Chapter 1

Computer Abstractions and Technology

Performance Summary

The BIG Picture

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, T_c

Concluding Remarks

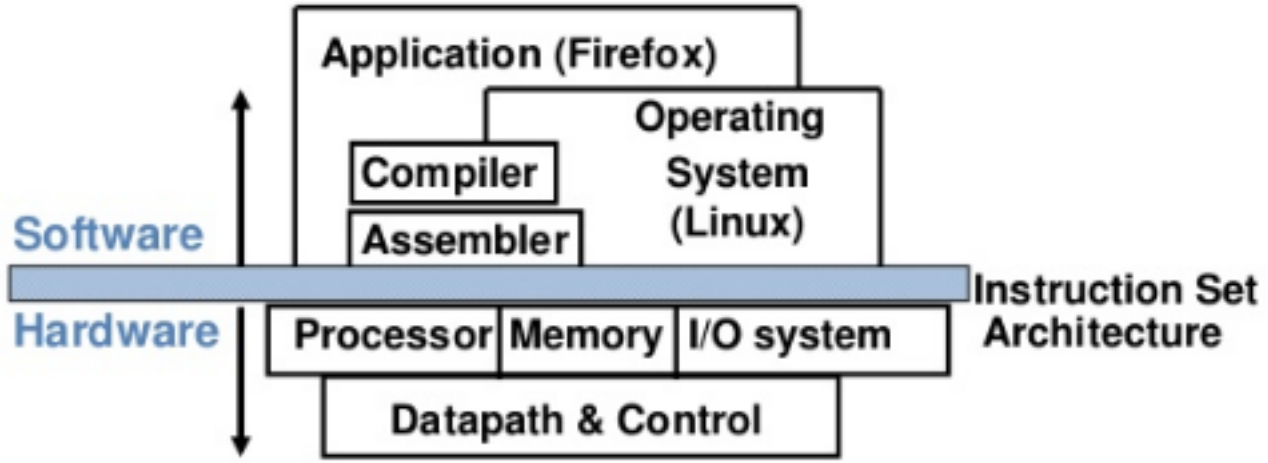
- Cost/performance is improving
  - Due to underlying technology development
- Hierarchical layers of abstraction
  - In both hardware and software
- Instruction set architecture
  - The hardware/software interface
- Execution time: the best performance measure
- Power is a limiting factor
  - Use parallelism to improve performance

Chapter 2

Instructions: Language of the Computer

Instruction Sets

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common



Register Operand Example

- C code:

```
f = (g + h) - (i + j);
```

  - f, g, h, i, and j in X19, X20, X21, X22, and X23 respectively.

- Compiled LEGv8 code:

```
ADD X9, X20, X21
ADD X10, X22, X23
SUB X19, X9, X10
```

Memory Operand Example

- C code:

```
A[12] = h + A[8]; // A is an array of doubleword
                  (8 bytes each)
```

  - h in X21, base address of A in X22

- Compiled LEGv8 code:
  - Index 8 requires offset of 64

```
LDUR    X9,[X22,#64] // U for "unscaled"
ADD     X9,X21,X9 // add the value found in A[8] to h
STUR    X9,[X22,#96] // store the result into A[12]
```

Immediate Operands

- Constant data specified in an instruction

```
ADDI X22, X22, #4
```

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Why 2's Complement?

Signed:

$+2_{10}$ = 0010

$+(-7_{10})$ = +1001

1011

= -5_{10}

Unsigned:

$+2_{10}$ = 0010

$+(9_{10})$ = +1001

1011

= 11_{10}

1. A single zero!
2. Addition can proceed w/out worrying about which operand is larger.
3. Simply add the two numbers.
4. One hardware adder works for both signed and unsigned operands.

Instruction Examples

- Addition/subtraction/logical operations
  - `R format`
- Load/store
  - `D format`
- Add/subtract immediate
  - `I format`

Conditional Operations

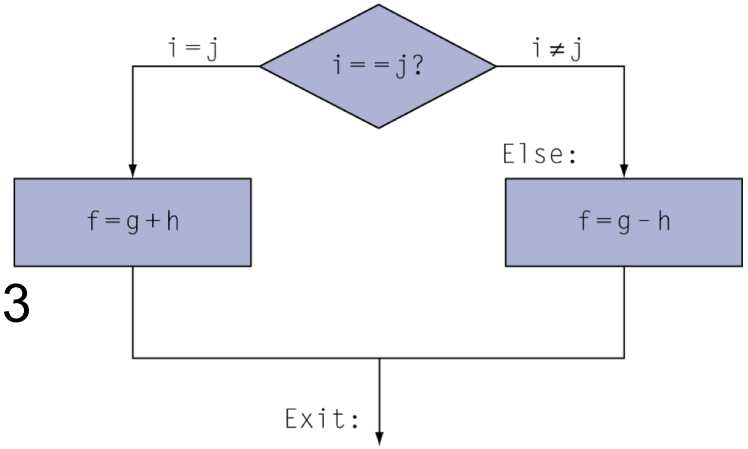
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `CBZ register, L1`
  - if (register == 0) branch to instruction labeled L1;
- `CBNZ register, L1`
  - if (register != 0) branch to instruction labeled L1;
- `B L1`
  - branch unconditionally to instruction labeled L1;

Compiling If Statements

- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

  - i in X20, j = X21, g in X22, h in X23



- Compiled LEGv8 code:

```
        SUB X9,X20,X21 // i-j, set Z flag
        CBNZ X9,Else
        ADD  X19,X22,X23 // g+h
        B Exit
Else:   SUB  X19,X22,x23 // g-h
Exit: …
```

Conditional Operations (opposites)

Condition Code		Opposite	
Code	Description	Code	Description
eq	Equal.	ne	Not equal.
hs (or cs)	Unsigned higher or same (or carry set).	lo (or cc)	Unsigned lower (or carry clear).
mi	Negative.	pl	Positive or zero.
vs	Signed overflow.	vc	No signed overflow.
hi	Unsigned higher.	ls	Unsigned lower or same.
ge	Signed greater than or equal.	lt	Signed less than.
gt	Signed greater than.	le	Signed less than or equal.
al (or omitted)	Always executed.	*There is no opposite to al.*	

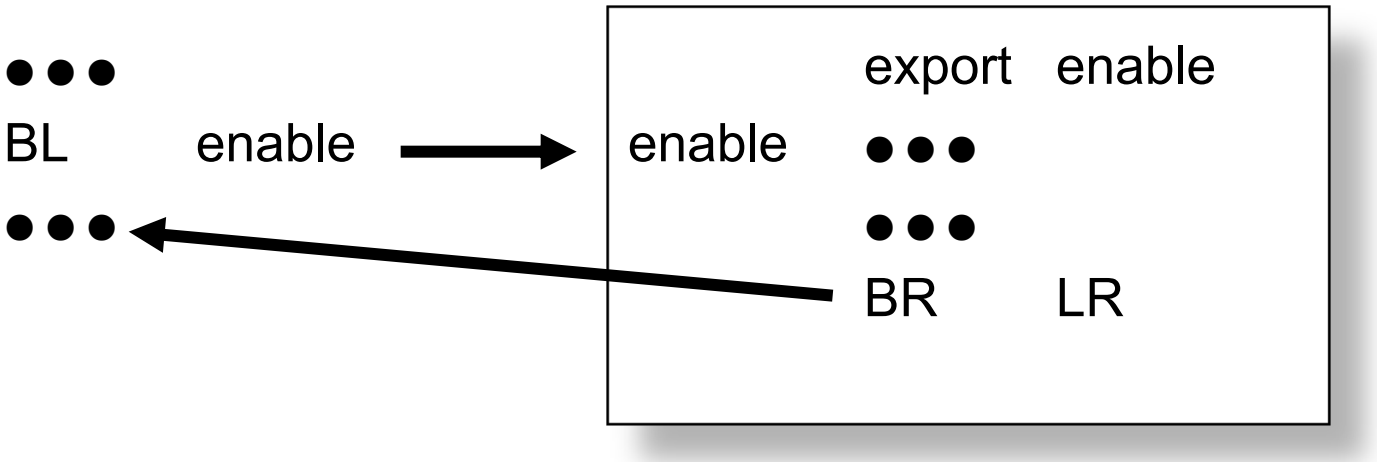
Register Usage

- X9 to X15: temporary registers
  - Not preserved by the callee
- X19 to X27: saved registers
  - If used, the callee saves and restores them
- X30 aka LR contains the return address in the calling function.

Function Call and Return

```
void enable(void) ;          //defining enable()
● ● ●
enable() ;                   //calling enable()
● ● ●
```

⇩ *Compiler*



LEGv8 Encoding Summary

Name	Fields							Comments
Field size		6 to 11 bits	5 to 10 bits	5 or 4 bits	2 bits	5 bits	5 bits	All LEGv8 instructions are 32 bits long
R-format	R	opcode	Rm	shamt		Rn	Rd	Arithmetic instruction format
I-format	I	opcode	immediate			Rn	Rd	Immediate format
D-format	D	opcode	address		op2	Rn	Rt	Data transfer format
B-format	B	opcode	address					Unconditional Branch format
CB-format	CB	opcode	address				Rt	Conditional Branch format
IW-format	IW	opcode	immediate				Rd	Wide Immediate format

Inter-Process Communication (IPC)

- Race Conditions
- Critical Regions
- Mutual Exclusion with Busy Waiting

Synchronization in LEGv8

- Load exclusive register: LDXR
- Store exclusive register: STXR
- To use:
  - Execute LDXR then STXR with same address
  - If there is an intervening change to the address, store fails (communicated with additional output register)
  - Only perform register instructions on the loaded register in between
  - ldxr rd, [rn,#0]
  - stxr rt, rm, [rn]
    - stxr success, set rm to 0
    - stxr failure, set rm to 1

Solution to: Race Condition

Same code on different processors

Assume X20 holds address of job counter,
X19 holds the job number of this processor

```
1a, 2a      jobstart:     ldxr X19, [X20, #0]
1b, 2b                    addi X9, X19, #1
1c, 2c                    stxr X9,  X10, [X20, #0]
1cc, 2cc                  cbnz X10, jobstart
1d, 2d                     <start work>
…
1z, 2z                    b  jobstart
```

Consider:
1a, 1b, 2a, 1c, 1cc
…
1d,2b,2c,1e,2 cc
…
2a, 1f, 1g, 2b, 2c, 2cc
…
2d, …, 1z

Chapter 3

Arithmetic for Computers

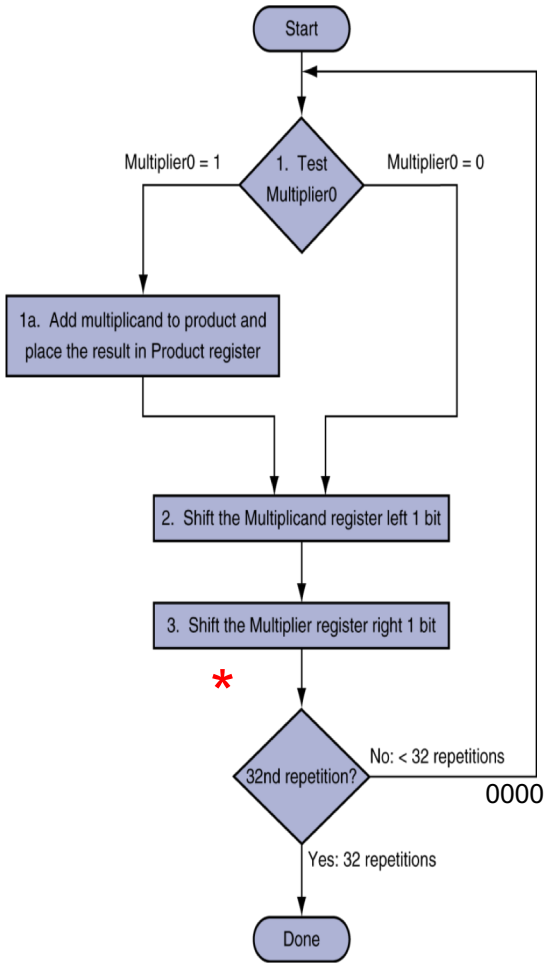
Arithmetic for Computers

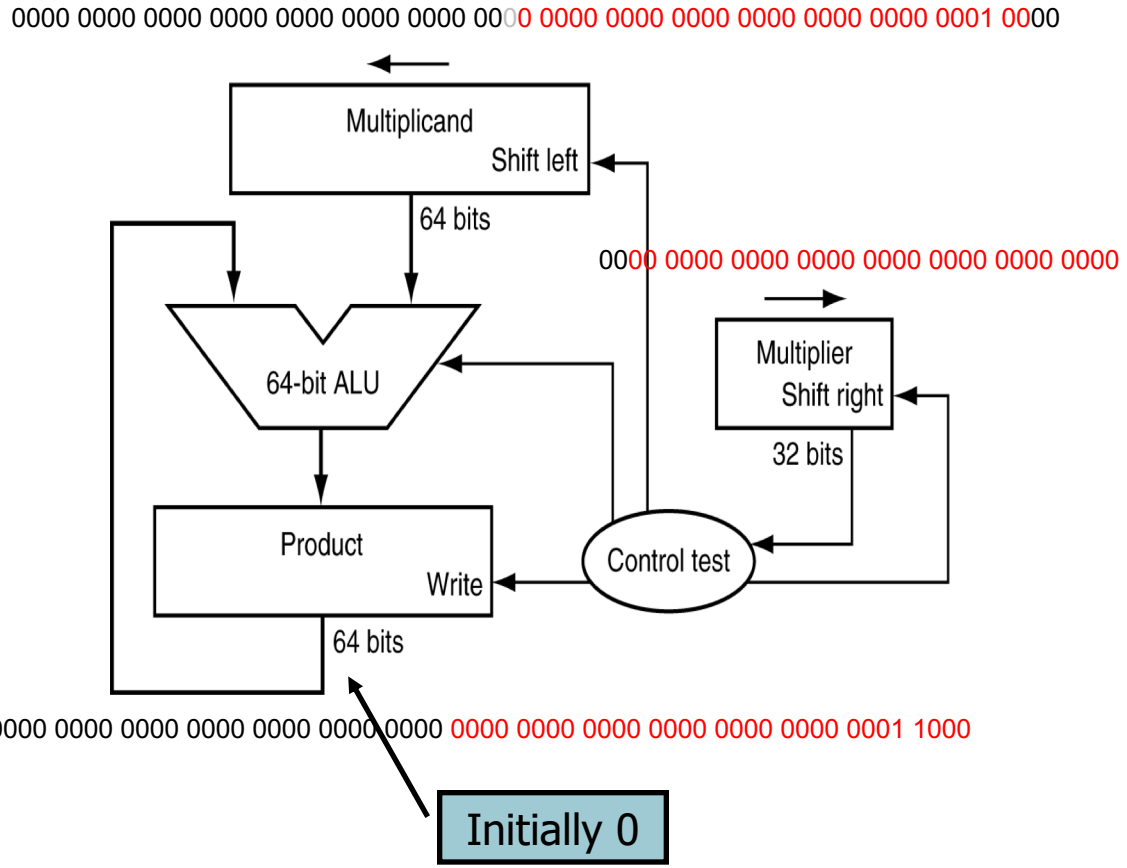
- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point real numbers
  - Representation and operations

Overflow Conditions for Addition and Subtraction

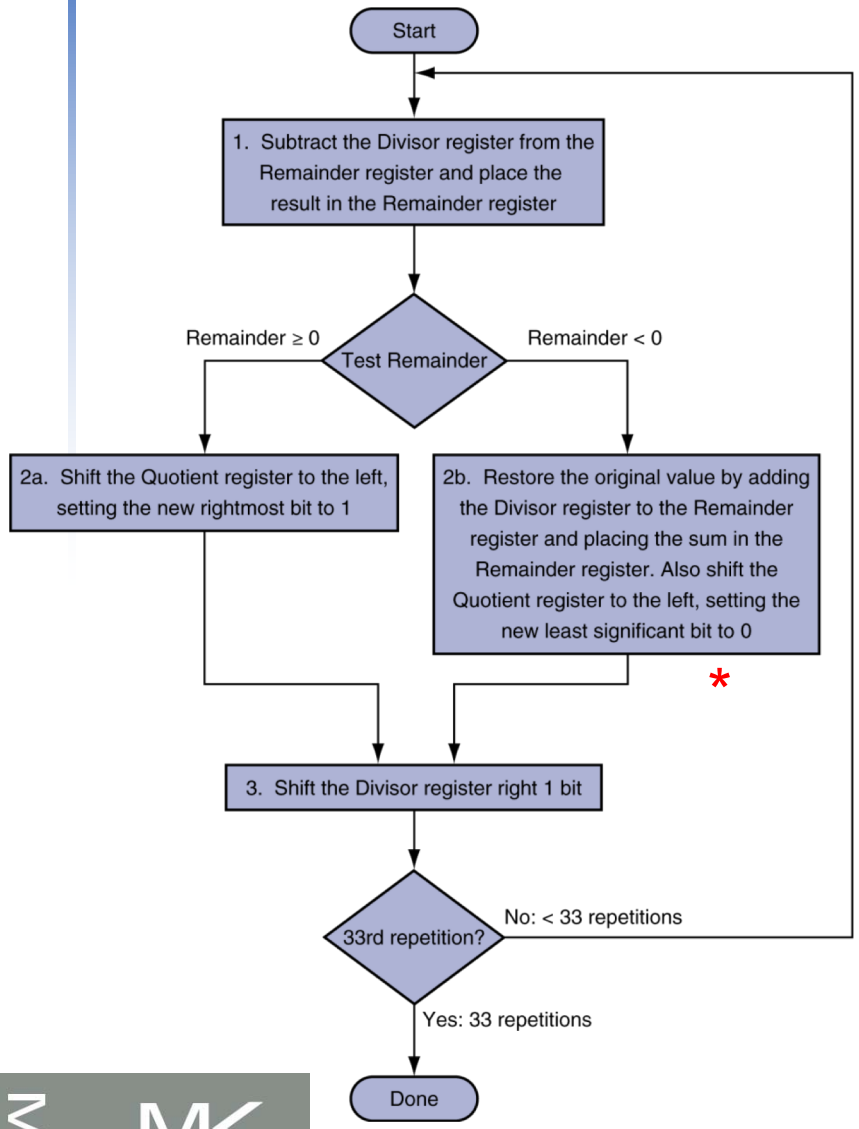
Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

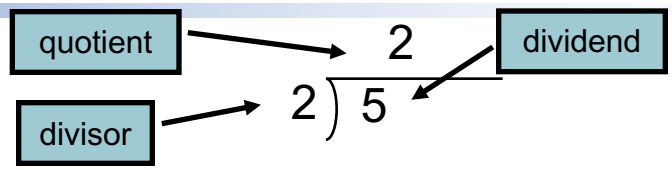
Multiplication Hardware

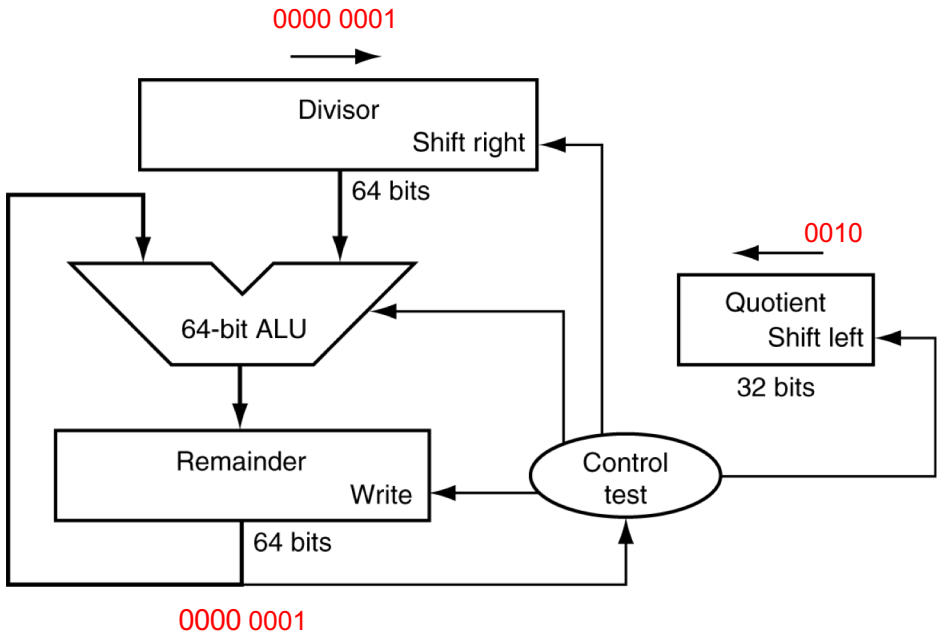




Division Hardware







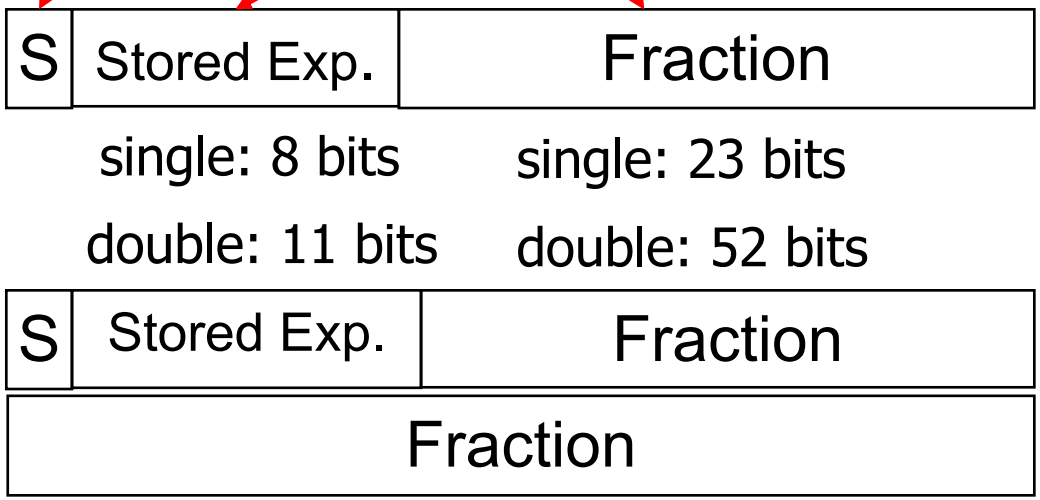
5th iteration → quotient = 0010, remainder = 0001

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56} ← normalized
  - $+0.002 \times 10^{-4}$ ← not normalized
  - $+987.02 \times 10^{9}$ ← not normalized
- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$ ← Normalized 1.0 < Significand<2.0
- Types `float` and `double` in C

IEEE Floating-Point Format

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Actual Exponent})}$$



Single precision:
Exponent (stored) = exponent (actual) + 127

Double precision:
Exponent (stored) = exponent (actual) + 1023

Floating-Point Example

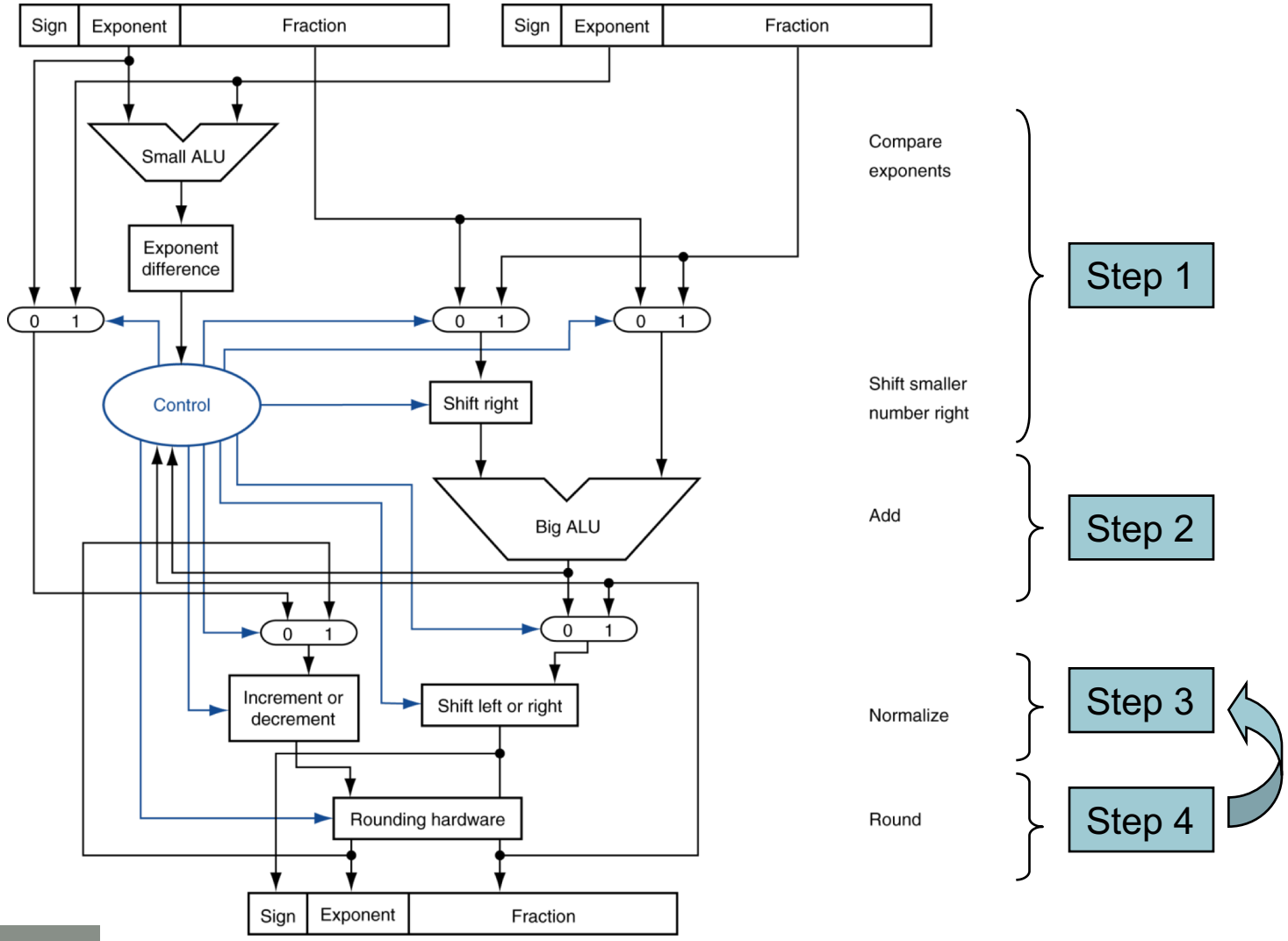
```
        S(1)   Exp+127(8u)(implied 1).Significand(23)

2.000   0      10000000     (1).00000000000000000000000
```

2.000_{10} => 10_2 (a +ve number so S = 0)
Then normalize it ➔ 1.0×2^1
Actual exponent = 1 so the stored exponent in single precision is calculated as:
$1+127 = 128_{10}$ or 10000000_2

Zero, Infinities, and NaNs

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

FP Adder Hardware



Chapter 4

The Processor

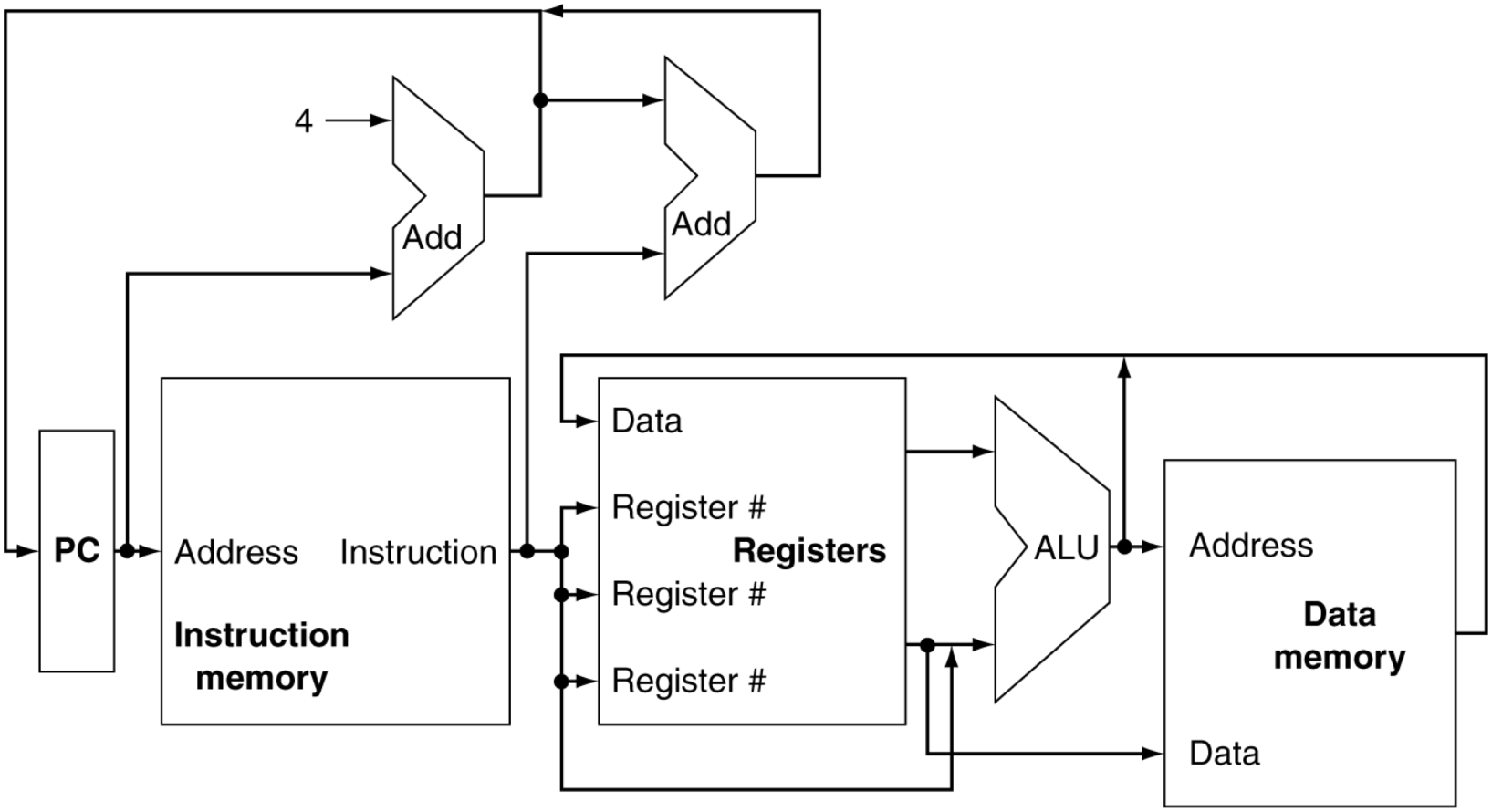
Introduction

- The performance of a computer is determined by three key factors:
  - Instruction Count (IC) → Determined by ISA and Compiler
  - Clock Cycles per Instruction (CPI)
  - Clock Cycle Time (T_c) → Determined by CPU H/W

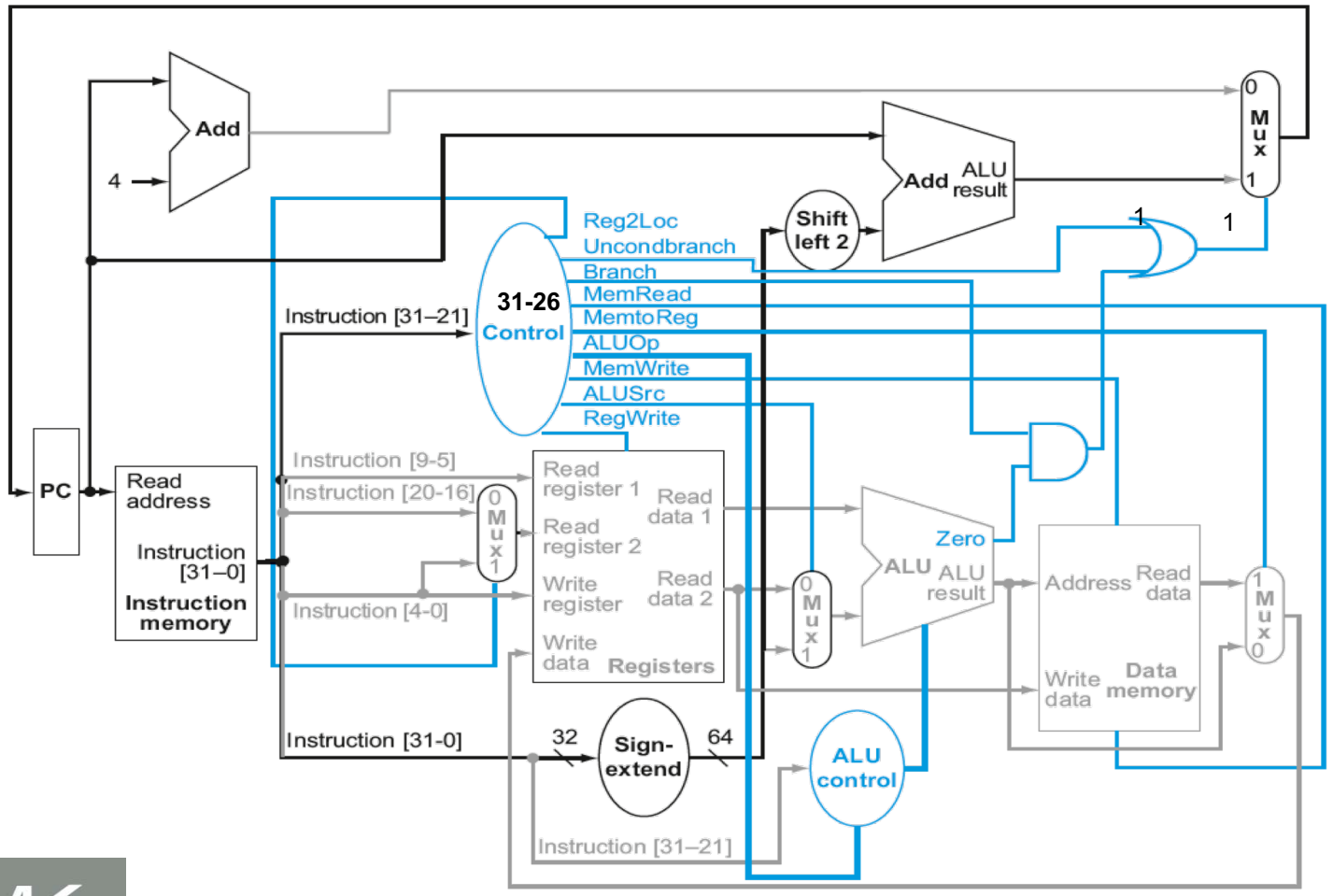
$$\text{Execution Time (seconds/program)} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Chapter 2 & 3
  - The compiler and the instruction set architecture determine the instruction count required for a given program
- Chapter 4 :
  - How implementation of the processor determines both clock cycle time and the CPI.

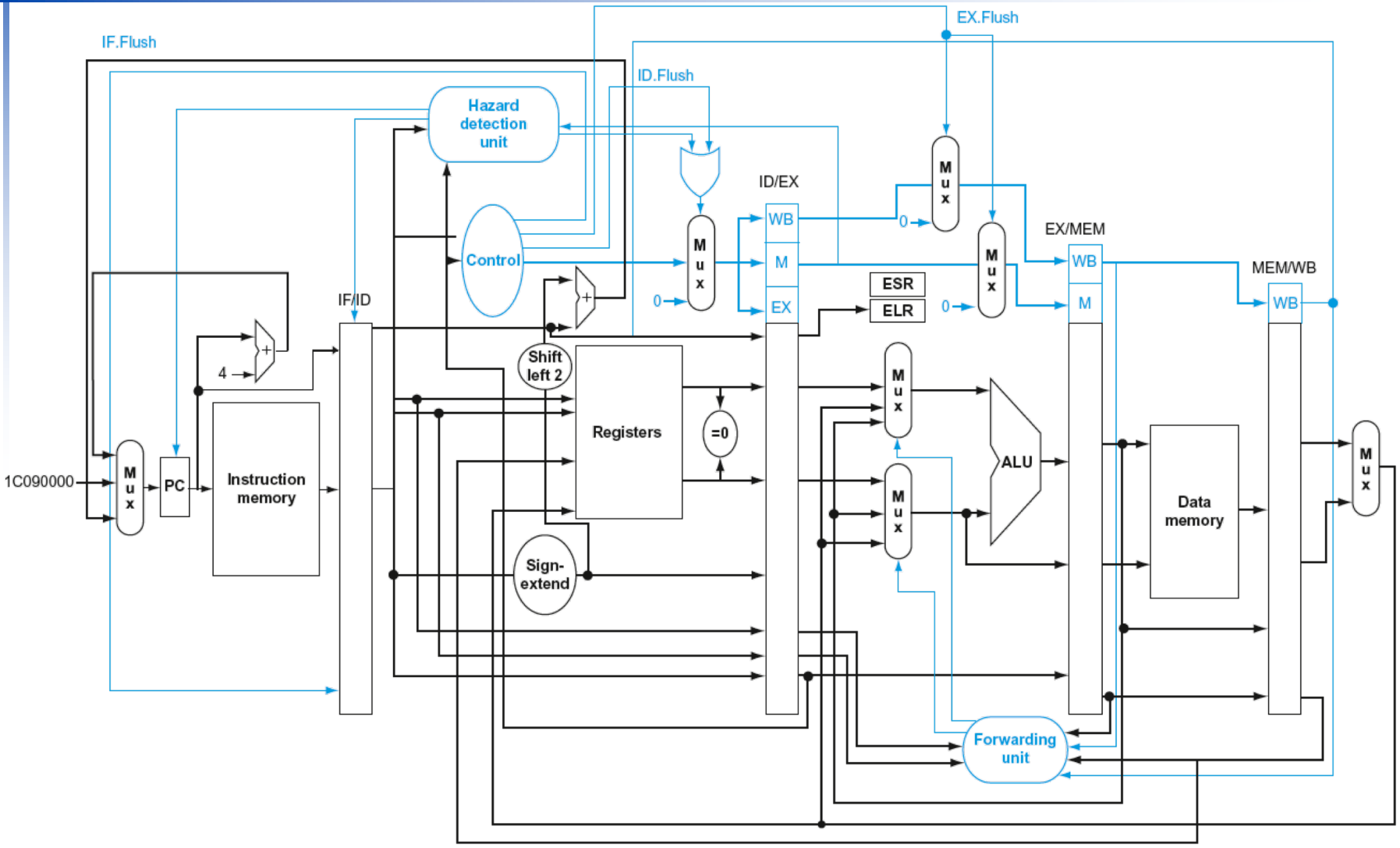
Starting Point: CPU Overview



Middle Point: Datapath With All IS Added



End Point: Pipeline with Exceptions



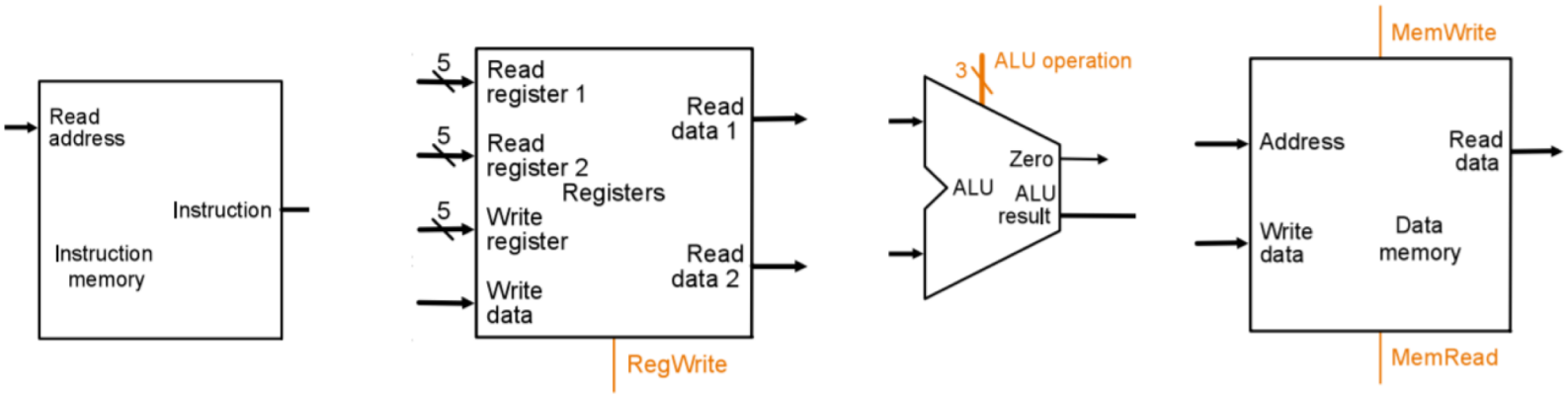
LEGv8 Instruction Formats

LEGv8

Name	Format	Example					Comments
ADD	R	1112	3	0	2	1	ADD X1, X2, X3
SUB	R	1624	3	0	2	1	SUB X1, X2, X3
ADDI	I	580	100		2	1	ADDI X1, X2, #100
SUBI	I	836	100		2	1	SUBI X1, X2, #100
LDUR	D	1986	100	0	2	1	LDUR X1, [X2, #100]
STUR	D	1984	100	0	2	1	STUR X1, [X2, #100]

Name	Fields							Comments
Field size		6 to 11 bits	5 to 10 bits	5 or 4 bits	2 bits	5 bits	5 bits	All LEGv8 instructions are 32 bits long
R-format	R	opcode	Rm	shamt		Rn	Rd	Arithmetic instruction format
I-format	I	opcode	immediate			Rn	Rd	Immediate format
D-format	D	opcode	address		op2	Rn	Rt	Data transfer format
B-format	B	opcode	address					Unconditional Branch format
CB-format	CB	opcode	address				Rt	Conditional Branch format
IW-format	IW	opcode	immediate				Rd	Wide Immediate format

Scratch Pad!

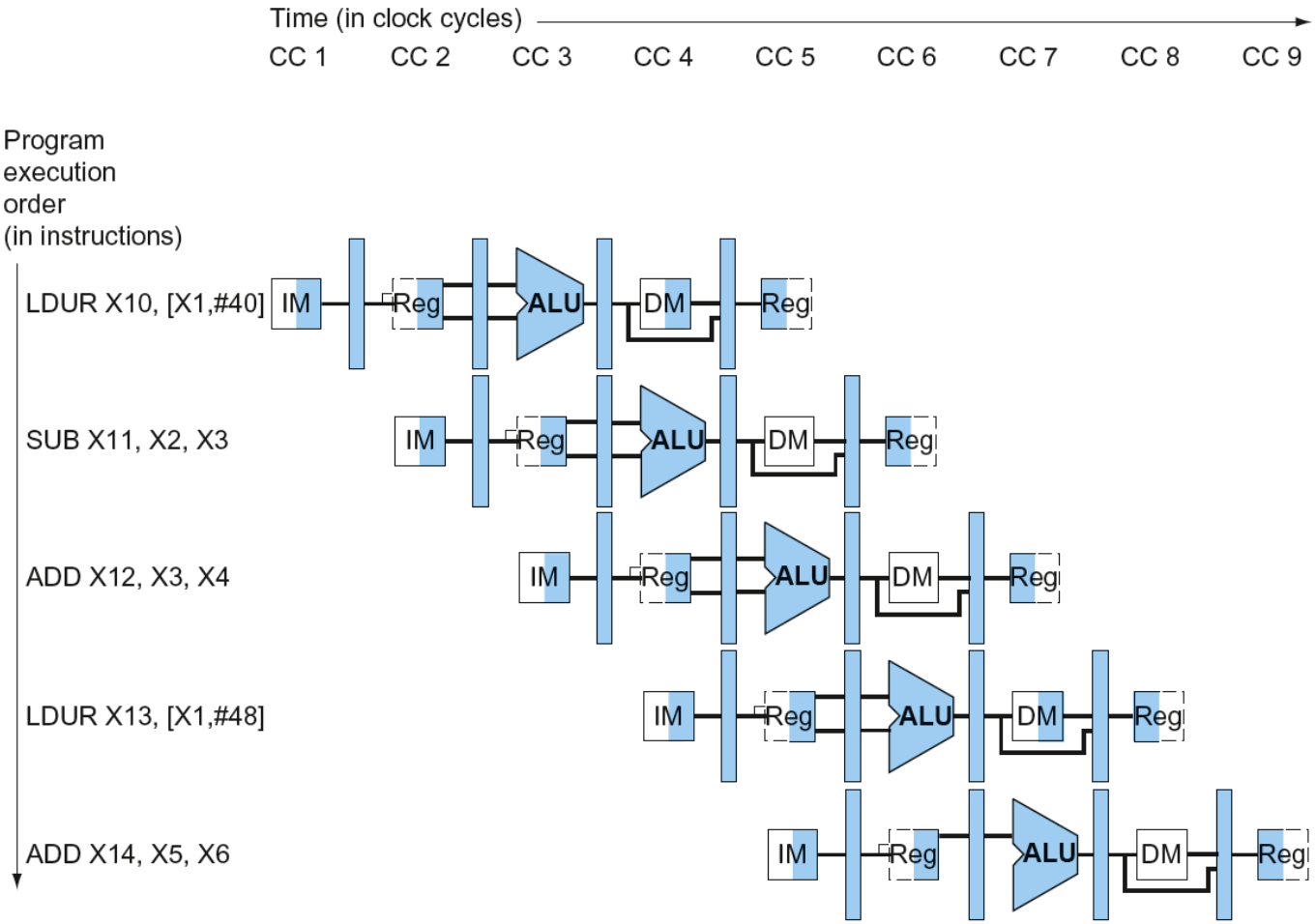

Read address
Instruction
Instruction memory
5
5
5
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
RegWrite
3
ALU operation
ALU
Zero
ALU result
MemWrite
Address
Read data
Write data
Data memory
MemRead

Pipelining and ISA Design

- LEGv8 ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

Multi-Cycle Pipeline Diagram

- Form showing resource usage

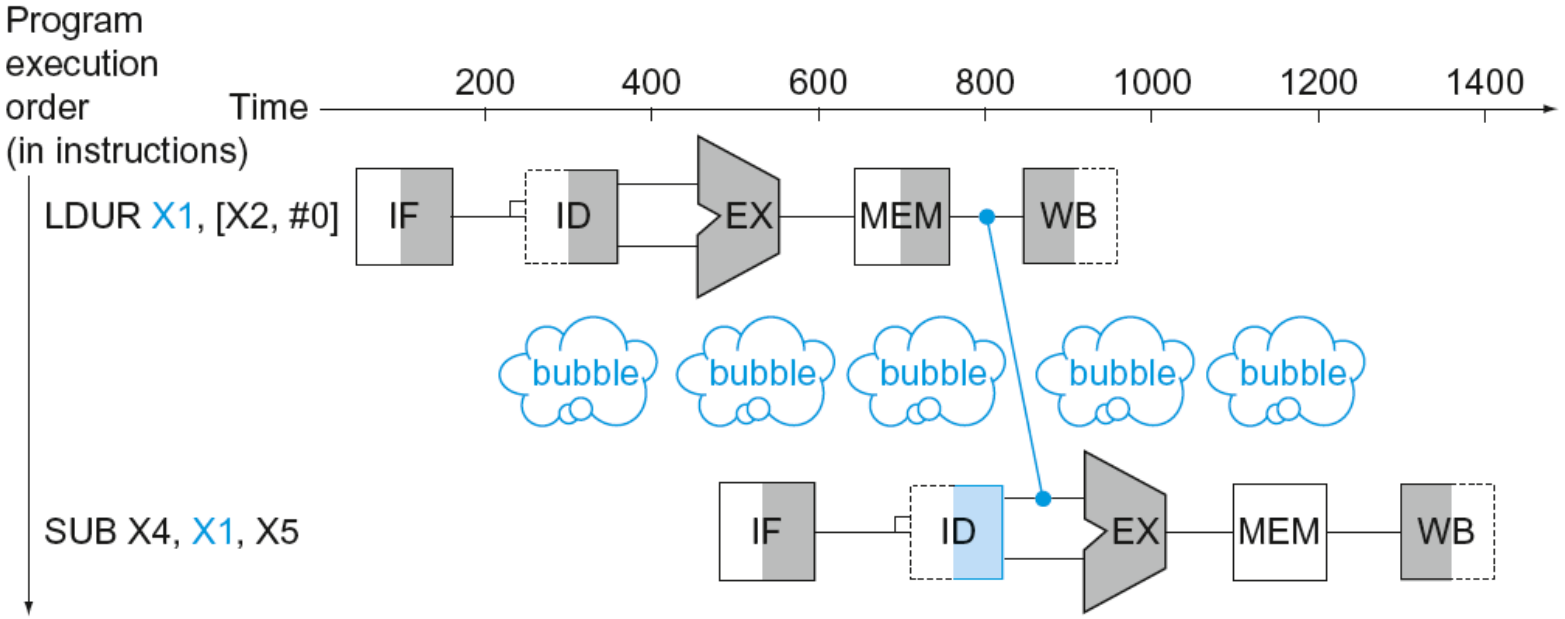


Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

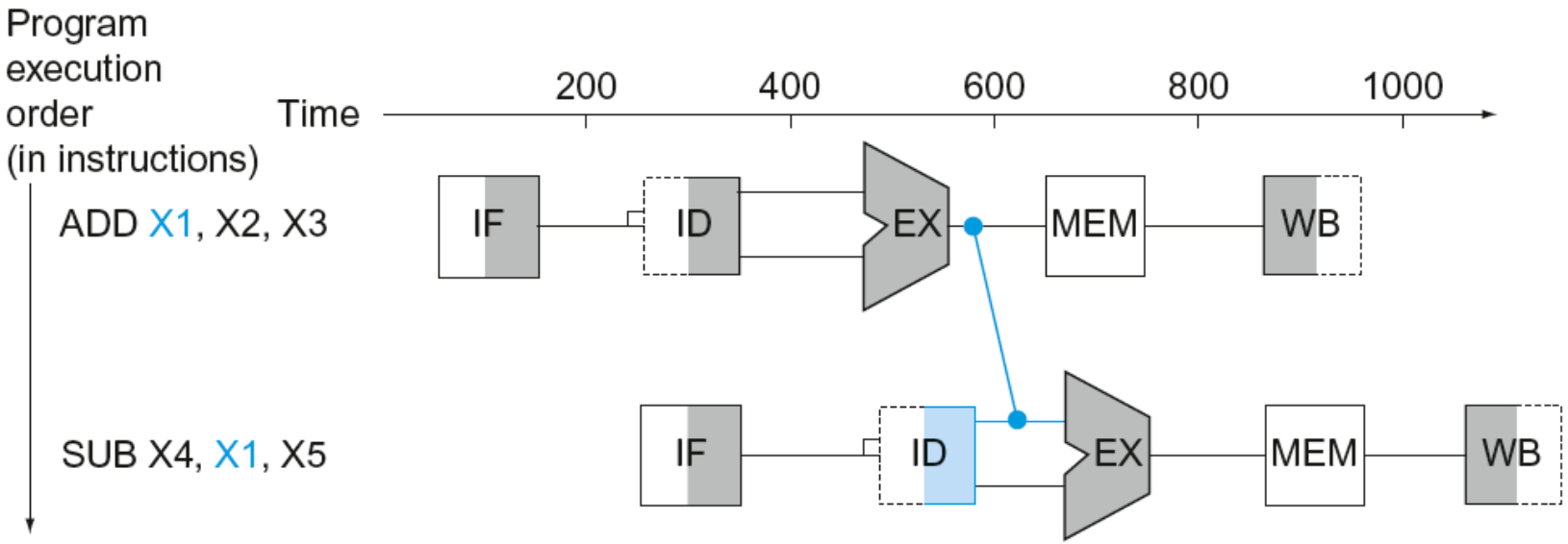
Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - ADD X19, X0, X1
    SUB X2, X19, X3

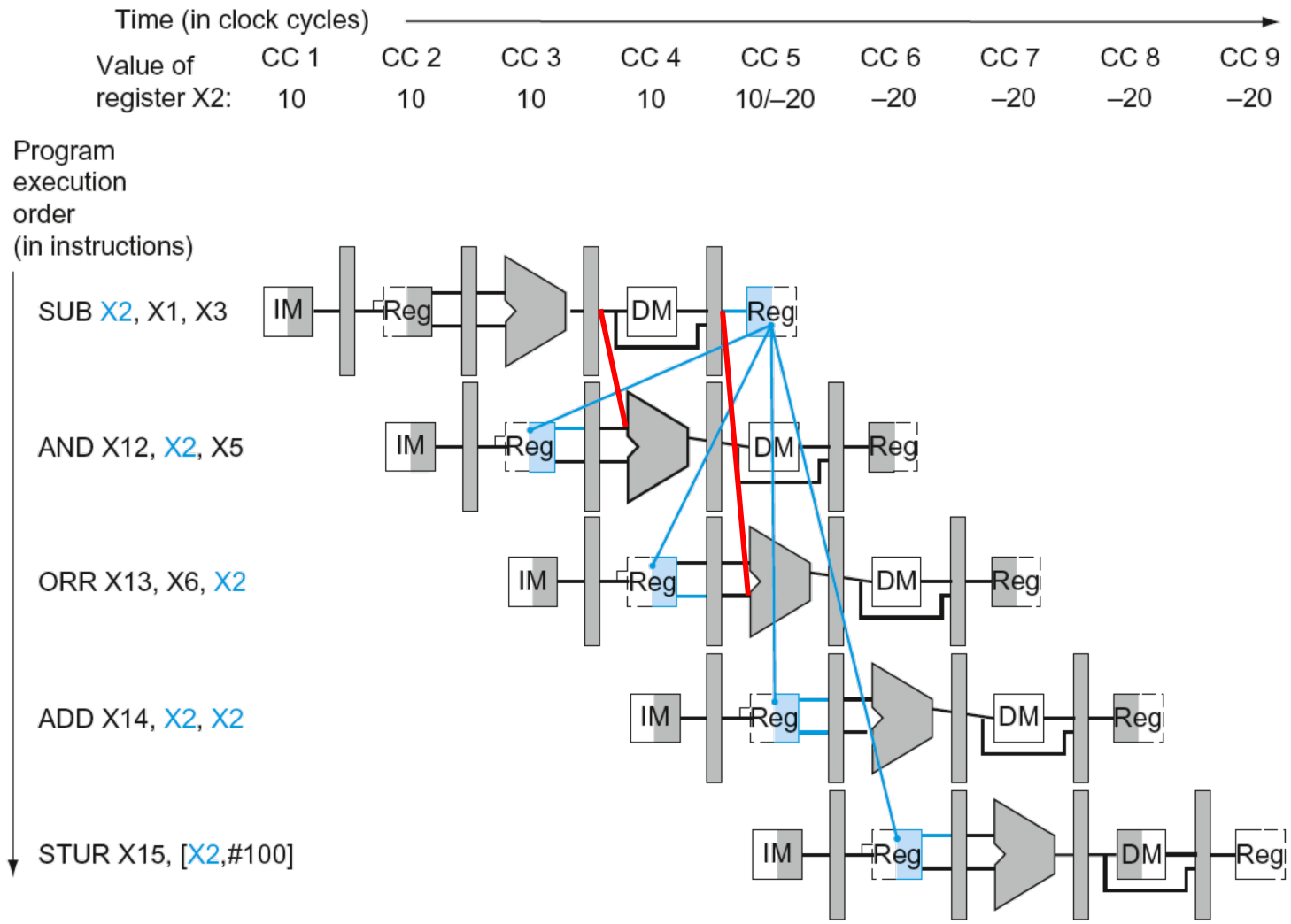


Forwarding (aka Bypassing)

- Use result when it is computed
  - Don’t wait for it to be stored in a register
  - Requires extra connections in the datapath

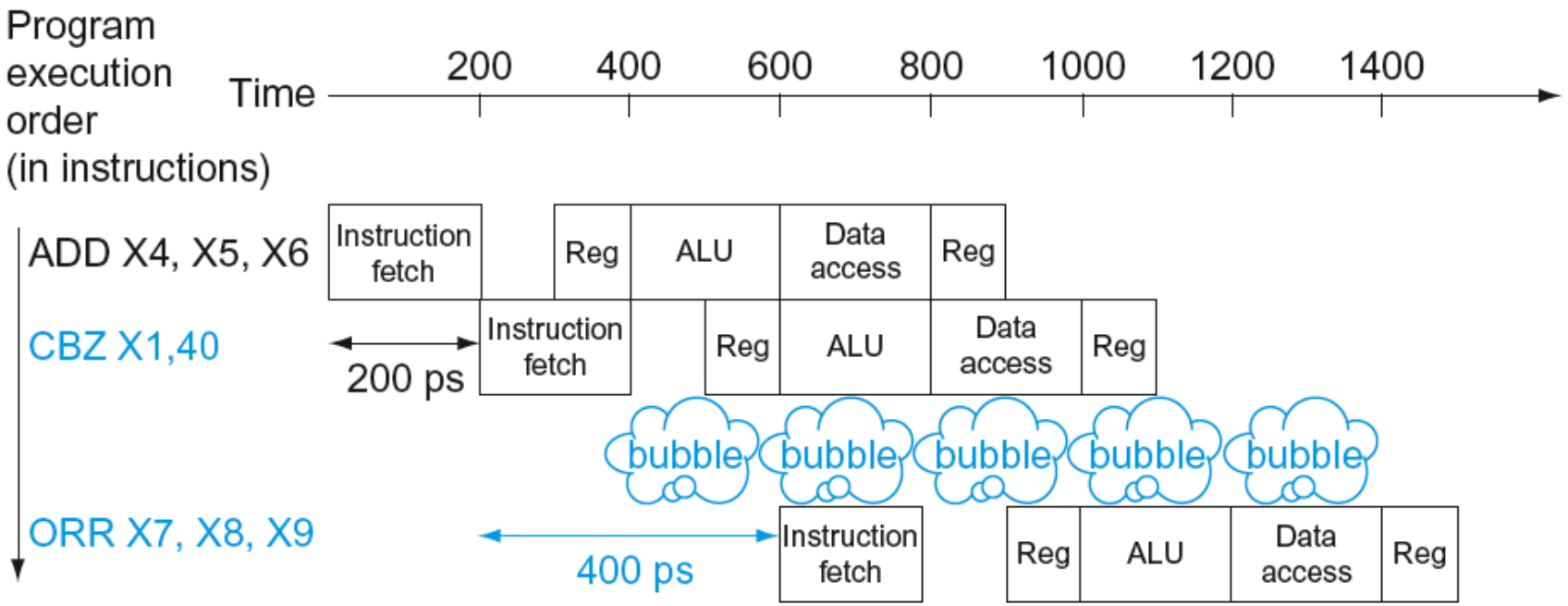


Dependencies & Forwarding

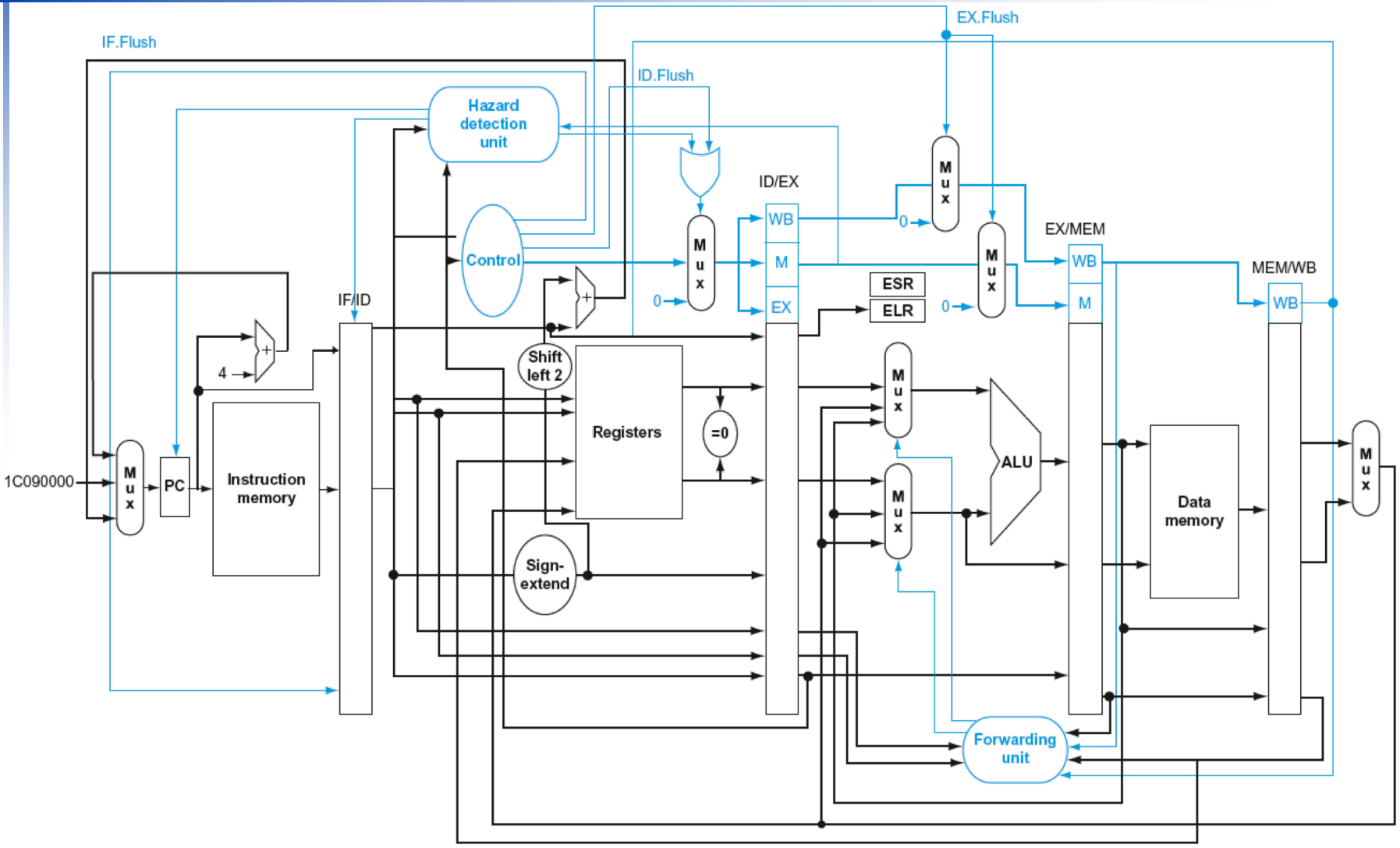


Stall on Branch

- Wait until branch outcome determined before fetching next instruction



End Point: Pipeline with Exceptions



Chapter 5

Large and Fast: Exploiting Memory Hierarchy

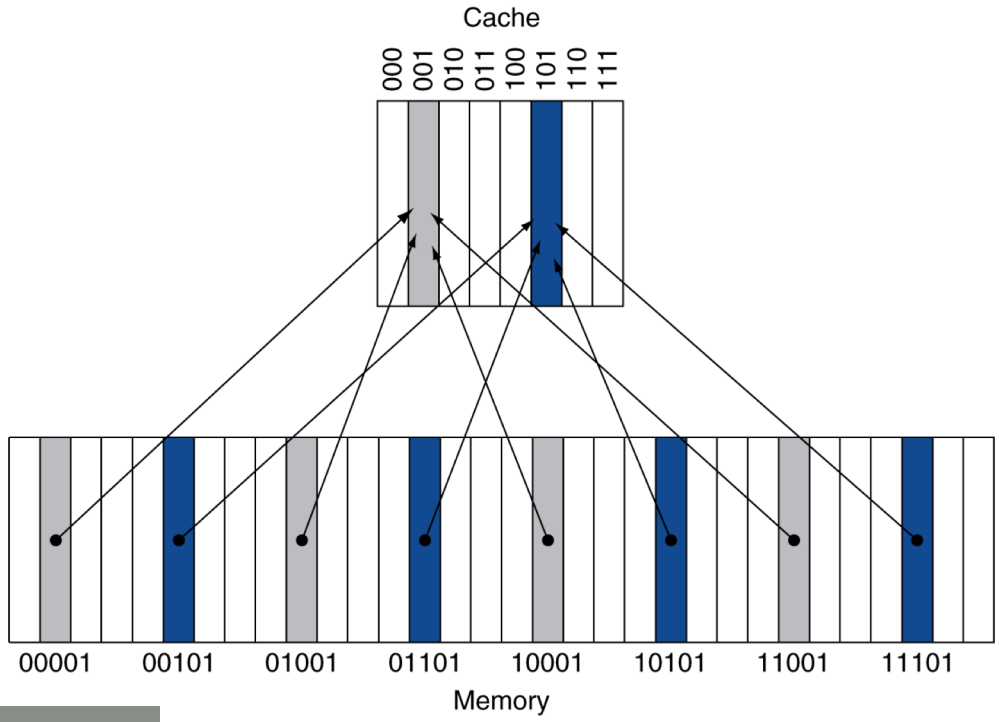
Principle of Locality

- Programs access a small proportion of their address space at any time
  - Temporal locality
    - Items accessed recently are likely to be accessed again soon
    - e.g., instructions in a loop, induction variables
  - Spatial locality
    - Items near those accessed recently are likely to be accessed soon
    - E.g., sequential instruction access, array data

	Spatial	*Temporal*
Data	arrays	loop counters
Code	no branch/jump	loop

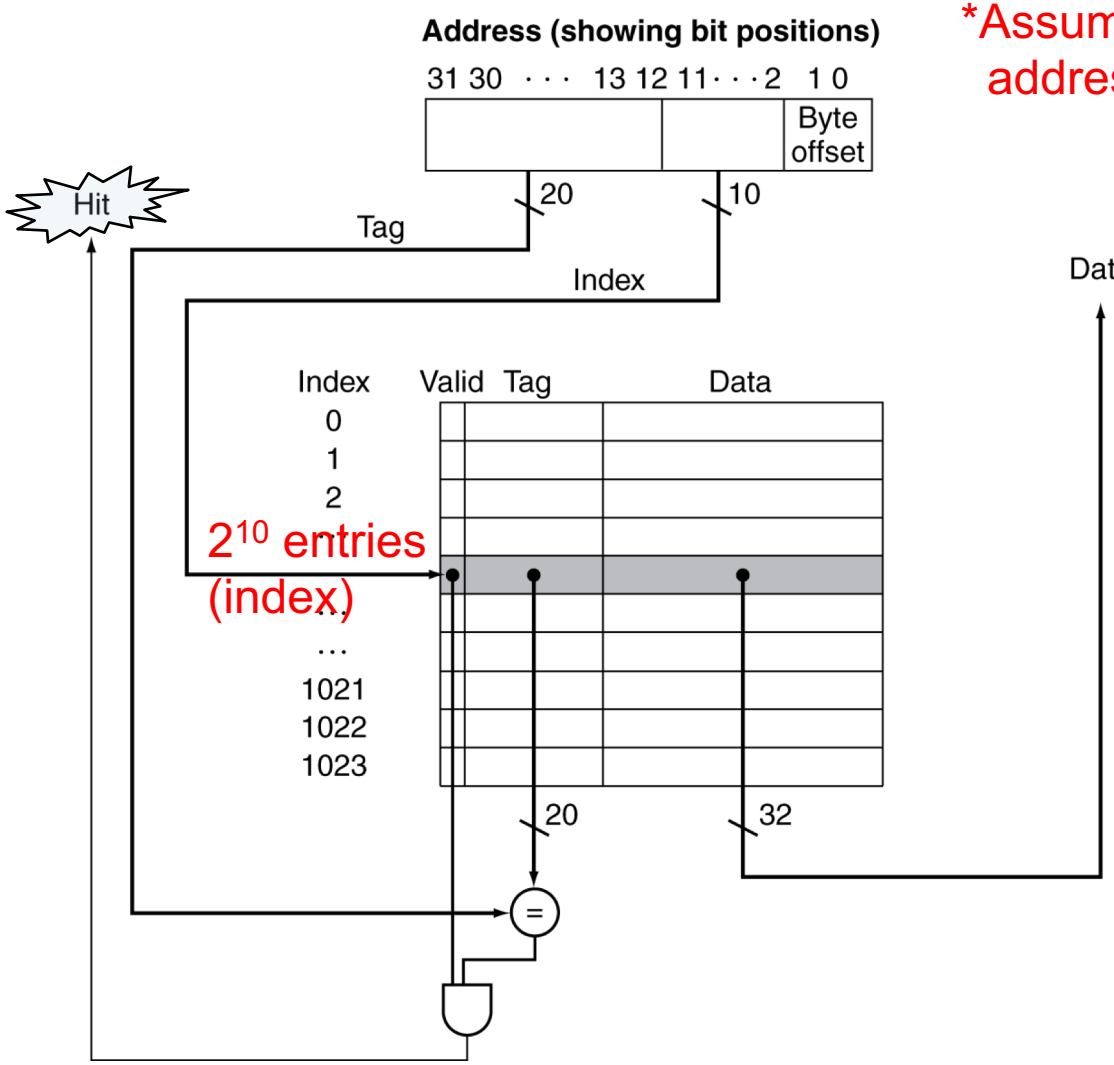
Direct Mapped Cache

- Cache location determined by address
- Direct mapped: only one choice
  - (Block address in mem) modulo (# of blocks in cache)



- # of blocks in cach is a power of 2
- Use low-order address bits

Address Subdivision



*Assuming a 32-bit (one word) addressing

* Assuming 1-word data blocks

Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
    - Instruction cache miss
      - Restart instruction fetch
    - Data cache miss
      - Complete data access

Write-Through

- On data-write hit (data to be updated exists in cache), could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: also update memory
- But memory writes take longer
  - e.g., if base CPI = 1, and 10% of instructions are stores, and write to memory taking 100 cycles
    - Effective CPI = 1 + 0.1 × 100 = 11 (a significant increase in CPI as a result of those memory writes)
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full

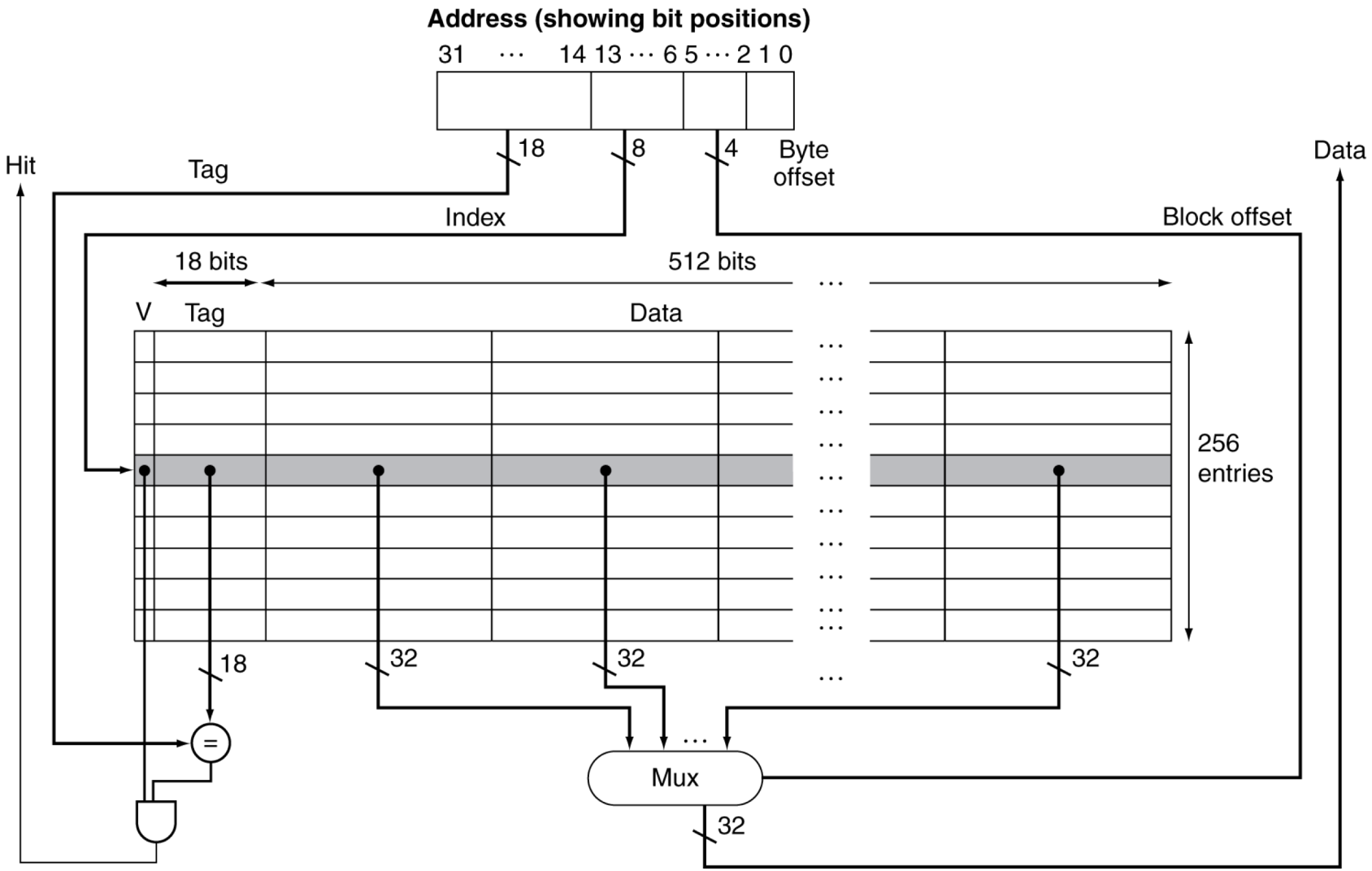
Write-Back

- Alternative: On data-write hit (data to be updated exists in cache), just update the block in cache
  - Keep track of whether each block is dirty
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first

Write Allocation

- What should happen on a write miss?
- Alternatives for write-through
  - Allocate on miss: fetch the block
  - Write around: don't fetch the block
    - Since programs often write a whole block before reading it (e.g., initialization)
- For write-back
  - Usually fetch the block

Example: Intrinsity FastMATH



Spectrum of Associativity

- For a cache with 8 entries

One-way set associative
(direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

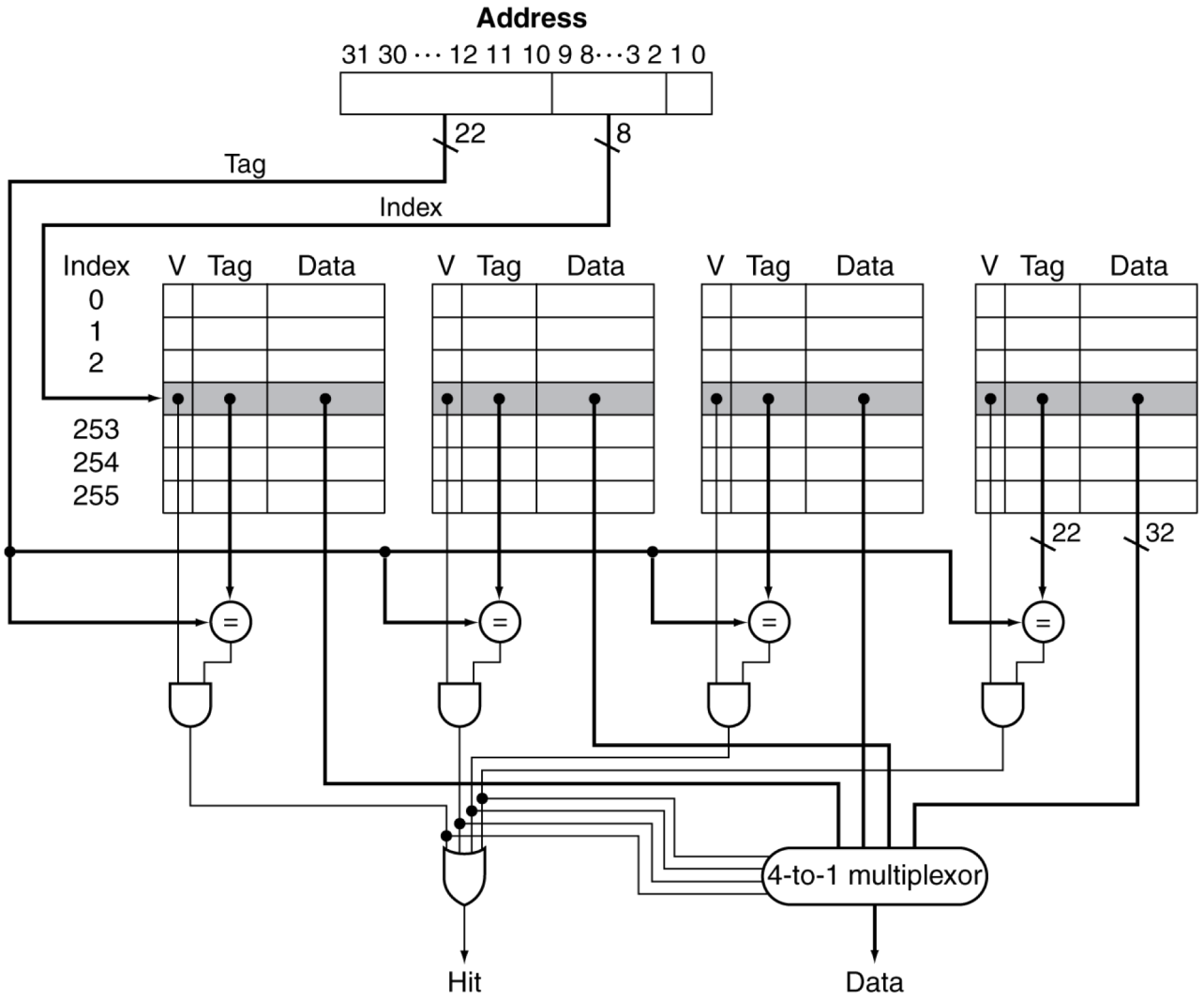
Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

Set Associative Cache Organization



Replacement Policy

- Direct mapped: no choice
- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
- Least-recently used (LRU)
  - Choose the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, too hard beyond that
- Random
  - Gives approximately the same performance as LRU for high associativity

Multilevel Caches

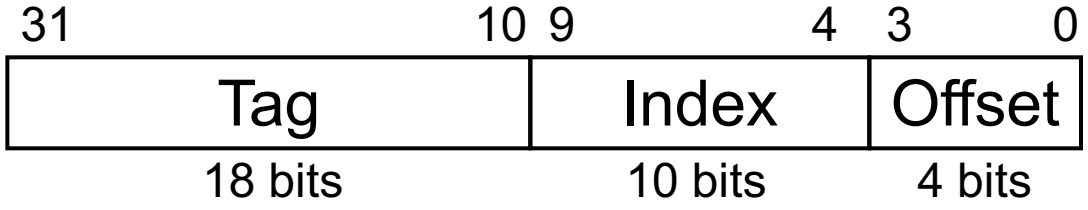
- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache

Sources of Misses

- Compulsory misses (aka cold start misses)
  - First access to a block
- Capacity misses
  - Due to finite cache size
  - A replaced block is later accessed again
- Conflict misses (aka collision misses)
  - In a non-fully associative cache
  - Due to competition for entries in a set
  - Would not occur in a fully associative cache of the same total size

Cache Control

- Example cache characteristics
  - Direct-mapped, write-back, write allocate
  - Block size: 4 words (16 bytes)
  - Cache size: 16 KB (1024 blocks)
  - 32-bit byte addresses
  - Valid bit and dirty bit per block
  - Blocking cache
    - CPU waits until access is complete

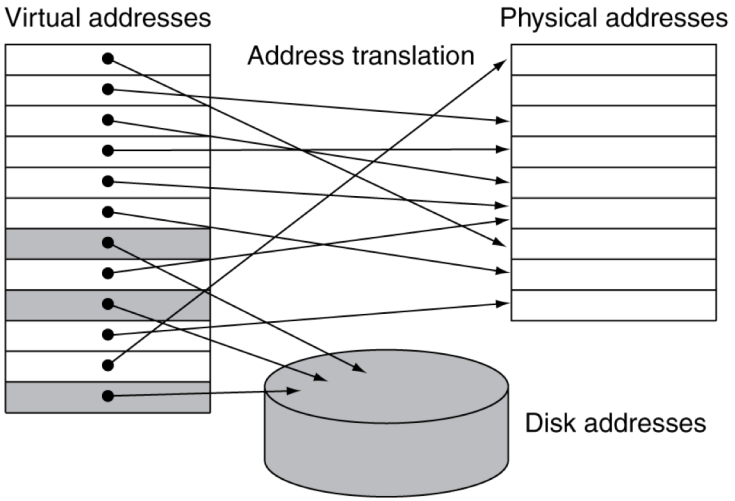


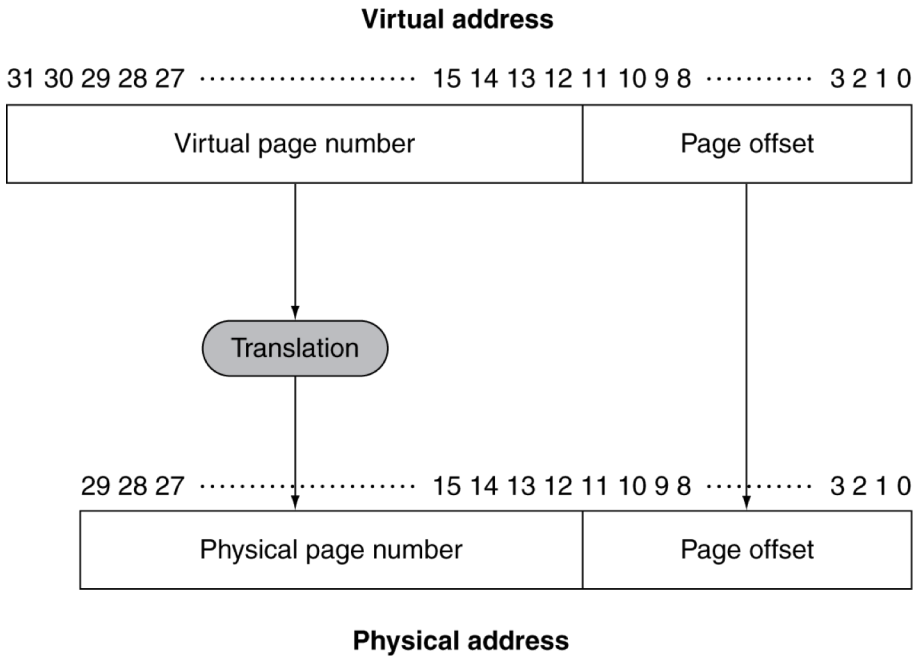
Virtual Memory

- Use main memory as a “cache” for secondary (disk) storage
  - Managed jointly by CPU hardware and the operating system (OS)
- Programs share main memory
  - Each gets a private virtual address space holding its frequently used code and data
  - Protected from other programs
- CPU and OS translate virtual addresses to physical addresses
  - VM “block” is called a page
  - VM translation “miss” is called a page fault

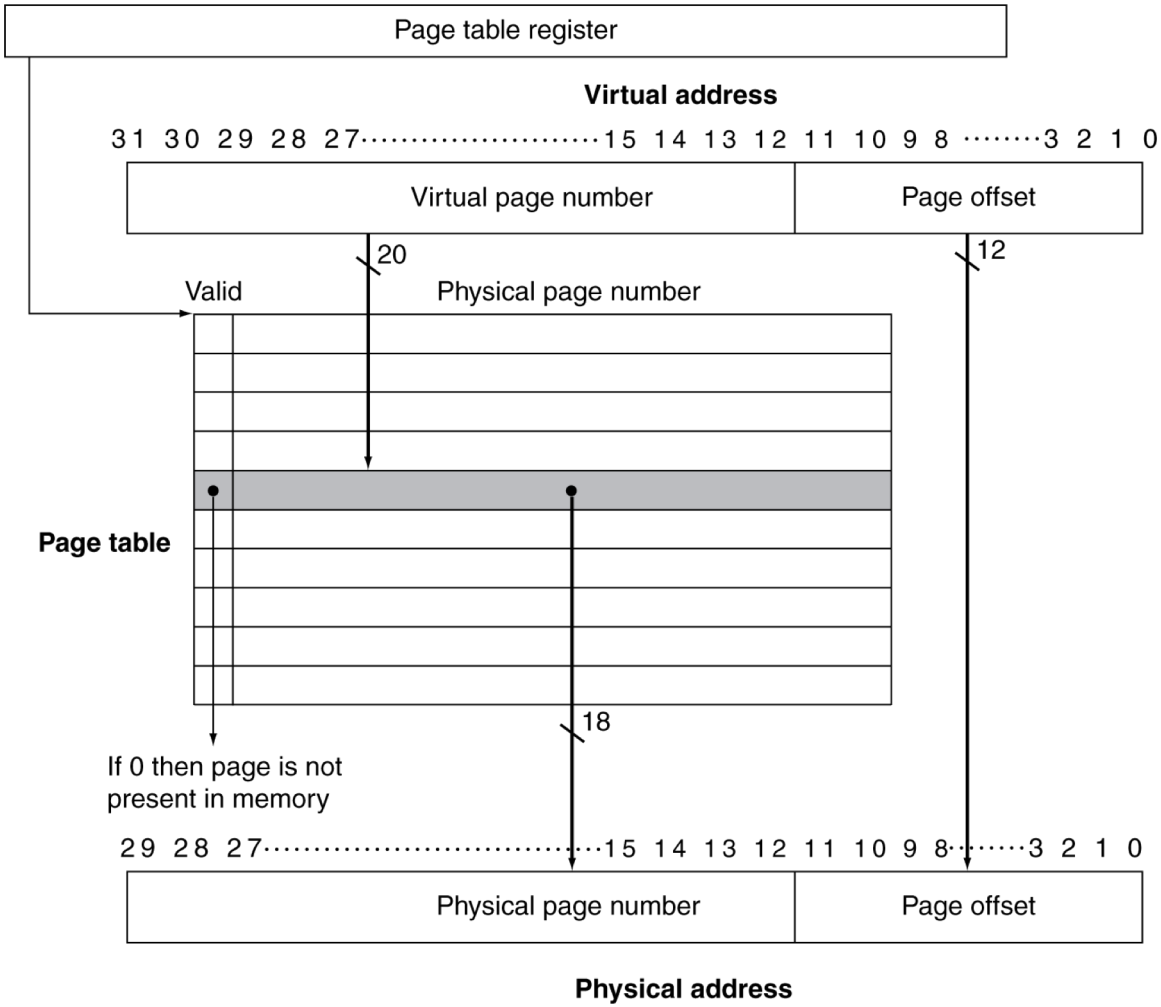
Address Translation

- Fixed-size pages (e.g., 4K)





Translation Using a Page Table



Example 2: Maping VA -> PA

Suppose

- virtual memory of 2^{32} bytes
- physical memory of 2^{24} bytes
- page size is 2^{10} (1 K) bytes

VPN	R	D	PPN
0	0	0	7
1	1	1	9
2	1	0	0
3	0	0	5
4	1	0	5
5	0	0	3
6	1	1	2
7	1	0	4
8	1	0	1
...			

1. How many pages can be stored in physical memory at once?
2. How many entries are there in the page table?
3. How many bits are necessary per entry in the page table? (Assume each entry has PPN, resident bit, dirty bit)
4. How many pages does the page table require?
5. What's the largest fraction of VM that might be resident?
6. A portion of the page table is given to the left. What is the physical address for virtual address 0x1804?

Page Replacement Algorithms

- Optimal algorithm
- Not recently used algorithm
- First-in, first-out (FIFO) algorithm
- Second-chance algorithm
- Clock algorithm
- Least recently used (LRU) algorithm
- Working set algorithm
- WSClock algorithm

Replacement and Writes

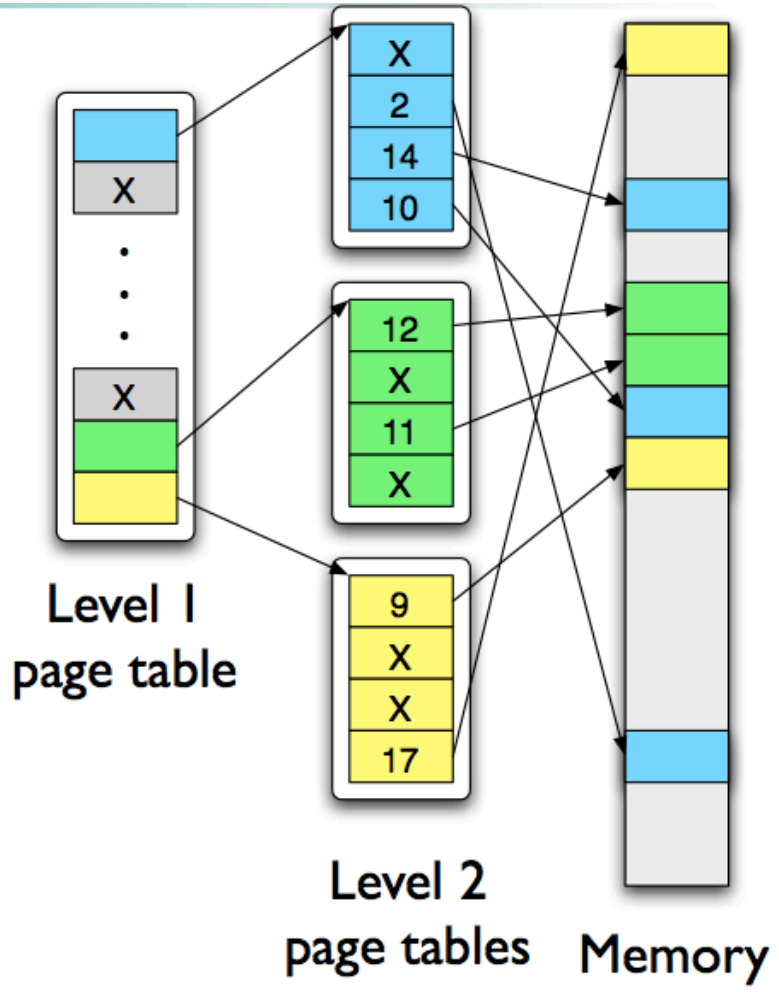
- To reduce page fault rate, prefer least-recently used (LRU) replacement
  - Reference bit (aka use bit) in PTE set to 1 on access to page
  - Periodically cleared to 0 by OS. Why?
  - A page with reference bit = 0 has not been used recently
- Disk writes take millions of cycles
  - Block at once, not individual locations
  - Write through is impractical
  - Use write-back
  - Dirty bit in PTE set when page is written

Multi-level Page Tables

- Problem: page tables can be too large
  - In 4KB page size, with 32 bit logical addresses
    - 2^{12} = 4096 (bytes), so 12 bits to get to the page byte offset
    - 2^{20} = 1 million PTEs , so 20 bits to get to the page address
- Solution: use multi-level page tables
  - 1st level page table has pointers to 2nd level page tables
  - 2nd level page table has actual physical page numbers in it
- http://www.youtube.com/watch?v=Z4kSOv49GNc

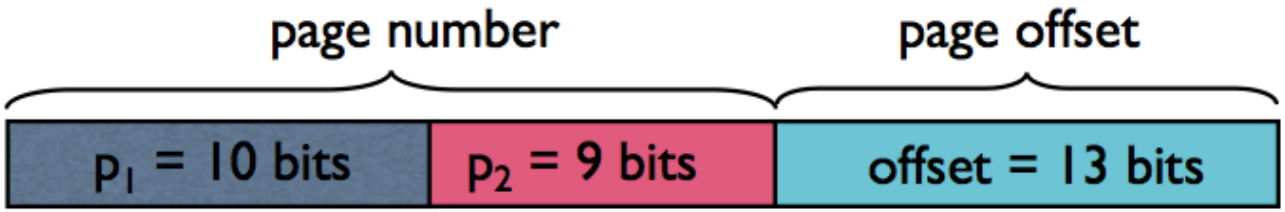
Multi-level Page Tables

- Problem: page tables can be too large
  - 2^{32} bytes in 4KB pages ⇒ 1 million PTEs
- Solution: use multi-level page tables
  - "Page size" in first page table is large (megabytes)
  - PTE marked invalid in first page table needs no 2nd level page table
- 1st level page table has pointers to 2nd level page tables
- 2nd level page table has actual physical page numbers in it

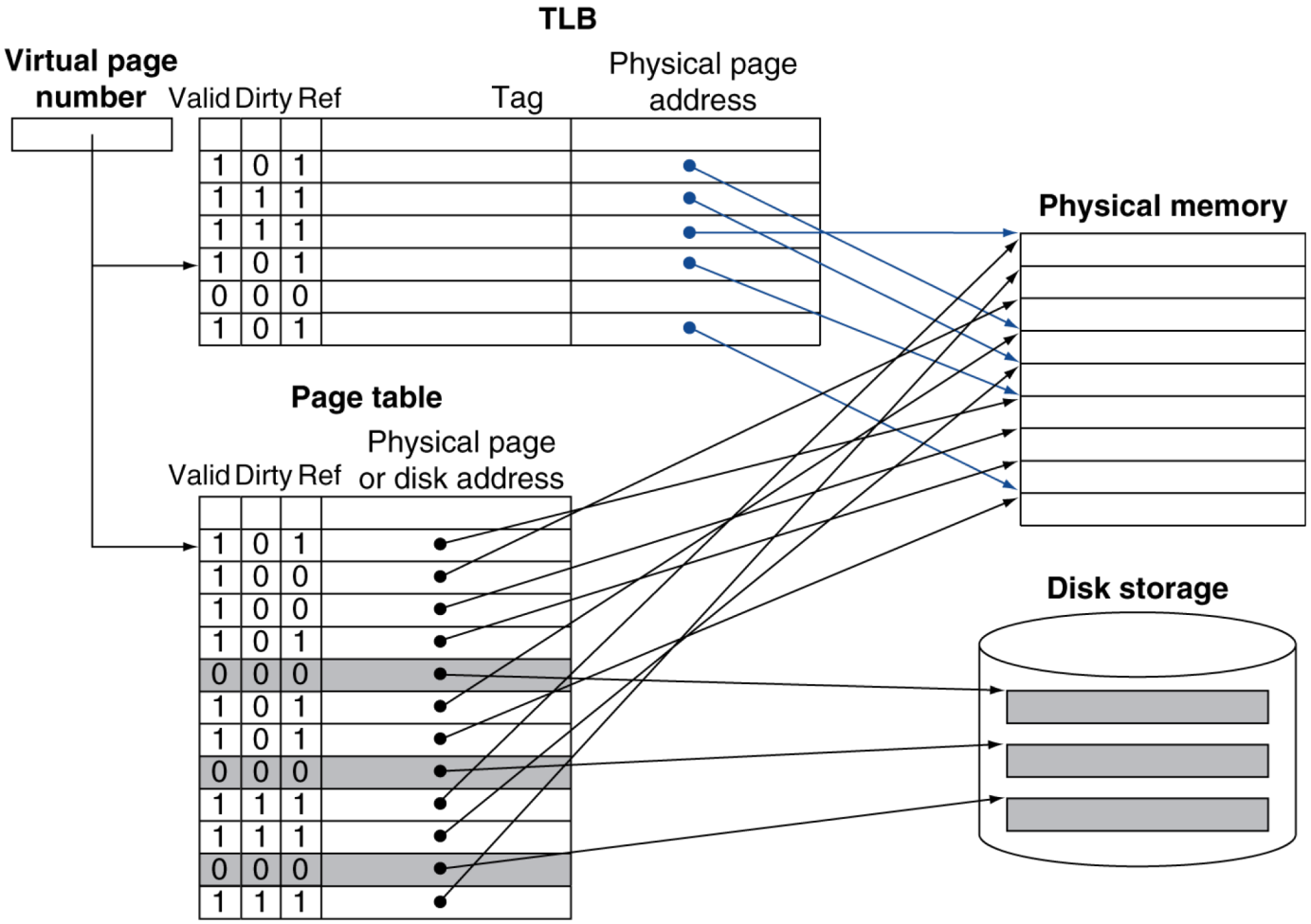


Multi-level Page Tables

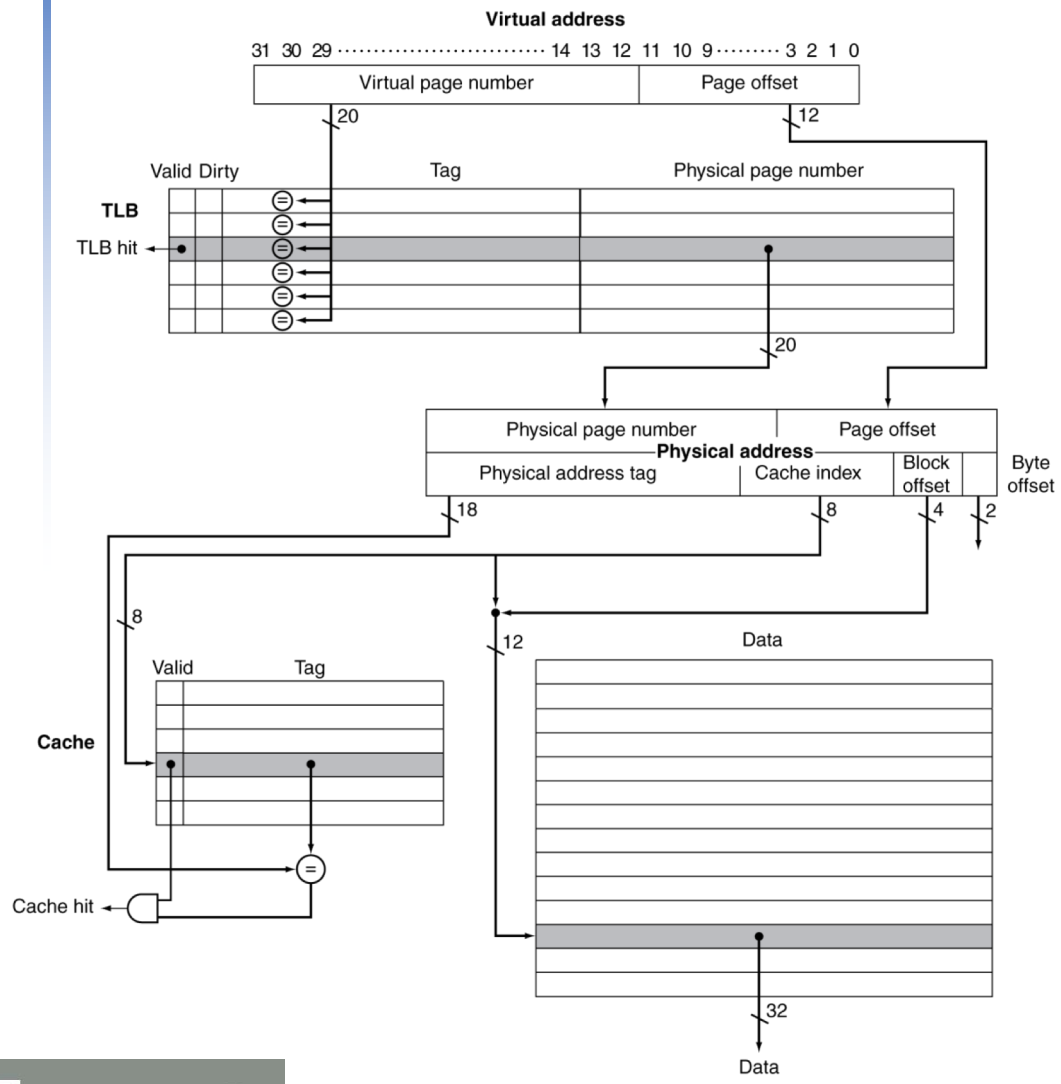
- System characteristics
  - 8 KB pages
  - 32-bit logical address divided into 13 bit page offset, 19 bit page number
- Page number divided into:
  - 10 bit page number
  - 9 bit page offset
- Logical address looks like this:
  - p1 is an index into the 1st level page table
  - p2 is an index into the 2nd level page table pointed to by p1



Fast Translation Using a TLB



TLB and Cache Interaction



- If cache tag uses physical address
  - Need to translate before cache lookup
- Alternative: use virtual address tag
  - Complications due to aliasing
    - Different virtual addresses for shared physical address

TLB Misses

- If page is in memory
  - Load the PTE from memory and retry
  - Could be handled in hardware
    - Can get complex for more complicated page table structures
  - Or in software
    - Raise a special exception, with optimized handler
- If page is not in memory (page fault)
  - OS handles fetching the page and updating the page table
  - Then restart the faulting instruction