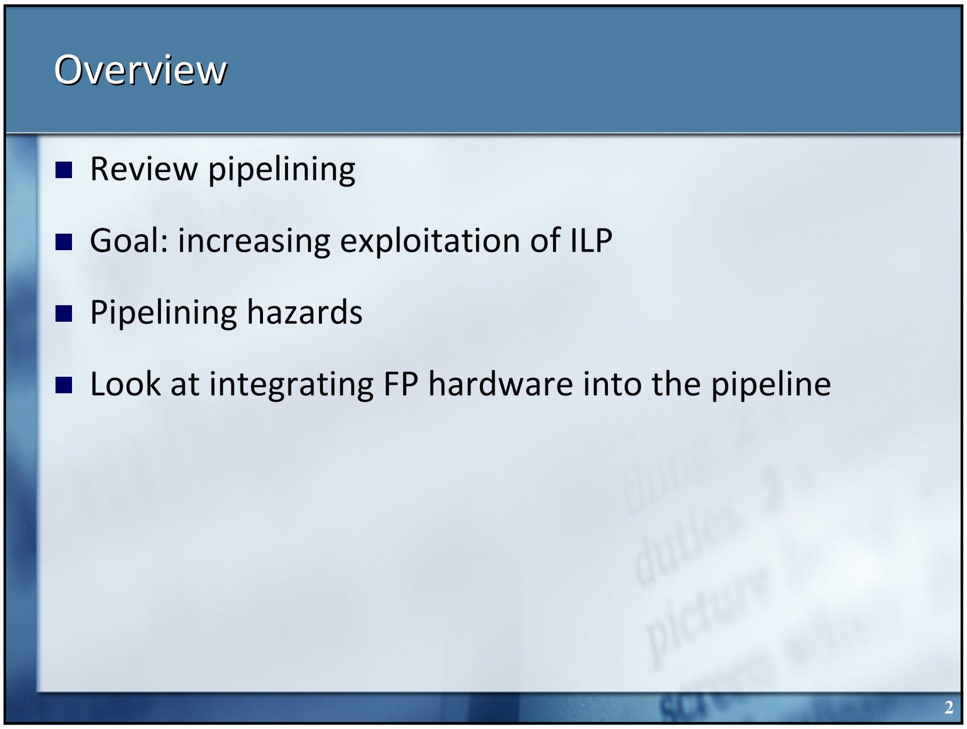
CDA 5106 Advanced Computer Architecture 1

Module 2 | Pipelining Overview



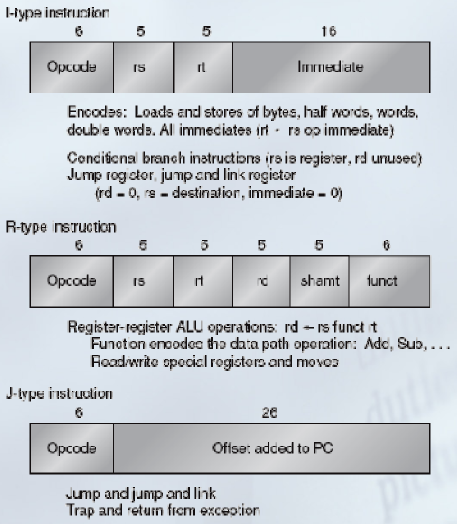
Overview

- Review pipelining
- Goal: increasing exploitation of ILP
- Pipelining hazards
- Look at integrating FP hardware into the pipeline

Designing a processor

- Design the ISA
- Classify instructions for the ISA (e.g., MIPS):
  - – Memory references
  - – Register-Register ALU Operations
  - – Register-Immediate ALU Operations
  - – Branches
- Work out the execution for each operation class
- Design appropriate hardware
- Look for opportunities to improve…
- …while maintaining correct execution

MIPS Instruction Layout



I-type instruction

6	5	5	16
Opcode	rs	rt	Immediate

Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd = 0, rs = destination, immediate = 0)

R-type instruction

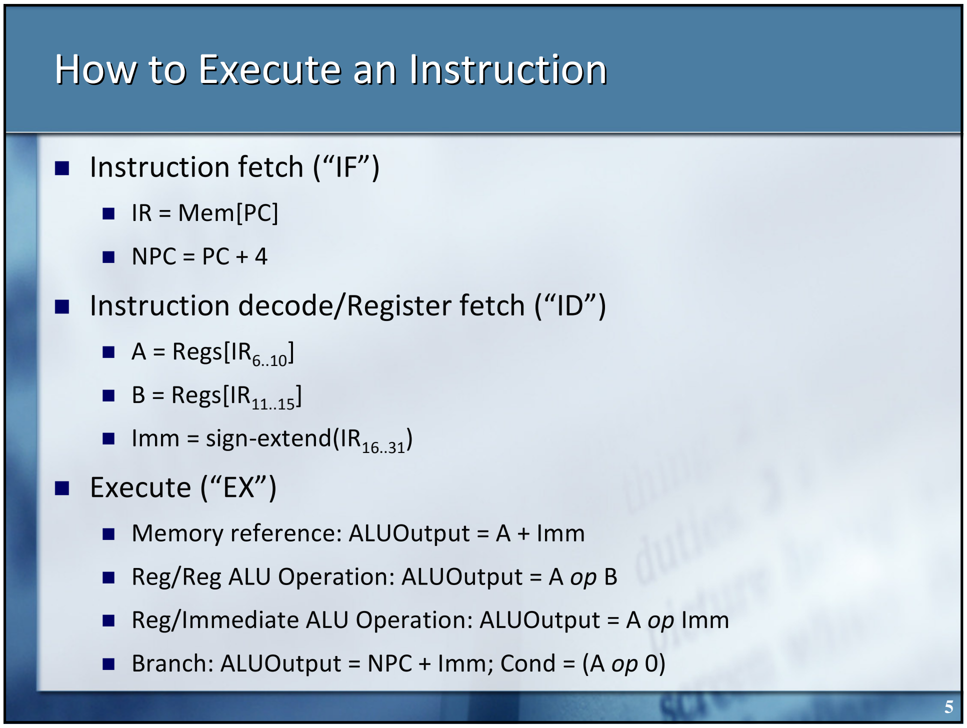
6	5	5	5	5	6
Opcode	rs	rt	rd	shamt	funct

Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, . . .
Read/write special registers and moves

J-type instruction

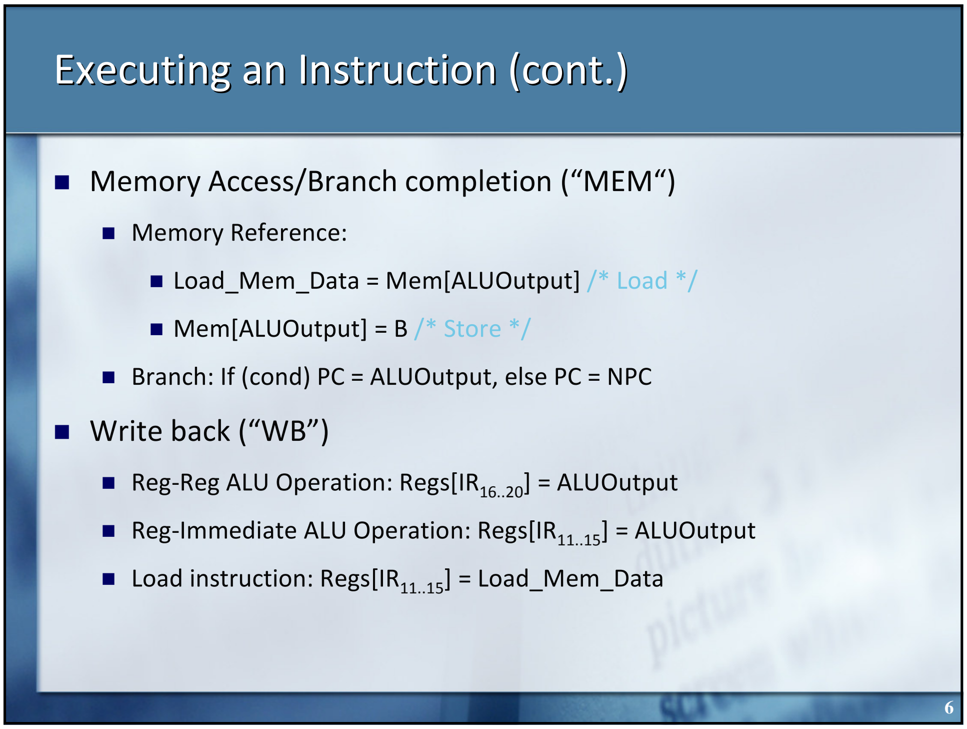
6	26
Opcode	Offset added to PC

Jump and jump and link
Trap and return from exception



How to Execute an Instruction

- Instruction fetch (“IF”)
  - IR = Mem[PC]
  - NPC = PC + 4
- Instruction decode/Register fetch (“ID”)
  - A = Regs[$IR_{6..10}$]
  - B = Regs[$IR_{11..15}$]
  - Imm = sign-extend($IR_{16..31}$)
- Execute (“EX”)
  - Memory reference: ALUOutput = A + Imm
  - Reg/Reg ALU Operation: ALUOutput = A *op* B
  - Reg/Immediate ALU Operation: ALUOutput = A *op* Imm
  - Branch: ALUOutput = NPC + Imm; Cond = (A *op* 0)

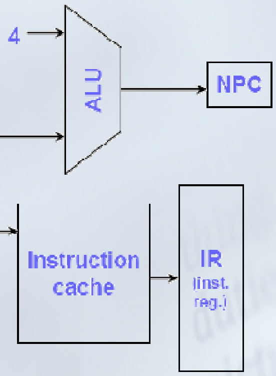


Executing an Instruction (cont.)

- Memory Access/Branch completion (“MEM“)
  - Memory Reference:
    - Load_Mem_Data = Mem[ALUOutput] /* Load */
    - Mem[ALUOutput] = B /* Store */
  - Branch: If (cond) PC = ALUOutput, else PC = NPC
- Write back (“WB”)
  - Reg-Reg ALU Operation: Regs[$IR_{16..20}$] = ALUOutput
  - Reg-Immediate ALU Operation: Regs[$IR_{11..15}$] = ALUOutput
  - Load instruction: Regs[$IR_{11..15}$] = Load_Mem_Data

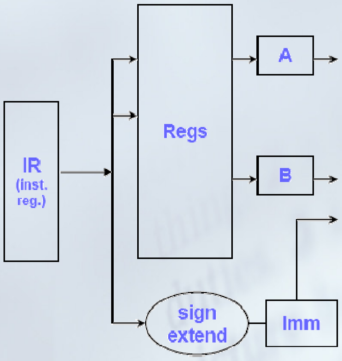
Executing an Instruction (cont.)

- Instruction fetch (“IF”)
  - IR = Mem[PC]
  - NPC = PC + 4



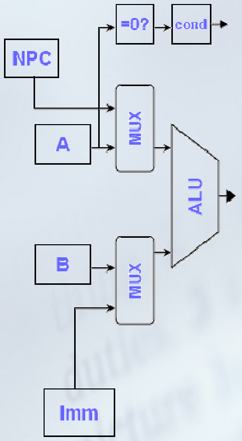
Executing an Instruction (cont.)

- Instruction decode/Register fetch (“ID”)
  - A = Regs[rs]
  - B = Regs[rt]
  - Imm= sign-extend(IR_{imm})



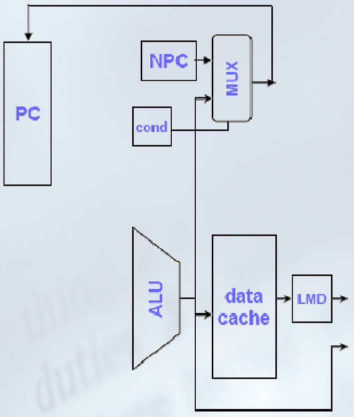
Executing an Instruction (cont.)

- Execute ("EX")
  - Memory reference:
    ALUOutput= A + Imm
  - Reg/Reg ALU Operation:
    ALUOutput= A *op* B
  - Reg/Immediate ALU Operation:
    ALUOutput= A *op* Imm
  - Branch:
    ALUOutput= NPC + Imm;
    Cond = (A *op* 0)



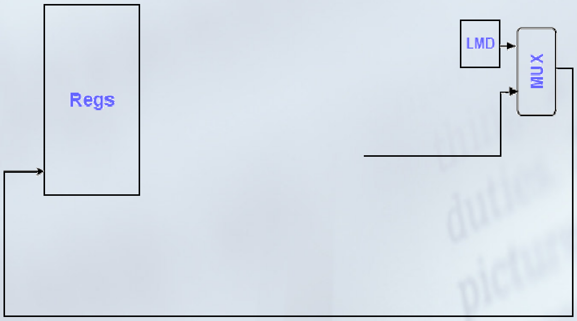
Executing an Instruction (cont.)

- Memory Access/Branch completion ("MEM")
  - Memory Reference:
    - Load_Mem_Data = Mem[ALUOutput]
      /* Load */
    - Mem[ALUOutput] = B
      /* Store */
  - Branch: If (cond) PC = ALUOutput,
    else PC = NPC

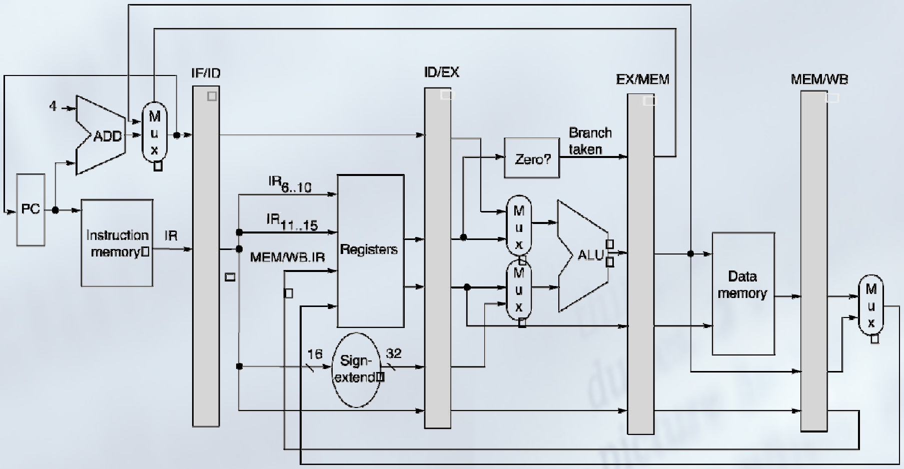


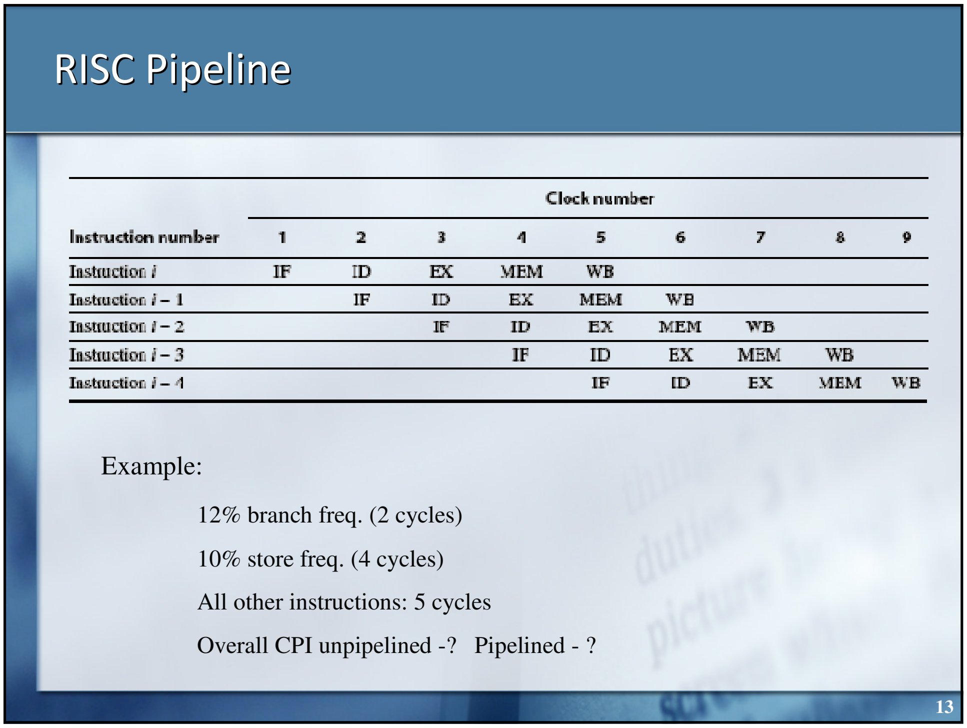
Executing an Instruction (cont.)

- Write back ("WB")
  - Reg-Reg ALU Operation: Regs[rd] = ALUOutput
  - Reg-Immediate ALU Operation: Regs[rt] = ALUOutput
  - Load instruction: Regs[rt] = Load_Mem_Data



Five-stage statically scheduled pipeline





RISC Pipeline

Instruction number	1	2	3	4	5	6	7	8	9
	Clock number								
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* – 1		IF	ID	EX	MEM	WB			
Instruction *i* – 2			IF	ID	EX	MEM	WB		
Instruction *i* – 3				IF	ID	EX	MEM	WB	
Instruction *i* – 4					IF	ID	EX	MEM	WB

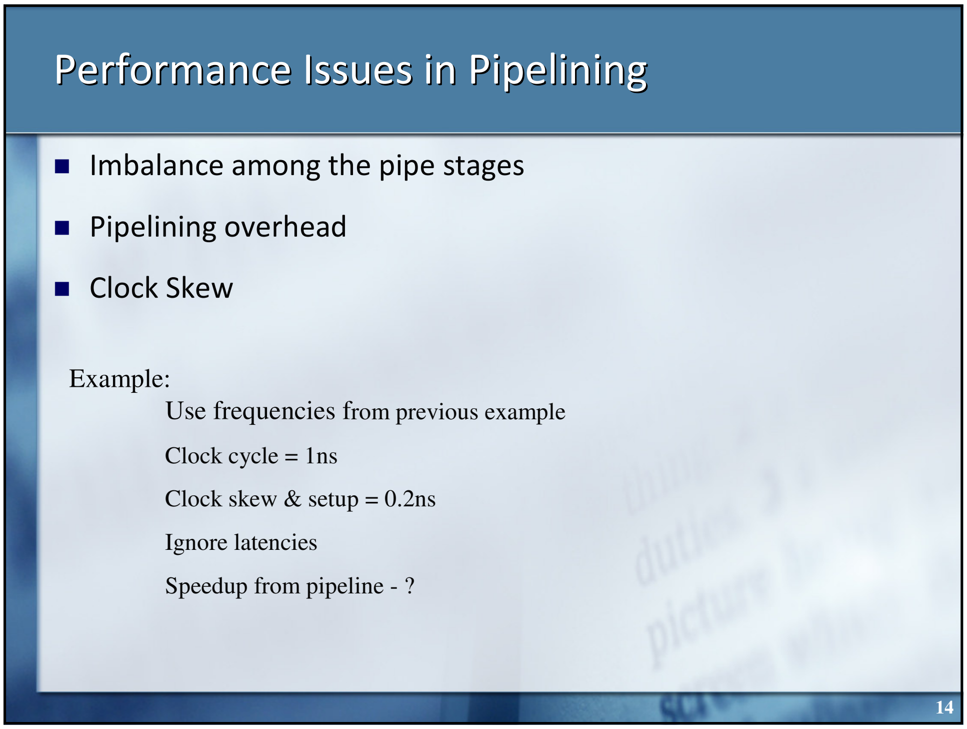
Example:

12% branch freq. (2 cycles)

10% store freq. (4 cycles)

All other instructions: 5 cycles

Overall CPI unpipelined -? Pipelined - ?



Performance Issues in Pipelining

- Imbalance among the pipe stages
- Pipelining overhead
- Clock Skew

Example:

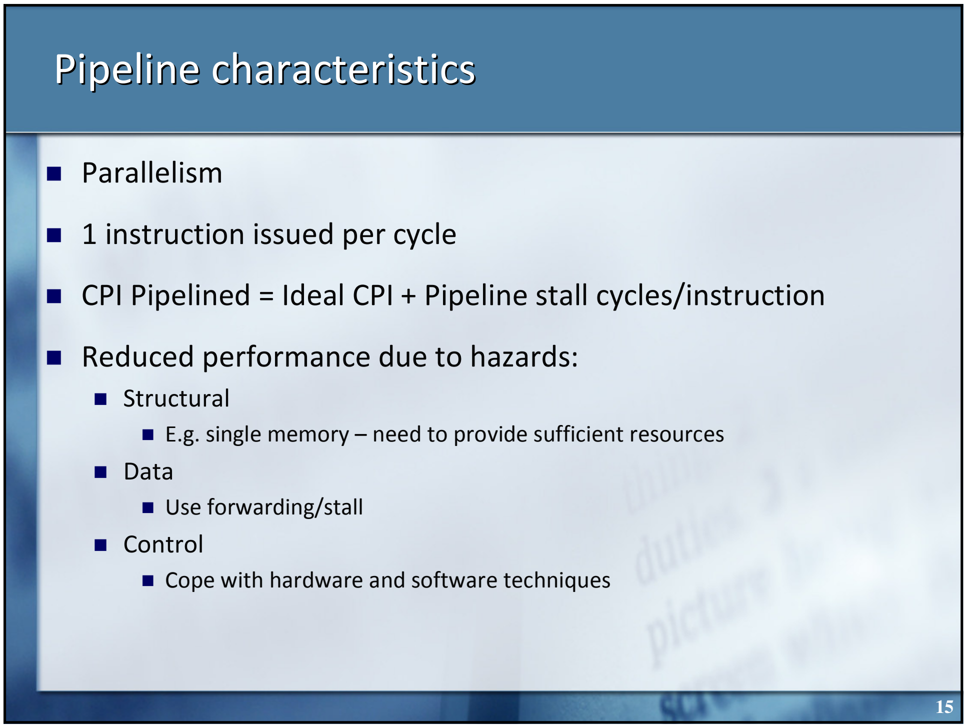
Use frequencies from previous example

Clock cycle = 1ns

Clock skew & setup = 0.2ns

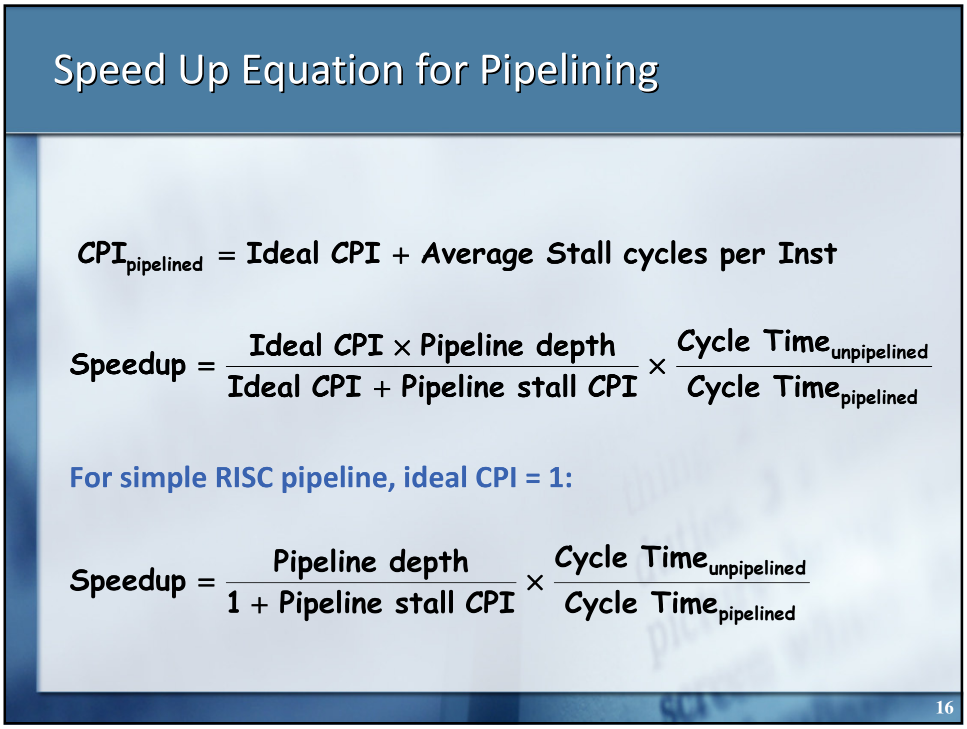
Ignore latencies

Speedup from pipeline - ?



Pipeline characteristics

- Parallelism
- 1 instruction issued per cycle
- CPI Pipelined = Ideal CPI + Pipeline stall cycles/instruction
- Reduced performance due to hazards:
  - Structural
    - E.g. single memory – need to provide sufficient resources
  - Data
    - Use forwarding/stall
  - Control
    - Cope with hardware and software techniques



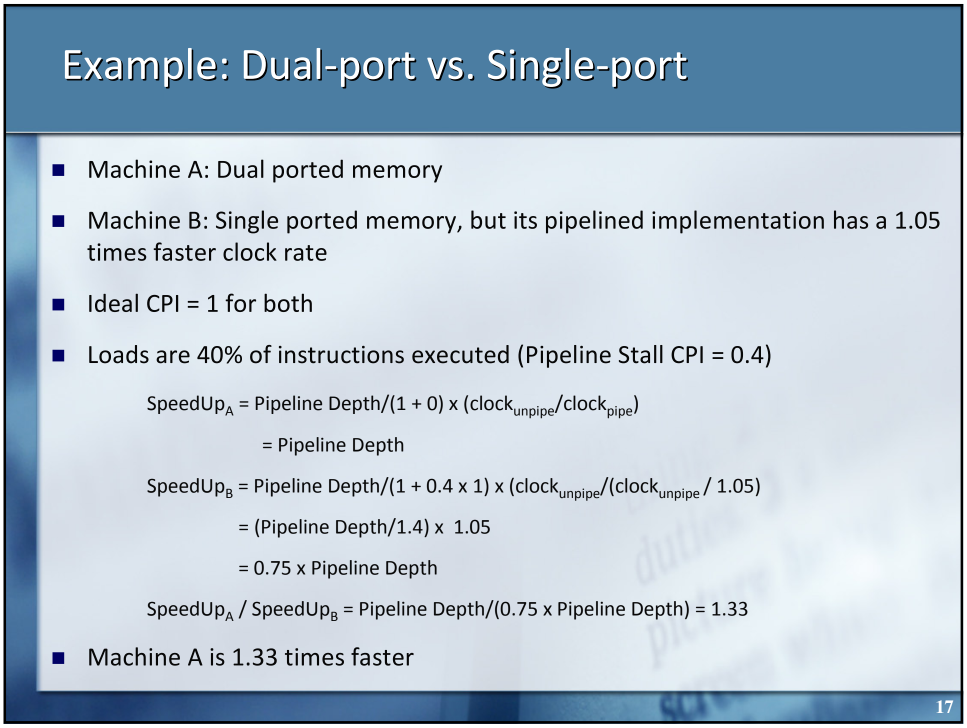
Speed Up Equation for Pipelining

$$CPI_{pipelined} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

For simple RISC pipeline, ideal CPI = 1:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$



Example: Dual-port vs. Single-port

- Machine A: Dual ported memory
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed (Pipeline Stall CPI = 0.4)

SpeedUp_A = Pipeline Depth/(1 + 0) x ($\text{clock}_{\text{unpipe}}/\text{clock}_{\text{pipe}}$)

= Pipeline Depth

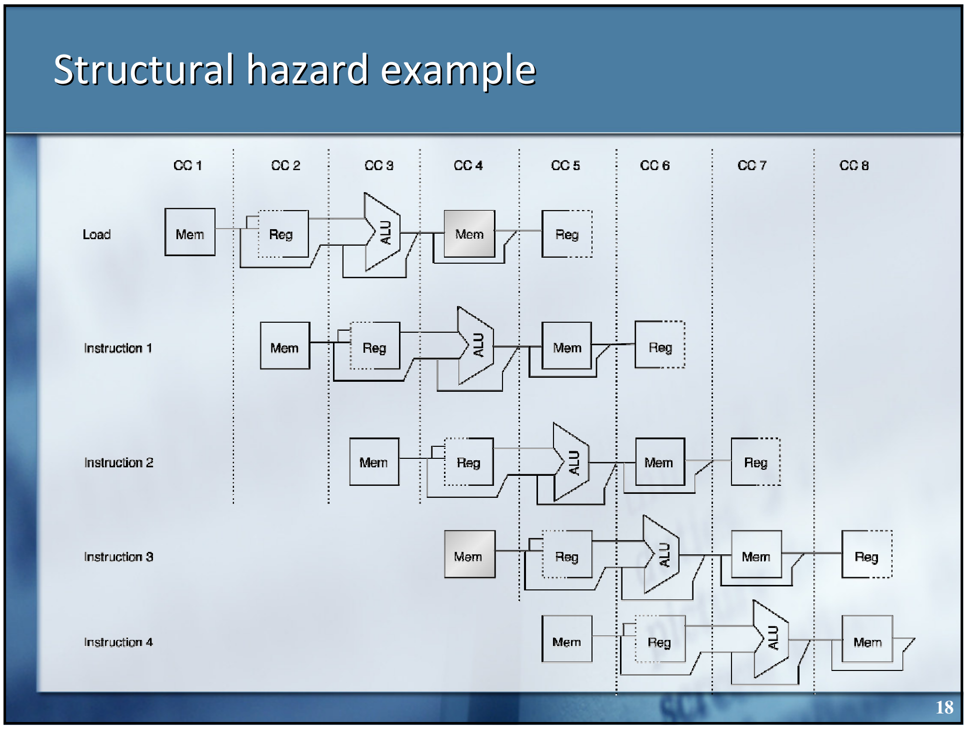
SpeedUp_B = Pipeline Depth/(1 + 0.4 x 1) x ($\text{clock}_{\text{unpipe}}/(\text{clock}_{\text{unpipe}}$ / 1.05)

= (Pipeline Depth/1.4) x 1.05

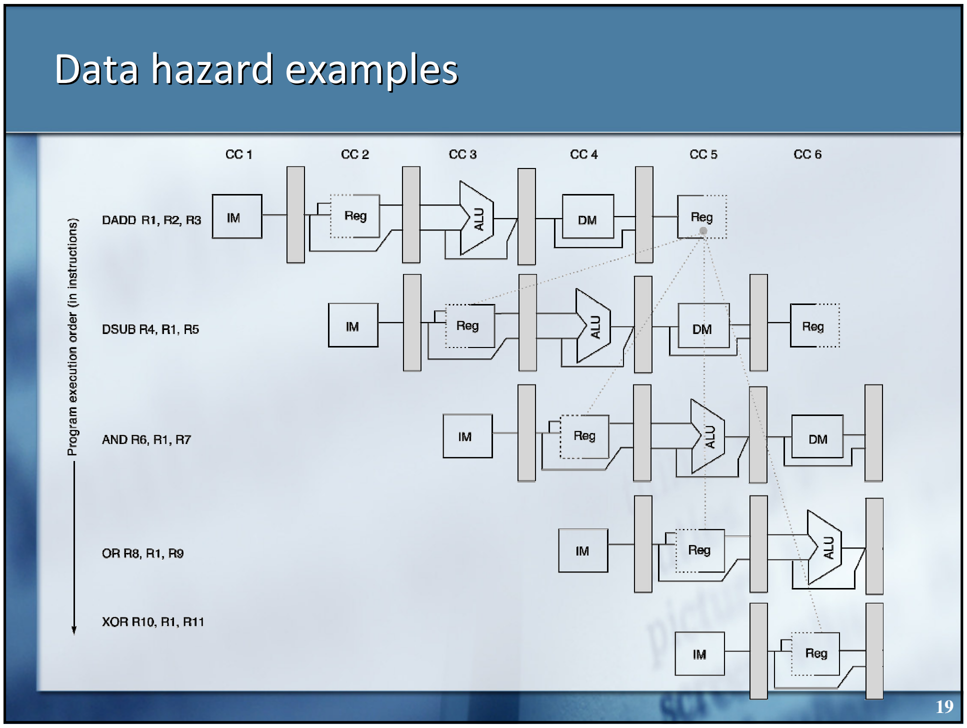
= 0.75 x Pipeline Depth

SpeedUp_A / SpeedUp_B = Pipeline Depth/(0.75 x Pipeline Depth) = 1.33

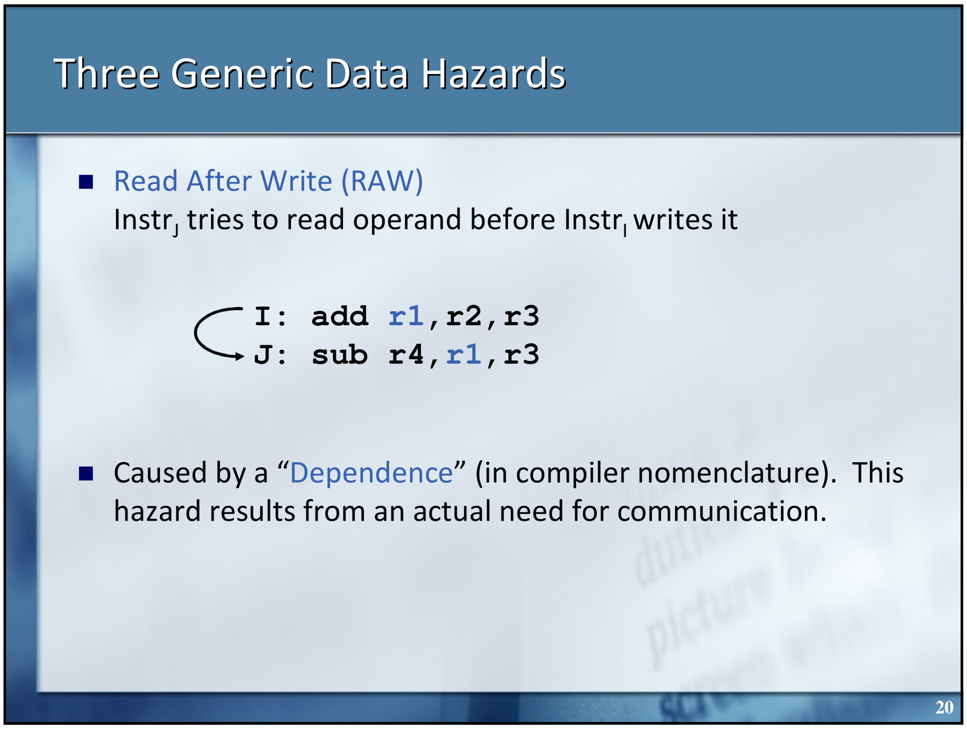
- Machine A is 1.33 times faster



Structural hazard example



Data hazard examples

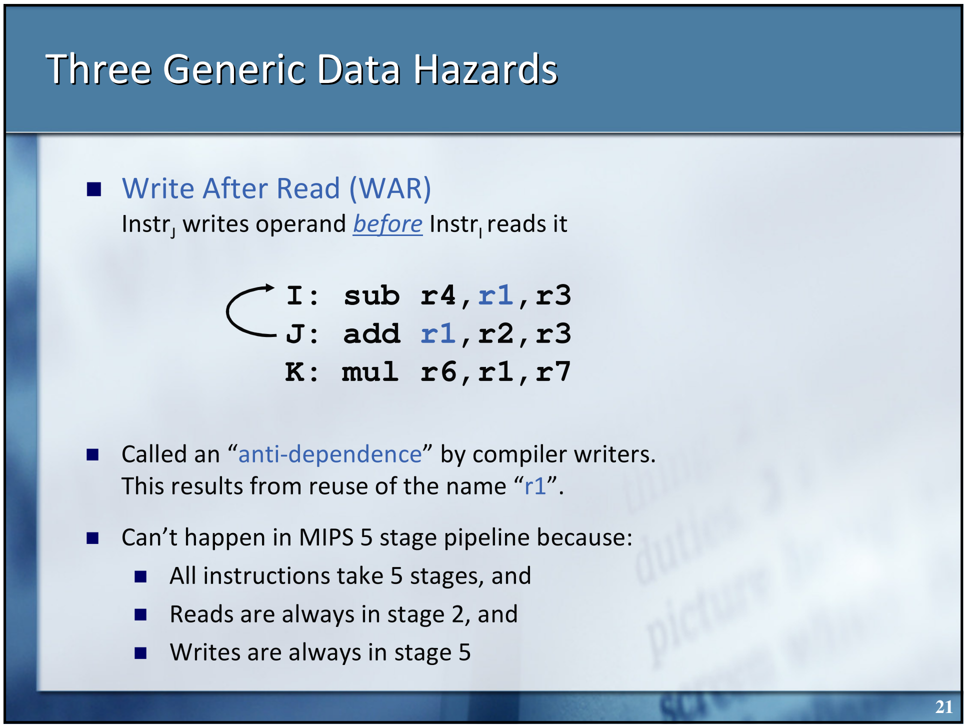


Three Generic Data Hazards

- 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.

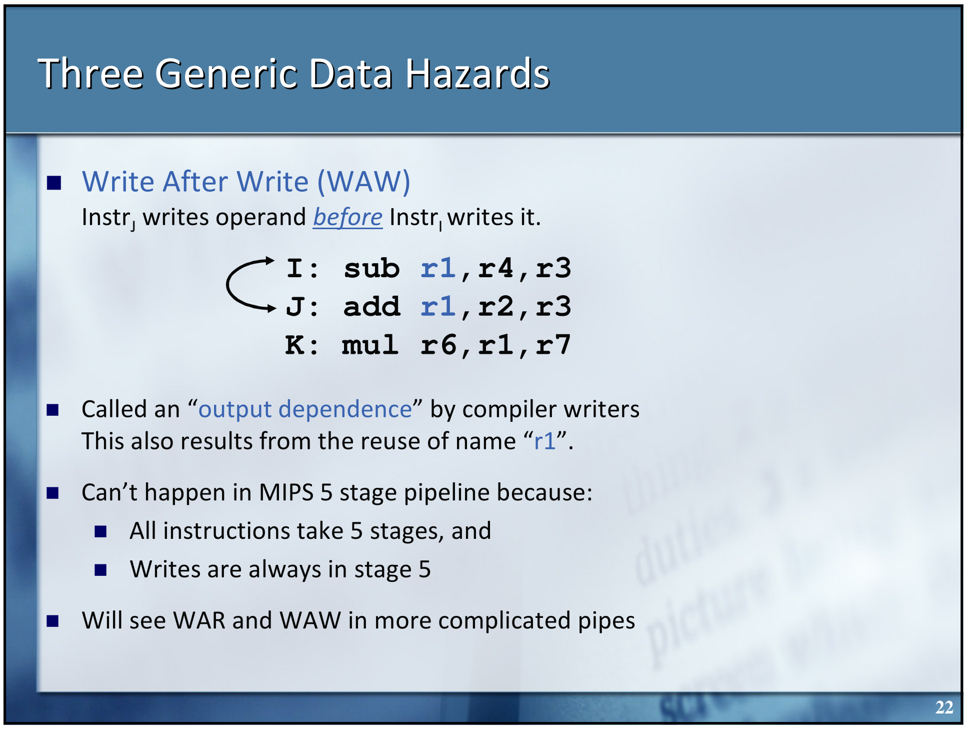


Three Generic Data Hazards

- 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
  - Reads are always in stage 2, and
  - Writes are always in stage 5



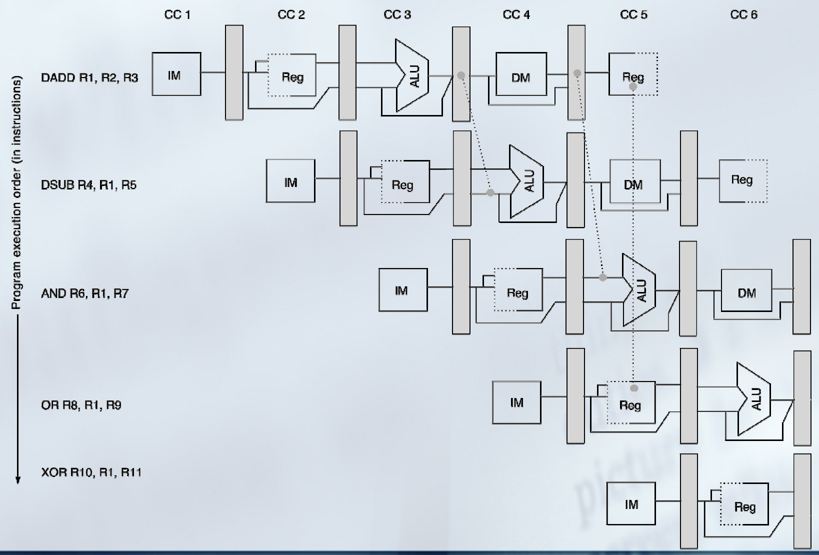
Three Generic Data Hazards

- Write After Write (WAW)
  Instr$_J$ writes operand *before* Instr$_I$ writes it.

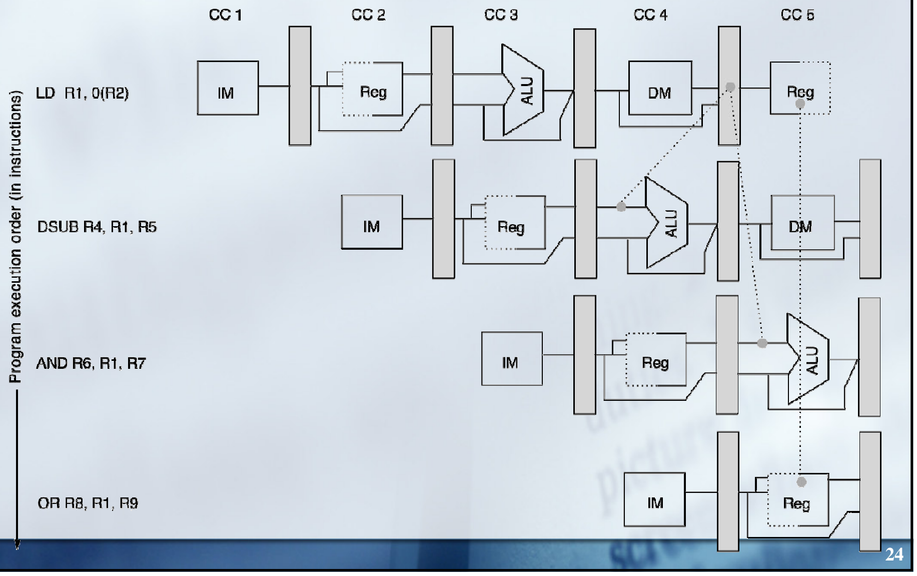
```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

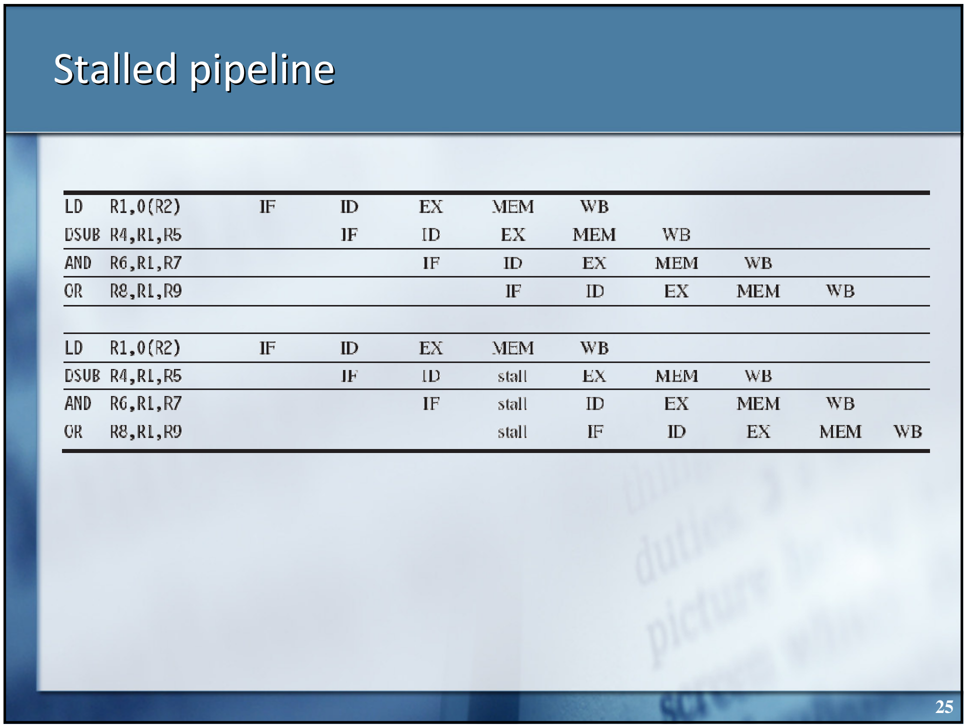
- Called an "output dependence" by compiler writers
  This also results from the reuse of name "r1".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5
- Will see WAR and WAW in more complicated pipes

Data hazard remedy – forwarding



Data hazard needing stall

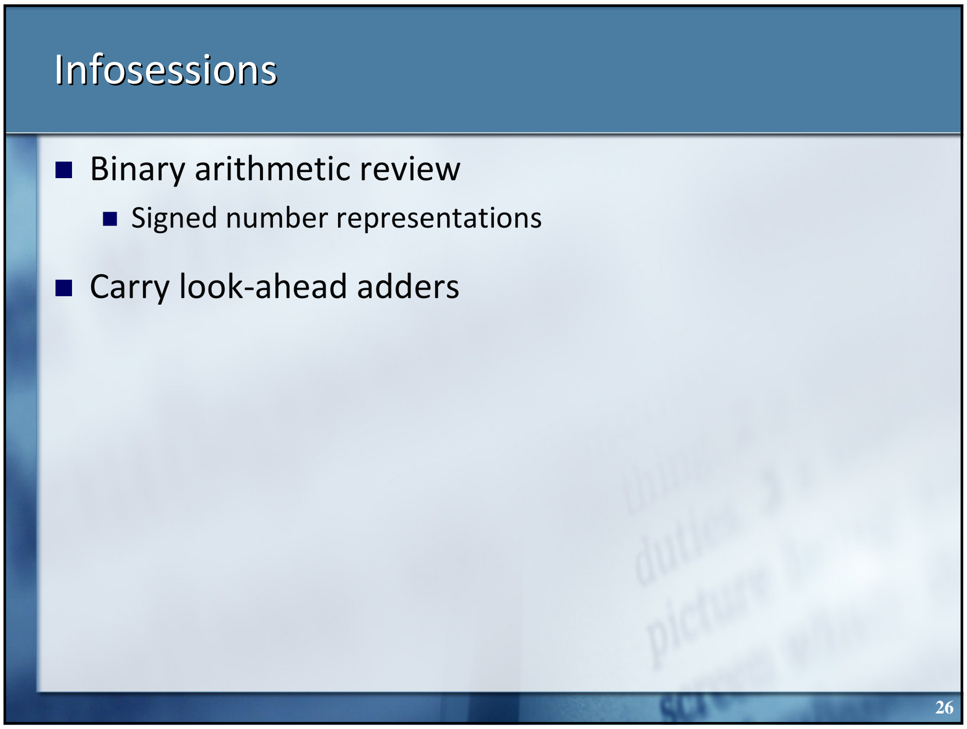




Stalled pipeline

LD R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB R4,R1,R5		IF	ID	EX	MEM	WB			
AND R6,R1,R7			IF	ID	EX	MEM	WB		
OR R8,R1,R9				IF	ID	EX	MEM	WB	

LD R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB R4,R1,R5		IF	ID	stall	EX	MEM	WB		
AND R6,R1,R7			IF	stall	ID	EX	MEM	WB	
OR R8,R1,R9				stall	IF	ID	EX	MEM	WB



Infosessions

- Binary arithmetic review
  - Signed number representations
- Carry look-ahead adders

Software Scheduling to Avoid Hazards

Try producing fast code for

a = b + c;

d = e – f;

assuming a, b, c, d ,e, and f in memory.

Slow code:

```
LW    Rb,b
LW    Rc,c
ADD   Ra,Rb,Rc
SW    a,Ra
LW    Re,e
LW    Rf,f
SUB   Rd,Re,Rf
SW    d,Rd
```

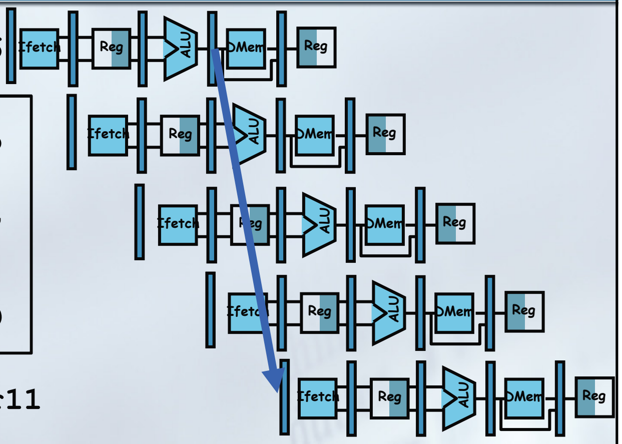
Fast code:

```
LW    Rb,b
LW    Rc,c
LW    Re,e
ADD   Ra,Rb,Rc
LW    Rf,f
SW    a,Ra
SUB   Rd,Re,Rf
SW    d,Rd
```

Compiler optimizes for performance. Hardware checks for safety.

Control Hazard on Branches
Three Stage Stall

```
10: beq r1,r3,36
14: and r2,r3,r5
18: or  r6,r1,r7
22: add r8,r1,r9
36: xor r10,r1,r11
```



What do you do with the 3 instructions in between?

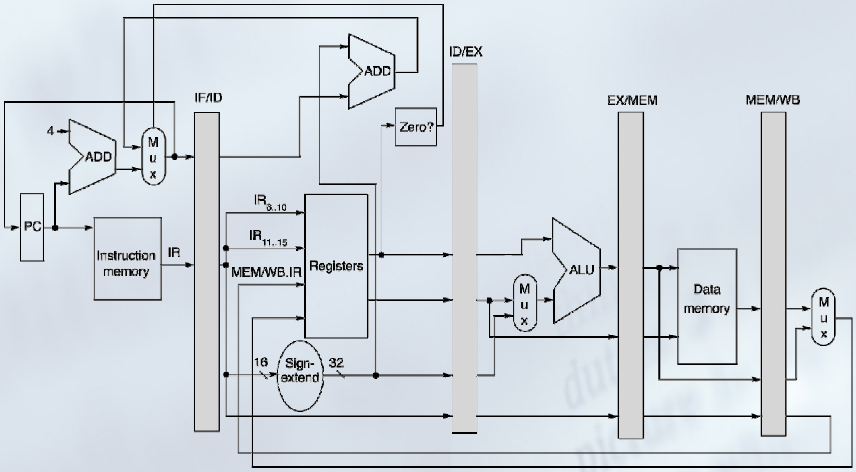
How do you do it?

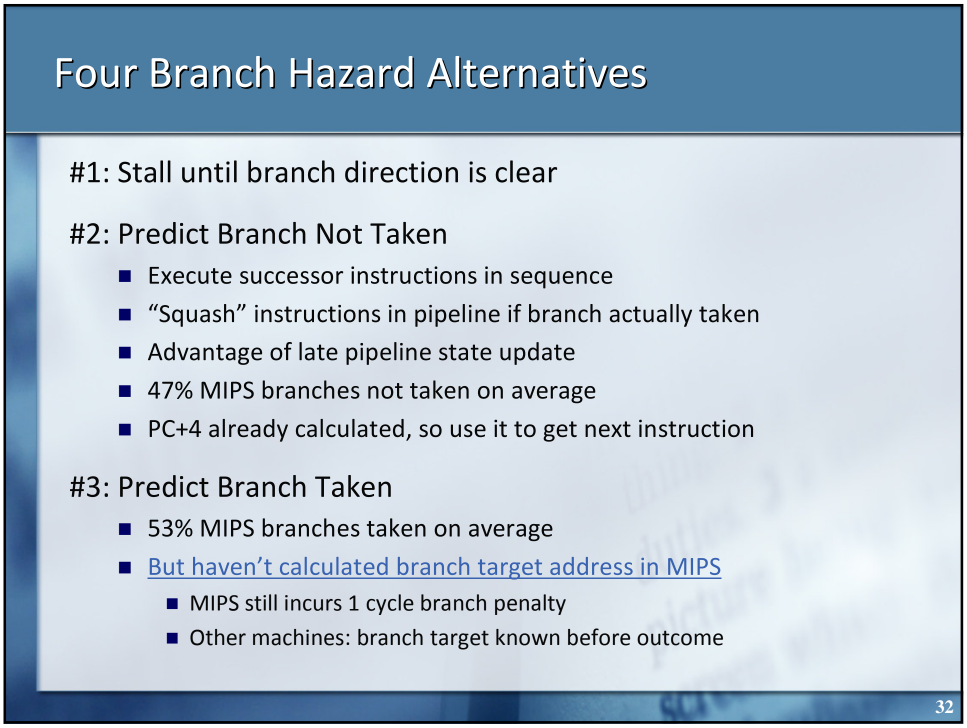
Where is the "commit"?

Branch Stall Impact

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

Branch Modification





Four Branch Hazard Alternatives

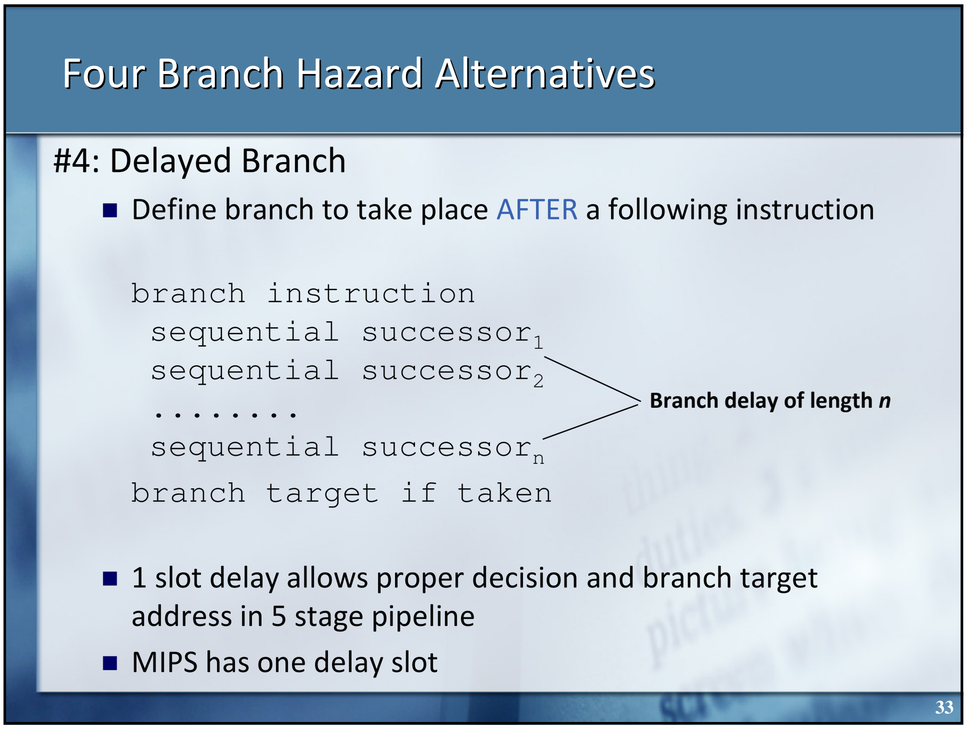
#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

- Execute successor instructions in sequence
- “Squash” 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



Four Branch Hazard Alternatives

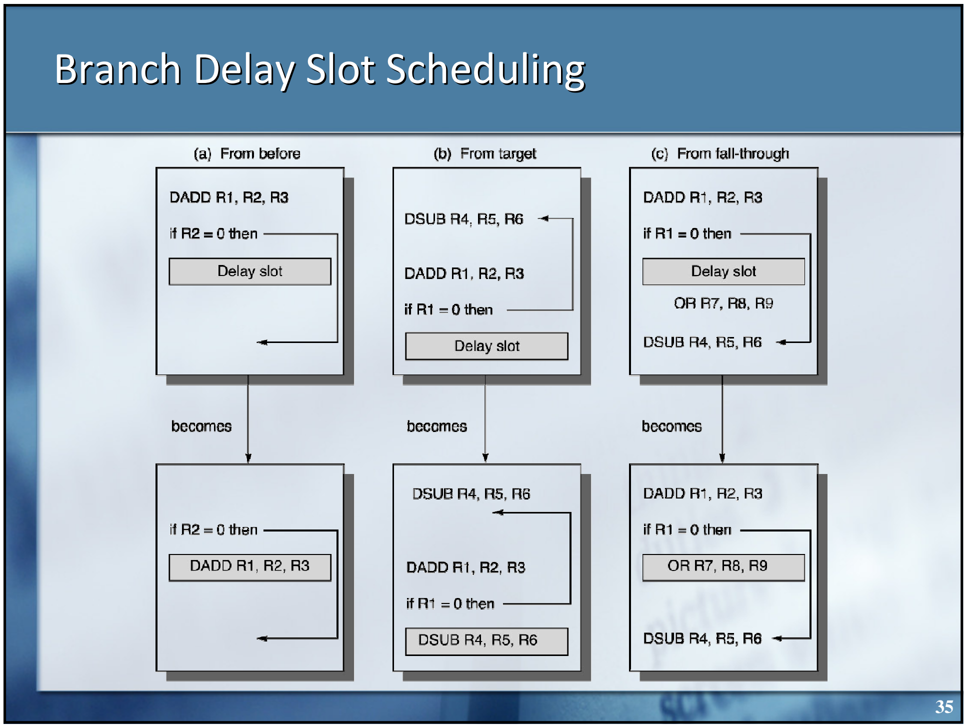
#4: Delayed Branch

- Define branch to take place AFTER a following instruction

branch instruction
  sequential successor$_1$
  sequential successor$_2$
  ....
  sequential successor$_n$
branch target if taken

Branch delay of length *n*

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



Branch Delay Slot Scheduling

(a) From before

DADD R1, R2, R3

if R2 = 0 then

Delay slot

becomes

if R2 = 0 then

DADD R1, R2, R3

(b) From target

DSUB R4, R5, R6

DADD R1, R2, R3

if R1 = 0 then

Delay slot

becomes

DSUB R4, R5, R6

DADD R1, R2, R3

if R1 = 0 then

DSUB R4, R5, R6

(c) From fall-through

DADD R1, R2, R3

if R1 = 0 then

Delay slot

OR R7, R8, R9

DSUB R4, R5, R6

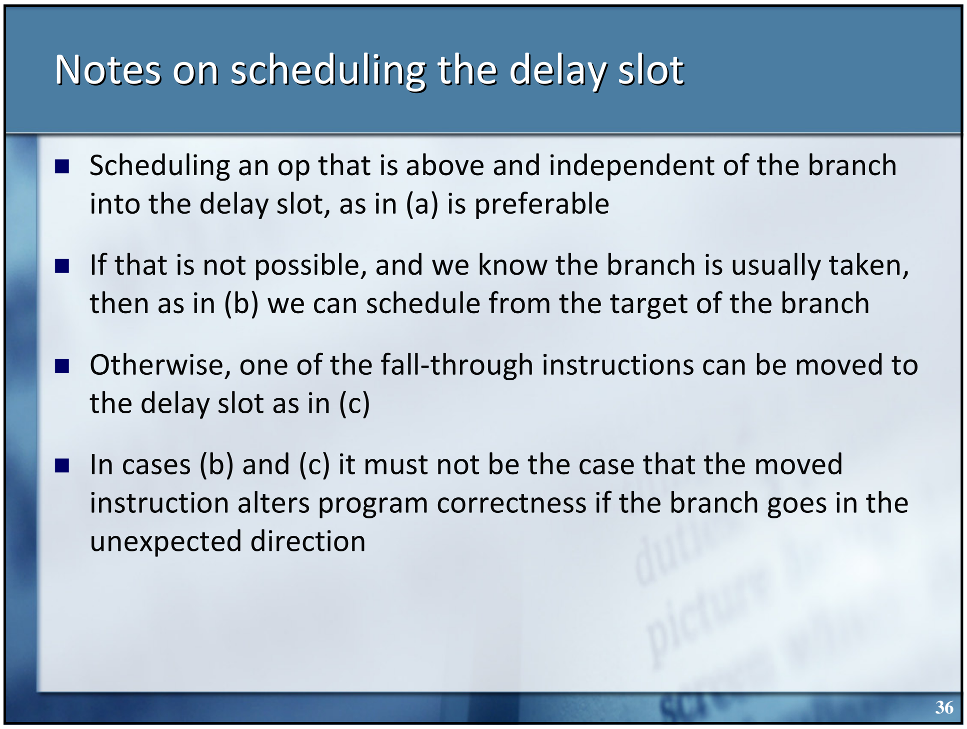
becomes

DADD R1, R2, R3

if R1 = 0 then

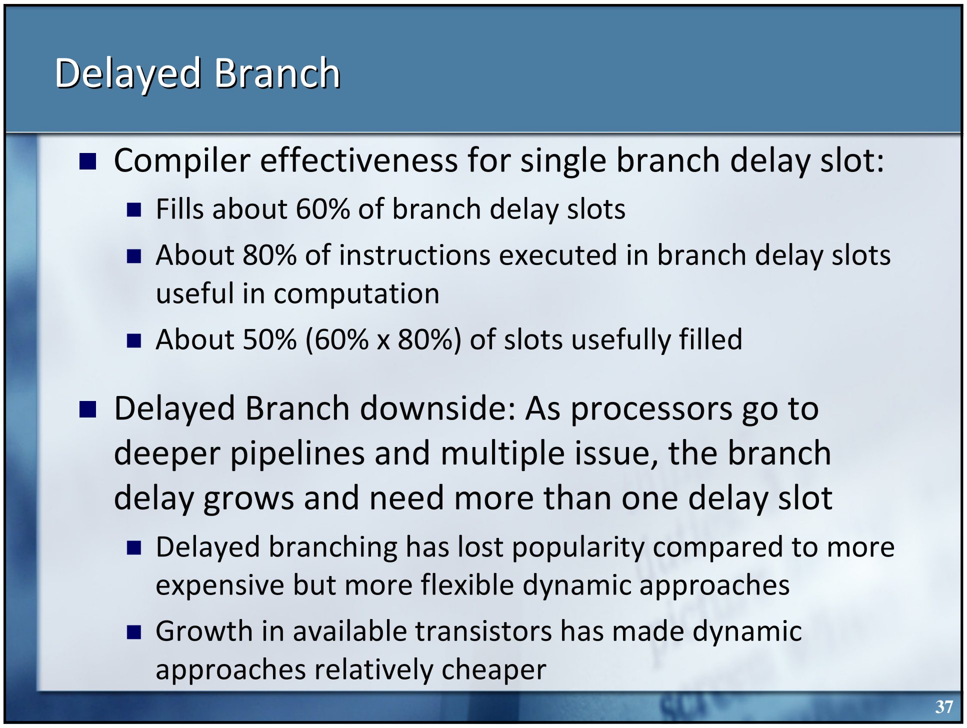
OR R7, R8, R9

DSUB R4, R5, R6



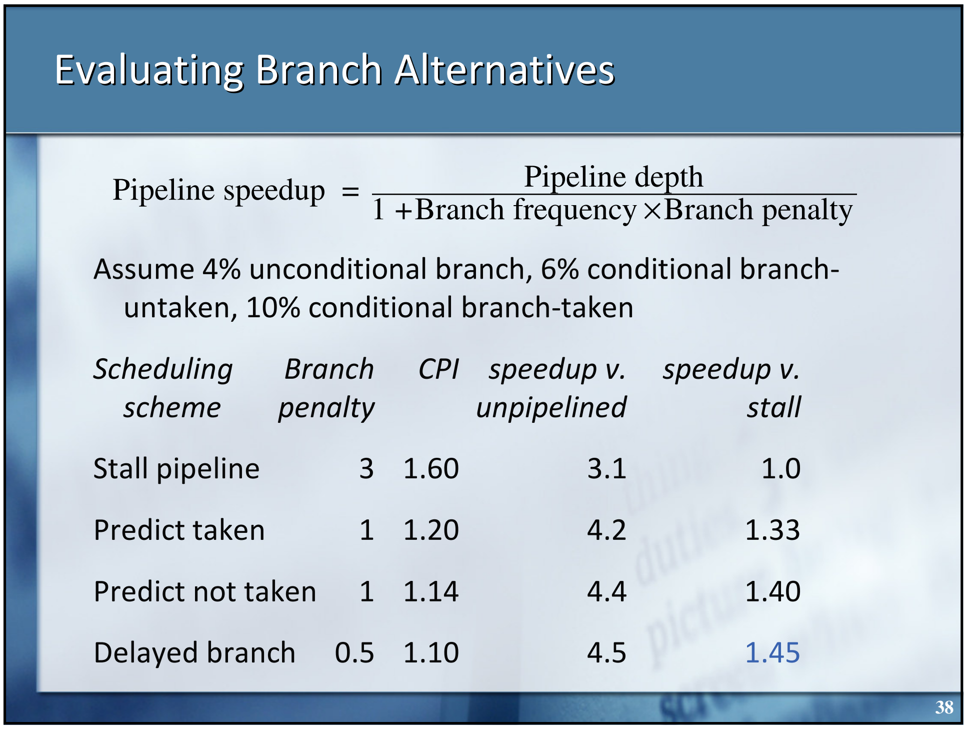
Notes on scheduling the delay slot

- Scheduling an op that is above and independent of the branch into the delay slot, as in (a) is preferable
- If that is not possible, and we know the branch is usually taken, then as in (b) we can schedule from the target of the branch
- Otherwise, one of the fall-through instructions can be moved to the delay slot as in (c)
- In cases (b) and (c) it must not be the case that the moved instruction alters program correctness if the branch goes in the unexpected direction



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 processors 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



Evaluating Branch Alternatives

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

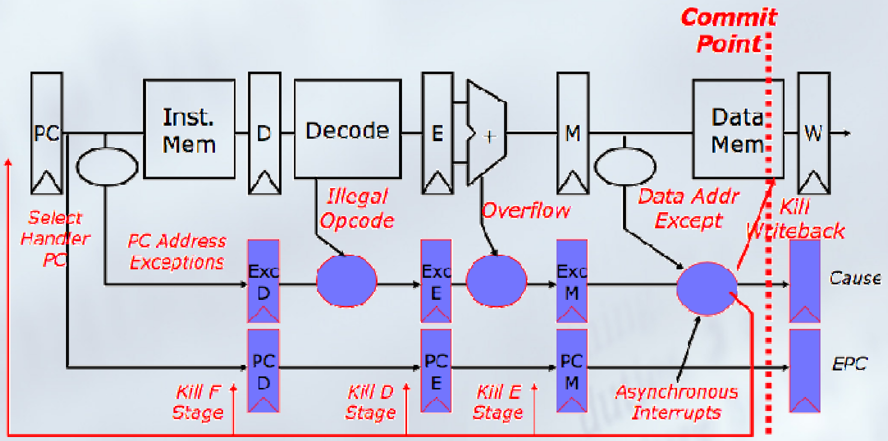
Assume 4% unconditional branch, 6% conditional branch-untaken, 10% conditional branch-taken

Scheduling scheme	*Branch penalty*	*CPI*	*speedup v. unpipelined*	*speedup v. stall*
Stall pipeline	3	1.60	3.1	1.0
Predict taken	1	1.20	4.2	1.33
Predict not taken	1	1.14	4.4	1.40
Delayed branch	0.5	1.10	4.5	1.45

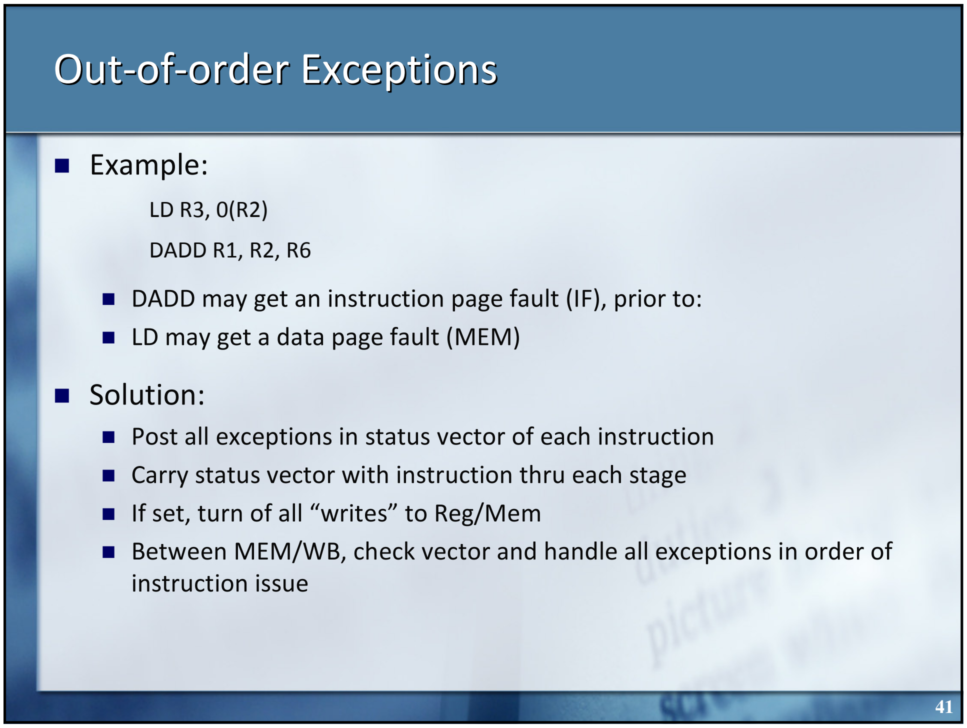
Problems with Pipelining

- 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
- The interrupt (exception) handler either aborts program or restarts at instruction I_{i+1}

Precise Exceptions in Static Pipelines



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

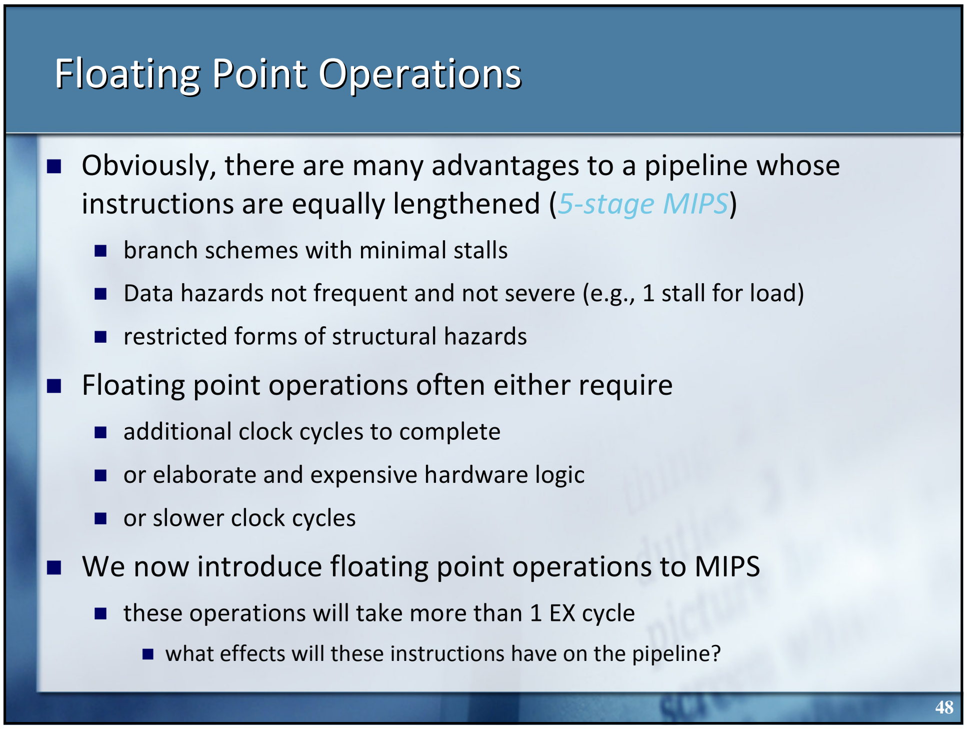


Out-of-order Exceptions

- Example:

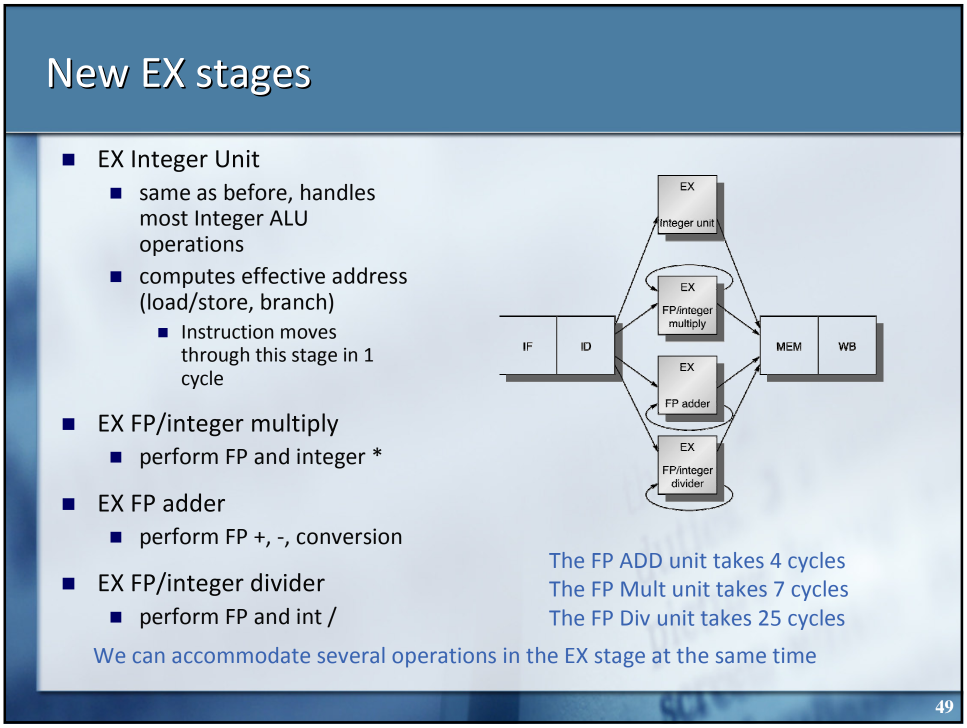
```
LD R3, 0(R2)
DADD R1, R2, R6
```

  - DADD may get an instruction page fault (IF), prior to:
  - LD may get a data page fault (MEM)
- Solution:
  - Post all exceptions in status vector of each instruction
  - Carry status vector with instruction thru each stage
  - If set, turn of all “writes” to Reg/Mem
  - Between MEM/WB, check vector and handle all exceptions in order of instruction issue



Floating Point Operations

- Obviously, there are many advantages to a pipeline whose instructions are equally lengthened (*5-stage MIPS*)
  - branch schemes with minimal stalls
  - Data hazards not frequent and not severe (e.g., 1 stall for load)
  - restricted forms of structural hazards
- Floating point operations often either require
  - additional clock cycles to complete
  - or elaborate and expensive hardware logic
  - or slower clock cycles
- We now introduce floating point operations to MIPS
  - these operations will take more than 1 EX cycle
    - what effects will these instructions have on the pipeline?



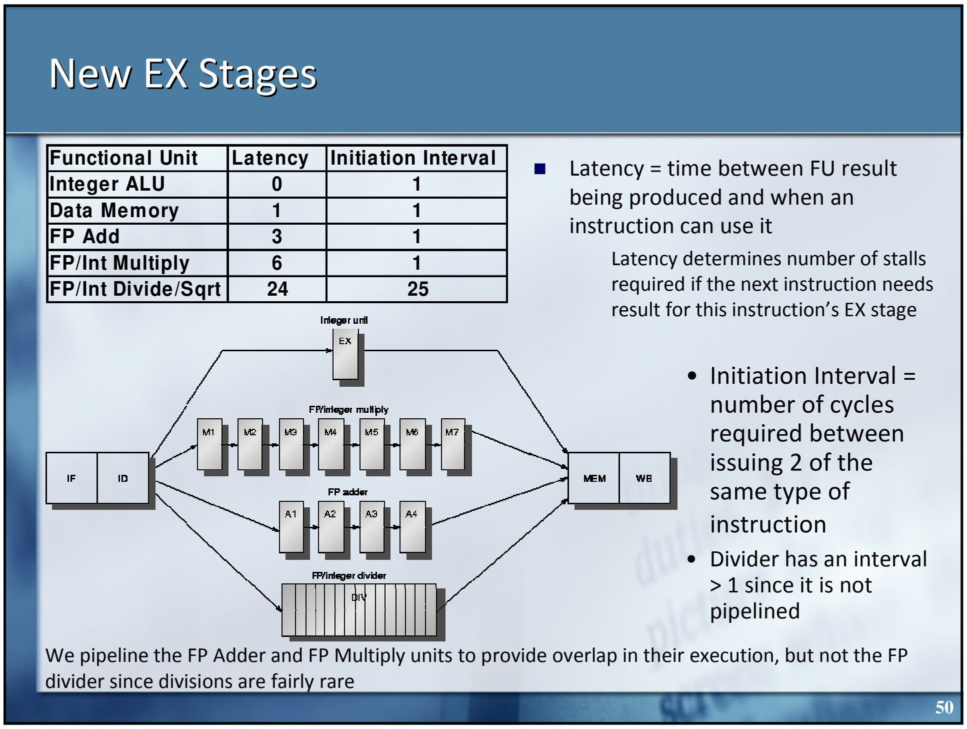
New EX stages

- EX Integer Unit
  - same as before, handles most Integer ALU operations
  - computes effective address (load/store, branch)
    - Instruction moves through this stage in 1 cycle
- EX FP/integer multiply
  - perform FP and integer *
- EX FP adder
  - perform FP +, -, conversion
- EX FP/integer divider
  - perform FP and int /

EX
Integer unit
EX
FP/integer multiply
IF
ID
MEM
WB
EX
FP adder
EX
FP/integer divider

The FP ADD unit takes 4 cycles
The FP Mult unit takes 7 cycles
The FP Div unit takes 25 cycles

We can accommodate several operations in the EX stage at the same time



New EX Stages

Functional Unit	Latency	Initiation Interval
Integer ALU	0	1
Data Memory	1	1
FP Add	3	1
FP/Int Multiply	6	1
FP/Int Divide/Sqrt	24	25

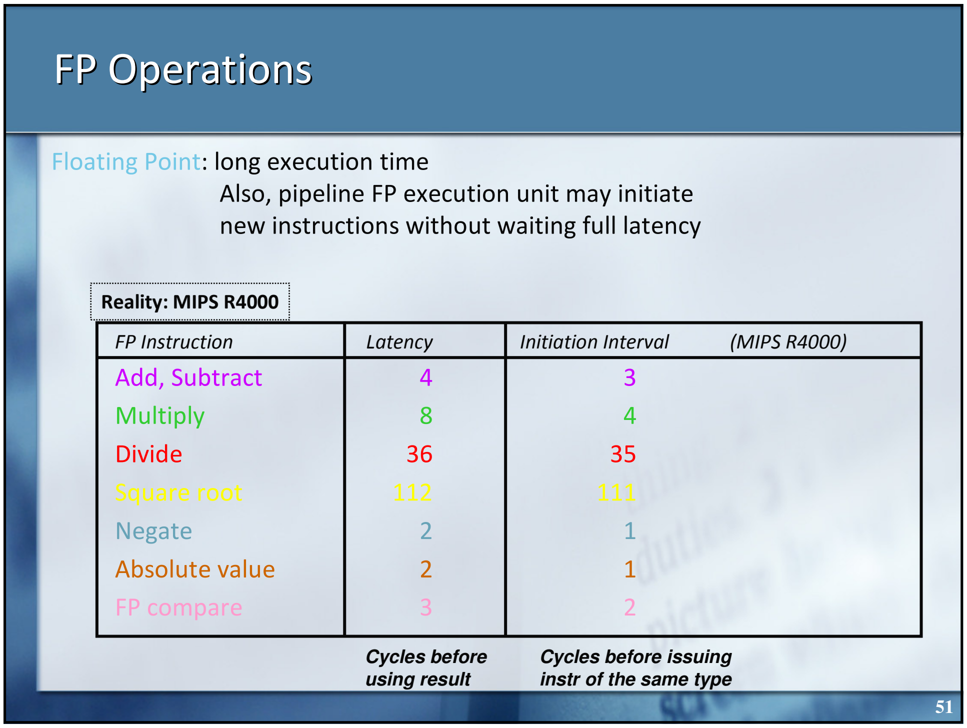
- Latency = time between FU result being produced and when an instruction can use it

  Latency determines number of stalls required if the next instruction needs result for this instruction's EX stage

Integer unit
EX
FP/integer multiply
M1
M2
M3
M4
M5
M6
M7
IF
ID
MEM
WB
FP adder
A1
A2
A3
A4
FP/integer divider
DIV

- Initiation Interval = number of cycles required between issuing 2 of the same type of instruction
- Divider has an interval > 1 since it is not pipelined

We pipeline the FP Adder and FP Multiply units to provide overlap in their execution, but not the FP divider since divisions are fairly rare



FP Operations

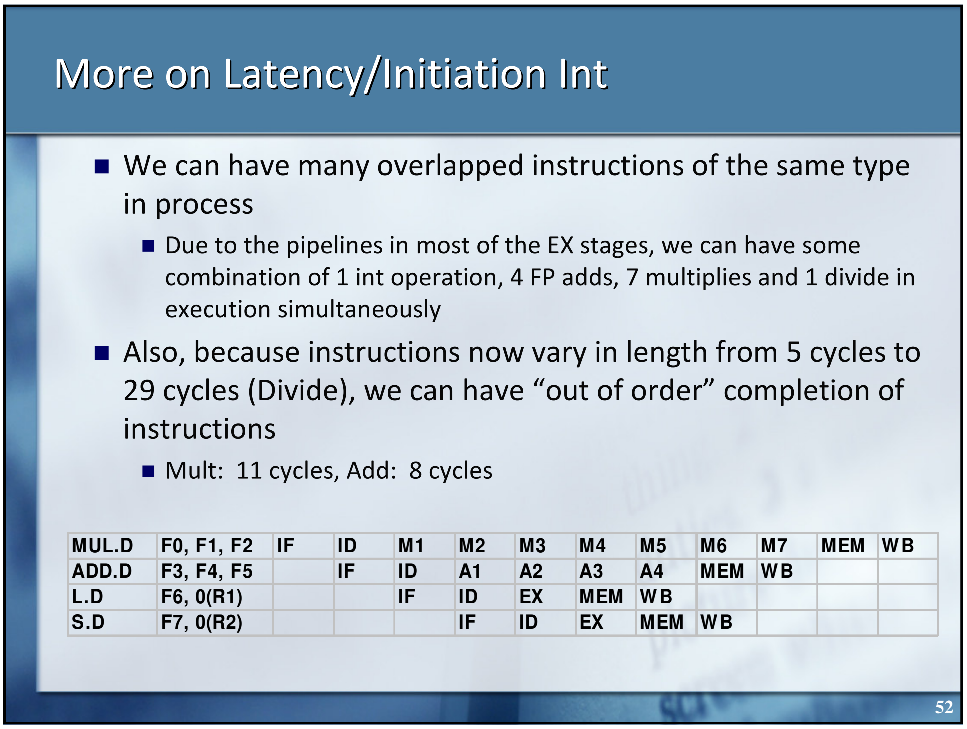
Floating Point: long execution time
Also, pipeline FP execution unit may initiate
new instructions without waiting full latency

Reality: MIPS R4000

FP Instruction	*Latency*	*Initiation Interval (MIPS R4000)*
Add, Subtract	4	3
Multiply	8	4
Divide	36	35
Square root	112	111
Negate	2	1
Absolute value	2	1
FP compare	3	2

Cycles before using result

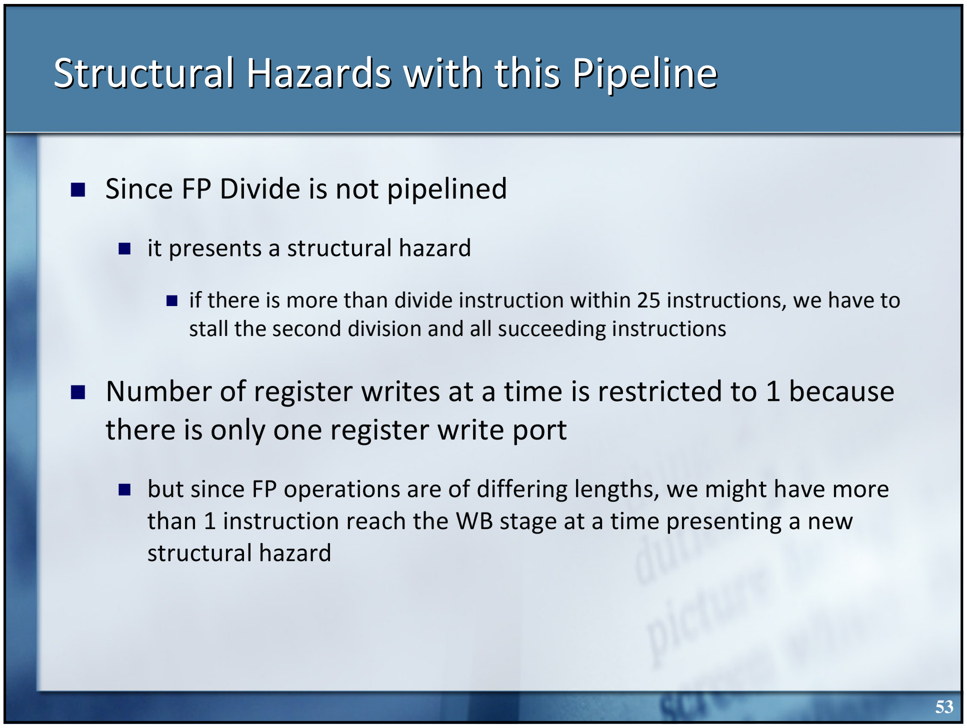
Cycles before issuing instr of the same type



More on Latency/Initiation Int

- We can have many overlapped instructions of the same type in process
  - Due to the pipelines in most of the EX stages, we can have some combination of 1 int operation, 4 FP adds, 7 multiplies and 1 divide in execution simultaneously
- Also, because instructions now vary in length from 5 cycles to 29 cycles (Divide), we can have "out of order" completion of instructions
  - Mult: 11 cycles, Add: 8 cycles

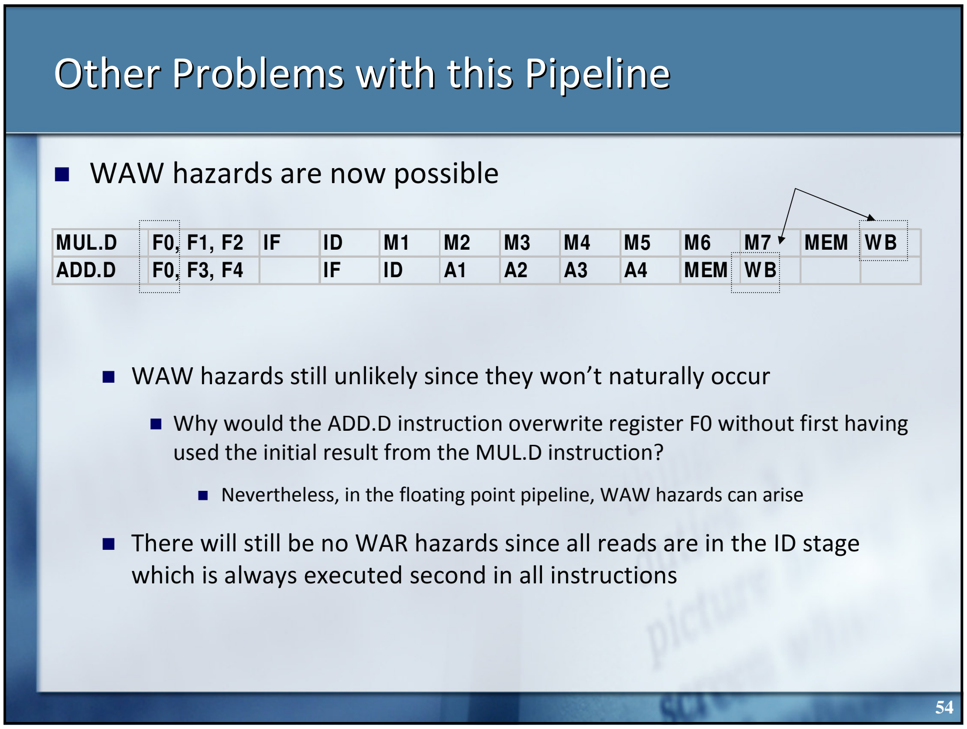
MUL.D	**F0, F1, F2**	**IF**	**ID**	**M1**	**M2**	**M3**	**M4**	**M5**	**M6**	**M7**	**MEM**	**WB**
ADD.D	**F3, F4, F5**		**IF**	**ID**	**A1**	**A2**	**A3**	**A4**	**MEM**	**WB**		
L.D	**F6, 0(R1)**			**IF**	**ID**	**EX**	**MEM**	**WB**				
S.D	**F7, 0(R2)**				**IF**	**ID**	**EX**	**MEM**	**WB**			



Structural Hazards with this Pipeline

- Since FP Divide is not pipelined
  - it presents a structural hazard
    - if there is more than divide instruction within 25 instructions, we have to stall the second division and all succeeding instructions
- Number of register writes at a time is restricted to 1 because there is only one register write port
  - but since FP operations are of differing lengths, we might have more than 1 instruction reach the WB stage at a time presenting a new structural hazard

53



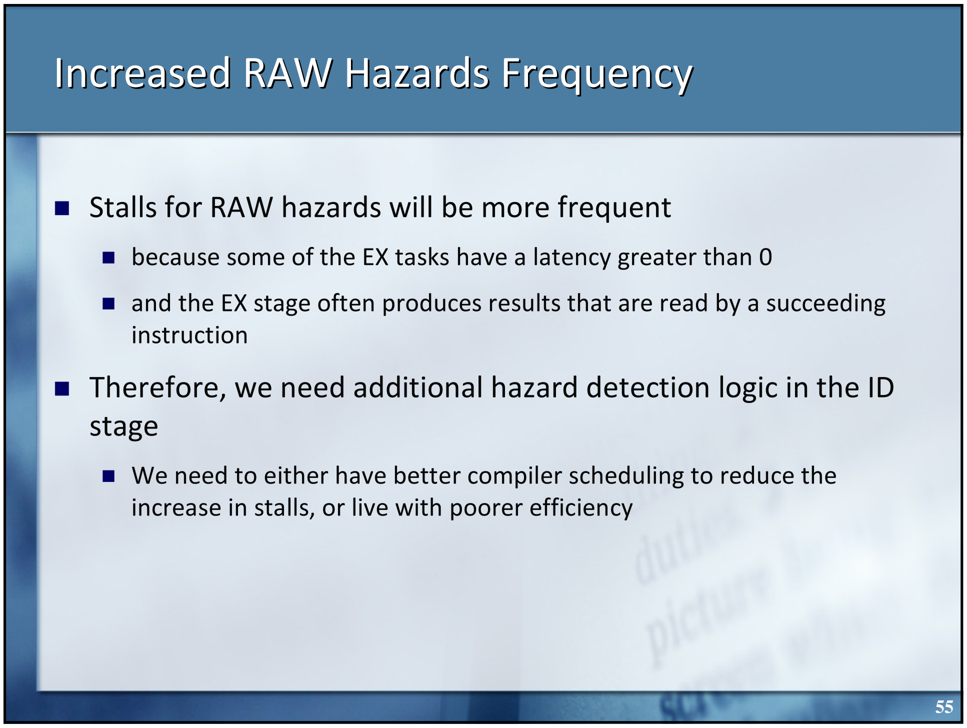
Other Problems with this Pipeline

- WAW hazards are now possible

MUL.D	F0, F1, F2	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
ADD.D	F0, F3, F4		IF	ID	A1	A2	A3	A4	MEM	WB		

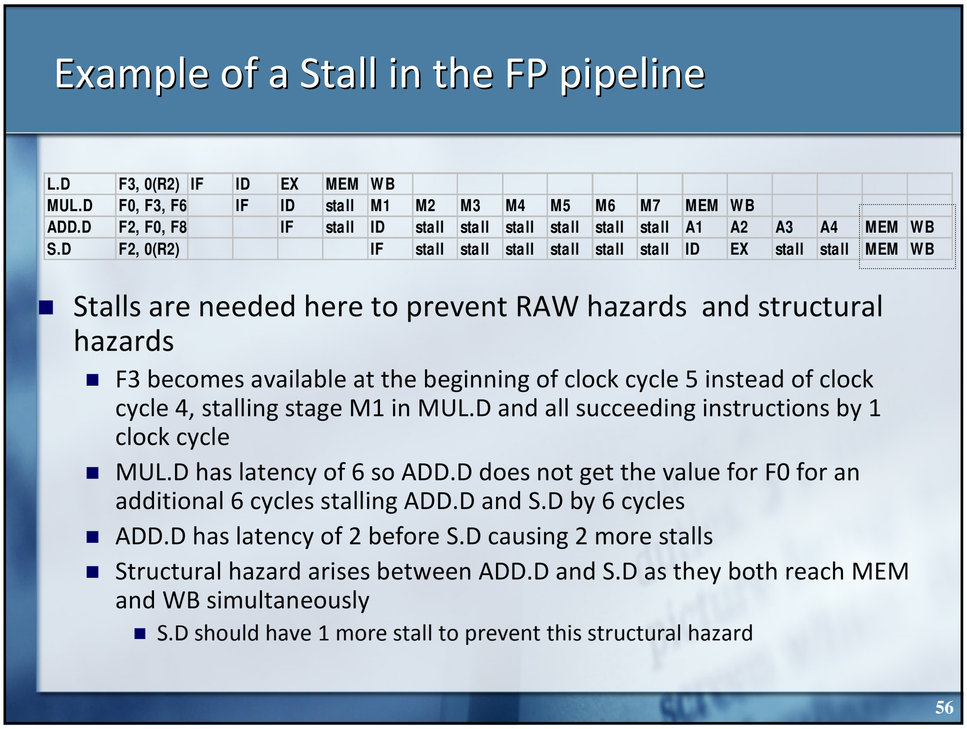
  - WAW hazards still unlikely since they won't naturally occur
    - Why would the ADD.D instruction overwrite register F0 without first having used the initial result from the MUL.D instruction?
      - Nevertheless, in the floating point pipeline, WAW hazards can arise
  - There will still be no WAR hazards since all reads are in the ID stage which is always executed second in all instructions

54



Increased RAW Hazards Frequency

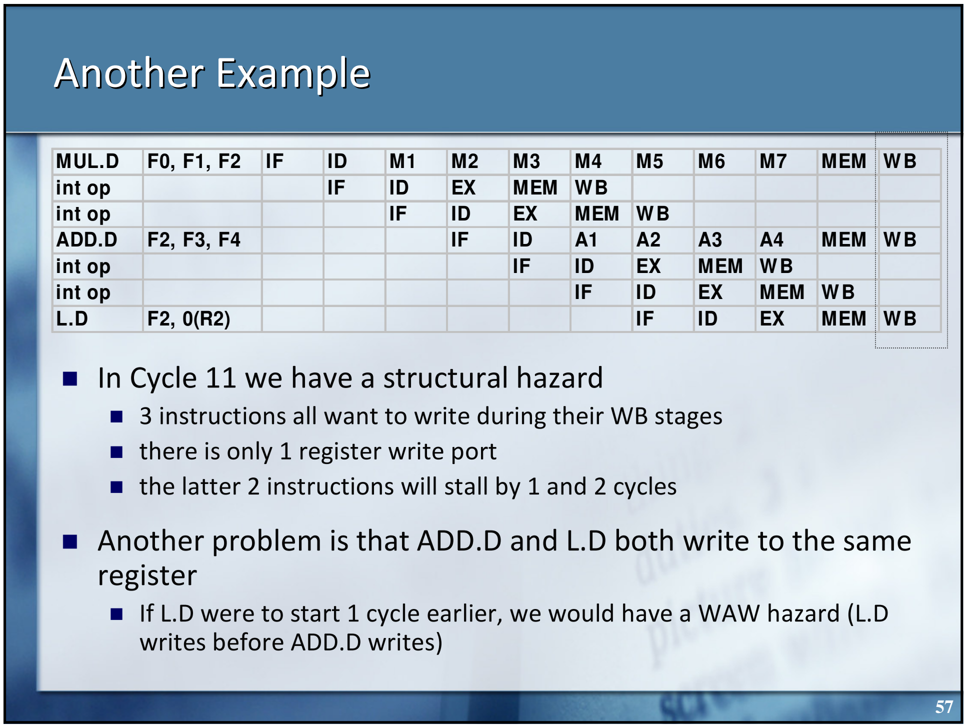
- Stalls for RAW hazards will be more frequent
  - because some of the EX tasks have a latency greater than 0
  - and the EX stage often produces results that are read by a succeeding instruction
- Therefore, we need additional hazard detection logic in the ID stage
  - We need to either have better compiler scheduling to reduce the increase in stalls, or live with poorer efficiency



Example of a Stall in the FP pipeline

L.D	F3, 0(R2)	IF	ID	EX	MEM	WB												
MUL.D	F0, F3, F6		IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADD.D	F2, F0, F8			IF	stall	ID	stall	stall	stall	stall	stall	stall	A1	A2	A3	A4	MEM	WB
S.D	F2, 0(R2)					IF	stall	stall	stall	stall	stall	stall	ID	EX	stall	stall	MEM	WB

- Stalls are needed here to prevent RAW hazards and structural hazards
  - F3 becomes available at the beginning of clock cycle 5 instead of clock cycle 4, stalling stage M1 in MUL.D and all succeeding instructions by 1 clock cycle
  - MUL.D has latency of 6 so ADD.D does not get the value for F0 for an additional 6 cycles stalling ADD.D and S.D by 6 cycles
  - ADD.D has latency of 2 before S.D causing 2 more stalls
  - Structural hazard arises between ADD.D and S.D as they both reach MEM and WB simultaneously
    - S.D should have 1 more stall to prevent this structural hazard

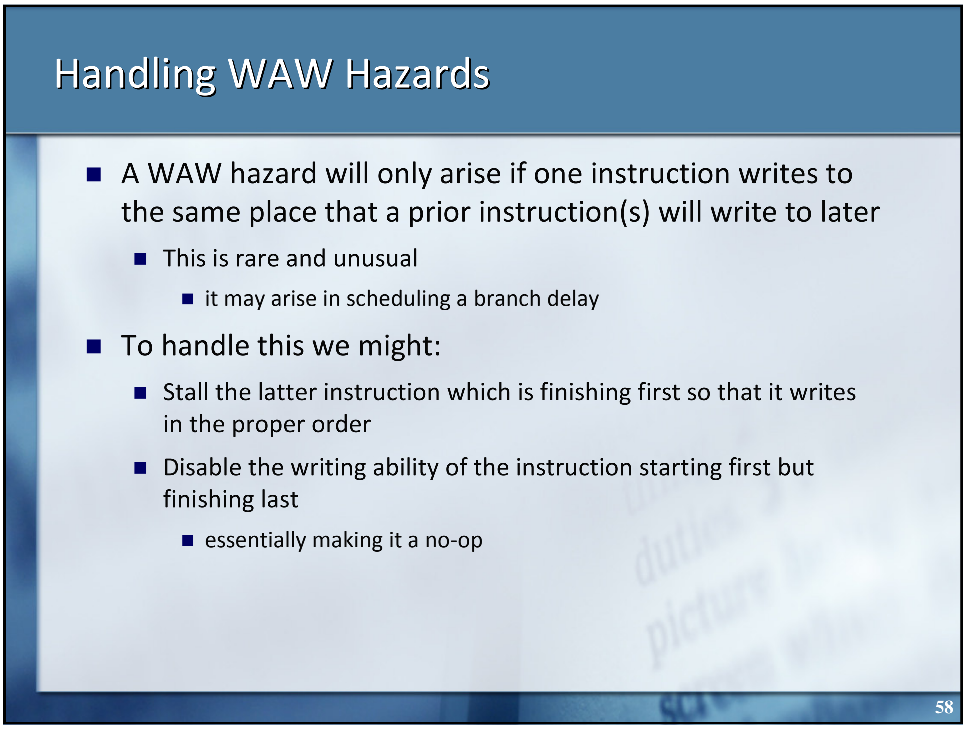


Another Example

MUL.D	F0, F1, F2	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
int op			IF	ID	EX	MEM	WB					
int op				IF	ID	EX	MEM	WB				
ADD.D	F2, F3, F4				IF	ID	A1	A2	A3	A4	MEM	WB
int op						IF	ID	EX	MEM	WB		
int op							IF	ID	EX	MEM	WB	
L.D	F2, 0(R2)							IF	ID	EX	MEM	WB

- In Cycle 11 we have a structural hazard
  - 3 instructions all want to write during their WB stages
  - there is only 1 register write port
  - the latter 2 instructions will stall by 1 and 2 cycles
- Another problem is that ADD.D and L.D both write to the same register
  - If L.D were to start 1 cycle earlier, we would have a WAW hazard (L.D writes before ADD.D writes)

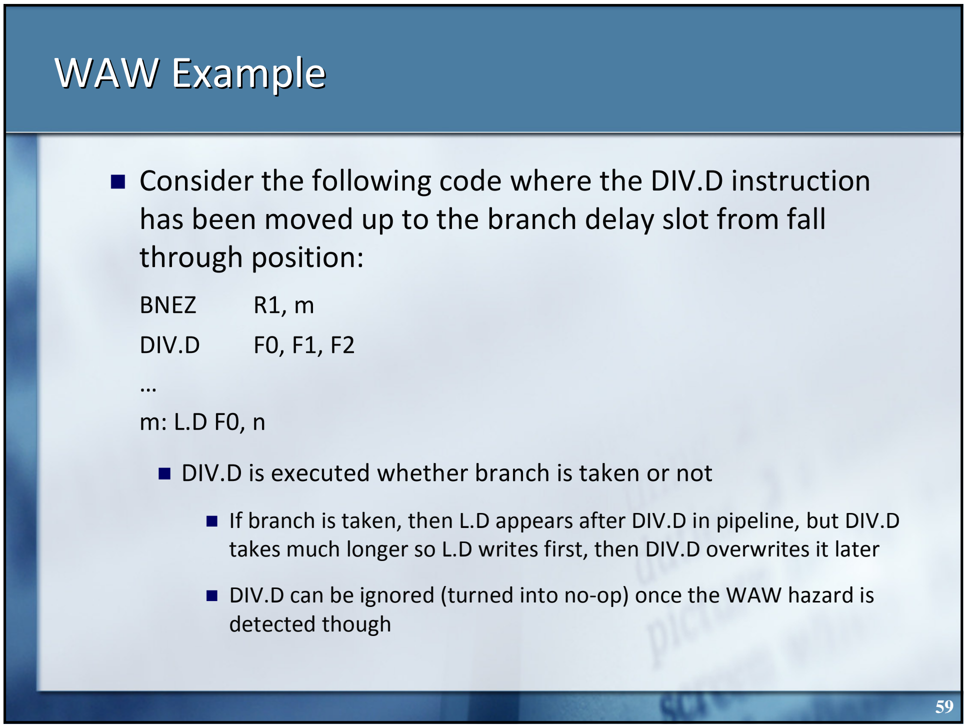
57



Handling WAW Hazards

- A WAW hazard will only arise if one instruction writes to the same place that a prior instruction(s) will write to later
  - This is rare and unusual
    - it may arise in scheduling a branch delay
- To handle this we might:
  - Stall the latter instruction which is finishing first so that it writes in the proper order
  - Disable the writing ability of the instruction starting first but finishing last
    - essentially making it a no-op

58

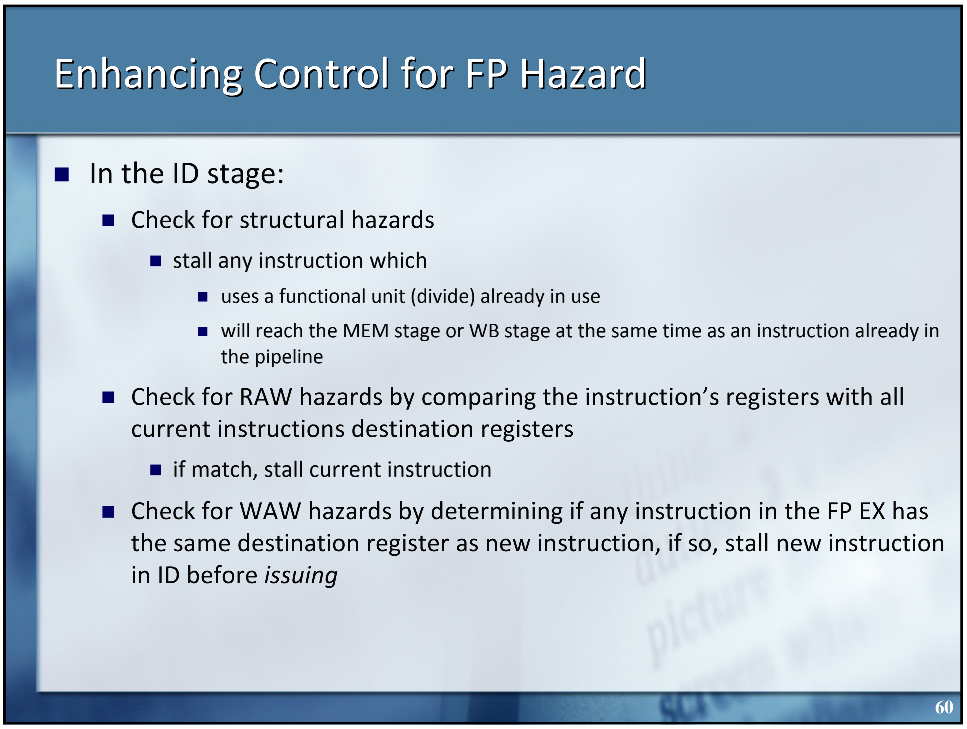


WAW Example

- Consider the following code where the DIV.D instruction has been moved up to the branch delay slot from fall through position:

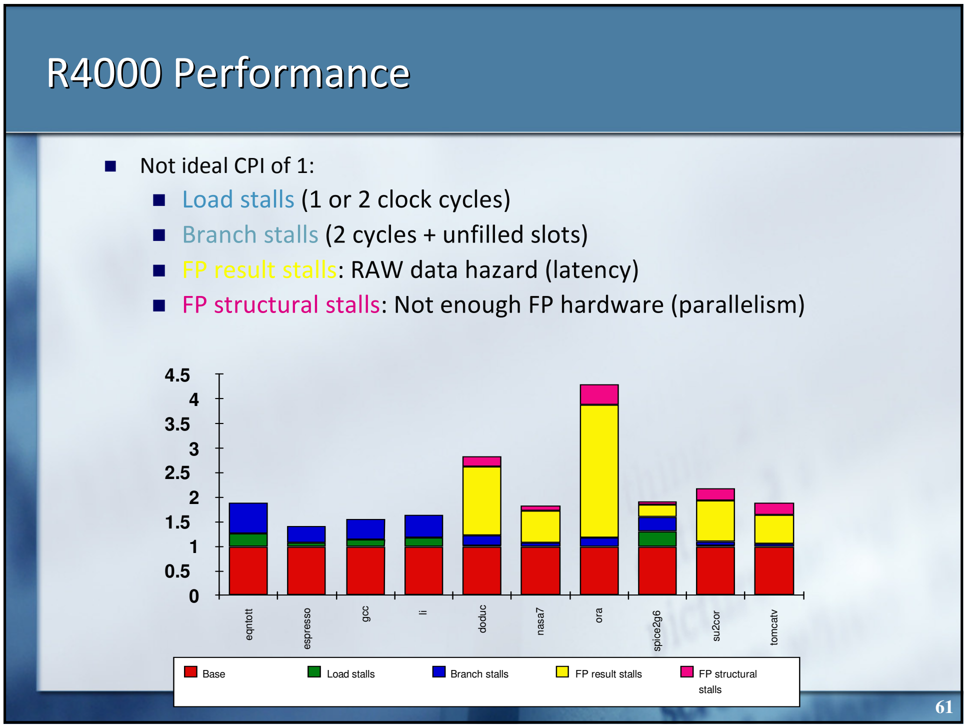
```
BNEZ      R1, m
DIV.D     F0, F1, F2
…
m: L.D F0, n
```

  - DIV.D is executed whether branch is taken or not
    - If branch is taken, then L.D appears after DIV.D in pipeline, but DIV.D takes much longer so L.D writes first, then DIV.D overwrites it later
    - DIV.D can be ignored (turned into no-op) once the WAW hazard is detected though



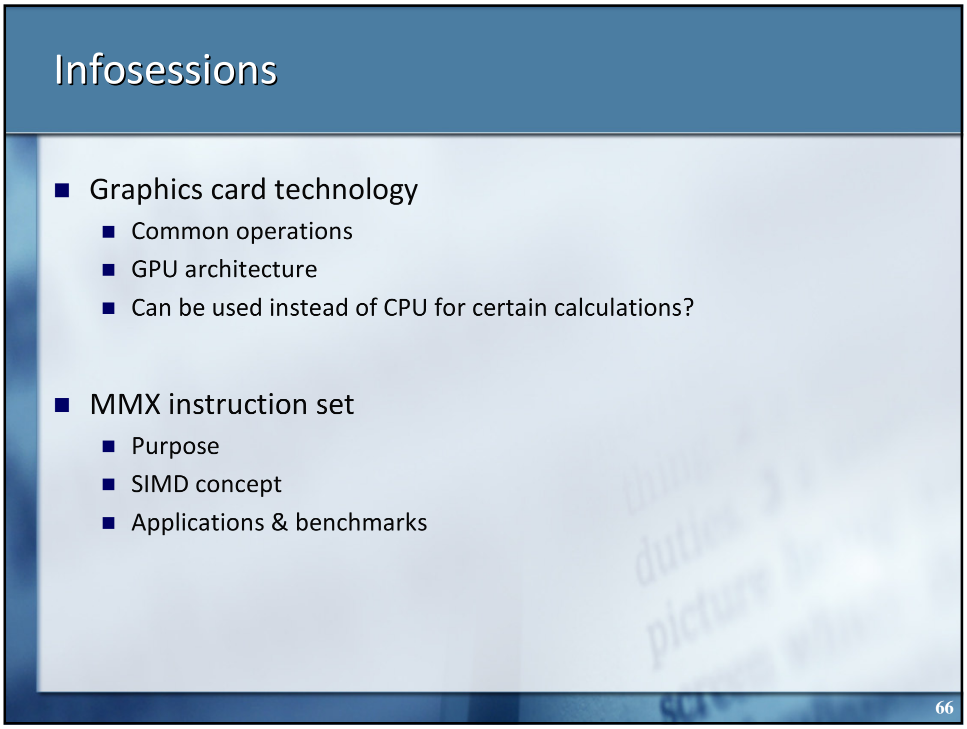
Enhancing Control for FP Hazard

- In the ID stage:
  - Check for structural hazards
    - stall any instruction which
      - uses a functional unit (divide) already in use
      - will reach the MEM stage or WB stage at the same time as an instruction already in the pipeline
  - Check for RAW hazards by comparing the instruction’s registers with all current instructions destination registers
    - if match, stall current instruction
  - Check for WAW hazards by determining if any instruction in the FP EX has the same destination register as new instruction, if so, stall new instruction in ID before *issuing*



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)



Infosessions

- Graphics card technology
  - Common operations
  - GPU architecture
  - Can be used instead of CPU for certain calculations?
- MMX instruction set
  - Purpose
  - SIMD concept
  - Applications & benchmarks