ECE C61
Computer Architecture
Lecture 3 – Instruction Set Architecture

Prof. Alok N. Choudhary

choudhar@ece.northwestern.edu

Today's Lecture

Quick Review of Last Week

Classification of Instruction Set Architectures

Instruction Set Architecture Design Decisions

- Operands

Annoucements

- Operations
- Memory Addressing
- Instruction Formats

Instruction Sequencing

Language and Compiler Driven Decisions

Summary of Lecture 2

Two Notions of "Performance"

Plane	DC to Paris	Speed	Passengers	Throughput (pmph)
Boeing 747	6.5 hours	610 mph	470	286,700
Concorde	3 hours	1350 mph	132	178,200

Which has higher performance?

Execution time (response time, latency, …)

- Time to do a task

Throughput (bandwidth, …)

- Tasks per unit of time

Response time and throughput often are in opposition

Definitions

Performance is typically in units-per-second

- bigger is better

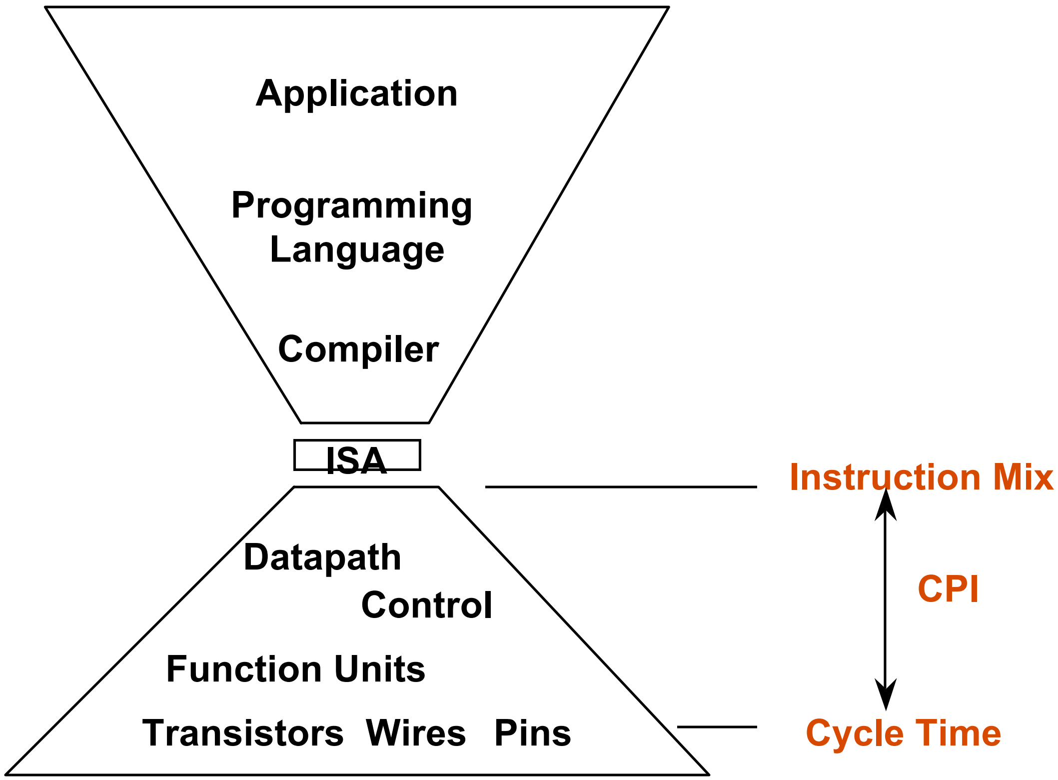
If we are primarily concerned with response time

- $$\boxed{performance = \frac{1}{ExecutionTime}}$$

" X is n times faster than Y" means

$$\frac{ExecutionTime_y}{ExecutionTime_x} = \frac{Performance_x}{Performance_y} = n$$

Organizational Trade-offs



CPI is a useful design measure relating the Instruction Set Architecture with the Implementation of that architecture, and the program measured

Principal Design Metrics: CPI and Cycle Time

$$Performance = \frac{1}{ExecutionTime}$$

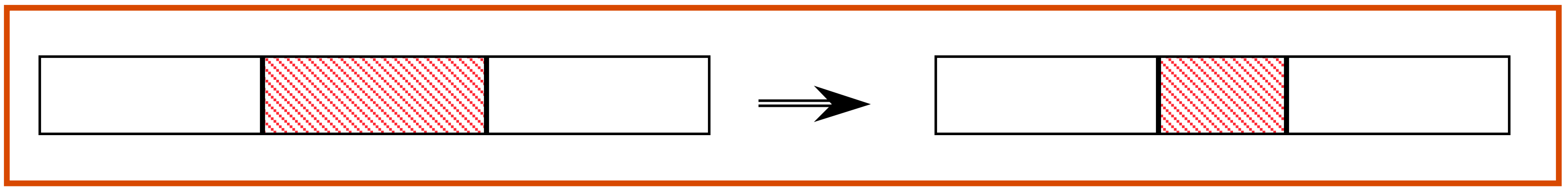
$$\boxed{Performance = \frac{1}{CPI \times CycleTime}}$$

$$Performance = \frac{1}{\frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}} = \frac{Instructions}{Seconds}$$

Amdahl's “Law”: Make the Common Case Fast

Speedup due to enhancement E:

$$\text{Speedup(E)} = \frac{\text{ExTime w/o E}}{\text{ExTime w/ E}} = \frac{\text{Performance w/ E}}{\text{Performance w/o E}}$$



Suppose that enhancement E accelerates a fraction F of the task by a factor S and the remainder of the task is unaffected then,

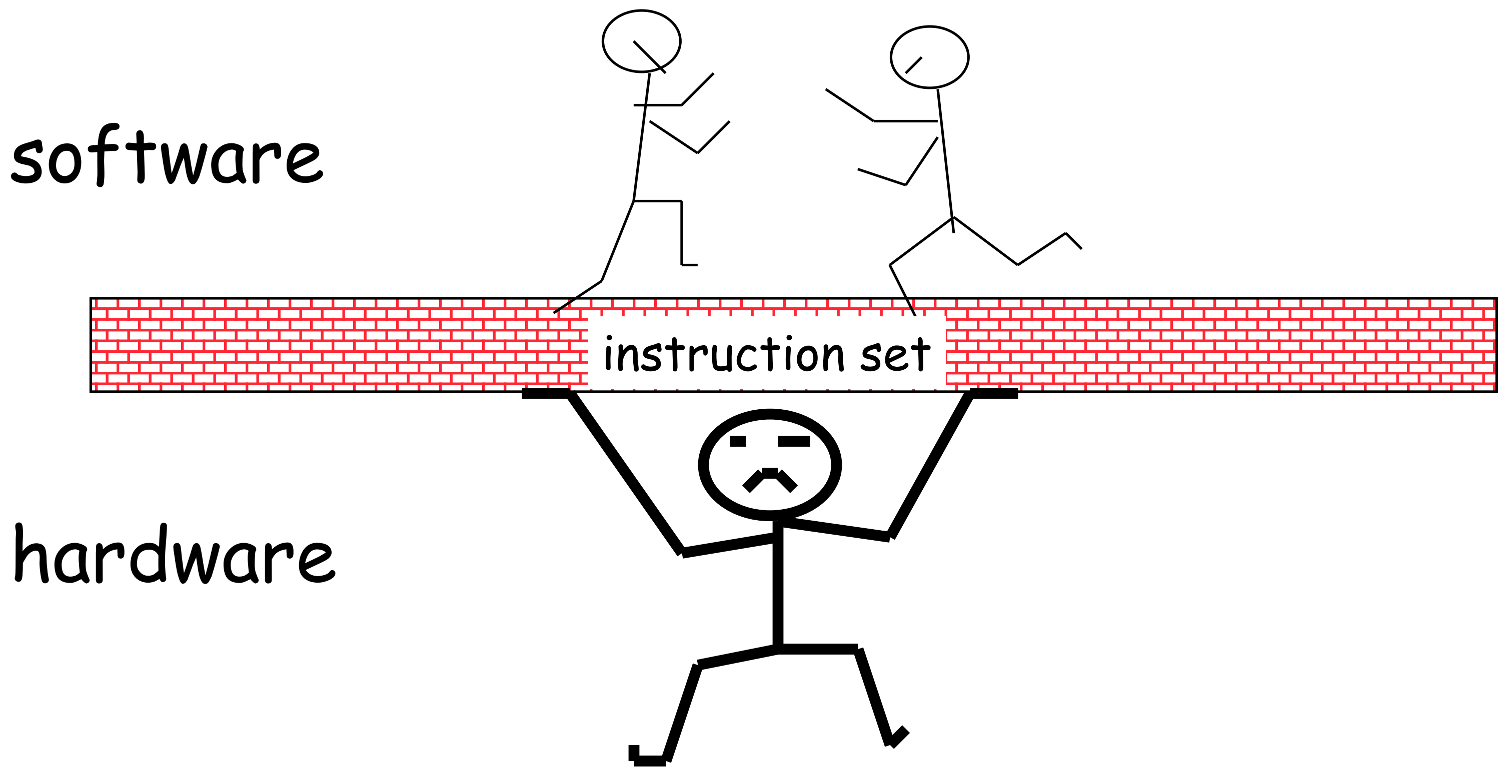
ExTime(with E) = ((1-F) + F/S) X ExTime(without E)

Performance improvement is limited by how much the improved feature is used → Invest resources where time is spent.

$$Speedup_{overall} = \frac{ExecTime_{old}}{ExecTime_{new}} = \frac{1}{\frac{Fraction_{enhanced}}{Speedup_{enhanced}} + (1 - Fraction_{enhanced})}$$

Classification of Instruction Set Architectures

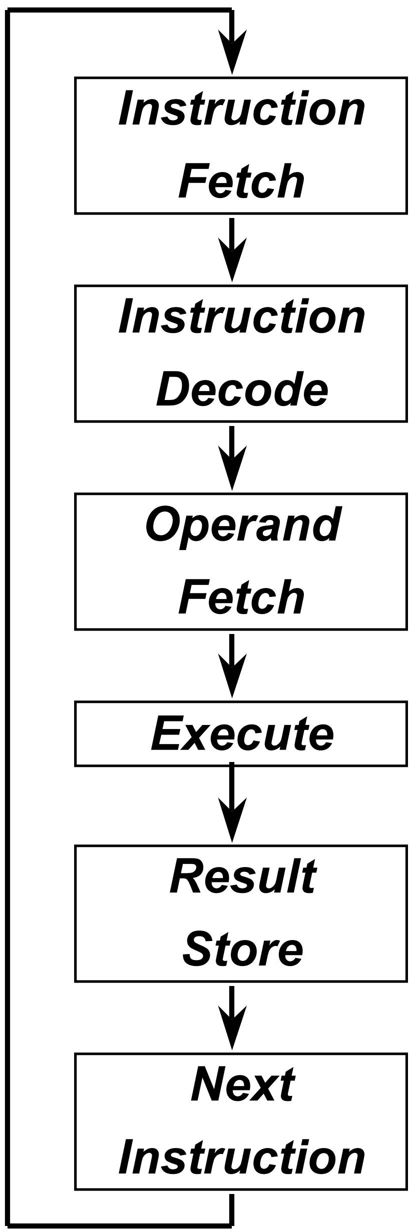
Instruction Set Design



Multiple Implementations: 8086 → Pentium 4

ISAs evolve: MIPS-I, MIPS-II, MIPS-II, MIPS-IV, MIPS,MDMX, MIPS-32, MIPS-64

Typical Processor 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 register or storage for later use

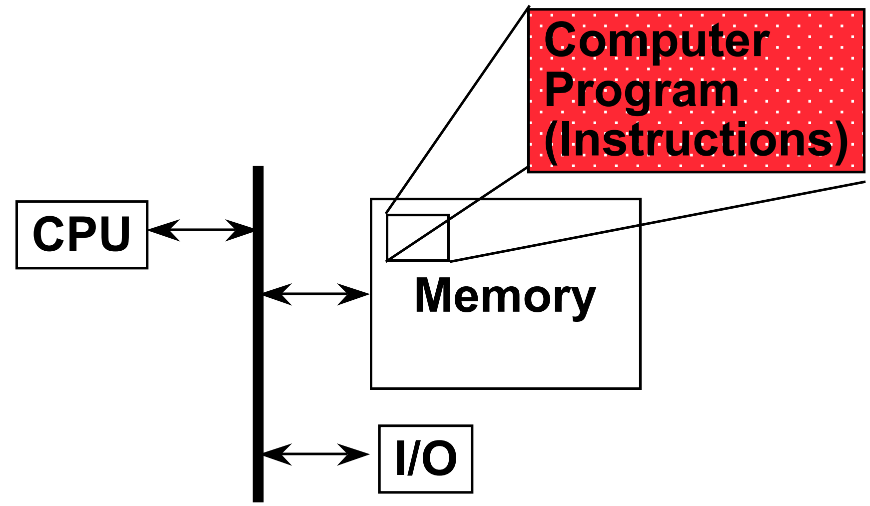
Determine successor instruction

Instruction and Data Memory: Unified or Separate

Programmer's View

ADD	**01010**
SUBTRACT	**01110**
AND	**10011**
OR	**10001**
COMPARE	**11010**
.	.
.	.
.	.

Computer's View



Princeton (Von Neumann) Architecture

--- Data and Instructions mixed in same unified memory

--- Program as data

--- Storage utilization

--- Single memory interface

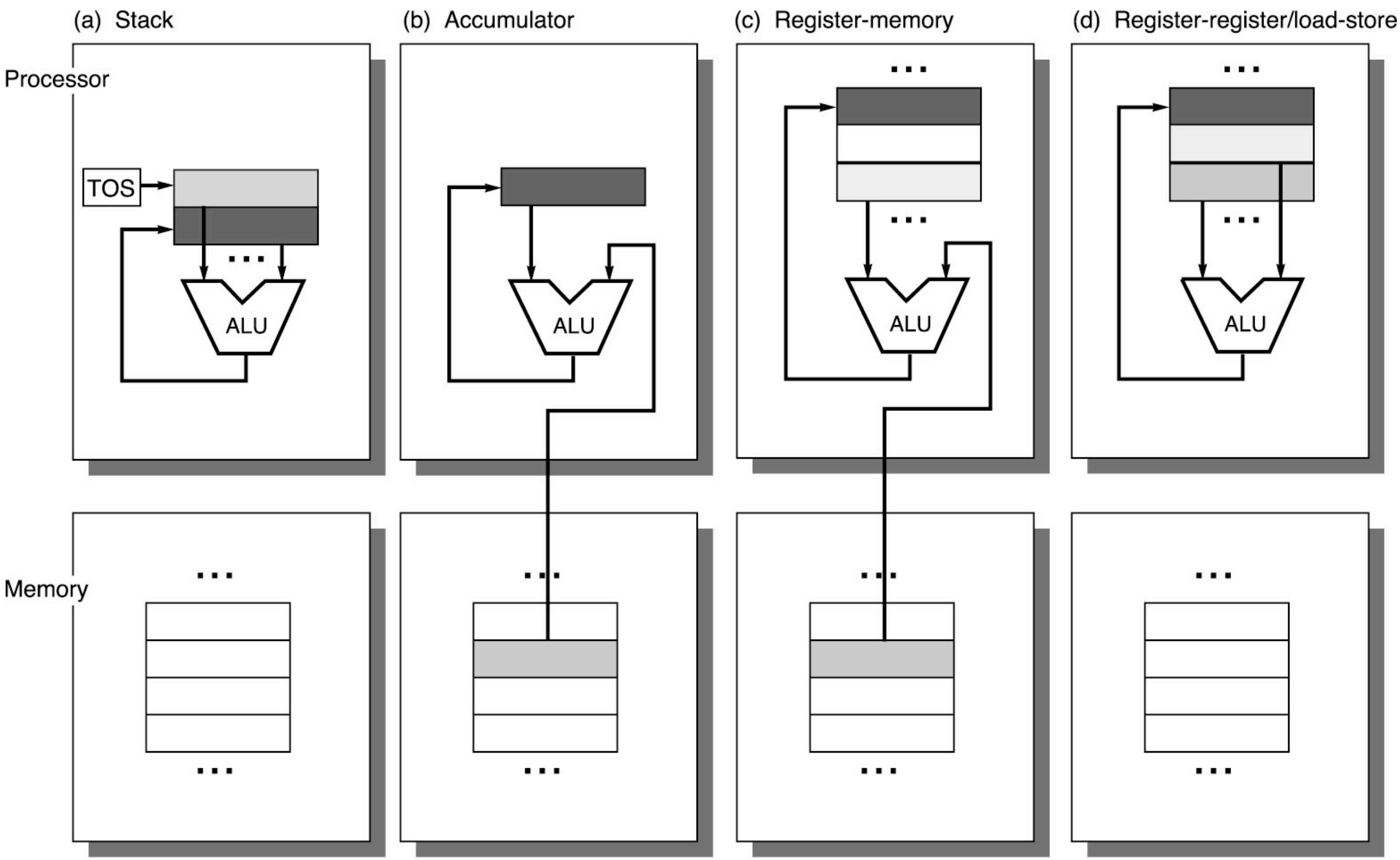
Harvard Architecture

--- Data & Instructions in separate memories

--- Has advantages in certain high performance implementations

--- Can optimize each memory

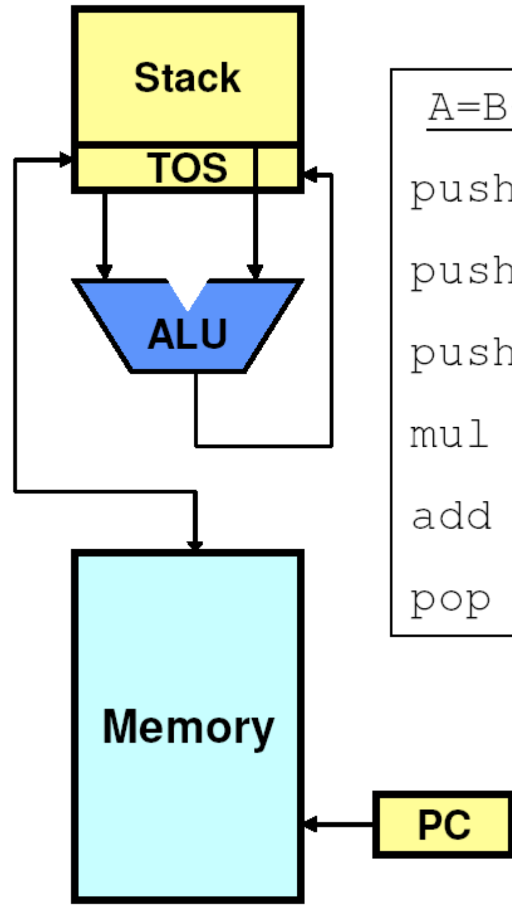
Basic Addressing Classes



Declining cost of registers

Stack Architectures

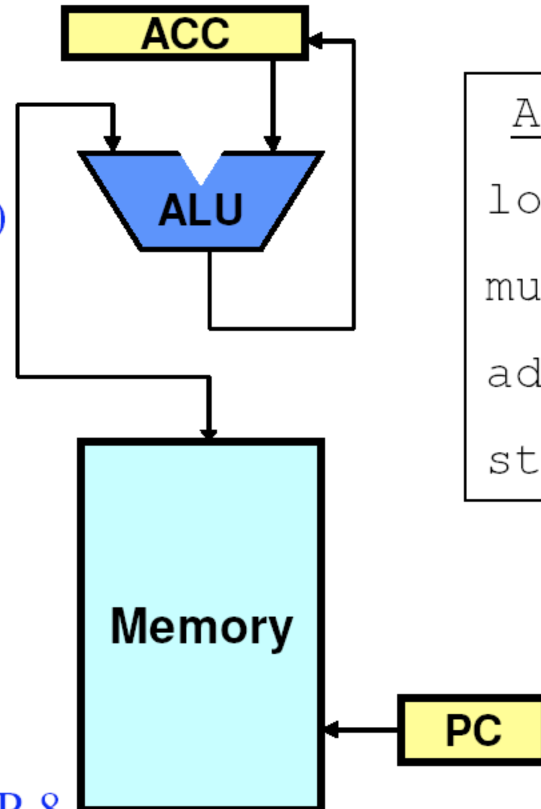
- Stack: First-In Last-Out data structure (FILO)
- Instruction operands
  - None for ALU operations
  - One for push/pop
- Advantages:
  - Short instructions
  - Compiler is easy to write
- Disadvantages
  - Code is inefficient
    - Fix: random access to stacked values
  - Stack size & access latency
    - Fix : register file or cache for top entries
- Examples
  - 60s: Burroughs B5500/6500, HP 3000/70
  - Today: Java VM




```
A=B+(C*D)
push B
push C
push D
mul
add
pop A
```

Accumulator Architectures

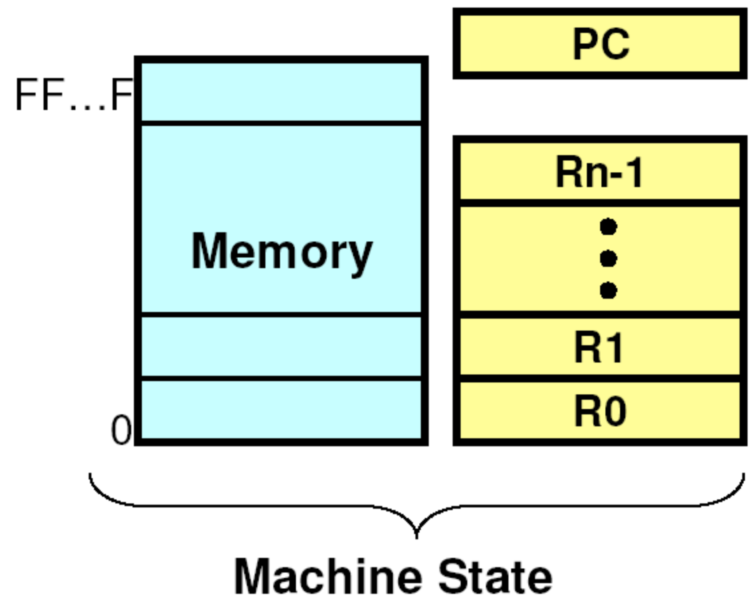
- Single register (accumulator)
- Instructions
  - ALU (Acc ← Acc + *M)
  - Load to accumulator (Acc ← *M)
  - Store from accumulator (*M ← Acc)
- Instruction operands
  - One explicit (memory address)
  - One implicit (accumulator)
- Attributes:
  - Short instructions
  - Minimal internal state; simple design
  - Many loads and stores
- Examples:
  - Early machines: IBM 7090, DEC PDP-8
  - Today: DSP architectures

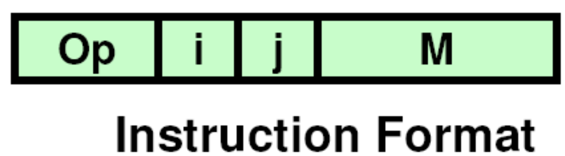



```
A=B+(C*D)
load C
mul D
add B
store A
```

Register-Set Architectures

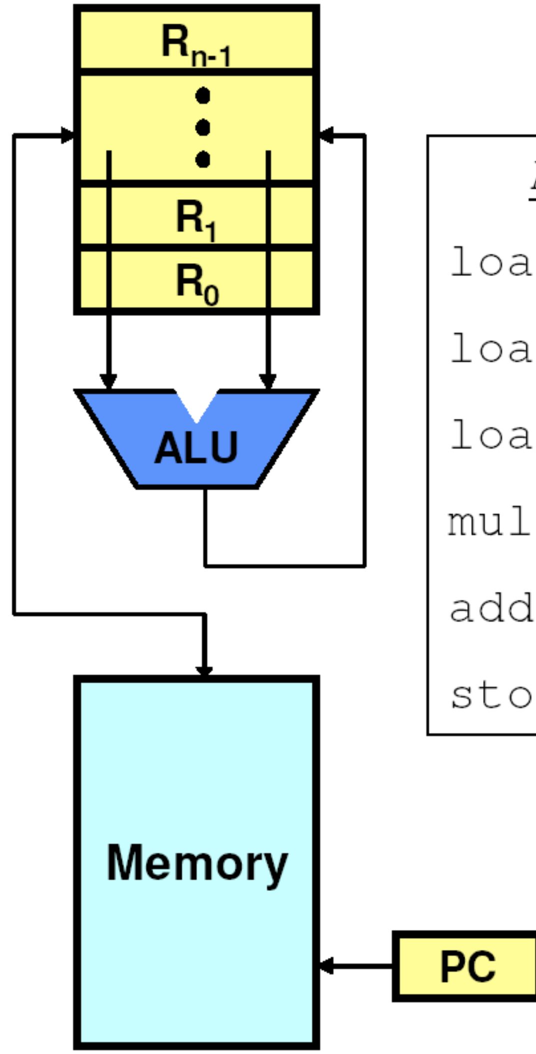
- General Purpose Registers (GPRs)
- Registers:
  - Explicitly managed memory for holding recently used values
- The dominant architecture: CDC 6600, IBM 360/370, PDP-11, 68000, RISC
- Advantages:
  - Allows fast access to temporary values
  - Permits clever compiler optimization
  - Reduced traffic to memory
- Disadvantages:
  - Longer instructions (than accumulator designs)





Register-to-Register: Load-Store Architectures

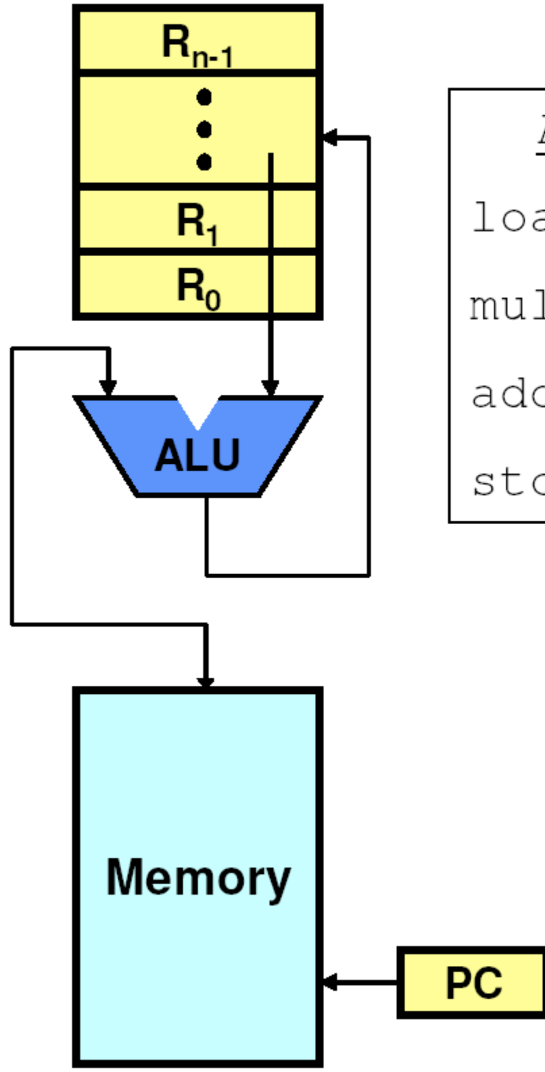
- No memory addresses in ALU ops
- Typically 3-operand ALU ops
  - Bigger encoding, but simplifies register allocation
- Advantages
  - Simple fixed-length instructions
  - Easily pipelined
- Disadvantages
  - Higher instruction count
- Examples
  - CDC6600, CRAY-1, most RISCs




```
A=B+(C*D)
load R1<-C
load R2<-D
load R3<-B
mul  R4<-R1*R2
add  R5<-R4+R3
store A<-R5
```

Register-to-Memory Architectures

- One memory address in ALU ops
- Typically 2-operand ALU ops
- Advantages
  - Small instruction count
  - Dense encoding
- Disadvantages
  - Result destroys an operand
  - Instruction length varies
  - Clocks per instruction varies
  - Harder to pipeline
- Examples
  - IBM 360/370, VAX




```
A=B+(C*D)
load R1<-C
mul  R1<-R1*D
add  R1<-R1+B
store A<-R1
```

Memory-to-Memory Architectures

- All ALU operands from memory addresses
- Advantages
  - No register wastage
  - Lowest instruction count
- Disadvantages
  - Large variation in instruction length
  - Large variation in clocks per instructions
  - Huge memory traffic
- Examples
  - VAX

```
      D=B+(C*D)
mul D <- C*D
add D <- D+B
```

Instruction Set Architecture Design Decisions

Basic Issues in Instruction Set Design

What data types are supported. What size.

What operations (and how many) should be provided

- LD/ST/INC/BRN sufficient to encode any computation, or just Sub and Branch!
- But not useful because programs too long!

How (and how many) operands are specified

Most operations are dyadic (eg, A <- B + C)

- Some are monadic (eg, A <- ~B)

Location of operands and result

- where other than memory?
- how many explicit operands?
- how are memory operands located?
- which can or cannot be in memory?
- How are they addressed

How to encode these into consistent instruction formats

- Instructions should be multiples of basic data/address widths
- Encoding

Typical instruction set:

•32 bit word
•basic operand addresses are 32 bits long
•basic operands, like integers, are 32 bits long
•in general case, instruction could reference 3 operands (A := B + C)

Typical challenge:

•encode operations in a small number of bits

Driven by static measurement and dynamic tracing of selected benchmarks and workloads.

Operands

Comparing Number of Instructions

Code sequence for (C = A + B) for four classes of instruction sets:

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R1,B	Load R2,B
Add	Store C	Store C, R1	Add R3,R1,R2
Pop C			Store C,R3

$$ExecutionTime = \frac{1}{Performance} = Instructions \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

Examples of Register Usage

Number of memory addresses per typical ALU instruction

Maximum number of operands per typical ALU instruction

		Examples
0	3	SPARC, MIPS, Precision Architecture, Power PC
1	2	Intel 80x86, Motorola 68000
2	2	VAX (also has 3-operand formats)
3	3	VAX (also has 2-operand formats)

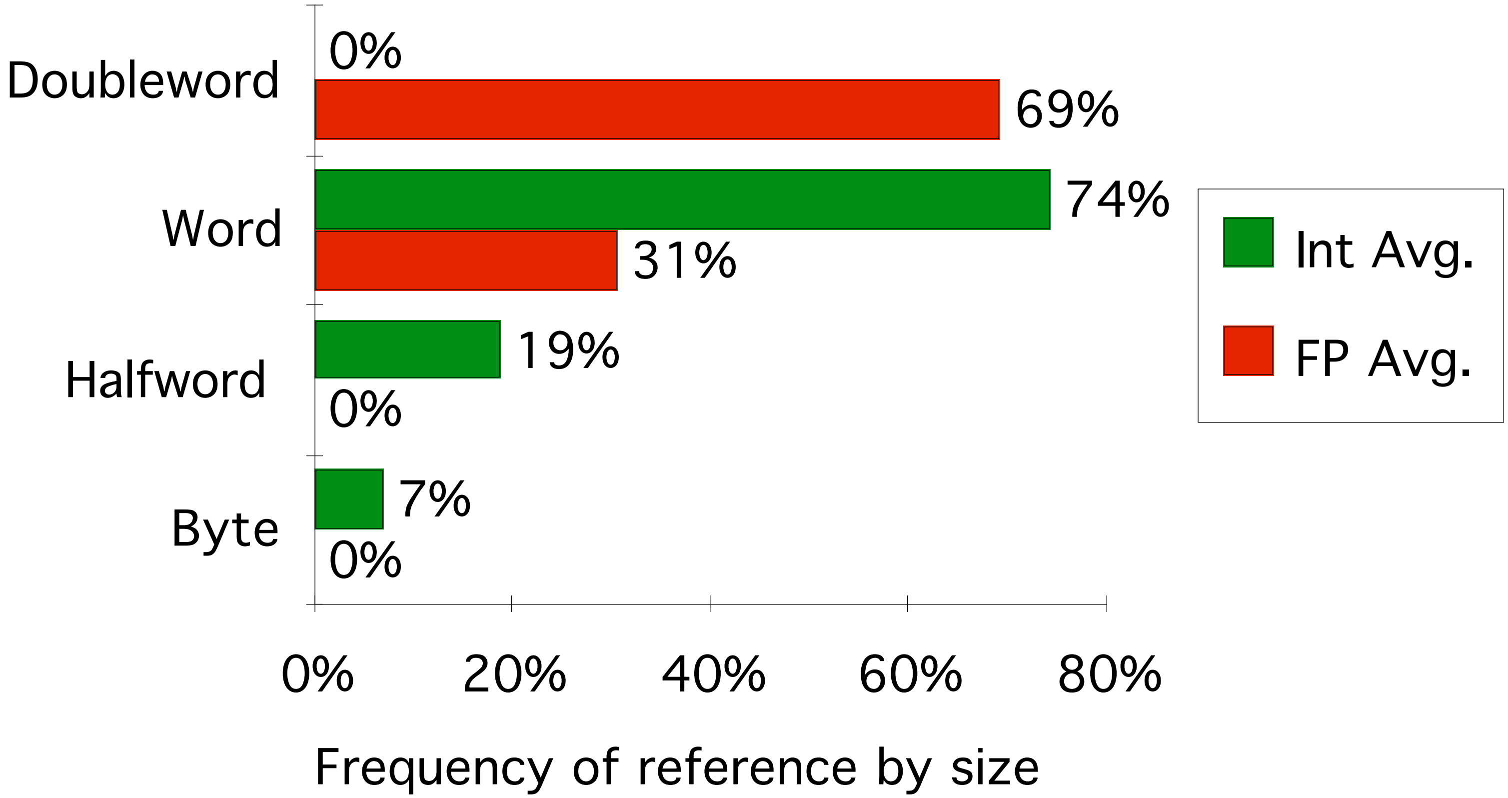
General Purpose Registers Dominate

1975-2002 all machines use general purpose registers

Advantages of registers

- Registers are faster than memory
- Registers compiler technology has evolved to efficiently generate code for register files
  - E.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack
- Registers can hold variables
  - Memory traffic is reduced, so program is sped up (since registers are faster than memory)
- Code density improves (since register named with fewer bits than memory location)
- Registers imply operand locality

Operand Size Usage



- Support for these data sizes and types:
  8-bit, 16-bit, 32-bit integers and
  32-bit and 64-bit IEEE 754 floating point numbers

Announcements

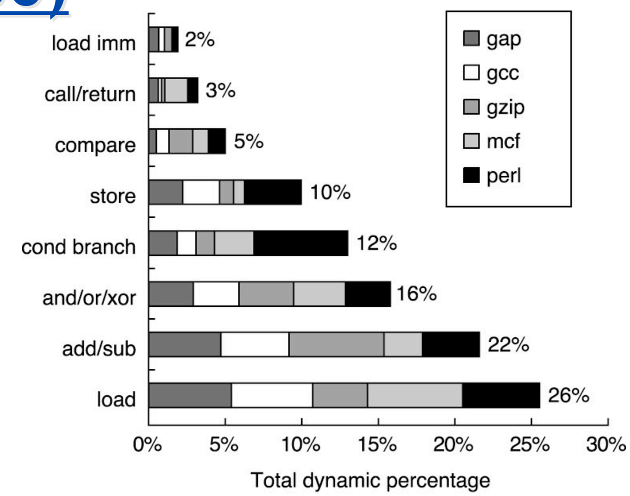
Next lecture

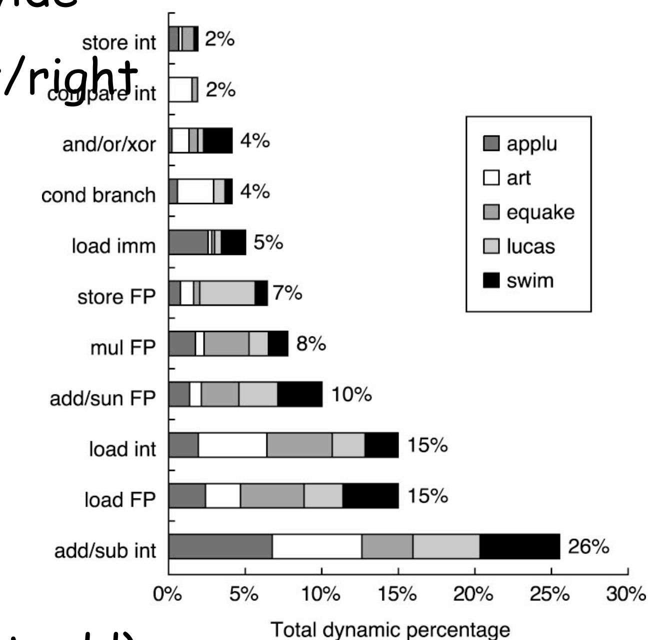
- MIPS Instruction Set

Operations

Typical Operations (little change since 1960)

Data Movement	Load (from memory) Store (to memory) memory-to-memory move register-to-register move input (from I/O device) output (to I/O device) push, pop (to/from stack)
Arithmetic	integer (binary + decimal) or FP Add, Subtract, Multiply, Divide
Shift	shift left/right, rotate left/right
Logical	not, and, or, set, clear
Control (Jump/Branch)	unconditional, conditional
Subroutine Linkage	call, return
Interrupt	trap, return
Synchronization	test & set (atomic r-m-w)
String	search, translate
Graphics (MMX)	parallel subword ops (4 16bit add)





Top 10 80x86 Instructions

- Rank instruction Integer Average Percent total executed

Rank	instruction	Integer Average Percent 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%

- Simple instructions dominate instruction frequency

Memory Addressing

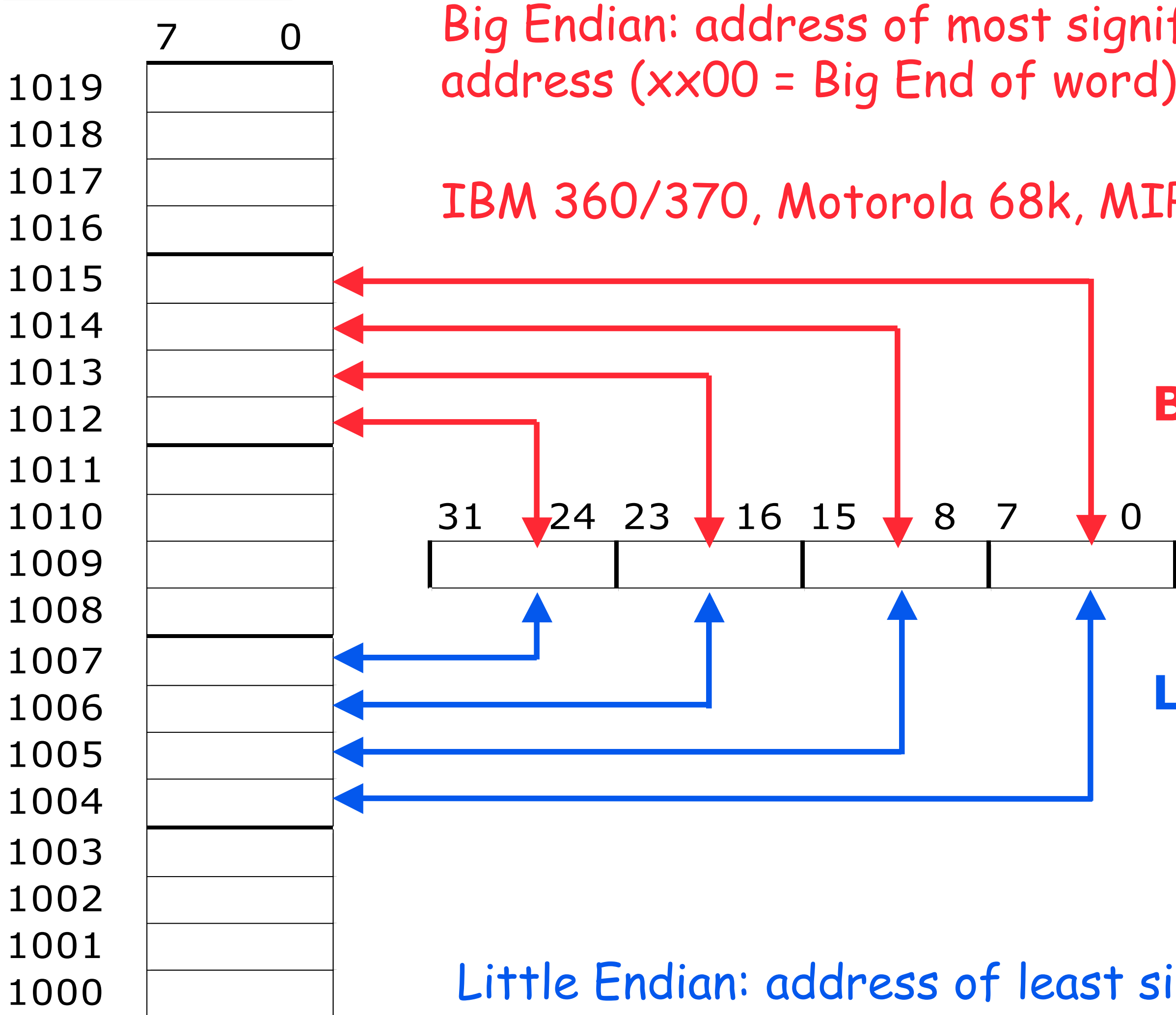
Memory Addressing

Since 1980, almost every machine uses addresses to level of 8-bits (byte)

Two questions for design of ISA:

- Since could read a 32-but word as four loads of bytes from sequential byte address of as one load word from a single byte address, how do byte addresses map onto words?
- Can a word be placed on any byte boundary?

Mapping Word Data into a Byte Addressable Memory: Endianess



Big Endian: address of most significant byte = word address (xx00 = Big End of word)

IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

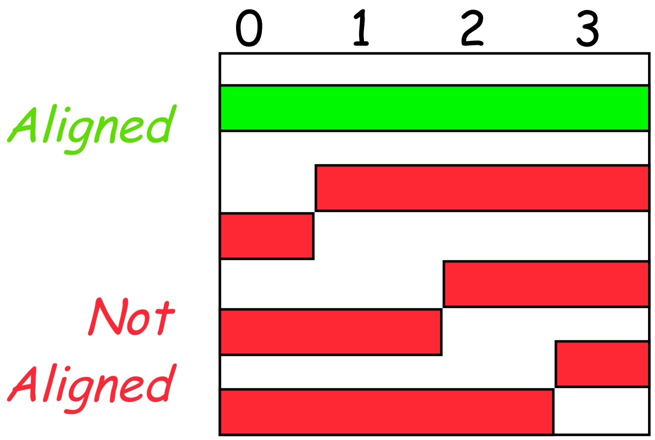
Big Endian

Little Endian

Little Endian: address of least significant byte = word address (xx00 = Little End of word)

Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

Mapping Word Data into a Byte Addressable Memory: Alignment

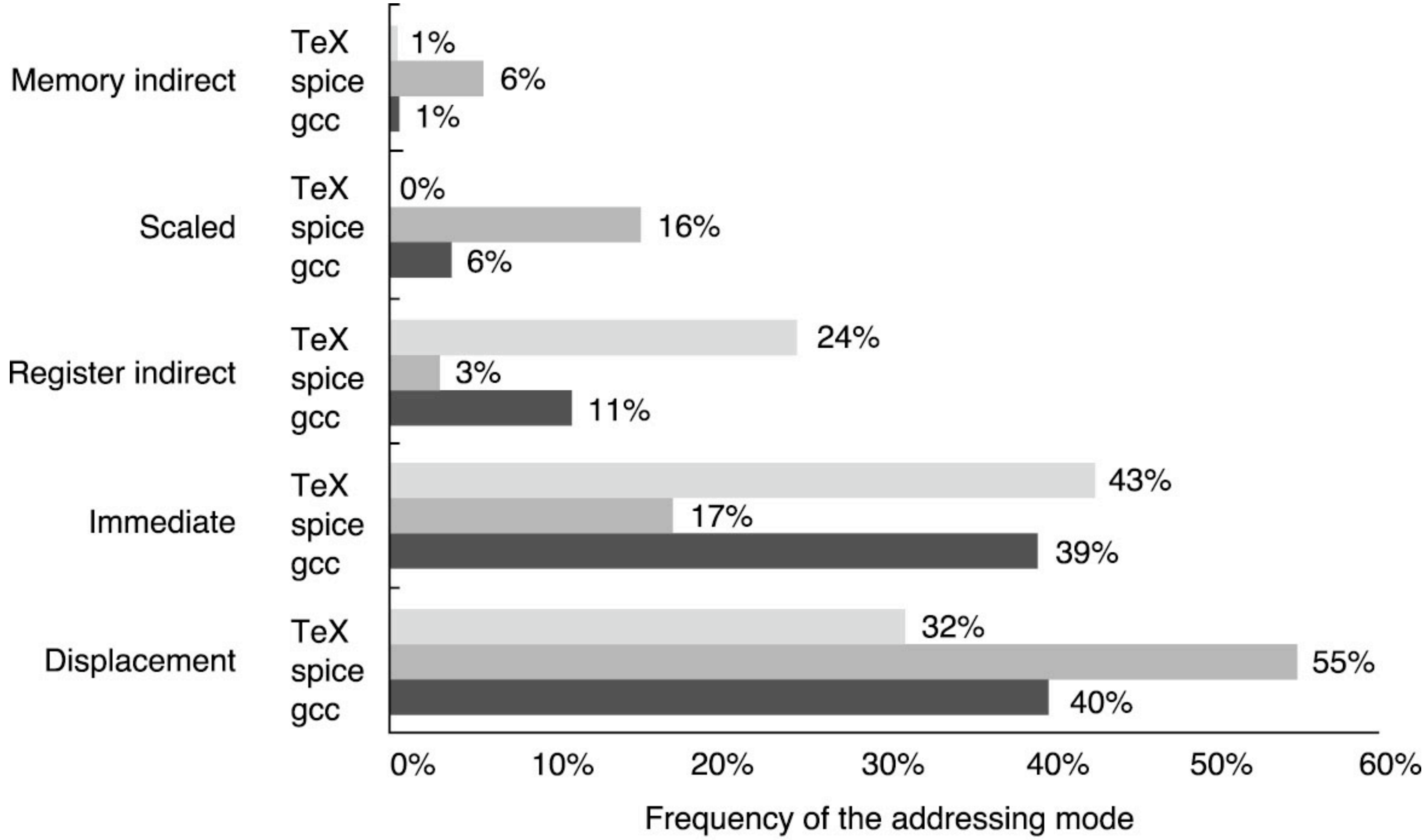


Alignment: require that objects fall on address that is multiple of their size.

Addressing Modes

- Addressing modes specify a constant, a register, or a location in memory
  - **Register** `add r1, r2` `r1 <- r1+r2`
  - **Immediate** `add r1, #5` `r1 <- r1+5`
  - **Direct** `add r1, (0x200)` `r1 <- r1+M[0x200]`
  - **Register indirect** `add r1, (r2)` `r1 <- r1+M[r2]`
  - **Displacement** `add r1, 100(r2)` `r1 <- r1+M[r2+100]`
  - **Indexed** `add r1, (r2+r3)` `r1 <- r1+M[r2+r3]`
  - **Scaled** `add r1, (r2+r3*4)` `r1 <- r1+M[r2+r3*4]`
  - **Memory indirect** `add r1, @(r2)` `r1 <- r1+M[M[r2]]`
  - **Auto-increment** `add r1, (r2)+` `r1 <- r1+M[r2], r2++`
  - **Auto-decrement** `add r1, -(r2)` `r2--, r1 <- r1+M[r2]`
- Complicated modes reduce instruction count at the cost of complex implementations

Common Memory Addressing Modes



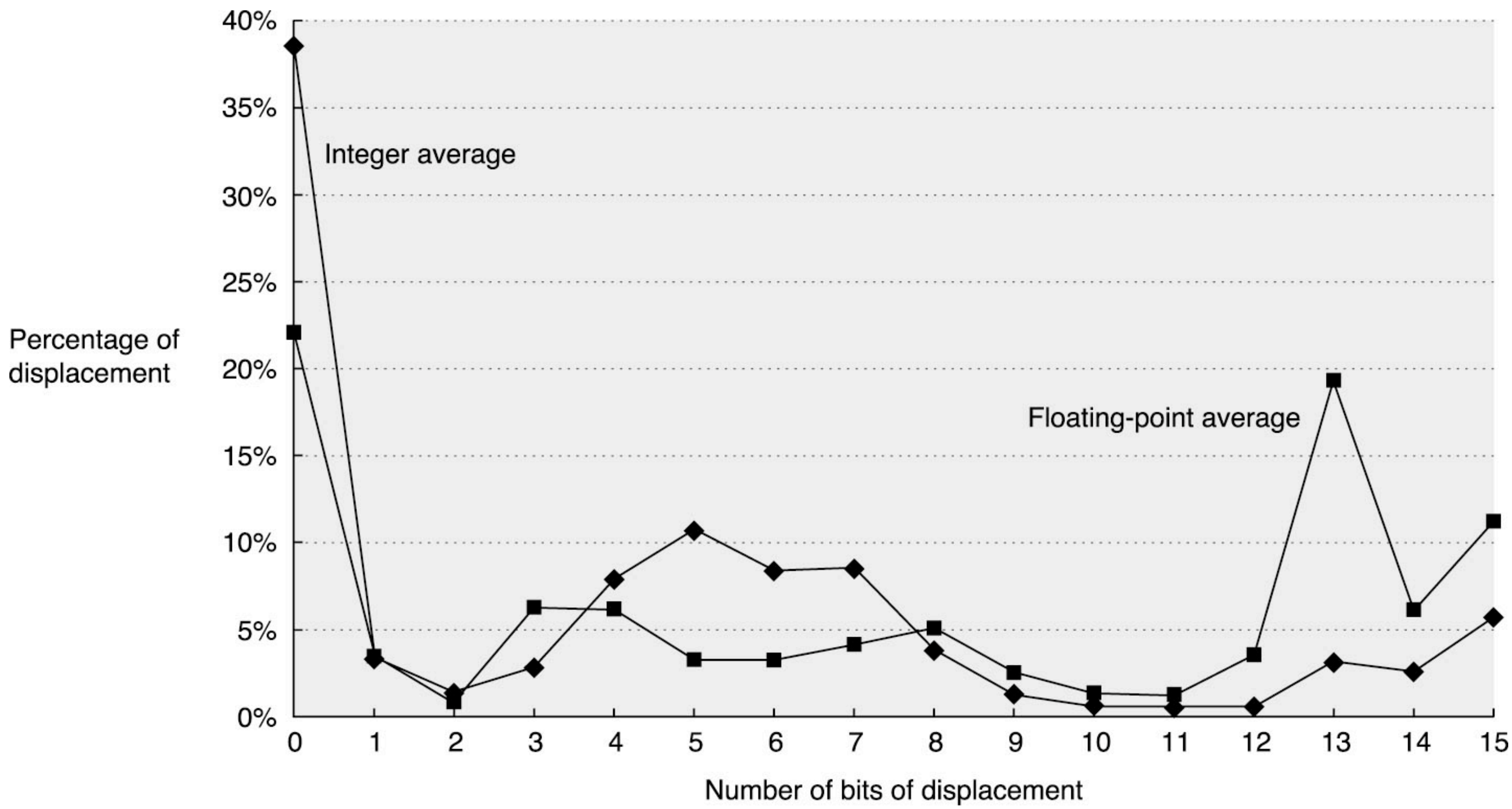
Measured on the VAX-11

Register operations account for 51% of all references

~75% - displacement and immediate

~85% - displacement, immediate and register indirect

Displacement Address Size

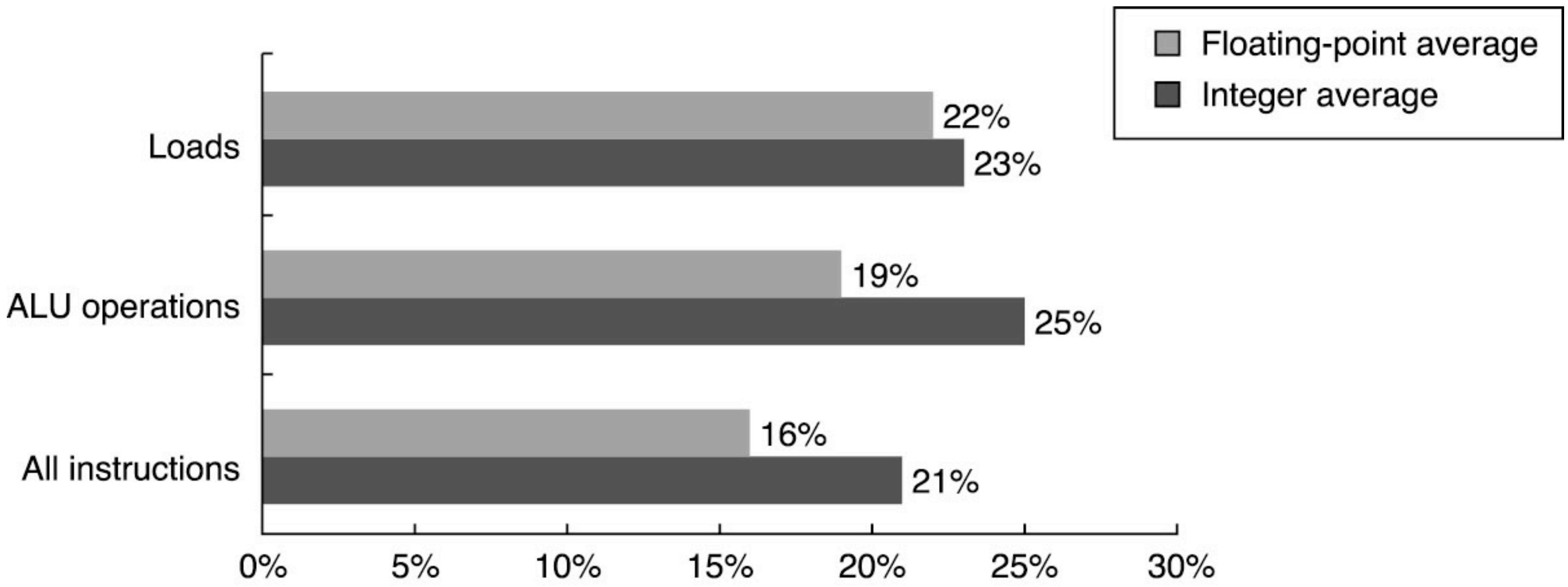


Average of 5 SPECint92 and 5 SPECfp92 programs

~1% of addresses > 16-bits

12 ~ 16 bits of displacement cover most usage (+ and -)

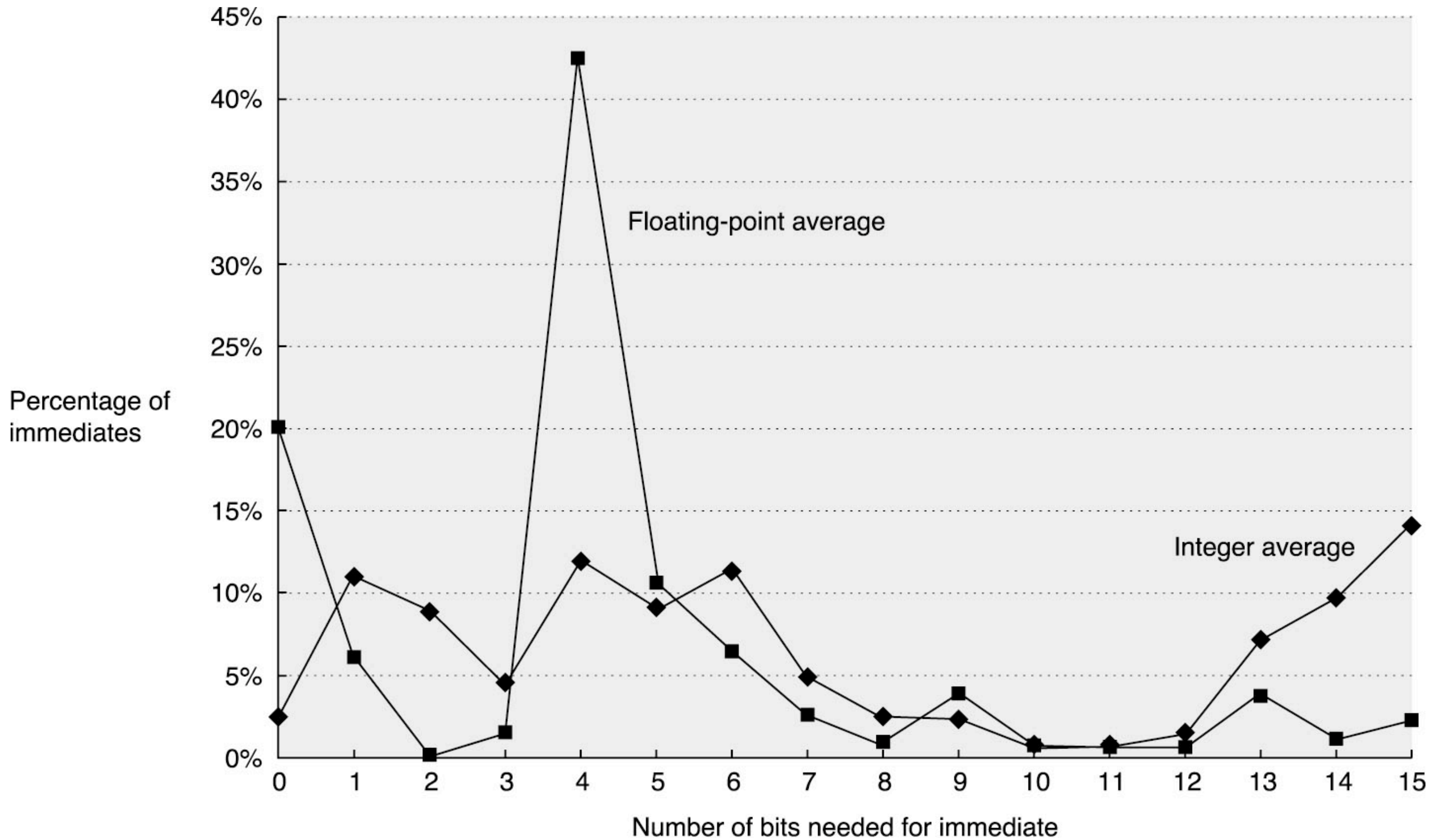
Frequency of Immediates (Instruction Literals)



~25% of all loads and ALU operations use immediates

15~20% of all instructions use immediates

Size of Immediates



50% to 60% fit within 8 bits

75% to 80% fit within 16 bits

Addressing Summary

Data Addressing modes that are important:

- Displacement, Immediate, Register Indirect

Displacement size should be 12 to 16 bits

Immediate size should be 8 to 16 bits

Instruction Formats

Instruction Format

Specify

- Operation / Data Type
- Operands

Stack and Accumulator architectures have implied operand addressing

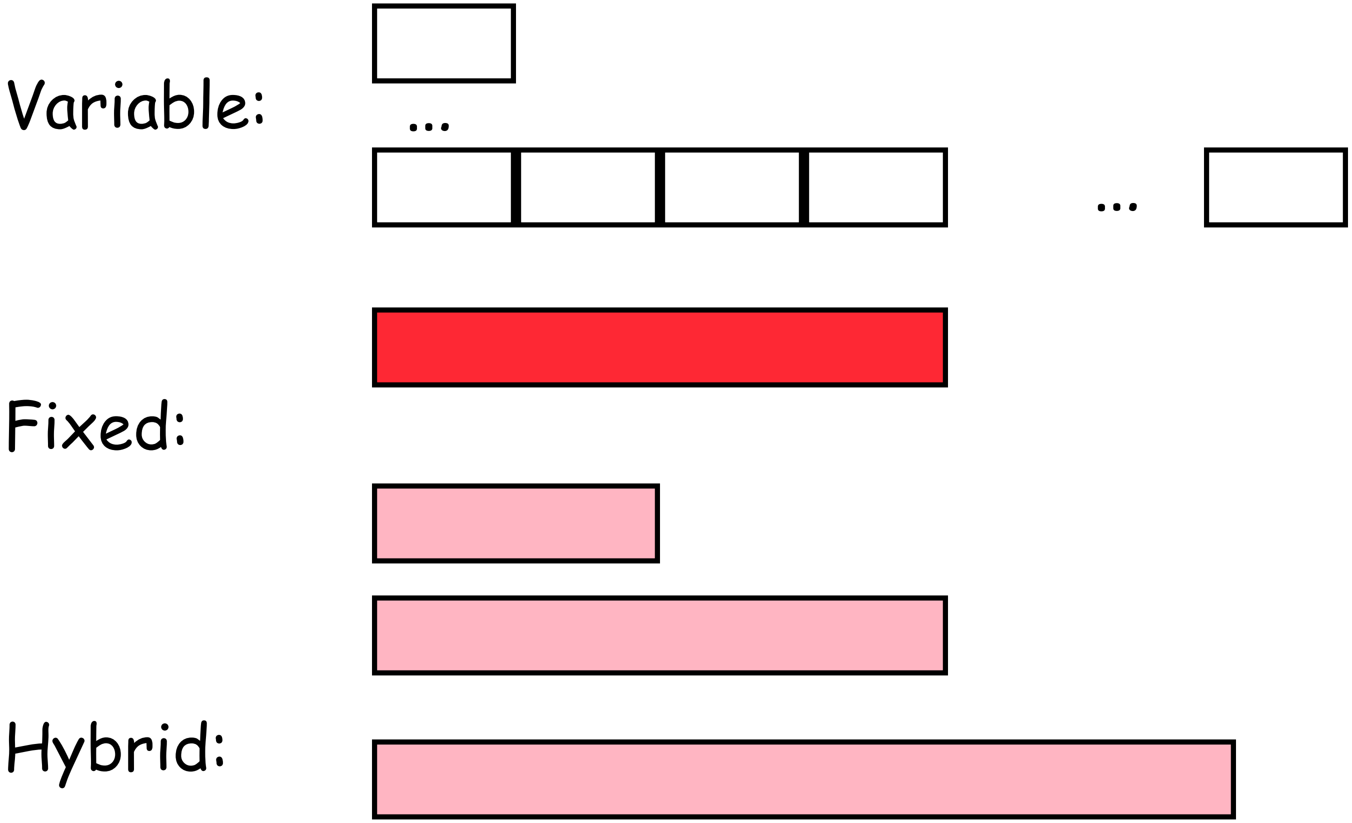
If have many memory operands per instruction and/or many addressing modes:

- Need one address specifier per operand

If have load-store machine with 1 address per instruction and one or two addressing modes:

- Can encode addressing mode in the opcode

Encoding



If code size is most important, use variable length instructions

If performance is most important, use fixed length instructions

Recent embedded machines (ARM, MIPS) added optional mode to execute subset of 16-bit wide instructions (Thumb, MIPS16); per procedure decide performance or density

Some architectures actually exploring on-the-fly decompression for more density.

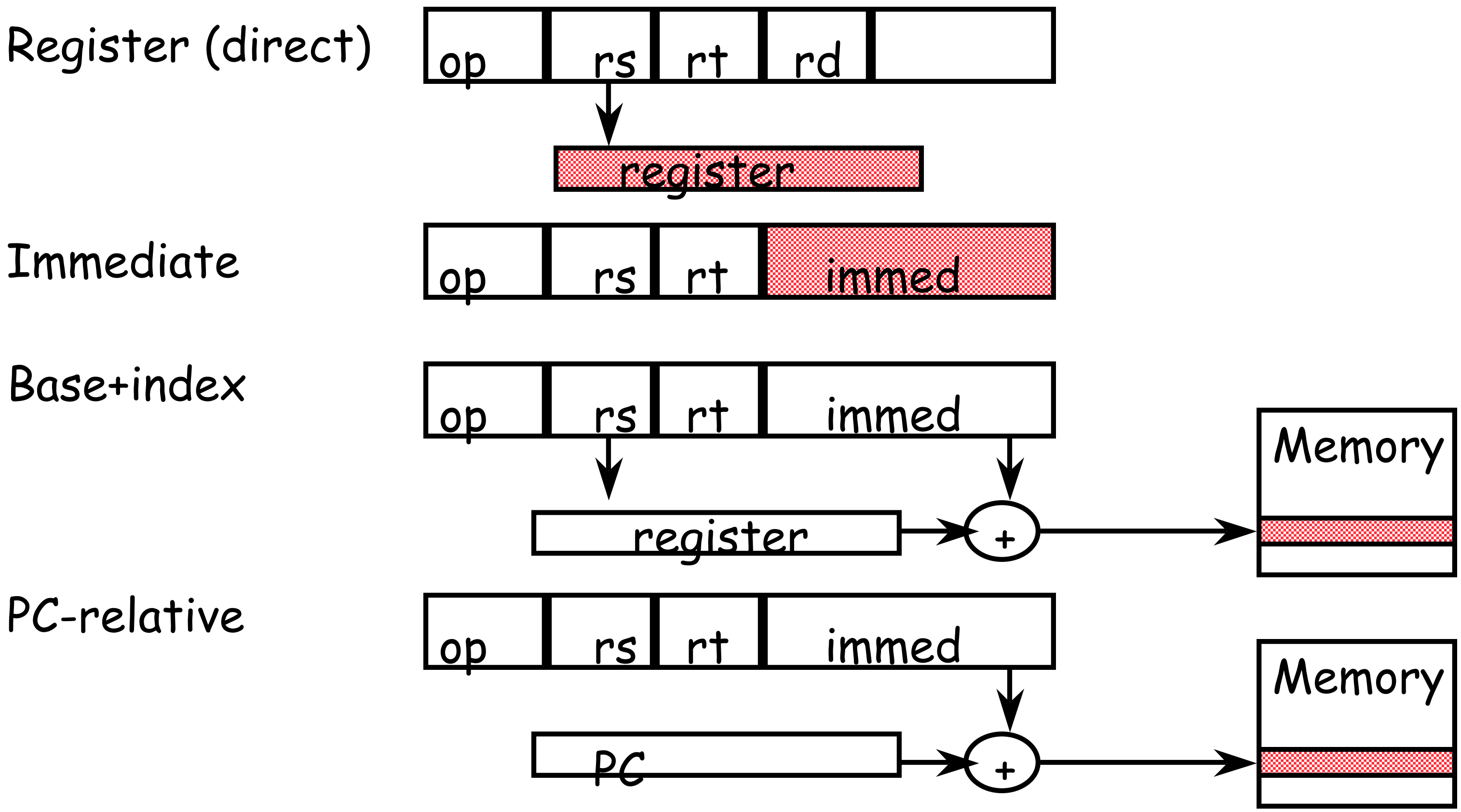
Operation Summary

Support these simple instructions, since they will dominate the number of instructions executed:

load,
store,
add,
subtract,
move register-register,
and,
shift,
compare equal, compare not equal,
branch,
jump,
call,
return;

Example: MIPS Instruction Formats and Addressing Modes

- All instructions 32 bits wide



Instruction Set Design Metrics

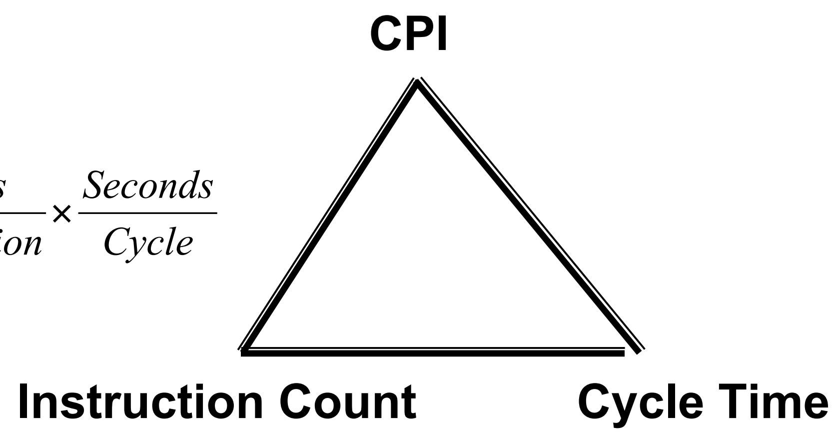
Static Metrics

- How many bytes does the program occupy in memory?

Dynamic Metrics

- How many instructions are executed?
- How many bytes does the processor fetch to execute the program?
- How many clocks are required per instruction?
- How "lean" a clock is practical?

$$ExecutionTime = \frac{1}{Performance} = Instructions \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

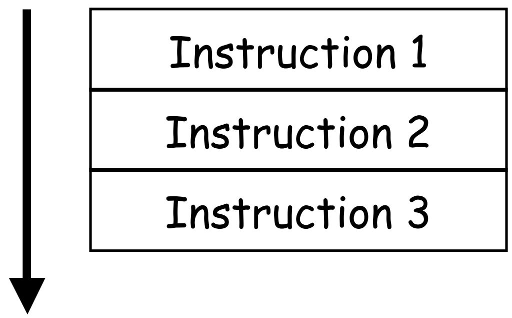


Instruction Sequencing

Instruction Sequencing

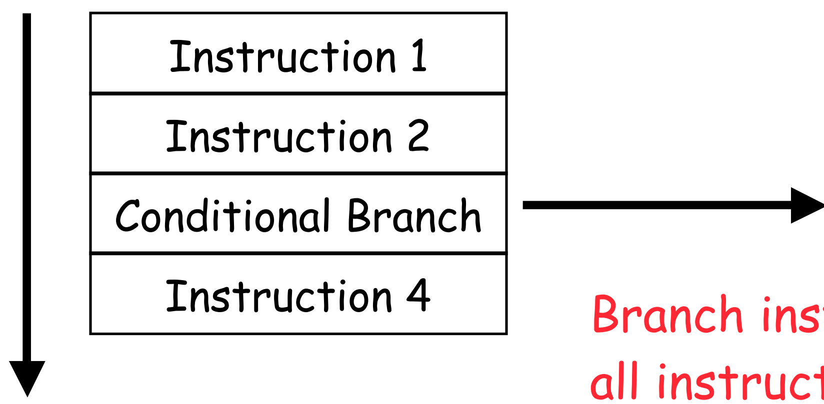
The next instruction to be executed is typically implied

- Instructions execute sequentially
- Instruction sequencing increments a Program Counter



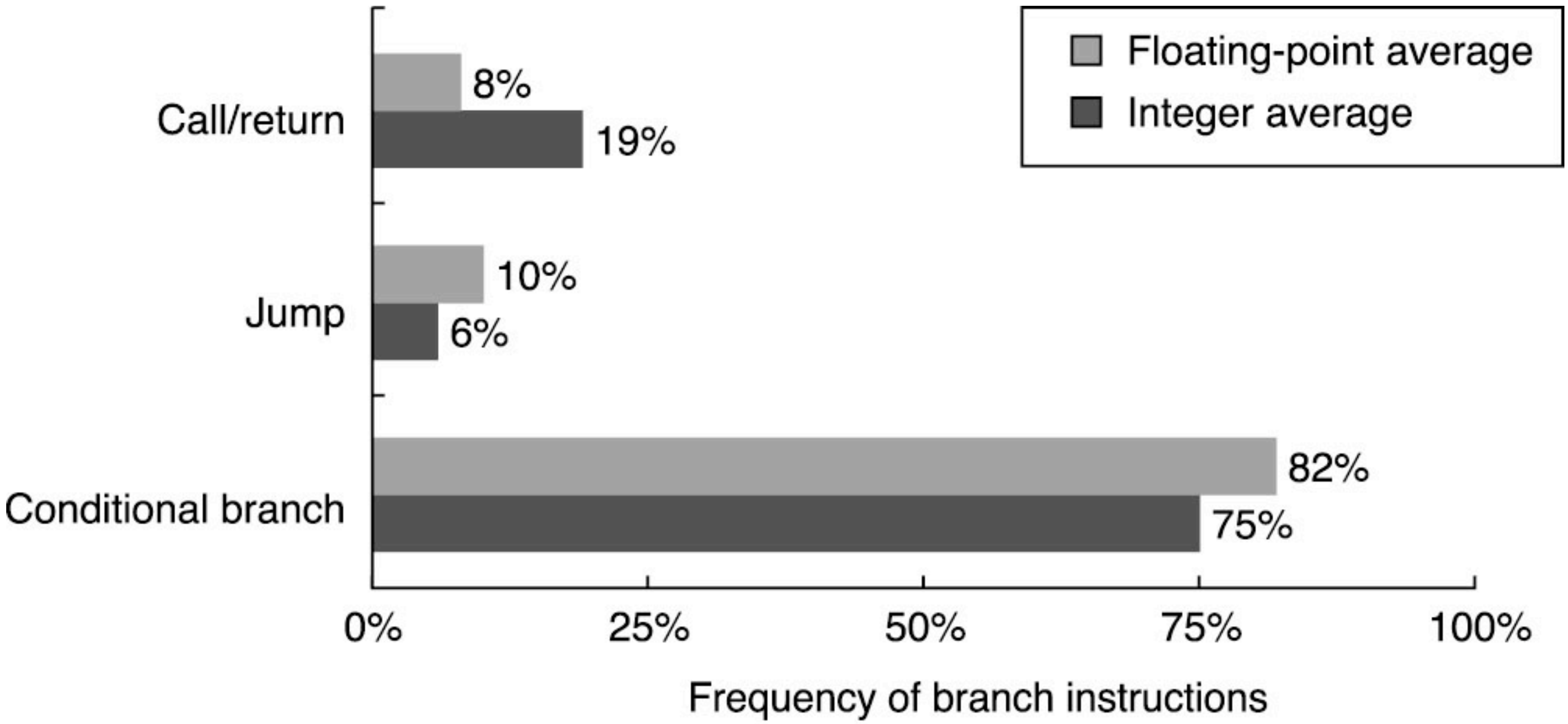
Sequencing flow is disrupted conditionally and unconditionally

- The ability of computers to test results and conditionally instructions is one of the reasons computers have become so useful



Branch instructions are ~20% of all instructions executed

Dynamic Frequency



Condition Testing

° Condition Codes

Processor status bits are set as a side-effect of arithmetic instructions (possibly on Moves) or explicitly by compare or test instructions.

ex: add r1, r2, r3

bz label

° Condition Register

Ex: cmp r1, r2, r3

bgt r1, label

° Compare and Branch

Ex: bgt r1, r2, label

Condition Codes

Setting CC as side effect can reduce the # of instructions

```
X:    .
      .
      .
      SUB  r0, #1, r0
      BRP  X
```

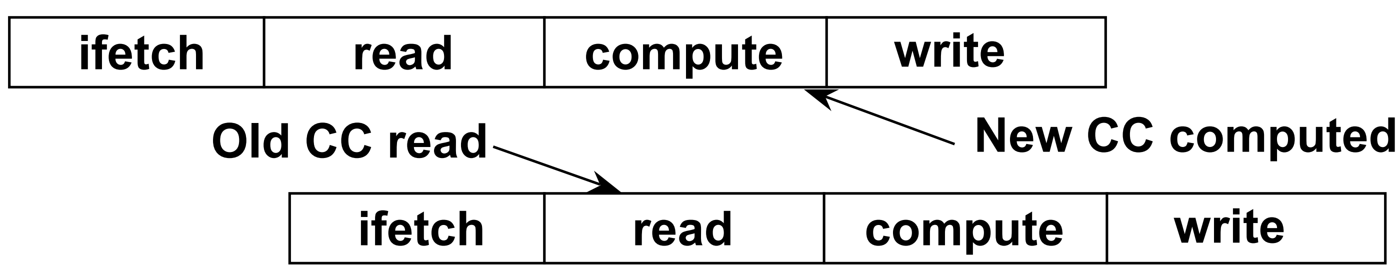
vs.

```
X:    .
      .
      .
      SUB  r0, #1, r0
      CMP  r0, #0
      BRP  X
```

But also has disadvantages:

**--- not all instructions set the condition codes
which do and which do not often confusing!**
e.g., shift instruction sets the carry bit

--- dependency between the instruction that sets the CC and the one that tests it



Branches

--- Conditional control transfers

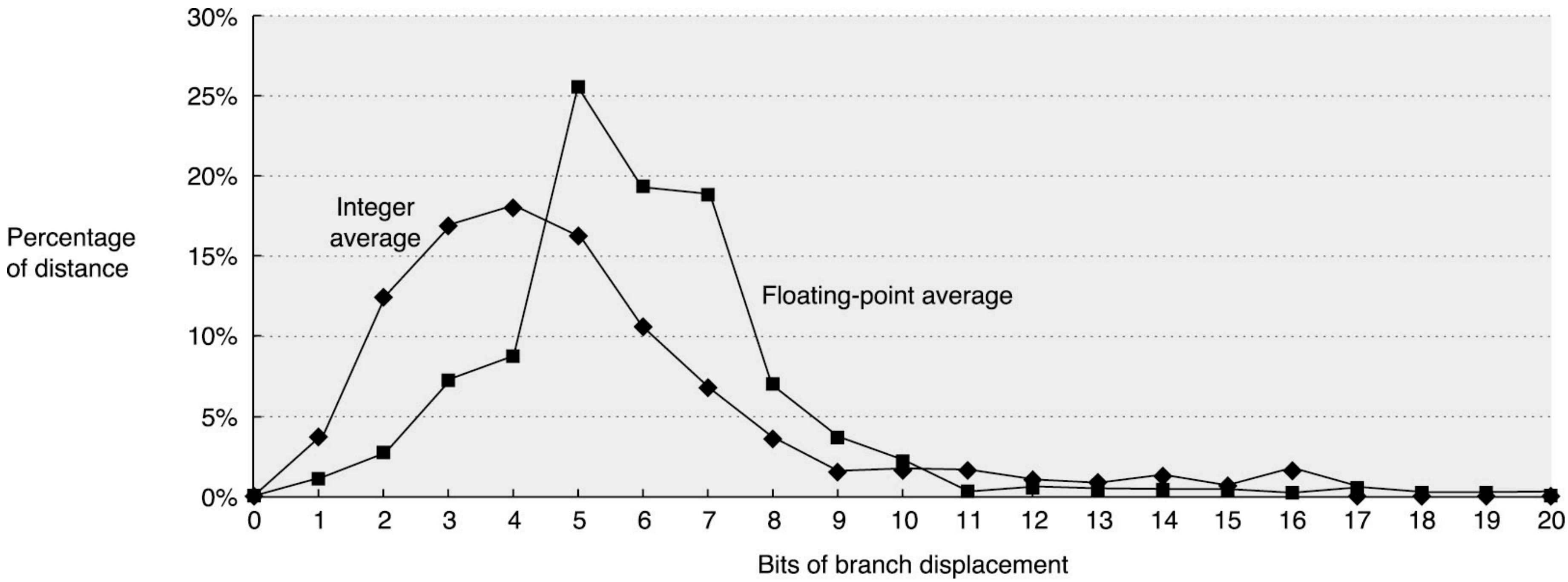
Four basic conditions:

N -- negative
Z -- zero
V -- overflow
C -- carry

Sixteen combinations of the basic four conditions:

Always	**Unconditional**
Never	**NOP**
Not Equal	$\sim Z$
Equal	Z
Greater	$\sim[Z + (N \oplus V)]$
Less or Equal	$Z + (N \oplus V)$
Greater or Equal	$\sim(N \oplus V)$
Less	$N \oplus V$
Greater Unsigned	$\sim(C + Z)$
Less or Equal Unsigned	$C + Z$
Carry Clear	$\sim C$
Carry Set	C
Positive	$\sim N$
Negative	N
Overflow Clear	$\sim V$
Overflow Set	V

Conditional Branch Distance



PC-relative (+-)

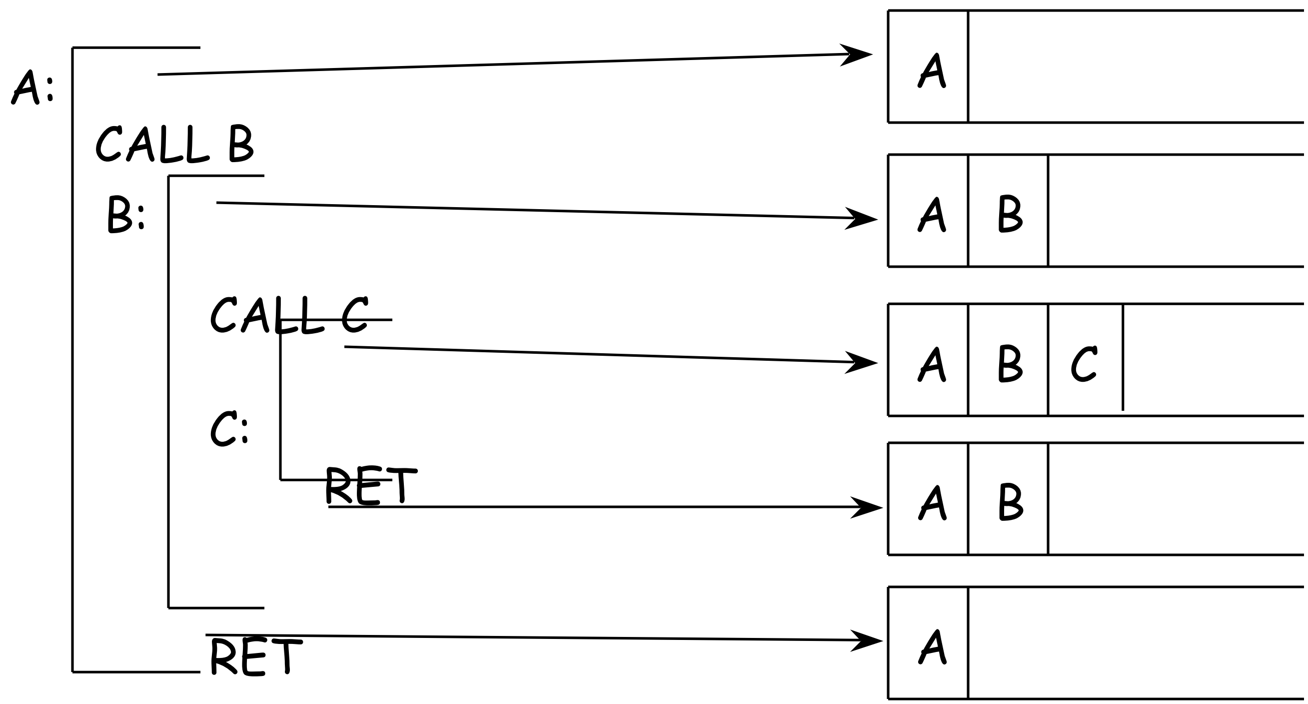
25% of integer branches are 2 to 4 instructions

At least 8 bits suggested (± 128 instructions)

Language and Compiler Driven Facilities

Calls: Why Are Stacks So Great?

Stacking of Subroutine Calls & Returns and Environments:



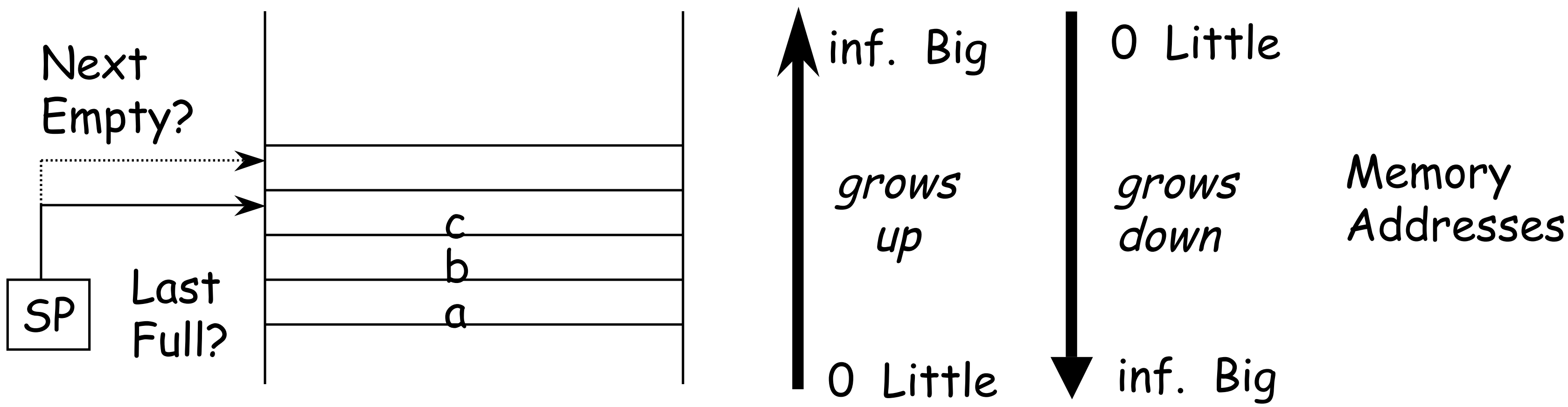
Some machines provide a memory stack as part of the architecture
(e.g., VAX)

Sometimes stacks are implemented via software convention
(e.g., MIPS)

Memory Stacks

Useful for stacked environments/subroutine call & return even if operand stack not part of architecture

Stacks that Grow Up vs. Stacks that Grow Down:



How is empty stack represented?

<u>Little --> Big/Last Full</u>

POP: Read from Mem(SP)
Decrement SP

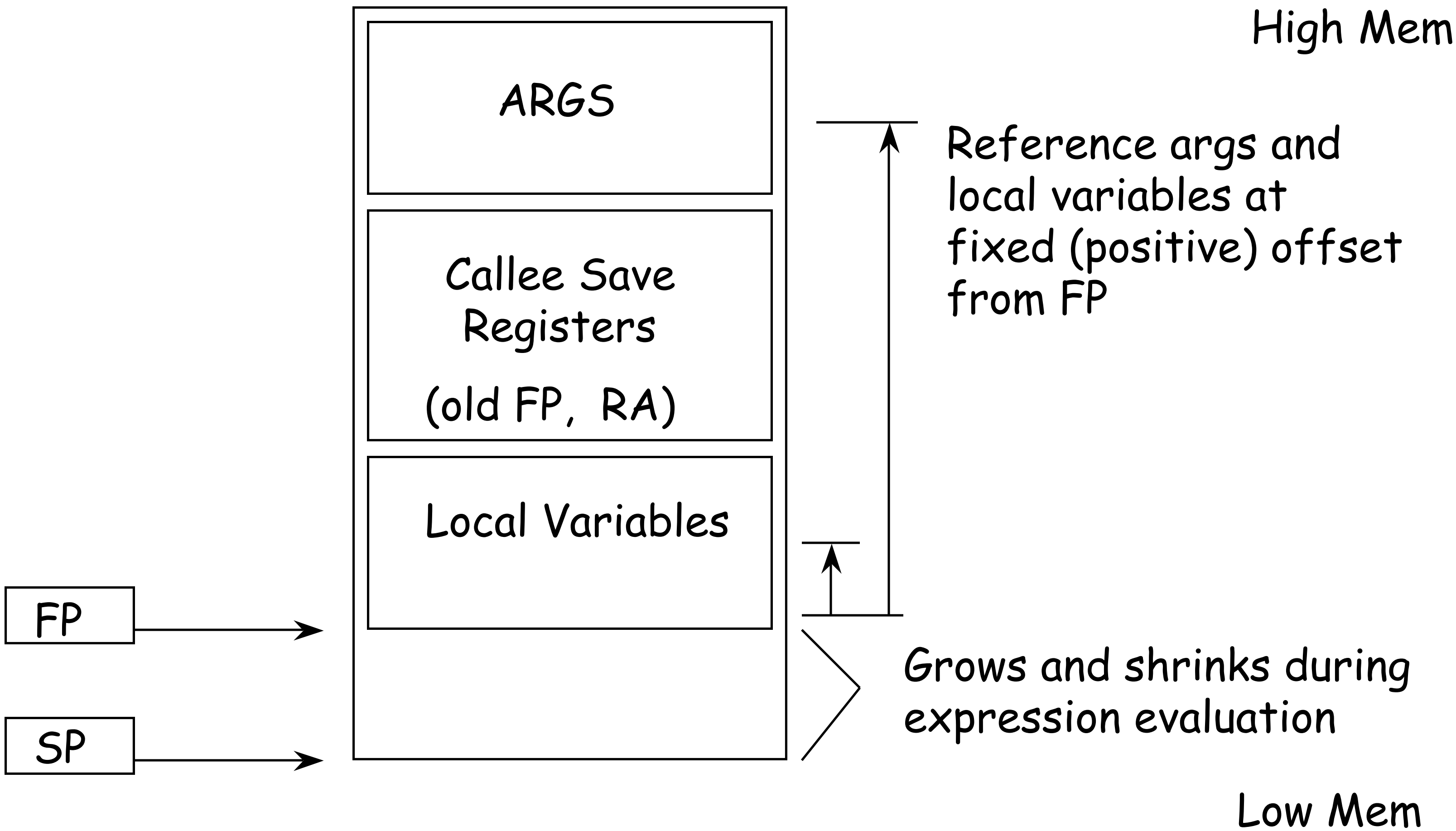
PUSH: Increment SP
Write to Mem(SP)

<u>Little --> Big/Next Empty</u>

POP: Decrement SP
Read from Mem(SP)

PUSH: Write to Mem(SP)
Increment SP

Call-Return Linkage: Stack Frames



Many variations on stacks possible (up/down, last pushed /next)

Compilers normally keep scalar variables in registers, not memory!

Compilers and Instruction Set Architectures

Ease of compilation

- Orthogonality: no special registers, few special cases, all operand modes available with any data type or instruction type
- Completeness: support for a wide range of operations and target applications
- Regularity: no overloading for the meanings of instruction fields
- Streamlined: resource needs easily determined

Register Assignment is critical too

- Easier if lots of registers

Provide at least 16 general purpose registers plus separate floating-point registers

Be sure all addressing modes apply to all data transfer instructions

Aim for a minimalist instruction set

Summary

Quick Review of Last Week

Classification of Instruction Set Architectures

Instruction Set Architecture Design Decisions

- Operands
- Operations
- Memory Addressing
- Instruction Formats

Instruction Sequencing

Language and Compiler Driven Decisions