
EECS 452 – Lecture 3

Pipelining and Interrupts

Instructor: Gokhan Memik

EECS Dept., Northwestern University

Pipelining



- Principles of pipelining
- Simple pipelining
- Structural Hazards
- Data Hazards
- Control Hazards
- Interrupts
- Multicycle operations
- Pipeline clocking

Sequential Execution Semantics



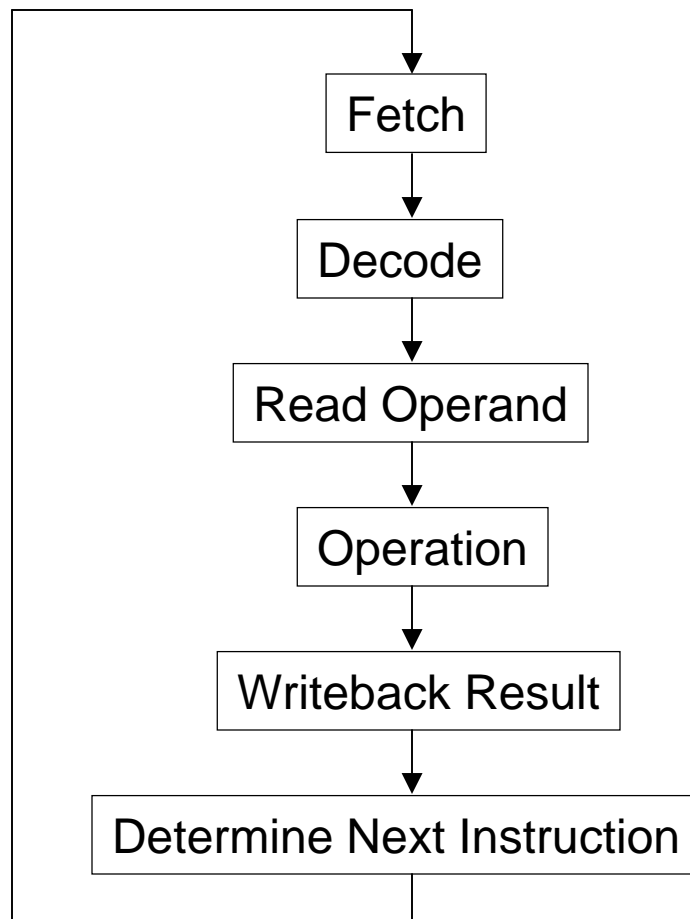
- We will be studying techniques that exploit the semantics of Sequential Execution.
- **Sequential Execution Semantics:**
 - instructions appear as if they executed in the program specified order and one after the other
- **Alternatively**
 - At any given point in time we should be able to identify an instruction so that:
 - **1. All preceding instructions have executed**
 - **2. None following has executed**

Exploiting Sequential Semantics



- **The “it should appear” is the key**
- **The only way one can inspect execution order is via the machine’s state**
 - **This includes registers, memory and any other named storage**
 - **We will looking at techniques that relax execution order while preserving sequential execution semantics**

Steps of Instruction Execution

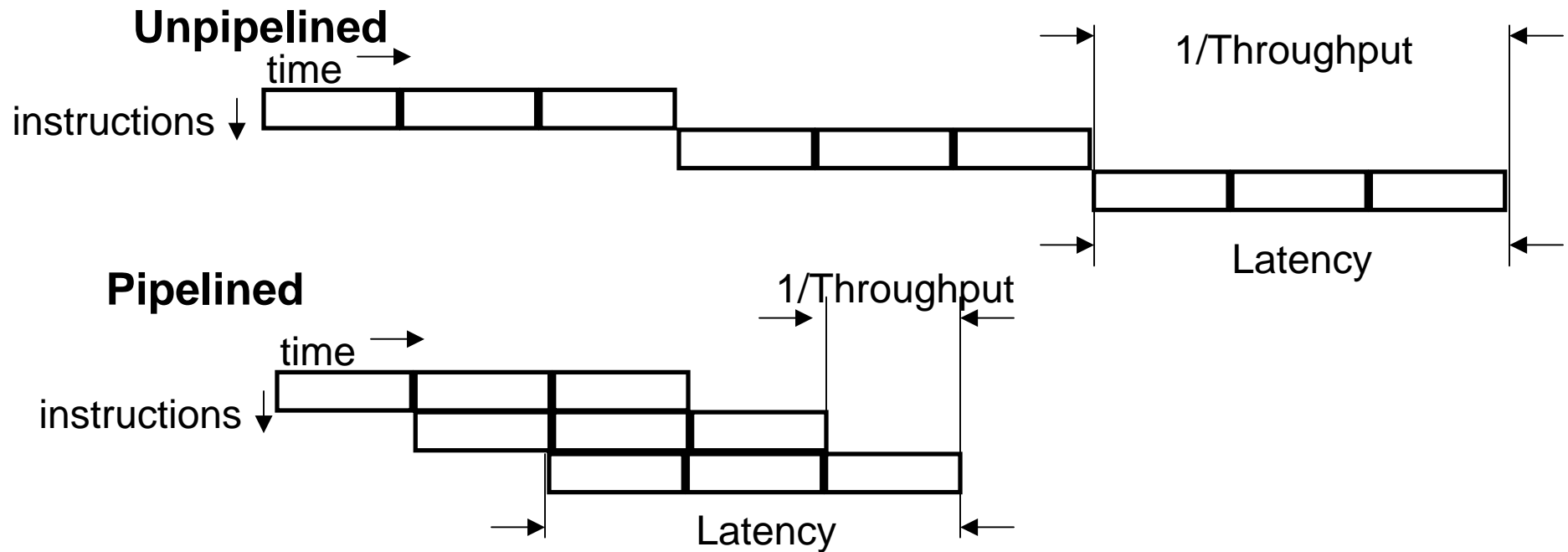


Instruction execution is not a monolithic action

There are multiple microactions involved



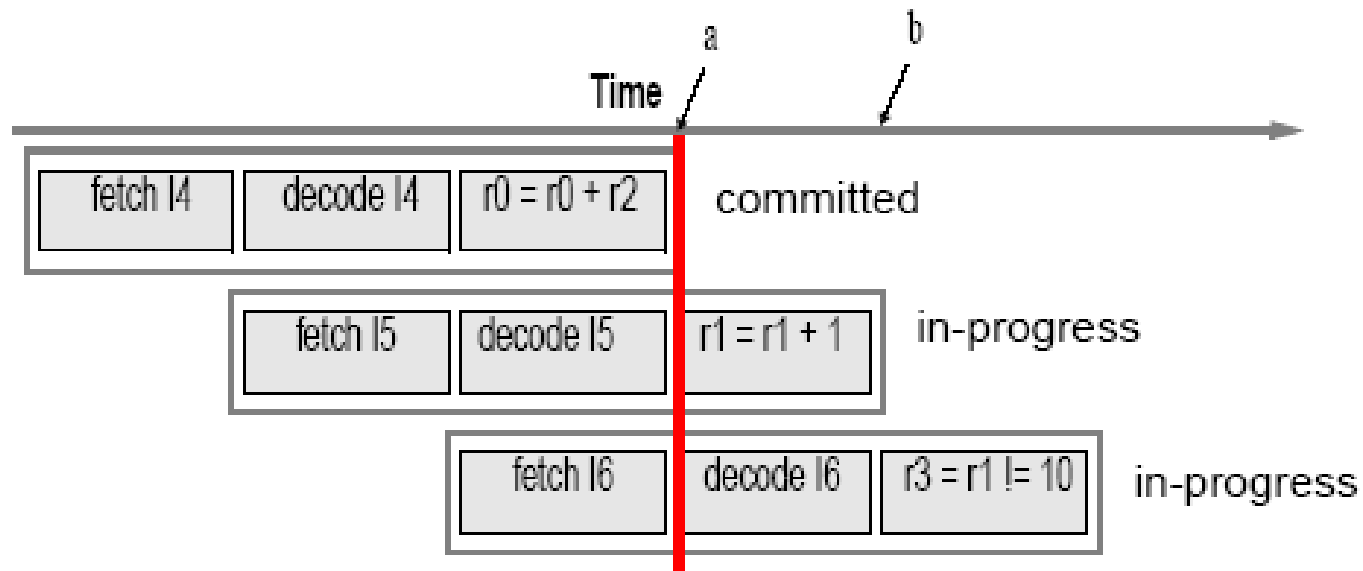
Pipelining: Partially Overlap Instructions



- **Ideally:** $Time_{pipeline} = \frac{Time_{sequential}}{PipelineDepth}$

- This ignores fill and drain times

Sequential Semantics Preserved?



- **Two execution states:**
 - **1. In-progress:** changes not visible to outsiders
 - **2. Committed:** changes visible

Principles of Pipelining: Ideal Case

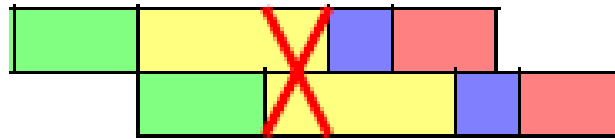


- Let **T** be the time to execute an instruction
- Instruction execution requires **n** stages, $t_1 \dots t_n$ taking
$$T = \sum t_i$$
- W/O pipelining: $TR = 1/T$ Latency = $T = 1/TR$
- W/ n-stage pipeline: $TR = \frac{1}{\max(t_i)} \leq \frac{n}{T}$ Latency = $n \times \max(t_i) \geq T$
- **if all t_i are equal, Speedup is n**
- **Ideally: Want higher Performance? Use more pipeline stages!!!**

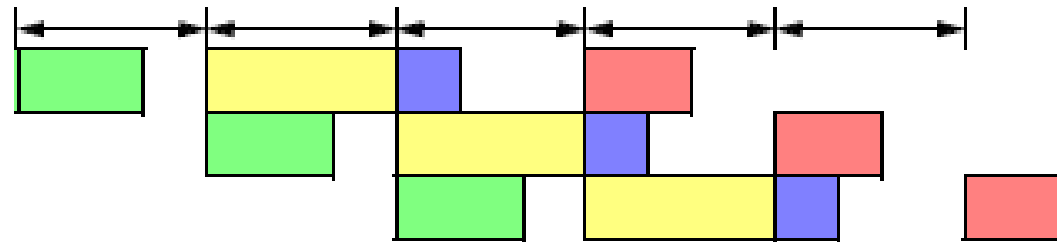
Principles of Pipelining: Example



Overlap!



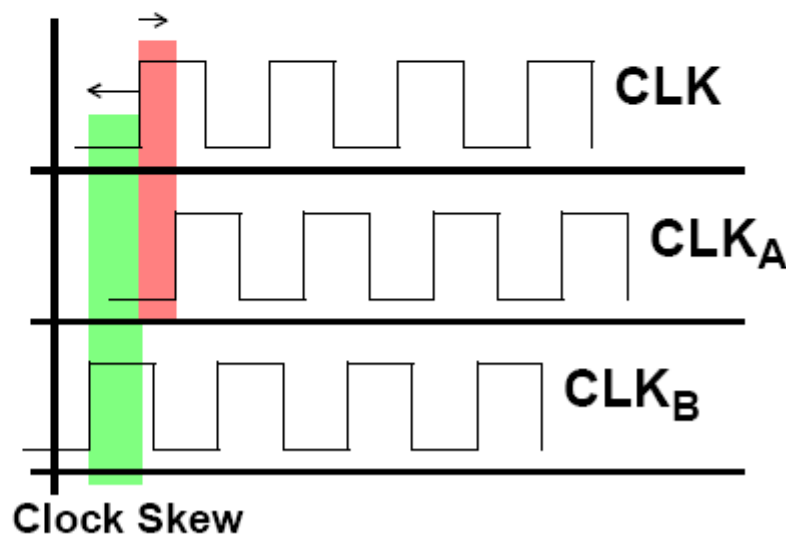
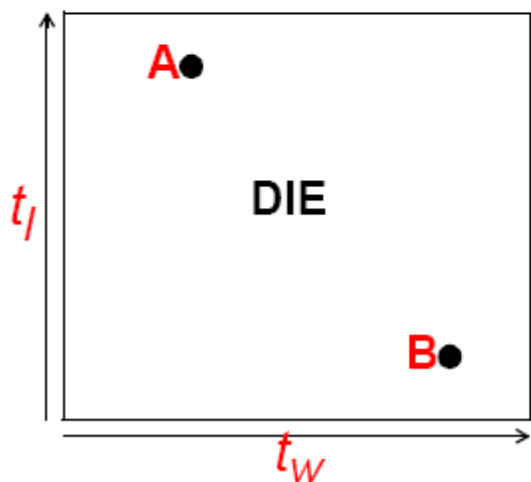
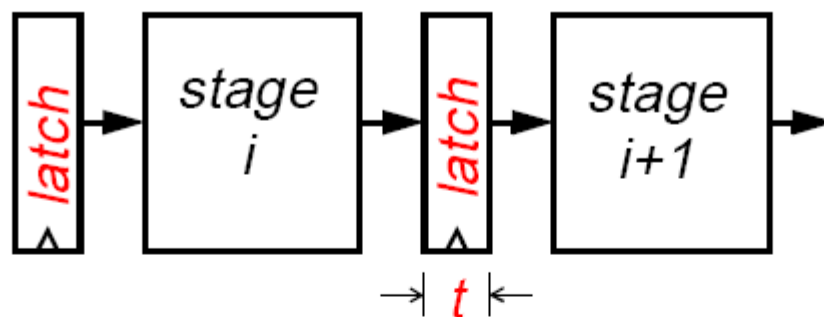
Pick Longest Stage



Critical Path Determines Clock Cycle



Why Can't Improve Performance that easily





Impact of Clock Skew and Latch Overheads

- **let X be extra delay per stage for**
 - latch overhead
 - clock/data skew
- **X limits the useful pipeline depth**
- **With n -stage pipeline (all t_i equal)**
 - throughput = $\frac{1}{X+t} \prec \frac{n}{T}$
 - latency = $n(X+t) = nX + T$
 - speedup = $\frac{T}{X+t} \leq n$
- **Real pipelines usually do not achieve this due to Hazards**

Simple pipelines

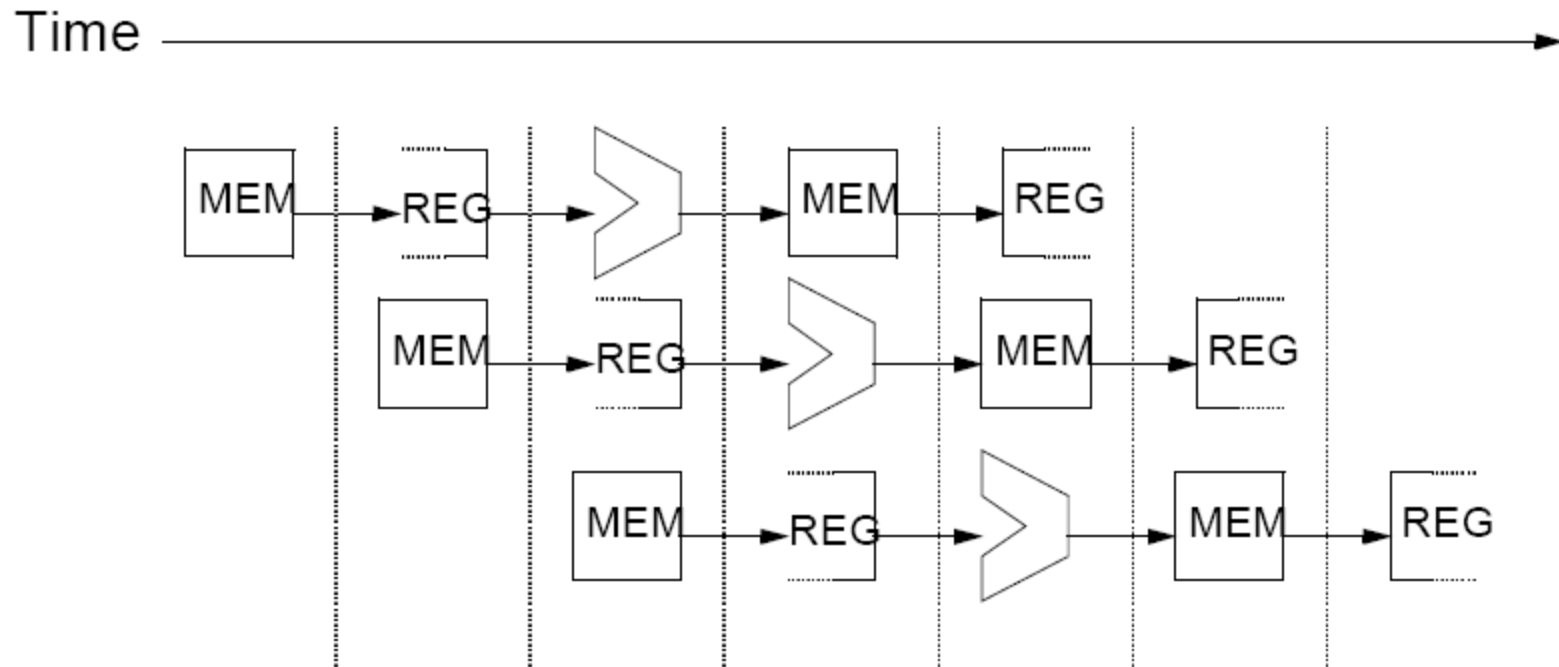


- **F- fetch, D - decode, X - execute, M - memory, W -writeback**

	1	2	3	4	5	6	7	8	9	10	11	12	13
i	F	D	X	M	W								
i+1		F	D	X	M	W							
i+2			F	D	X	M	W						
i+3				F	D	X	M	W					
i+4					F	D	X	M	W				

Classic 5-stage Pipeline

Pipelining as Datapaths in Time



Hazards



■ Hazards

- conditions that lead to incorrect behavior if not fixed

■ Structural Hazard

- two different instructions use same resource in same cycle

■ Data Hazard

- two different instructions use same storage
- must appear as if the instructions execute in correct order

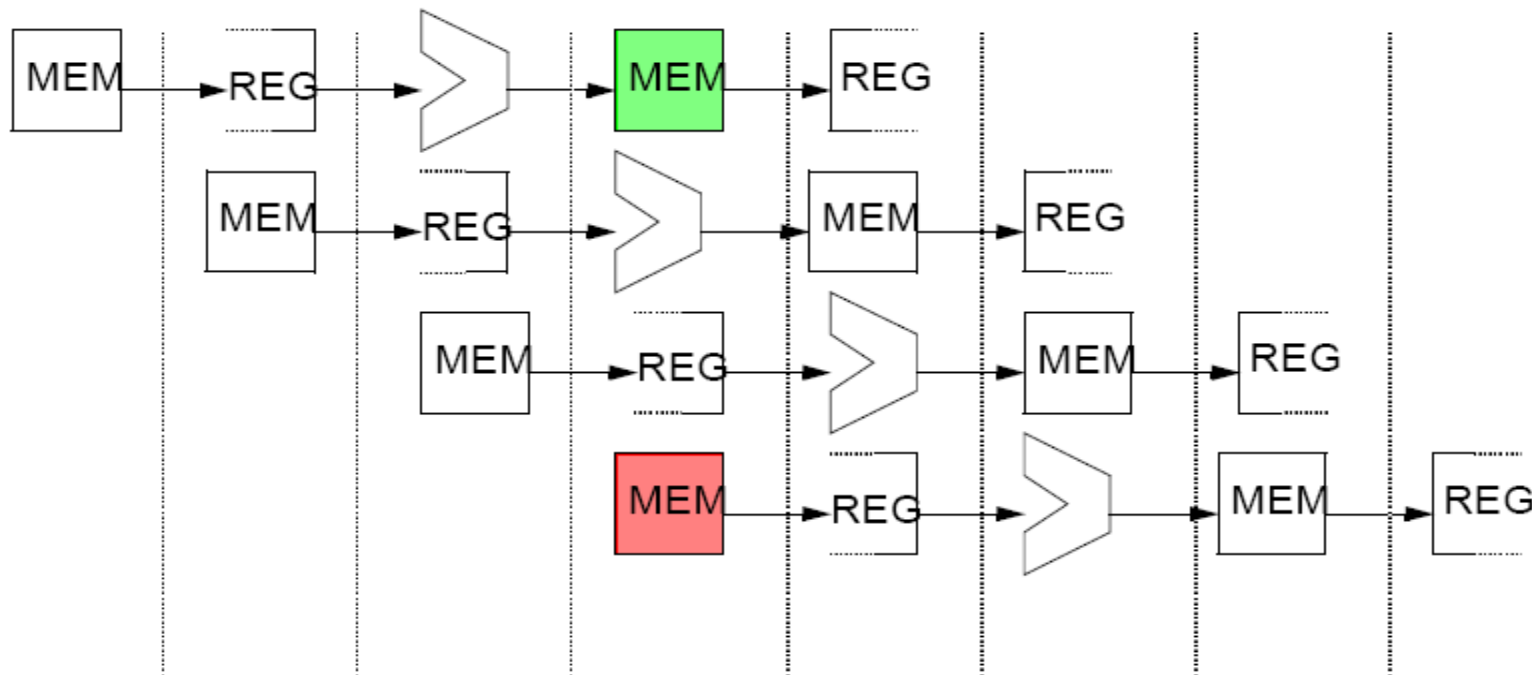
■ Control Hazard

- one instruction affects which instruction is next

Structural Hazards



- **When two or more different instructions want to use the same hardware resource in the same cycle**
 - e.g., load and stores use the same memory port as IF



Dealing with Structural Hazards



■ Stall:

- + low cost, simple
- – decrease IPC
- use for rare case

■ Pipeline Hardware Resource:

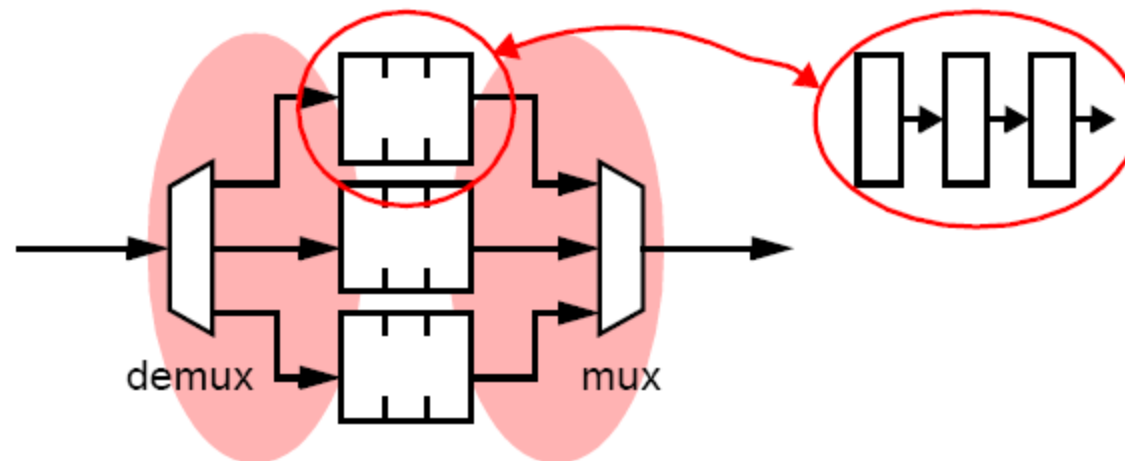
- useful for multicycle resources
- + good performance
- - sometimes complex e.g., RAM
- – Example 2-stage cache pipeline: decode, read or write data (wave pipelining - generalization)

Dealing with Structural Hazards



■ Replicate resource

- + good performance
- – increases cost
- – increased interconnect delay ?
- use for cheap or divisible resources





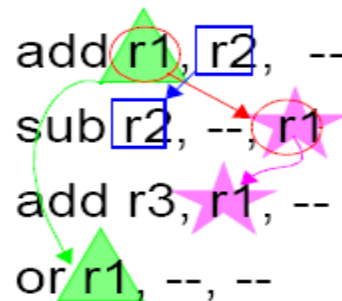
Impact of ISA on Structural Hazards

- **Structural hazards are reduced**
 - If each instruction uses a resource at most once
 - Always in same pipeline stage
 - For one cycle
- **Many RISC ISAs designed with this in mind**

Data Hazards



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



- **read-after-write** (RAW, true dependence) – real
- **write-after-read** (WAR, anti-dependence) – artificial
- **write-after-write** (WAW, output-dependence) – artificial
- **read-after-read** (no hazard)

Examples of RAW



add r1, --, -- IF ID EX MEM **WB** *r1 written*
sub --, r1, -- IF **ID** EX MEM WB
r1 read - not OK

load r1, --, -- IF ID EX MEM **WB** *r1 written*
sub --, r1, -- IF **ID** EX MEM WB
r1 read - not OK

sw r1, 100(r2) IF ID EX **MEM** WB
lw r1, 100(r2) IF ID EX **MEM** WB
OK

unless 100(r2) is the PC of the load (self-modifying code)

Simple Solution to RAW



- **Hardware detects RAW and stalls**

add r1, --, -- IF ID EX MEM **WB** *r1 written*
sub --, r1, -- IF stall stall IF **ID** EX MEM WB

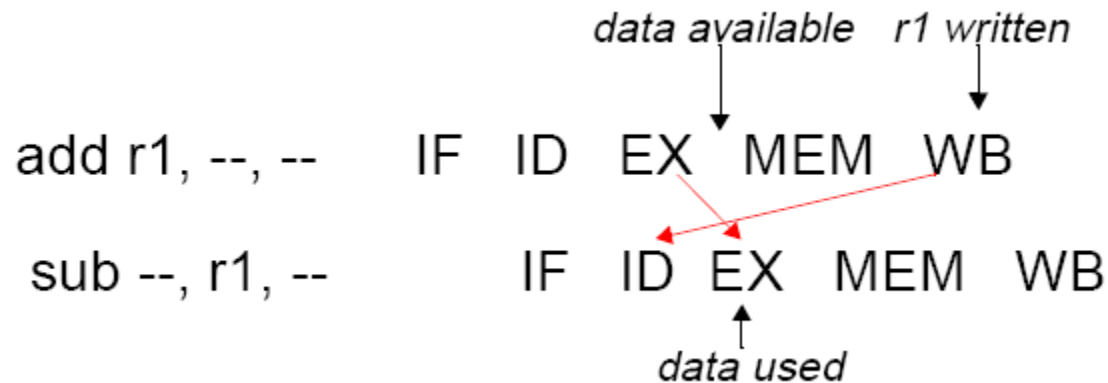
- + low cost, simple
- – reduces IPC

- **Maybe we should try to minimize stalls**

Minimizing RAW stalls



■ Bypass or Forward or Short-Circuit



■ Use data before it is in register

- + reduces/avoids stalls
- – complex
- – Deeper pipelines -> more places to bypass from
- crucial for common RAW hazards

Bypass



- **Interlock logic**
 - detect hazard
 - bypass correct result to ALU
- **Hardware detection requires extra hardware**
 - instruction latches for each stage
 - comparators to detect the hazards

Bypass: Control Example



■ Mux control

- if $\text{insn}(\text{EX})$ uses immediate then select IMM
- else if $\text{rs2}(\text{EX}) == \text{rd}(\text{MEM})$ then $\text{ALUOUT}(\text{MEM})$
- else if $\text{rs2}(\text{EX}) == \text{rd}(\text{WB})$ then $\text{ALUOUT}(\text{WB})$
- else select B

RAW solutions



- **Hybrid (i.e., stall and bypass) solutions required sometimes**

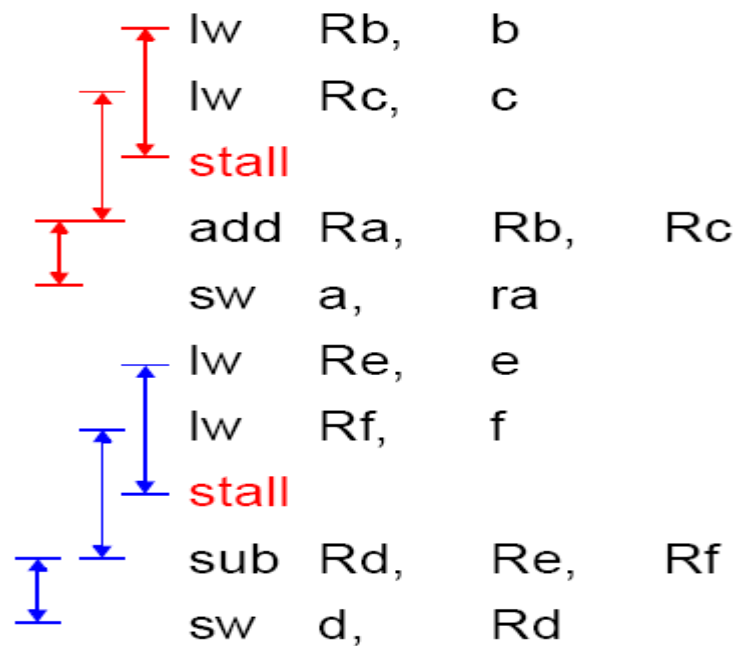
```
load r1, --, --   IF  ID  EX  MEM  WB
sub --, r1, --    stall IF  ID  EX  MEM  WB
```

- **DLX has one cycle bubble if load result used in next instruction**
- **Try to separate stall logic from bypass logic**
 - avoid irregular bypasses



Pipeline Scheduling - Compilers can help

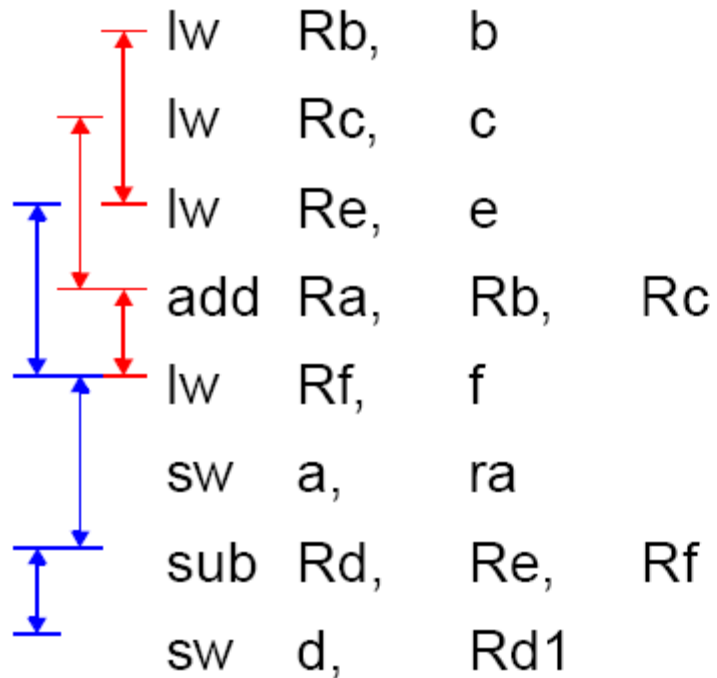
- Instructions scheduled by compiler to reduce stalls
- $a = b + c; d = e + f$
- -- Prior to scheduling



Pipeline Scheduling



■ After scheduling



No Stalls

Delayed Load



- Avoid hardware solutions - Let the compiler deal with it
- Instruction Immediately after load can't/shouldn't see load result
- Compiler has to fill in the delay slot - NOP might be necessary

UNSCHEDULED

```
lw Rb, b
lw Rc, c
nop
add Ra, Rb, Rc
sw a, Ra
lw Re, efs
lw Rf, f
nop
add Rd, Re, Rf ...
```

SCHEDULED

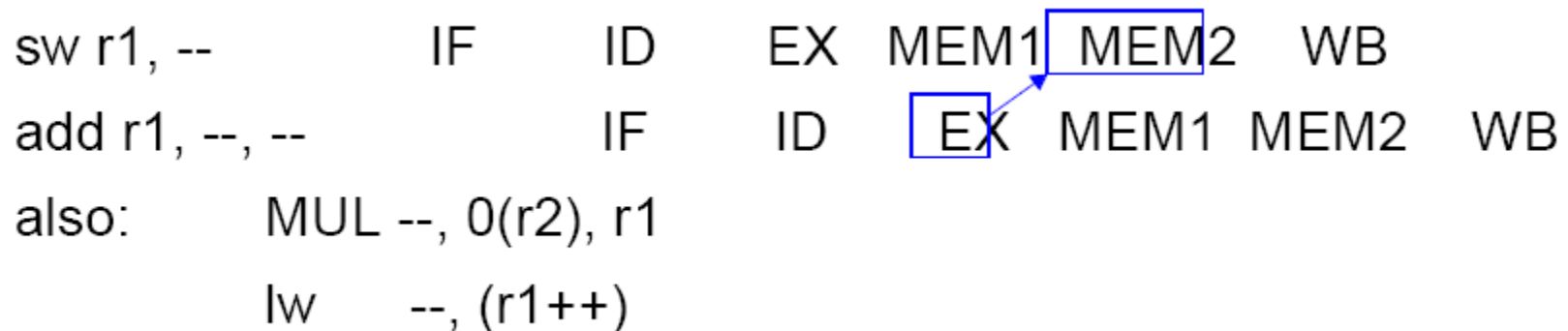
```
lw Rb, b
lw Rc, c
lw Rf, f
add Ra, Rb, Rc
lw Rf, f
sw a, Ra
sub Rd, Re, Rf
sw d, Rd
```

Other Data Hazards



- **WAR**
add r1, r2, --
sub r2, --, r1
or r1, --, --

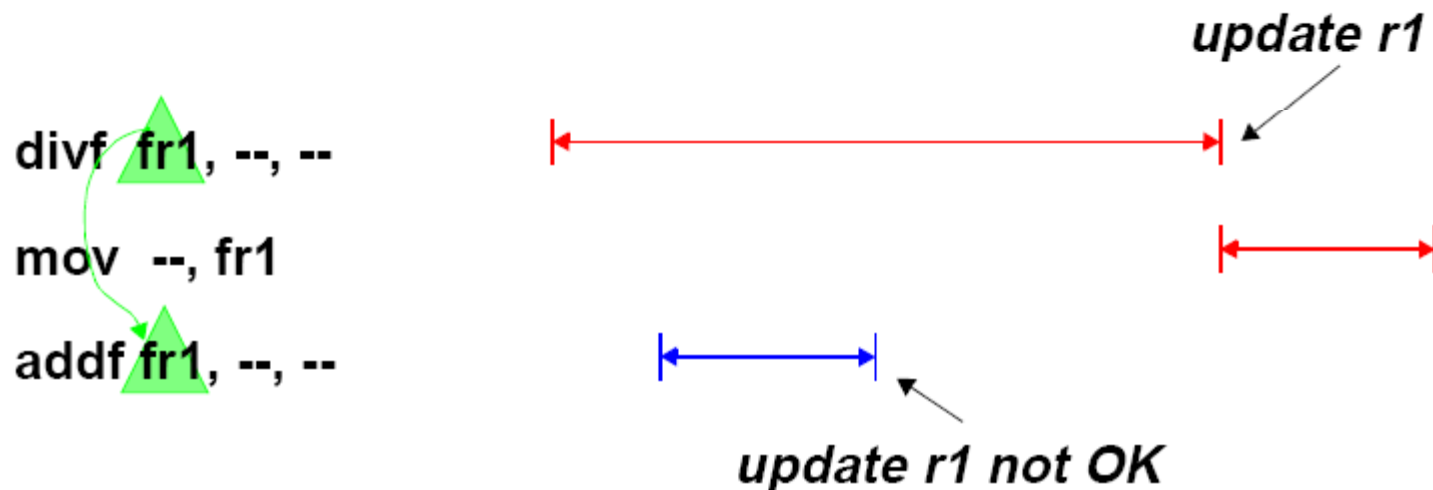
- Not possible in DLX - read early write late
- Consider late read then early write
 - ALU ops writeback at EX stage
 - MEM takes two cycles and stores need source reg after 1 cycle



Other Data Hazards



- **WAW**
- **Not in DLX : register writes are in order**
- **consider slow then fast operation**



Control Hazards



- **When an instruction affects which instruction execute next or changes the PC**
 - `sw $4, 0($5)`
 - `bne $2, $3, loop`
 - `sub -, -, -`

	1	2	3	4	5	6	7	8	9
<code>sw</code>	F	D	X	M	W				
<code>bne</code>		F	D	X*	M	W			
<code>??</code>					F	D	X	M	W

Control Hazards



- **Handling control hazards is very important**
- **VAX e.g.,**
 - Emer and Clark report 39% of instr. change the PC
 - Naive solution adds approx. 5 cycles every time
 - Or, adds 2 to CPI or ~20% increase
- **DLX e.g.,**
 - H&P report 13% branches
 - Naive solution adds 3 cycles per branch
 - Or, 0.39 added to CPI or ~30% increase

Handling Control Hazards



- **Move control point earlier in the pipeline**
 - Find out whether branch is taken earlier
 - Compute target address fast

Both need to be done

- **e.g., in ID stage**
 - $\text{target} := \text{PC} + \text{immediate}$
 - $\text{if } (Rs_1 \text{ op } 0) \text{ PC} := \text{target}$

Handling Control Hazards



	1	2	3	4	5	6	7	8	9
N: sw	F	D	X	M	W				
N+1: bne		F	D	X	M	W			
N+2: add			F			squashed			
Y: sub				F	D	X			

- **Implies only one cycle bubble but**
 - special PC adder required
 - How about register file?

ISA and Control Hazard



- **Comparisons in ID stage**
 - must be fast
 - can't afford to subtract
 - compares with 0 are simple
 - gt, lt test sign-bit
 - eq, ne must OR all bits
- **More general conditions need ALU**
 - DLX uses conditional sets

Handling Control Hazards



- **Branch prediction**
 - guess the direction of branch
 - minimize penalty when right
 - may increase penalty when wrong
- **Techniques**
 - static - by compiler
 - dynamic - by hardware
 - MORE ON THIS LATER ON

Handling Control Hazards



- **Static techniques**
 - predict always not-taken
 - predict always taken
 - predict backward taken
 - predict specific opcodes taken
 - delayed branches
- **Dynamic techniques**
 - Discussed with ILP

Handling Control Hazards



■ Predict not-taken always

	1	2	3	4	5	6	7	8	9
i	F	D	X	M	W				
i+1		F	D	X	M	W			
i+2			F	D	X	M			

■ if taken then squash (aka abort or rollback)

- will work only if no state change until branch is resolved
- DLX - ok - why?
- VAX - autoincrement addressing?

Handling Control Hazards



- **Predict taken always**

	1	2	3	4	5	6	7	8	9
i	F	D	X	M	W				
i+8		F	D	X	M	W			
i+9			F	D	X	M			

- **For DLX must know target before branch is decoded!**
 - can use prediction
 - special hardware for fast decode
- **Execute both paths**

Handling Control Hazards



- **Delayed branch** - execute next instruction whether taken or not
- `i: beqz r1, #8`
- `i+1: sub --, --, --`
- `.....`
- `i+8 : or --, --, --` (reused by RISC invented by microcode)

	1	2	3	4	5	6	7	8	9
<code>i</code>	F	D	X	M	W				
<code>i+1 (delay slot)</code>		F	D	X	M	W			
<code>i+8</code>			F	D	X	M			

Filling in Delay slots

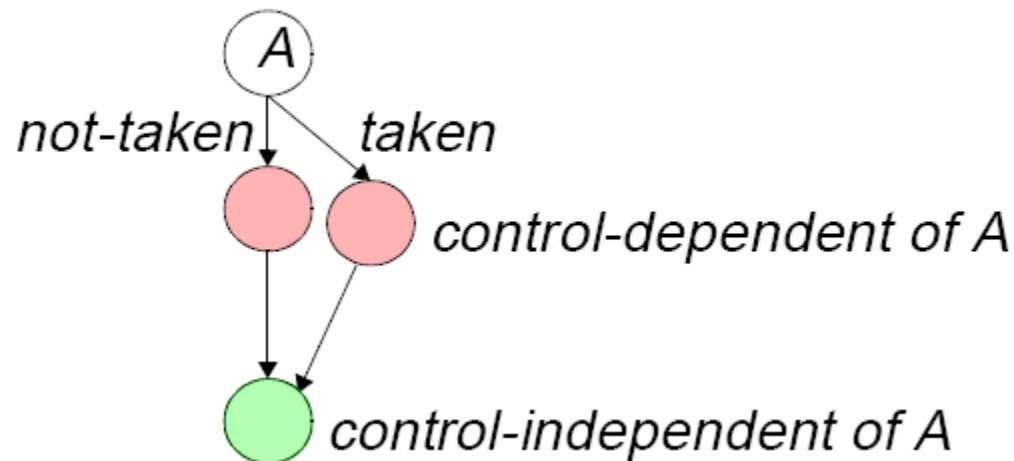


- **Fill with an instr before branch**
 - When? if branch and instr are independent.
 - Helps? Always
- **Fill from target (taken path)**
 - When? if safe to execute target, may have to duplicate code
 - Helps? on taken branch, may increase code size
- **Fill from fall-through (not-taken path)**
 - when? if safe to execute instruction
 - helps? when not-taken

Filling in Delay Slots cont.



- From Control-Independent code:
 - code that will be eventually visited no matter where the branch goes



- **Nullifying or Canceling or Likely Branches:**
 - Specify when delay slot is execute and when is squashed
 - **Why?** Increase fill opportunities
 - **Major Concern w/ DS:** Exposes implementation optimization

Comparison of Branch Schemes



- **Cond. Branch statistics – DLX**
 - 14%-17% of all insts (integer)
 - 3%-12% of all insts (floating-point)
 - Overall 20% (int) and 10% (fp) control-flow insts.
 - About 67% are taken
- **Branch-Penalty = %branches x (%taken x taken-penalty + %not-taken x not-taken-penalty)**

Comparison of Branch Schemes



scheme	taken penalty	not-taken pen.	CPI penalty
naive	3	3	0.420
fast branch	1	1	0.140
not-taken	1	0	0.091
taken	0	1	0.049
delayed branch	0.5	0.5	0.070

- **Assuming: branch% = 14%, taken% = 65%, 50% delay slots are filled w/ useful work**
- **ideal CPI is 1**

Impact of Pipeline Depth



- Assume that now penalties are doubled
- **For example we double clock frequency**

scheme	taken penalty	not-taken pen.	CPI penalty
naive	6	6	0.840
fast branch	2	2	0.280
not-taken	2	0	0.182
taken	0	2	0.098
delayed branch	?	?	?

- **Delayed Branches need special support for interrupts**

Interrupts



- **Examples:**
 - power failing, arithmetic overflow
 - I/O device request, OS call, page fault
 - Invalid opcode, breakpoint, protection violation
- **Interrupts (aka faults, exceptions, traps) often require**
 - surprise jump (to vectored address)
 - linking return address
 - saving of PSW (including CCs)
 - state change (e.g., to kernel mode)

Classifying Interrupts



- **1a. Synchronous**
 - function of program state (e.g., overflow, page fault)
- **1b. Asynchronous**
 - external device or hardware malfunction
- **2a. user request**
 - OS call
- **2b. Coerced**
 - from OS or hardware (page fault, protection violation)

Classifying Interrupts



- **3a. User Maskable**
 - User can disable processing
- **3b. Non-Maskable**
 - User cannot disable processing
- **4a. Between Instructions**
 - Usually asynchronous
- **4b. Within an instruction**
 - Usually synchronous - Harder to deal with
- **5a. Resume**
 - As if nothing happened? Program will continue execution
- **5b. Termination**

Restartable Pipelines



- Interrupts within an instruction are not catastrophic
 - Most machines today support this
 - Needed for virtual memory
 - Some machines did not support this
 - Why?
 - Cost
 - Slowdown
- Key: Precise Interrupts
- First let's consider a simple DLX-style pipeline

Handling Interrupts



- **Precise interrupts (sequential semantics)**

- Complete instructions before the offending instr
- Squash (effects of) instructions after
- Save PC (& next PC with delayed branches)
- Force trap instruction into IF

- **Must handle simultaneous interrupts**

- IF, M - memory access (page fault, misaligned, protection)
- ID - illegal/privileged instruction
- EX - arithmetic exception

Interrupts



- **E.g., data page fault**

	1	2	3	4	5	6	7	8	9	
i	F	D	X	M	W					
i+1		F	D	X	M	W				<- page fault
i+2			F	D	X					<- squash
i+3				F	D					<- squash
i+4					F					<- squash
x		trap ->				F	D	X	M	W
x+1		trap handler ->				F	D	X	M	W

Interrupts



- **Preceding instructions already complete**
- **Squash succeeding instructions**
 - prevent them from modifying state (registers, CC, memory)
- **Trap instruction jumps to trap handler**
- **hardware saves PC in IAR**
- **trap handler must save IAR**

Interrupts



- **E.g., arithmetic exception**

	1	2	3	4	5	6	7	8	9	
i	F	D	X	M	W					
i+1		F	D	X	M	W				
i+2			F	D	X					<- exception
i+3				F	D					<- squash
i+4					F					<- squash
x		trap ->				F	D	X	M	W
x+1		trap handler ->				F	D	X	M	W

Interrupts



- **E.g., Instruction fetch page fault**

	1	2	3	4	5	6	7	8	9		
i	F	D	X	M	W						
i+1		F	D	X	M	W					
i+2			F	D	X	M	W				
i+3				F	D	X	M	W			
i+4					F				<- page fault		
x		trap ->				F	D	X	M	W	
x+1		trap handler ->					F	D	X	M	W

Interrupts



- **Let preceding instructions complete**
- **No succeeding instructions**
- **What happens if $i+3$ causes a data page fault?**

Interrupts



■ Out-of-order interrupts

- which page fault should we take?

	1	2	3	4	5	6	7	8	9
i	F	D	X	M	W				page fault (Mem)
i+1		F	D	X	M	W			page fault (fetch)
i+2			F	D	X	M	W		
i+3				F	D	X	M	W	

Out-of-Order Interrupts



- **Post interrupts**

- check interrupt bit on entering WB
- precise interrupts
- longer latency

- **Handle immediately**

- not fully precise
- interrupt may occur in order different from sequential CPU
- may cause implementation headaches!

Interrupts



- **Other complications**
 - odd bits of state (e.g., CC)
 - early-writes (e.g., autoincrement)
 - instruction buffers and prefetch logic
 - dynamic scheduling
 - out-of-order execution
- **Interrupts come at random times**
- **Both Performance and Correctness**
 - frequent case not everything
 - rare case **MUST** work correctly

Delayed Branches and Interrupts



- **What happens on interrupt while in delay slot**
 - next instruction is not sequential
- **Solution #1: save multiple PCs**
 - save current and next PC
 - special return sequence, more complex hardware
- **Solution #2: single PC plus**
 - branch delay bit
 - PC points to branch instruction
 - SW Restrictions

Multicycle operations



- **Not all operations complete in 1 cycle**
 - FP slower than integer
 - 2-4 cycles multiply or add
 - 20-50 cycles divide
- **Extend DLX pipeline**
 - EX stage repeated multiple times
 - multiple, parallel functional units
 - not pipelined for now

Handling Multicycle Operations



■ Four functional units

- EX: integer E*: FP/integer multiplier
- E+: FP adder E/: FP/integer divider

■ Assume

- EX takes 1 cycle and all FP take 4 cycles
- separate integer and FP registers
- all FP arithmetic in FP registers

■ Worry about hazards

- structural, RAW (forwarding), WAR/WAW (between I & FP)

Simple Multicycle Example



	1	2	3	4	5	6	7	8	9	10	11	
int	F	D	X	M	W							
fp*		F	D	E*	E*	E*	E*	M	W			
int			F	D	EX	M	W	(1)				
fp/				F	D	E/	E/	E/	E/	M	W	
int					F	D	EX	M	W	(2)		
fp/						F	D	--	--	E/	E/	(3)
fp*							F	--	--	D	X	(4)

Simple Multicycle Example



■ Notes:

- (1) - no WAW but complicates interrupts
- (2) - no WB conflict
- (3) - stall forced by structural hazard
- (4) - stall forced by in-order issue

■ Different FP operation times are possible

- Makes FP WAW hazards possible
- Further complicates interrupts

FP Instruction Issue



- **Check for structural hazards**
 - wait until functional unit is free
- **Check for RAW - wait until**
 - source regs are not used as destinations by instrs in Ex_i
- **Check for forwarding**
 - bypass data from MEM or WB if needed
- **What about overlapping instructions?**
 - contention in WB
 - possible WAR/WAW hazards
 - interrupt headaches

Overlapping Instructions



- **Contention in WB**
 - static priority
 - e.g., FU with longest latency
 - instructions stall after issue
- **WAR hazards**
 - always read registers at same pipe stage
- **WAW hazards**
 - divf f0, f2, f4 followed by subf f0, f8, f10
 - stall subf or abort divf's WB

Multicycle Operations

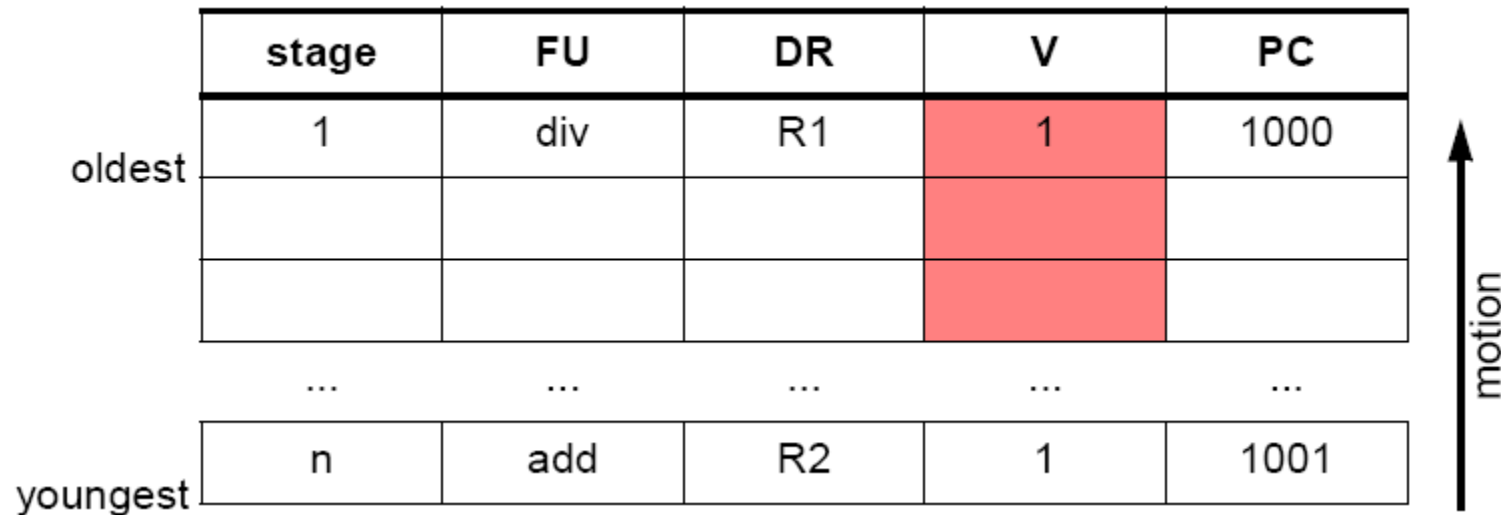


- **Problems with interrupts**
 - DIVF f0, f2, f4
 - ADDF f2, f8, f10
 - SUBF f6, f4, f10
- **ADDF completes before DIVF**
 - Out-Of-Order completion
 - Possible imprecise interrupts
- **What if divf excepts after addf/subf complete?**
- **Precise Interrupts Paper**

Precise Interrupts

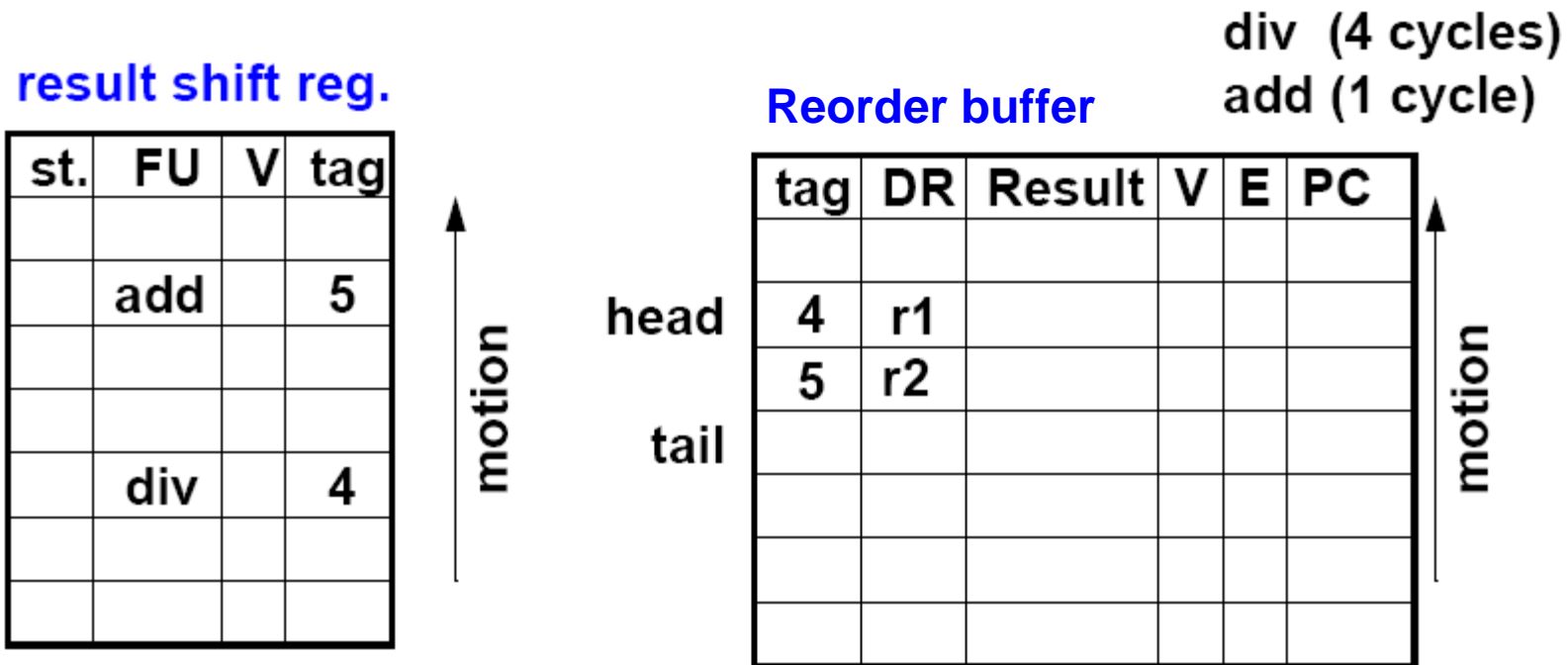


- Simple solution: Modify state only when all preceding insts. are known to be exception free.
- Mechanism: Result Shift Register



- Reserve all stages for the duration of the instruction
- **Memory: Either stall stores at decode or use dummy store**

Reorder Buffer

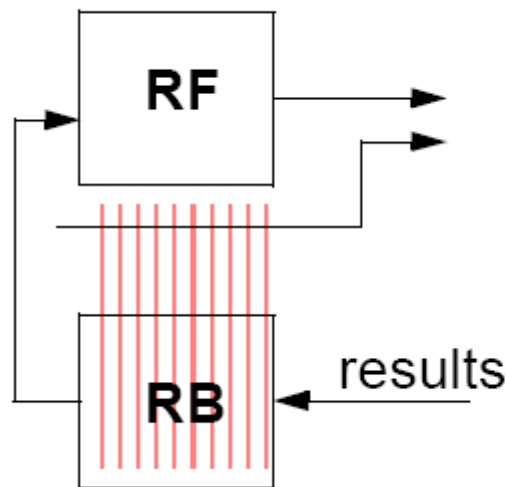


- Out-of-order completion
- Commit: Write results to register file or memory
- Reorder buffer holds not yet committed state

Reorder Buffer Complications



- State is kept in the reorder buffer
- May have to bypass from every entry
- Need to determine the latest write
- If read not at same stage need to determine closest earlier write

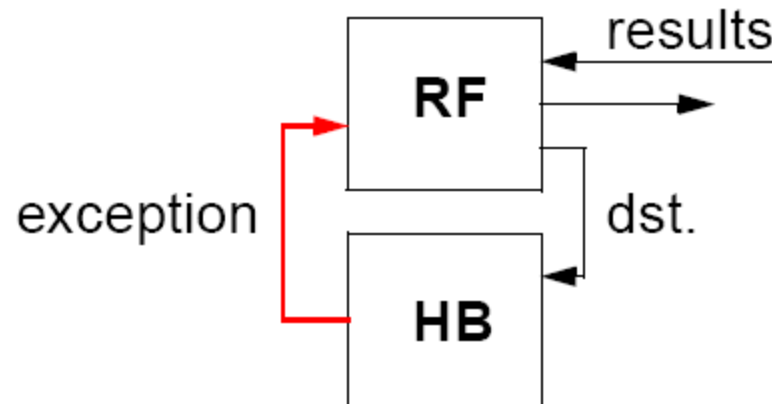


Two Solutions:
History Buffer
Future File

History Buffer



- Allow out-of-order register file updates
- At decode record current value of target register in reorder buffer entry.
- On commit: do nothing
- On exception: scan following reorder buffer entries restoring register values



Future File



- Two register files:
 - One updated out-of-order (FUTURE)
 - assume no exceptions will occur
 - One updated in order (ARCHITECTURAL)
- Advantage: No delay to restore state on exception

