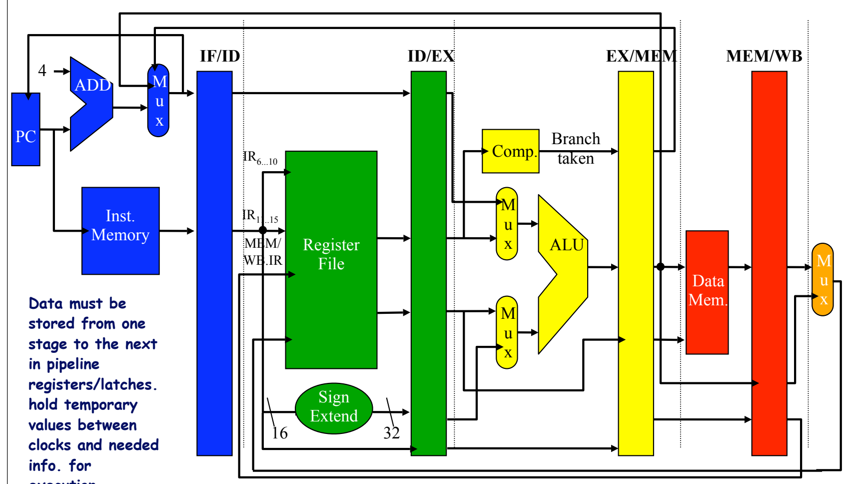
Lecture 20-21
Pipelining Hazards and Examples

Review

The "new look" dataflow

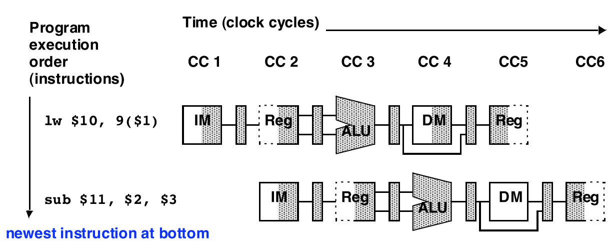


Data must be stored from one stage to the next in pipeline registers/latches. hold temporary values between clocks and needed info. for execution.

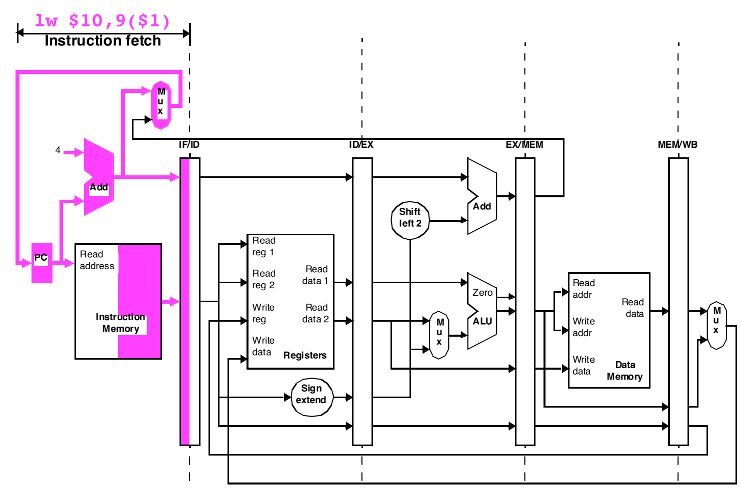
A quick reminder

A 2 instruction sequence

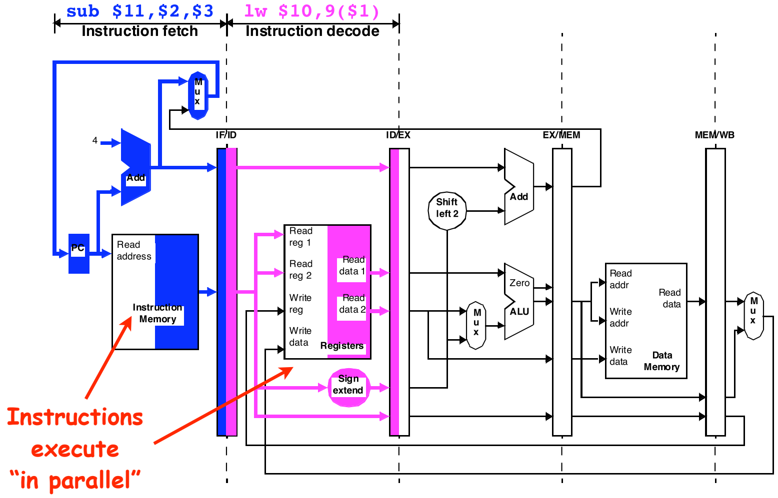
- Examine multiple-cycle & single-cycle diagrams for a sequence of 2 independent instructions
  - (i.e. no common registers b/t them)
    - lw $10, 9($1)
    - sub $11, $2, $3



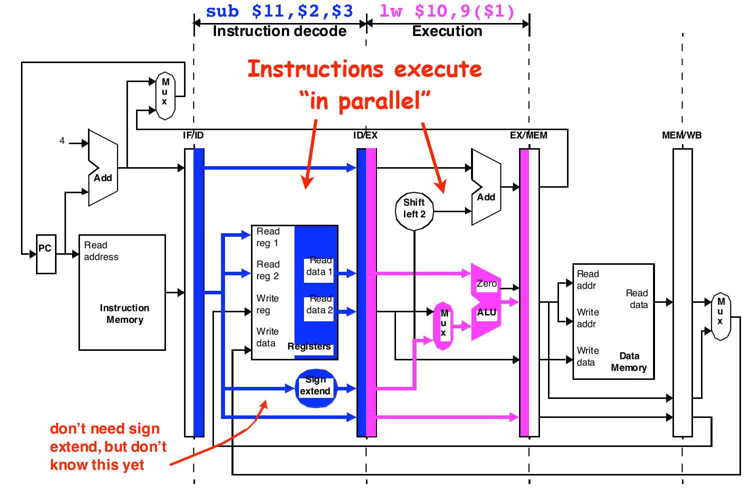
Single-cycle diagrams: cycle 1



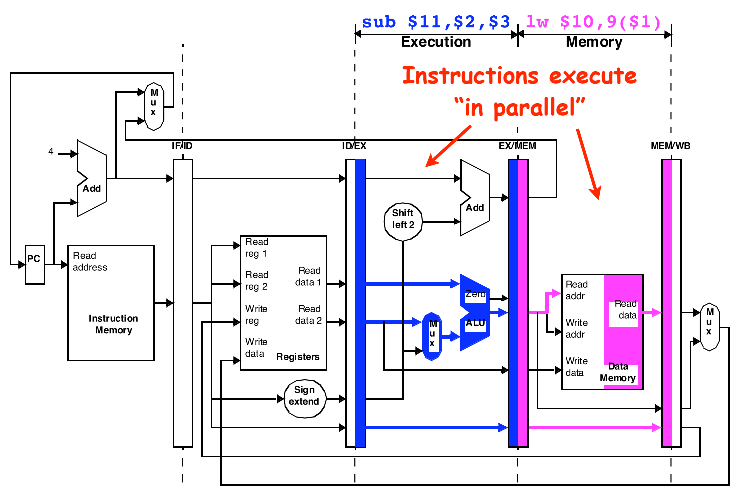
Single-cycle diagrams: cycle 2



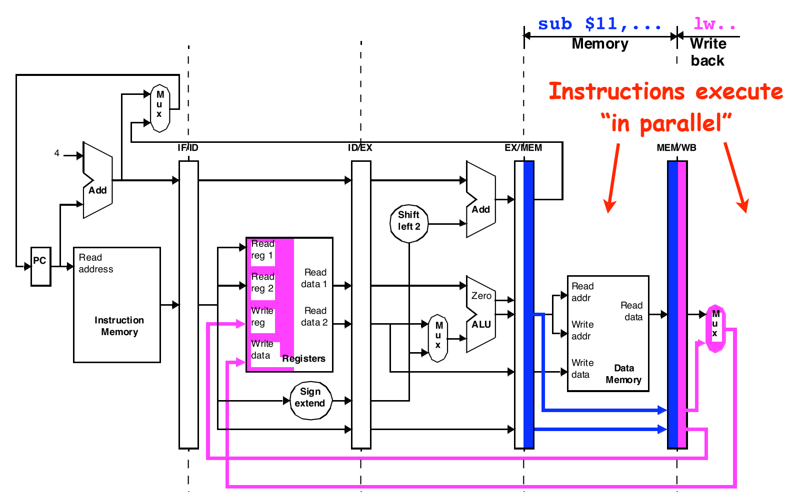
Single-cycle diagrams: cycle 3



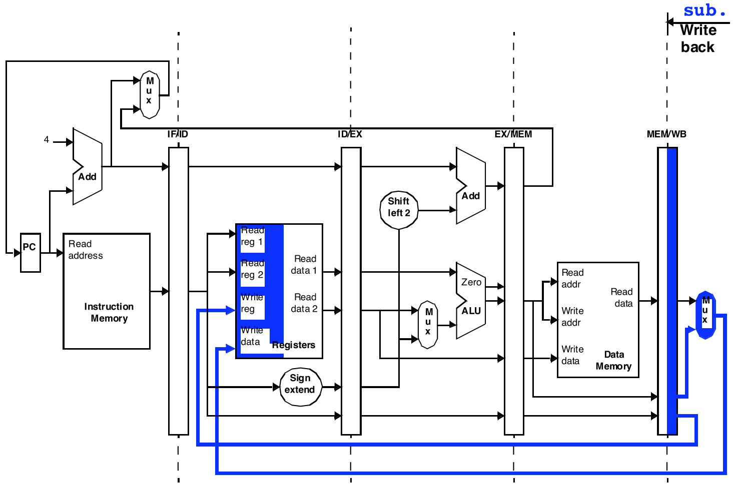
Single-cycle diagrams: cycle 4



Single-cycle diagrams: cycle 5



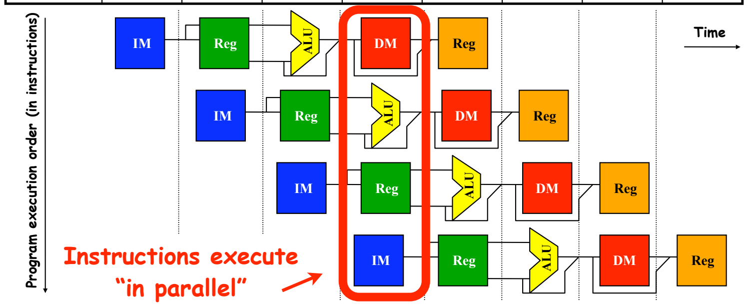
Single-cycle diagrams: cycle 6



Another way to look at it…

Clock Number

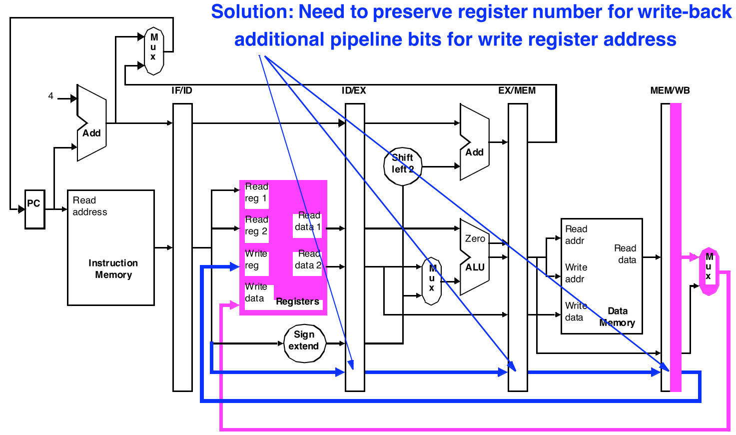
Inst. #	1	2	3	4	5	6	7	8
Inst. i	IF	ID	EX	MEM	WB			
Inst. i+1		IF	ID	EX	MEM	WB		
Inst. i+2			IF	ID	EX	MEM	WB	
Inst. i+3				IF	ID	EX	MEM	WB



Pipelining does come with extra overhead

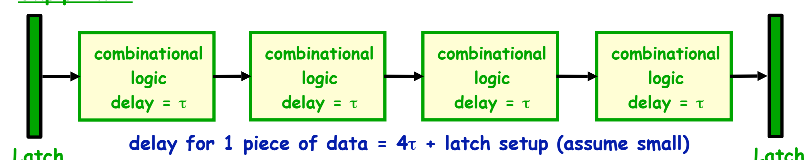
Bug: source for Write Reg is invalid

Solution: Need to preserve register number for write-back additional pipeline bits for write register address



Need to factor in latch overhead too

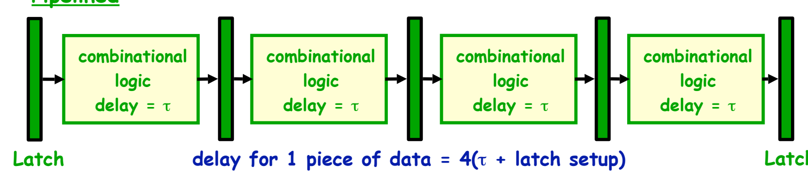
Unpipelined



delay for 1 piece of data = 4τ + latch setup (assume small)

approximate delay for 1000 pieces of data = 4000τ

Pipelined



delay for 1 piece of data = 4(τ + latch setup)

approximate delay for 1000 pieces of data = 3τ + 1000τ

speedup for 1000 pieces of data = $\frac{4000}{1003}$ = ~ 4

Ideal speedup = # of pipeline stages

quick CPU time calculation

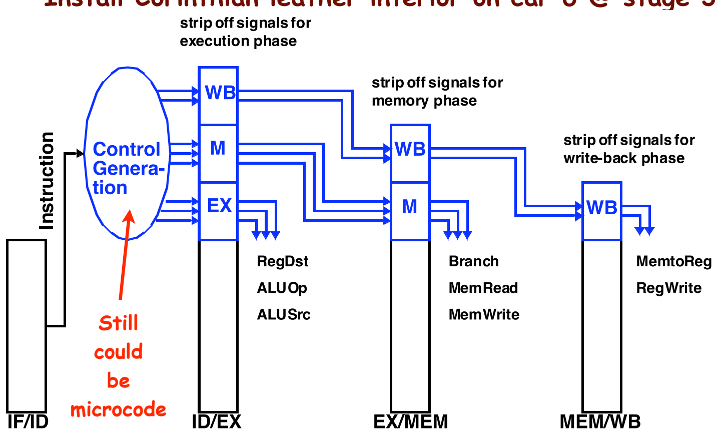
What about control signals?

Questions about control signals

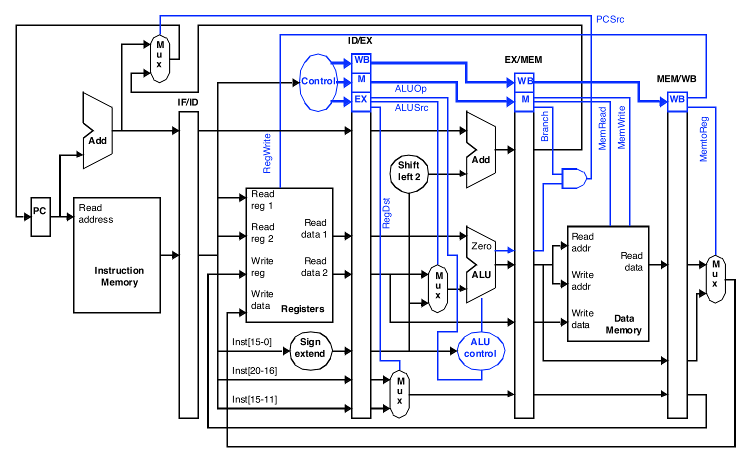
- Following discussion relevant to a single instruction
- Q: *Are all control signals active at the same time?*
- A: ?
- Q: *Can we generate all these signals at the same time?*
- A: ?

Passing control w/pipe registers

- *Analogy:* send instruction with car on assembly line
  - "Install Corinthian leather interior on car 6 @ stage 3"



Pipelined datapath w/control signals



Hazards
(Let's start on the chalkboard)

The hazards of pipelining

- Pipeline hazards prevent next instruction from executing during designated clock cycle
- There are 3 classes of hazards:
  - Structural Hazards:
    - Arise from resource conflicts
    - HW cannot support all possible combinations of instructions
  - Data Hazards:
    - Occur when given instruction depends on data from an instruction ahead of it in pipeline
  - Control Hazards:
    - Result from branch, other instructions that change flow of program (i.e. change PC)

How do we deal with hazards?

- Often, pipeline must be stalled
- Stalling pipeline usually lets some instruction(s) in pipeline proceed, another/others wait for data, resource, etc.
- A note on terminology:
  - If we say an instruction was "issued later than instruction x", we mean that it was issued after instruction x and is not as far along in the pipeline
  - If we say an instruction was "issued earlier than instruction x", we mean that it was issued before instruction x and is further along in the pipeline

Stalls and performance

- Stalls impede progress of a pipeline and result in deviation from 1 instruction executing/clock cycle
- Pipelining can be viewed to:
  - Decrease CPI or clock cycle time for instruction
  - Let's see what affect stalls have on CPI…
- CPI pipelined =
  - Ideal CPI + Pipeline stall cycles per instruction
  - 1 + Pipeline stall cycles per instruction
- Ignoring overhead and assuming stages are balanced:

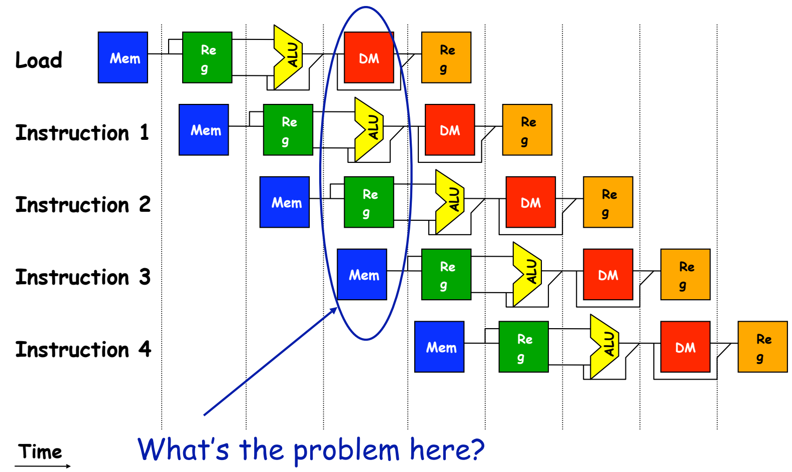
$$Speedup = \frac{CPI\ unpipelined}{1 + pipeline\ stall\ cycles\ per\ instruction}$$

(Recall combinational logic slide)

Structural hazards

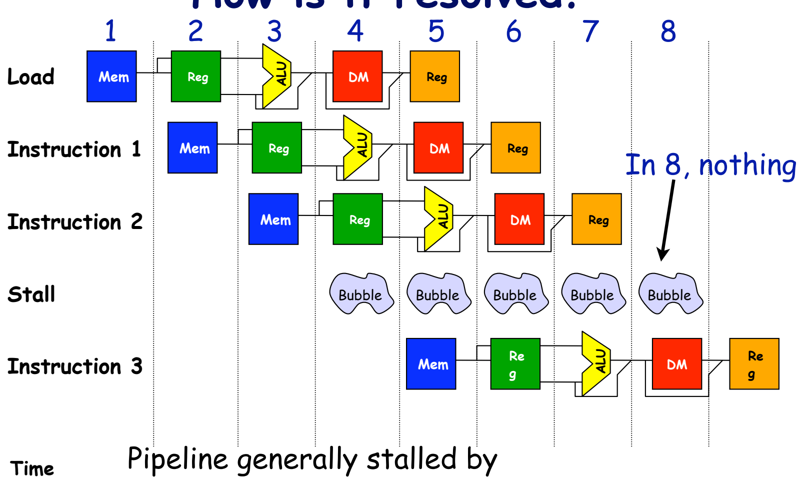
- 1 way to avoid structural hazards is to duplicate resources
  - i.e.: An ALU to perform an arithmetic operation and an adder to increment PC
- If not all possible combinations of instructions can be executed, structural hazards occur
- Most common instances of structural hazards:
  - When a functional unit not fully pipelined
  - When some resource not duplicated enough
- Pipelines stall result of hazards, CPI increased from the usual "1"

An example of a structural hazard



What's the problem here?

How is it resolved?



Pipeline generally stalled by inserting a "bubble" or NOP

Or alternatively…

Clock Number

Inst. #	1	2	3	4	5	6	7	8	9	10
LOAD	IF	ID	EX	MEM	WB					
Inst. i+1		IF	ID	EX	MEM	WB				
Inst. i+2			IF	ID	EX	MEM	WB			
Inst. i+3				stall	IF	ID	EX	MEM	WB	
Inst. i+4						IF	ID	EX	MEM	WB
Inst. i+5							IF	ID	EX	MEM
Inst. i+6								IF	ID	EX

- LOAD instruction "steals" an instruction fetch cycle which will cause the pipeline to stall.
- Thus, no instruction completes on clock cycle 8

On the board…

- A simple example...

Remember the common case!

- All things being equal, a machine without structural hazards will always have a lower CPI.
- But, in some cases it may be better to allow them than to eliminate them.
- These are situations a computer architect might have to consider:
  - Is pipelining functional units or duplicating them costly in terms of HW?
  - Does structural hazard occur often?
  - What's the common case???

What's the realistic solution?

- Answer: Add more hardware.
  - As we'll see, CPI degrades quickly from our ideal '1' for even the simplest of cases…

Data hazards

- These exist because of pipelining
- Why do they exist???
  - Pipelining changes order or read/write accesses to operands
  - Order differs from order seen by sequentially executing instructions on unpipelined machine
- Consider this example:
  - ADD R1, R2, R3
  - SUB R4, R1, R5
  - AND R6, R1, R7
  - OR R8, R1, R9
  - XOR R10, R1, R11

All instructions after ADD use result of ADD

ADD writes the register in WB but SUB needs it in ID.

<u>This is a data hazard</u>

Illustrating a data hazard

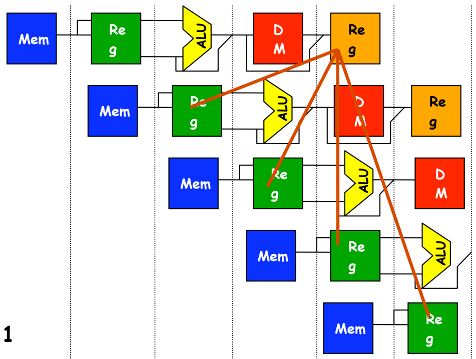
ADD R1, R2, R3

SUB R4, R1, R5

AND R6, R1, R7

OR R8, R1, R9

XOR R10, R1, R11



Time

ADD instruction causes a hazard in next 3 instructions b/c register not written until after those 3 read it.

Forwarding

- Problem illustrated on previous slide can actually be solved relatively easily – with forwarding
- In this example, result of the ADD instruction not really needed until after ADD actually produces it
- Can we move the result from EX/MEM register to the beginning of ALU (where SUB needs it)?
  - Yes! Hence this slide!
- Generally speaking:
  - Forwarding occurs when a result is passed directly to functional unit that requires it.
  - Result goes from output of one unit to input of another

When can we forward?

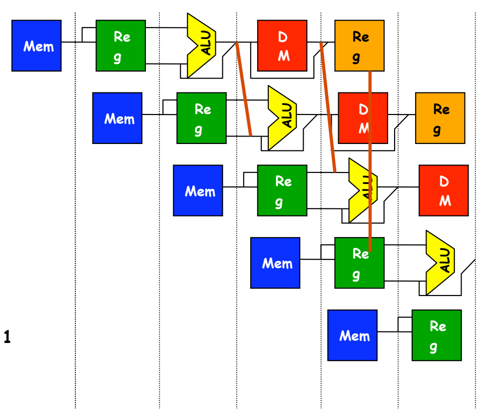
ADD R1, R2, R3

SUB R4, R1, R5

AND R6, R1, R7

OR R8, R1, R9

XOR R10, R1, R11



SUB gets info. from EX/MEM pipe register

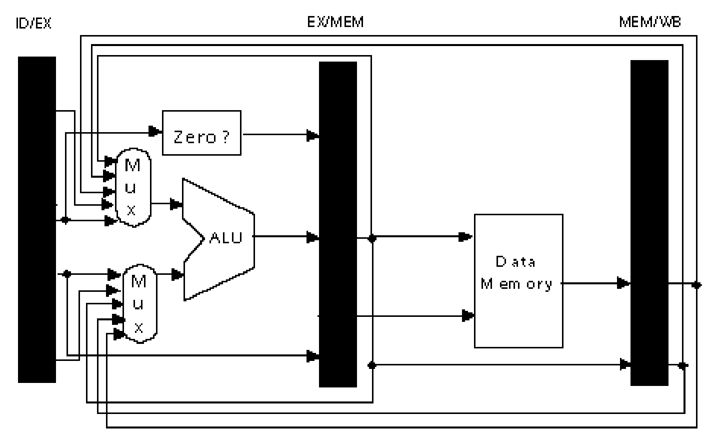
AND gets info. from MEM/WB pipe register

OR gets info. by forwarding from register file

Time

Rule of thumb: If line goes "forward" you can do forwarding. If its drawn backward, it's physically impossible.

HW Change for Forwarding

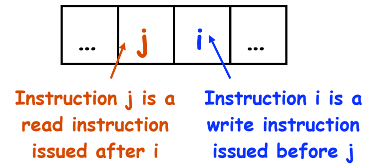


Data hazard specifics

- There are actually 3 different kinds of data hazards!
  - Read After Write (RAW)
  - Write After Write (WAW)
  - Write After Read (WAR)
- We'll discuss/illustrate each on forthcoming slides. However, 1st a note on convention.
  - Discussion of hazards will use generic instructions i & j.
  - i is always issued before j.
  - Thus, i will always be further along in pipeline than j.
- With an in-order issue/in-order completion machine, we're not as concerned with WAW, WAR

Read after write (RAW) hazards

- With RAW hazard, instruction j tries to read a source operand before instruction i writes it.
- Thus, j would incorrectly receive an old or incorrect value
- Graphically/Example:



i: ADD R1, R2, R3
j: SUB R4, R1, R6

- Can use stalling or forwarding to resolve this hazard

Memory Data Hazards

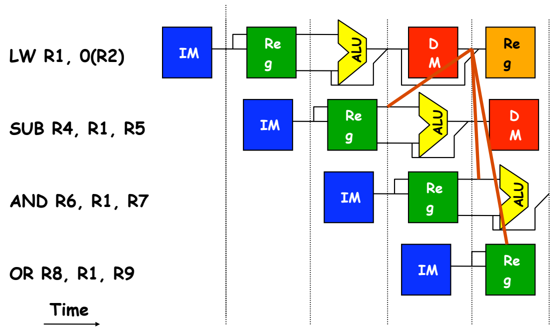
- Seen register hazards, can also have memory hazards
  - RAW:
    - store R1, 0(SP)
    - load R4, 0(SP)

	1	2	3	4	5	6
Store R1, 0(SP)	F	D	EX	M	WB	
Load R1, 0(SP)		F	D	EX	M	WB

  - In simple pipeline, memory hazards are easy
    - In order, one at a time, read & write in same stage
  - In general though, more difficult than register hazards

	1	2	3	4	5	6
Store R1, 0(SP)	F	D	EX	M	WB	
Load R1, 0(SP)		F	D	EX	M	WB

Forwarding: It doesn't always work

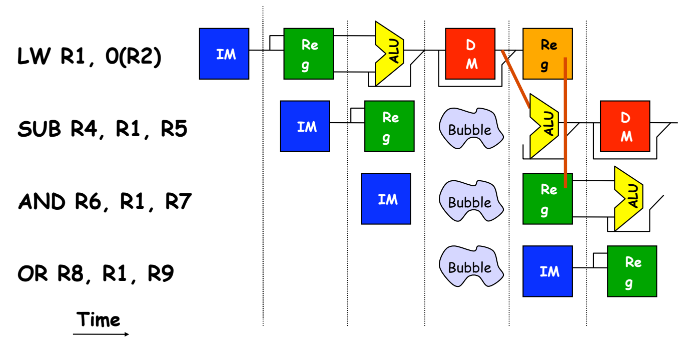


Load has a latency that forwarding can't solve.

Pipeline must stall until hazard cleared (starting with instruction that wants to use data until source produces it).

Can't get data to subtract instruction unless...

The solution pictorially



Insertion of bubble causes # of cycles to complete this sequence to grow by 1

Data hazards and the compiler

- Compiler should be able to help eliminate some stalls caused by data hazards
- i.e. compiler could not generate a LOAD instruction that is immediately followed by instruction that uses result of LOAD's destination register.
- Technique is called "pipeline/instruction scheduling"

What about control logic?

- For MIPS integer pipeline, all data hazards can be checked during ID phase of pipeline
- If data hazard, instruction stalled before its issued
- Whether forwarding is needed can also be determined at this stage, controls signals set
- If hazard detected, control unit of pipeline must stall pipeline and prevent instructions in IF, ID from advancing
- All control information carried along in pipeline registers so only these fields must be changed

Some example situations

Situation	Example	Action

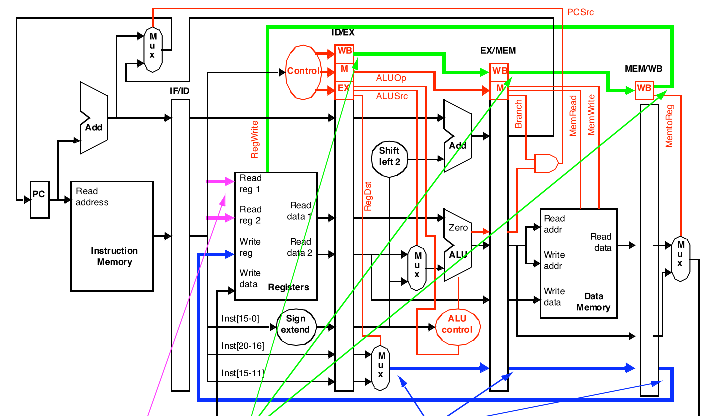
No Dependence	LW R1, 45(R2)
	ADD R5, R6, R7
	SUB R8, R6, R7
	OR R9, R6, R7
		No hazard possible because no dependence exists on R1 in the immediately following three instructions.

Dependence requiring stall	LW R1, 45(R2)
	ADD R5, R1, R7
	SUB R8, R6, R7
	OR R9, R6, R7
		Comparators detect the use of R1 in the ADD and stall the ADD (and SUB and OR) before the ADD begins EX

Dependence overcome by forwarding	LW R1, 45(R2)
	ADD R5, R6, R7
	SUB R8, R1, R7
	OR R9, R6, R7
		Comparators detect the use of R1 in SUB and forward the result of LOAD to the ALU in time for SUB to begin with EX

Dependence with accesses in order	LW R1, 45(R2)
	ADD R5, R6, R7
	SUB R8, R6, R7
	OR R9, R1, R7
		No action is required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half.

Detecting Data Hazards



Hazard **if a current register read address = any register write address in pipeline and RegWrite is asserted in that pipeline stage**

Hazard Detection Logic

- Insert a bubble into pipeline if any are true:
  - ID/EX.RegWrite AND
    - ((ID/EX.RegDst=0 AND ID/EX.WriteRegRt=IF/ID.ReadRegRs) OR
    - (ID/EX.RegDst=1 AND ID/EX.WriteRegRd=IF/ID.ReadRegRs) OR
    - (ID/EX.RegDst=0 AND ID/EX.WriteRegRt=IF/ID.ReadRegRt) OR
    - (ID/EX.RegDst=1 AND ID/EX.WriteRegRd=IF/ID.ReadRegRt))
  - OR EX/MEM AND
    - ((EX/MEM.WriteReg = IF/ID.ReadRegRs) OR
    - (EX/MEM.WriteReg = IF/ID.ReadRegRt))
  - OR MEM/WB.RegWrite AND
    - ((MEM/WB.WriteReg = IF/ID.ReadRegRs) OR
    - (MEM/WB.WriteReg = IF/ID.ReadRegRt))

Pipeline Register → Notation ID/EX.RegDst ← Field

RAW: Detect and Stall

- detect RAW & stall instruction at ID before register read
  - mechanics? disable PC, F/D write
  - RAW detection? compare register names
    - notation: rs1(D) = src register #1 of inst. in D stage
    - compare: rs1(D) & rs2(D) w/ rd(D/X), rd(X/M), rd(M/W)
    - stall (disable PC + F/D, clear D/X) on any match
  - RAW detection? register busy-bits
    - set for rd(D/X) when instruction passes ID
    - clear for rd(M/W)
    - stall if rs1(D) or rs2(D) are "busy"
  - (plus) low cost, simple
  - (minus) low performance (many stalls)

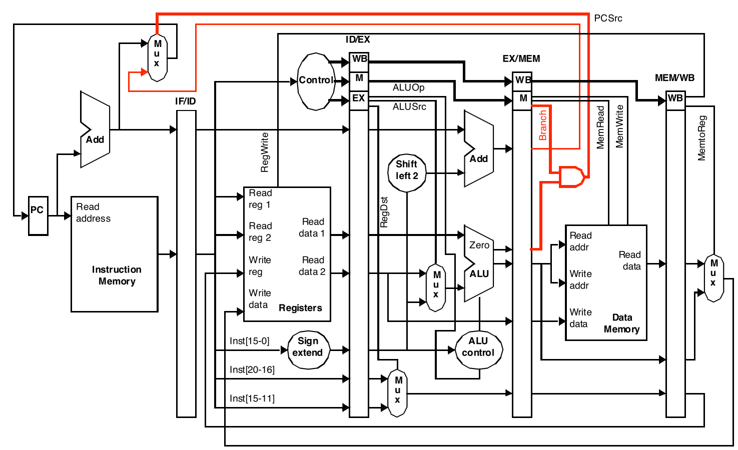
Hazards vs. Dependencies

- dependence: fixed property of instruction stream
  - (i.e., program)
- hazard: property of program and processor organization
  - implies potential for executing things in wrong order
    - potential only exists if instructions can be simultaneously "in-flight"
    - property of dynamic distance between instructions vs. pipeline depth
- For example, can have RAW dependence with or without hazard
  - depends on pipeline

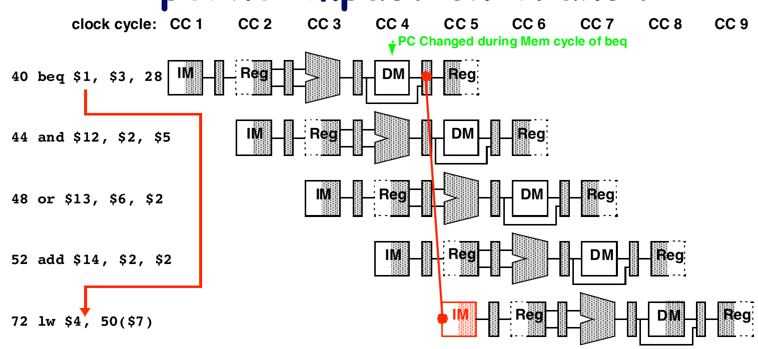
Branch/Control Hazards

- So far, we've limited discussion of hazards to:
  - Arithmetic/logic operations
  - Data transfers
- Also need to consider hazards involving branches:
  - Example:
    - 40: beq $1, $3, $28 # ($28 gives address 72)
    - 44: and $12, $2, $5
    - 48: or $13, $6, $2
    - 52: add $14, $2, $2
    - 72: lw $4, 50($7)
- How long will it take before the branch decision takes effect?
  - What happens in the meantime?

Branch signal determined in MEM stage



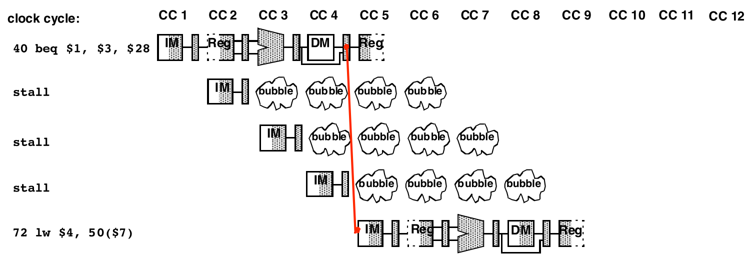
Pipeline impact on branch



- If branch condition true, must skip 44, 48, 52
  - But, these have already started down the pipeline
  - They will complete unless we do something about it
- How do we deal with this?
  - We'll consider 2 possibilities

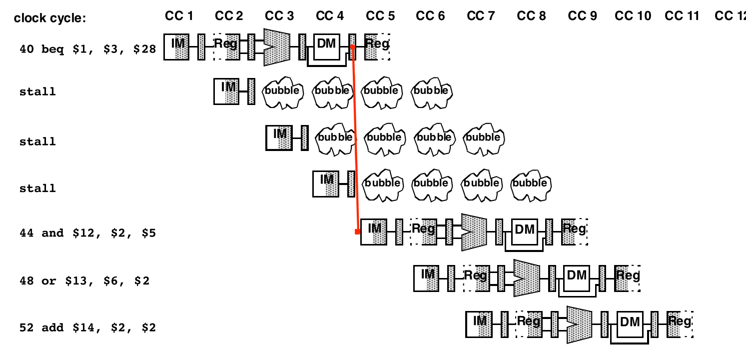
Dealing w/branch hazards: always stall

- Branch taken
  - Wait 3 cycles
  - No proper instructions in the pipeline
  - Same delay as without stalls (no time lost)



Dealing w/branch hazards: always stall

- Branch not taken
  - Still must wait 3 cycles
  - Time lost
  - Could have spent cycles fetching and decoding next instructions



Dealing w/branch hazards: assume branch not taken

- On average, branches are taken ½ the time
  - If branch not taken…
    - Continue normal processing
  - Else, if branch is taken…
    - Need to flush improper instruction from pipeline

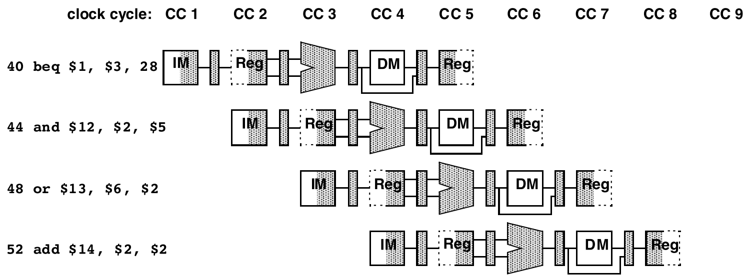
- Cuts overall time for branch processing in ½

Flushing unwanted instructions from pipeline

- Useful to compare w/stalling pipeline:
  - Simple stall: inject bubble into pipe at ID stage only
    - Change control to 0 in the ID stage
    - Let "bubbles" percolate to the right
  - Flushing pipe: must change inst. In IF, ID, and EX
    - IF Stage:
      - Zero instruction field of IF/ID pipeline register
      - Use new control signal IF.Flush
    - ID Stage:
      - Use existing "bubble injection" mux that zeros control for stalls
      - Signal ID.Flush is ORed w/stall signal from hazard detection unit
    - EX Stage:
      - Add new muxes to zero EX pipeline register control lines
      - Both muxes controlled by single EX.Flush signal
- Control determines when to flush:
  - Depends on Opcode and value of branch condition

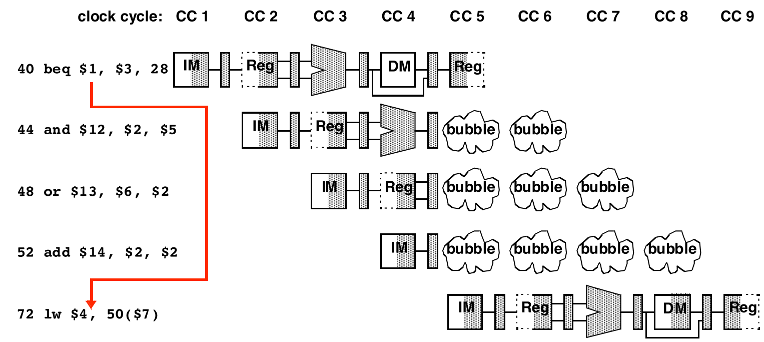
Assume "branch not taken"…and branch is not taken…

- Execution proceeds normally – no penalty



Assume "branch not taken"…and branch is taken…

- Bubbles injected into 3 stages during cycle 5



Branch Penalty Impact

- Assume 16% of all instructions are branches
  - 4% unconditional branches: 3 cycle penalty
  - 12% conditional: 50% taken
- For a sequence of N instructions (assume N is large)
    - N cycles to initiate each
    - 3 * 0.04 * N delays due to unconditional branches
    - 0.5 * 3 * 0.12 * N delays due to conditional taken
    - Also, an extra 4 cycles for pipeline to empty
- Total:
  - 1.3*N + 4 total cycles (or 1.3 cycles/instruction) (CPI)
    - 30% Performance Hit!!! (Bad thing)

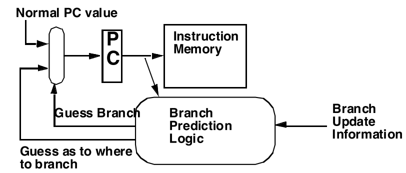
Branch Penalty Impact

- Some solutions:
  - In ISA: branches always execute next 1 or 2 instructions
    - Instruction so executed said to be in delay slot
    - See SPARC ISA
    - (example – loop counter update)
  - In organization: move comparator to ID stage and decide in the ID stage
    - Reduces branch delay by 2 cycles
    - Increases the cycle time

Branch Prediction

- Prior solutions are "ugly"
- Better (& more common): guess in IF stage
  - Technique is called "branch predicting"; needs 2 parts:
    - "Predictor" to guess where/if instruction will branch (and to where)
    - "Recovery Mechanism": i.e. a way to fix your mistake
  - Prior strategy:
    - Predictor: always guess branch never taken
    - Recovery: flush instructions if branch taken
  - Alternative: accumulate info. in IF stage as to…
    - Whether or not for any particular PC value a branch was taken next
    - To where it is taken
    - How to update with information from later stages

A Branch Predictor



Computing Performance

- Program assumptions:
  - 23% loads and in ½ of cases, next instruction uses load value
  - 13% stores
  - 19% conditional branches
  - 2% unconditional branches
  - 43% other
- Machine Assumptions:
  - 5 stage pipe with all forwarding
    - Only penalty is 1 cycle on use of load value immediately after a load)
    - Jumps are totally resolved in ID stage for a 1 cycle branch penalty
    - 75% branch prediction accuracy
    - 1 cycle delay on misprediction

The Answer:

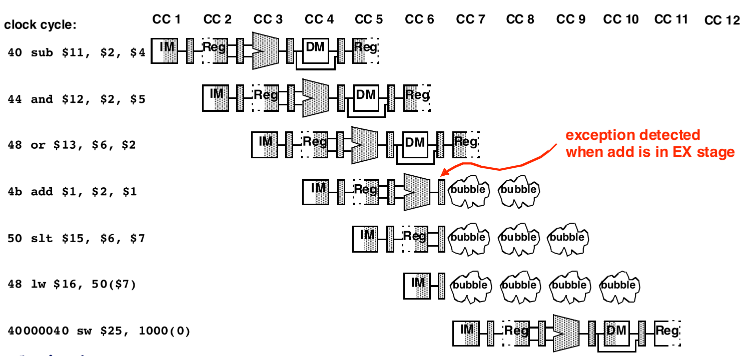
Lots more examples...
(handout)

Exception Hazards

- 40_{hex}: sub $11, $2, $4
- 44_{hex}: and $12, $2, $5
- 48_{hex}: or $13, $6, $2
- $4b_{hex}$: add $1, $2, $1 (overflow in EX stage)
- 50_{hex}: slt $15, $6, $7 (already in ID stage)
- 54_{hex}: lw $16, 50($7) (already in IF stage)
- …
- 40000040_{hex}: sw $25, 1000($0) exception handler
- 40000044_{hex}: sw $26, 1004($0)

- Need to transfer control to exception handler ASAP
  - Don't want invalid data to contaminate registers or memory
  - Need to flush instructions already in the pipeline
  - Start fetching instructions from 40000040_{hex}
  - Save addr. following offending instruction (50_{hex}) in TrapPC (EPC)
  - Don't clobber $1 – use for debugging

Flushing pipeline after exception



- Cycle 6:
  - Exception detected, flush signals generated, bubbles injected
- Cycle 7
  - 3 bubbles appear in ID, EX, MEM stages
  - PC gets 40000040_{hex}, TrapPC gets 50_{hex}

Managing exception hazards gets much worse!

- Different exception types may occur in different stages:

Exception Cause	Where it occurs
Undefined instruction	ID
Invoking OS	EX
I/O device request	Flexible
Hardware malfunction	Anywhere/flexible

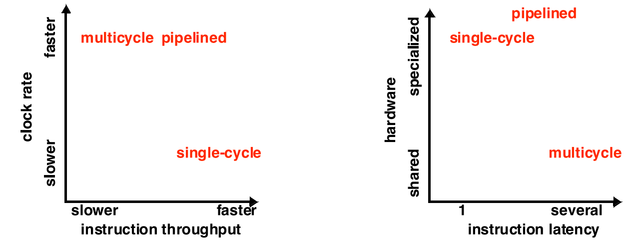
- Challenge is to associate exception with proper instruction: difficult!
  - Relax this requirement in non-critical cases: imprecise exceptions
    - Most machines use precise instructions
  - Further challenge: exceptions can happen at same time

Discussion

- How does instruction set design impact pipelining?

- Does increasing the depth of pipelining always increase performance?

Comparative Performance



- Throughput: instructions per clock cycle = 1/cpi
  - Pipeline has fast throughput and fast clock rate
- Latency: inherent execution time, in cycles
  - High latency for pipelining causes problems
    - Increased time to resolve hazards

Summary

- Performance:
  - Execution time *or* throughput
  - Amdahl's law
- Multi-bus/multi-unit circuits
  - one long clock cycle or N shorter cycles
- Pipelining
  - overlap independent tasks
- Pipelining in processors
  - "hazards" limit opportunities for overlap