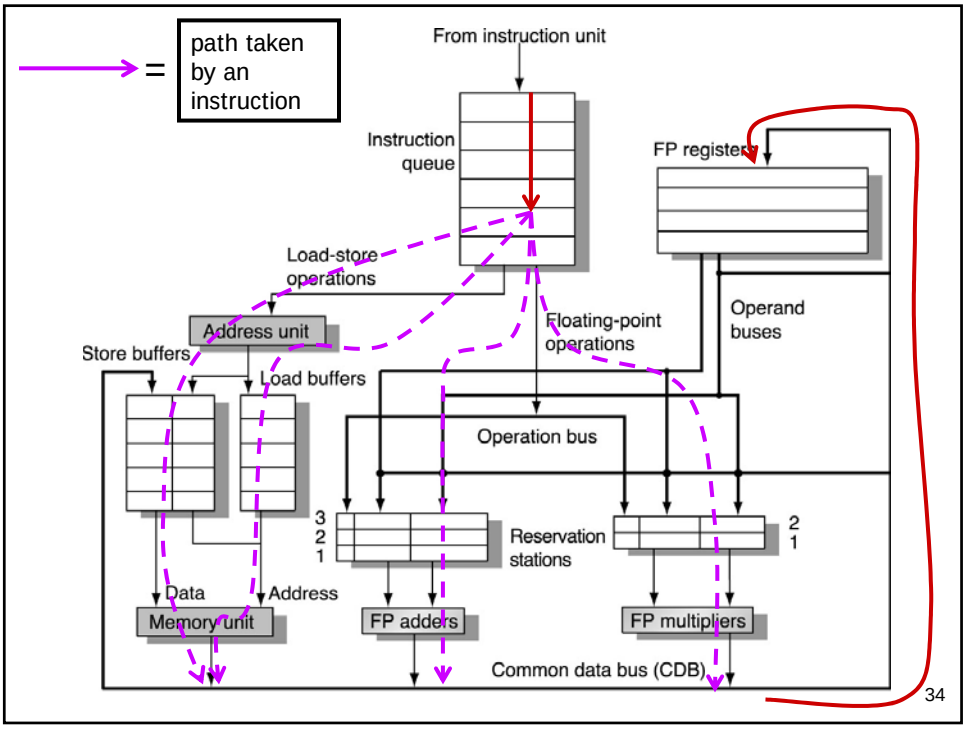
- Scorboard says an instruction X cannot be sent from Issue to Read Operand stage if there is an active instruction that has yet to write to destination register of X

```
div f0, f2, f4
add f0, f0, f8
sub f6, f10, f0
```

- assume div is executing in the divide unit
- will add be fetched, be issued?

- this WAW inefficiency is fixed in Tomasulo's algorithm by register renaming

33



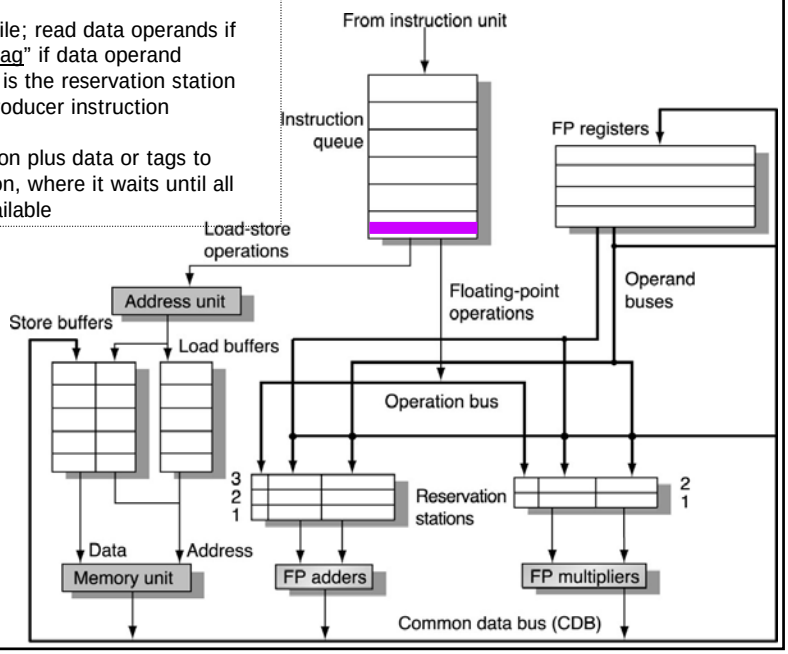
34

Stage 1: Issue

1) check for structural hazard; stall in issue stage if no free reservation station

2) read register file; read data operands if available; read "tag" if data operand unavailable; tag is the reservation station number of the producer instruction

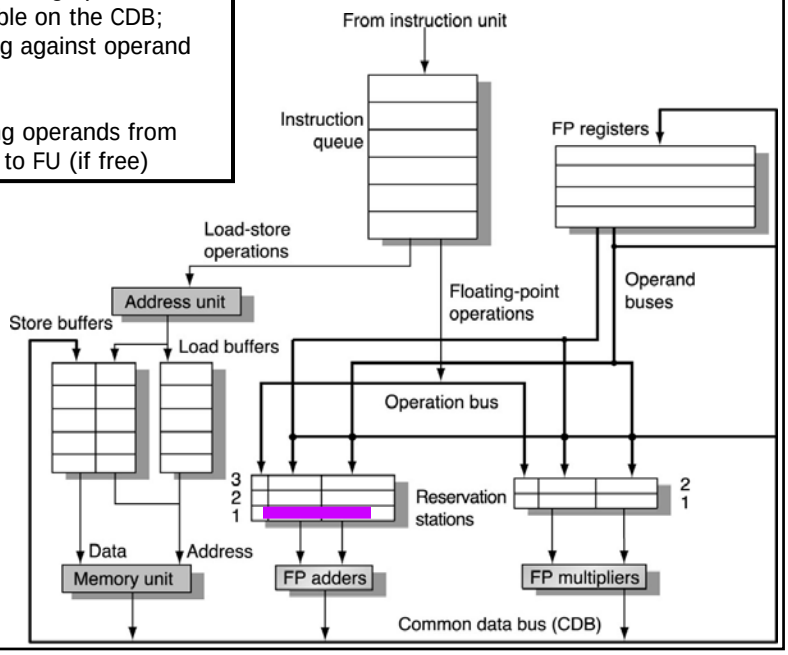
3) route instruction plus data or tags to reservation station, where it waits until all operands are available



Stage 2: Read Pending Operands

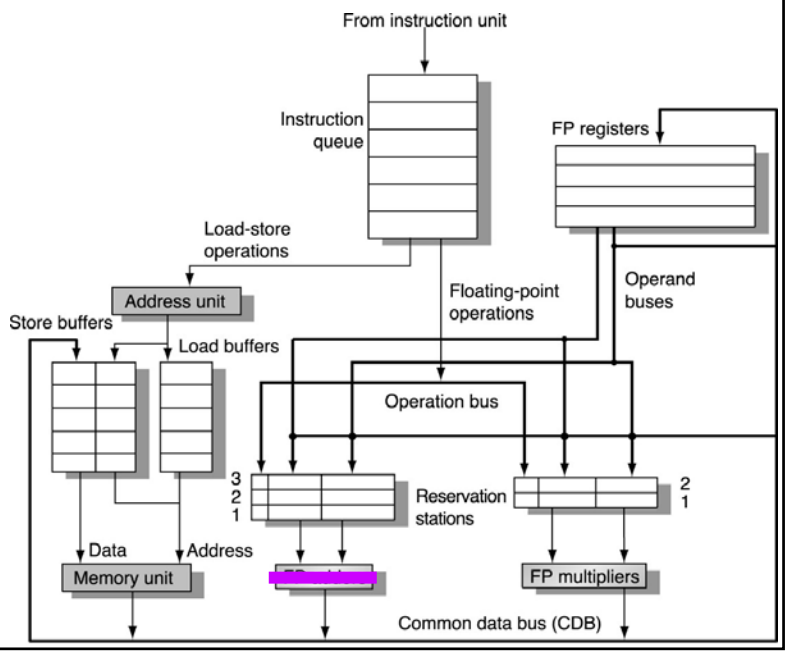
RPO (Read Pending Operands)
1) wait until pending operands become available on the CDB; (match CDB tag against operand tags)

2) grab pending operands from CDB and issue to FU (if free)



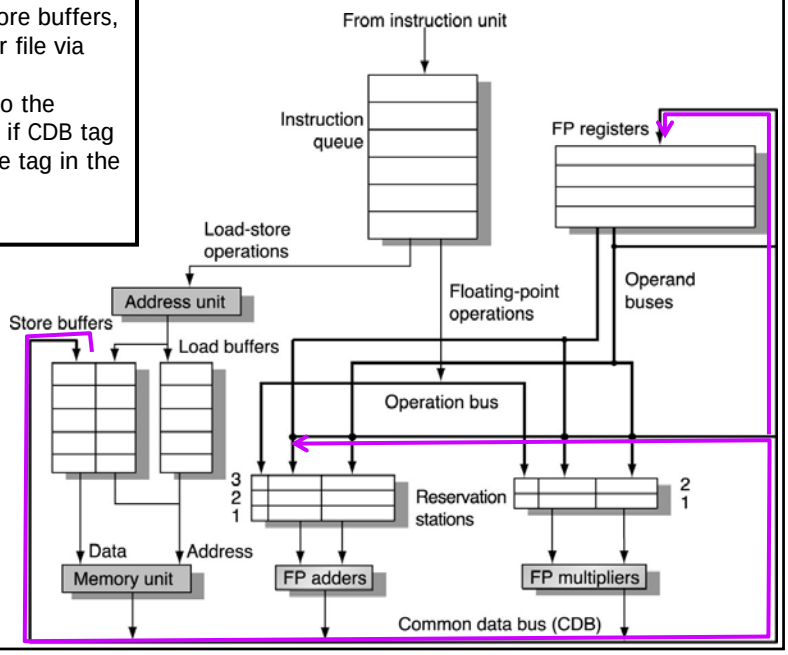
Just execute...

Stage 3: Execute



1) broadcast result + tag to all reservation stations, store buffers, and register file via the CDB
2) only write to the register file if CDB tag matches the tag in the register file

Stage 4: Write Result Back



finishing times

Instruction	j	k
LD F6	34	R2
LD F2	45	R3
MULTD F0	F2	F4
SUBD F8	F6	F2
DIVD F10	F0	F6
ADDD F6	F8	F2

IS	RPO	EXE	WR

RS	RS Tag	Busy	Op	Vj	Vk	Qj	Qk
Add1	1	No					
Add2	2	No					
Load1	3	No					
Load2	4	No					
Add3	5	No					
Mult1	6	No					
Mult2	7	No					

Integer Reg #	Q	V
1	0	213
2	0	50
3	0	100
4	0	417
5	0	87

FP Reg #	Q	V
0	0	18
2	0	32
4	0	56
6	0	31
8	0	98
10	0	66

39

finishing times

Instruction	j	k
LD F6	34	R2
LD F2	45	R3
MULTD F0	F2	F4
SUBD F8	F6	F2
DIVD F10	F0	F6
ADDD F6	F8	F2

IS	RPO	EXE	WR
1	1		

Tag or Q is allocated to an instruction **and** reg file by the issue logic.

CC 1

RS	RS Tag	Busy	Op	Vj	Vk	Qj	Qk
Add1	1	No					
Add2	2	No					
Load1	3	Yes	load	34	50	0	0
Load2	4	No					
Add3	5	No					
Mult1	6	No					
Mult2	7	No					

Integer Reg #	Q	V
1	0	213
2	0	50
3	0	100
4	0	417
5	0	87

FP Reg #	Q	V
0	0	18
2	0	32
4	0	56
6	3	
8	0	98
10	0	66

40

finishing times

Instruction	j	k
LD F6	34	R2
LD F2	45	R3
MULTD F0	F2	F4
SUBD F8	F6	F2
DIVD F10	F0	F6
ADDD F6	F8	F2

IS	RPO	EXE	WR
1	1		
2	2		

CC 2

RS	RS Tag	Busy	Op	Vj	Vk	Qj	Qk
Add1	1	No					
Add2	2	No					
Load1	3	Yes	load	34	50	0	0
Load2	4	Yes	load	45	100	0	0
Add3	5	No					
Mult1	6	No					
Mult2	7	No					

Integer Reg #	Q	V
1	0	213
2	0	50
3	0	100
4	0	417
5	0	87

FP Reg #	Q	V
0	0	18
2	4	
4	0	56
6	3	
8	0	98
10	0	66

41

finishing times

Instruction	j	k
LD F6	34	R2
LD F2	45	R3
MULTD F0	F2	F4
SUBD F8	F6	F2
DIVD F10	F0	F6
ADDD F6	F8	F2

IS	RPO	EXE	WR
1	1	3	
2	2		
3			

CC 3

RS	RS Tag	Busy	Op	Vj	Vk	Qj	Qk
Add1	1	No					
Add2	2	No					
Load1	3	Yes	load	34	50	0	0
Load2	4	Yes	load	45	100	0	0
Add3	5	No					
Mult1	6	Yes	mult		56	4	0
Mult2	7	No					

Integer Reg #	Q	V
1	0	213
2	0	50
3	0	100
4	0	417
5	0	87

FP Reg #	Q	V
0	6	
2	4	
4	0	56
6	3	
8	0	98
10	0	66

42

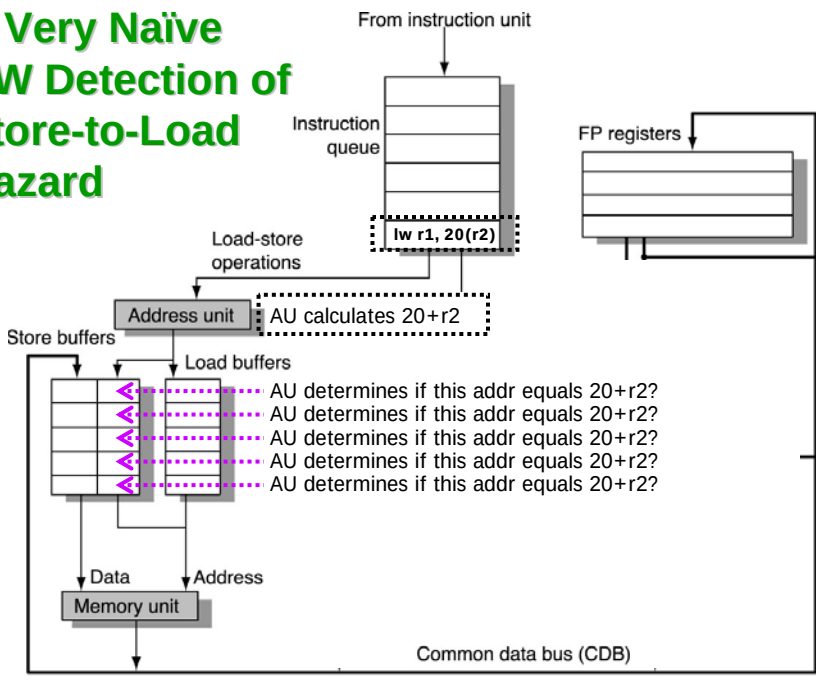
Data Hazards That A Compiler Cannot See

```
SW 16(R2), R7                    LW  R8, 20(R4)
...                              ...
...        can we do this?       ...
...                              ...
LW  R8, 20(R4)                   SW 16(R2), R7
```

- okay to re-order if (16+R2) is not equal to (20+R4)
- otherwise this relative ordering must never be changed for correct program execution
  - a potential RAW hazard exists
- how does Tomasulo's algorithm ensure that "relative ordering" of a store-load sequence is not changed?

43

A Very Naïve HW Detection of Store-to-Load Hazard



44

A Very Naïve HW Detection of Store-to-Load Hazard

- before sending a load instruction to its reservation station (called a load buffer), do this:
  1. calculate the memory address of the load
     (what if load address operand is pending?)
  2. compare it with the memory address of all store instructions currently in the store buffers (reservation stations for store instructions)
     (what if store address operand is pending?)
     (what if data to be stored is pending?)
  3. if there is an address match, do not send the load instruction to load buffer
- above three steps performed by the address unit
- essentially, a load instruction waits in the instruction queue until all stores issued before have completed

45

More Aggressive Solution

- SW-LW Dependency Speculation
- as soon as the load's effective address is calculated, the load is sent to memory
  - without waiting for earlier issued stores to complete
- any instructions waiting on load's result are allowed to execute
- when load comes up at the ROB head, its "speculative bit" is examined
  - if set, entire store buffer is checked to see if any addresses there match the load's address
    - if yes, ROB is flushed
    - if no, load is committed
    - if some store addresses still unknown, make load wait in ROB

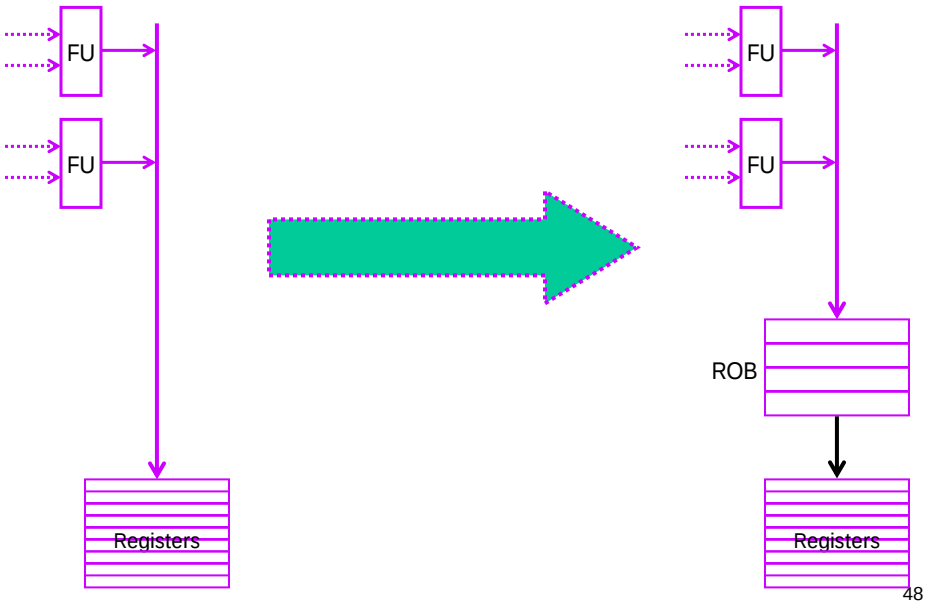
46

Fixing Tom's Alg For Precise Interrupts

- Tomasulo's algorithm must be modified if interrupts are to be precise
- solution: allow instructions to execute out of order, **but** force them to commit in order
- one implementation of above solution = re-order buffer, ROB
- re-order buffer holds the result of an instruction between the time the operation associated with an instruction completes and the time instruction commitss

47

Structure of ROB: The Middle Man



48

Modifying Tom's Alg For Precise Interrupts

- the modified algorithm has an additional stage, called commit
- stages are:
  - issue
  - read pending operand
  - execute
  - write result
  - commit, i.e., instruction's result in written into its destination (register or memory)
- or, we can merge RPO into execute (the way book does)
  - issue
  - execute
  - write result
  - commit

49

Changes in HW to Incorporate ROB

- each reservation station will have one new field "Dest"

RS	RS Tag	Busy	Op	Vj	Vk	Qj	Qk
Add1	1	Yes	sub	84	42	0	0

RS	Busy	Op	Vj	Vk	Qj	Qk	Dest
Add1	Yes	sub	84	42	0	0	#3

- Dest specifies the rob entry number of the instruction being housed in the RS
- symbols Qj, Qk will refer to rob entry number and NOT reservation station numbers

50

ROB Fields

head tail

entry #	busy	instruction	ready	destin-ation	value	exce-ption	PC
1	yes	MULTD	no	F0		no	1032
2	yes	SUBD	yes	F8	42	no	1036
3	yes	DIVD	no	F10		no	1040
4	no						

busy	*ready*	*meaning*
no	yes/no	vacant entry
yes	no	instruction issued but not yet completed the WR stage
yes	yes	instruction issued and completed the WR stage but not yet committed

"completed" instruction

51

Commit Stage of Modified Tomasulo's Alg - 1

- two different sequences
- examine the instruction at the head of ROB
- if "ready" field is set to "yes" (i.e., if result been written to ROB)
  - if "exception" field is set to "no" (i.e., instr did not raise exception)
    - copy the "value" field into:
      - the register (for loads and ALU instructions) if this instruction is supposed to write the register
      - memory (for stores)
    - remove the instr from ROB by setting busy to "no"

entry #	busy	instruction	ready	destin-ation	value	exce-ption	PC
1	yes	MULTD	yes	F0	30	no	1032
2	yes	SUBD	yes	F8	42	no	1036
3	yes	DIVD	no	F10		no	1040
4	no						

52

Commit Stage of Modified Tomasulo's Alg - 2

- two different sequences
- examine the instruction at the head of ROB
- if "ready" field is set to "yes" (i.e., if result been written to ROB)
  - ***if "exception" field is set to "yes"***
    - copy PC of offending instr somewhere
    - flush entire ROB
    - service the interrupt
    - restart execution by loading PC of the offending instruction

entry #	*busy*	*instruction*	*ready*	*destin-ation*	*value*	*exce-ption*	*PC*
1	yes	MULTD	yes	F0	30	yes	1032
2	yes	SUBD	yes	F8	42	no	1036
3	yes	DIVD	no	F10		no	1040
4	no						

53

ROB Allows Speculative Execution

- result from a completed but not committed instruction X is used by later consumer instructions speculatively
- if X raises an exception, the entire ROB is flushed
  - i.e., results of all later instructions (possibly speculatively executed) are discarded
- the process of using an instruction's result before knowing if it is valid is called "speculative execution"

54

Other Forms of Speculation - 1

I realize we have not done branch prediction schemes yet. Will cover soon.

- branch outcome speculation:
  - assume the HW predicted that a particular branch will be taken
  - based on the prediction, HW executed a few instructions after the branch
    - that is, some speculative execution was done
  - then HW finds out that the branch was not taken!!!
  - *open question to class: How can we use re-order buffer to ensure that incorrect code execution will not be done in the above case?*
    - you do not need to know the methods for predicting branches to answer this question

55

Commit Stage For Predicted Branches

- add a "branch prediction" field to the ROB entry
- set to "nok" when it is known that branch was incorrectly predicted
- set to "ok" when it is known that branch was correctly predicted

- assume that a branch instruction is at the head of ROB
- examine the ***branch*** instruction
  - if the "branch prediction" is set to "ok"
    - commit branch, i.e., "accept" the results of the instructions speculatively executed on the predicted path
  - if the "branch prediction" is set to "nok"
    - flush entire ROB
    - restart execution by loading _____

56