

Review of Computer Architecture

-- Instruction sets, pipelines and caches

Review, #1

- **Designing to Last through Trends**

	<u>Capacity</u>	<u>Speed</u>
Logic	2x in 3 years	2x in 3 years
DRAM	4x in 3 years	2x in 10 years
Disk	4x in 3 years	2x in 10 years
Processor	(n.a.)	2x in 1.5 years

- **Time to run the task**

- Execution time, response time, latency

- **Tasks per day, hour, week, sec, ns, ...**

- Throughput, bandwidth

- **“X is n times faster than Y” means**

$$\frac{\text{ExTime}(Y)}{\text{ExTime}(X)} = \frac{\text{Performance}(X)}{\text{Performance}(Y)}$$

Review, #2

- **Amdahl's Law:**

$$\text{Speedup}_{\text{overall}} = \frac{\text{ExTime}_{\text{old}}}{\text{ExTime}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

- **CPI Law:**

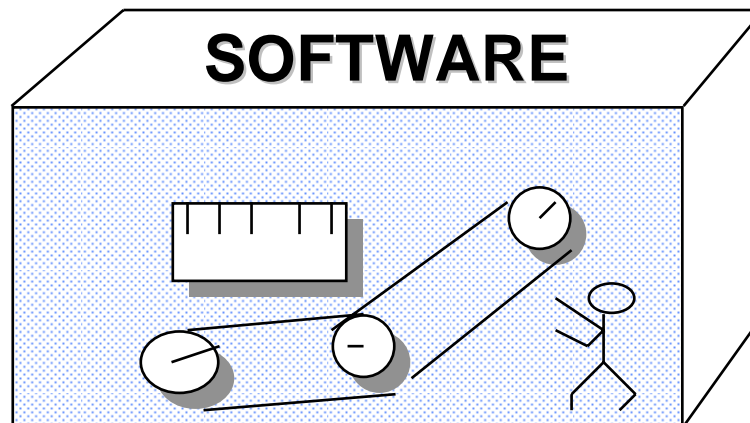
$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$

- **Execution time is the REAL measure of computer performance!**
- **Good products created when have:**
 - Good benchmarks
 - Good ways to summarize performance
- **Die Cost goes roughly with die area⁴**

Computer Architecture Is ...

the attributes of a [computing] system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls the logic design, and the physical implementation.

Amdahl, Blaaw, and Brooks, 1964



Evolution of Instruction Sets

- **Major advances in computer architecture are typically associated with landmark instruction set designs**
 - Ex: Stack vs GPR (System 360)
- **Design decisions must take into account:**
 - technology
 - machine organization
 - programming languages
 - compiler technology
 - operating systems
- **And they in turn influence these**

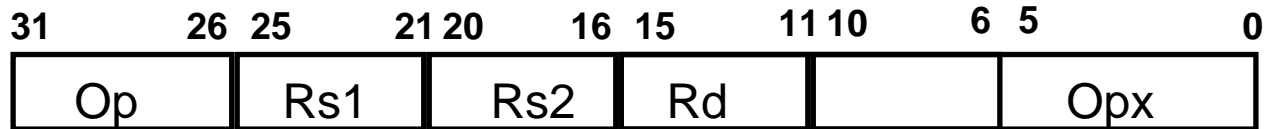
A "Typical" RISC

- **32-bit fixed format instruction (3 formats)**
- **32 32-bit GPR (R0 contains zero, DP take pair)**
- **3-address, reg-reg arithmetic instruction**
- **Single address mode for load/store:
base + displacement**
 - no indirection
- **Simple branch conditions**
- **Delayed branch**

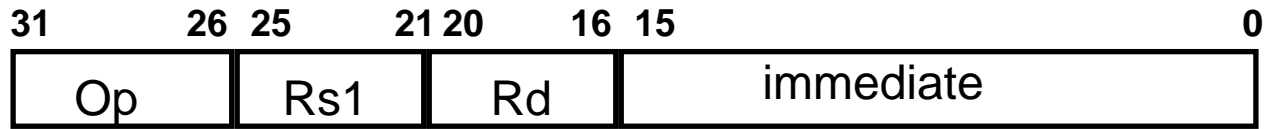
**see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC,
CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3**

Example: MIPS (- DLX)

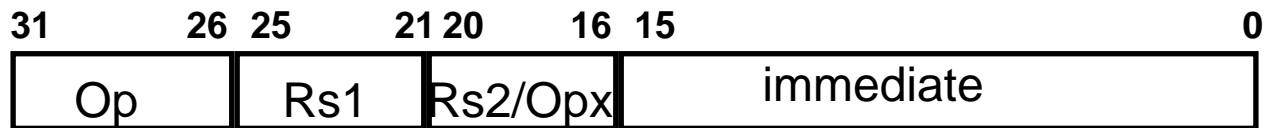
Register-Register



Register-Immediate



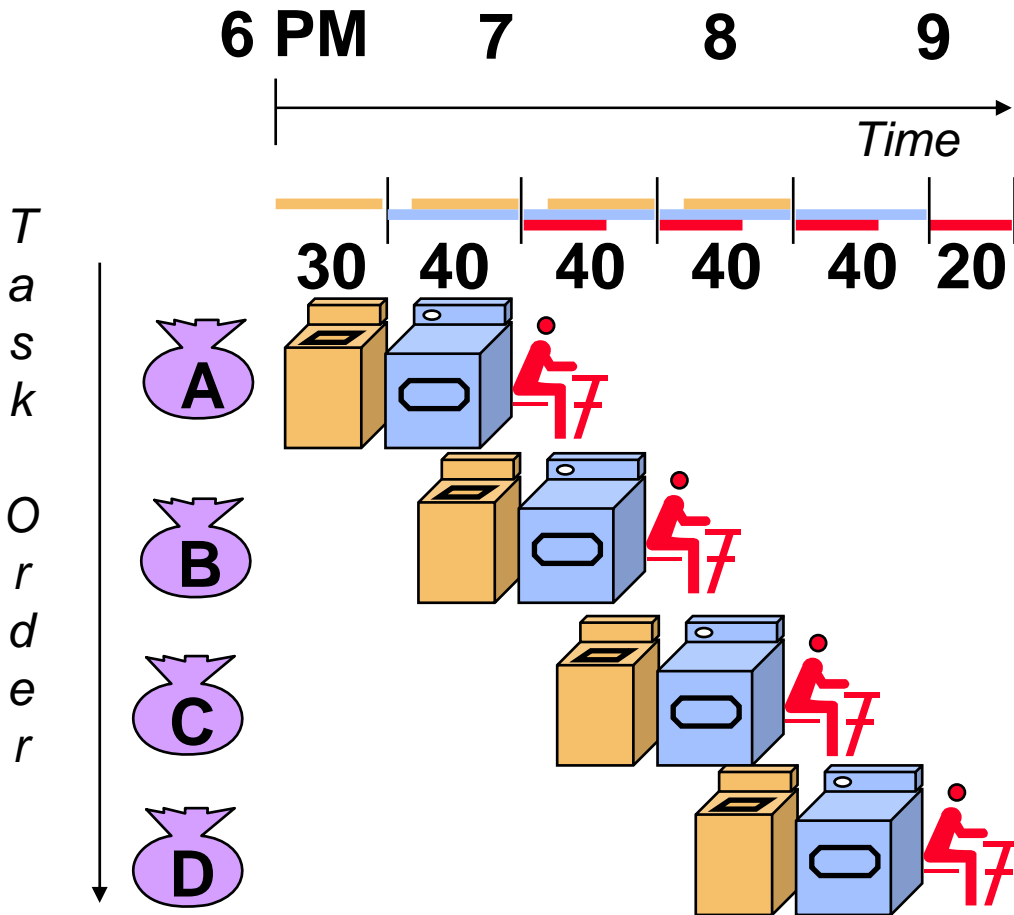
Branch



Jump / Call



Pipelining Lessons



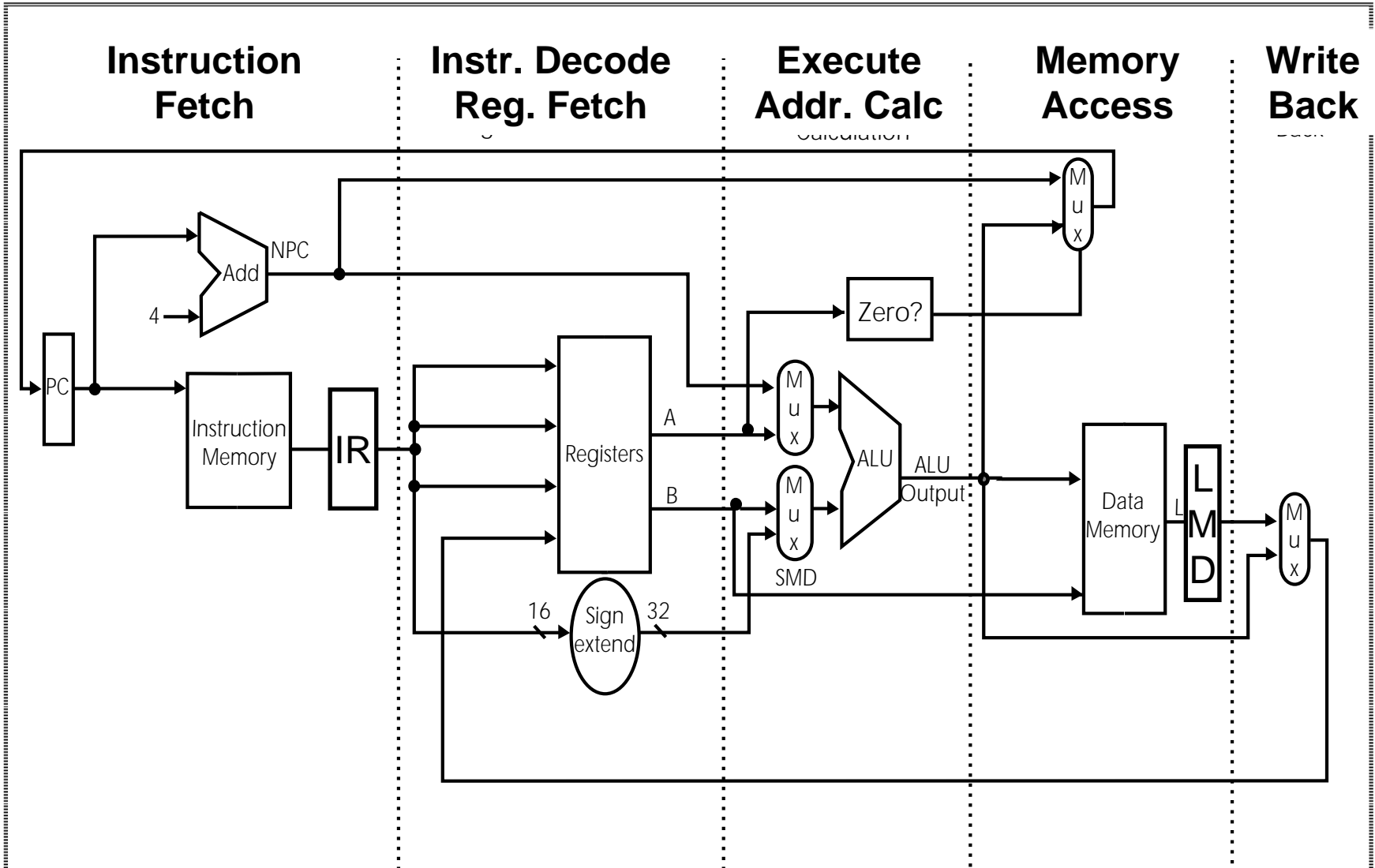
- Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload
- Pipeline rate limited by **slowest** pipeline stage
- **Multiple** tasks operating simultaneously
- Potential speedup = **Number pipe stages**
- Unbalanced lengths of pipe stages reduces speedup
- Time to “**fill**” pipeline and time to “**drain**” it reduces speedup

Computer Pipelines

- **Execute billions of instructions, so throughput is what matters**
- **DLX desirable features: all instructions same length, registers located in same place in instruction format, memory operands only in loads or stores**

5 Steps of DLX Datapath

Figure 3.1, Page 130



Its Not That Easy for Computers

- Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
 - **Structural hazards**: HW cannot support this combination of instructions (single person to fold and put clothes away)
 - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline (missing sock)
 - **Control hazards**: Pipelining of branches & other instructions that change the PC
 - Common solution is to **stall** the pipeline until the hazard is resolved, inserting one or more “**bubbles**” in the pipeline

Speed Up Equation for Pipelining

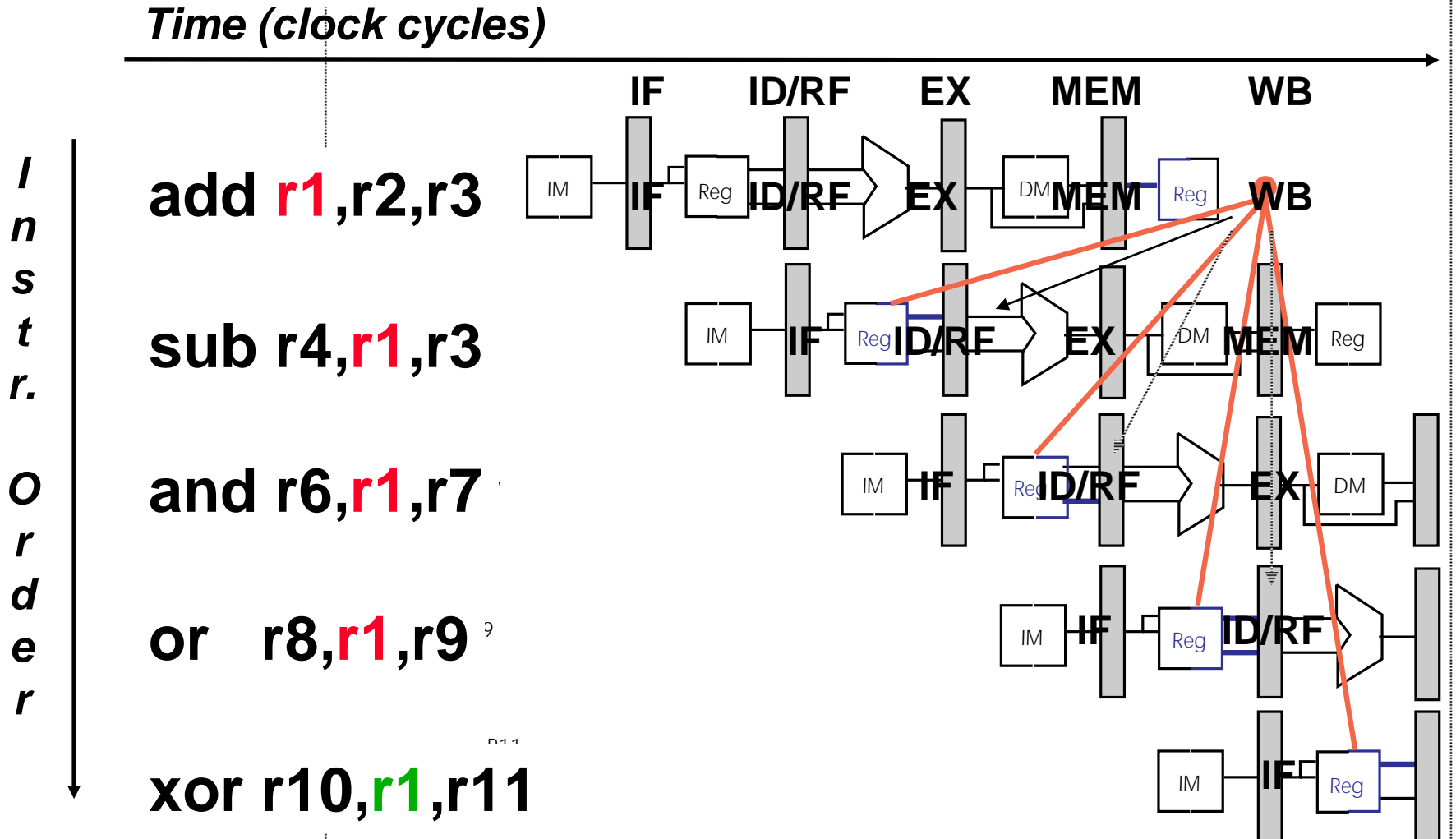
$$\text{CPI}_{\text{pipelined}} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instr}$$

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

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

Data Hazard on R1

Figure 3.9, page 147



Three Generic Data Hazards

Instr_i followed by Instr_j

- **Read After Write (RAW)**
Instr_j tries to read operand before Instr_i writes it

Three Generic Data Hazards

Instr_i followed by Instr_j

- **Write After Read (WAR)**

Instr_j tries to write operand before Instr_i reads it

- Gets wrong operand

- **Can't happen in DLX 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

Instr_i followed by Instr_j

- **Write After Write (WAW)**
Instr_j tries to write operand before Instr_i writes it
 - Leaves wrong result (Instr_i not Instr_j)
- **Can't happen in DLX 5 stage pipeline because:**
 - All instructions take 5 stages, and
 - Writes are always in stage 5
- **Will see WAR and WAW in later more complicated pipes**

Forwarding to Avoid Data Hazard

Figure 3.10, Page 149

Time (clock cycles)

Instruction Order

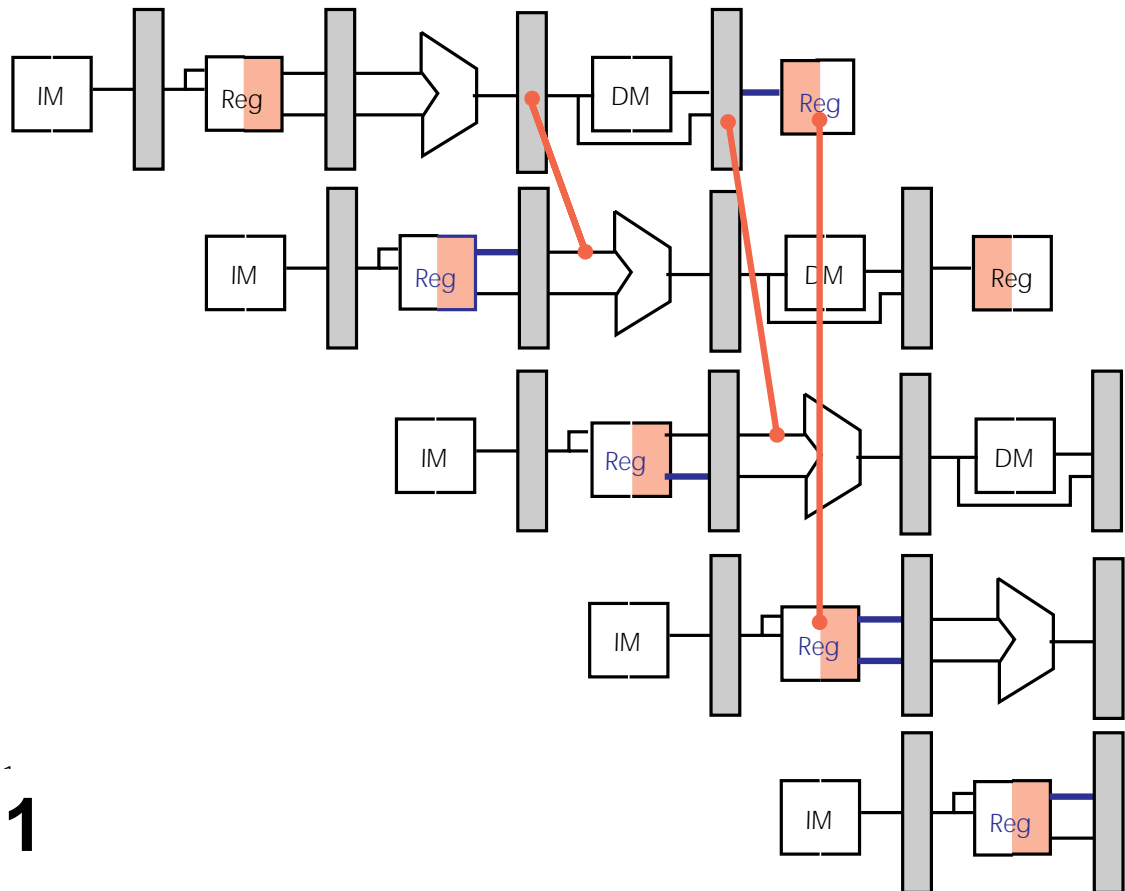
add r1,r2,r3

sub r4,r1,r3

and r6,r1,r7

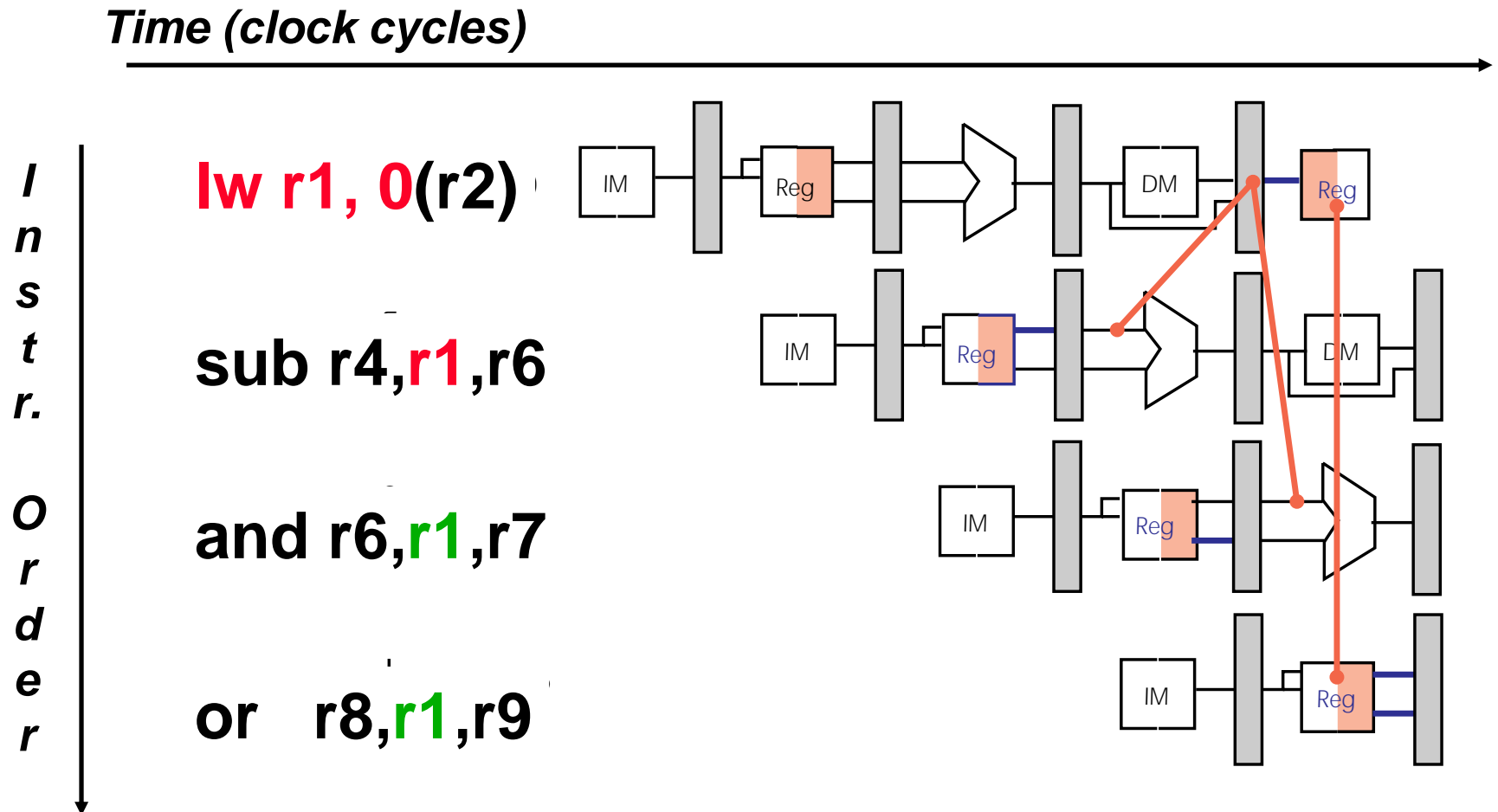
or r8,r1,r9

xor r10,r1,r11



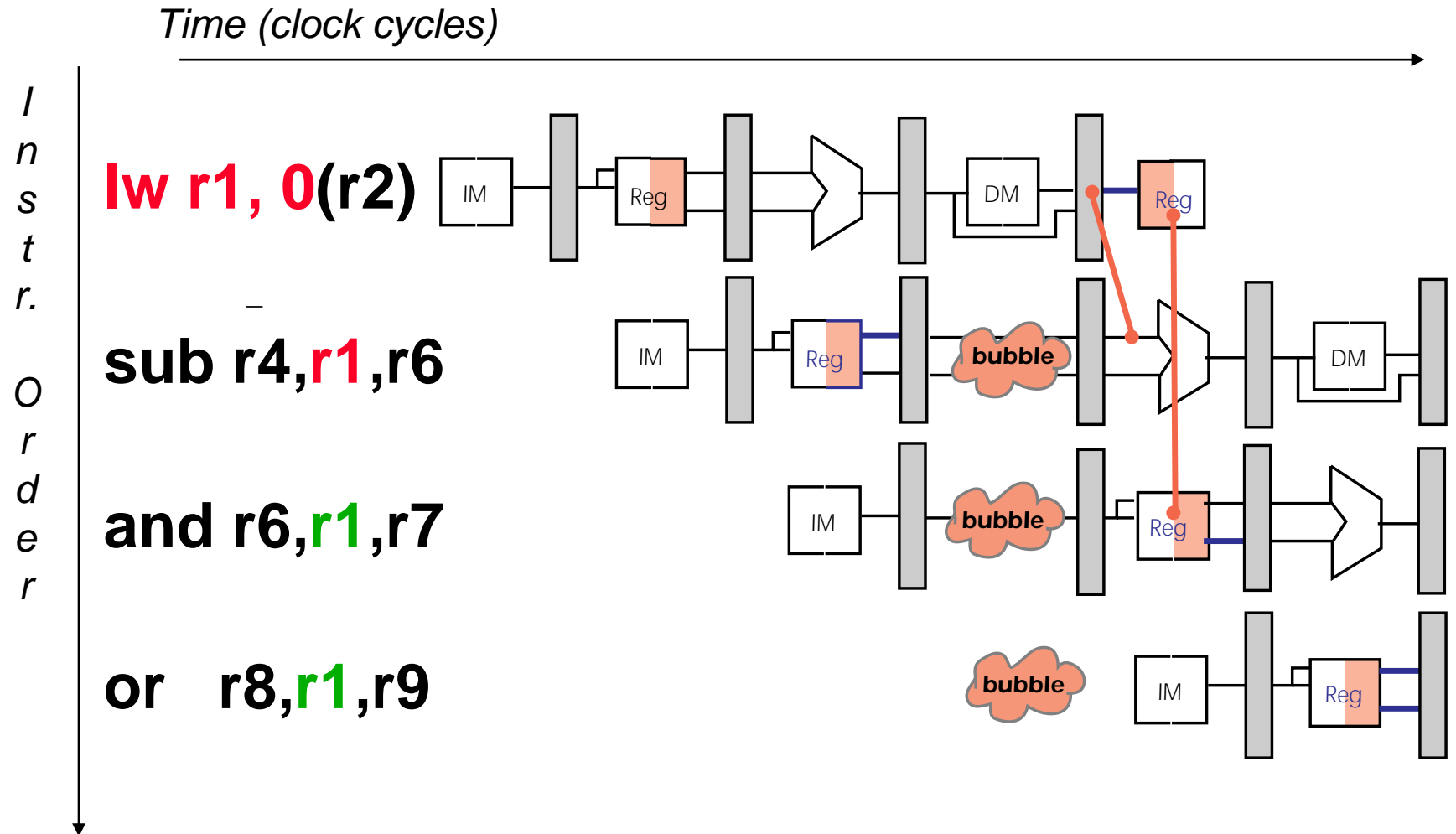
Data Hazard Even with Forwarding

Figure 3.12, Page 153



Data Hazard Even with Forwarding

Figure 3.13, Page 154



Software Scheduling to Avoid Load Hazards

Try producing fast code for

$a = b + c;$

$d = e - f;$

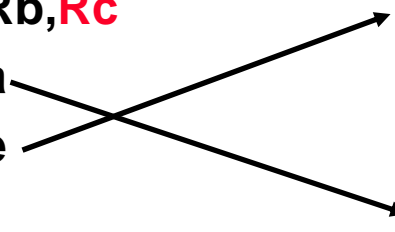
assuming $a, b, c, d, e,$ and f in memory.

Slow code:

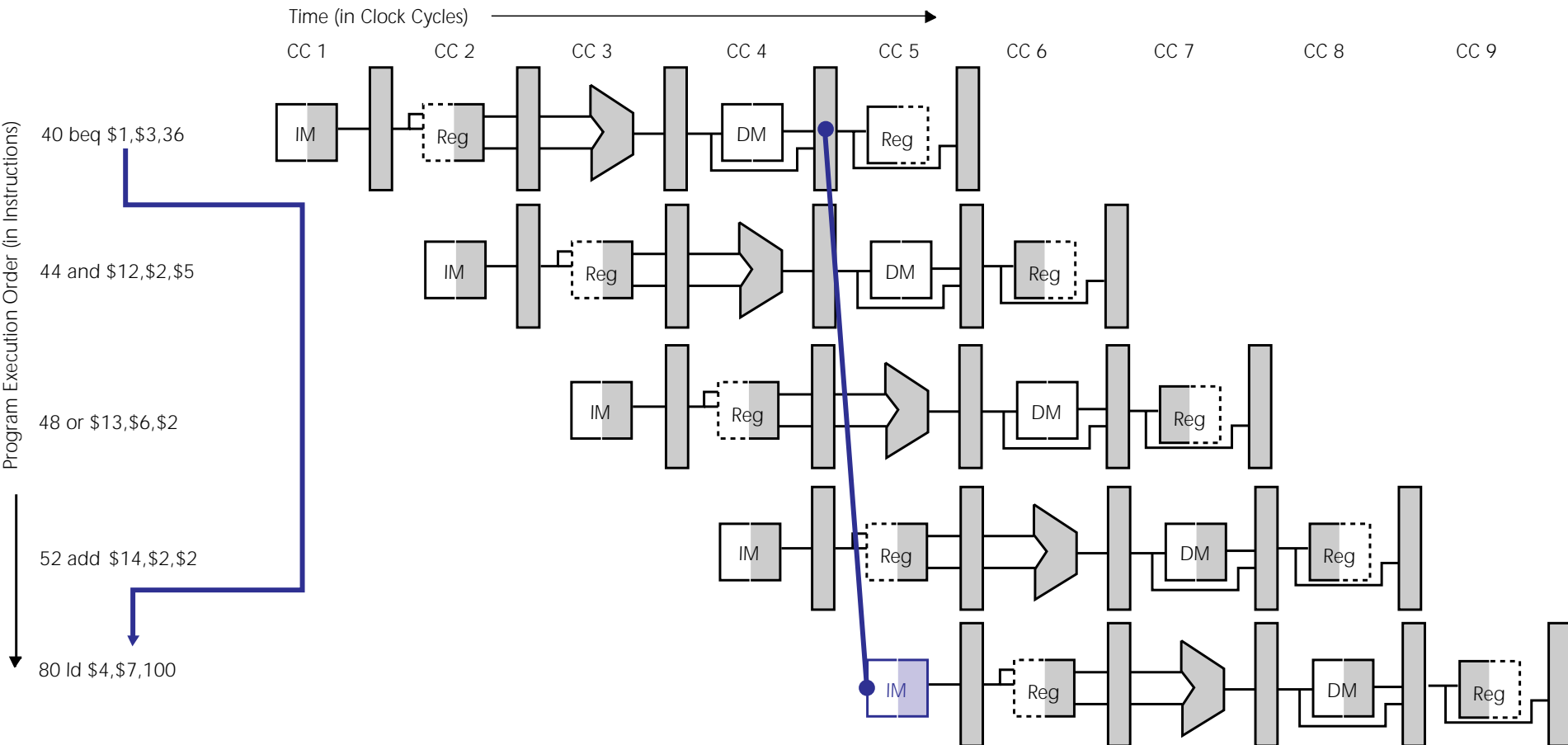
```
LW    Rb,b
LW    Rc,c
ADD   Ra,Rb,Rc
SW    a,Ra
LW    Re,e
LW    Rf,f
SUB   Rd,Re,Rf
SW    d,Rd
```

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
```



Control Hazard on Branches Three Stage Stall



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
- **DLX branch tests if register = 0 or ° 0**
- **DLX Solution:**
 - Move Zero test to ID/RF stage
 - Adder to calculate new PC in ID/RF stage
 - 1 clock cycle penalty for branch versus 3

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% DLX branches not taken on average
- PC+4 already calculated, so use it to get next instruction

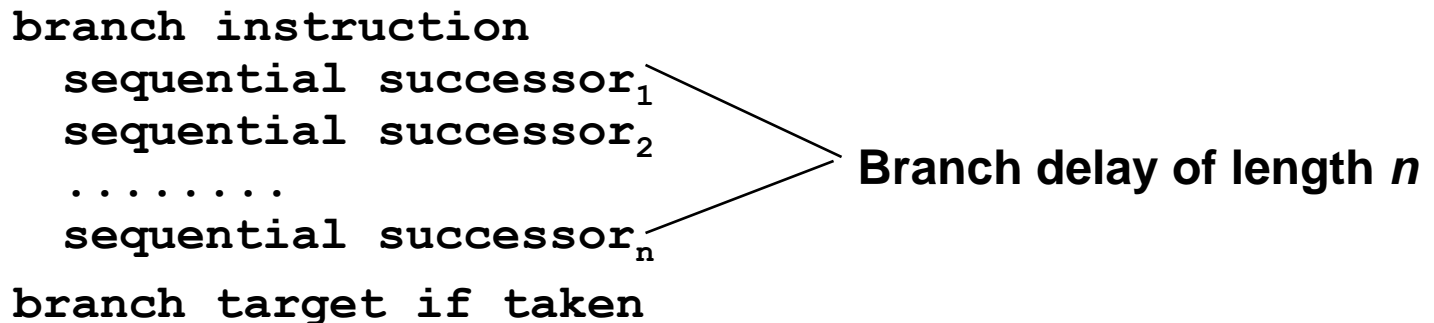
#3: Predict Branch Taken

- 53% DLX branches taken on average
- But haven't calculated branch target address in DLX
 - » DLX 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



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

Delayed Branch

- **Where to get instructions to fill branch delay slot?**
 - Before branch instruction
 - From the target address: only valuable when branch taken
 - From fall through: only valuable when branch not taken
 - Cancelling branches allow more slots to be filled
- **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: 7-8 stage pipelines, multiple instructions issued per clock (superscalar)**

Pipelining Introduction

Summary

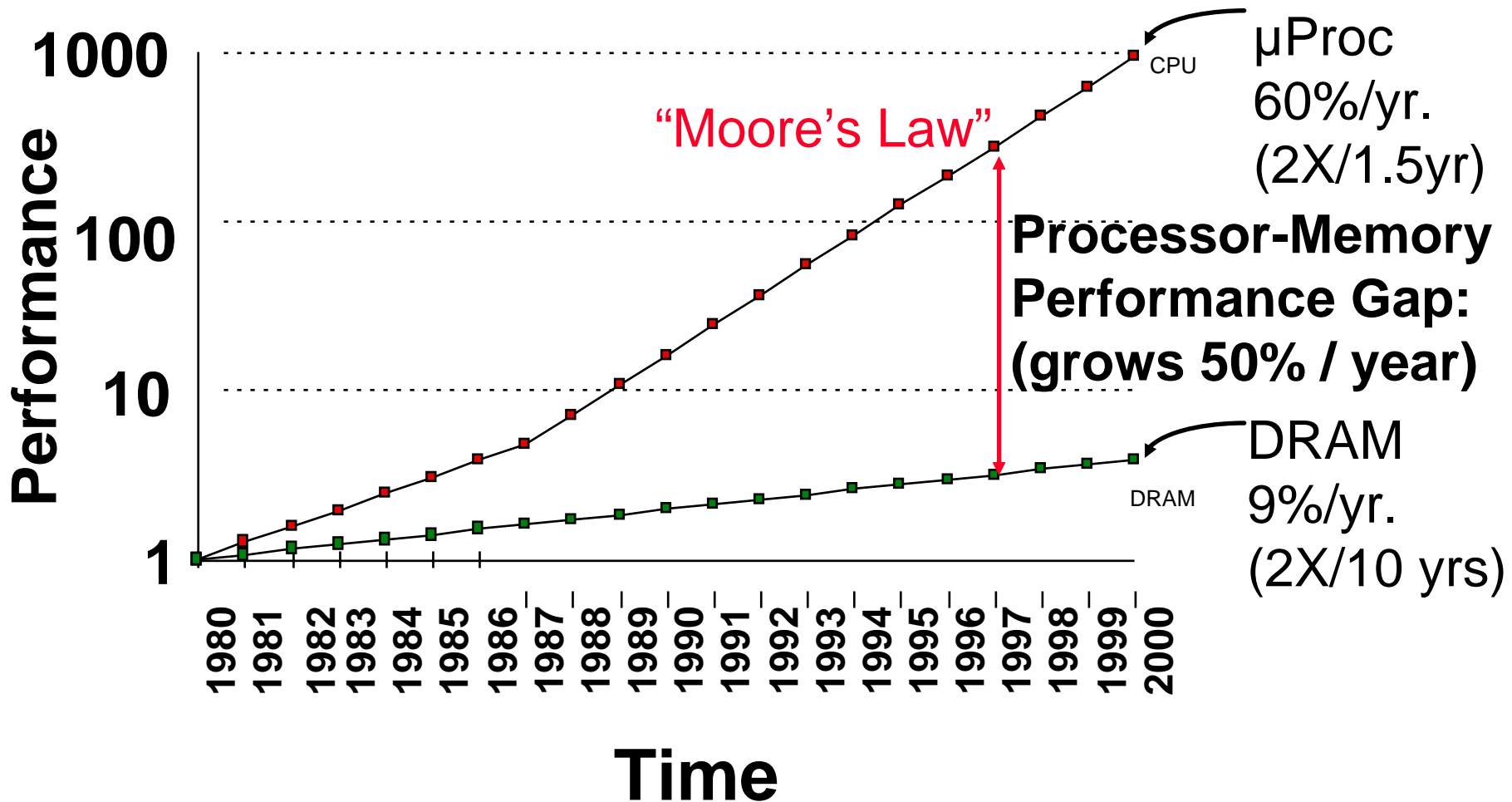
- **Just overlap tasks, and easy if tasks are independent**
- **Speed Up \propto Pipeline Depth; if ideal CPI is 1, then:**

$$\text{Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Clock Cycle Unpipelined}}{\text{Clock Cycle Pipelined}}$$

- **Hazards limit performance on computers:**
 - **Structural: need more HW resources**
 - **Data (RAW,WAR,WAW): need forwarding, compiler scheduling**
 - **Control: delayed branch, prediction**

Recap: Who Cares About the Memory Hierarchy?

Processor-DRAM Memory Gap (latency)



Levels of the Memory Hierarchy

Capacity
Access Time
Cost

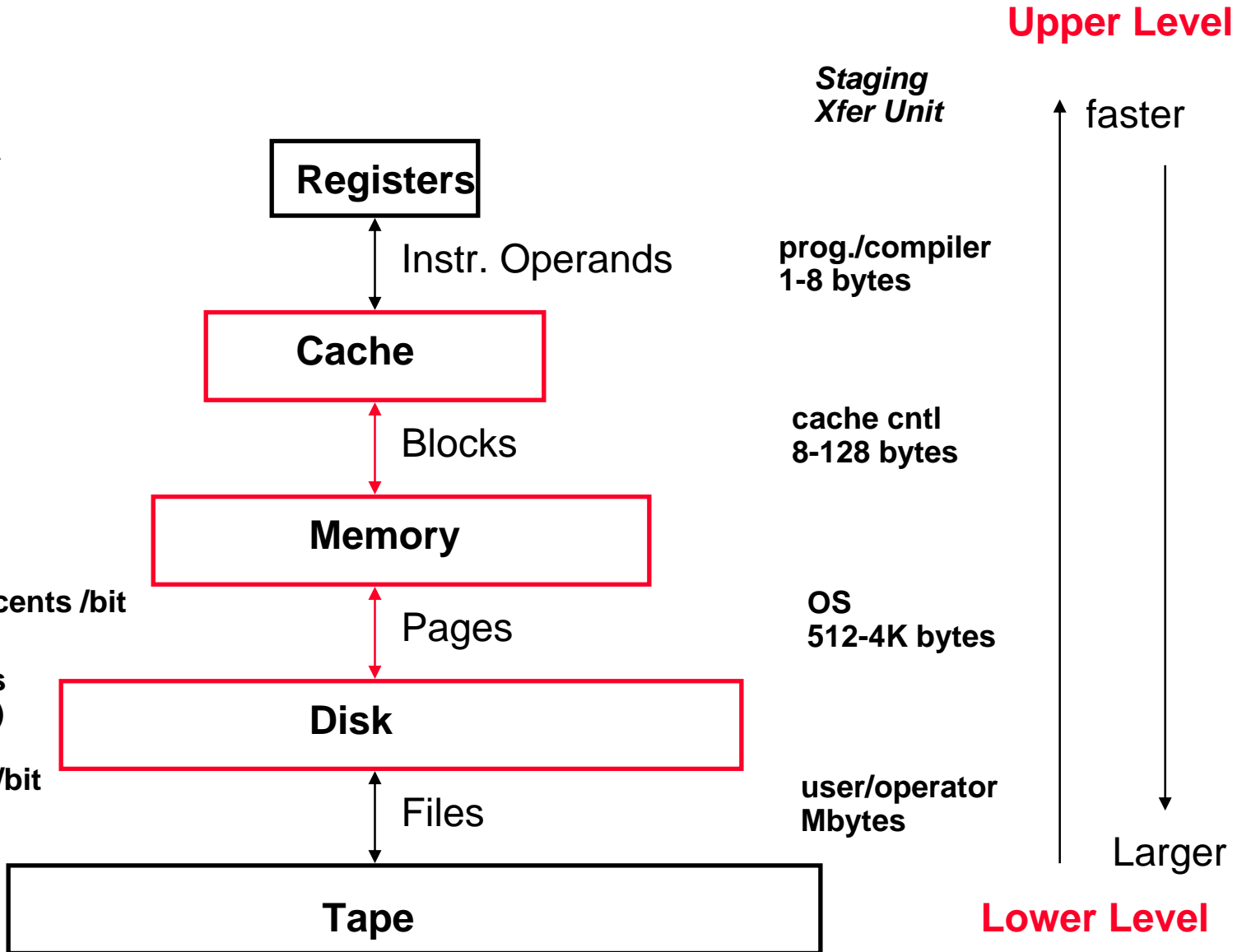
CPU Registers
100s Bytes
<10s ns

Cache
K Bytes
10-100 ns
1-0.1 cents/bit

Main Memory
M Bytes
200ns- 500ns
\$.0001-.00001 cents /bit

Disk
G Bytes, 10 ms
(10,000,000 ns)
 10^{-5} - 10^{-6} cents/bit

Tape
infinite
sec-min
 10^{-8}

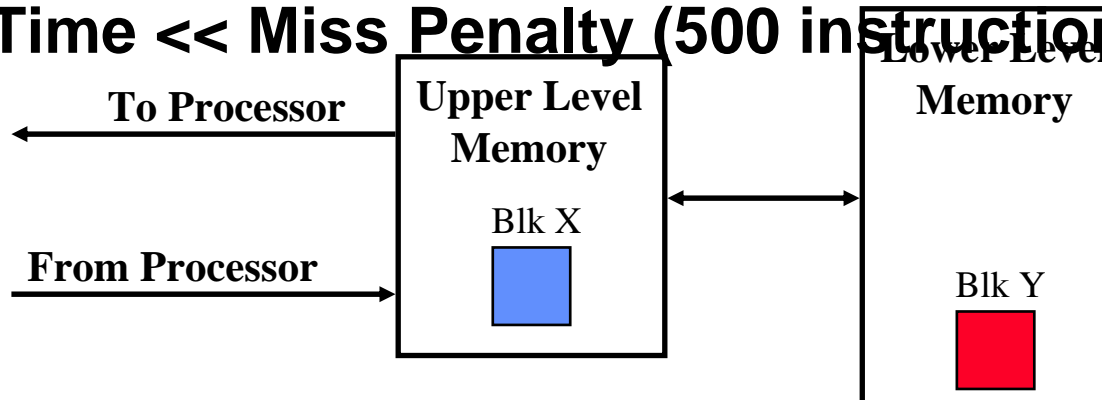


The Principle of Locality

- **The Principle of Locality:**
 - Program access a relatively small portion of the address space at any instant of time.
- **Two Different Types of Locality:**
 - **Temporal Locality** (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
 - **Spatial Locality** (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straightline code, array access)
- **Last 15 years, HW relied on locality for speed**

Memory Hierarchy: Terminology

- **Hit**: data appears in some block in the upper level (example: Block X)
 - **Hit Rate**: the fraction of memory access found in the upper level
 - **Hit Time**: Time to access the upper level which consists of RAM access time + Time to determine hit/miss
- **Miss**: data needs to be retrieve from a block in the lower level (Block Y)
 - **Miss Rate** = $1 - (\text{Hit Rate})$
 - **Miss Penalty**: Time to replace a block in the upper level + Time to deliver the block the processor
- **Hit Time** \ll **Miss Penalty** (500 instructions on 21264!)

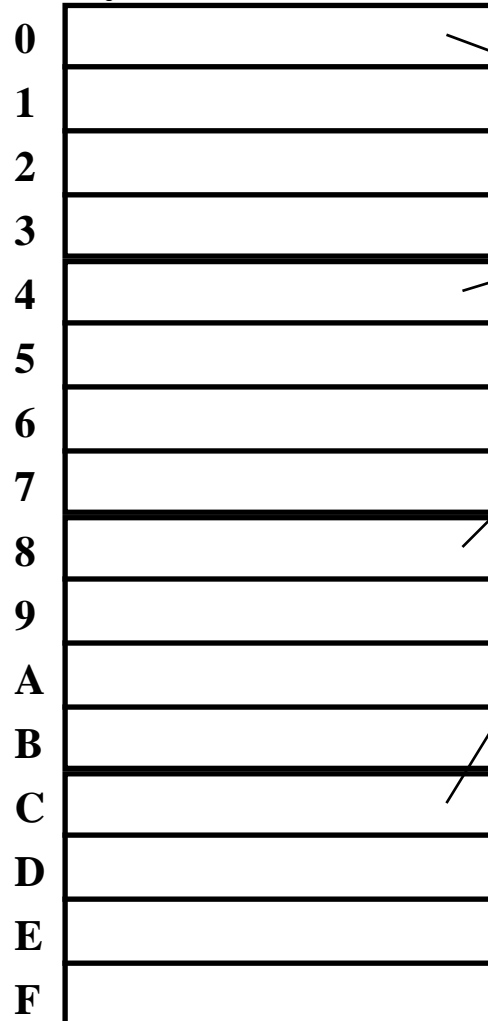


Cache Measures

- ***Hit rate***: fraction found in that level
 - So high that usually talk about ***Miss rate***
 - Miss rate fallacy: as MIPS to CPU performance, miss rate to average memory access time in memory
- **Average memory-access time**
= Hit time + Miss rate x Miss penalty
(ns or clocks)
- ***Miss penalty***: time to replace a block from lower level, including time to replace in CPU
 - ***access time***: time to lower level
= f(latency to lower level)
 - ***transfer time***: time to transfer block
= f(BW between upper & lower levels)

Simplest Cache: Direct Mapped

Memory Address Memory



4 Byte Direct Mapped Cache

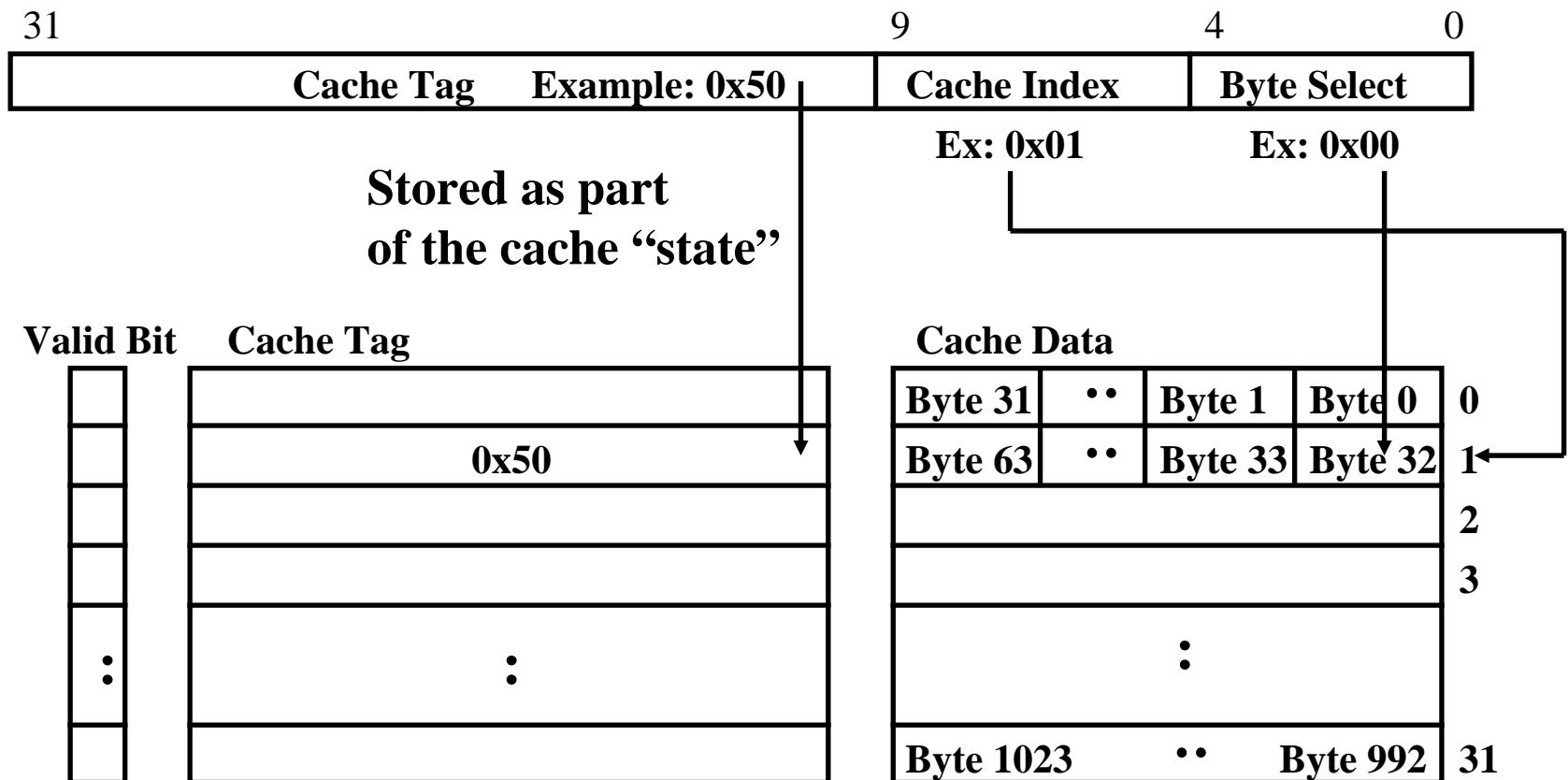
Cache Index



- **Location 0 can be occupied by data from:**
 - Memory location 0, 4, 8, ... etc.
 - In general: any memory location whose 2 LSBs of the address are 0s
 - Address<1:0> => cache index
- **Which one should we place in the cache?**
- **How can we tell which one is in the cache?**

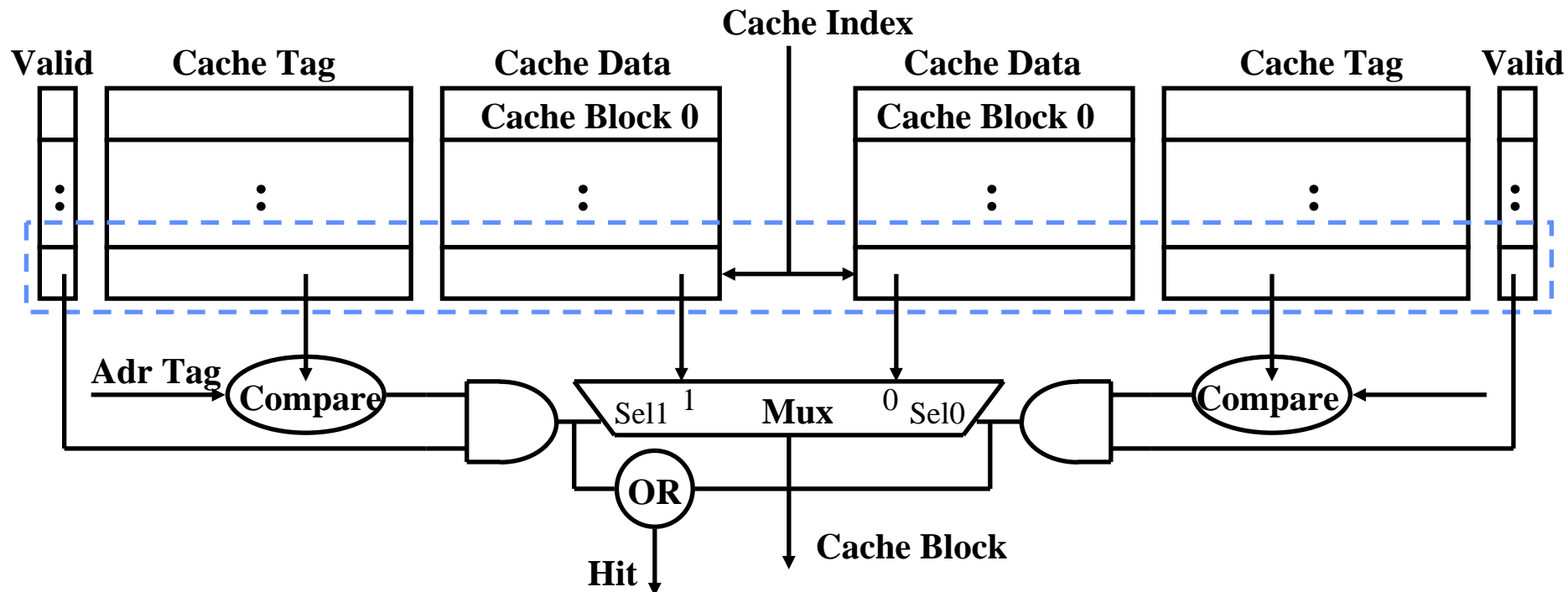
1 KB Direct Mapped Cache, 32B blocks

- For a $2^{**} N$ byte cache:
 - The uppermost $(32 - N)$ bits are always the Cache Tag
 - The lowest M bits are the Byte Select (Block Size = $2^{**} M$)



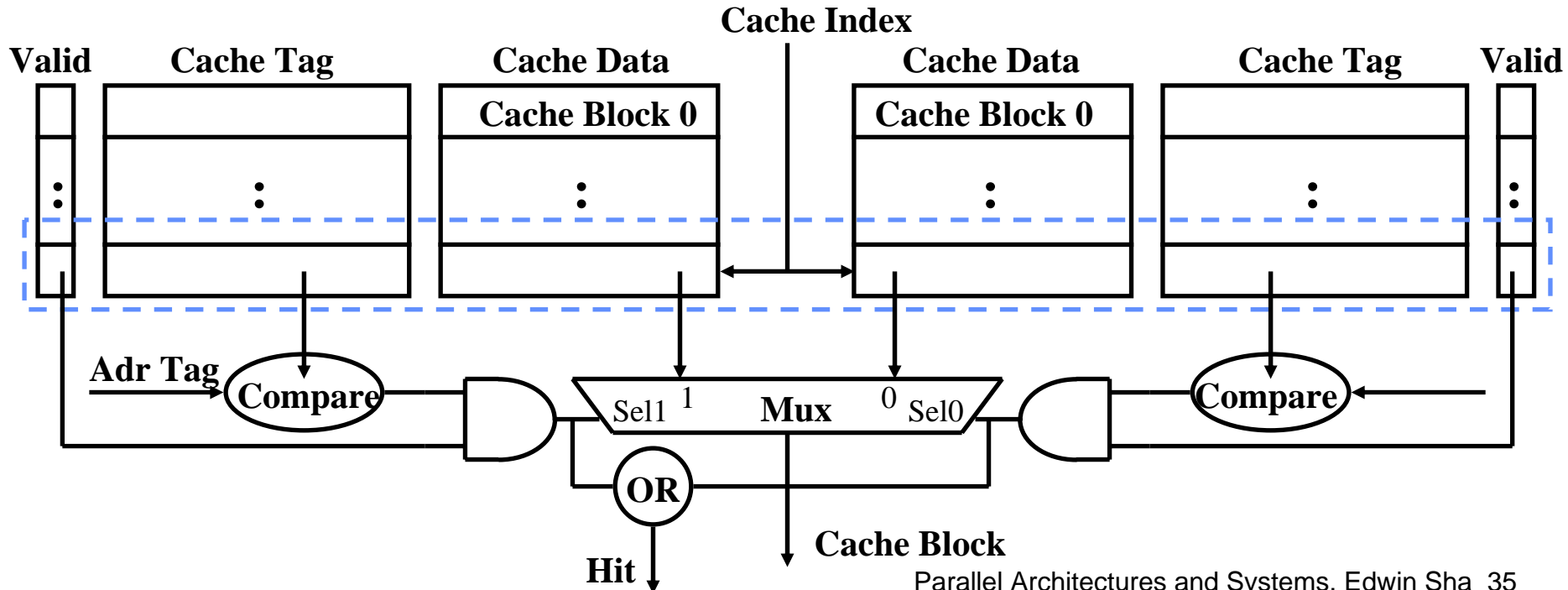
Two-way Set Associative Cache

- **N-way set associative: N entries for each Cache Index**
 - N direct mapped caches operates in parallel (N typically 2 to 4)
- **Example: Two-way set associative cache**
 - Cache Index selects a “set” from the cache
 - The two tags in the set are compared in parallel
 - Data is selected based on the tag result



Disadvantage of Set Associative Cache

- **N-way Set Associative Cache v. Direct Mapped Cache:**
 - N comparators vs. 1
 - Extra MUX delay for the data
 - Data comes AFTER Hit/Miss
- **In a direct mapped cache, Cache Block is available BEFORE Hit/Miss:**
 - Possible to assume a hit and continue. Recover later if miss.

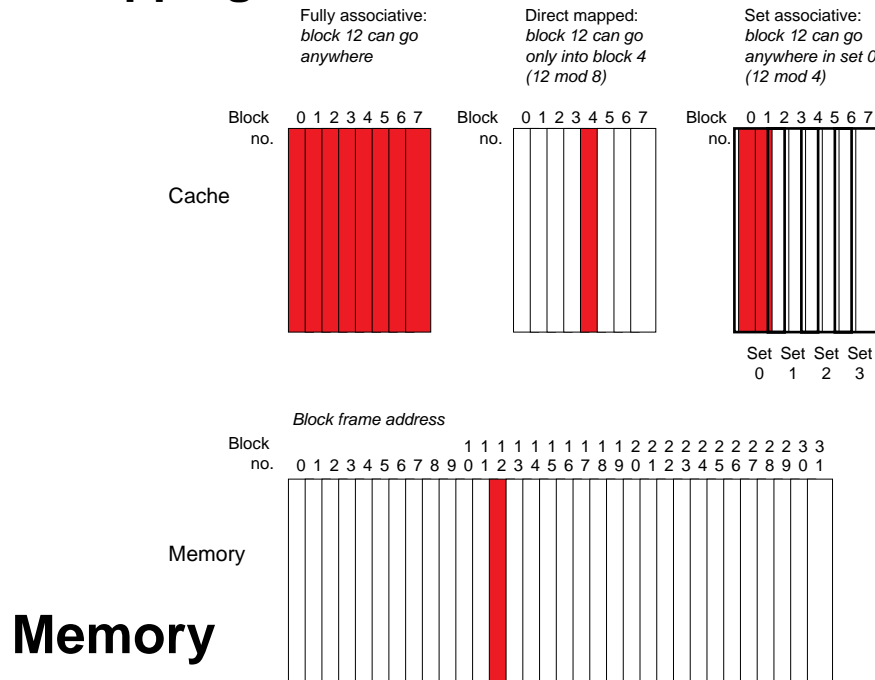


4 Questions for Memory Hierarchy

- Q1: Where can a block be placed in the upper level?
(Block placement)
- Q2: How is a block found if it is in the upper level?
(Block identification)
- Q3: Which block should be replaced on a miss?
(Block replacement)
- Q4: What happens on a write?
(Write strategy)

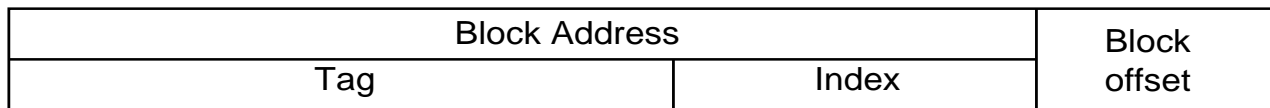
Q1: Where can a block be placed in the upper level?

- **Block 12 placed in 8 block cache:**
 - Fully associative, direct mapped, 2-way set associative
 - **S.A. Mapping = Block Number Modulo Number Sets**



Q2: How is a block found if it is in the upper level?

- **Tag on each block**
 - No need to check index or block offset
- **Increasing associativity shrinks index, → expands tag →**



Q3: Which block should be replaced on a miss?

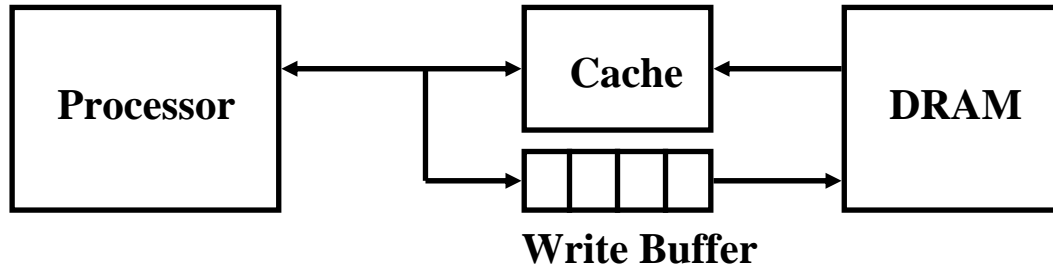
- Easy for Direct Mapped
- Set Associative or Fully Associative:
 - Random
 - LRU (Least Recently Used)

Associativity:	2-way		4-way		8-way	
Size	LRU	Random	LRU	Random	LRU	Random
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Q4: What happens on a write?

- **Write through**—The information is written to both the block in the cache and to the block in the lower-level memory.
- **Write back**—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
 - is block clean or dirty?
- **Pros and Cons of each?**
 - WT: read misses cannot result in writes
 - WB: no repeated writes to same location
- **WT always combined with write buffers so that don't wait for lower level memory**

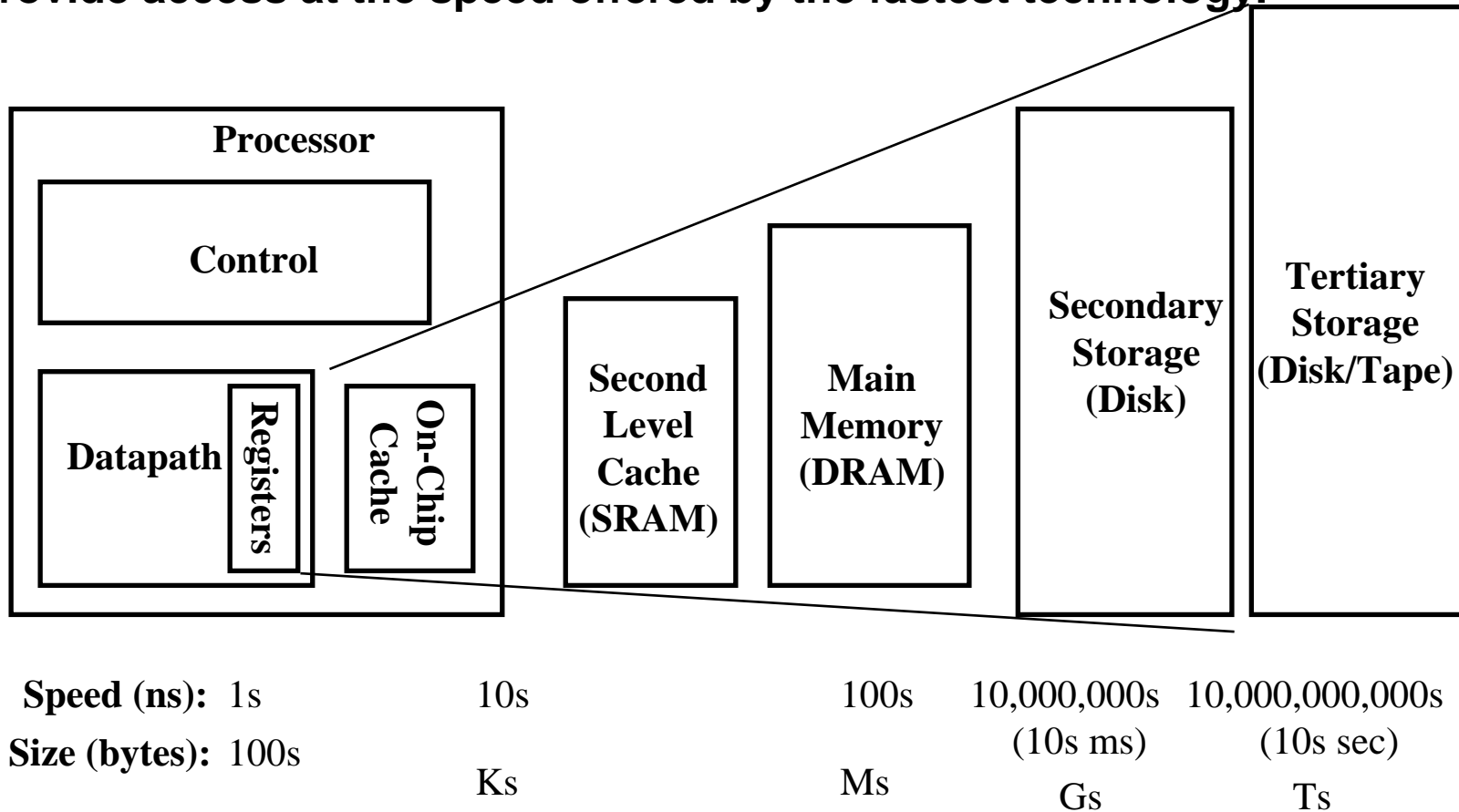
Write Buffer for Write Through



- **A Write Buffer is needed between the Cache and Memory**
 - Processor: writes data into the cache and the write buffer
 - Memory controller: write contents of the buffer to memory
- **Write buffer is just a FIFO:**
 - Typical number of entries: 4
 - Works fine if: Store frequency (w.r.t. time) $\ll 1 / \text{DRAM write cycle}$
- **Memory system designer's nightmare:**
 - Store frequency (w.r.t. time) $\rightarrow 1 / \text{DRAM write cycle}$
 - Write buffer saturation

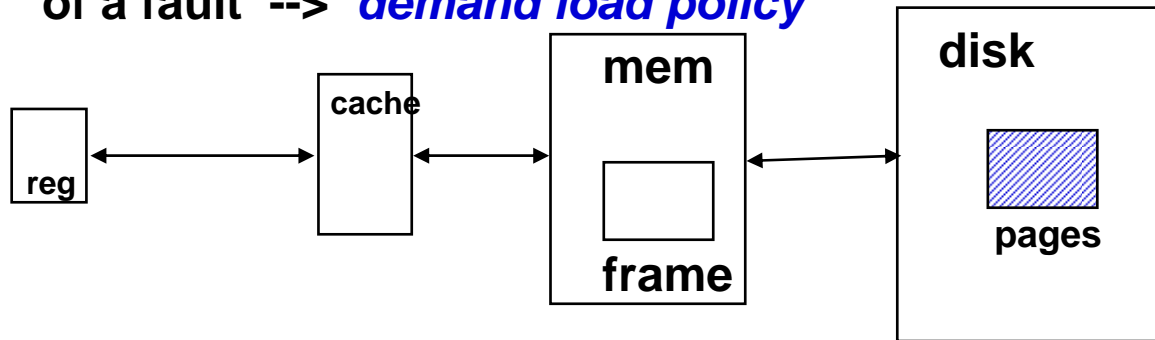
A Modern Memory Hierarchy

- **By taking advantage of the principle of locality:**
 - Present the user with as much memory as is available in the cheapest technology.
 - Provide access at the speed offered by the fastest technology.



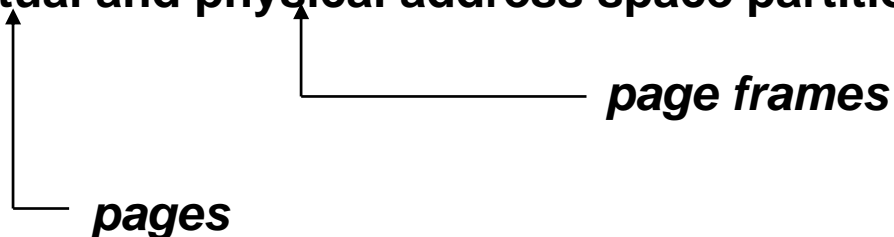
Basic Issues in VM System Design

- size of information blocks that are transferred from secondary to main storage (M)
- block of information brought into M, and M is full, then some region of M must be released to make room for the new block --> *replacement policy*
- which region of M is to hold the new block --> *placement policy*
- missing item fetched from secondary memory only on the occurrence of a fault --> *demand load policy*



Paging Organization

virtual and physical address space partitioned into blocks of equal size



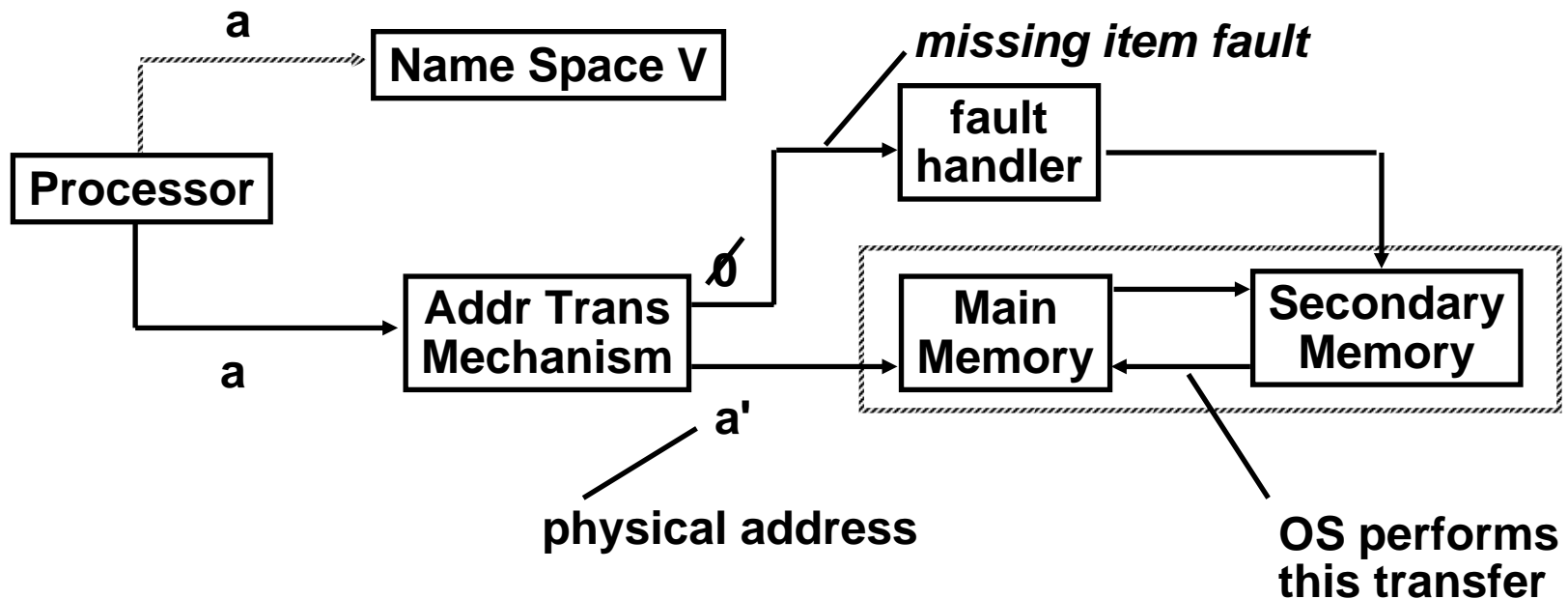
Address Map

$V = \{0, 1, \dots, n - 1\}$ virtual address space $n > m$
 $M = \{0, 1, \dots, m - 1\}$ physical address space

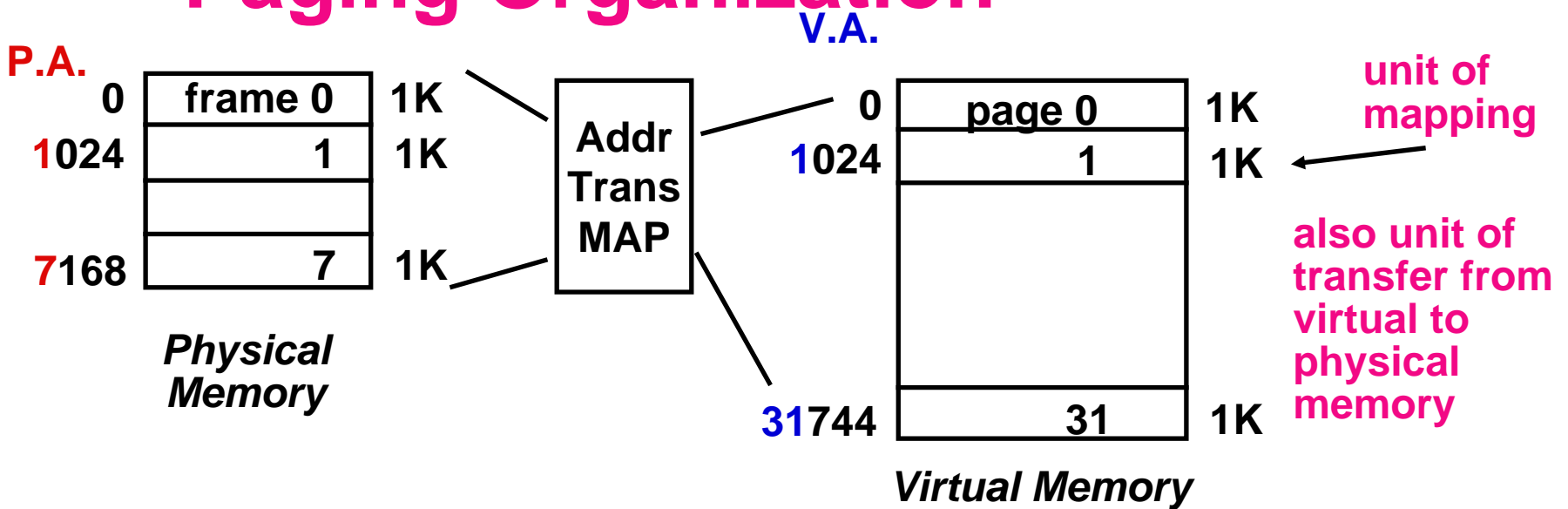
MAP: $V \rightarrow M \cup \{\emptyset\}$ address mapping function

MAP(a) = a' if data at virtual address a is present in physical address a' and a' in M

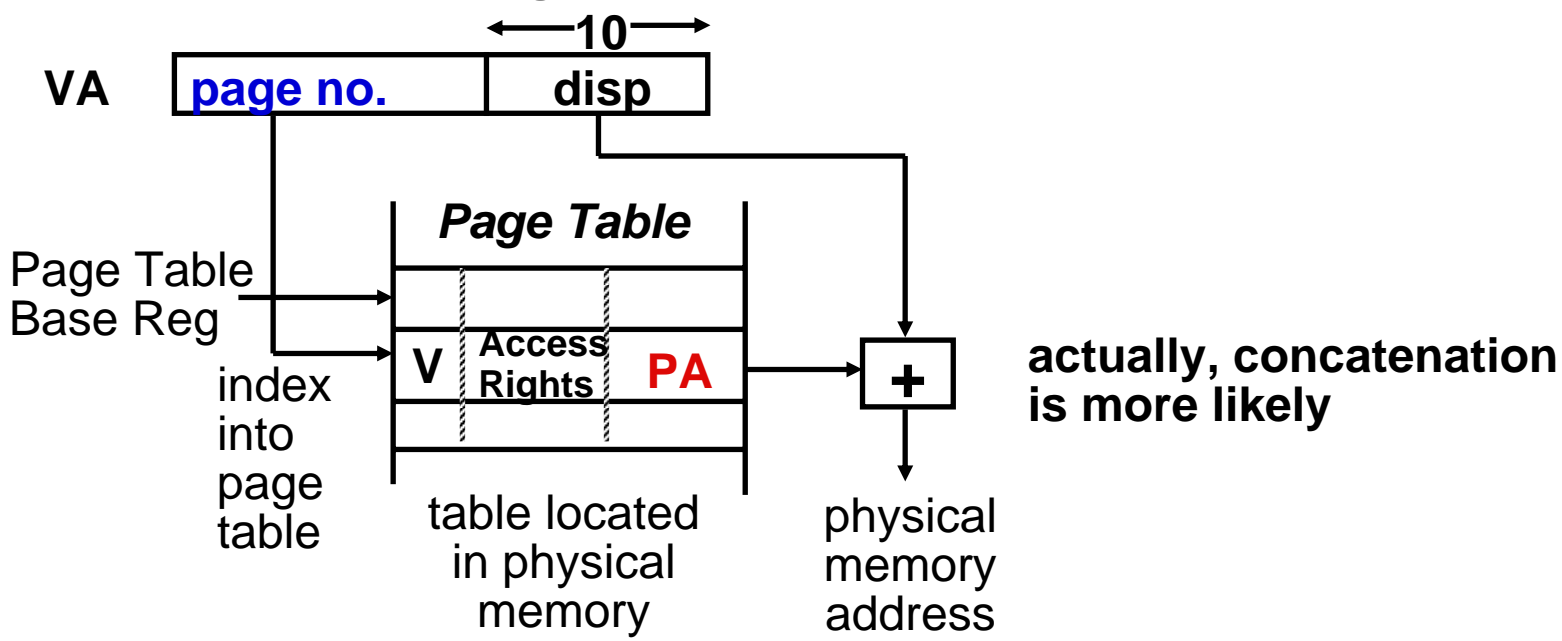
= \emptyset if data at virtual address a is not present in M



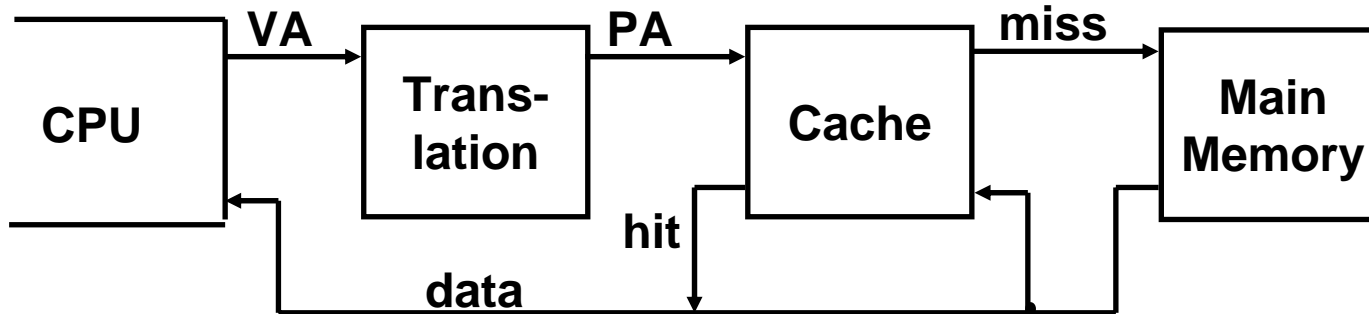
Paging Organization



Address Mapping



Virtual Address and a Cache



It takes an extra memory access to translate VA to PA

This makes cache access very expensive, and this is the "innermost loop" that you want to go as fast as possible

ASIDE: Why access cache with PA at all? VA caches have a problem!
synonym / alias problem: two different virtual addresses map to same physical address => two different cache entries holding data for the same physical address!

for update: must update all cache entries with same physical address or memory becomes inconsistent

determining this requires significant hardware, essentially an associative lookup on the physical address tags to see if you have multiple hits; or

software enforced **alias boundary**: same lsb of VA & PA > cache size

TLBs

A way to speed up translation is to use a special cache of recently used page table entries -- this has many names, but the most frequently used is *Translation Lookaside Buffer* or *TLB*

Virtual Address	Physical Address	Dirty	Ref	Valid	Access

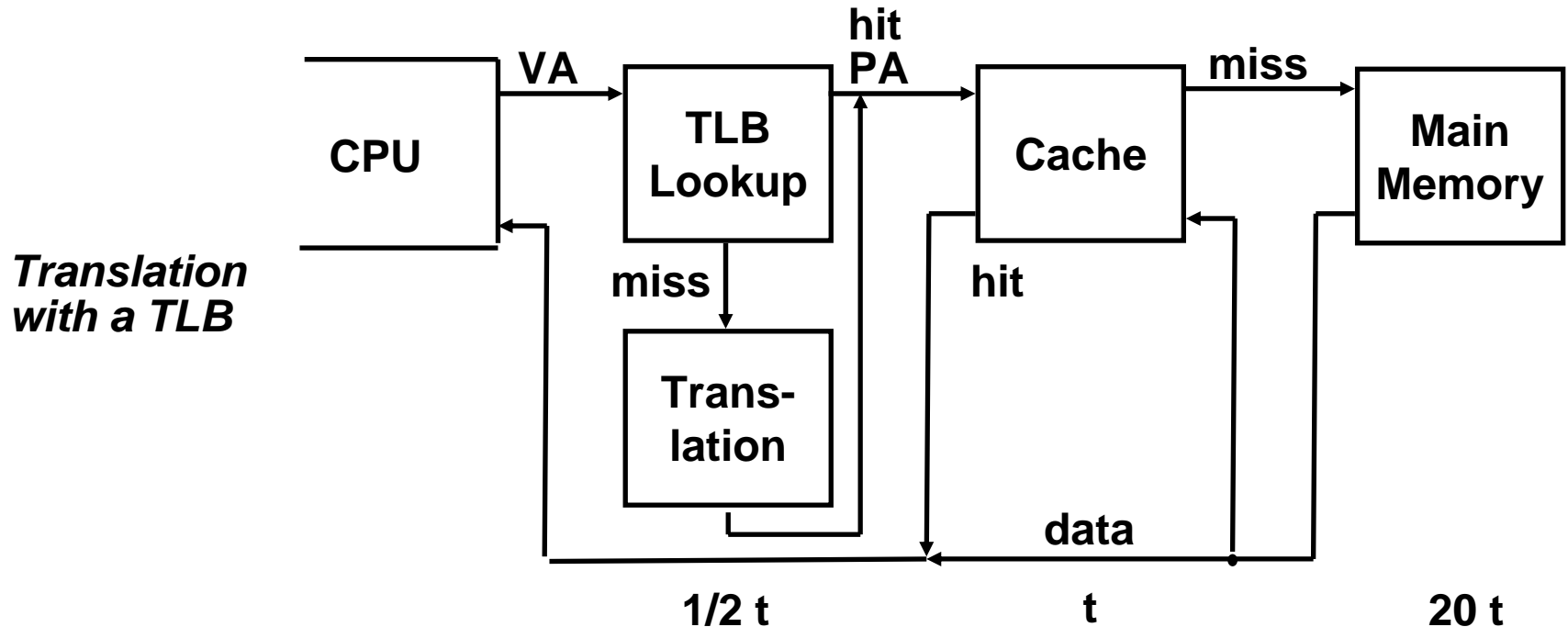
Really just a cache on the page table mappings

TLB access time comparable to cache access time
(much less than main memory access time)

Translation Look-Aside Buffers

Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped

TLBs are usually small, typically not more than 128 - 256 entries even on high end machines. This permits fully associative lookup on these machines. Most mid-range machines use small n-way set associative organizations.



Reducing Translation Time

Machines with TLBs go one step further to reduce # cycles/cache access

They overlap the cache access with the TLB access:

high order bits of the VA are used to look in the TLB while low order bits are used as index into cache

Summary #1/4:

- **The Principle of Locality:**
 - Program access a relatively small portion of the address space at any instant of time.
 - » Temporal Locality: Locality in Time
 - » Spatial Locality: Locality in Space
- **Three Major Categories of Cache Misses:**
 - **Compulsory Misses**: sad facts of life. Example: cold start misses.
 - **Capacity Misses**: increase cache size
 - **Conflict Misses**: increase cache size and/or associativity.
Nightmare Scenario: ping pong effect!
- **Write Policy:**
 - **Write Through**: needs a **write buffer**. Nightmare: WB saturation
 - **Write Back**: control can be complex

Summary #2 / 4: The Cache Design Space

- **Several interacting dimensions**

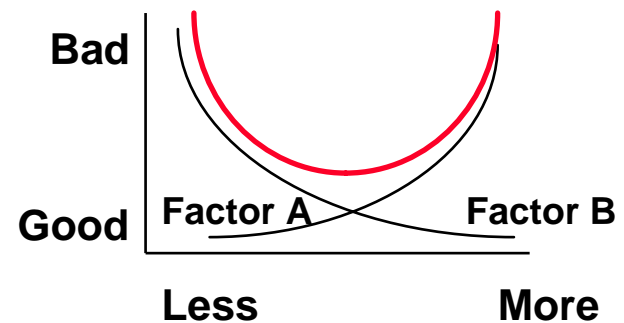
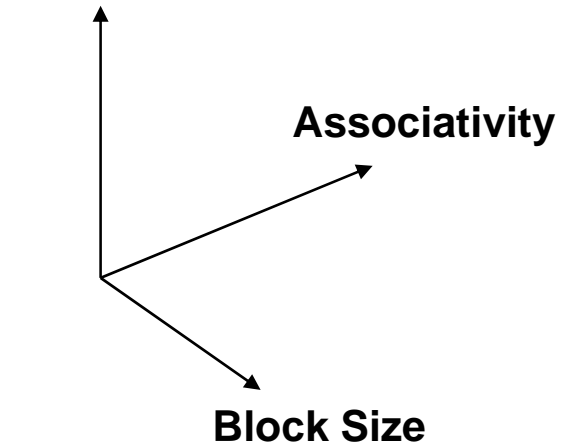
- cache size
- block size
- associativity
- replacement policy
- write-through vs write-back
- write allocation

- **The optimal choice is a compromise**

- depends on access characteristics
 - » workload
 - » use (I-cache, D-cache, TLB)
- depends on technology / cost

- **Simplicity often wins**

Cache Size



Summary #3/4: TLB, Virtual Memory

- **Caches, TLBs, Virtual Memory all understood by examining how they deal with 4 questions: 1) Where can block be placed? 2) How is block found? 3) What block is replaced on miss? 4) How are writes handled?**
- **Page tables map virtual address to physical address**
- **TLBs are important for fast translation**
- **TLB misses are significant in processor performance**
 - funny times, as most systems can't access all of 2nd level cache without TLB misses!

Summary #4/4: Memory Hierachy

- Virtual memory was controversial at the time: can SW automatically manage 64KB across many programs?
 - 1000X DRAM growth removed the controversy
- Today VM allows many processes to share single memory without having to swap all processes to disk; today VM protection is more important than memory hierarchy
- Today CPU time is a function of (ops, cache misses) vs. just $f(\text{ops})$:
What does this mean to Compilers, Data structures, Algorithms?

Review: Summary of Pipelining Basics

- **Hazards limit performance**
 - Structural: need more HW resources
 - Data: need forwarding, compiler scheduling
 - Control: early evaluation & PC, delayed branch, prediction
- **Increasing length of pipe increases impact of hazards; pipelining helps instruction bandwidth, not latency**
- **Interrupts, Instruction Set, FP makes pipelining harder**
- **Compilers reduce cost of data and control hazards**
 - Load delay slots
 - Branch delay slots
 - Branch prediction
- **Today: Longer pipelines (R4000) => Better branch prediction, more instruction parallelism?**

Advanced Pipelining and Instruction Level Parallelism (ILP)

- **ILP: Overlap execution of unrelated instructions**
- **gcc 17% control transfer**
 - 5 instructions + 1 branch
 - Beyond single block to get more instruction level parallelism
- **Loop level parallelism one opportunity, SW and HW**
- **Do examples and then explain nomenclature**
- **DLX Floating Point as example**
 - Measurements suggests R4000 performance FP execution has room for improvement

FP Loop: Where are the Hazards?

```
Loop:  LD    F0,0(R1)    ;F0=vector element
      ADDD  F4,F0,F2    ;add scalar from F2
      SD    0(R1),F4    ;store result
      SUBI  R1,R1,8     ;decrement pointer 8B (DW)
      BNEZ  R1,Loop     ;branch R1!=zero
      NOP                ;delayed branch slot
```

<i>Instruction producing result</i>	<i>Instruction using result</i>	<i>Latency in clock cycles</i>
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

- **Where are the stalls?**

FP Loop Showing Stalls

```
1 Loop: LD      F0,0(R1)      ;F0=vector element
2          stall
3          ADDD  F4,F0,F2      ;add scalar in F2
4          stall
5          stall
6 SD      0(R1),F4            ;store result
7 SUBI    R1,R1,8             ;decrement pointer 8B (DW)
8 BNEZ    R1,Loop            ;branch R1!=zero
9          stall              ;delayed branch slot
```

<i>Instruction producing result</i>	<i>Instruction using result</i>	<i>Latency in clock cycles</i>
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

- **9 clocks: Rewrite code to minimize stalls?**

Revised FP Loop Minimizing Stalls

```
1 Loop:  LD      F0,0(R1)
2          stall
3          ADDD  F4,F0,F2
4          SUBI  R1,R1,8
5          BNEZ  R1,Loop      ;delayed branch
6 SD      8(R1),F4           ;altered when move past SUBI
```

Swap BNEZ and SD by changing address of SD

<i>Instruction producing result</i>	<i>Instruction using result</i>	<i>Latency in clock cycles</i>
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

6 clocks: Unroll loop 4 times code to make faster?

Unroll Loop Four Times (straightforward way)

```
1 Loop: LD      F0,0(R1)
2      ADDD    F4,F0,F2
3      SD      0(R1),F4      ;drop SUBI & BNEZ
4      LD      F6,-8(R1)
5      ADDD    F8,F6,F2
6      SD      -8(R1),F8     ;drop SUBI & BNEZ
7      LD      F10,-16(R1)
8      ADDD    F12,F10,F2
9      SD      -16(R1),F12  ;drop SUBI & BNEZ
10     LD      F14,-24(R1)
11     ADDD    F16,F14,F2
12     SD      -24(R1),F16
13     SUBI    R1,R1,#32     ;alter to 4*8
14     BNEZ    R1,LOOP
15     NOP
```

Rewrite loop to
minimize stalls?

$15 + 4 \times (1+2) = 27$ clock cycles, or 6.8 per iteration
Assumes R1 is multiple of 4

Unrolled Loop That Minimizes Stalls

```
1 Loop: LD      F0,0(R1)
2      LD      F6,-8(R1)
3      LD      F10,-16(R1)
4      LD      F14,-24(R1)
5      ADDD    F4,F0,F2
6      ADDD    F8,F6,F2
7      ADDD    F12,F10,F2
8      ADDD    F16,F14,F2
9      SD      0(R1),F4
10     SD      -8(R1),F8
11     SD      -16(R1),F12
12     SUBI    R1,R1,#32
13     BNEZ    R1,LOOP
14     SD      8(R1),F16      ; 8-32 = -24
```

- **What assumptions made when moved code?**
 - OK to move store past SUBI even though changes register
 - OK to move loads before stores: get right data?
 - When is it safe for compiler to do such changes?

14 clock cycles, or 3.5 per iteration

When safe to move instructions?

Compiler Perspectives on Code Movement

- **Definitions:** compiler concerned about dependencies in **program**, whether or not a HW hazard depends on a given **pipeline**
- Try to schedule to avoid hazards
- (True) **Data dependencies** (RAW if a hazard for HW)
 - Instruction i produces a result used by instruction j, or
 - Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.
- If dependent, can't execute in parallel
- Easy to determine for registers (fixed names)
- Hard for memory:
 - Does $100(R4) = 20(R6)$?
 - From different loop iterations, does $20(R6) = 20(R6)$?

Compiler Perspectives on Code Movement

- Another kind of dependence called **name dependence**: two instructions use same name (register or memory location) but don't exchange data
- **Antidependence** (WAR if a hazard for HW)
 - Instruction j writes a register or memory location that instruction i reads from and instruction i is executed first
- **Output dependence** (WAW if a hazard for HW)
 - Instruction i and instruction j write the same register or memory location; ordering between instructions must be preserved.

Where are the name dependencies?

```
1 Loop: LD      F0,0(R1)
2      ADDD    F4,F0,F2
3      SD      0(R1),F4      ;drop SUBI & BNEZ
4      LD      F0,-8(R1)
2      ADDD    F4,F0,F2
3      SD      -8(R1),F4     ;drop SUBI & BNEZ
7      LD      F0,-16(R1)
8      ADDD    F4,F0,F2
9      SD      -16(R1),F4    ;drop SUBI & BNEZ
10     LD      F0,-24(R1)
11     ADDD    F4,F0,F2
12     SD      -24(R1),F4
13     SUBI    R1,R1,#32     ;alter to 4*8
14     BNEZ    R1,LOOP
15     NOP
```

How can remove them?

Compiler Perspectives on Code Movement

- **Again Name Dependence is Hard for Memory Accesses**
 - Does $100(R4) = 20(R6)$?
 - From different loop iterations, does $20(R6) = 20(R6)$?
- **Our example required compiler to know that if R1 doesn't change then:**

$$0(R1) \neq -8(R1) \neq -16(R1) \neq -24(R1)$$

There were no dependencies between some loads and stores so they could be moved by each other

Compiler Perspectives on Code Movement

- Final kind of dependence called **control dependence**
- **Example**

```
if p1 {S1;};
```

```
if p2 {S2;};
```

S1 is control dependent on p1 and S2 is control dependent on p2 but not on p1.

Compiler Perspectives on Code Movement

- **Two (obvious) constraints on control dependences:**
 - An instruction that is **control dependent** on a branch cannot be moved **before** the branch so that its execution is no longer controlled by the branch.
 - An instruction that is not **control dependent** on a branch cannot be moved to **after** the branch so that its execution is controlled by the branch.
- **Control dependencies relaxed to get parallelism; get same effect if preserve order of exceptions (address in register checked by branch before use) and data flow (value in register depends on branch)**

Where are the control dependencies?

```
1 Loop: LD      F0,0(R1)
2      ADDD    F4,F0,F2
3      SD      0(R1),F4
4      SUBI    R1,R1,8
5      BEQZ    R1,exit
6      LD      F0,0(R1)
7      ADDD    F4,F0,F2
8      SD      0(R1),F4
9      SUBI    R1,R1,8
10     BEQZ    R1,exit
11     LD      F0,0(R1)
12     ADDD    F4,F0,F2
13     SD      0(R1),F4
14     SUBI    R1,R1,8
15     BEQZ    R1,exit
....
```

When Safe to Unroll Loop?

- **Example: Where are data dependencies?
(A,B,C distinct & nonoverlapping)**

```
for (i=1; i<=100; i=i+1) {  
    A[i+1] = A[i] + C[i]; /* S1 */  
    B[i+1] = B[i] + A[i+1]; } /* S2 */
```

1. S2 uses the value, A[i+1], computed by S1 in the same iteration.
2. S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1].

This is a “**loop-carried dependence**”: between iterations

- Implies that iterations are dependent, and can't be executed in parallel
- Not the case for our prior example; each iteration was distinct

HW Schemes: Instruction Parallelism

- **Why in HW at run time?**
 - Works when can't know real dependence at compile time
 - Compiler simpler
 - Code for one machine runs well on another
- **Key idea: Allow instructions behind stall to proceed**

`DIVD F0, F2, F4`

`ADDD F10, F0, F8`

`SUBD F12, F8, F14`

- Enables out-of-order execution => out-of-order completion
- ID stage checked both for structural scoreboard dates to CDC 6600 in 1963

HW Schemes: Instruction Parallelism

- **Out-of-order execution divides ID stage:**
 - 1. Issue**—decode instructions, check for structural hazards
 - 2. Read operands**—wait until no data hazards, then read operands
- **Scoreboards allow instruction to execute whenever 1 & 2 hold, not waiting for prior instructions**
- **CDC 6600: In order issue, out of order execution, out of order commit (also called completion)**

Reference:

**“Computer Architecture – A Quantitative Approach”,
chp.1 - 4**

-- By John L Hennessy & David A. Patterson