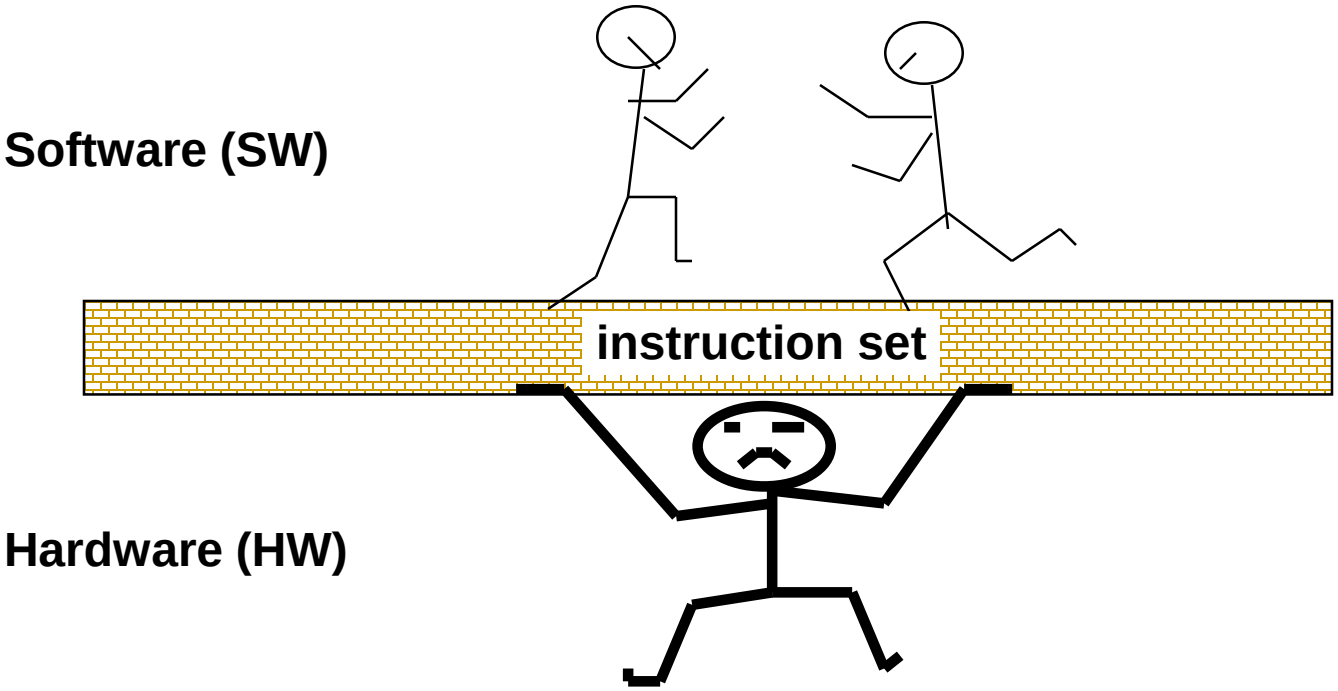
Instruction Set Principles and Examples

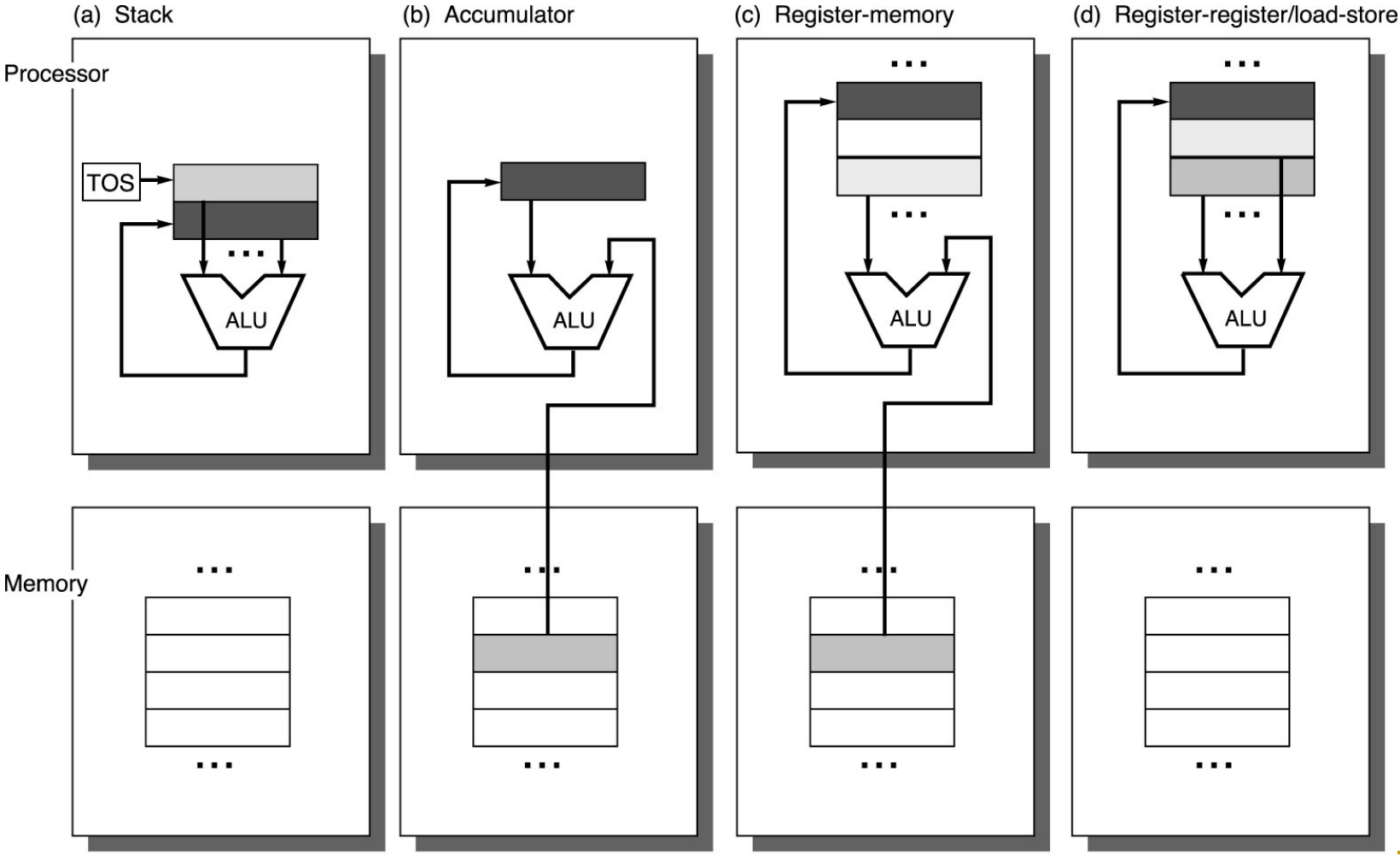
The Instruction Set: a Critical Interface



- Machine Language: Instruction
- Vocabulary: Instruction Set

- Properties of a good abstraction
  - Lasts through many generations (portability)
  - Used in many different ways (generality)
  - Provides convenient functionality to higher levels
  - Permits an efficient implementation at lower levels

Classifying Instruction Set Architectures



Classifying Instruction Set Architectures

C=A+B

Stack	Accumulator	Register (Register-Memory)	Register (Load-Store)
Push A Push B Add Pop C	Load A Add B Store C	Load R1,A Add R1,B Store C,R1	Load R1,A Load R2,B Add R3,R1,R2 Store C,R3

- Explicit operand
- Implicit operand

- *Stack architecture*
  - -operands are implicitly on the top of the stack
- Accumulator
  - - one operand is implicitly the accumulator
- *General Purpose Register (GPR) architecture*
  - - only explicit operands

Advantages of GPR Architecture

- Reasons using registers:
  - Faster than memory
  - More efficient for a compiler to use than other forms of internal storage.

- (A*B)–(B*C)–(A*D)
  - may be evaluated by doing the multiplications in any order, which may be more efficient (because of the location of the operands or because of pipelining concerns)

  - on a stack computer the hardware must evaluate the expression in only one order, since operands are hidden on the stack, and it may have to load an operand multiple times.

Advantages of GPR Architecture

- Registers can be used to hold variables.
- When variables are allocated to registers, the memory traffic reduces, the program speeds up
- Code density improves (since a register can be named with fewer bits than can a memory location)

How many registers are sufficient?

- Depends on the effectiveness of the compiler.
- Most compilers reserve registers for
  - Expression evaluation
  - parameter passing
  - hold variables
- Modern compiler technology and its ability to effectively use larger number of registers has led to an increase in register counts in more recent architectures.

ALU instructions

# Memory Address	Max # Operands	Examples
0	3	SPARC, MIPS, PowerPC, ALPHA
1	2	Intel 80x86, Motorola 68000
2	2	VAX (also have three-operand formats)
3	3	VAX (also have two-operand formats)

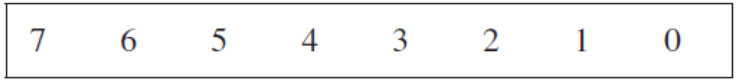
Comparisons

Type	Advantages	Disadvantages
Reg-reg (0,3)	▪ Simple, fixed-length ▪ Simple code-generation ▪ Take similar CPI	▪ Higher IC ▪ Lower instruction density leads to larger program
Reg-mem (1,2)	▪ Data can be accessed without loading instruction ▪ Format easy to encode	▪ Operands are not equivalent ▪ CPI varies
Mem-mem (3,3)	▪ Most compact ▪ Doesn't waste reg for temp	▪ Large variation in instruction size ▪ Large variation in CPI ▪ Memory bottleneck (not used today)

Memory Addressing

- **Interpreting Memory Addresses**
- Instruction sets discussed here are byte addressed and provide access for bytes (8 bits), half words (16 bits), and words (32 bits), double words (64 bits)

- Two different conventions for ordering the bytes within a larger object.
  - *Little Endian*



puts the byte whose address is “x . . . x000” at the least-significant position in the double word

  - *Big Endian*

puts the byte whose address is “x . . . x000” at the most significant position in the double word (the big end)

Aligned and misaligned addresses

- In some computers, accesses to objects larger than a byte must be *aligned*.
- An access to an object of size *s* bytes at byte address *A* is aligned if *A* mod *s* = 0.

Width of object	0	1	2	3	4	5	6	7
1 byte (byte)	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned
2 bytes (half word)	Aligned		Aligned		Aligned		Aligned	
2 bytes (half word)		Misaligned		Misaligned		Misaligned		Misaligned
4 bytes (word)	Aligned				Aligned			
4 bytes (word)		Misaligned				Misaligned		
4 bytes (word)			Misaligned				Misaligned	
4 bytes (word)				Misaligned				Misaligned
8 bytes (double word)	Aligned							
8 bytes (double word)		Misaligned						
8 bytes (double word)			Misaligned					
8 bytes (double word)				Misaligned				
8 bytes (double word)					Misaligned			
8 bytes (double word)						Misaligned		
8 bytes (double word)							Misaligned	
8 bytes (double word)								Misaligned

Addressing Mode

- Register
- Immediate (for constants)
- Displacement
- Register Indirect
- Indexed
- Direct/Absolute
- Memory Indirect

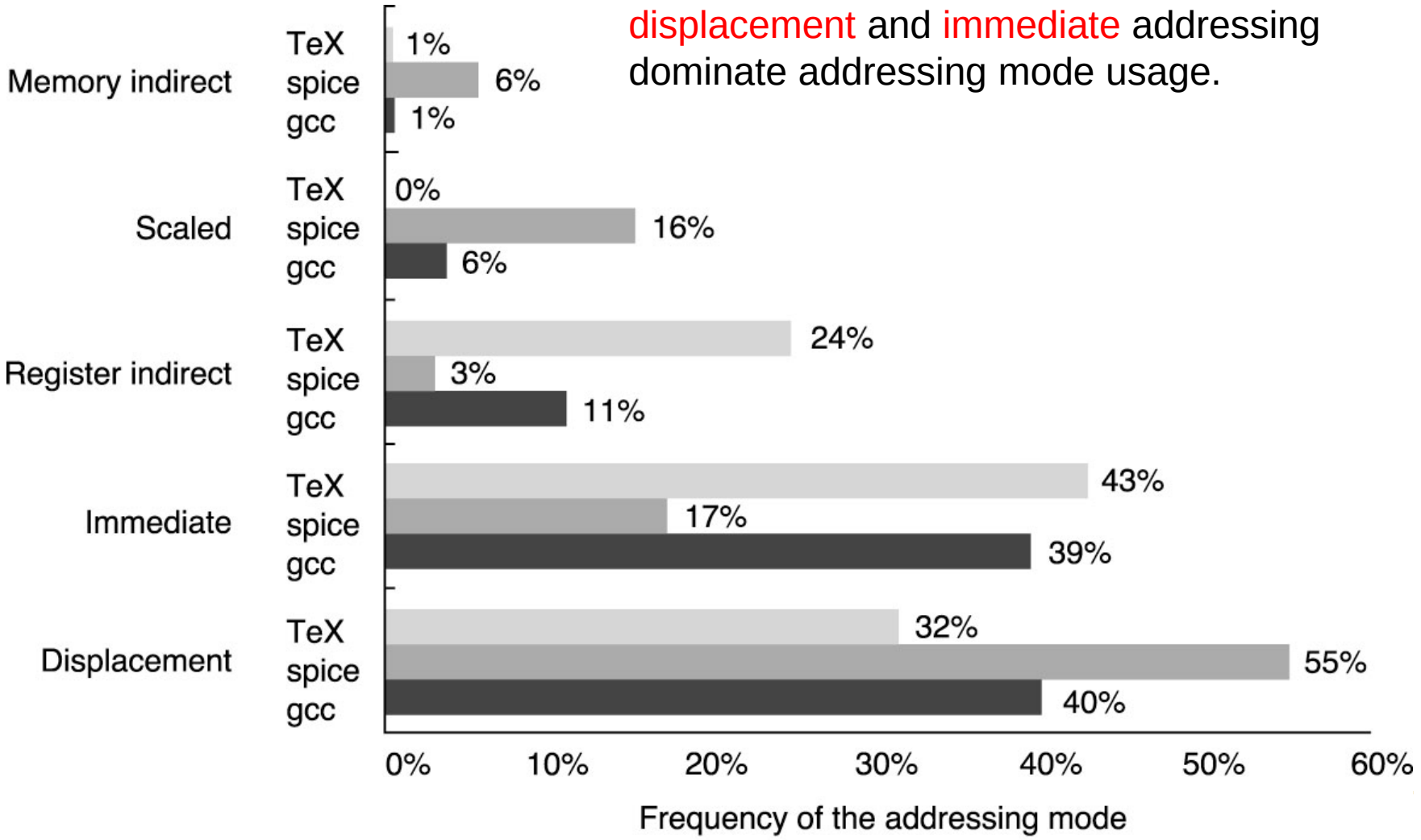
Addressing Modes

Addressing mode	Example instruction	Meaning	When used
Register	`Add R4,R3`	`Regs[R4] ← Regs[R4] + Regs[R3]`	When a value is in a register.
Immediate	`Add R4,#3`	`Regs[R4] ← Regs[R4] + 3`	For constants.
Displacement	`Add R4,100(R1)`	`Regs[R4] ← Regs[R4] + Mem[100+Regs[R1]]`	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	`Add R4,(R1)`	`Regs[R4] ← Regs[R4] + Mem[Regs[R1]]`	Accessing using a pointer or a computed address.
Indexed	`Add R3,(R1+R2)`	`Regs[R3] ← Regs[R3] + Mem[Regs[R1]+Regs[R2]]`	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	`Add R1,(1001)`	`Regs[R1] ← Regs[R1] + Mem[1001]`	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	`Add R1,@(R3)`	`Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]`	If R3 is the address of a pointer *p*, then mode yields $*p$.

Addressing Modes

Autoincrement	`Add R1,(R2)+`	`Regs[R1] ← Regs[R1] + Mem[Regs[R2]]` `Regs[R2] ← Regs[R2] + d`	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, *d*.
Autodecrement	`Add R1,–(R2)`	`Regs[R2] ← Regs[R2] – d` `Regs[R1] ← Regs[R1] + Mem[Regs[R2]]`	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	`Add R1,100(R2)[R3]`	`Regs[R1] ← Regs[R1] + Mem[100+Regs[R2] + Regs[R3]*d]`	Used to index arrays. May be applied to any indexed addressing mode in some computers.

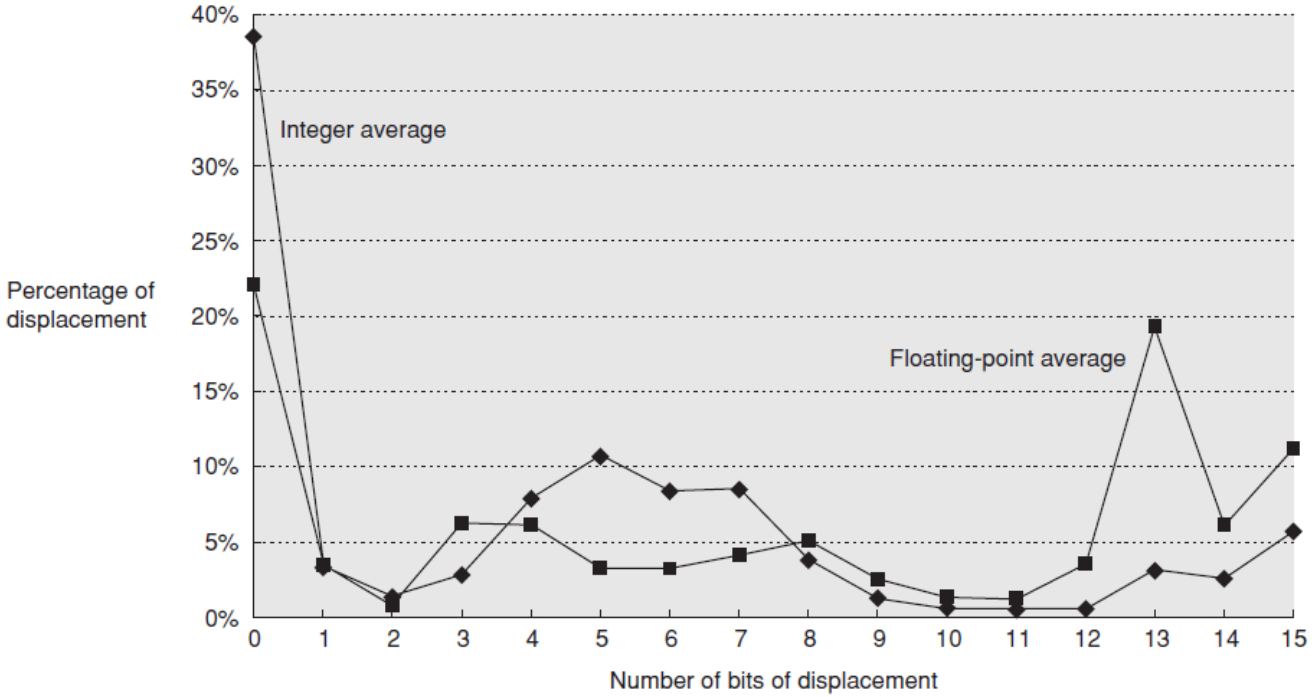
Frequency of the addressing mode



displacement and immediate addressing dominate addressing mode usage.

Displacement Addressing Mode

- The range of displacements : displacement field size directly affects the instruction length

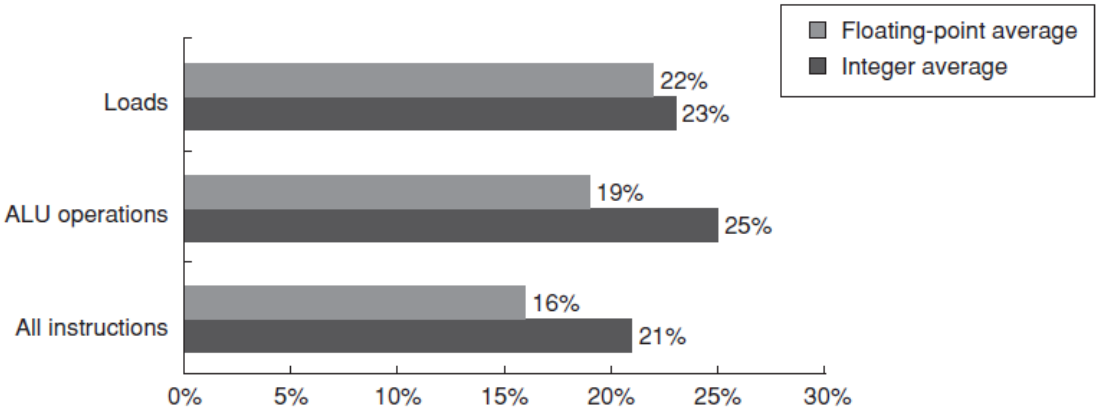


• distribution of displacement values due to:

- multiple storage areas for variables
- different displacements to access them
- overall addressing scheme the compiler uses

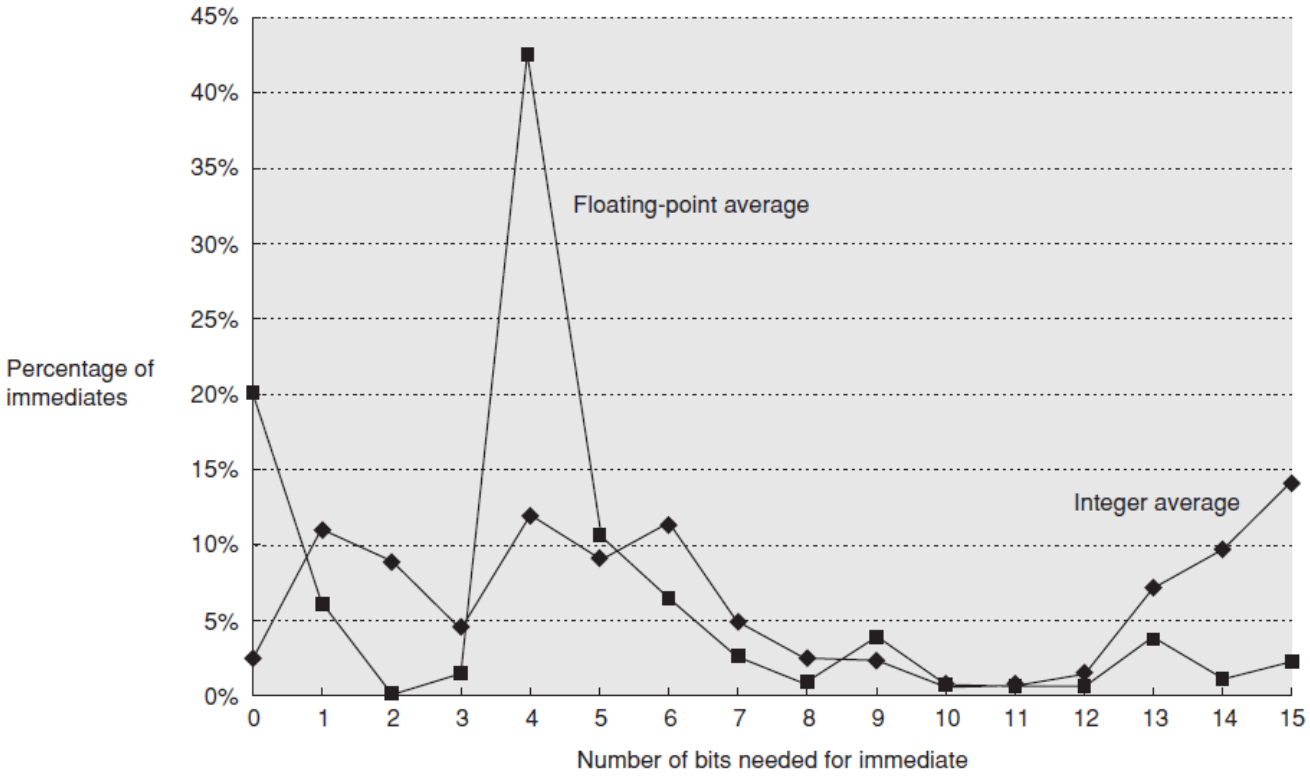
Immediate or Literal Addressing Mode

- Immediates can be used in
  - arithmetic operations
  - comparisons (primarily for branches)
  - constants written in the code (tend to be small)
  - address constants (tend to be large)



About one-quarter of data transfers and ALU operations have an immediate operand.

Size of immediate values



Operations in the Instruction Set

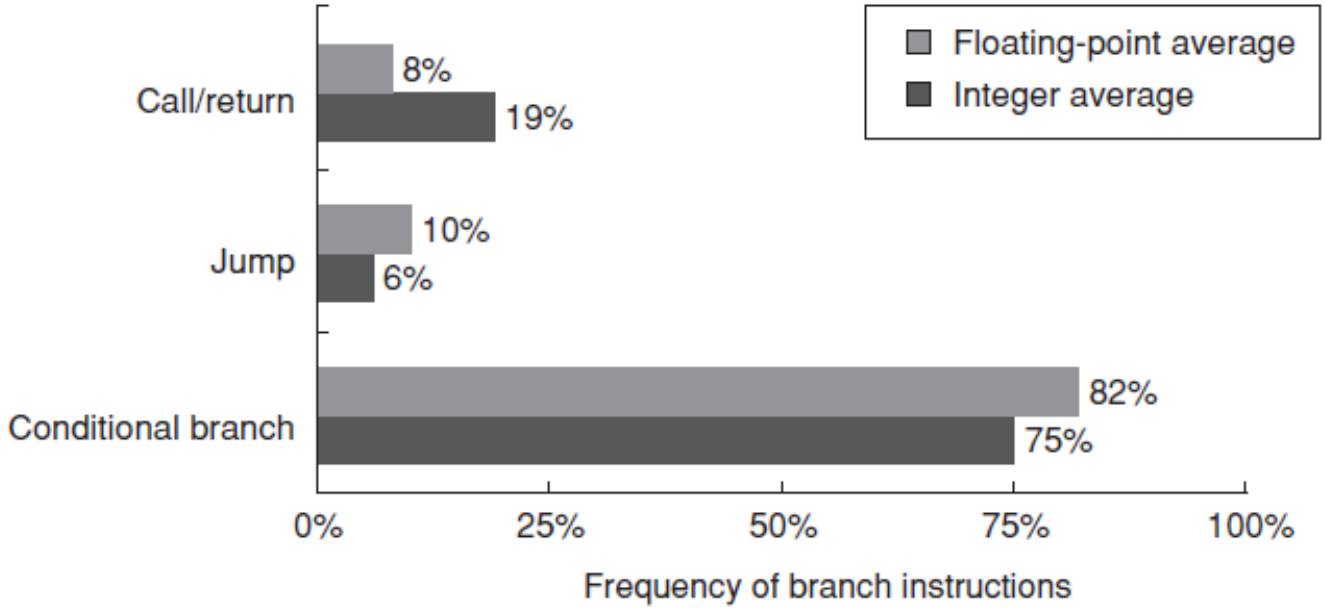
Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiply, divide
Data transfer	Loads-stores (move instructions on computers with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiply, divide, compare
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression/decompression operations

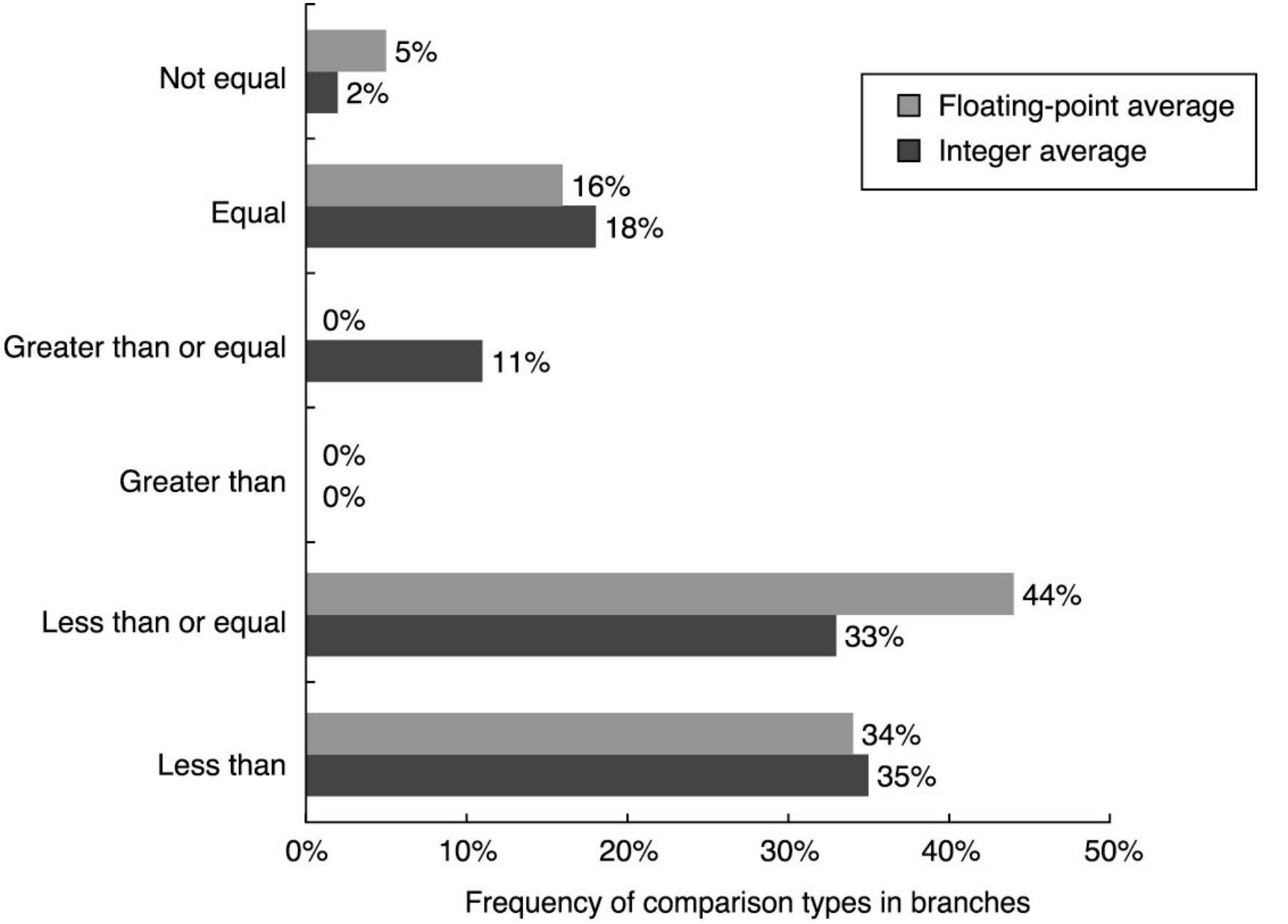
The top 10 instructions for the 80x86

Rank	80x86 instruction	Integer average (% total executed)
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
Total		96%

Instructions for Control Flow

- Conditional branches
- Jumps
- Procedure calls
- Procedure returns




Not equal
5%
2%
Equal
16%
18%
Greater than or equal
0%
11%
Greater than
0%
0%
Less than or equal
44%
33%
Less than
34%
35%
Floating-point average
Integer average
0%
10%
20%
30%
40%
50%
Frequency of comparison types in branches

Addressing Modes for Control Flow Instructions

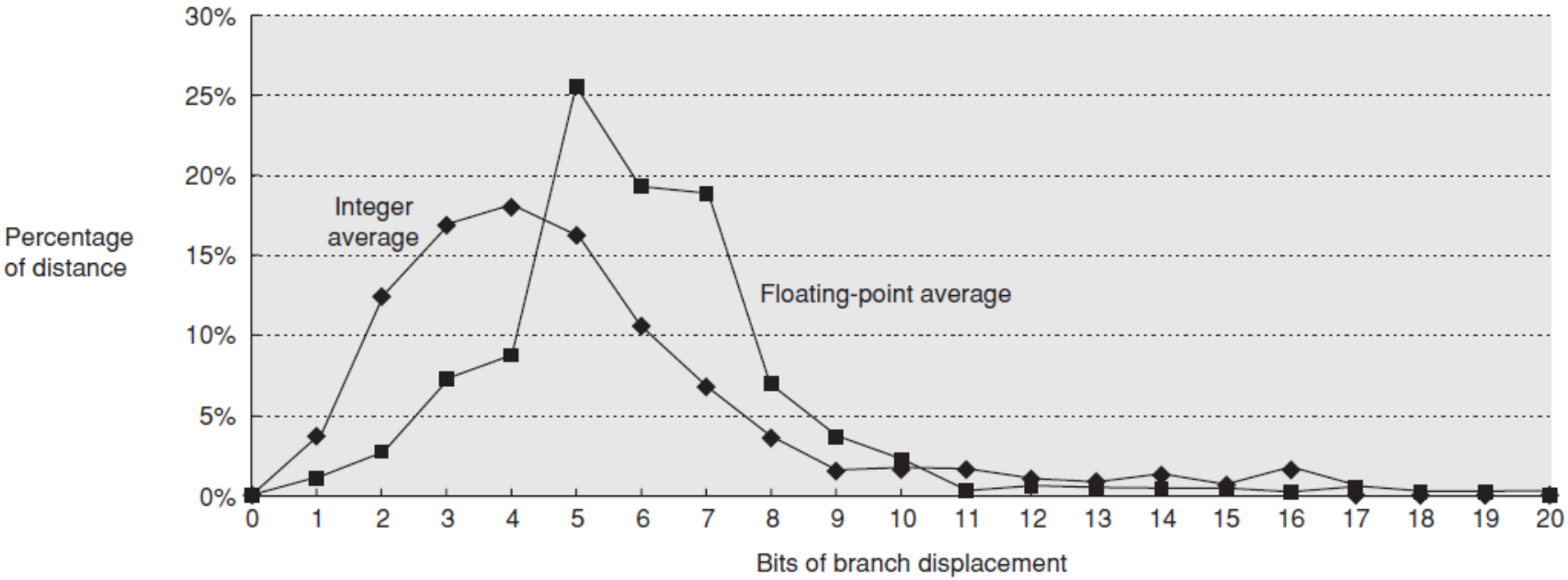
- PC-relative
  - A displacement is added to the program counter
  - advantageous because the target is often near the current instruction
  - Permits the code to run independently of where it is loaded (position independence)

Addressing Modes for Control Flow Instructions

- To implement returns and indirect jumps when the target is not known at compile time
- Specify the target dynamically, so that it can change at run time (Other than PC-relative)
  - target address is not known at compile time
  - usually loaded from memory into a register before the register indirect jump
  - Register indirect jumps are also useful for
    - *Virtual functions* or *methods*
    - *High-order functions* or *function pointers*
    - *Dynamically shared libraries*

Branch distances between the target and the branch instruction

- choosing what branch offsets to support affects the instruction length and encoding

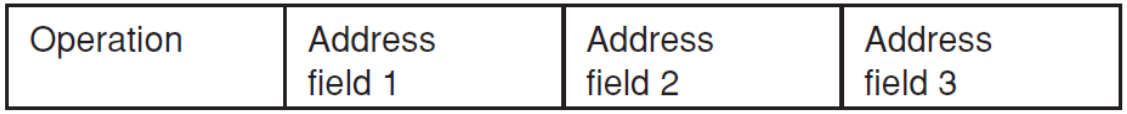


Conditional Branch Options

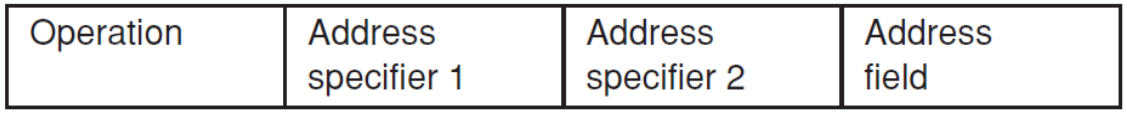
Name	Examples	How condition is tested	Advantages	Disadvantages
Condition code (CC)	80x86, ARM, PowerPC, SPARC, SuperH	Tests special bits set by ALU operations, possibly under program control.	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructions since they pass information from one instruction to a branch.
Condition register	Alpha, MIPS	Tests arbitrary register with the result of a comparison.	Simple.	Uses up a register.
Compare and branch	PA-RISC, VAX	Compare is part of the branch. Often compare is limited to subset.	One instruction rather than two for a branch.	May be too much work per instruction for pipelined execution.

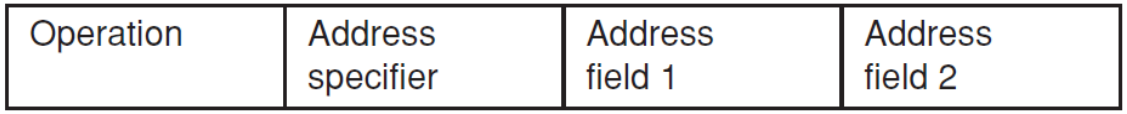
Encoding an Instruction Set

(a) Variable (e.g., Intel 80x86, VAX)



(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)





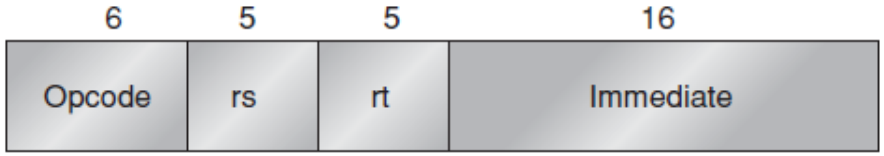
(c) Hybrid (e.g., IBM 360/370, MIPS16, Thumb, TI TMS320C54x)

MIPS instruction types

- **Arithmetic logical**
  - Add, AddU, Sub, SubU, And, Or, Xor, Nor
- **Memory Access**
  - Load, Store
- **Control**
  - Conditional Branch
  - Unconditional Jump

Example: MIPS instruction

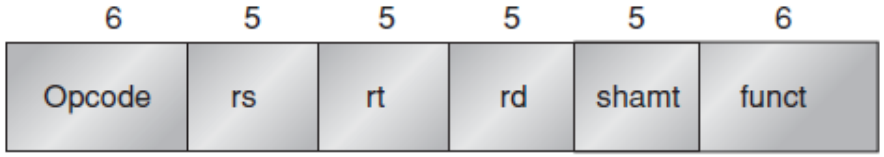
I-type instruction



Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

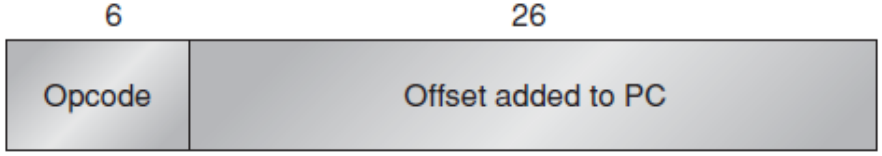
Conditional branch instructions (rs is register, rd unused)

R-type instruction



Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, . . .
Read/write special registers and moves

J-type instruction

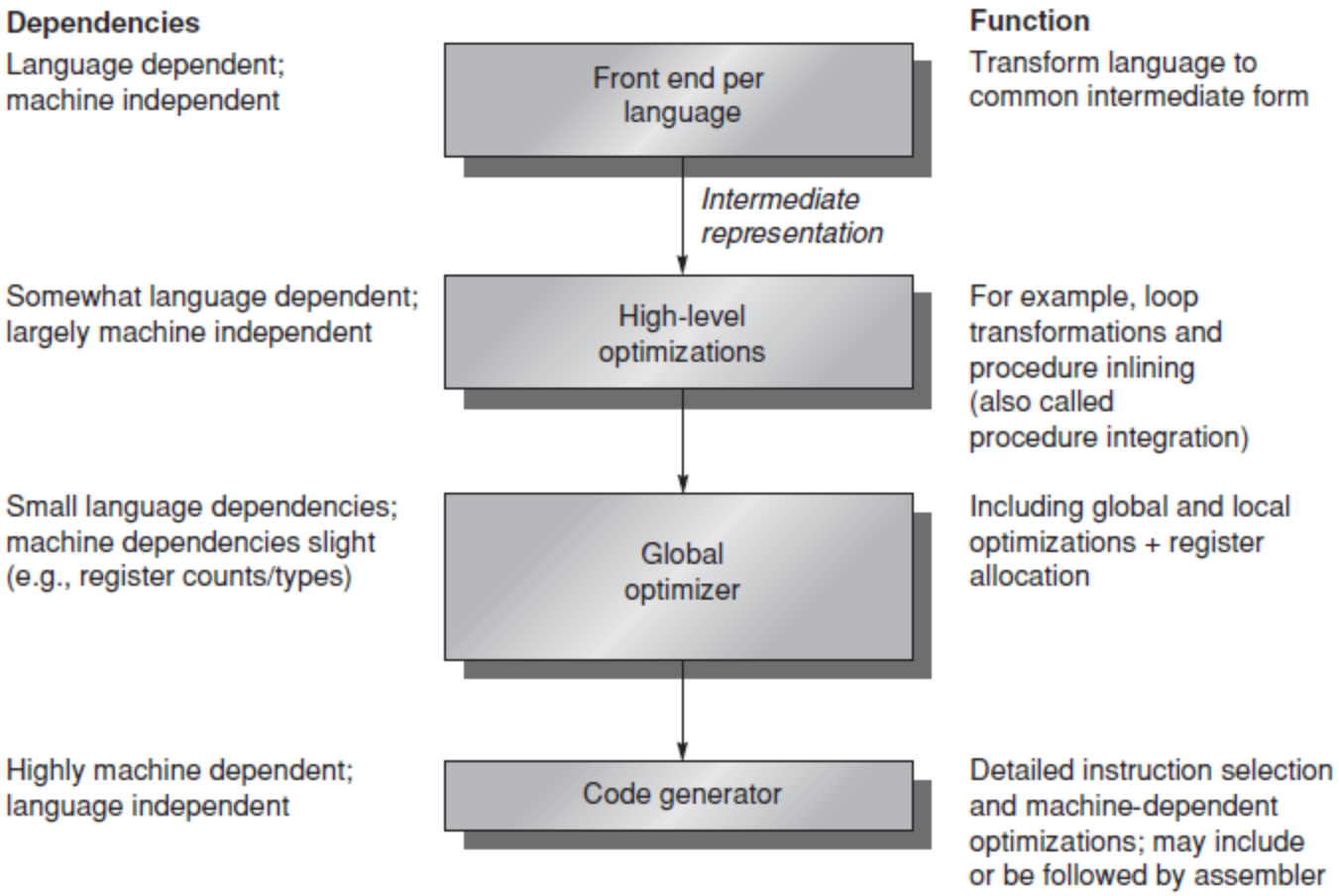


Jump and jump and link
Trap and return from exception

VAX and MIPS

- VAX (CISC)
  - Simple compiler and code density
  - Powerful addressing mode and instructions
  - Efficient instruction coding
  - Few registers
- MIPS (SISC)
  - High performance via pipelining
  - Ease of hardware implementation
  - Compatibility with highly optimizing compilers
  - Simple addressing mode and instructions
  - Fixed-length instruction format
  - Large number of registers

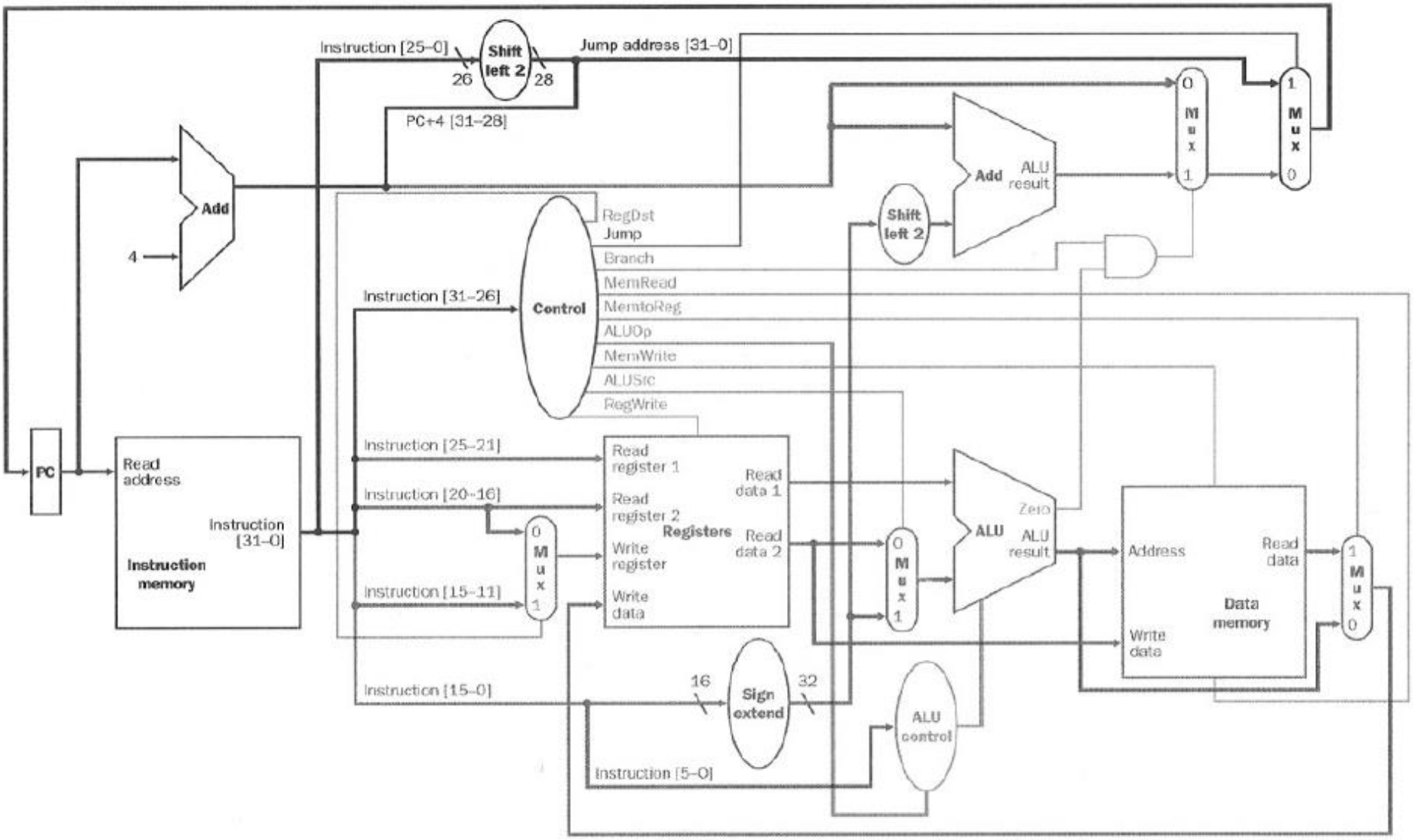
Compiler Perspective



Fallacies

- There is such a thing as a typical program.
  - Programs can vary significantly.
- An architecture with flaws cannot be successful
- You can design a flawless architecture.
  - All architecture design involves trade-offs.

Single Cycle Control and Data path



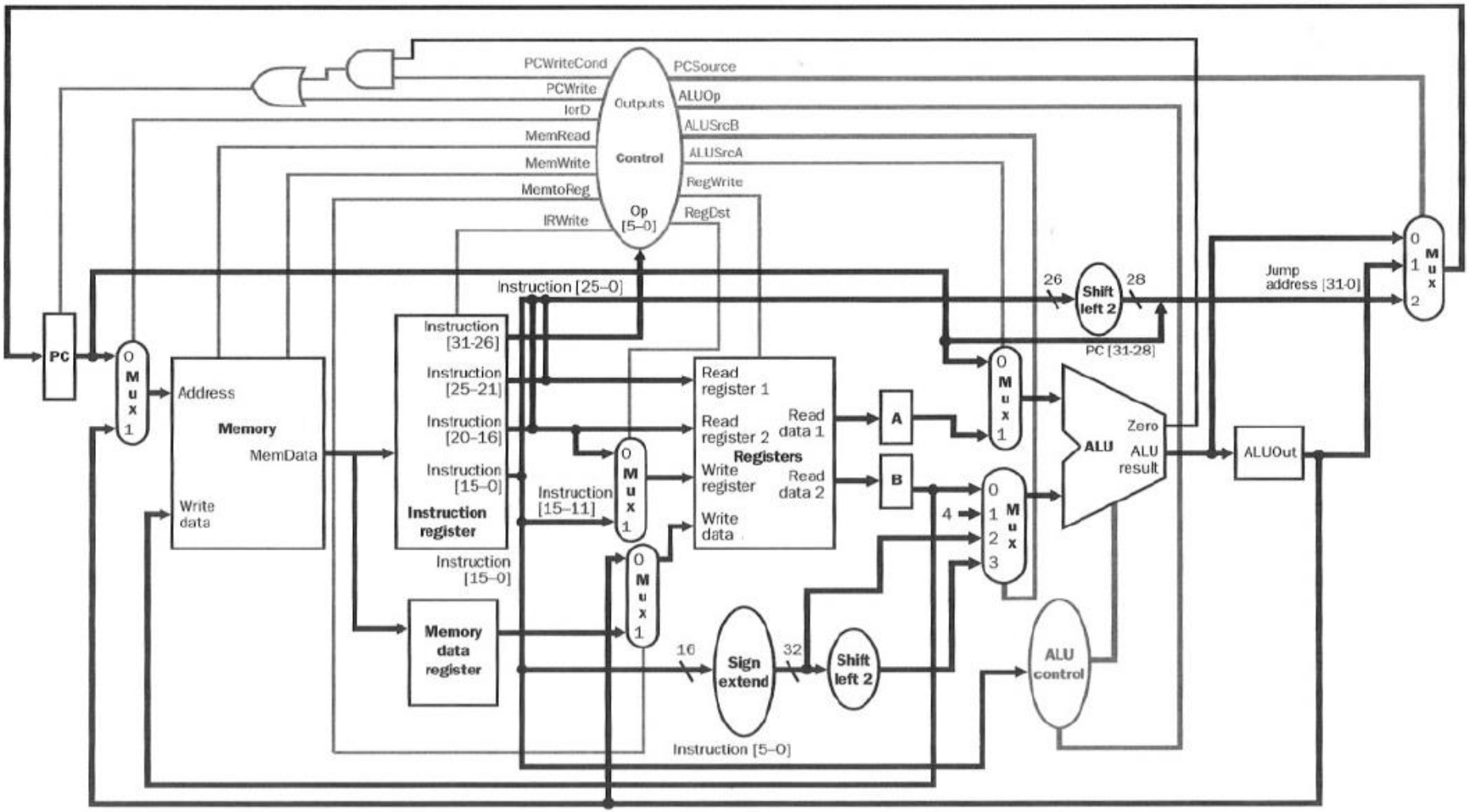
Why a Single-Cycle Implementation is not used?

- Inefficient
- The clock cycle is determined by the longest possible path
  - Load instruction: use five functional units in series: Instruction memory, register file, ALU, data memory, register file
- Other instructions could fit in a shorter clock cycle
- Therefore, although CPI=1, the overall performance is not good

Multi-cycle Implementation

- Each step takes 1 clock cycle
- Allows a functional unit to be used more than once per instruction, as long as it is used on different cycles
  - Help reduce the amount of hardware required
- Main advantages
  - Allow instructions to take different # of clock cycles
  - Ability to share functional units within the execution of a single instruction

Multi-cycle Implementation



Multi-cycle Implementation

- Main difference compared to single cycle implementation
  - Registers are added after every major functional unit to hold the output of that unit until the value is used in a subsequent clock cycle
    - Instruction register
    - Memory data register
    - A and B registers
    - ALUOut register

Every instruction can be implemented in at most 5 clock cycles

1. *Instruction fetch cycle* (IF):

   ```
   IR ← Mem[PC];
   NPC ← PC + 4;
   ```

2. *Instruction decode/register fetch cycle* (ID):

   ```
   A ← Regs[rs];
   B ← Regs[rt];
   Imm ← sign-extended immediate field of IR;
   ```

3. *Execution/effective address cycle* (EX):
   - Memory reference:

     ```
     ALUOutput ← A + Imm;
     ```
   - Register-Register ALU instruction:

     ```
     ALUOutput ← A func B;
     ```
   - Register-Immediate ALU instruction:

     ```
     ALUOutput ← A op Imm;
     ```
   - Branch:

     ```
     ALUOutput ← NPC + (Imm << 2);
     Cond ← (A == 0)
     ```

4. *Memory access/branch completion cycle* (MEM):

   The PC is updated for all instructions: `PC ← NPC;`
   - Memory reference:

     ```
     LMD ← Mem[ALUOutput] or
     Mem[ALUOutput] ← B;
     ```
   - Branch:

     ```
     if (cond) PC ← ALUOutput
     ```

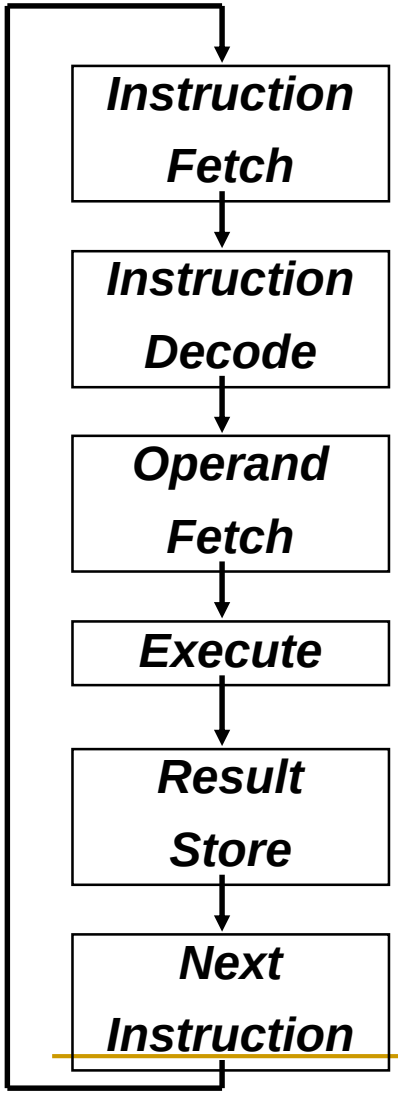
5. *Write-back cycle* (WB):
   - Register-Register ALU instruction:

     ```
     Regs[rd] ← ALUOutput;
     ```
   - Register-Immediate ALU instruction:

     ```
     Regs[rt] ← ALUOutput;
     ```
   - Load instruction:

     ```
     Regs[rt] ← LMD;
     ```

Fundamental Execution Cycle



Obtain instruction from program storage

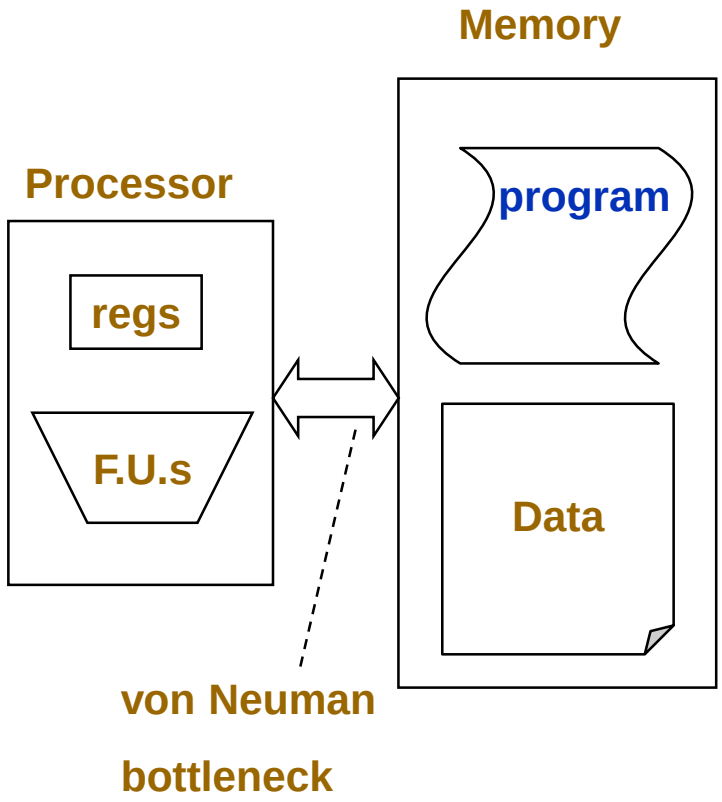
Determine required actions and instruction size

Locate and obtain operand data

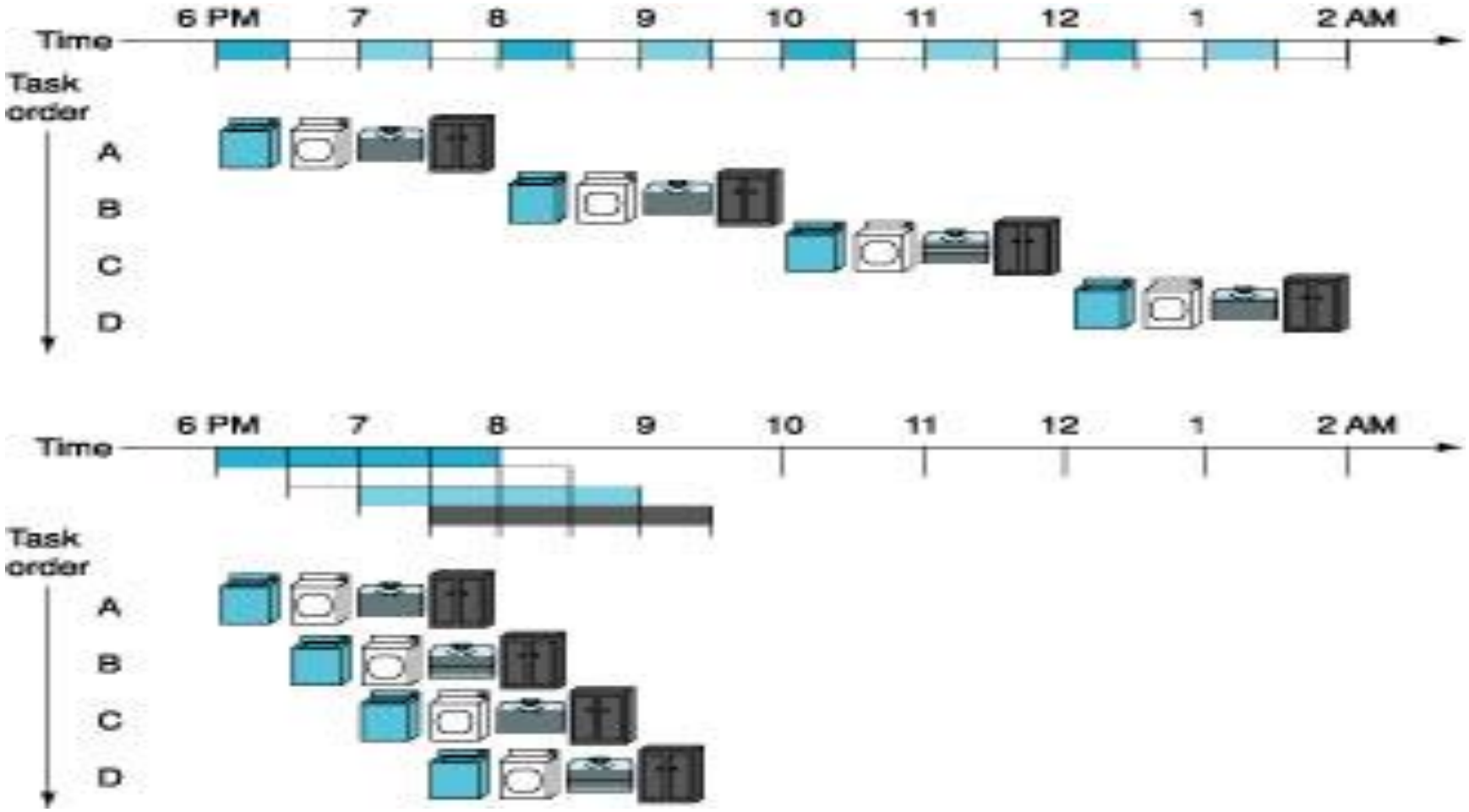
Compute result value or status

Deposit results in storage for later use

Determine successor instruction



Nature of Pipelining



5 Steps of DLX Datapath