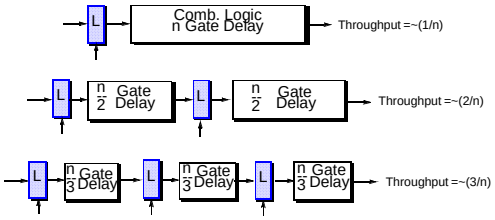
Pipelined Processors

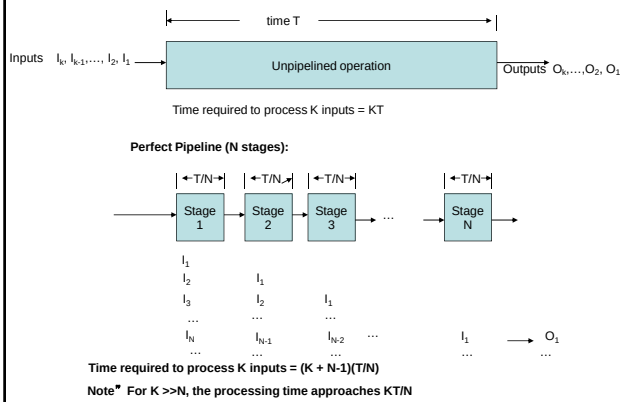
HPCA, Spring 2011

Ideal Pipelining



- Ideally, throughput increases linearly with pipeline depth

Ideal Pipeline Performance



Factors Inhibiting Ideal Pipeline Performance

- Unequal distribution of work among stages
  - Clock cycle time must accommodate slowest stage
- Staging logic introduces additional delays
- May not be able to keep the pipeline full
  - Stall behavior
  - Much more about this later

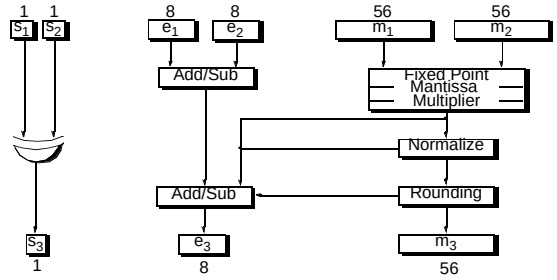
Example: FP Multiplier

- Exponent: excess 128 (8 bits)
- Mantissa: sign-magnitude fraction with hidden bit (57 bits total)

- Algorithm:

1. Check if any operand is ZERO.
2. ADD the two characteristics (physical bit patterns of the exponents) and correct for the excess 128 bias, i.e. e1+ (e2-128)
3. Perform fixed-point MULTIPLICATION of the mantissas.
4. NORMALIZE the product of the mantissas, i.e. may require one left shift and decrement the exponent.
5. ROUND the result by adding 1 to the first guard bit; if mantissa overflows, then shift right one bit and increment the exponent.

Nonpipelined Implementation



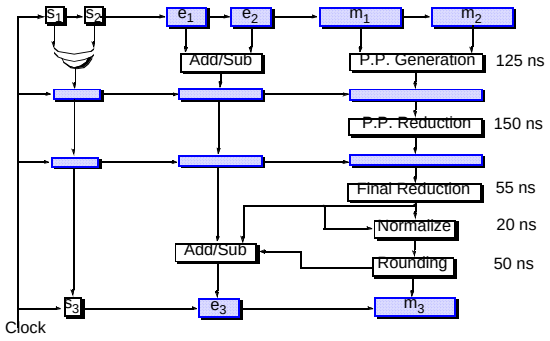
Nonpipelined Implementation

Total Chip counts and delays:

	Chip Count	Delay
P. P. Generation	34	125 ns
P. O. Reduction	72	150 ns
Final Reduction	21	55 ns
Normalization	2	20 ns
Rounding	15	50 ns
Exponent Section	4	--------
Input Registers	17	--------
Output Registers	10	--------
	175	400 ns

- Unpipelined clock period = 400 nsec. (2.5 MFLOPS) (based on very old IC technology)

Pipelined Implementation



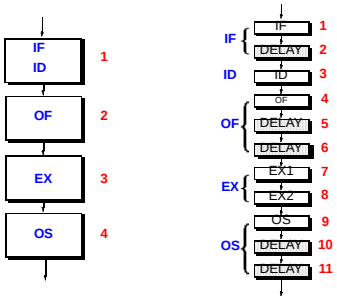
Pipelined Implementation

- Three Stage Pipelining:
  - Longest delay path within a stage (PP Reduction) = 150 nsec.
- Hence can have pipeline clock period of 150 nsec. plus 22 nsec. in pipeline overheads (totaling 172 nsec.)
- Number of ICs added: 82 edge-triggered registers; 175 + 82 = 257
  - Original total delay - 400 nsec (2.5 MFLOPS)
  - New min. clock period - 172 nsec (5.8 MFLOPS)
  - Original no. of ICs - 175 chips
  - New total of ICs - 257 chips
- Less than 50% increase in hardware more than doubles the throughput (from 2.5 to 5.8 MFLOPS)
- Note that an ideal 3-stage pipeline would have achieved a clock period of 400/3 = 133 nsec. and a maximum throughput of 7.5 MFLOPS

Processor Pipelining

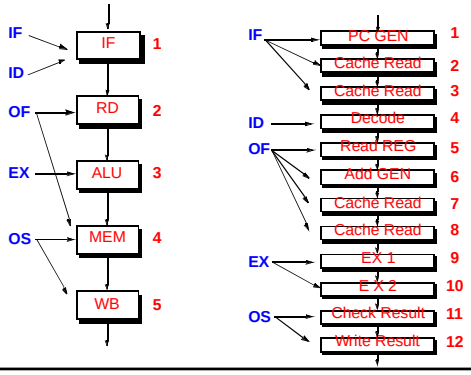
- **The "computation" to be pipelined.**
  - **Instruction Fetch (IF)**
  - **Instruction Decode (ID)**
  - **Operand(s) Fetch (OF)**
  - **Instruction Execution (EX)**
  - **Operand Store (OS)**
  - **Update Program Counter (PC)**

Granularity of Pipeline Stages



- **Logic needed for each pipeline stage.**
- **Register file ports needed to support all the stages**
- **Memory accessing ports needed to support all the stages**

Example Pipelines



Development of a simple RISC Pipeline

- Consider a simple MIPS-like ISA
  - Complete ISA Specification provided in Lecture notes section of class web site
  - Some example instructions

LW R_2, 10(R_1) // Reg[R_2] <- Mem[Reg[R_1]+10]
SW 10(R_1), R_2 // Mem[Reg[R_1]+10] -> R_2
ADD R_1,R_2,R_3 // Reg[R_1] <- Reg[R_2]+Reg[R_3]
BEQZ R_1, 16 // If (Reg[R_1]==0) PC <- PC + 16
JMP -24 // PC <- PC -24

ALU Instruction Specification (MIPS-like ISA)

Generic subcomputations	1. ALU Instruction Type:	
	Integer instruction	Floating-point instruction
IF	- Fetch instruction (access I-memory)	- Fetch instruction (access I-memory)
ID	- Decode instruction	- Decode instruction
OF	- Access register file	- Access FP register file
EX	- Perform ALU operation	- Perform FP operation
OS	- Write back to reg. file	- Write back to FP reg. file

Memory Instruction Specification

Generic subcomputations	2. Load/Store Instruction Type:	
	Load instruction	Store instruction
IF	**- Fetch instruction (access I-cache)**	**- Fetch instruction (access I-cache)**
ID	**- Decode instruction**	**- Decode instruction**
OF	**- Access register file (base address) - Generate effective address (base + offset) - Access (read) memory location (D-mem)**	**- Access register file (register operand, and base address)**
EX	-	-
OS	**- Write back to reg. file**	**- Generate effective address (base + offset) - Access (write) memory location (D-mem)**

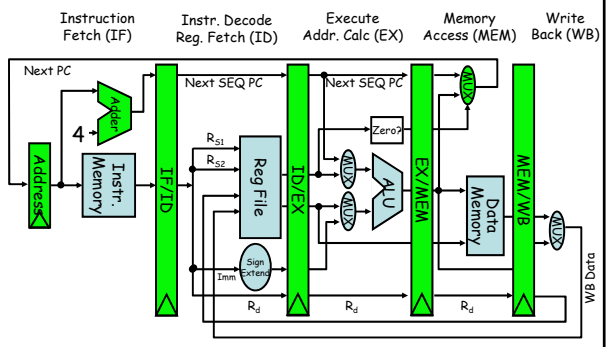
Branch Instruction Specification

Generic subcomputations	3. Branch Instruction Type:	
	Jump (uncond.) instruction	Conditional branch instr.
IF	**- Fetch instruction (access I-memory)**	**- Fetch instruction (access I-memory)**
ID	**- Decode instruction**	**- Decode instruction**
OF	**- Access register file (base address) - Generate effective address (base + offset)**	**- Access register file (base address) & test reg - Generate effective address (base + offset)**
EX	-	**- Evaluate branch condition**
OS	**- Update program counter with target address**	**- If condition is true, update program counter with target address**

The Unified Pipeline

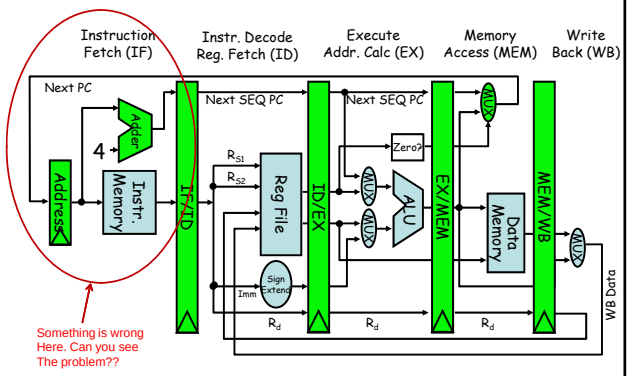
	ALU instr.	LOAD instr.	STORE instr.	BRANCH instr.
IF stage	**Read Instr. From I_Mem; PC++**	**Read Instr. From I_Mem; PC++**	**Read Instr. From I_Mem; PC++**	**Read Instr. From I_Mem; PC++**
ID/RD stage	Decode Instr. Read Regs (Src. operands)	Decode Instr. Read Reg (mem base addr.)	Decode Instr. Read Regs (mem base addr; store data)	Decode Instr. Read Reg (test reg)
ALU stage	ALU Operation	Compute Mem. Address	Compute Mem. Address	Compute Branch Target Address (PC + displ.) Test branch condition
MEM stage		Memory Read	Memory Write	PC Update
WB stage	Write Result to Dest. Reg	Write Data to Dst. Reg.		

5-stage MIPS Pipeline (Datapath)



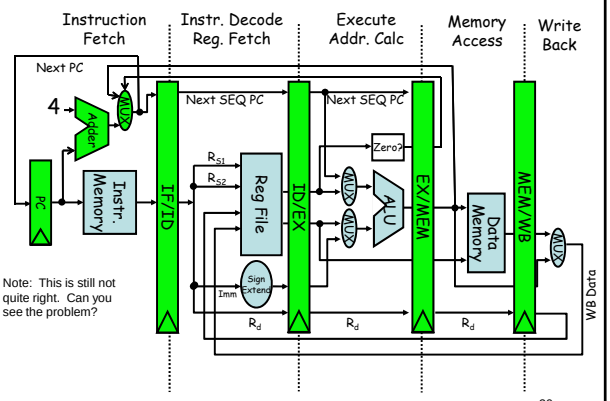
18

5-stage MIPS Pipeline (Datapath)



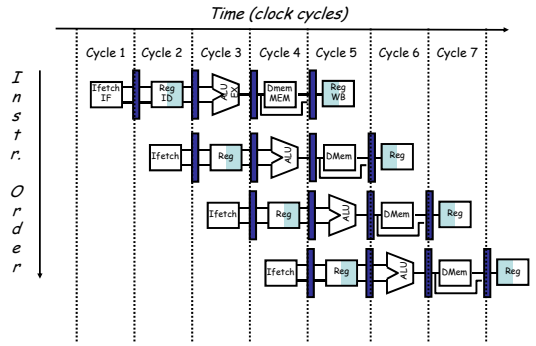
19

5 Steps of MIPS-like Datapath (corrected)



20

Visualizing Pipelining



21

An Even Simpler View

Clock cycle

	1	2	3	4	5	6	7	8
Instr i	IF	ID	EX	MEM	WB			
Instr i+1		IF	ID	EX	MEM	WB		
Instr i+2			IF	ID	EX	MEM	WB	
Instr i+3				IF	ID	EX	MEM	WB

Theoretical Speedup of 5-Stage MIPS Pipeline

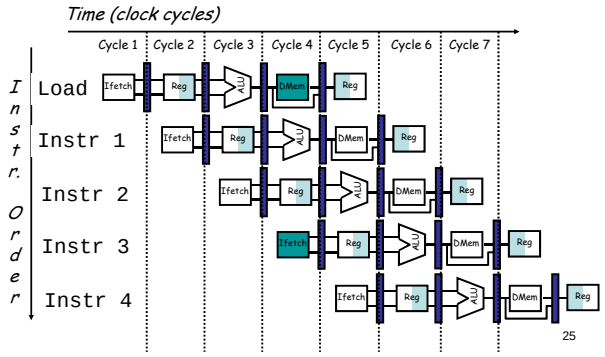
- Assume:
  - Cycle Time of non-pipelined implementation of MIPS datapath is t
  - Cycle time of pipelined data path (5 stages) is t/5
  - Pipeline always operates at full capacity
- Then:
  - Speedup of pipelined implementation versus non-pipelined version approaches FIVE.

But, Pipelining is not quite that easy!

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support this combination of instructions (single person to fold and put clothes away)
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

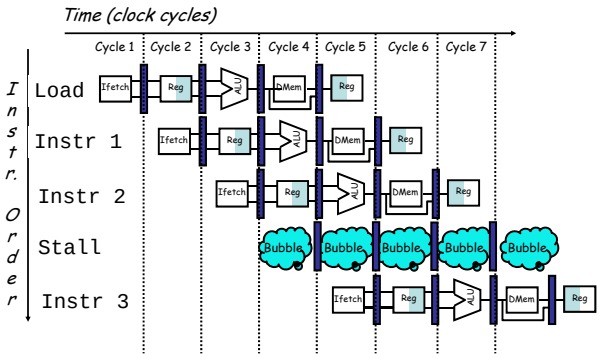
24

One Memory Port/Structural Hazard



25

One Memory Port/Structural Hazards



How do you "bubble" the pipe?

26

Processor Performance Equation for Pipelined Processor (accounting for Stalls)

Time/Program =
Instructions/Program x (Ideal CPI + Stalls/instruction) x CycleTime

For simple (scalar) RISC, Ideal CPI = 1, so:

Time/Program =
Instructions/Program x (1 + Stalls/instruction) x CycleTime

27

Example: Dual-port vs. Single-port

- Machine A: Dual ported memory ("Harvard Architecture")
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Load/stores are 40% of instructions executed

$T_A = N \times 1 \times 1 = N$

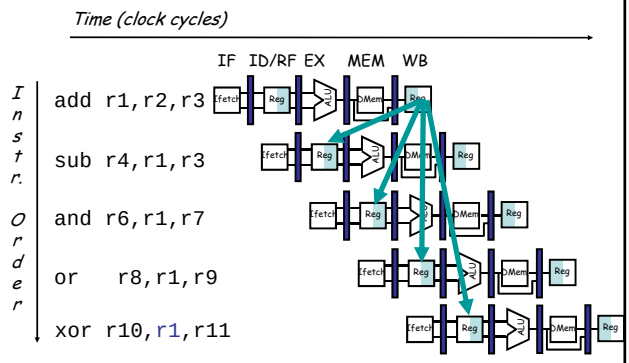
$T_B = N \times (1 + 0.4(1)) \times 1/1.05 = 1.33$

Speedup = TB/TA = 1.33/1 = 1.33

So Machine A is 1.33 times faster than Machine B

28

Data Hazard on R1



29

Three Generic Data Hazards

- Read After Write (RAW)
  $Instr_J$ tries to read operand before $Instr_I$ writes it

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

- Caused by a "Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.

30

Three Generic Data Hazards

- Write After Read (WAR)
  $Instr_J$ writes operand *before* $Instr_I$ reads it

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an "anti-dependence" by compiler writers. This results from reuse of the name "r1".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Register Reads are always in stage 2, and
  - Register Writes are always in stage 5

31

Three Generic Data Hazards

- Write After Write (WAW)
  $Instr_J$ writes operand *before* $Instr_I$ writes it.
-

```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an "output dependence" by compiler writers This also results from the reuse of name "r1".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Register Writes are always in stage 5
- Will see WAR and WAW in more complicated pipes

32

Data Hazard Example (quicksort/MIPS)

```
#       for (;  (j < high)  &&  (array[j] < array[low])  ;  ++j  );
#       $10  =  j
#       $9  =  high
#       $6  =  array
#       $8  =  low
        bge             done,  $10,    $9
        mul             $15,   $10,    4
        addu            $24,   $6,     $15
        lw              $25,   0($24)
        mul             $13,   $8,     4
        addu            $14,   $6,     $13
        lw              $15,   0($14)
        bge             done,  $25,    $15
cont:
        addu            $10,  $10,  1
        . . .
done:
        addu            $11,  $11,  -1
```

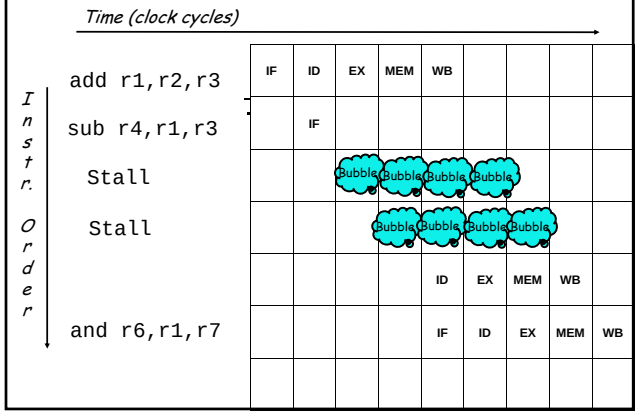

Resolution of Pipeline Hazards

- Pipeline hazards
  - Potential violations of program dependences
  - Must ensure program dependences are not violated
- Hazard resolution
  - Static: compiler/programmer guarantees correctness
  - Dynamic: hardware performs checks at runtime
- Pipeline interlock
  - Hardware mechanism for dynamic hazard resolution
  - Must detect and enforce dependences at runtime

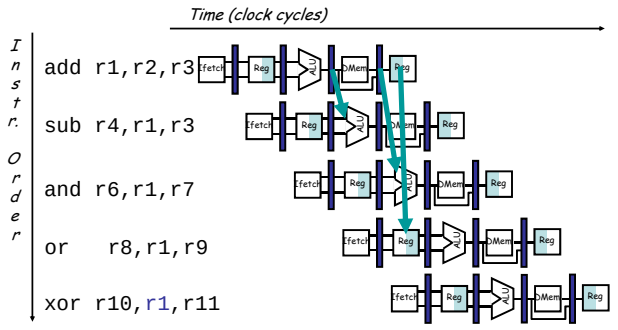
Data Hazard Mitigation

- A better response – forwarding
  - Also called bypassing
- Comparators ensure register is read after it is written
- Instead of stalling until write occurs
  - Use mux to select forwarded value rather than register value
  - Control mux with hazard detection logic

Data Hazard on R1

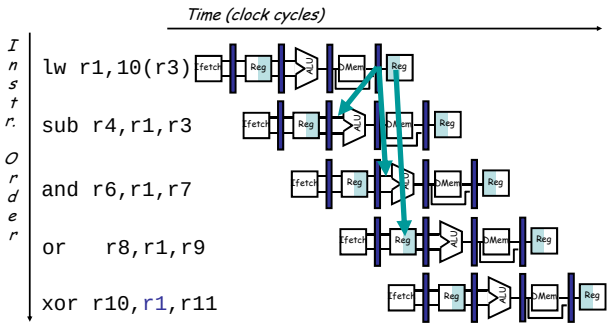


Forwarding to Avoid Data Hazards



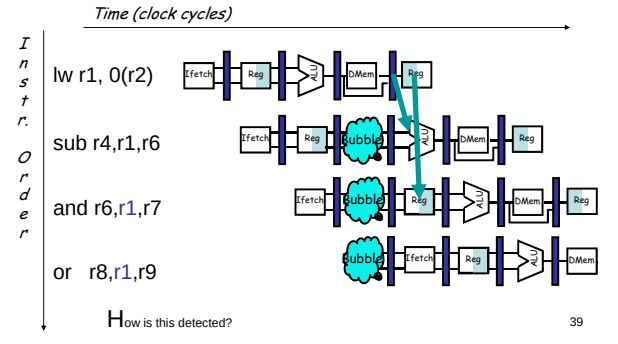
37

RAW Data Hazards Involving Loads



38

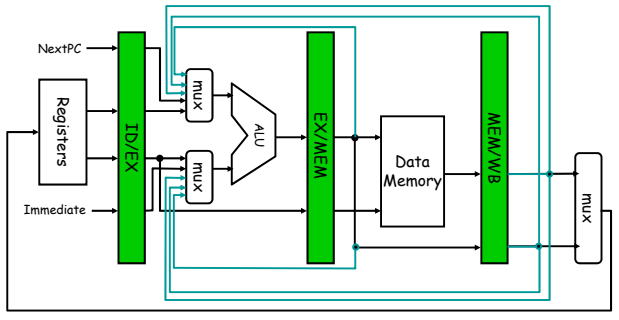
Data Hazard Even with Forwarding



How is this detected?

39

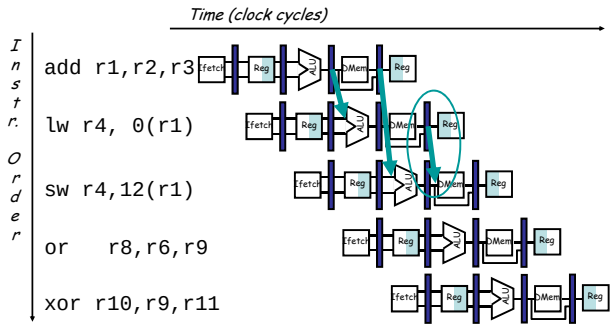
HW Change for Forwarding



What circuit detects and resolves this hazard?

40

Forwarding to Avoid LW-SW Data Hazard



41

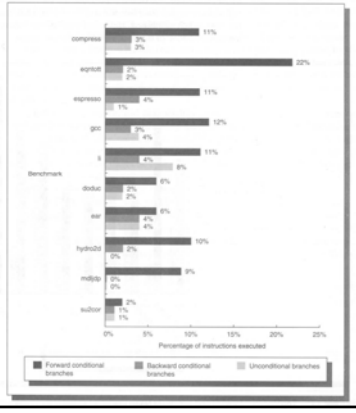
Control Dependences

- Conditional branches
  - Branch must execute to determine which instruction to fetch next
  - Instructions following a conditional branch are control dependent on the branch instruction
- Unconditional Branches (including subroutine calls
  - Branch can't take place until branch target address is calculated
- Exceptions
  - Interrupts
  - Hardware Exceptions
  - Trap Instructions

42

Branch Frequencies

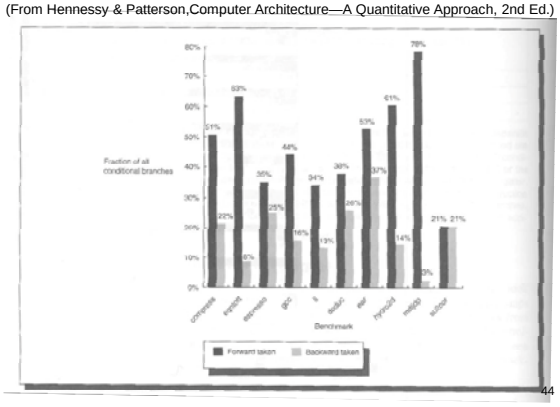
(From Hennessy & Patterson,Computer Architecture—A Quantitative Approach, 2nd Ed



43

Branching Behavior

(From Hennessy & Patterson,Computer Architecture—A Quantitative Approach, 2nd Ed.)



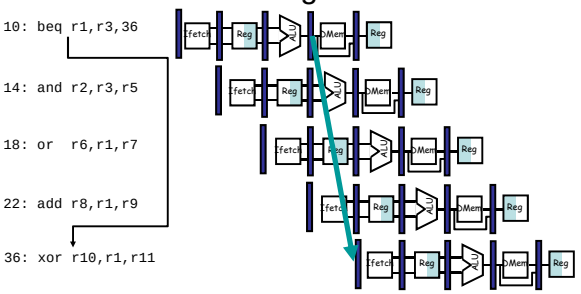
44

Control Flow Hazards

- Important Pipeline Considerations:
  - Where is branch target address (BTA) computed?
  - For conditional branches, how/where is the branch outcome determined.
- For our 5 stage pipeline
  - BTA is computed in EX stage, PC update done during IF stage
  - Branch Outcome is determined during EX stage.

45

Control Hazard on Branches Three Stage Stall



What do you do with the 3 instructions in between?
How do you do it?
Where is the "commit"?

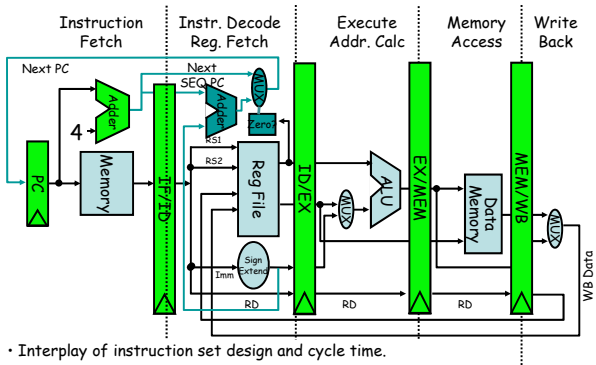
46

Branch Stall Impact

- If CPI = 1, 30% branch,
  Stall 3 cycles => new CPI = 1.9!
- Two part solution:
  - Determine branch outcome(taken/not-taken) sooner, AND
  - Compute branch target address earlier
- MIPS branch tests if register = 0 or ≠ 0
- MIPS Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

47

Pipelined MIPS Datapath



- Interplay of instruction set design and cycle time.
- Hardware Cost: Additional Adder for BTA generation

48

Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken
- Execute successor instructions in sequence
- "Cancel" instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- 47% MIPS branches not taken on average
- PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken
- 53% MIPS branches taken on average
- But haven't calculated branch target address in MIPS
  - MIPS still incurs 1 cycle branch penalty
  - Other machines: branch target known before outcome

49

Four Branch Hazard Alternatives

#4: Delayed Branch
- Define branch to take place AFTER following instruction(s)

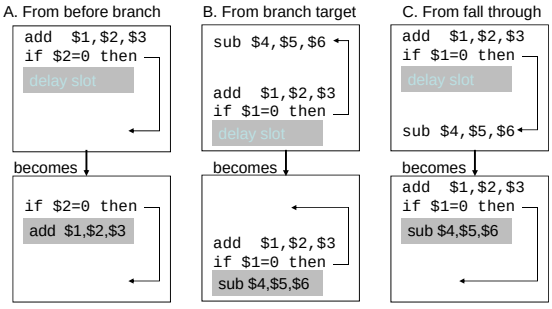
```
branch instruction
 sequential successor1
 sequential successor2
 ........
 sequential successorn
branch target if taken
```

Branch delay of length *n* *(branch shadow)*

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

50

Scheduling Branch Delay Slots



- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the sub instruction may need to be copied, increasing IC
- In B and C, must be okay to execute sub when branch fails

51

Delayed Branch

- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled
- Delayed Branch downside: As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot
  - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
  - Growth in available transistors has made dynamic approaches relatively cheaper

52

Evaluating Branch Alternatives

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

Assume 4% unconditional branch, 6% conditional branch- untaken, 10% conditional branch-taken

Scheduling scheme	*Branch penalty*	*CPI*	*speedup v. unpipelined*	*speedup v. stall*
Stall pipeline	1	1.2	4.17	1.0
Predict not taken	1*	1.14	4.39	1.05
Delayed branch	0.5	1.10	4.55	1.09

* Only for wrong prediction

Assumes Branch Outcome determination and BTA generation in decode stage, 50% of delay slots filled with useful instructions for delayed branching

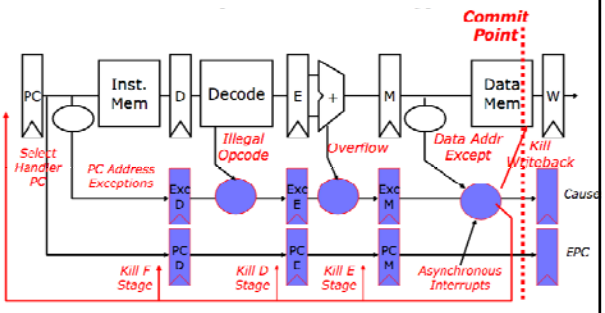
53

Problems with Pipelining

- Exception: An unusual event happens to an instruction during its execution
  - Examples: divide by zero, undefined opcode
- Interrupt: Hardware signal to switch the processor to a new instruction stream
  - Example: a sound card interrupts when it needs more audio output samples (an audio "click" happens if it is left waiting)
- Problem: It must appear that the exception or interrupt must appear between 2 instructions (I_i and I_{i+1})
  - The effect of all instructions up to and including I_i is totally complete
  - No effect of any instruction after I_i can take place
- or The interrupt (exception) handler either aborts program restarts at instruction I_{i+1}

54

Precise Exceptions in Static Pipelines



Key observation: architected state only change in memory and register write stages.

55

Limits on Scalar Pipeline Performance [Agerwala and Cocke 1987]

- Internal IBM study: Limits of a scalar pipeline?
- Memory Bandwidth
  - Fetch 1 instr/cycle from I-cache
  - 40% of instructions are load/store (D-cache)
- Code characteristics (dynamic)
  - Loads – 25%
  - Stores 15%
  - ALU/RR – 40%
  - Branches – 20%
    - 1/3 unconditional (always taken)
    - 1/3 conditional taken
    - 1/3 conditional not taken

Limits on Scalar Processor Performance

- Cache Performance
  - Assume 100% hit ratio (upper bound)
  - Cache latency: I = D = 1 cycle default
- Load and branch scheduling
  - Loads
    - 25% cannot be scheduled (delay slot empty)
    - 65% can be moved back 1 or 2 instructions
    - 10% can be moved back 1 instruction
  - Branches
    - Unconditional – 100% schedulable (fill one delay slot)
    - Conditional – 50% schedulable (fill one delay slot)

CPI Optimizations

- Goal and impediments
  - CPI = 1, prevented by pipeline stalls
- No cache bypass of RF, no load/branch scheduling
  - Load penalty: 2 cycles: 0.25 x 2 = 0.5 CPI
  - Branch penalty: 2 cycles: 0.2 x 2/3 x 2 = 0.27 CPI
  - Total CPI: 1 + 0.5 + 0.27 = 1.77 CPI
- Bypass, no load/branch scheduling
  - Load penalty: 1 cycle: 0.25 x 1 = 0.25 CPI
  - Total CPI: 1 + 0.25 + 0.27 = 1.52 CPI

More CPI Optimizations

- Bypass, scheduling of loads/branches
  - Load penalty:
    - 65% + 10% = 75% moved back, no penalty
    - 25% => 1 cycle penalty
    - 0.25 x 0.25 x 1 = 0.0625 CPI
  - Branch Penalty
    - 1/3 unconditional 100% schedulable => 1 cycle
    - 1/3 cond. not-taken, => no penalty (predict not-taken)
    - 1/3 cond. Taken, 50% schedulable => 1 cycle
    - 1/3 cond. Taken, 50% unschedulable => 2 cycles
    - 0.25 x [1/3 x 1 + 1/3 x 0.5 x 1 + 1/3 x 0.5 x 2] = 0.167
- Total CPI: 1 + 0.063 + 0.167 = 1.23 CPI

Simplify Branches

- Assume 90% can be PC-relative
  - No register indirect, no register access
  - Separate adder (like MIPS R3000)
  - Branch penalty reduced
- Total CPI: 1 + 0.063 + 0.085 = 1.15 CPI

15% Overhead from program dependences

PC-relative	Schedulable	Penalty
Yes (90%)	Yes (50%)	0 cycle
Yes (90%)	No (50%)	1 cycle
No (10%)	Yes (50%)	1 cycle
No (10%)	No (50%)	2 cycles