
Pipelining and Precise Interrupts

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

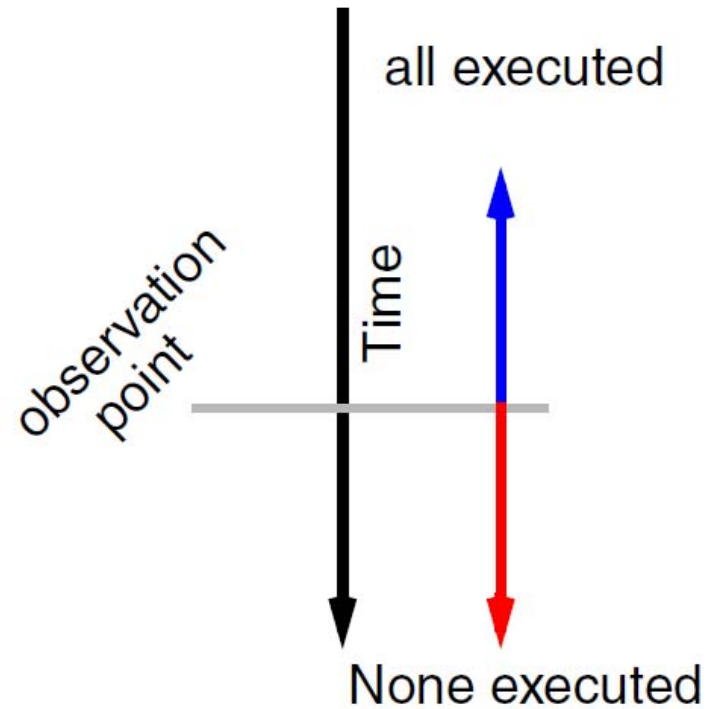
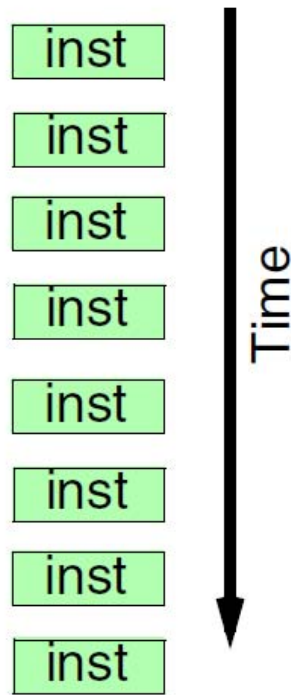
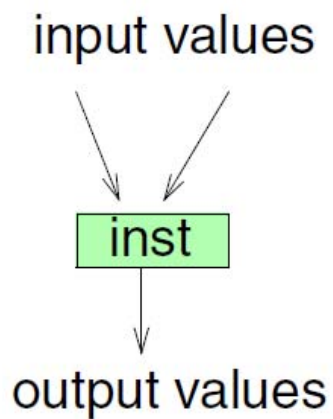
Alternatively

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**

Sequential Execution Semantics

- **Contract:** This is how the machine *appears to behave*



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

First is PIPELINING:

Partially Overlap Instructions

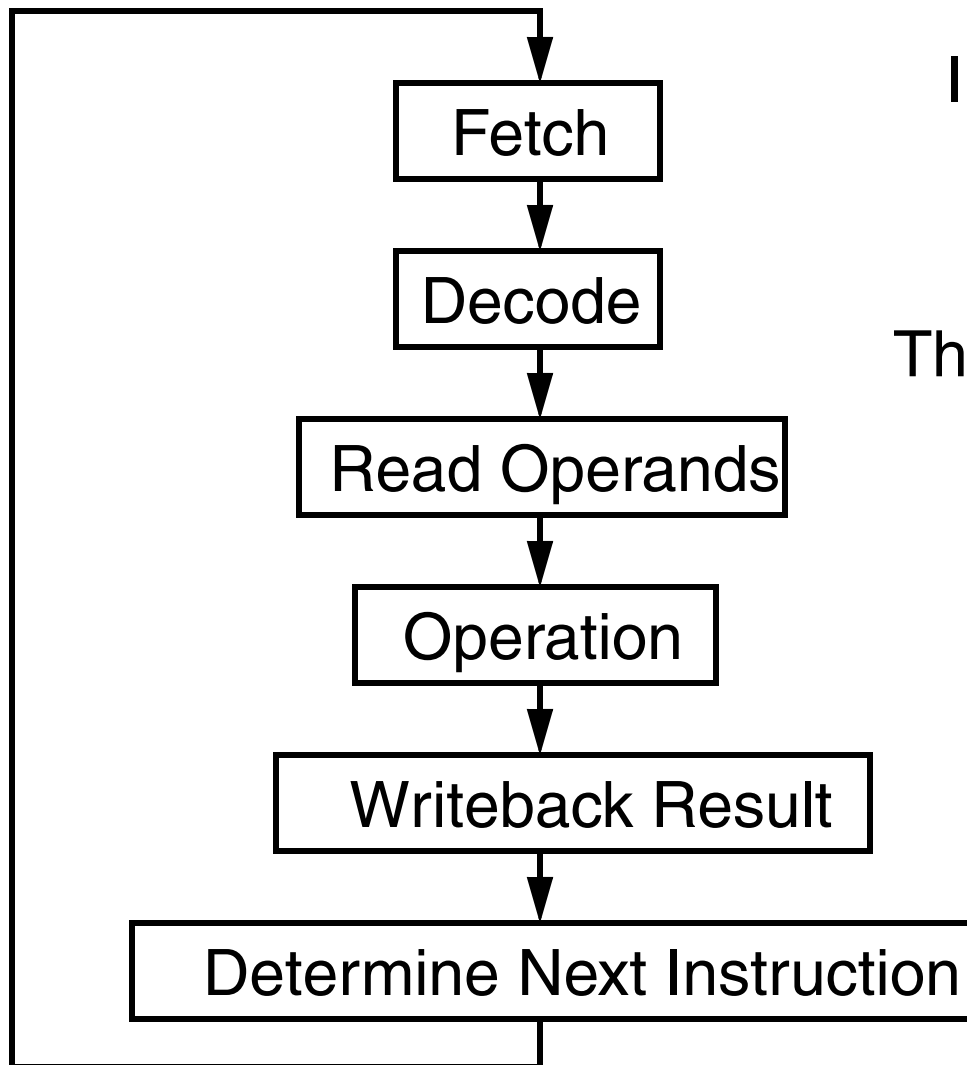
Treat each as a sequence of micro-actions and overlap those

Disecting Instructions

- One way or another instructions fall under the following categories:
 - 1. Data movement:** memory or register read and write
 - 2. Data manipulation:** add, subtract, etc.
 - 3. Control Flow:** Based on data decide what to do next, e.g., branch, jump.

How much of this and how can greatly impact the pipelinability of an instruction set architecture.

Steps of Instruction Execution

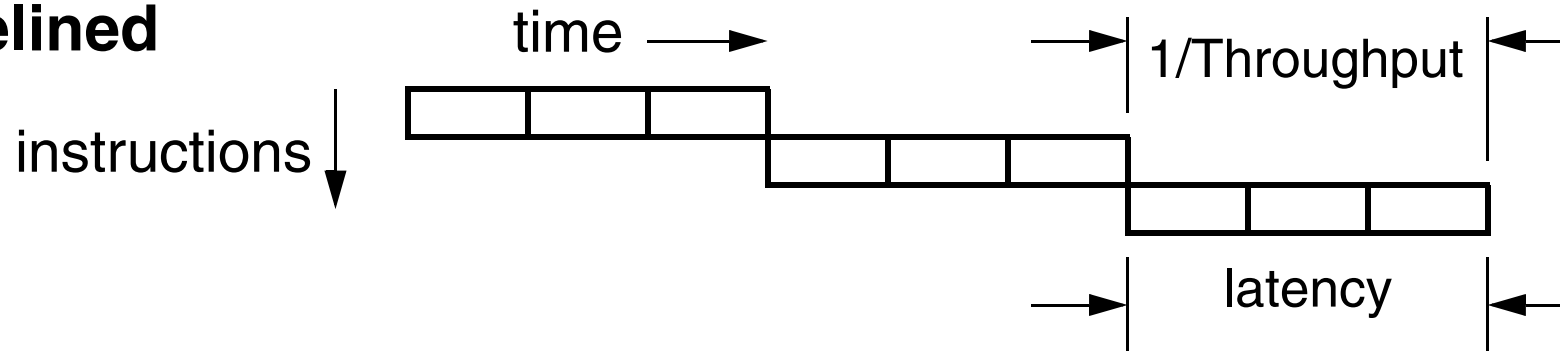


Instruction execution is not a monolithic action

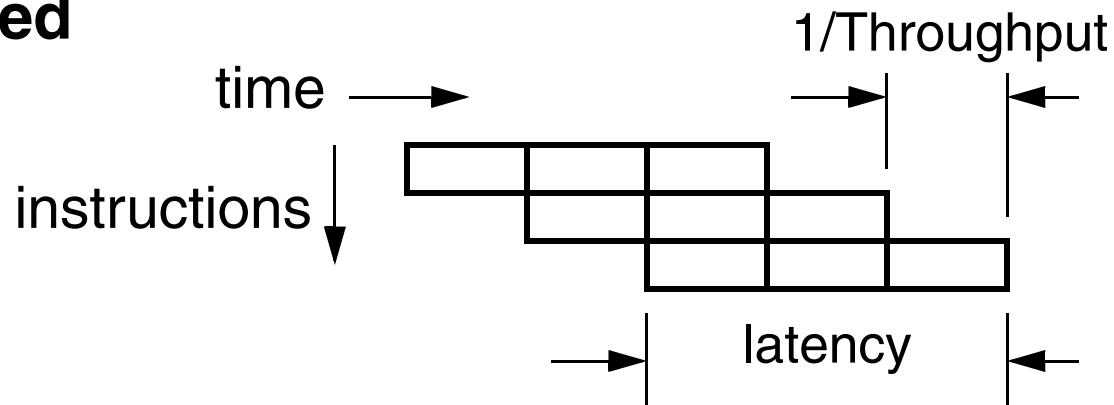
There are multiple *micro-actions* involved

Pipelining: Partially Overlap Instructions

Unpipelined



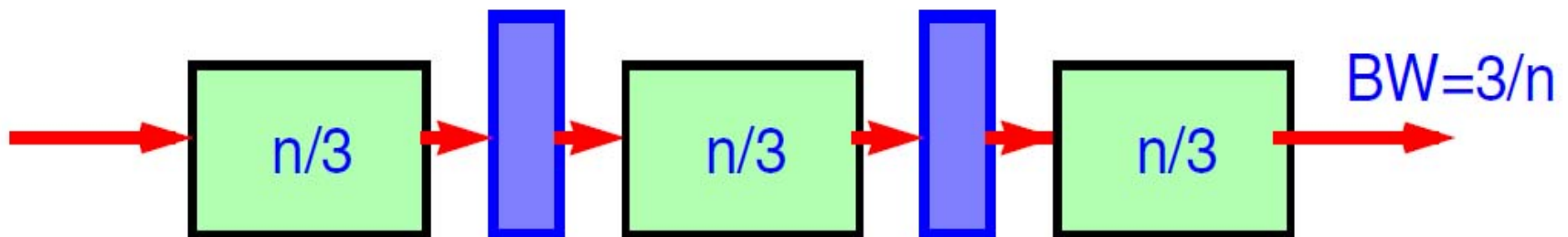
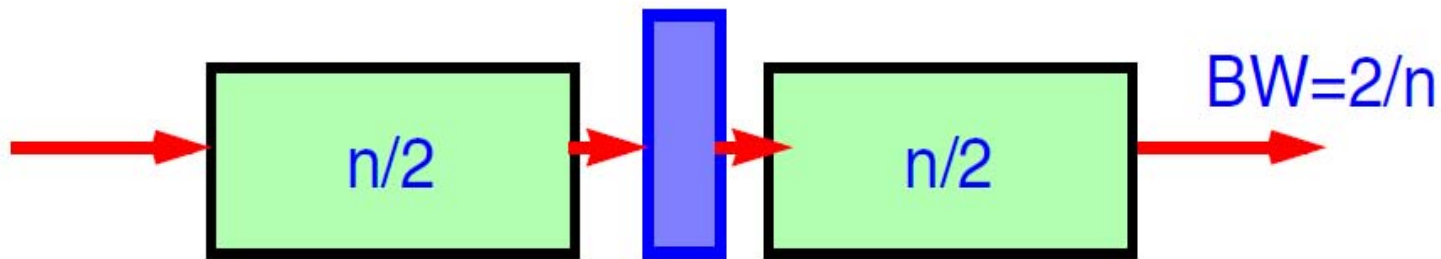
Pipelined



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

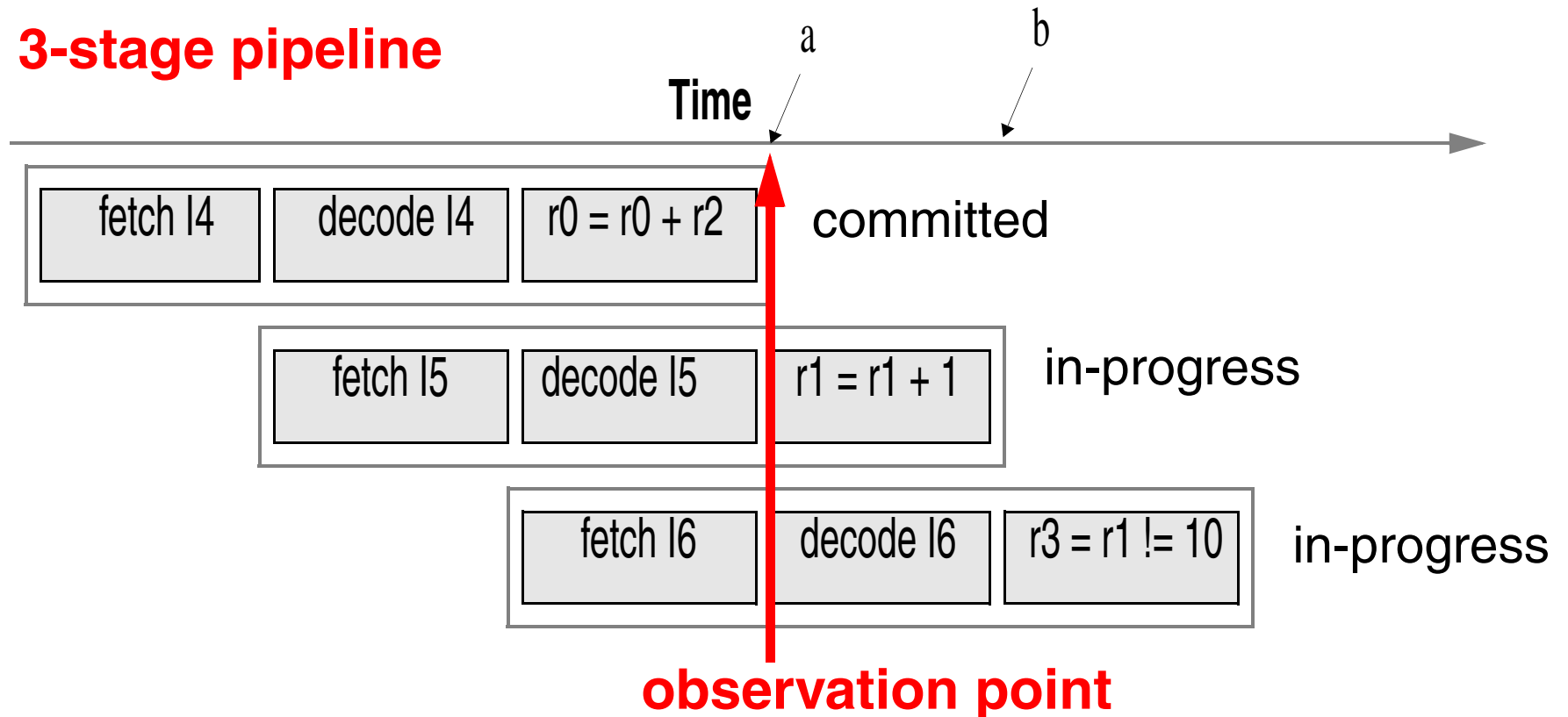
This ignores fill and drain times

Pipelining is a more general concept



Sequential Semantics Preserved?

3-stage pipeline



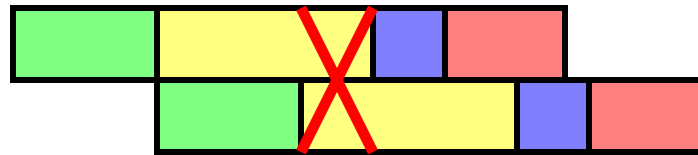
Two execution states:

1. In-progress: changes not visible to outsiders

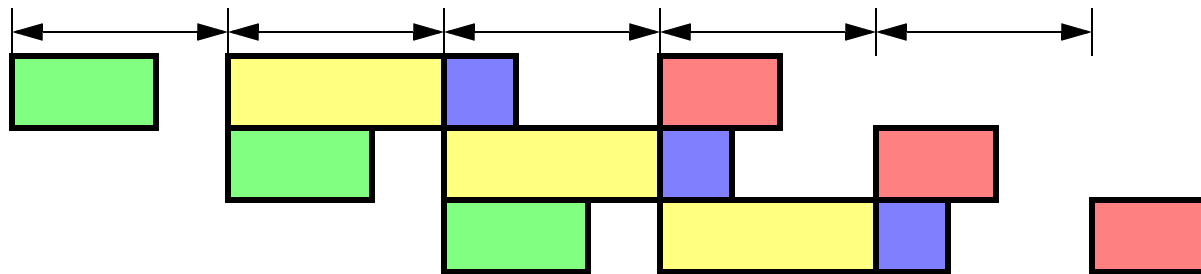
2. Committed: changes visible

Critical Path Determines Clock Cycle

Overlap



Pick Longest Stage



Ideal Speedup

Let **T** be the time to execute an instruction

Instruction execution requires **n** stages, $t_1 \dots t_n$ taking $T = \sum t_i$

$$\text{W/O pipelining: } TR = \frac{1}{T} = \frac{1}{\sum t_i} \quad \text{Latency} = T = \frac{1}{TR}$$

$$\text{W/ n-stage pipeline: } TR = \frac{1}{\max(t_i)} \leq \frac{n}{T} \quad \text{Latency} = n \times \max(t_i) \geq T$$

$$\text{Speedup} = \frac{\sum t_i}{\max(t_i)} \leq n$$

If all t_i are equal, Speedup is n

Ideally: Want higher Performance? Use more pipeline stages



- **We can just add more stages**

Circuits just work

- **Same latency micro-actions**

Perfectly balanced stages

- **Identical micro-actions per instruction**

Must perform the same steps per instruction

- **Independence of micro-actions across instructions**

No need to wait for a previous instruction to finish

No need to use the same resource at the same time

Pipelining Limits

- **After a certain number of stages benefits level off and later they start diminishing**

- **Pipeline utility is limited by:**

1. Implementation

- a. Logic Delay

- b. Clock Skew

- c. Latch Delay

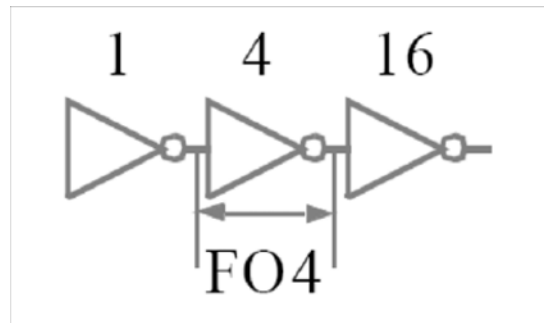
2. Structures

3. Programs

2 & 3 will be called HAZARDS

Parenthetic Topic: Logic Delay FO4

- How does the speed of a gate depend on technology
- Use Fanout of 4 inverter metric

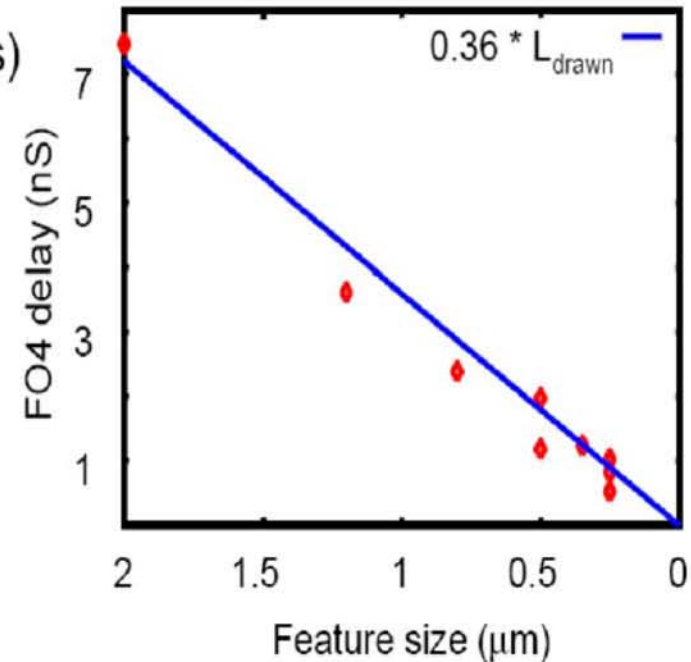


- Measure the delay of an inverted with $C_{out}/C_{in} = 4$
- Divide speed of circuit by speed of FO4 inverter
 - Get circuit delay measured in FO4
 - Metric pretty stable, over process, temp, and voltage

Source: M. Horowitz

FO4 Inverter Delay Under Scaling

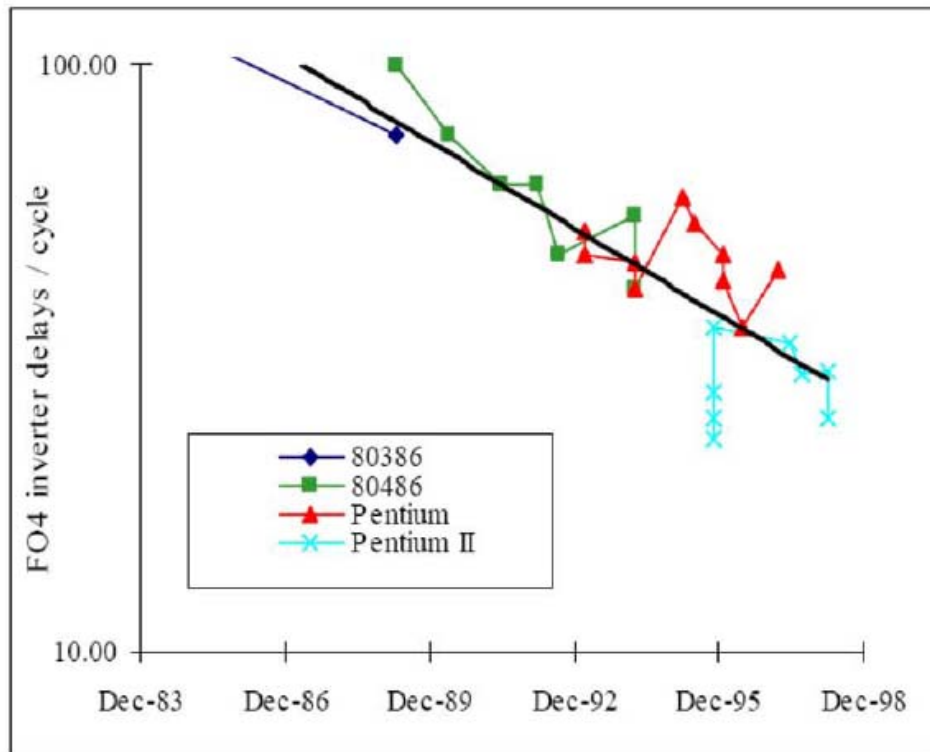
- Device performance will scale
 - FO4 delay has been linear with tech
 - Approximately $0.36 \text{ nS}/\mu\text{m} * L_{\text{drawn}}$ at TT
 - ($0.5 \text{ nS}/\mu\text{m}$ under worst-case conditions)
- Easy to predict gate performance
 - We can measure them
 - Labs have built $0.04\mu\text{m}$ devices
 - Key issue is voltage scaling



Source: M. Horowitz

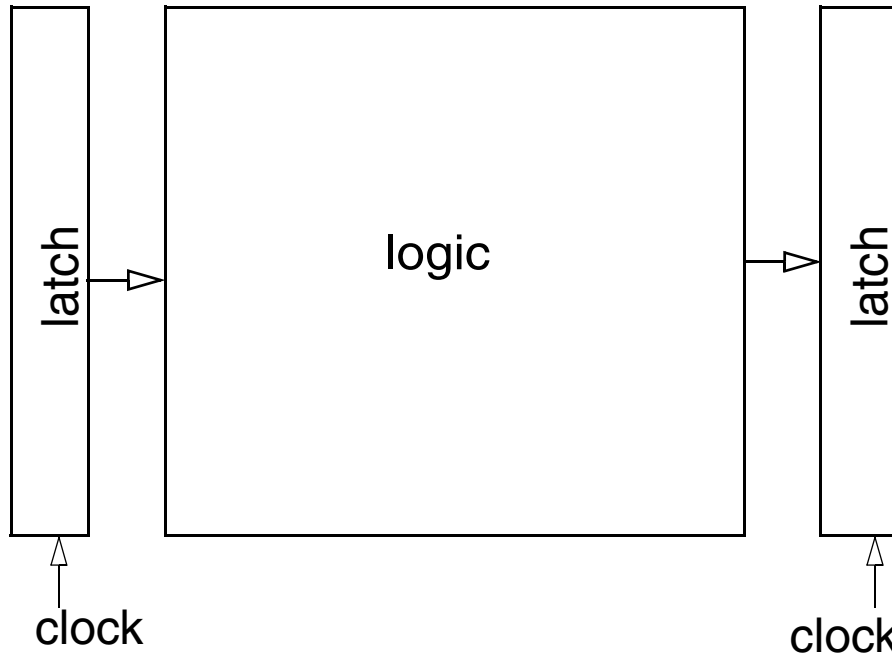
Gates Per Clock

- Clock speed has been scaling faster than base technology
- Number of FO4 delays in a cycle has been falling



- Number of gates decrease 1.4x each generation
- Caused by:
 - Faster circuit families (dynamic logic)
 - Better optimization
- Approaching a limit:
 - <16 FO4 is hard
 - < 8 FO4 is very hard

Pipeline Limits: #1 Logic Delay



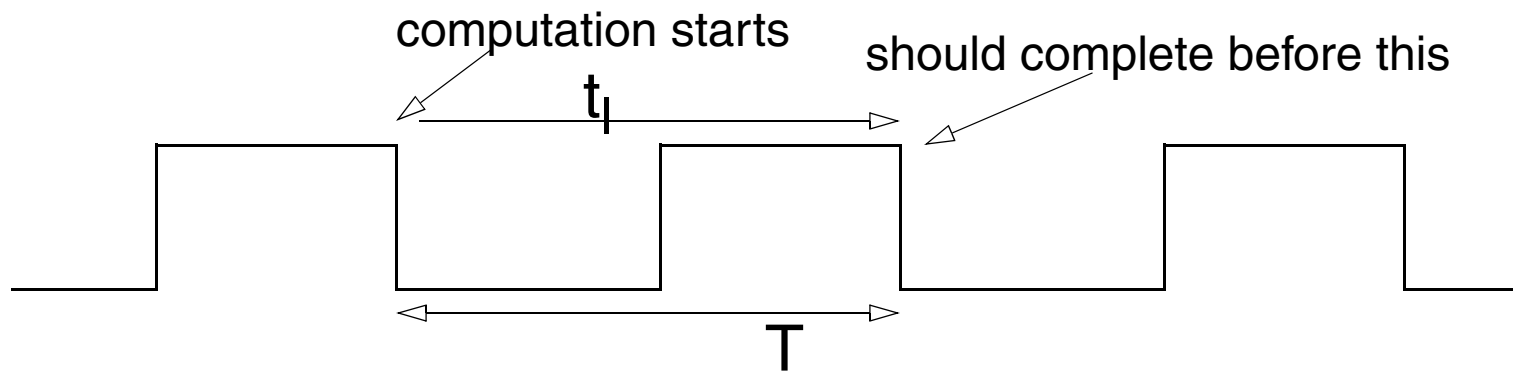
- t_1 = logic block's **worst case delay**

- T = clock period

$$T \geq t_1$$

- **Today's Procs:**

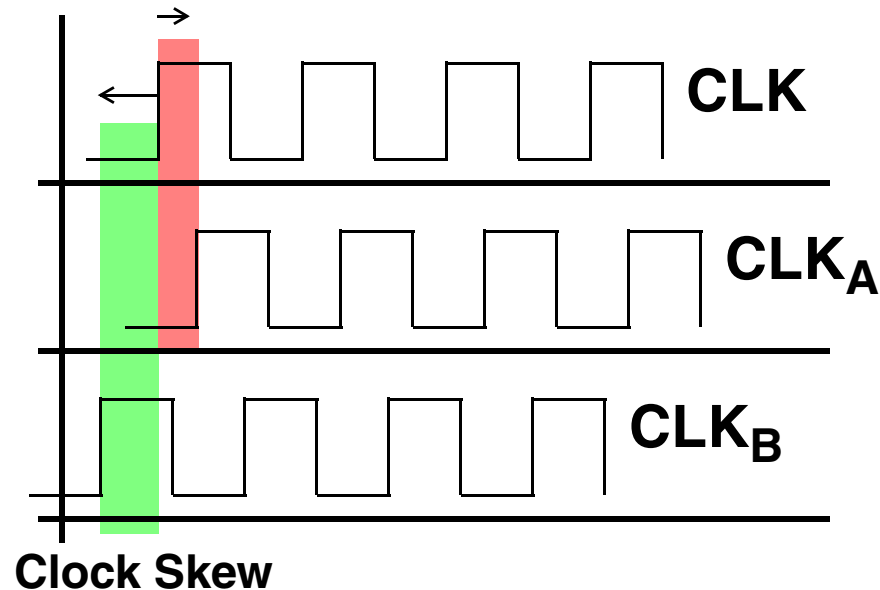
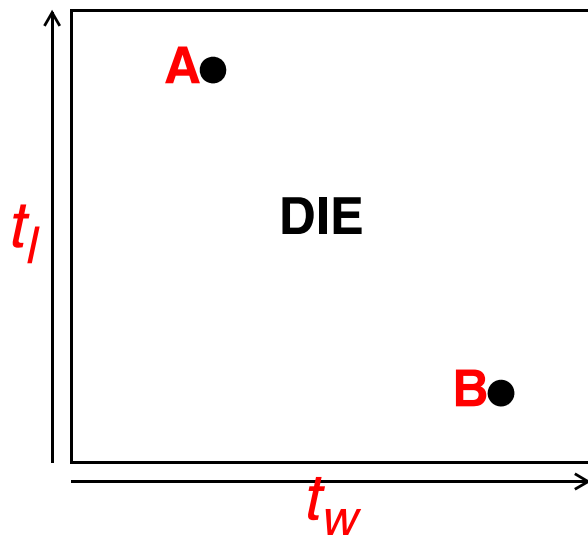
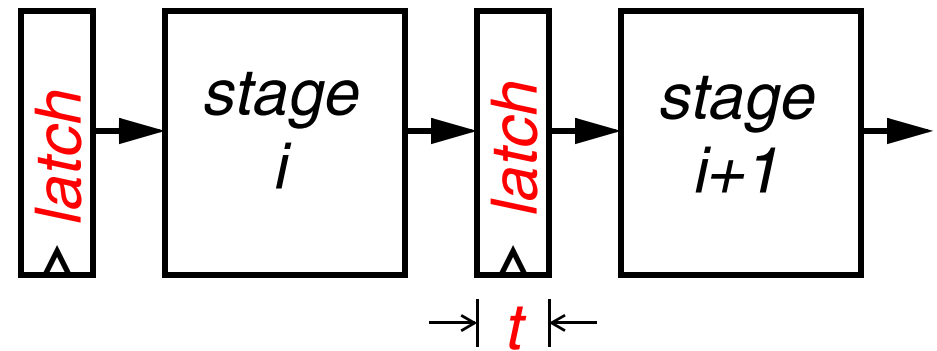
6-12 2-input gates per stage



Pipelining Limits: #2 Clock Skew

CLOCK SKEW

- Clock takes time to travel
- Arrival depends on distance/load
- Skew amongst different connections
- Creates additional constraints

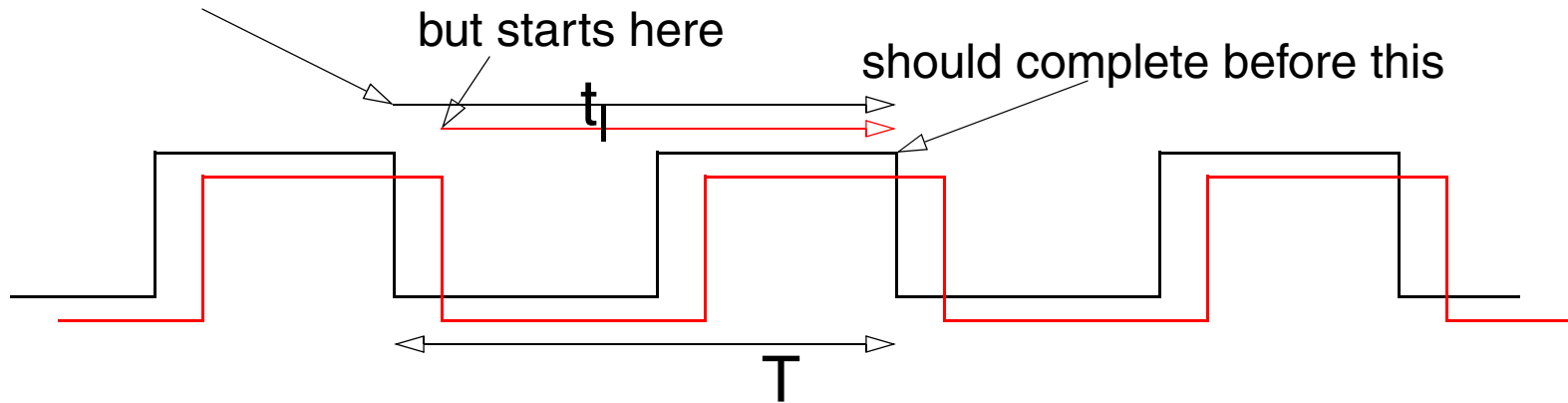


Clock Skew contd.

computation would have started here

but starts here

should complete before this



- **Worst case scenario:**

- **Output from stage with late clock feeds stage with early clock**

Pipelining Limits: #3 Latch Overhead

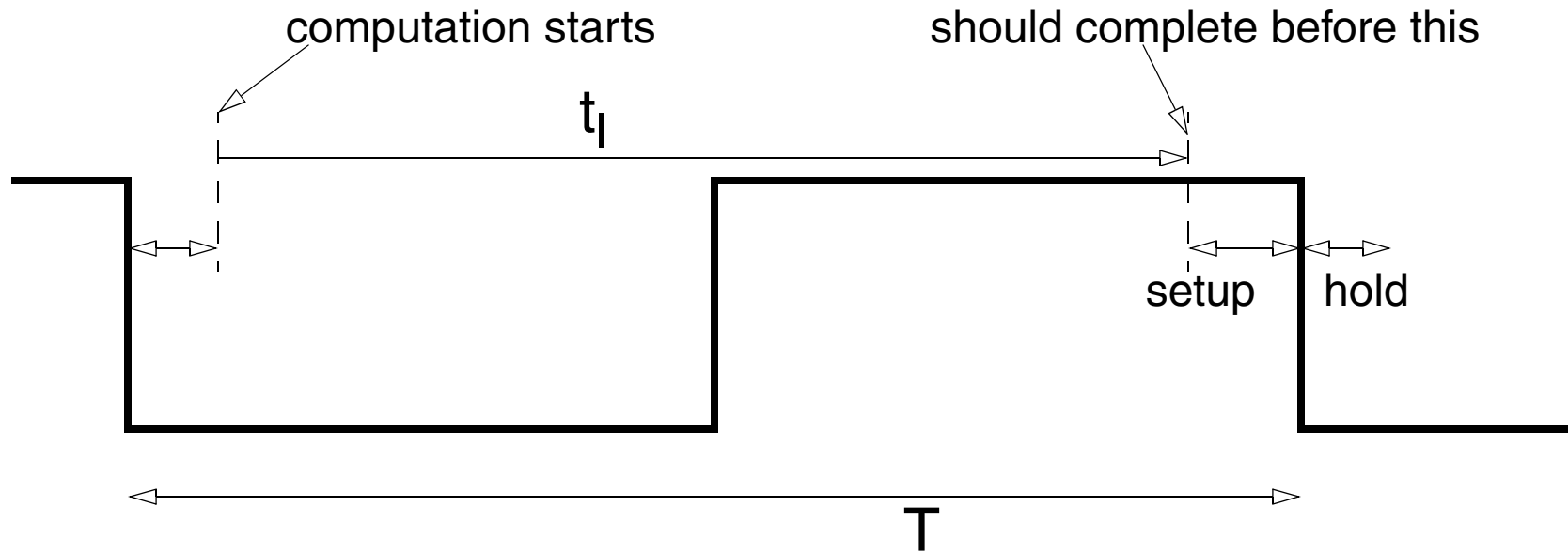
- **Latch takes time**

- **Setup Time**

data must stay stable before clock edge

- **Hold Time**

data must stay stable after clock edge



Impact of Clock Skew and Latch Overheads

let X be extra delay per stage for

- latch overhead
- clock/data skew

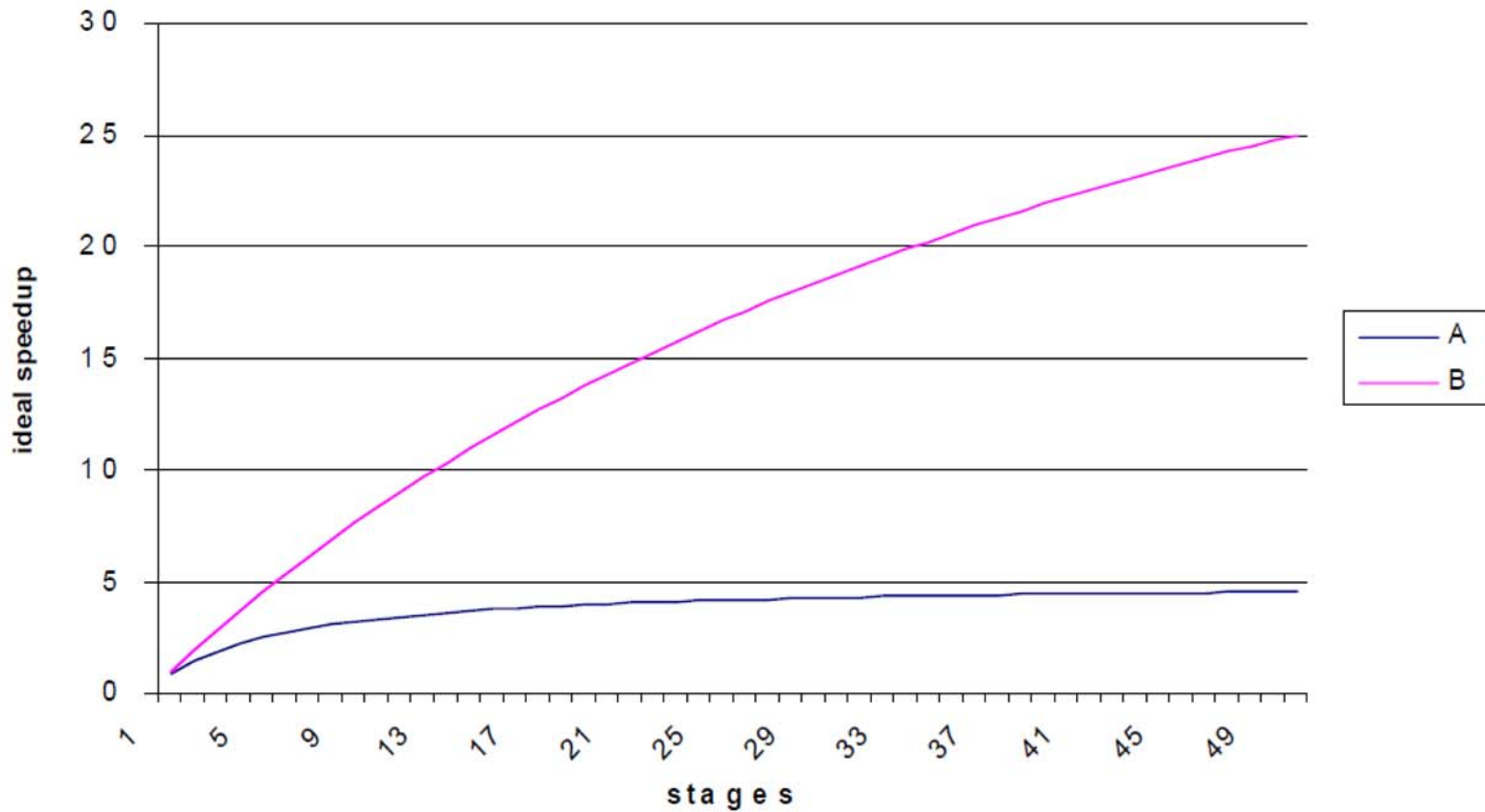
X limits the useful pipeline depth

With n -stage pipeline (all t_i equal) ($T = n \times t$)

- throughput = $\frac{1}{X+t} < \frac{n}{T}$
- latency = $n \times (X+t) = n \times X + T$
- speedup = $\frac{T}{(X+t)} \leq n$

Real pipelines usually do not achieve this due to **Hazards**

Ideal speedup with per stage overheads



T = 500. A: X=100, B: X=10

Cost/Performance Tradeoff

Cost = Stages x Latch_Cost + Base Cost

Performance = 1 / (Overhead_{per_stage} + Cycle / Stages)

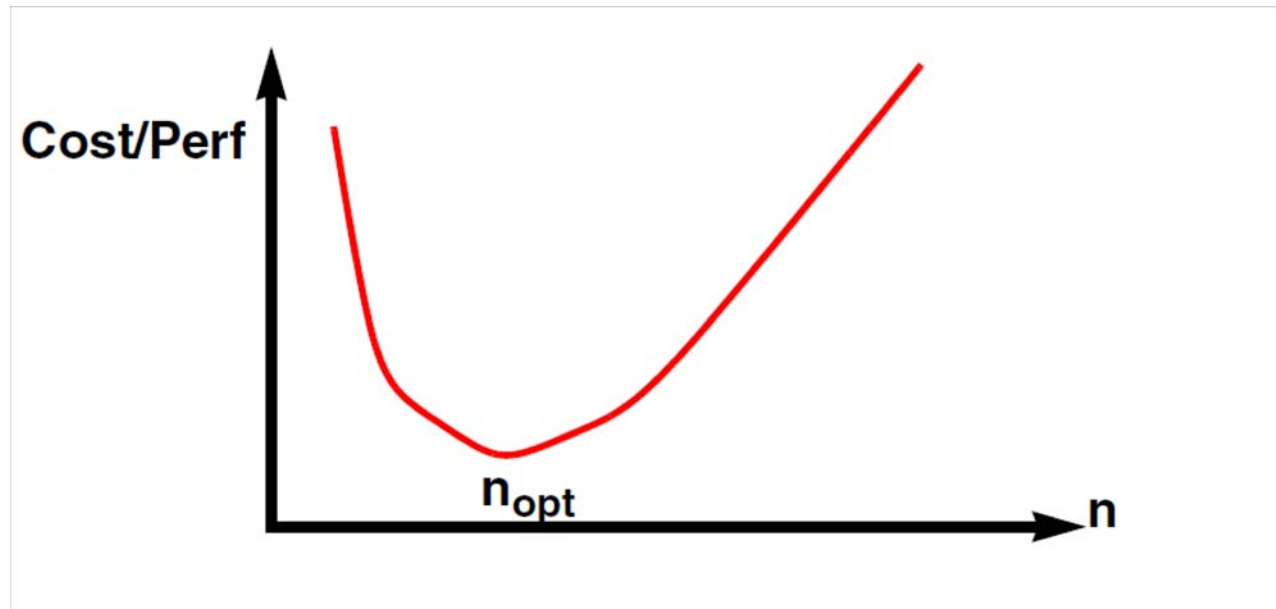
Source: P. Kogge: The architecture of pipelined computers.

**Cost / Performance = LatchCost x CycleTime +
BaseCost x Overhead +
LatchCost x Overhead x Stages +
BaseCost x CycleTime / Stages**

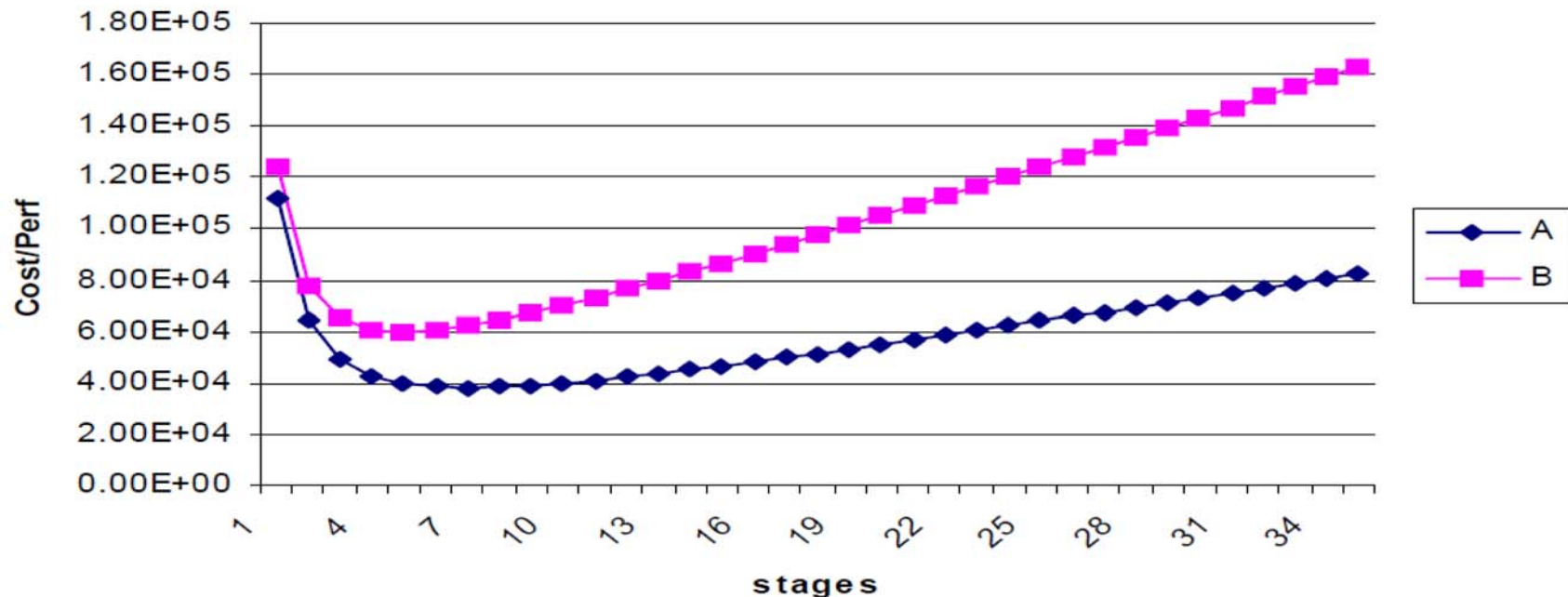
**Stages_{Optimal} = Sqrt (BaseCost x CycleTime /
(LatchCost x Overhead))**

Optimal Number of Stages

n = stages, source C. Kozyrakis



Cost/Performance trade-off



Cycle = 500, BaseCost = 200

A: Overhead = 10, LatchCost = 20 / B: Overhead = 20 & LatchC = 40

As latchcost and overhead become larger w/ respect to base cost, the number of stages must be reduced -- Deep pipelines no good.

Pipeline Idealism

- **We can just add more stages**

Circuits just work

- **Same latency micro-actions**

Perfectly balanced stages

- **Identical micro-actions per instruction**

Must perform the same steps per instruction

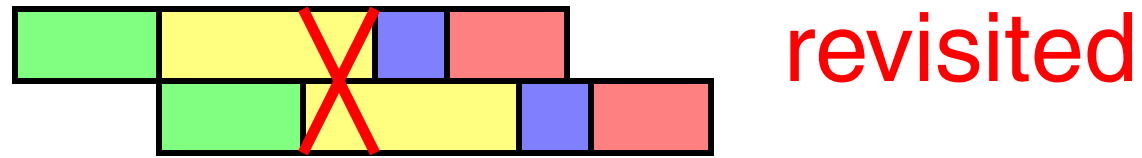
- **Independence of micro-actions across instructions**

No need to wait for a previous instruction to finish

No need to use the same resource at the same time

Skew can be your friend

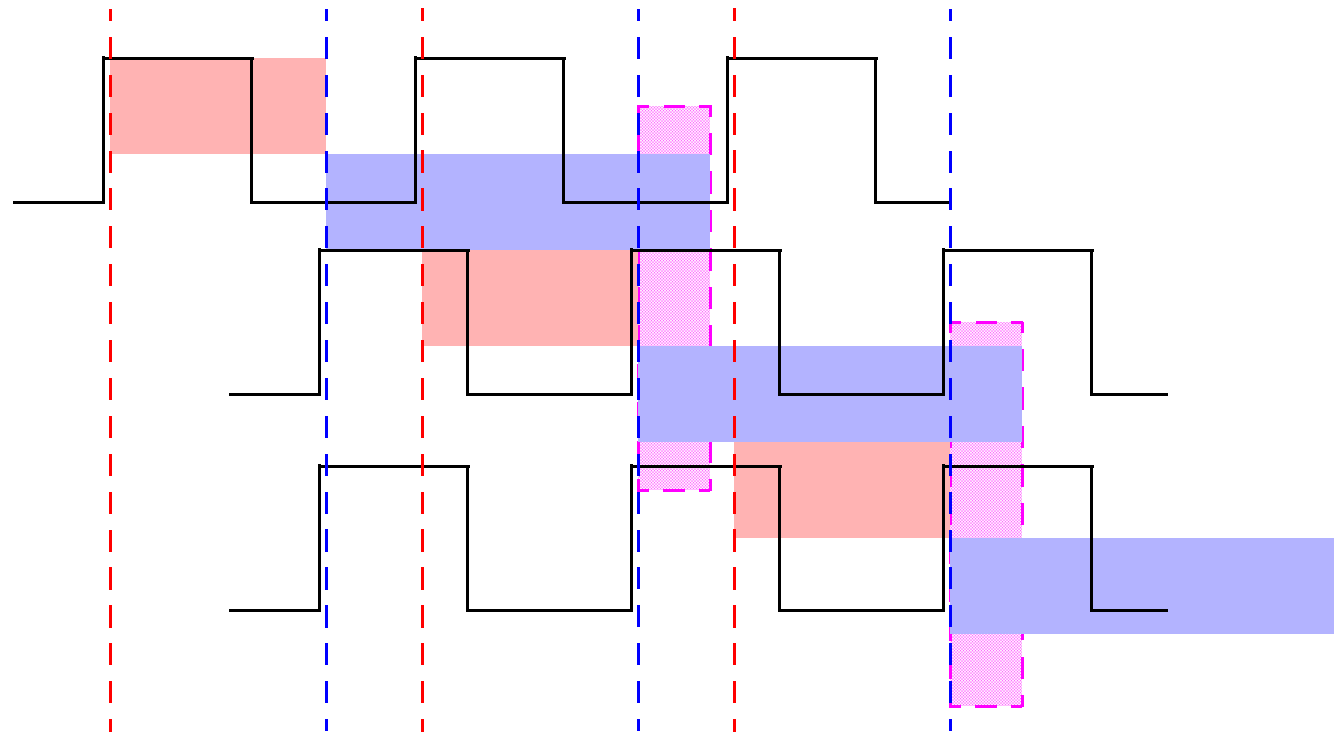
time borrowing



i IN

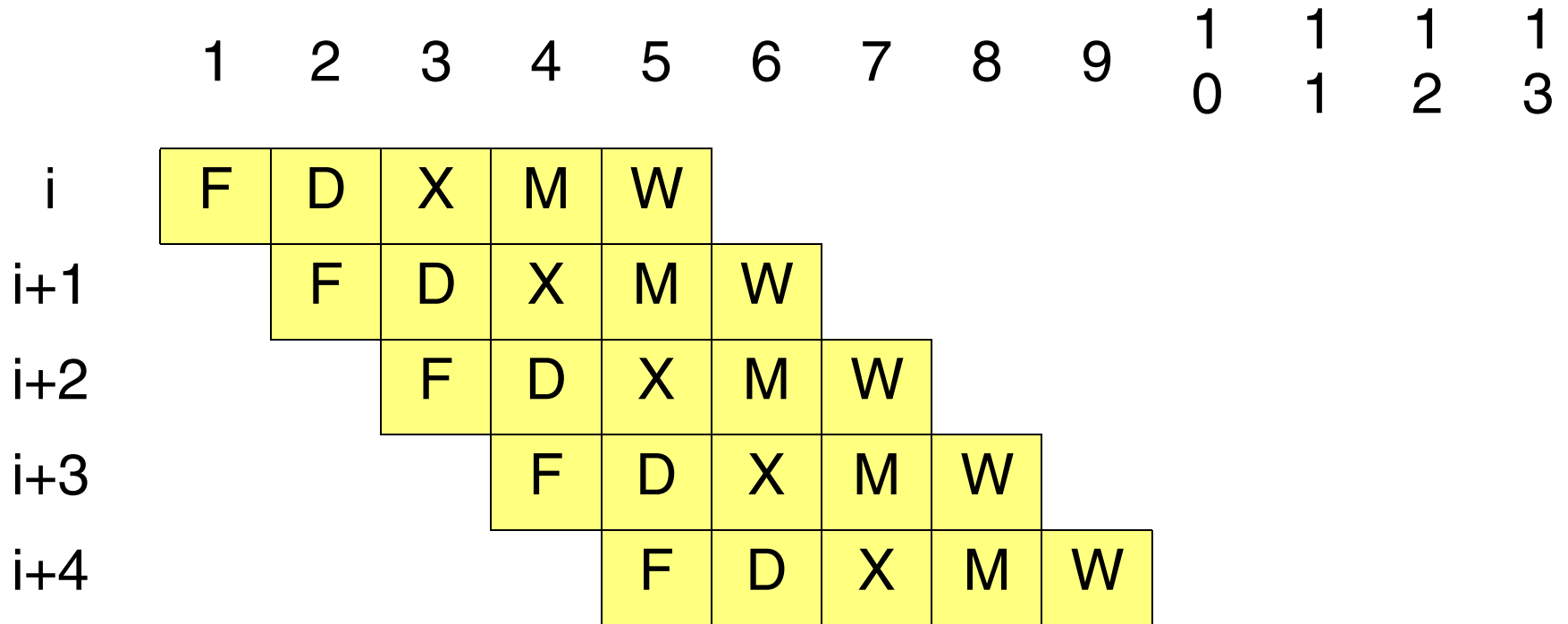
i Out/ $i+1$ IN

$i+1$ Out



Simple pipelines

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



Classic 5-stage Pipeline

MIPS micro-actions per instruction

- **integer/logic operations**

add \$1, \$2, \$3 --> read 2 regs, write one

- **branches**

beq \$1, \$2, LALA --> read 2 regs, compare, change PC

- **load/stores**

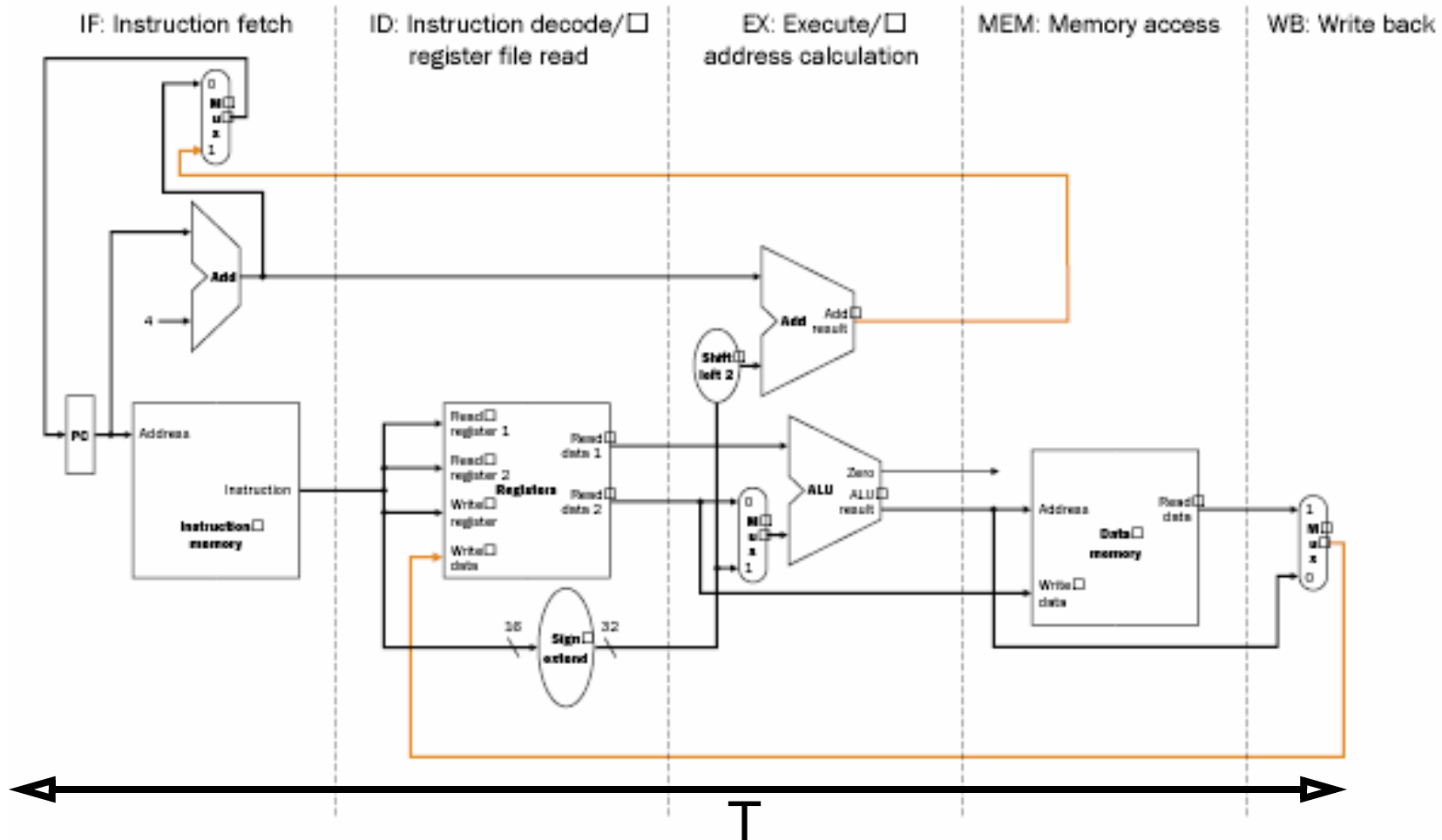
lw \$1, 10(\$3) --> read 1 reg, add, read memory, write reg

sw \$1, 10(\$3) --> read 2 regs, add, write memory

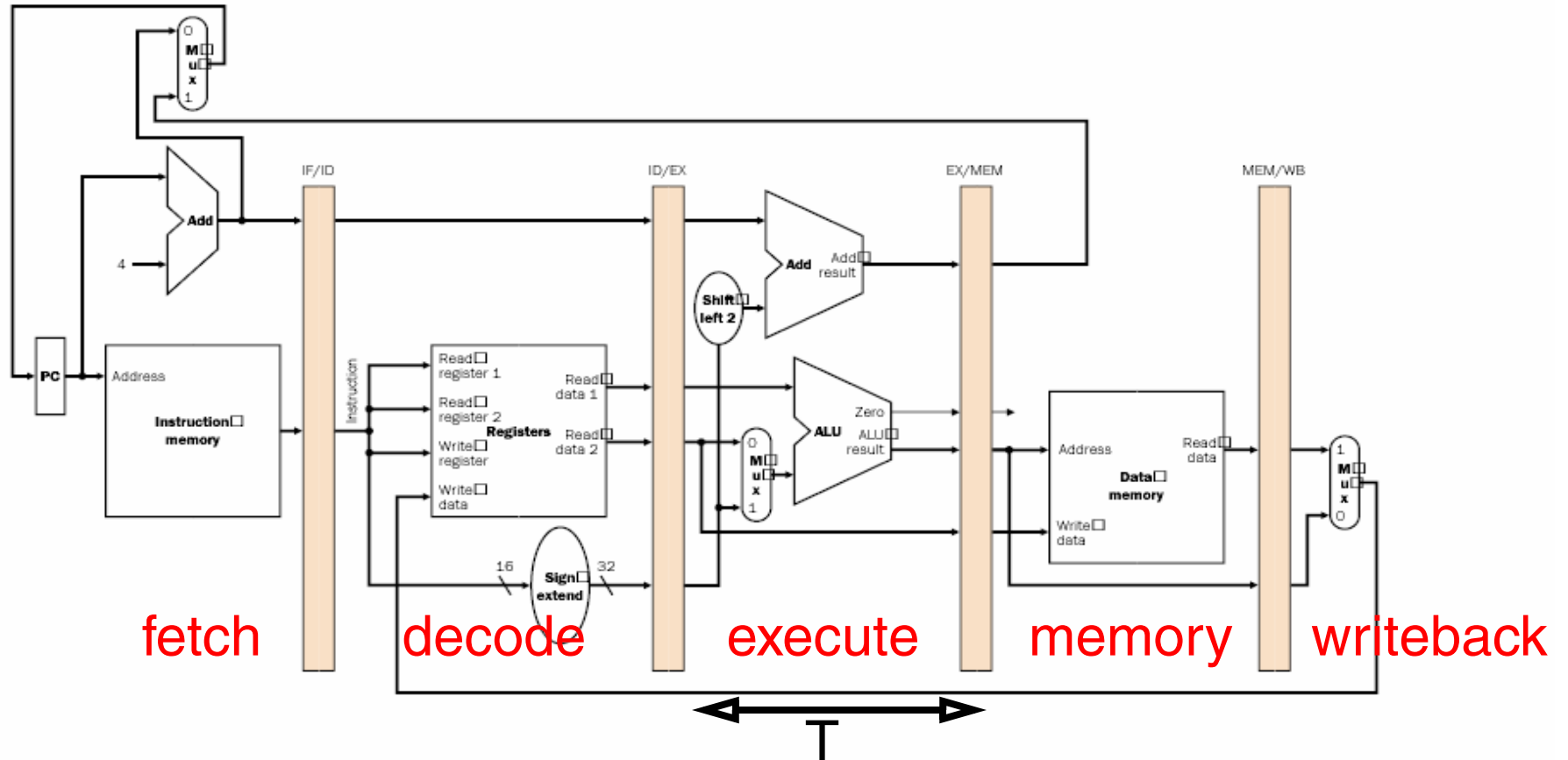
- **special ops: syscall, jumps, call/return**

read at most 1 reg, write at most 1 reg, change PC

Non-Pipelined Implementation



Pipelined Implementation



Pipelined Implementation: Ideally 5x performance

Hazards

Hazards

- conditions that lead to incorrect behavior if not fixed

Structural Hazard

- two different instructions use same resource in same cycle

Data Hazard

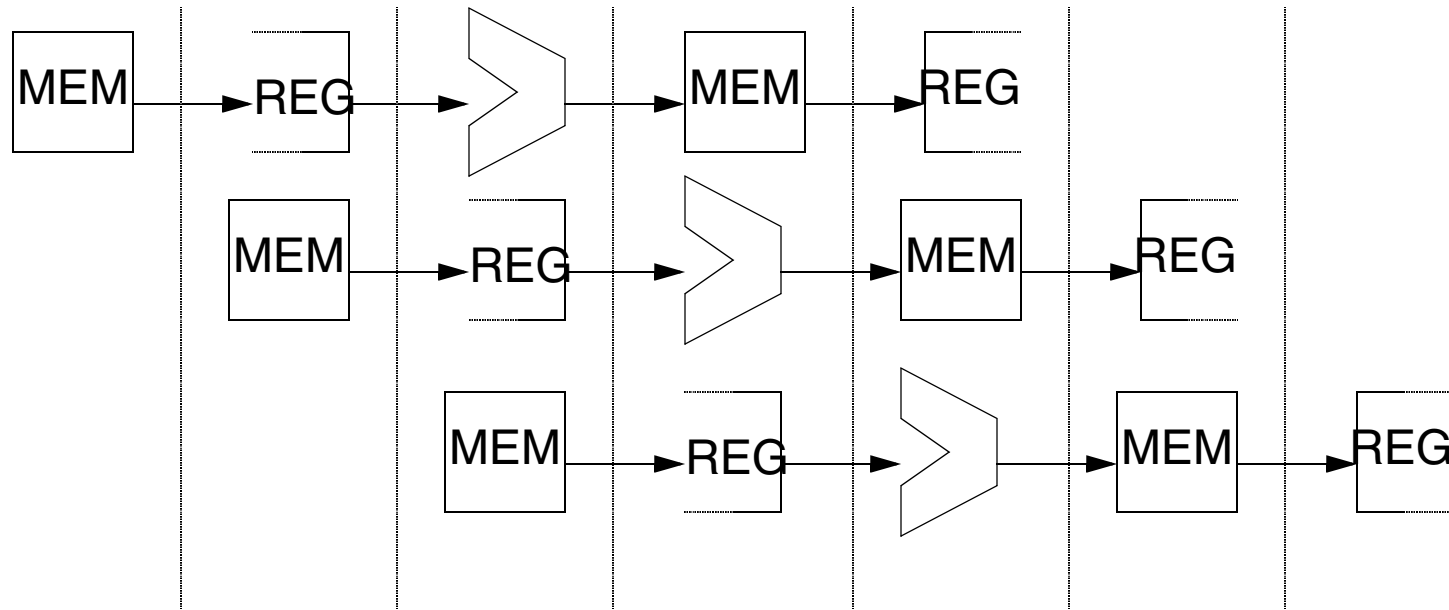
- two different instructions use same storage
- must appear as if the instructions execute in correct order

Control Hazard

- one instruction affects which instruction is next

Pipelining as Datapaths in Time

Time

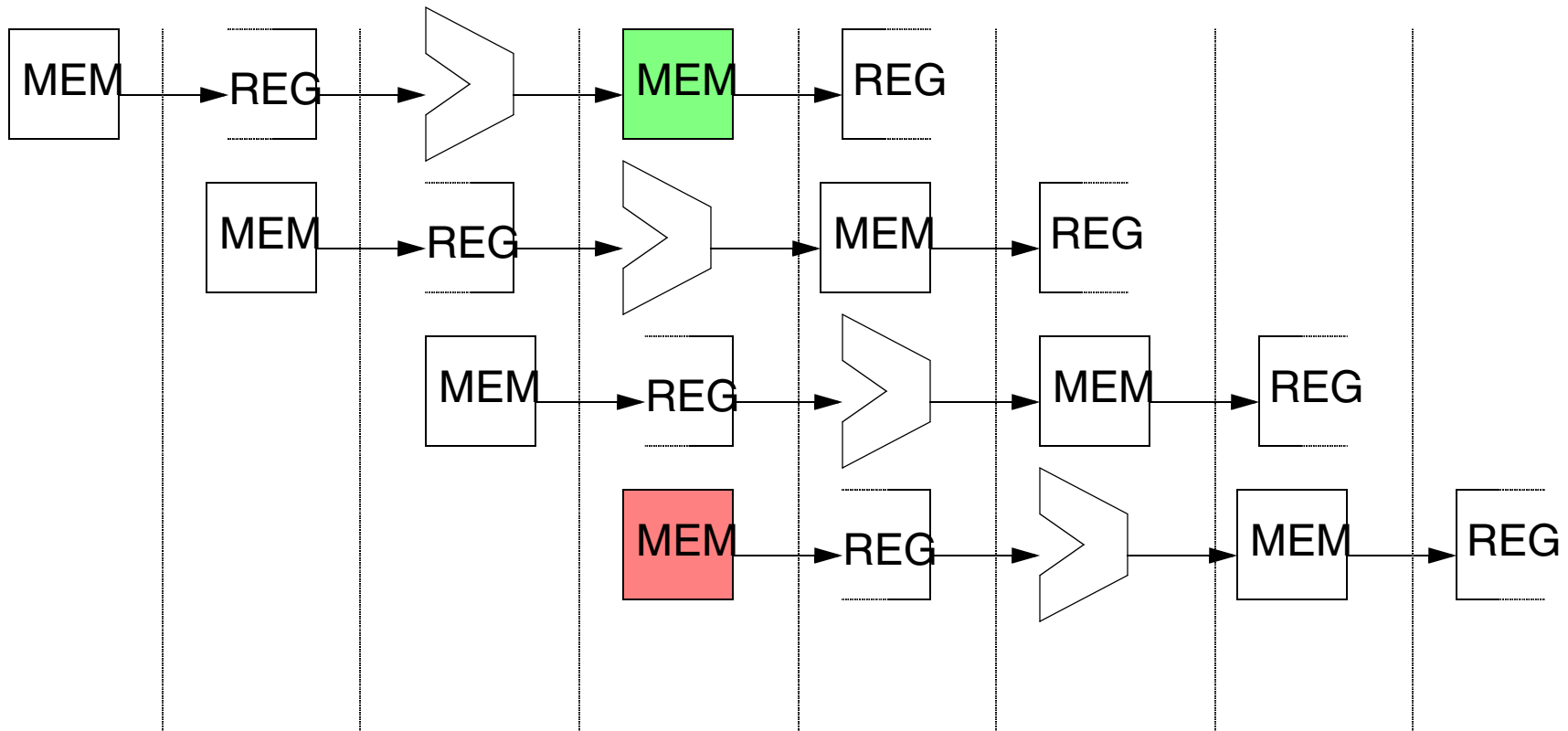


Useful for identifying hazards

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

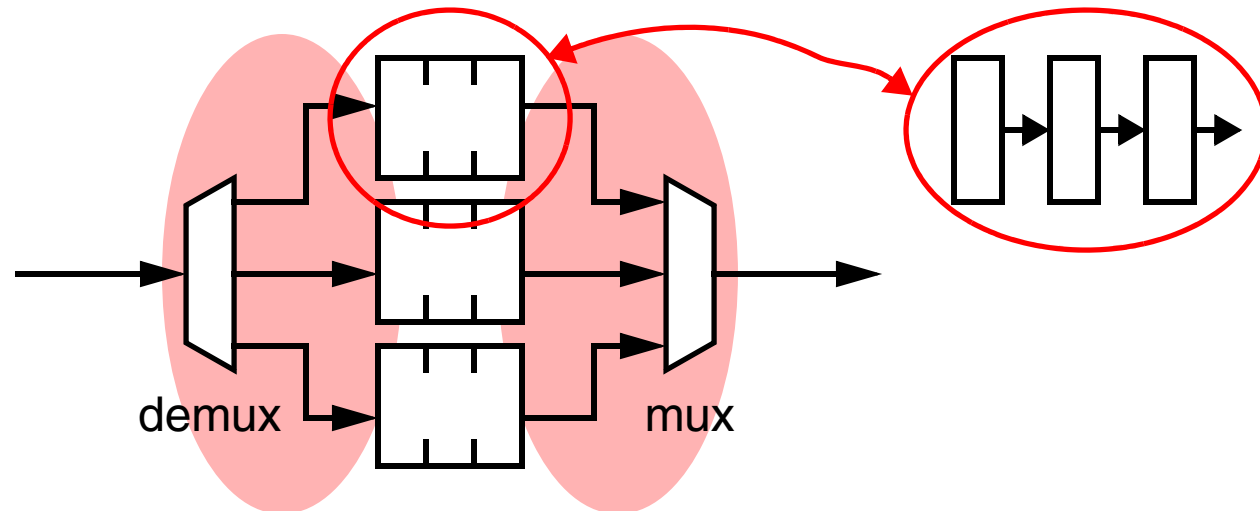
Pipeline Hardware Resource:

- 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

RISC = Reduced Instruction Set Architecture

Today: Load/Store Architectures, not necessarily few instructions

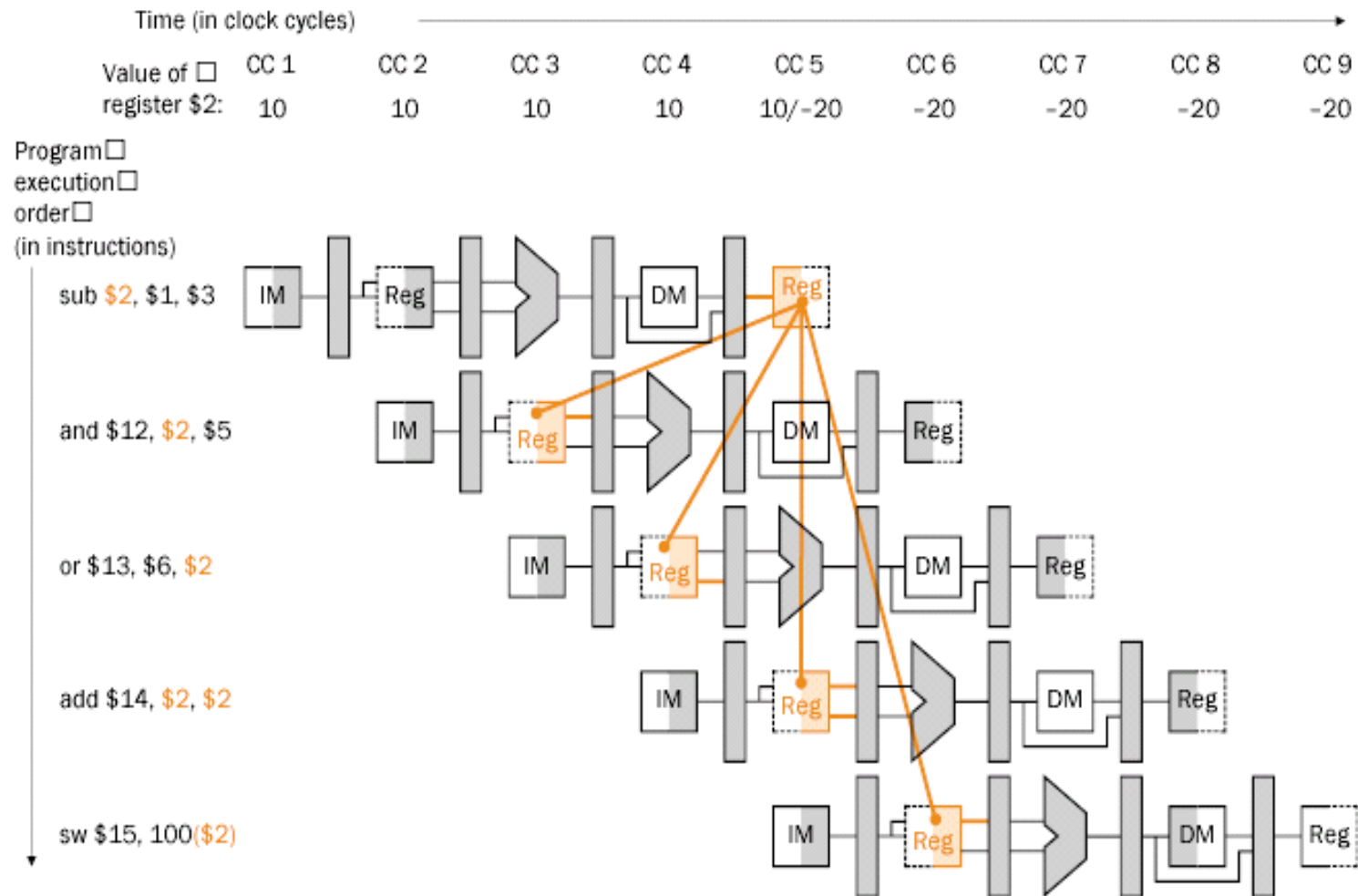
Data Hazards

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

```
add r1, r2, --
sub r2, --, r1
add r3, r1, --
or r1, --, --
```

- *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)

Dependences in a 5-Stage Pipeline



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

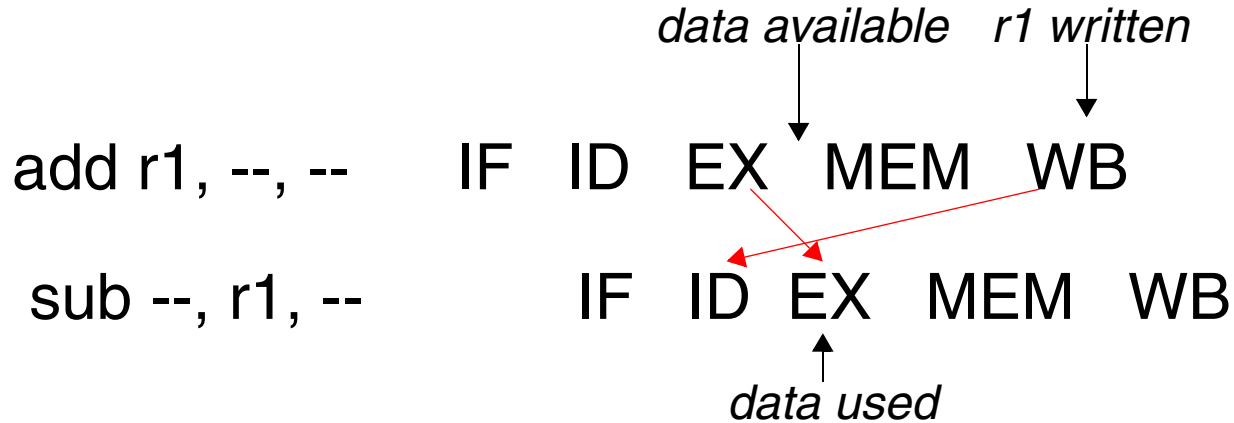
+ low cost, simple

– 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

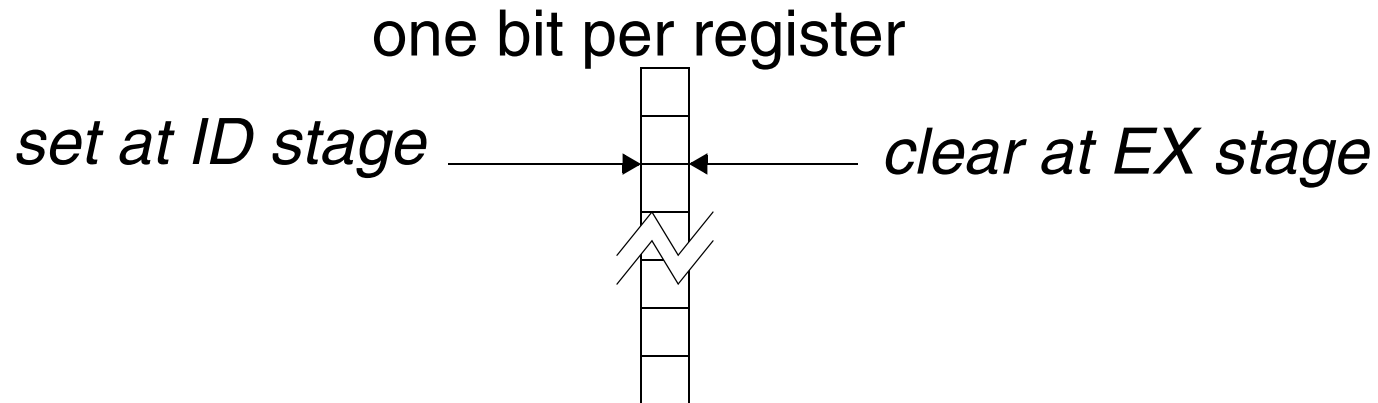
- if $\text{insn}(\text{EX})$ uses immediate then select IMM
- else if $\text{rs2}(\text{EX}) == \text{rd}(\text{MEM})$ then ALUOUT(MEM)
- else if $\text{rs2}(\text{EX}) == \text{rd}(\text{WB})$ then ALUOUT(WB)
- else select B

Stall Method w/ Bypass paths

- Use register reservation bits:

is the result available the next cycle

Only loads set the bits --> instructions that cannot be bypassed



loads reserve at ID stage
release at EX stage

check source Reg bit
stall if reserved

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

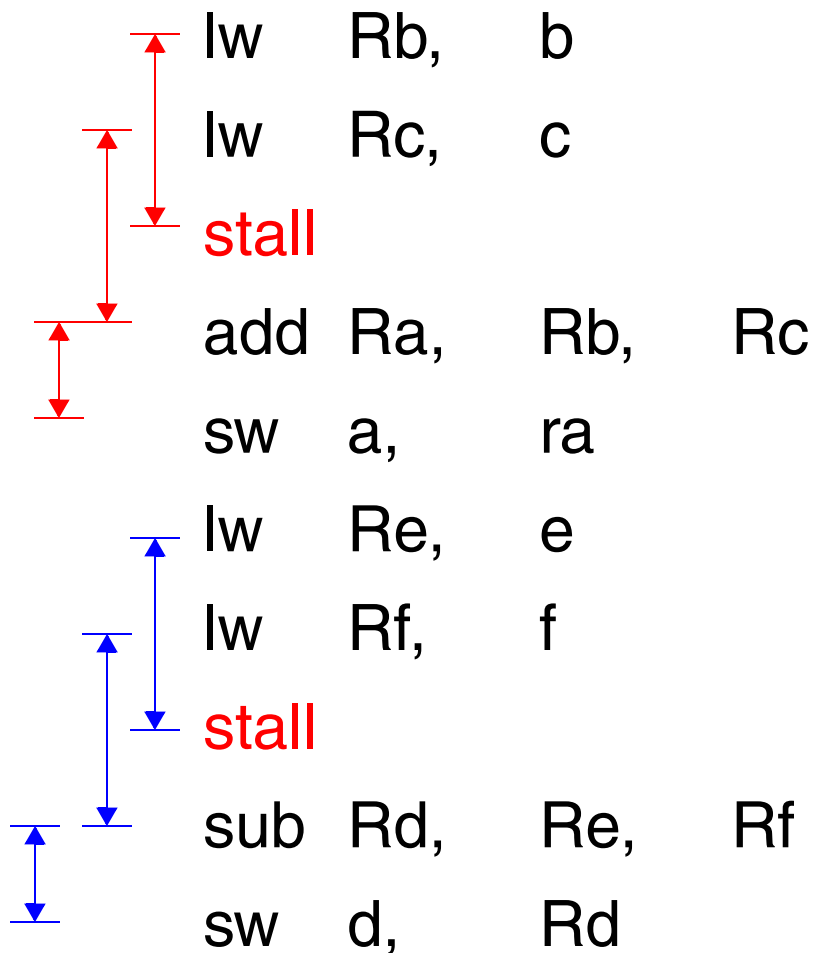
Try to separate stall logic from bypass logic

- avoid irregular bypasses

Pipeline Scheduling - Compilers can Help

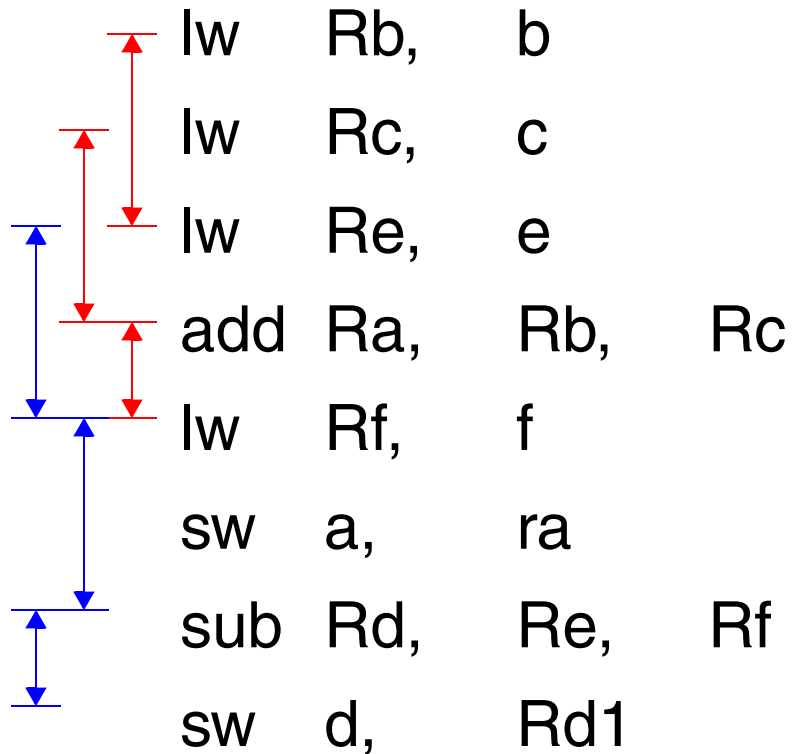
Instructions scheduled by compiler to reduce stalls

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

WAR

```
add r1, r2, --  
sub r2, --, r1  
or r1, --, --
```

Not possible in MIPS - read early write late

Consider late read then early write: delayed reg reads for stores

ALU ops writeback at EX stage

MEM takes two cycles and stores need source reg after 1 cycle

sw r1, --	IF	ID	EX	MEM1	MEM2	WB	
add r1, --, --		IF	ID	EX	MEM1	MEM2	WB

also: MUL --, 0(r2), r1

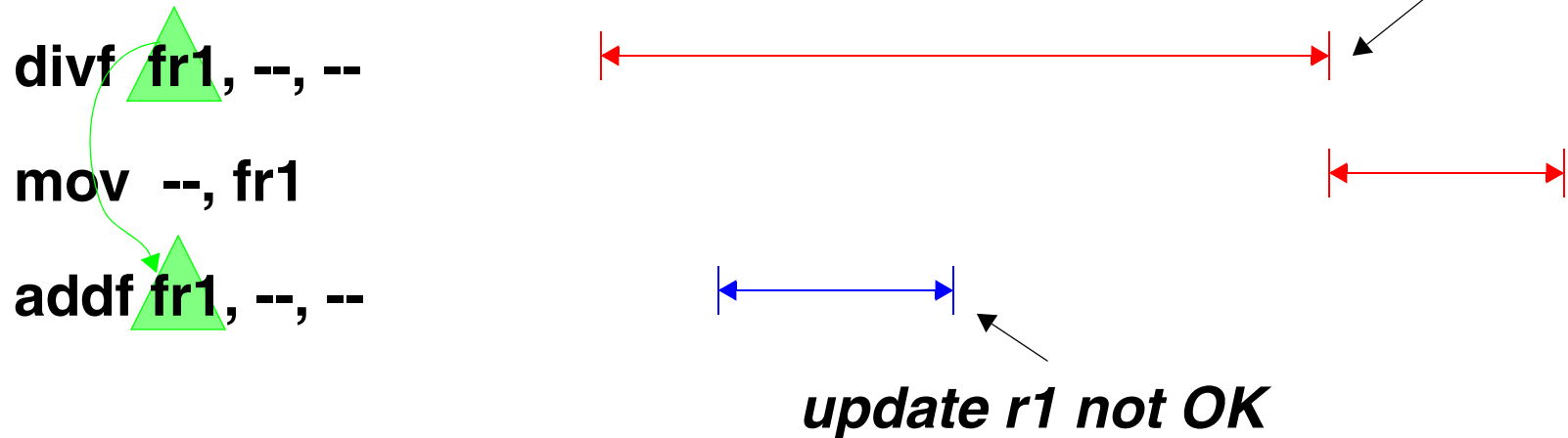
lw --, (r1++)

Other Data Hazards: WAW

WAW

Not in MIPS : register writes are in order

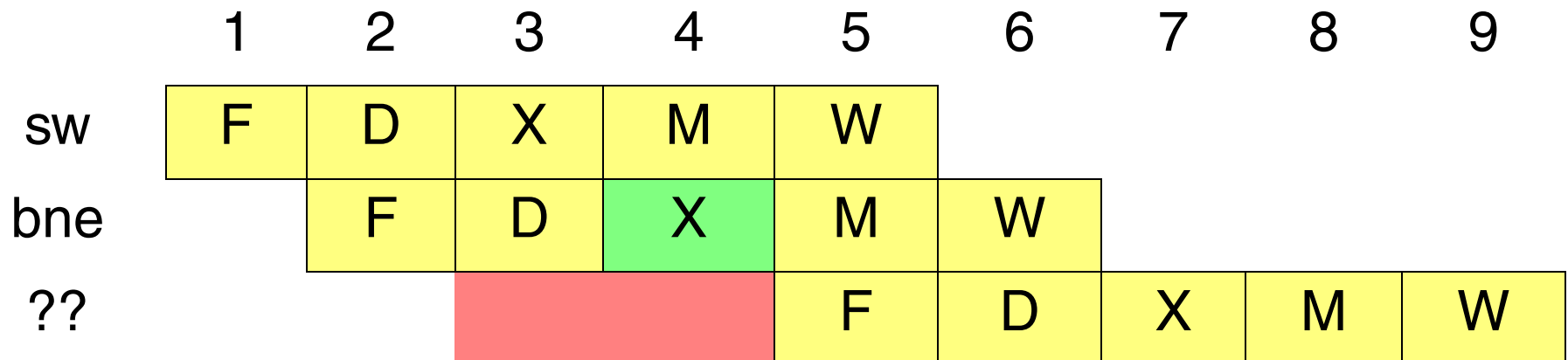
consider slow then fast operation



Control Hazards

When an instruction affects which instruction execute next
or changes the PC

- sw \$4, 0(\$5)
- bne \$2, \$3, loop
- sub -, - , -



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*
- ***RULE of THUMB: 1/5 instructions is a BRANCH***

Handling Control Hazards

Move control point earlier in the pipeline

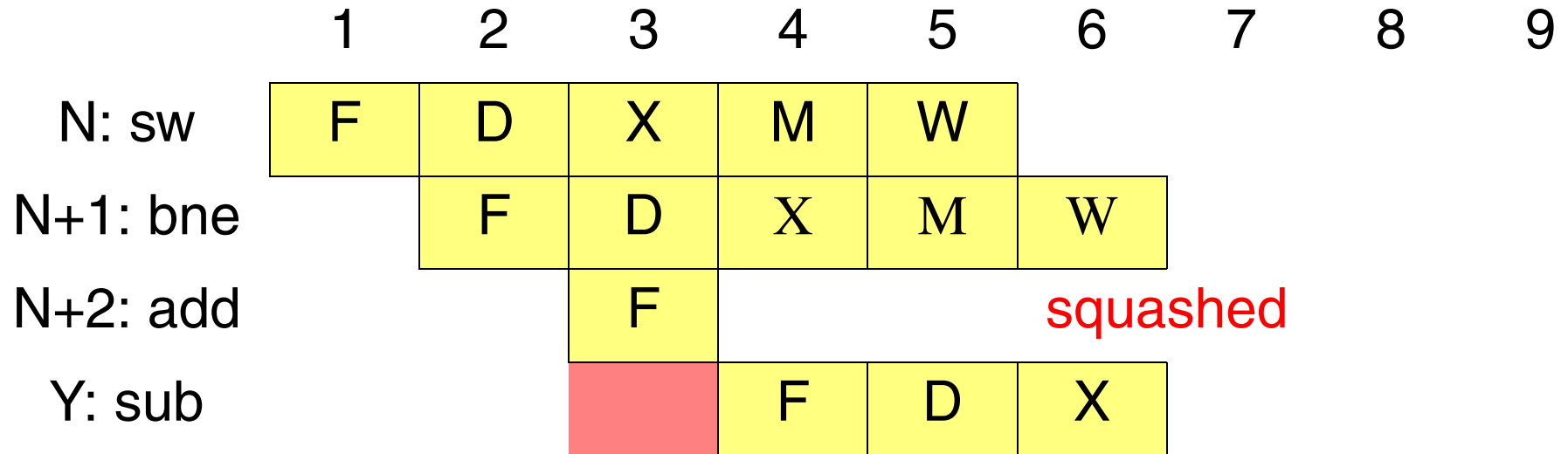
- Find out whether branch is taken earlier
- Compute target address fast

Both need to be done

e.g., in ID stage

- $\text{target} := \text{PC} + \text{immediate}$
- if (Rs1 op 0) $\text{PC} := \text{target}$

Handling Control Hazards



Implies only one cycle bubble but

- special PC adder required

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

- MIPS 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

this is the interesting column



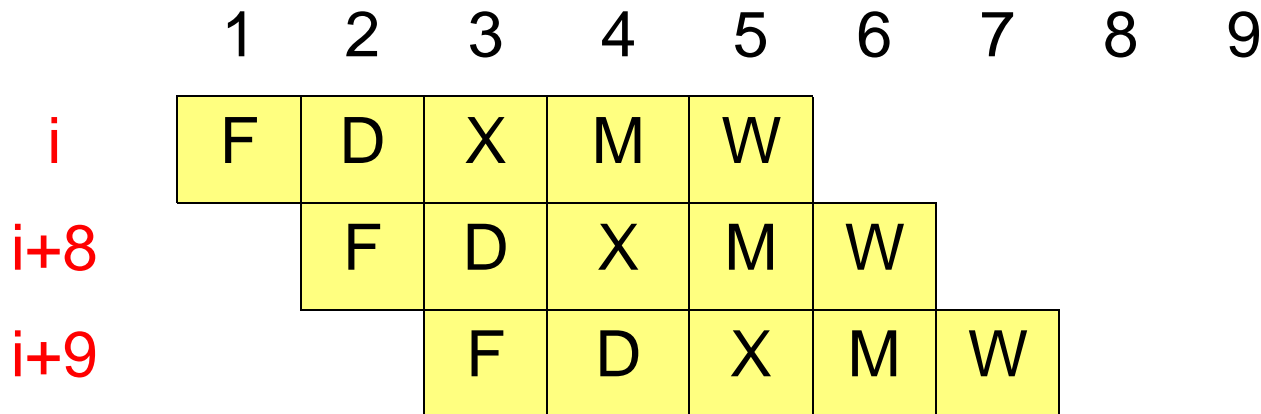
	1	2	3	4	5	6	7	8	9
i	F	D	X	M	W				
$i+1$		F	D	X	M	W			
$i+2$			F	D	X	M	W		

if taken then squash (aka abort or rollback)

- will work only if no state change until branch is resolved
- Simple 5-stage Pipeline, e.g., DLX - ok - why?
- Other pipelines, e.g., VAX - autoincrement addressing?

Handling Control Hazards

Predict taken always



For DLX must know target before branch is decoded

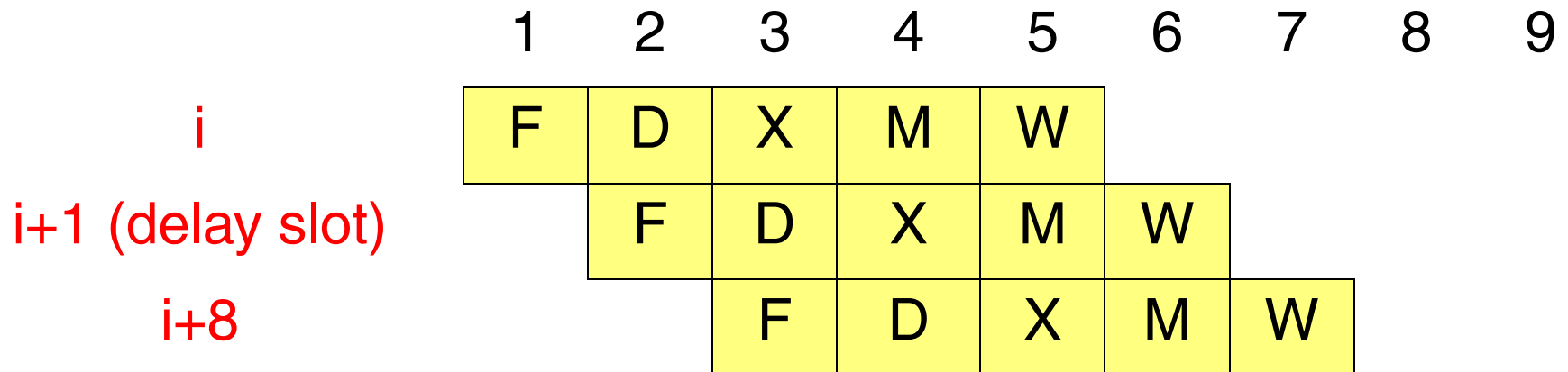
- can use prediction
- special hardware for fast decode

Execute both paths - hardware/memory b/w expensive

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)



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

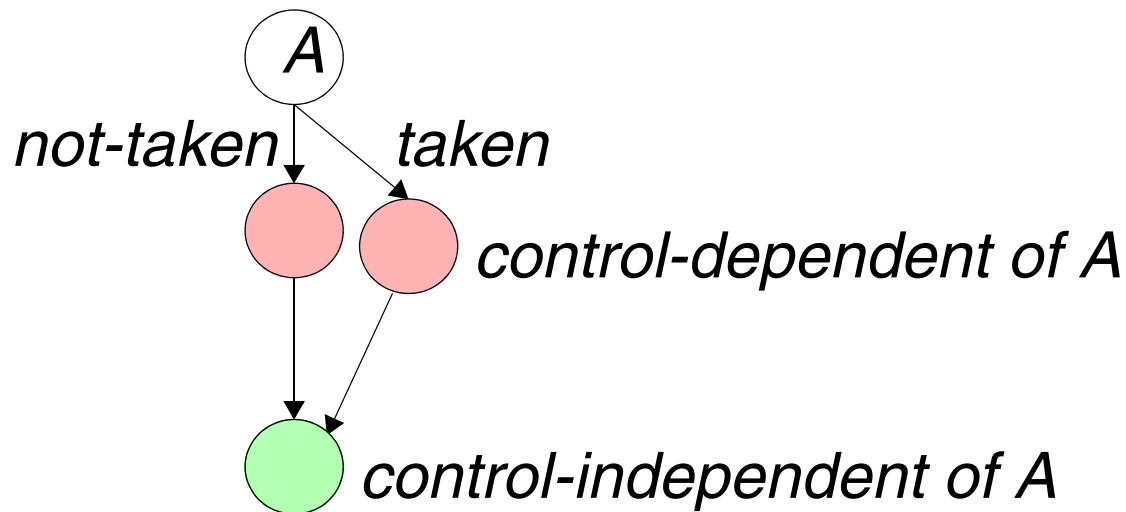
Fill from fall-through (not-taken path)

- when? if safe to execute instruction
- helps? when not-taken

Filling in Delay Slots cont.

From Control-Independent code:

that's code that will be *eventually* visited no matter where the branch goes



Nullifying or Cancelling 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 H&P

- 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	?	?	?

Precise 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

Will return to this soon

First let's consider a simple 5-stage pipeline

Handling Interrupts

Precise interrupts (sequential semantics)

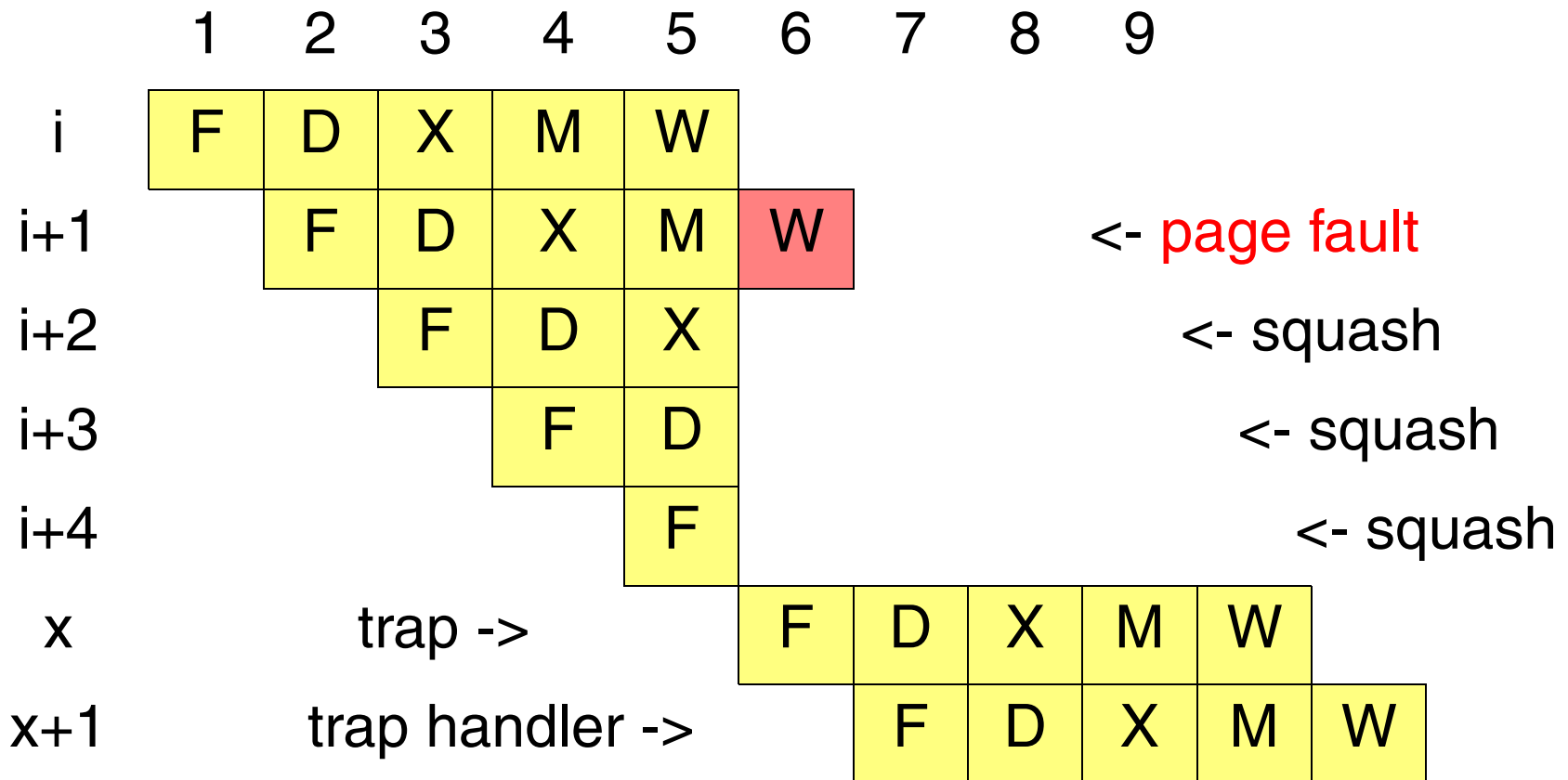
- Complete instructions before the offending instr
- Squash (effects of) instructions after
- Save PC (& next PC with delayed branches)
- Force trap instruction into IF

Must handle simultaneous interrupts

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

Interrupts: Data Memory Page Fault

E.g., data page fault



Interrupts

Preceding instructions already complete

Squash succeeding instructions

- prevent them from modifying state (registers, CC, memory)

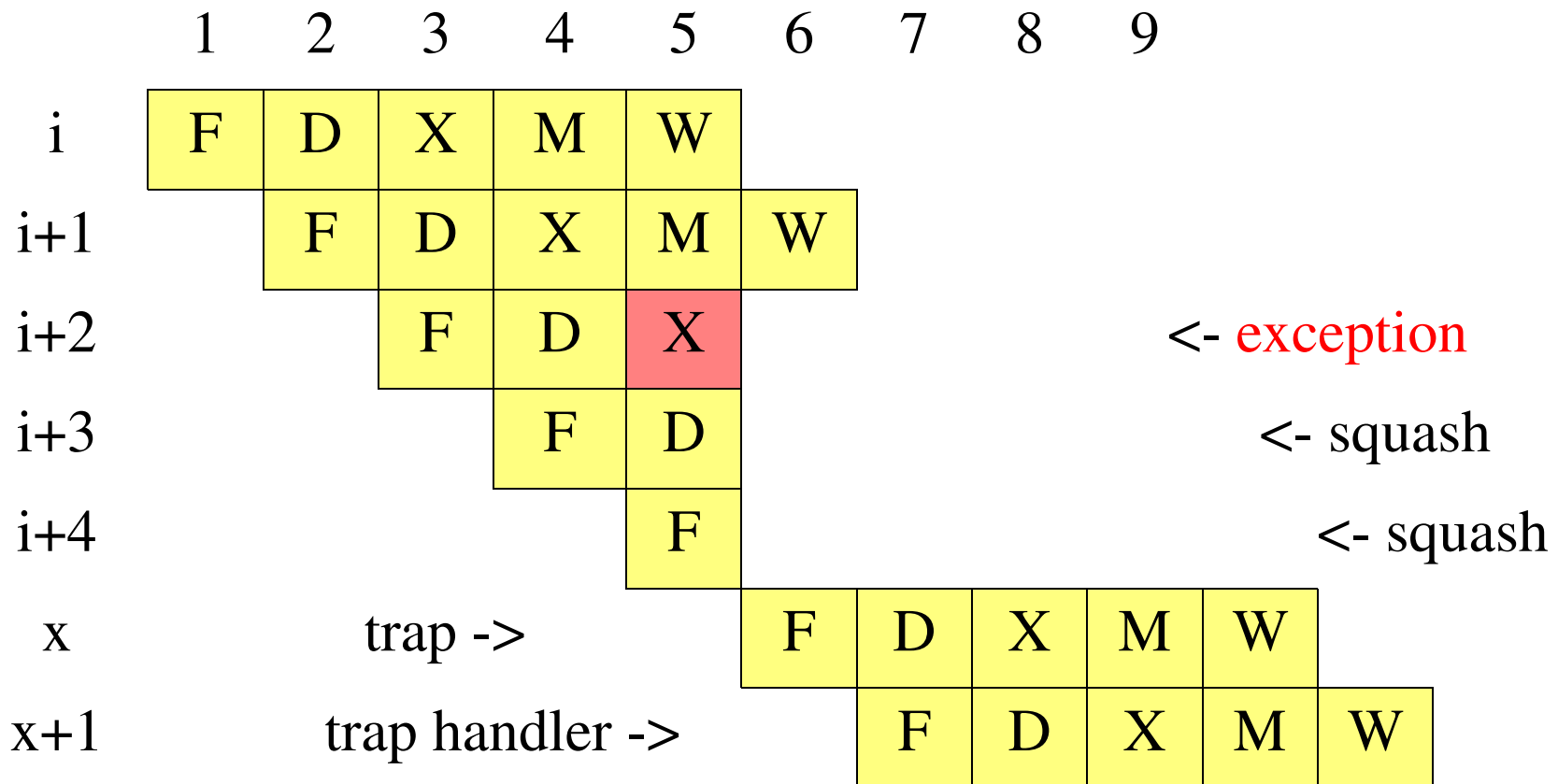
trap instruction jumps to trap handler

hardware saves PC in IAR

trap handler must save IAR

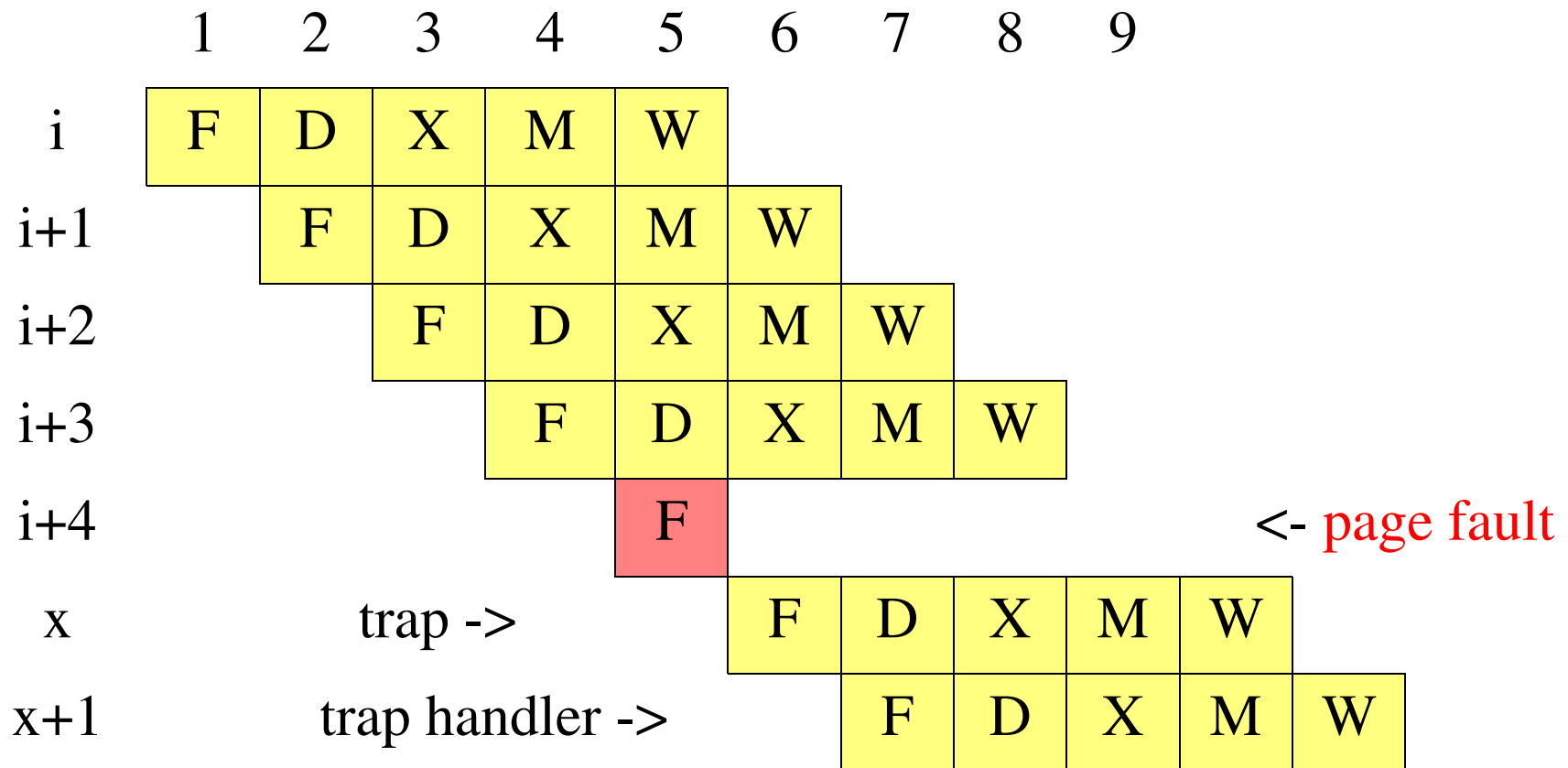
Interrupts: Arithmetic Exception

E.g., arithmetic exception



Interrupts: Instruction Page Fault

E.g., Instruction fetch page fault



Interrupts

Let preceding instructions complete

No succeeding instructions

What happens if $i+3$ causes a data page fault?

Out-of-Order 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				
i+1		F	D	X	M	W			
i+2			F	D	X	M	W		
i+3				F	D	X	M	W	

page fault (Mem)

page fault (fetch)

Out-of-Order Interrupts

Post interrupts: **In-order handling**

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

Handle immediately: **out-of-order handling**

- not fully precise
- interrupt may occur in order different from sequential CPU
- may cause implementation headaches
- Violates Sequential Model
- But offers high performance.

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 must perform well
- 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				
fp/				F	D	E/	E/	E/	E/	M	W
int					F	D	EX	M	W		
fp/						F	D	--	--	E/	E/
fp*							F	--	--	D	X

(1) *Interrupts? no WAW*

(2) *no WB conflict*

(3) *structural*

(4) *in-order issue*

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

In-Order Completion

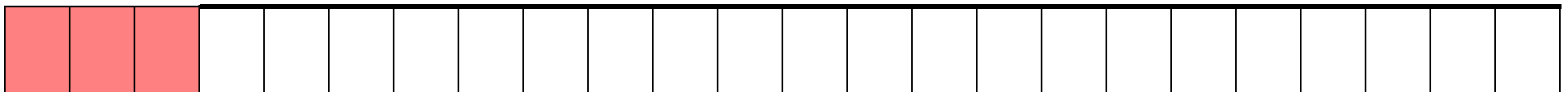
Time as an infinite sequence of WB slots:

Time -->

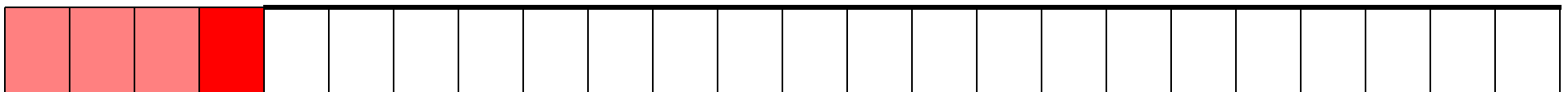


As instructions come in (decode) they reserve all time slots up to their completion time: e.g., an add, followed by a sub in a 5-stage pipeline:

Time --> ADD reserves 3 slots, done at decode



Time --> SUB reserves 3 (two already reserved plus one more)



In-Order Completion: Long Latency Instr

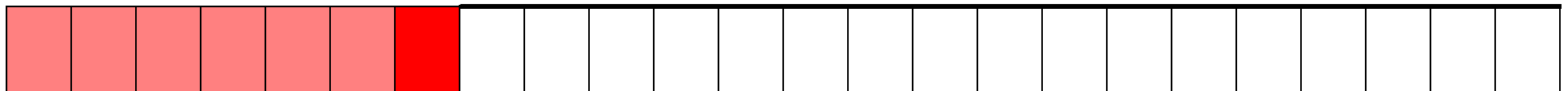
mul f0, f1, 10

- **add r1, r2, r3**

Time --> MULF reserves 6 slots, at time 0



Time --> ADD tries to reserve 3 slots, at time 1, slot 1+3 = 4 is taken, so is 5, and 6

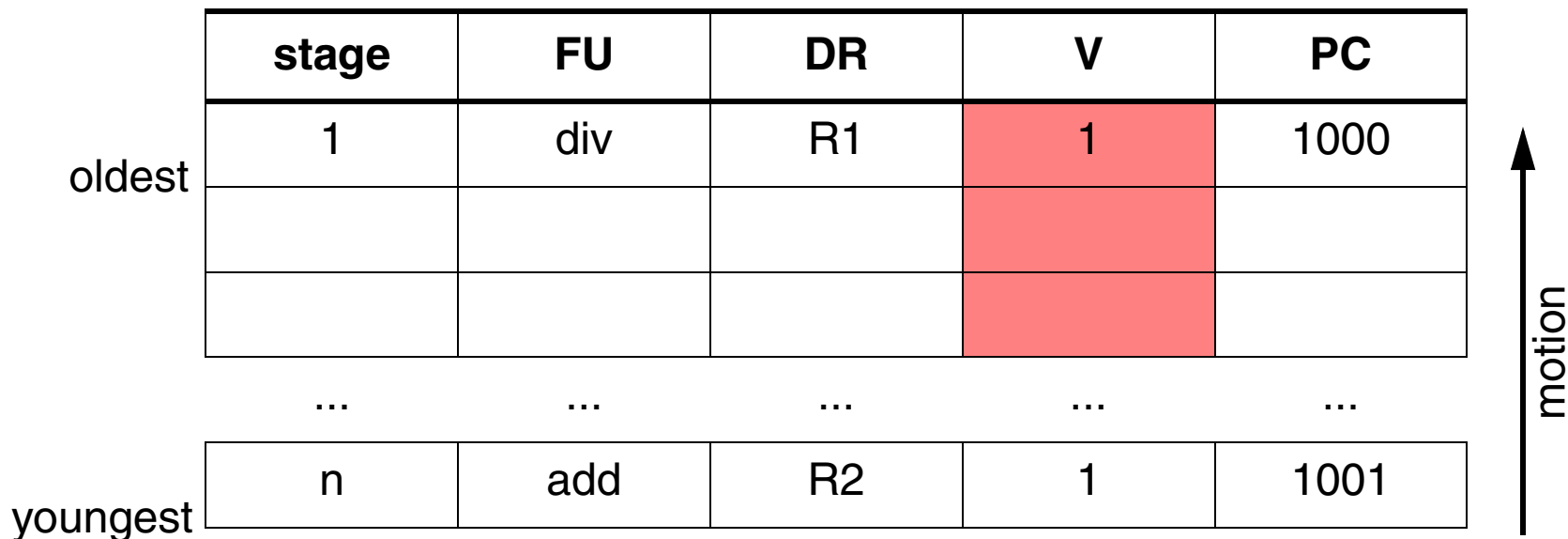


	1	2	3	4	5	6	7	8	9
<i>i</i>	F	D	X	X	X	X	X	W	
<i>i+1</i>		F				D	X	M	W

Implementation: Result Shift Register

- Simple solution: Modify state only when all preceding insts. are *known* to be exception free.

Mechanism: *Result Shift Register*

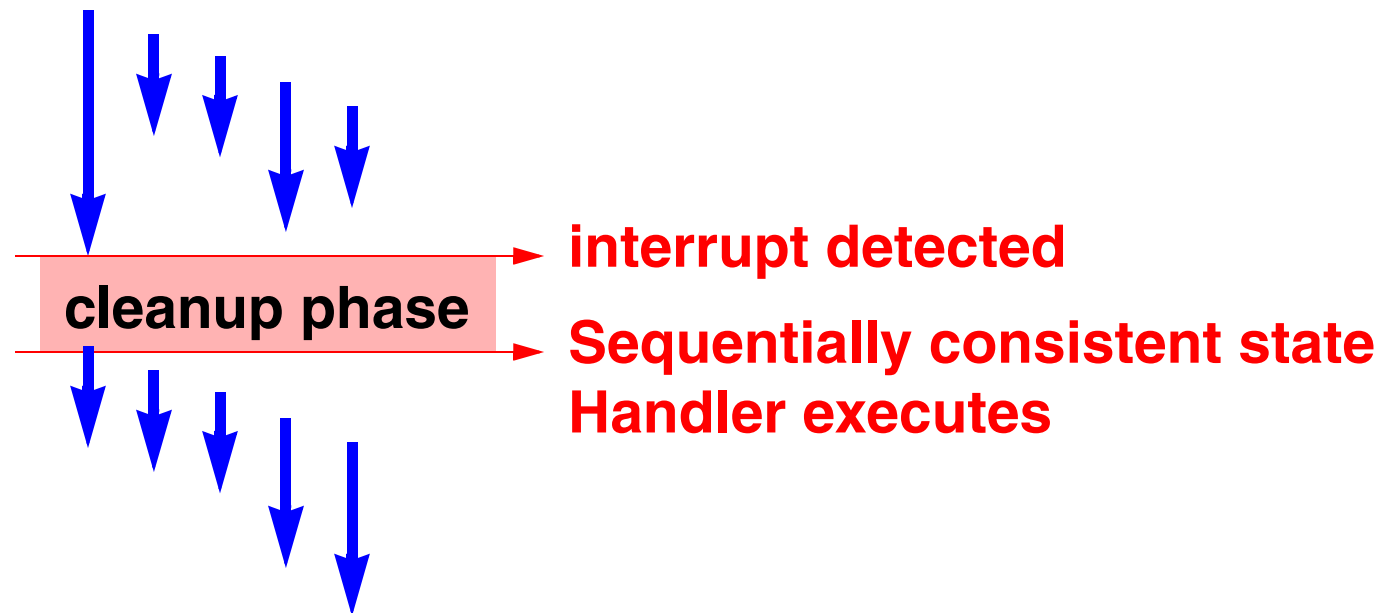


Reserve all stages for the duration of the instruction

Think of this as a wheel over the infinite sequence of WB slots

Out-of-Order Completion

- **Simple approach: Performance loss**
- **High-Performance approach:**
 - **Allow out-of-order WBs**
 - **Upon an interrupt cleanup and then present state to the user**



Cleanup Approaches: History File

keep a log of all changes made out-of-order

e.g., I updated register R10 and before I did it's value was 20

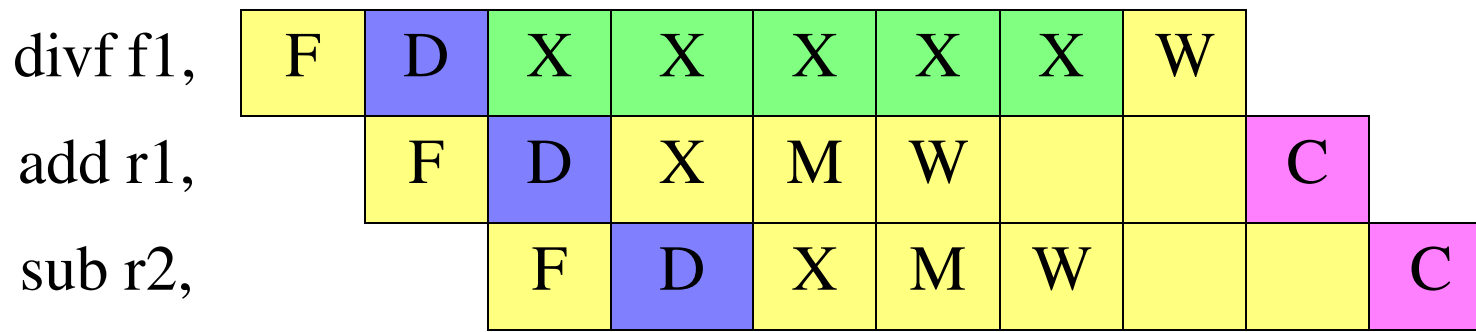
To cleanup, revert all changes, do so in reverse program order*

this will be needed for out-of-order execution w/ register renaming

divf f1, __, __ save previous value of f1

add r1, __, __ save previous value of r1

sub r2, r1, __ save previous value of r2



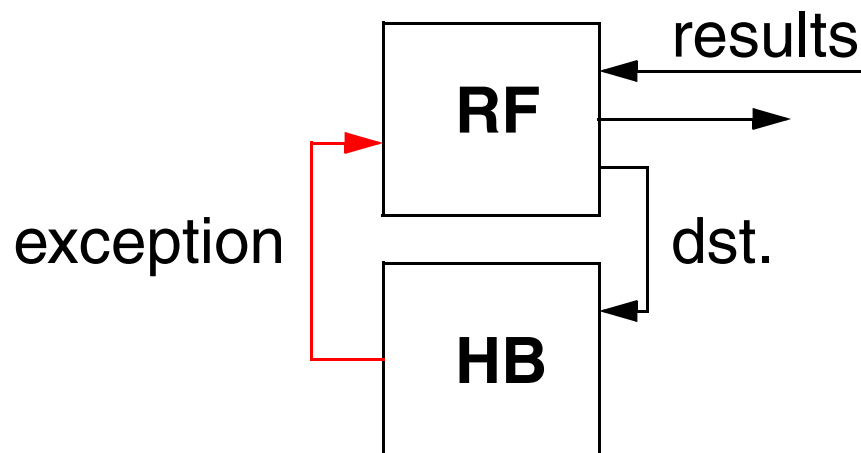
History File Contd.

divf f1,	F	D	X	X	X	X	X			
add r1,		F	D	X	M	W		R		
sub r2,			F	D	X	M	W	R	R	C

Cleaning up after an exception

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

- **Keep two states:**

Architectural: Updated in-order always consistent sequentially

Future: Updated out-of-order can be inconsistent

- **On an exception reset the Future and use the Architectural**

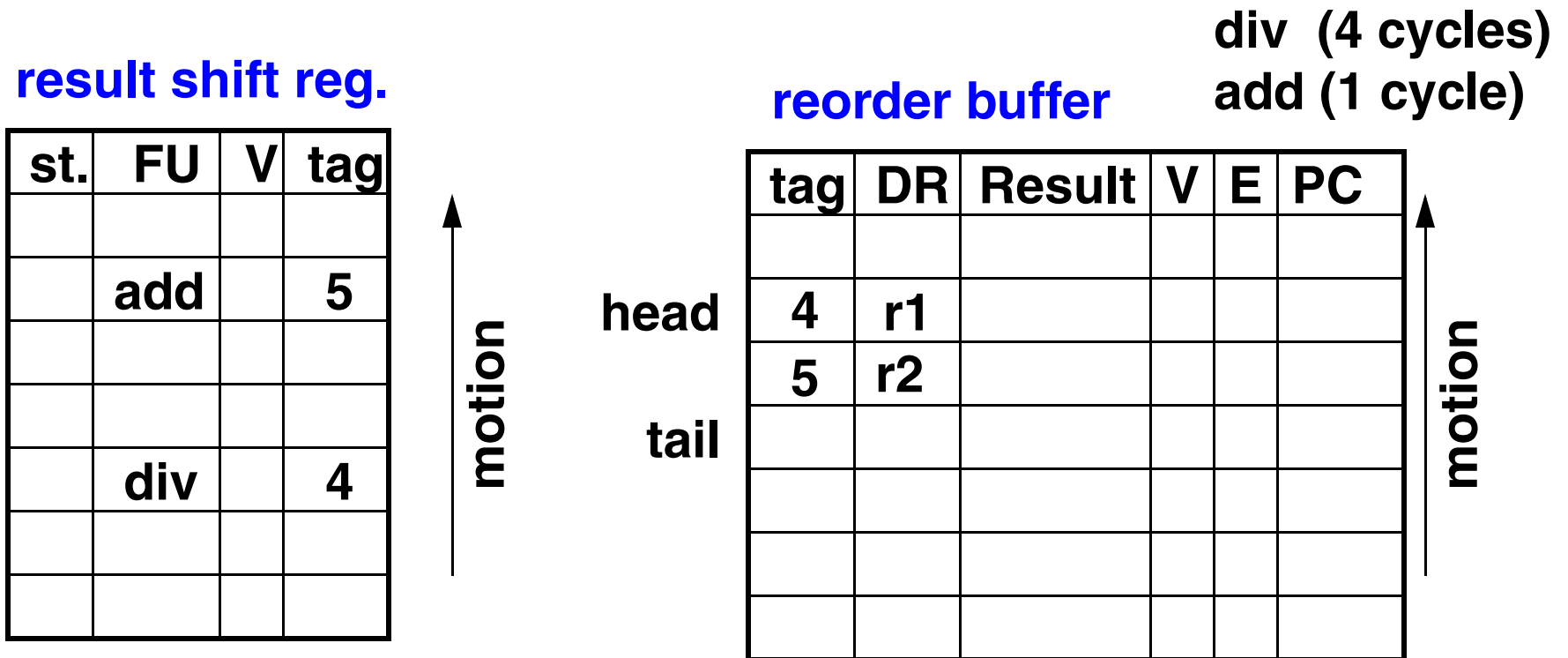
- **For regular execution use Future and if empty, Architectural**

Instructions do TWO writes

Future File Example

divf f1,	F	D	X	X	X	X	X	A			
								/F			
add r1,		F	D	X	M	F				A	
sub r2,			F	D	X	M	F				A

Reorder Buffer: Keeping Order



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

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

