UML 16.671
Advanced Computer Architecture

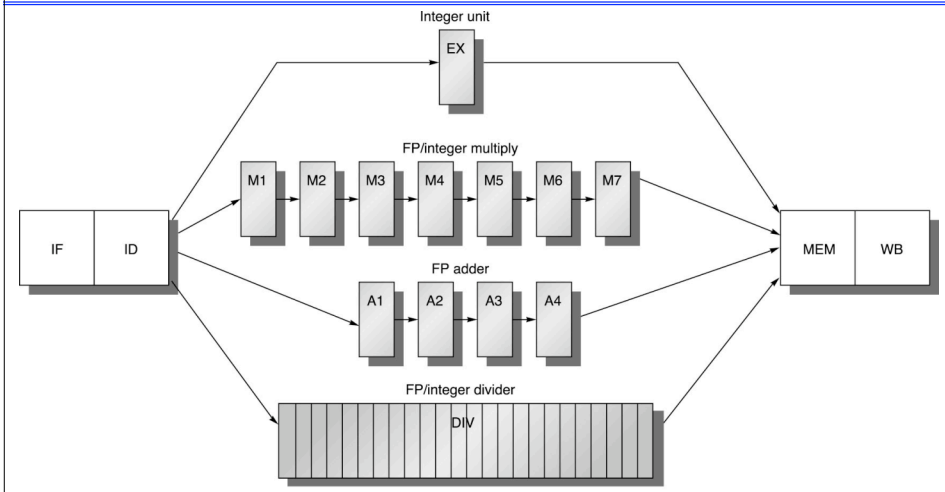
Chapter 3 and Appedix A

Pipeline and Hazard

Instructor: Prof. Yan Luo

Some slides are adapted from Roth

A pipeline with multi-cycle FP operations



Pipeline Hazards

- Hazards are caused by conflicts between instructions. Will lead to incorrect behavior if not fixed.
  - Three types:
    - Structural: two instructions use same h/w in the same cycle – resource conflicts (e.g. one memory port, unpipelined divider etc).
    - Data: two instructions use same data storage (register/memory) – dependent instructions.
    - Control: one instruction affects which instruction is next – PC modifying instruction, changes control flow of program.

3

Handling Hazards

- Force stalls or bubbles in the pipeline.
  - Stop some younger instructions in the stage when hazard happen
  - Make younger instr. Wait for older ones to complete
- Flush pipeline
  - Blow instructions out of the pipeline
  - Refetch new instructions later – solving control hazards
  - Implementation: assert clear signals on pipeline registers

4

Structural Hazards

- Example
  - Assume unified cache memory, i.e., instruction and data are stored in a single cache, and each cycle only one request can be processed (either instruction or data) – this cache has only one port

	1	2	3	4	5	6	7	8	9
Load	f	d	x	m	w				
inst1		f	d	x	m	w			
inst2			f	d	x	m	w		
inst3				f	d	x	m	w	

5

Fixing Structural Hazards Using Stalls

- Stall Pipeline

	1	2	3	4	5	6	7	8	9	10
Load	f	d	x	m	w					
inst1		f	d	x	m	w				
inst2			f	d	x	m	w			
inst3				-	f	d	x	m	w	
inst4				-	-	f	d	x	m	w

- Duplicate Resource – Separate IM and DM

6

Data Hazards

- Two different instructions use the same storage location
  - It must appear as if they executed in sequential order

read-after-write (RAW)	write-after-read (WAR)	write-after-write (WAW)
True dependence (real)	anti dependence (artificial)	output dependence (artificial)

What about read-after-read dependence ?

7

Reducing RAW Hazards: Bypassing

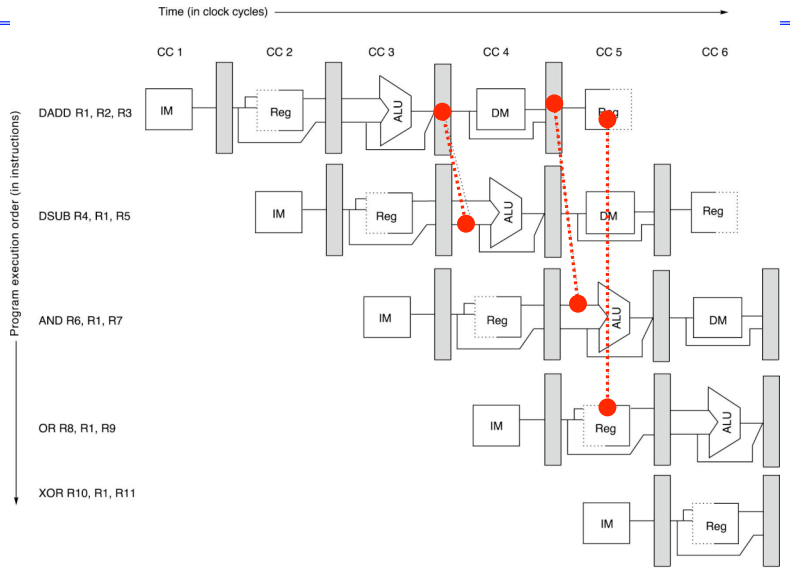
- Data available at the end of EX stage, why wait until WB stage?
  - Bypass (forward) data directly to input of EX
  - \+ Reduces/avoids stalls in a big way
    - Large fraction of input operands are bypassed
  - – Complex
  - Important: does not relieve you from having to perform WB

	1	2	3	4	5	6	7	8	9
add R1, R2, R3	f	d	x	m	w				
sub R2, R4, R1		f	d	x	m	w			

- Can bypass from MEM also

8

Minimizing Data Hazard Stalls by Forwarding



But …

- Even with bypassing, not all RAWs stalls can be avoided
  - Load to an ALU immediately after
  - Can be eliminated with compiler scheduling

	1	2	3	4	5	6	7	8	9
lw R1, 16(R3)	f	d	x	m	w				
sub R2, R4, R1		f	-	d	x	m	w		

You can also stall before EX stage, but it is better to separate stall logic from bypassing logic

Compiler Scheduling

- Compiler moves instructions around to reduce stalls
  - E.g. code sequence: a = b+c, d = e–f

before scheduling	after scheduling
lw Rb, b	lw Rb, b
lw Rc, c	lw Rc, c
add Ra, Rb, Rc //stall	lw Re, e
sw Ra, a	add Ra, Rb, Rc//no stall
lw Re, e	lw Rf, f
lw Rf, f	sw Ra, a
sub Rd, Re, Rf //stall	sub Rd, Re, Rf//no stall
sw Rd, d	sw Rd, d

11

WAR: Why do they exist? (Antidependence)

- Recall WAR

```
add R1, R2, R3
sub R2, R4, R1
or  R1, R6, R3
```

- Problem: swap means introducing false RAW hazards
- Artificial: can be removed if sub used a different destination register
- Can't happen in in-order pipeline since reads happen in ID but writes happen in WB
- Can happen in out-of-order reads, e.g. out-of-order execution

12

WAW (Output Depndence)

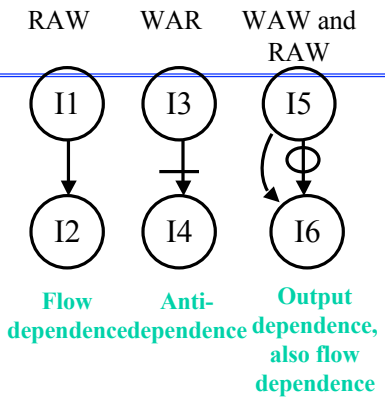
```
add   R1, R2, R3
sub   R2, R4, R1
or    R1, R6, R3
```

- Problem: scheduling would leave wrong value in R1 for the sub
- Artificial: using different destination register would solve
- Can't happen in in-order pipeline in which every instruction takes same cycles since writes are in-order
- Can happen in the presence of multi-cycle operations, i.e., out-of-order writes

EXAMPLE

Program order

I1. Load R1, A /R1 ← Memory(A)/
I2. Add R2, R1 /R2 ← (R2)+(R1)/
I3. Add R3, R4 /R3 ← (R3)+(R4)/
I4. Mul R4, R5 /R4 ← (R4)*(R5)/
I5. Comp R6 /R6 ← Not(R6)/
I6. Mul R6, R7 /R6 ← (R6)*(R7)/



Due to Superscalar Processing, it is possible that I4 completes before I3 starts. Similarly the value of R6 depends on the beginning and end of I5 and I6. Unpredictable result!

A sample program and its dependence graph, where I2 and I3 share the adder and I4 and I6 share the same multiplier. These two dependences can be removed by duplicating the resources, or pipelined adders and multipliers.

Register Renaming

Rewrite the previous program as:

- I1. R1b ← Memory (A)
- I2. R2b ← (R2a) + (R1b)
- I3. R3b ← (R3a) + (R4a)
- I4. R4b ← (R4a) * (R5a)
- I5. R6b ← -(R6a)
- I6. R6c ← (R6b) * (R7a)

Allocate more registers and rename the registers that really do not have flow dependency. The WAR hazard between I3 and I4, and WAW hazard between I5 and I6 have been removed.

These two hazards also called Name dependencies

15

Control Hazards

- **Branch problem:**
  - branches are resolved in EX stage

  → 2 cycles penalty on taken branches

  Ideal CPI =1. Assuming 2 cycles for all branches and 32% branch instructions → new CPI = 1 + 0.32*2 = 1.64

- **Solutions:**
  - Reduce branch penalty: change the datapath – new adder needed in ID stage.
  - Fill branch delay slot(s) with a useful instruction.
  - Fixed branch prediction.
  - Static branch prediction.
  - Dynamic branch prediction.

16

Control Hazards – branch delay slots

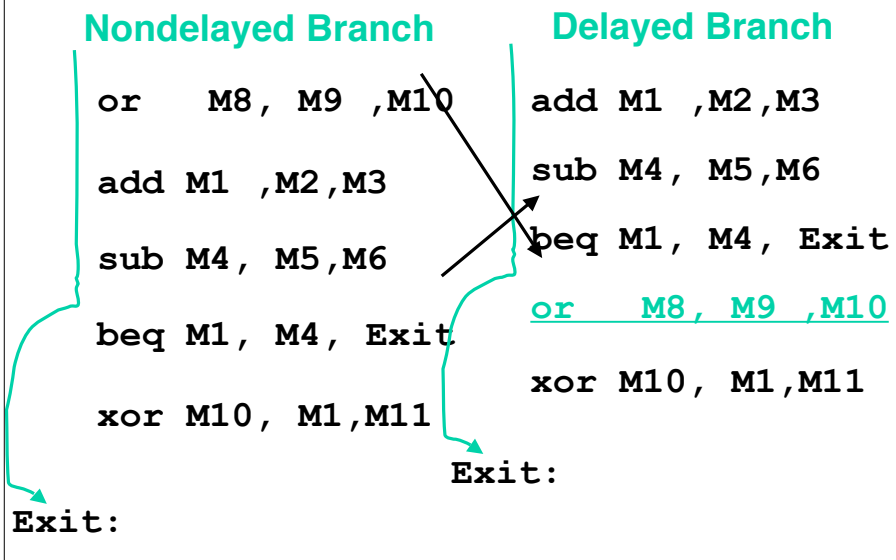
- **Reduced branch penalty:**
  - Compute condition and target address in the ID stage: 1 cycle stall.
  - Target and condition computed even when instruction is not a branch.
- **Branch delay slot filling:**

  move an instruction into the slot right after the branch, hoping that its execution is necessary. Three alternatives (next slide)

  Limitations: restrictions on which instructions can be rescheduled, compile time prediction of taken or untaken branches.

17

Example Nondelayed vs. Delayed Branch



18

Control Hazards: Branch Prediction

- **Idea: doing something is better than waiting around doing nothing**
  - o Guess branch target, start executing at guessed position
  - o Execute branch, verify (check) your guess
  - \+ minimize penalty if guess is right (to zero)
  - – May increase penalty for wrong guesses
  - o Heavily researched area in the last 15 years
- **Fixed branch prediction.**

  Each of these strategies must be applied to all branch instructions indiscriminately.
  - **Predict not-taken** (47% actually not taken):
    - continue to fetch instruction without stalling;
    - do not change any state (no register write);
    - if branch is taken turn the fetched instruction into no-op, restart fetch at target address: 1 cycle penalty.

19

Control Hazards: Branch Prediction

  - **Predict taken** (53%): more difficult, must know target before branch is decoded. no advantage in our simple 5-stage pipeline.
- **Static branch prediction.**
  - Opcode-based: prediction based on opcode itself and related condition. Examples: MC 88110, PowerPC 601/603.
  - Displacement based prediction: if d < 0 predict taken, if d >= 0 predict not taken. Examples: Alpha 21064 (as option), PowerPC 601/603 for regular conditional branches.
  - Compiler-directed prediction: compiler sets or clears a predict bit in the instruction itself. Examples: AT&T 9210 Hobbit, PowerPC 601/603 (predict bit reverses opcode or displacement predictions), HP PA 8000 (as option).

20

Control Hazards: Branch Prediction

- **Dynamic branch prediction**
  - **Later**

21

MIPS R4000 FP Pipe Stages

FP Instr	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	…
Add, Subtract	U	S+*A*	*A*+R	R+S					
Multiply	U	E+M	M	M	M	N	N+*A*	R	
Divide	U	*A*	R	D^{28}	…	D+*A*	D+R, D+R, D+*A*, D+R, *A*, R		
Square root	U	E	$(A+R)^{108}$	…	*A*	R			
Negate	U	S							
Absolute value	U	S							
FP compare	U	*A*	R						

Stages:

M	*First stage of multiplier*	***A***	***Mantissa ADD stage***
N	*Second stage of multiplier*	***D***	***Divide pipeline stage***
R	*Rounding stage*	***E***	***Exception test stage***
S	*Operand shift stage*		
U	*Unpack FP numbers*		

22

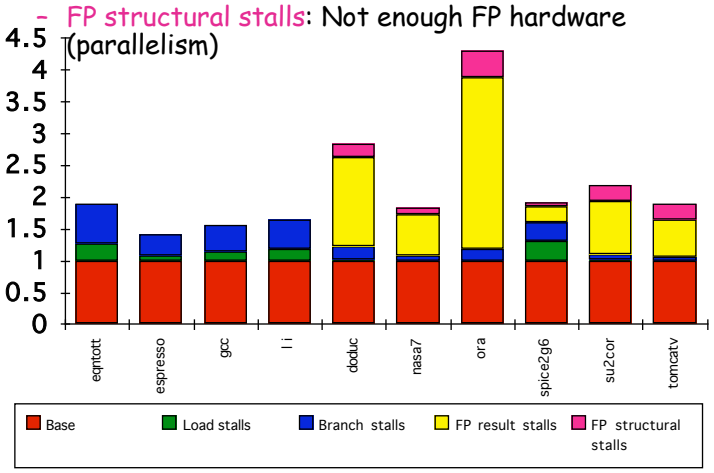
Example: Multiple and Add

Op	Issue/ Stall	0	1	2	3	4	5	6	7	8	9	10	11
Multiple	Issue	U	E+M	M	M	M	N	N+A	R				
Add	Issue		U	S+A	A+R	R+S							
	Issue			U	S+A	A+R	R+S						
	Issue				U	S+A	A+R	R+S					
	Stall					U	S+A	A+R	R+S				
	Stall						U	S+A	A+R	R+S			
	Issue							U	S+A	A+R	R+S		
	Issue								U	S+A	A+R	R+S	

23

R4000 Performance

- ~~Not ideal CPI of 1:~~
  - Load stalls (1 or 2 clock cycles)
  - Branch stalls (2 cycles + unfilled slots)
  - FP result stalls: RAW data hazard (latency)
  - FP structural stalls: Not enough FP hardware (parallelism)



24