Performance Key Points

- We need to be precise about how to specify performance
- Performance is only meaningful in the context of a workload, and its data set and the compiler
- Be careful how you summarize performance, use geo mean for summarizing ratios
- Amdahl's law is about the part that gets no improvement
- Watch out for diminishing returns
- ET = IC * CPI * CT

Principles of Computer Architecture

CSE 240A

Fall 2024

Hadi Esmaeilzadeh

hadi@ucsd.edu

University of California, San Diego

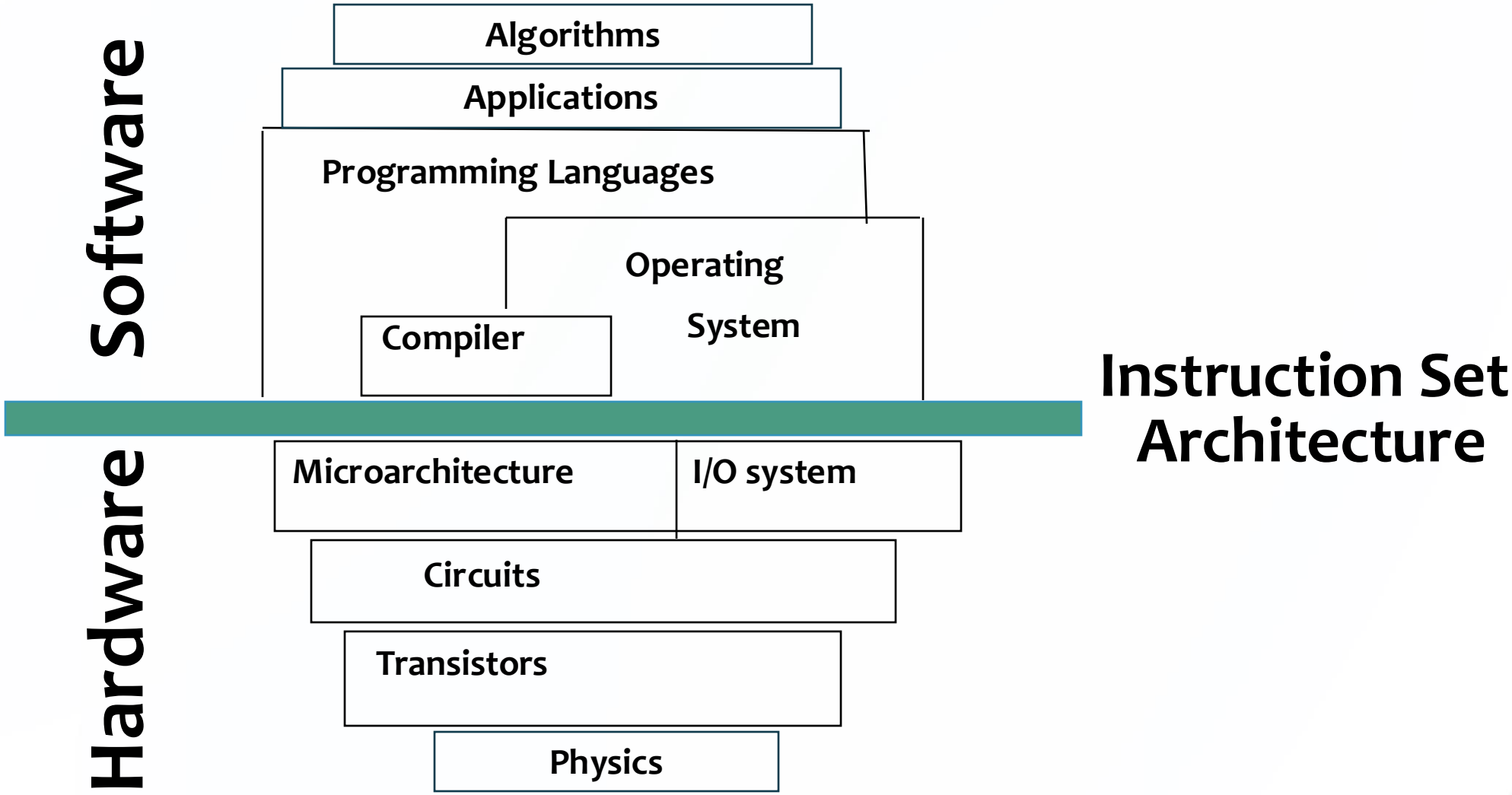
Instruction Set Architecture

or

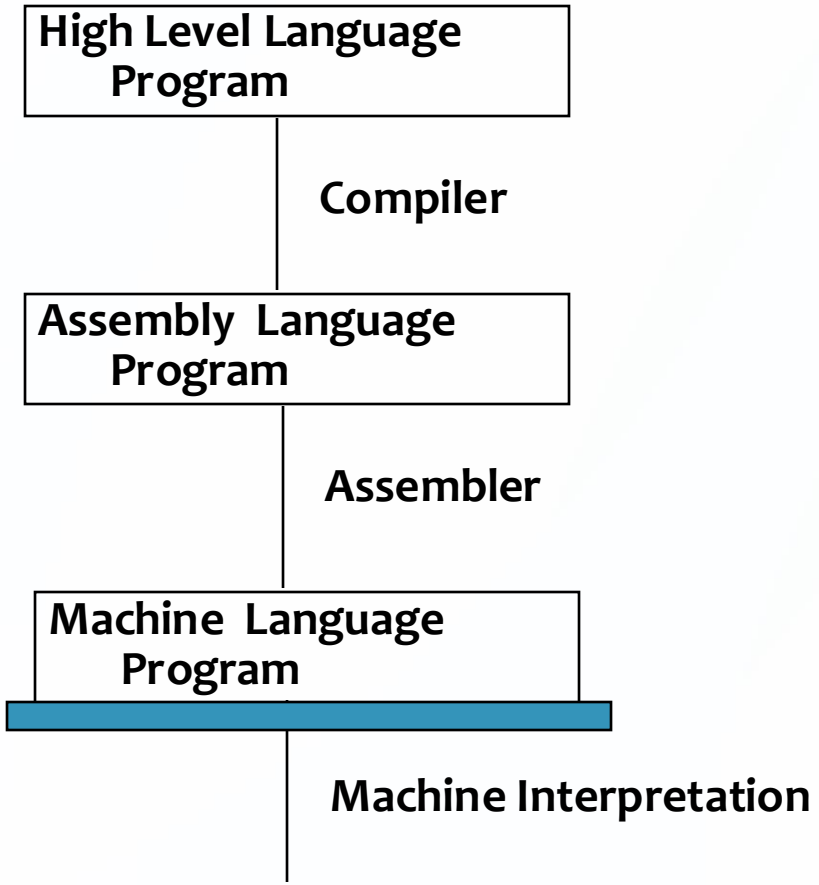
"How to talk to computers"

"What is possible in the machine without specifying how"

Computational Stack



How to Speak Computer




```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw  $15,    0($2)
lw  $16,    4($2)
sw $16,     0($2)
sw $15,     4($2)
```

```
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
```

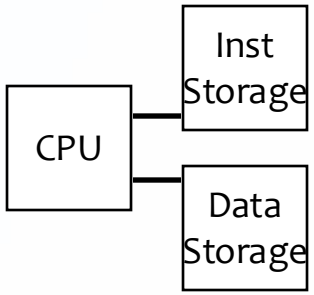
ALUOP[0:3] <= InstReg[9:11] & MASK

Crafting an ISA

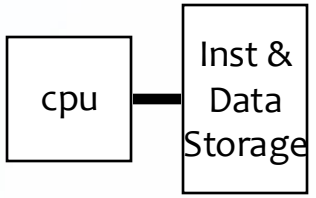
- Designing an ISA is both an art and a science
- ISA design involves dealing in an extremely rare resource – instruction bits!
- Some things we want out of our ISA
  - completeness
  - regularity and simplicity
  - compactness
  - ease of programming
  - ease of implementation

Where are the instructions?

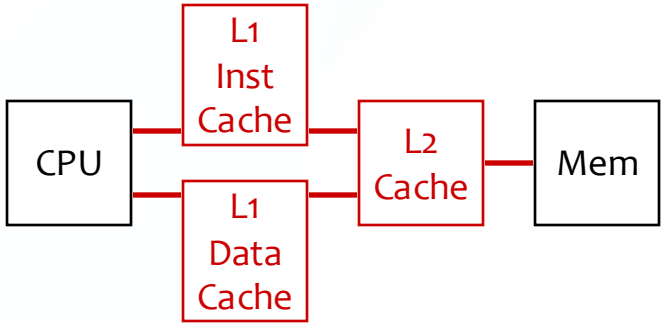
- Harvard architecture



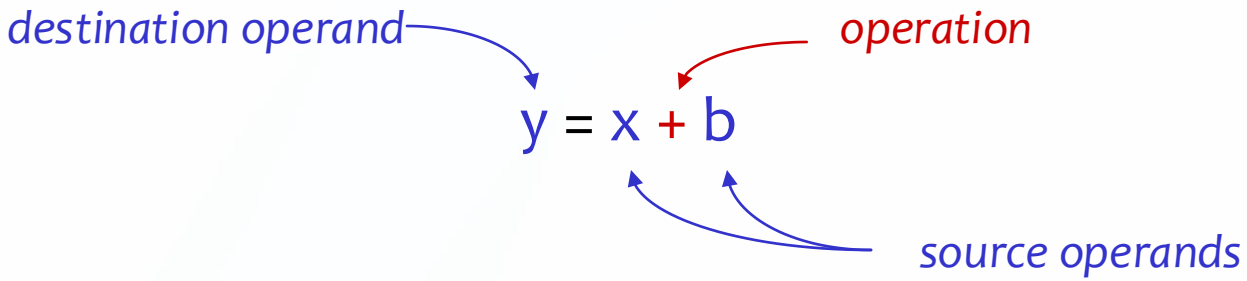
- Von Neumann architecture



"stored-program" computer



Key ISA decisions



- operations
  - how many?
  - which ones
- operands
  - how many?
  - location
  - types
  - how to specify?
- instruction format
  - size
  - how many formats?

how does the computer know what
0001 0100 1101 1111
means?

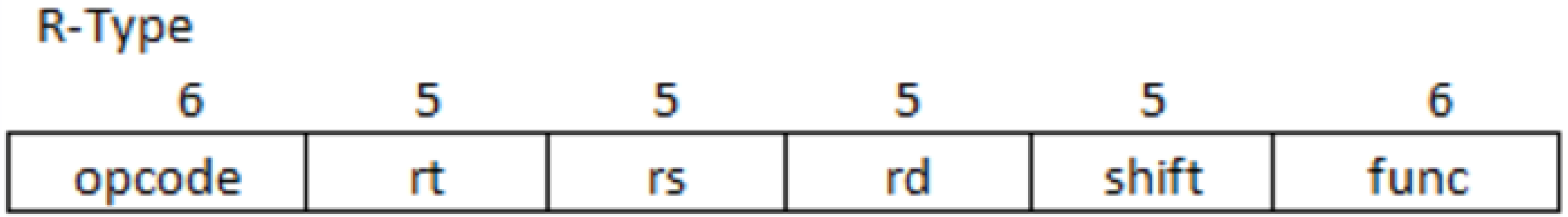
Your architecture supports 16 instructions and 32 registers (0-31). You have a fixed width instructions which are 16 bits. How many register operands can you specify (explicitly) in an add instruction?

Selection	operands
A	<= 1
B	<= 2
C	<= 3
D	<= 4
E	None of the above

R-type

rd = rs op rt

OpCode: 6 bits
Register Source: RS = 5
Register Temp: RT = 5
Register Destination = 5
Shift Amount = 5
Funct: 6

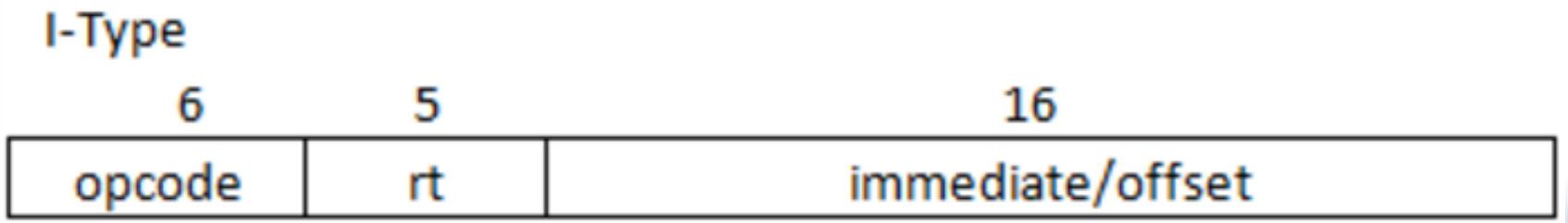


I-type

rt = rs + sign_ext (imm)
lw => rt = M[rs + sign_ext(imm)]
sw > M[rs + sign_ext(imm)] = rt

OpCode: 6 bits
Register Source: RS = 5
Register Temp: RT = 5 => RD
Imm/disp/offset = 16

Base Displacement
Register Direct: Imm = 0 => M[rs]
Memory Direct: rs = 0 =>
M[sign_ext(imm)]



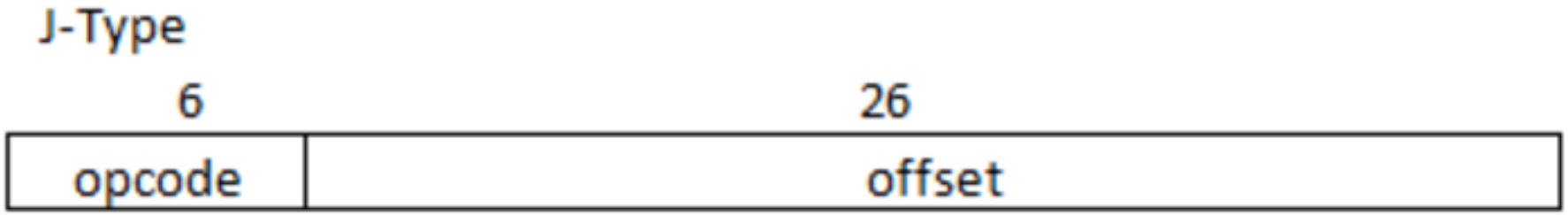
Branch
If (rs op rt) =>
PC = PC + sign_ext(offset << 2)

J-type

Jump imm => PC = imm << 2
Jump register => PC = $rs

OpCode: 6 bits
24 bits Jump Address

Jump and Link:
jal imm;
PC = imm << 2;
$ra=$31=PC+4



What enables performance in today's machines?

- This wasn't true in the era in which most classical ISAs were defined…

- Parallelism!!
  - Superscalar
  - Pipelining
  - Multicore

Choice 1: Operand Location

- Accumulator
- Stack
- Registers
- Memory

- We can classify most machines into 4 types: *accumulator*, *stack*, *register-memory* (most operands can be registers or memory), *load-store* (arithmetic operations must have register operands).

Choice 1B: How Many Operands?

Basic ISA Classes

Accumulator:

1 address	add A	acc ← acc + mem[A]

Stack:

0 address	add	tos ← tos + next
1 address	push A	tos ← mem[A]

General Purpose Register:

2 address	add A B	EA(A) ← EA(A) + EA(B)
3 address	add A B C	EA(A) ← EA(B) + EA(C)

Load/Store:

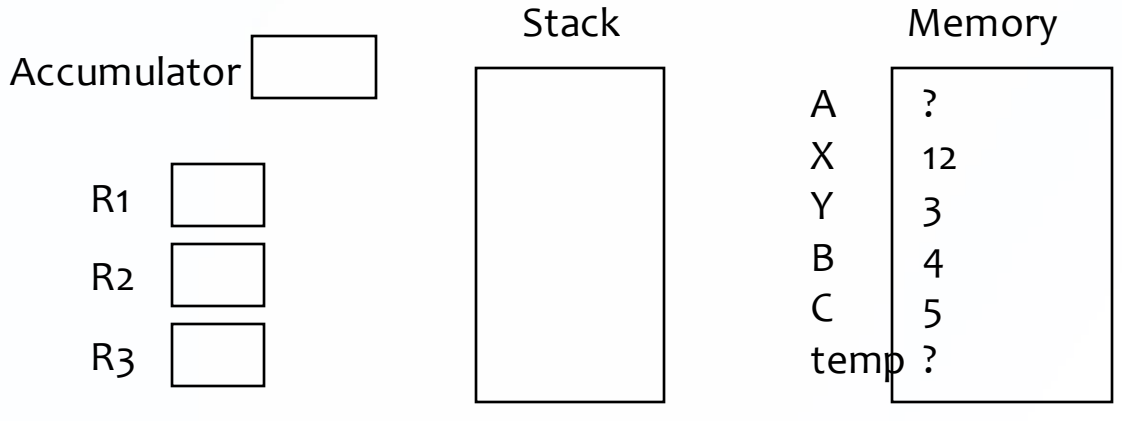
3 address	add Ra Rb Rc	Ra ← Rb + Rc
	load Ra Rb	Ra ← mem[Rb]
	store Ra Rb	mem[Rb] ← Ra

A *load/store* architecture has instructions that do either ALU operations or access memory, but never both.

Alternative ISA's

$$A = X*Y - B*C$$

Stack Architecture	Accumulator	GPR	GPR (Load-store)
push B	load B	R1 = mul B, C	R1 = load X
push C	mul C	R2 = mul X,Y	R2 = load Y
mul	store temp	A = sub R1, R2	R3 = load B
push X	load X		R4 = load C
push Y	mul Y		R5 = mul R1, R2
mul	sub temp		R6 = mul R3, R4
sub	store A		R1 = sub R5, R6
pop A			Store A = R1



A = BC+XY

ISOMORPHIC

Stack	Acc	Reg-Mem	Reg-Reg
push B	Load B	R1 = B*C	R1 = B
push C	Mult C	R2 = X*Y	R2 = C
mult	Store temp	A = R1+R2	R3 = X
push X	Load X		R4 = Y
push Y	Mult Y		R5 = R1*R2
Mult	Add temp		R6 = R3*R4
Add	Store A		R7 = R5+R6
pop A			A = R7

In an alternative universe, **memory is VERY SLOW to access** relative to registers (internal storage). Which ISA would you most likely find in this universe?

A. Stack
B. Accumulator
C. Reg-Reg
D. Accumulator and Reg-mem
E. Stack and Accumulator

A = BC+XY

Stack	Acc	Reg-Mem	Reg-Reg
pushB	Load B	R1 = B*C	R1 = B
pushC	Mult C	R2 = X*Y	R2 = C
Mult	Store temp	A = R1+R2	R3 = X
push X	Load X		R4 = Y
push Y	Mult Y		R5 = R1*R2
Mult	Add temp		R6 = R3*R4
add	Store A		R7 = R5+R6
pop A			A = R7

In an alternative universe – registers (internal storage) are **very expensive and memory is fast**. Which ISA would you most likely find in this universe?

A.Stack
B.Accumulator
C.Reg-Reg
D.Reg-Mem and Stack
E.Both Reg-Reg and Reg-Mem

Tradeoffs

- Stack
  + Code Size
- Accumulator
  + If registers were expensive / Fixed Length
- GPR
  + Parallelism
  + Low Instruction Count
- Load-Store
  + Parallelism
  + Low Instruction Count

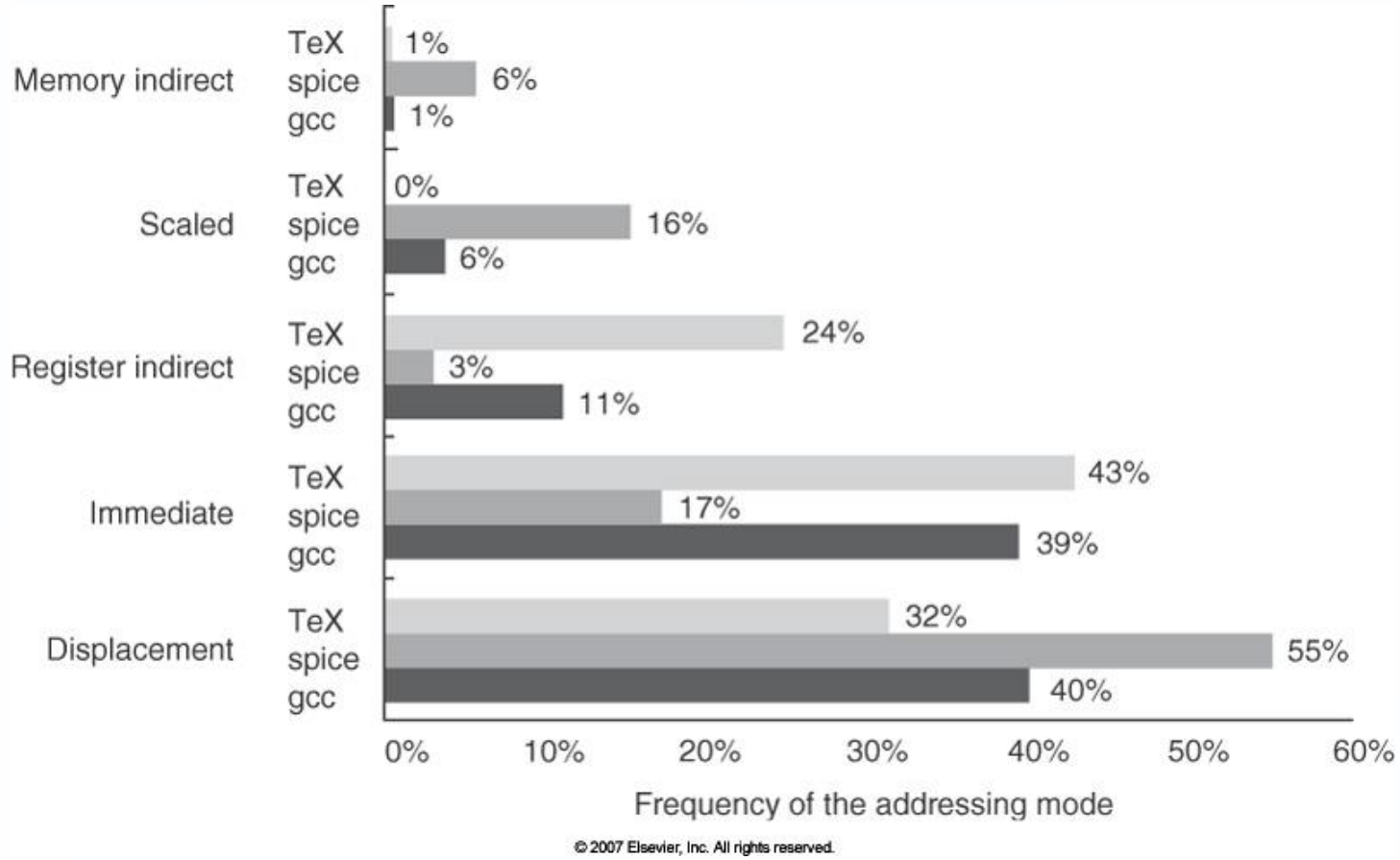
- Stack
  - No parallelism, dependence to top of the stack
  - Mostly two accesses to the memory
- Accumulator
  - No parallelism, dependence to the accumulator
- GPR
  - Variable-length, complicates decode logic
- Load-Store
  - High instruction count (big code)

Choice 2: Addressing Modes

how do we specify the operand we want?

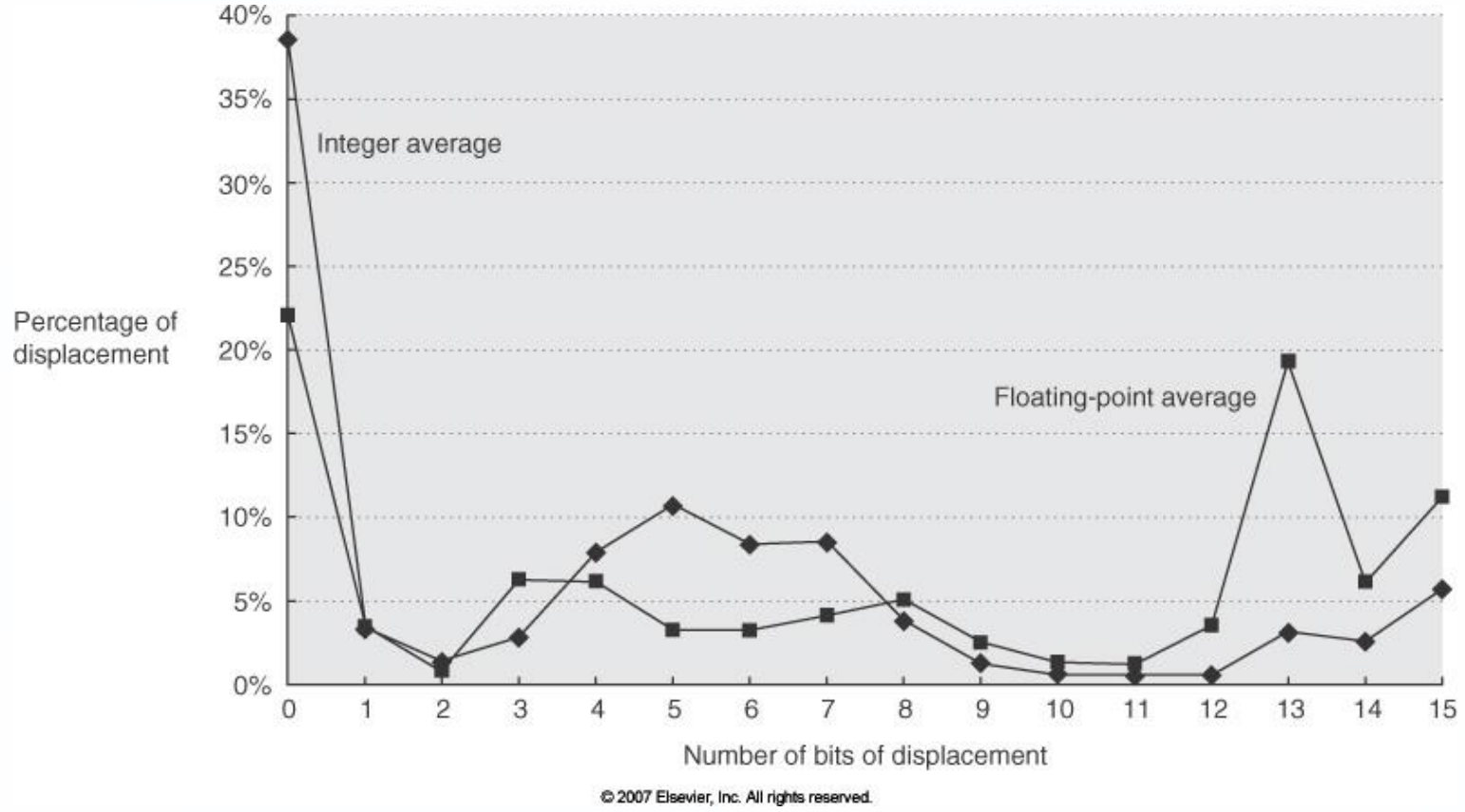
• Register direct	R3	R6 = R5 + *R3*
• Immediate (literal)	#25	R6 = R5 + *25*
• Direct (absolute)	M[10000]	R6 = *M[10000]*
• Register indirect	M[R3]	R6 = *M[R3]*
(a.k.a register deferred)		
• Memory Indirect	M[M[R3]]	
• Displacement	M[R3 + 10000]	…
• Index	M[R3 + R4]	
• Scaled	M[R3 + R4*d + 10000]	
• Autoincrement	M[R3++]	
• Autodecrement	M[R3 - -]	

Addressing Mode Utilization



Conclusion?

Displacement Size



Conclusions – 16 bits is usually enough.
If not, just use another instruction.

Choice 3: Which Operations?

- arithmetic
  - add, subtract, multiply, divide
- logical
  - and, or, shift left, shift right
- data transfer
  - load word, store word
- control flow

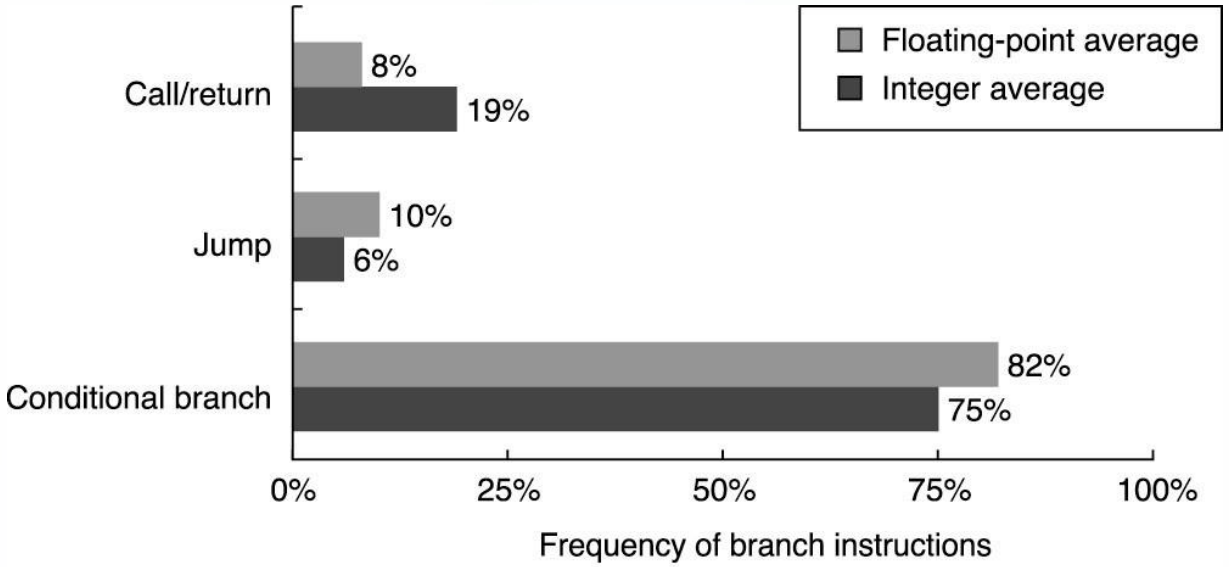
Does it make sense to have more complex instructions?
-e.g., square root, mult-add, matrix multiply, cross product …

Types of branches (control flow)

• conditional branch	beq r1,r2, label
• jump	jump label
• procedure call	call label
• procedure return	return

Types of branches (control flow)

• conditional branch	beq r1,r2, label
• jump	jump label
• procedure call	call label
• procedure return	return



Conditional branch

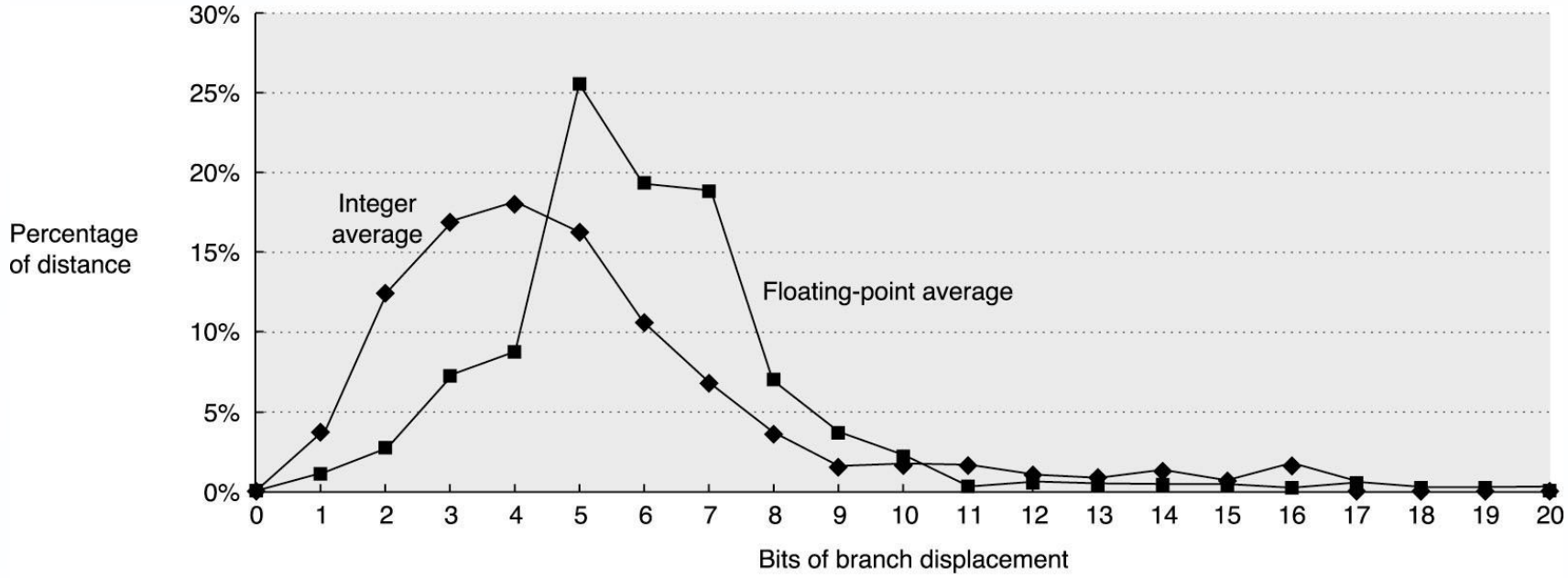
- How do you specify the destination (target) of a branch/jump?
- How do we specify the condition of the branch?

Destination first – how to specify a 32-bit quantity in a 32-bit instruction?

What form of addressing is used by MIPS branch instructions?

	Addressing Mode	Best explanation
A	Absolute	Branch instructions require a full 32-bit address to know the branch target
B	Absolute	A 32-bit immediate gives us enough bits to specify a full address
C	Relative	Branches tend to be backward branches which require a negative immediate
D	Relative	Branch targets tend to be close to the branch instruction
E	Register Indirect	We can load a 32 bit full address in a register – which lets us branch anywhere

Branch distance



- Average distance (in bits needed to specify) from branch to target.
- Conclusions?

Branch condition

Condition Codes

Processor status bits are set as a side-effect of arithmetic instructions or explicitly by compare or test instructions.

ex: sub r1, r2, r3

bz label

What's the real problem with condition codes? (see next question – inability to Do scheduling)

Condition Register

Ex: cmp r1, r2, r3

bgtz r1, label

Compare and Branch

Ex: beq/bne r1, r2, label

Branch condition

Condition Codes

Processor status bits are set as a side-effect of arithmetic instructions or explicitly by compare or test instructions.

ex: sub r1, r2, r3
bz label

Condition Register

Ex: cmp r1, r2, r3
bgtz r1, label

Compare and Branch

Ex: beq/bne r1, r2, label

Is one of these is problematic for modern processors/ compilers? If so, which one and why?

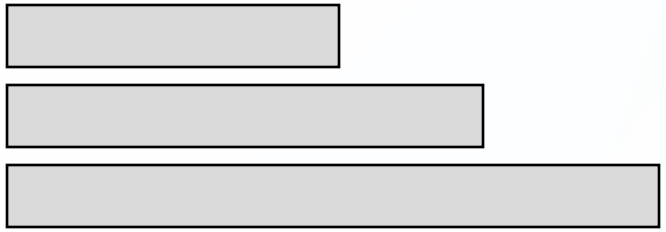
A. Condition codes
B. Condition register
C. Compare and branch
D. None of the above

Choice 4: Instruction Format

Fixed (e.g., all RISC processors -- SPARC, MIPS, Alpha)

Variable (VAX, ...)

Hybrid



- Tradeoffs?
- Conclusions?