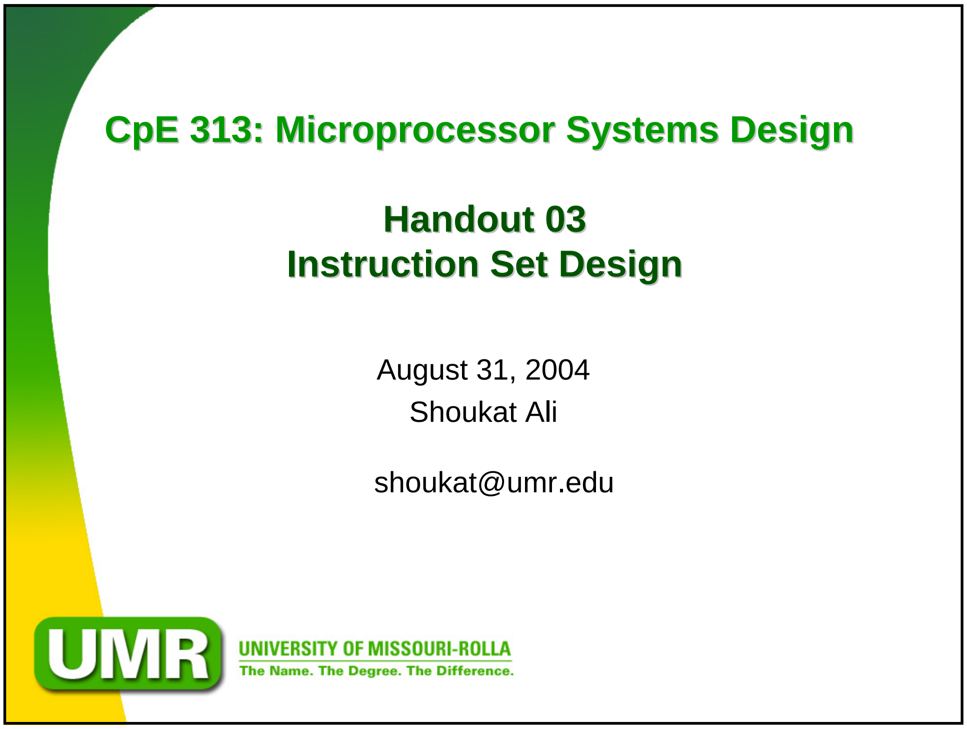


CpE 313: Microprocessor Systems Design

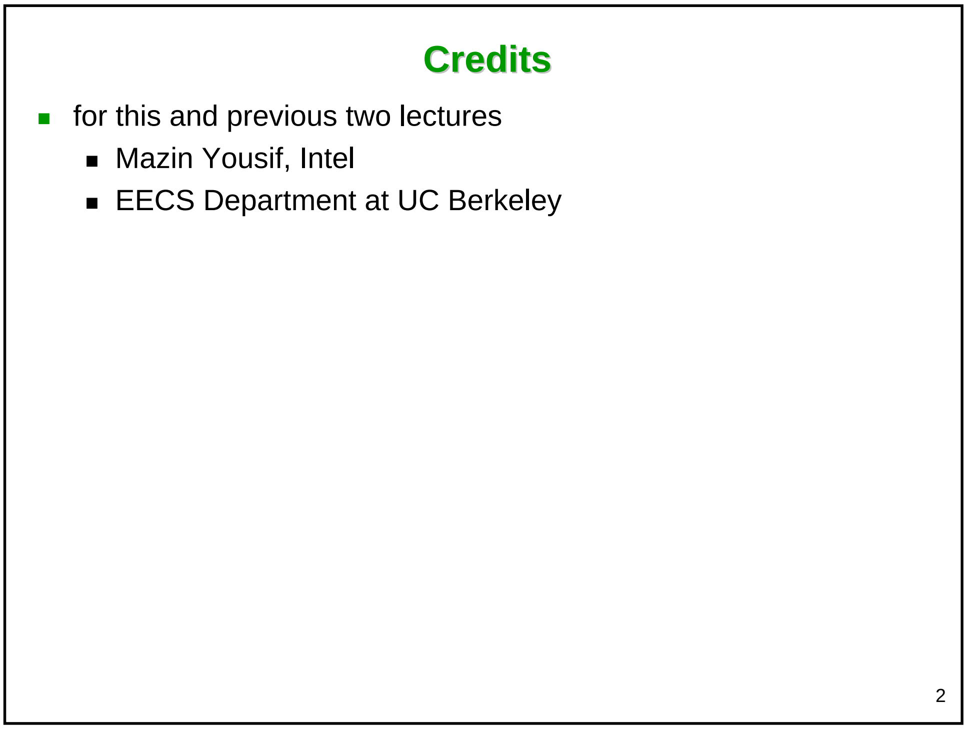
Handout 03
Instruction Set Design

August 31, 2004

Shoukat Ali

shoukat@umr.edu

UNIVERSITY OF MISSOURI-ROLLA
The Name. The Degree. The Difference.

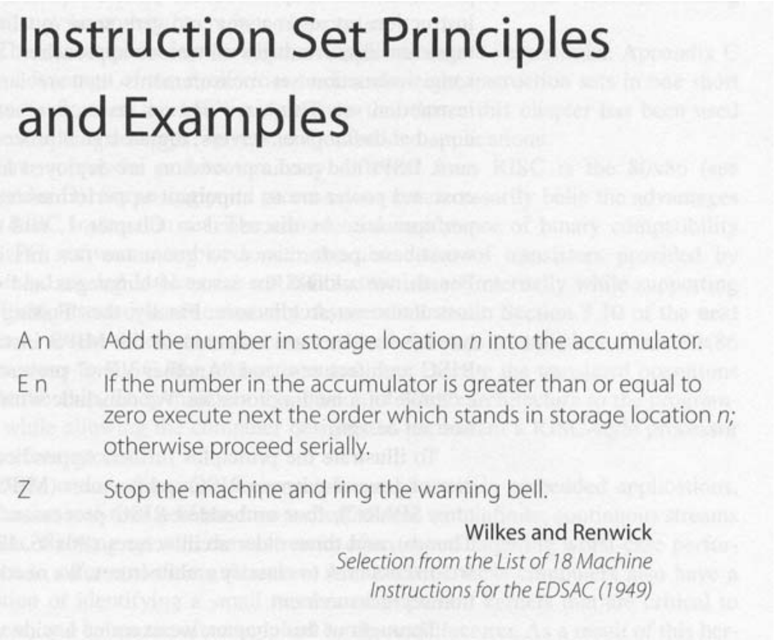


Credits

- for this and previous two lectures
  - Mazin Yousif, Intel
  - EECS Department at UC Berkeley

Issues in Chapter 2

- beginning chapter 2 today
- will learn
  - how to write a short program using stack, accumulator, and general-purpose-register machines
    - what a load-store architecture is
    - the advantages and disadvantages of these approaches
  - the instruction types, memory-addressing modes, control-flow instruction types, and operand sizes and types
  - what issues drive the choice of offset and immediate sizes in an instruction encoding
  - what an architecture should provide to help the compiler writer
  - what the different major memory segments are (stack, global-data, heap, and text)



Instruction Set Principles and Examples

A n Add the number in storage location *n* into the accumulator.

E n If the number in the accumulator is greater than or equal to zero execute next the order which stands in storage location *n;* otherwise proceed serially.

Z Stop the machine and ring the warning bell.

Wilkes and Renwick

Selection from the List of 18 Machine Instructions for the EDSAC (1949)

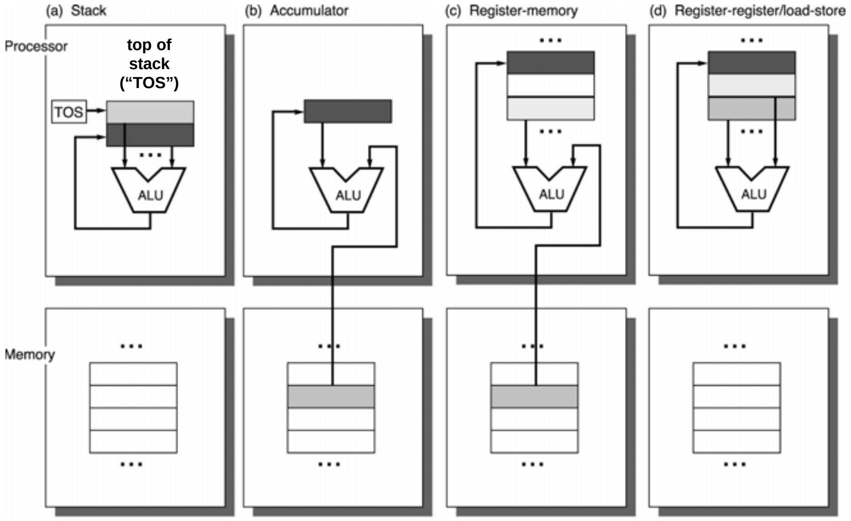
Instruction Set Architecture: What Must be Specified?

- assume you just fetched a 16-bit long ALU instruction from memory and want to execute it
  - assume instruction = 1001 0011 0101 1110
- questions you should ask:
  - about instruction format or encoding
    - how does computer decode this binary sequence?
      - which bits specify the operation to be performed?
      - which bits specify the operands?
  - location of operands and result
    - how many operands must be *explicitly* specified?
      - zero in stack architecture
      - 1 in accumulator architecture
      - 2 or 3 in general purpose register architecture
    - are operands in memory or in processor storage?
    - how does the processor locate these operands?

Classifying Instructions Sets – Location of Operands

- differentiating ISAs on the basis of location of operands of an ALU operation (add, sub, and, nand, etc)
- Stack Architectures
  - operands are at top of the stack
    - ALU operations do not have to specify operand locations
- Accumulator Architecture
  - one operand is stored in an accumulator
    - ALU ops do not have to specify this operand
  - other operand in memory
- General Purpose Register (GPR) Architecture
  - operands are in the general purpose register area or in memory
    - all operand locations have to be specified

Classifying Instructions Sets – Location of Operands



for memory-memory GPR arch, all operands are in memory

Doing C = A + B in Three Basic ISA Classes

	stack	accumulator	register-register

C = A+B	TOS ← mem(A)
	TOS ← mem(B)
	add
	mem(C) ← TOS
		ACC ← mem(A)
		add mem(B)
		mem(C) ← ACC
			R1 ← mem(A)
			R2 ← mem(B)
			add R3,R1,R2
			mem(C) ← R3

equivalent
representations

	stack	accumulator	register-register
C = A+B	push A push B add pop C	load A add B store C	load R1, A load R2, B add R3,R1,R2 store R3, C

register-register architecture also known as ***load-store architecture,*** implying that ***only load and store instructions access memory.*** Other instructions, e.g., ALU operations do NOT access memory.

A Bigger Example

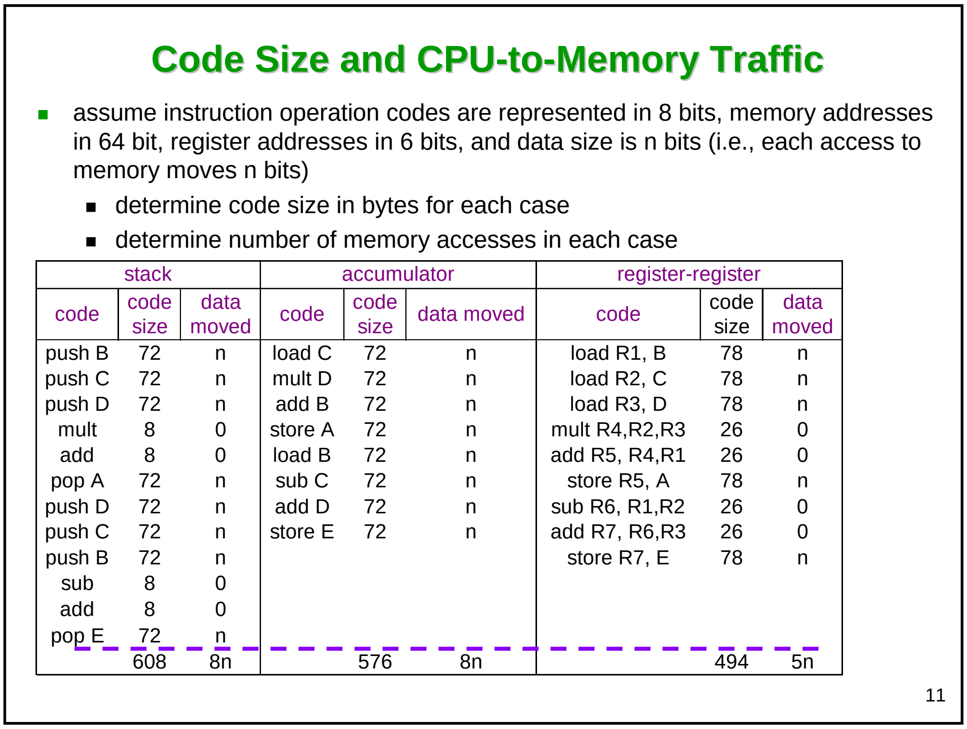
	stack	accumulator	register-register
	push B	load C	load , B
	push C	mult D	load , C
	push D	add B	load , D
	mult	store A	mult R4,R2,R3
	add	load B	add R5, R4,R1
A = B +C*D	pop A	sub C	store R5, A
E = B - C + D	push D	add D	sub R6, R1,R2
	push C	store E	add R7, R6,R3
	push B		store R7, E
	sub		
	add		
	pop E		

Question about load/store architecture: would it become harder to compile the C code if the processor had fewer registers? What if it had only 5 registers?

Register-Register Is Flexible!

stack	accumulator	register-register	register-register	register-register
push B	load C	load R1, B		
push C	mult D	load R2, C		
push D	add B	load R3, D	load R3, D	load R3, D
mult	store A	mult R4,R2,R3	mult R4,R2,R3	sub R6, R1,R2
add	load B	add R5, R4,R1	add R5, R4,R1	add R7, R6,R3
pop A	sub C	store R5, A	store R5, A	store R7, E
push D	add D	sub R6, R1,R2	sub R6, R1,R2	
push C	store E	add R7, R6,R3	add R7, R6,R3	
push B		store R7, E	store R7, E	
sub				
add				
pop E				

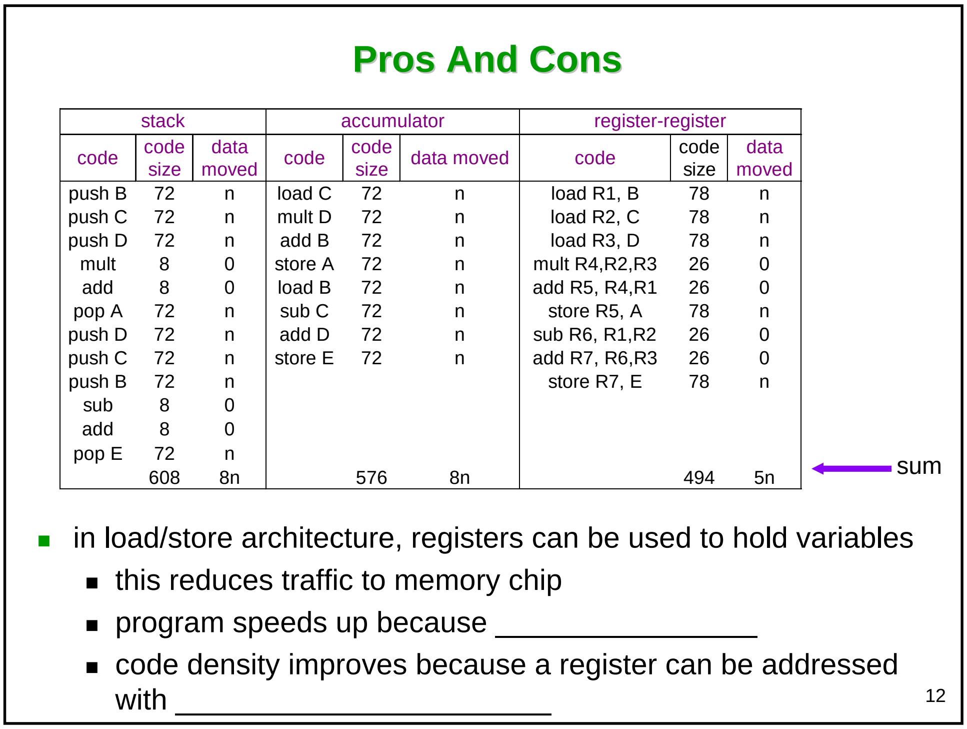
for load/store arch, some instructions can be by compiler. This is important for caches and pipelining! We will see why later.



Code Size and CPU-to-Memory Traffic

- assume instruction operation codes are represented in 8 bits, memory addresses in 64 bit, register addresses in 6 bits, and data size is n bits (i.e., each access to memory moves n bits)
  - determine code size in bytes for each case
  - determine number of memory accesses in each case

stack			accumulator			register-register		
code	code size	data moved	code	code size	data moved	code	code size	data moved
push B	72	n	load C	72	n	load R1, B	78	n
push C	72	n	mult D	72	n	load R2, C	78	n
push D	72	n	add B	72	n	load R3, D	78	n
mult	8	0	store A	72	n	mult R4,R2,R3	26	0
add	8	0	load B	72	n	add R5, R4,R1	26	0
pop A	72	n	sub C	72	n	store R5, A	78	n
push D	72	n	add D	72	n	sub R6, R1,R2	26	0
push C	72	n	store E	72	n	add R7, R6,R3	26	0
push B	72	n				store R7, E	78	n
sub	8	0						
add	8	0						
pop E	72	n						
	608	8n		576	8n		494	5n

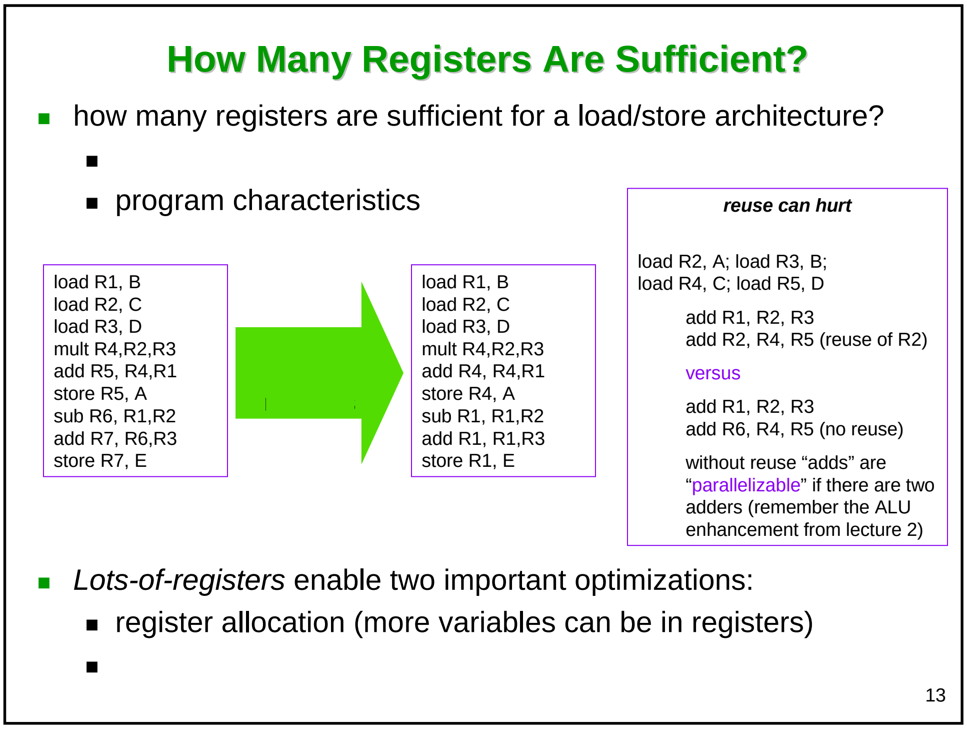


Pros And Cons

stack			accumulator			register-register		
code	code size	data moved	code	code size	data moved	code	code size	data moved
push B	72	n	load C	72	n	load R1, B	78	n
push C	72	n	mult D	72	n	load R2, C	78	n
push D	72	n	add B	72	n	load R3, D	78	n
mult	8	0	store A	72	n	mult R4,R2,R3	26	0
add	8	0	load B	72	n	add R5, R4,R1	26	0
pop A	72	n	sub C	72	n	store R5, A	78	n
push D	72	n	add D	72	n	sub R6, R1,R2	26	0
push C	72	n	store E	72	n	add R7, R6,R3	26	0
push B	72	n				store R7, E	78	n
sub	8	0						
add	8	0						
pop E	72	n						
	608	8n		576	8n		494	5n

sum

- in load/store architecture, registers can be used to hold variables
  - this reduces traffic to memory chip
  - program speeds up because ________________
  - code density improves because a register can be addressed with ________________________



How Many Registers Are Sufficient?

- how many registers are sufficient for a load/store architecture?
  -
  - program characteristics

```
load R1, B
load R2, C
load R3, D
mult R4,R2,R3
add R5, R4,R1
store R5, A
sub R6, R1,R2
add R7, R6,R3
store R7, E
```

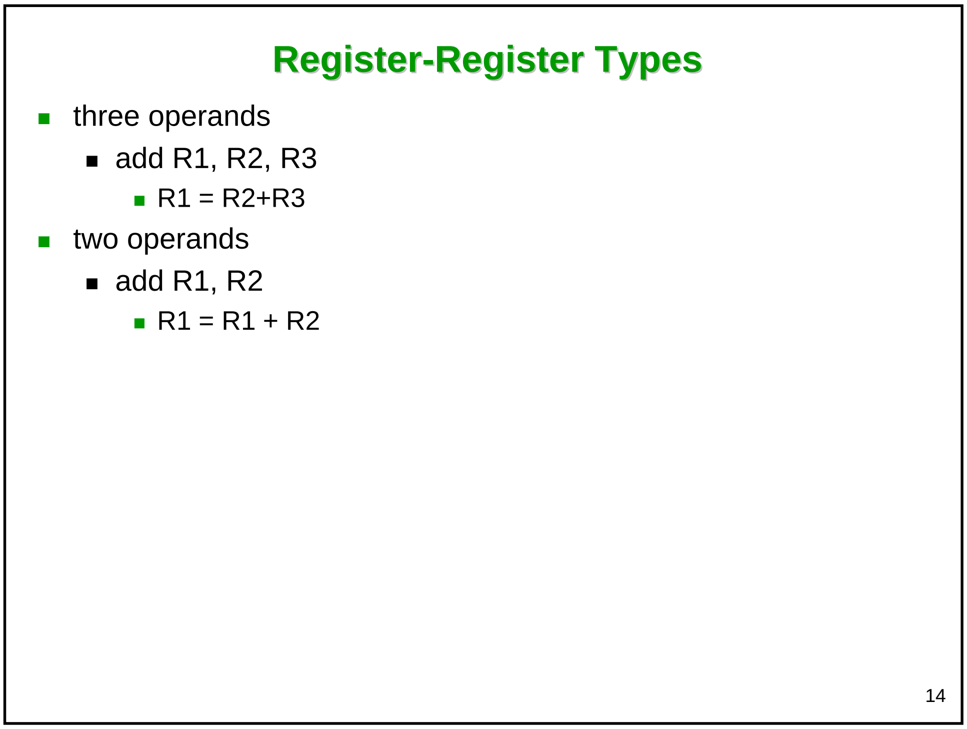
```
load R1, B
load R2, C
load R3, D
mult R4,R2,R3
add R4, R4,R1
store R4, A
sub R1, R1,R2
add R1, R1,R3
store R1, E
```

reuse can hurt

```
load R2, A; load R3, B;
load R4, C; load R5, D
        add R1, R2, R3
        add R2, R4, R5 (reuse of R2)
        versus
        add R1, R2, R3
        add R6, R4, R5 (no reuse)
```

without reuse "adds" are "parallelizable" if there are two adders (remember the ALU enhancement from lecture 2)

- *Lots-of-registers* enable two important optimizations:
  - register allocation (more variables can be in registers)
  -



Register-Register Types

- three operands
  - add R1, R2, R3
    - R1 = R2+R3
- two operands
  - add R1, R2
    - R1 = R1 + R2

Five Minute Break

Memory Addressing

- issues
  - how does the processor interpret a memory address?
  - how does an instruction specify a memory address?
    - “addressing modes”

Interpreting a Memory Address

- first issue: how does the processor interpret a memory address?
- ◦ usually a memory address is address of a byte
  - addresses 0, 1, 2, 3 point to successive bytes
  - addresses 0, 2, 4, 6 point to successive half-words
    - a half-word = 16 bits
  - addresses 0, 4, 8, 12 point to successive words
    - a word = 32 bits
- “processor wants to store a word $B_0B_1B_2B_3$ at address 4” means
  processor will store bytes at addr 4, the next byte at addr 5, …, and the fourth byte at addr 7
  - which byte at address 4?
    - long political debate has raged over this issue; see “On Holy Wars and a Plea for Peace” at http://www.networksorcery.com/enp/ien/ien137.txt

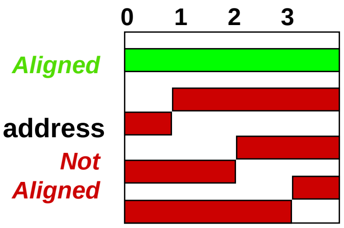
Big Endian or Little Endian

- big endian approach: store the most significant byte, B_0, first
  - B_0 at addr 4, B_1 at addr 5, …, B_3 at addr 7
    - IBM 360/370, Motorola 68k, Sparc, HP PA
- little endian approach: store the least significant byte, B_3, first
  - B_3 at addr 4, B_2 at addr 5, …, B_0 at addr 7
    - Intel 80x86, DEC VAX, DEC Alpha
- PowerPC and MIPS are bi-endian
  - can be programmed to any “end”
-
  - big endian: byte at address 4 is loaded as the most significant byte into the register
  - big endian: byte at address 4 is loaded as the least significant byte into the register

Object Alignment

- for objects that are larger than a byte in size, e.g., a word, double word, etc,
  - at which address do they start?
  - most computers require that objects start at an address which is a multiple of their size
    - e.g., word-size objects must start at addresses that are multiples of 4

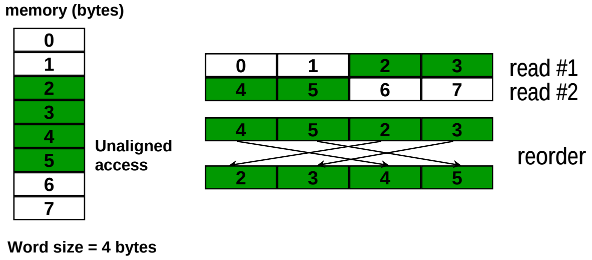
Alignment: require that objects fall on address that is multiple of their size.



Why Enforce Object Alignment?

- DRAM/SRAM is so organized that a “memory access” reads/writes starting from a multiple of 4
- so asking for a word that begins at a multiple of 4 is okay

Asking for words beginning at 0 or 4 is OK. Asking for other words requires two reads (e.g., word starting at 2)



- so enforcing object alignment
  -
  -

Addressing Modes

- what we have done so far
  - discussed stack, accumulator, and GPR architectures
  - given an address for an object (word, double word etc.), we know which bytes to access in memory and how to determine which byte is MSB
- but how do we determine an address to start with?

How To Determine an Address: Addressing Modes

These are all the addressing modes used in recent computers

Addressing mode	Example	Meaning
Register	**Add R4,R3**	**R4← R4+R3**
Immediate	**Add R4,#3**	**R4 ← R4+3**
Displacement	**Add R4,100(R1)**	**R4 ← R4+Mem[100+R1]**
Register indirect	**Add R4,(R1)**	**R4 ← R4+Mem[R1]**
Indexed / Base	**Add R3,(R1+R2)**	**R3 ← R3+Mem[R1+R2]**
Direct or absolute	**Add R1,(1001)**	**R1 ← R1+Mem[1001]**
Memory indirect	**Add R1,@(R3)**	**R1 ← R1+Mem[Mem[R3]]**
Post-increment	**Add R1,(R2)+**	**R1 ← R1+Mem[R2]; R2 ← R2+d**
Pre-decrement	**Add R1,–(R2)**	**R2 ← R2–d; R1 ← R1+Mem[R2]**
Scaled	**Add R1,100(R2)[R3]**	**R1 ← R1+Mem[100+R2+R3*d]**

Addressing Mode Usage? (ignore register mode)

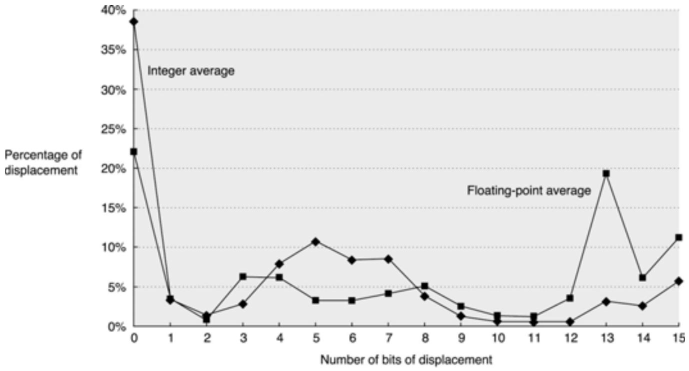
- 3 SPEC89 programs measured on a machine with all address modes (DEC VAX)
- results:

---	Displacement:	42% avg, 32% to 55%	75%	
---	Immediate:	33% avg, 17% to 43%		85%
---	Register deferred (indirect):	13% avg, 3% to 24%		
---	Scaled:	7% avg, 0% to 16%		
---	Memory indirect:	3% avg, 1% to 6%		
---	Misc:	2% avg, 0% to 3%		

75% displacement & immediate
85% displacement, immediate & register indirect

Displacement Address Mode: Size?

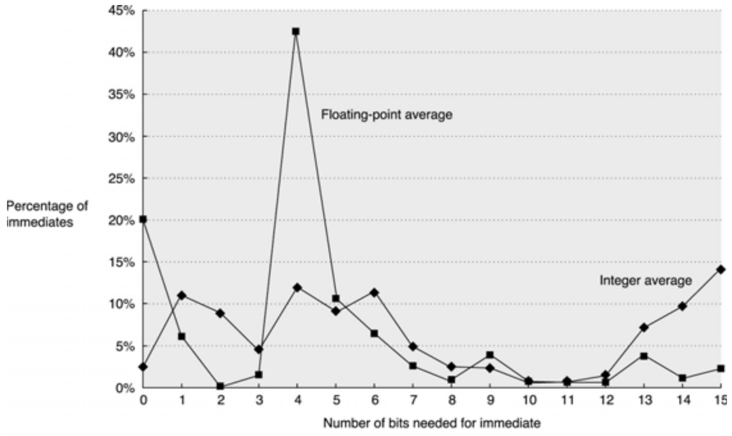
Displacement **Add R4,100(R1)** **R4 ← R4+Mem[100+R1]**



- ° **Avg. of CINT2000 programs v. avg. of CFP2000 programs on Alpha**
- ° **only 1% of addresses needed more than 16-bits**
- ° **12 - 16 bits of displacement needed**

Immediate Address Mode: Size?

Immediate **Add R4,#3** **R4 ← R4+3**



- 50% to 60% fit within 8 bits
- 75% to 80% fit within 16 bits

Addressing Mode 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

Type and Size of Operands

- what we have done so far
  - discussed stack, accumulator, and GPR architectures
  - given an address for an object (word, double word etc.), we know which bytes to access in memory and how to determine which byte is MSB
  - given an address field and an addressing mode, we know how to form the effective address
- now that we fetched the object from the memory, how do we know if we fetched an integer, a floating point number, a character, or some other data type?
  - btw, how many and what data types should a computer support?

Operand Types

<u>Bit</u>: 0, 1

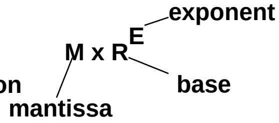
<u>Bit String</u>: sequence of bits of a particular length
4 bits is a nibble
8 bits is a byte
16 bits is a half-word
32 bits is a word
64 bits is a double-word

<u>Character</u>:
ASCII 7 bit code
UNICODE 16 bit code

<u>Decimal</u>:
digits 0-9 encoded as 0000b thru 1001b
two decimal digits packed per 8 bit byte

<u>Integers</u>:
2's Complement

<u>Floating Point</u>:
Single Precision
Double Precision
Extended Precision

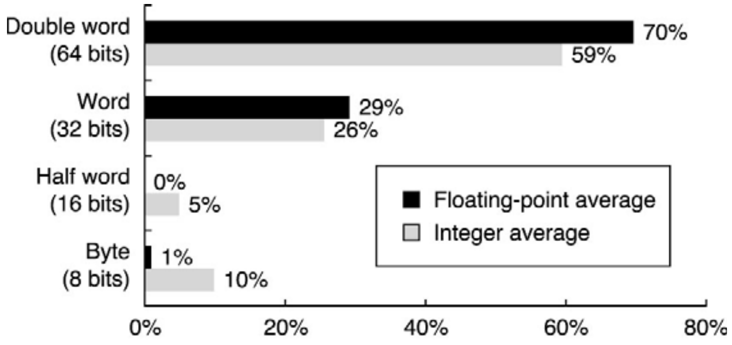


$M \times R^E$

IEEE standard 754 states how to specify a floating point number. See Appendix H for more details.

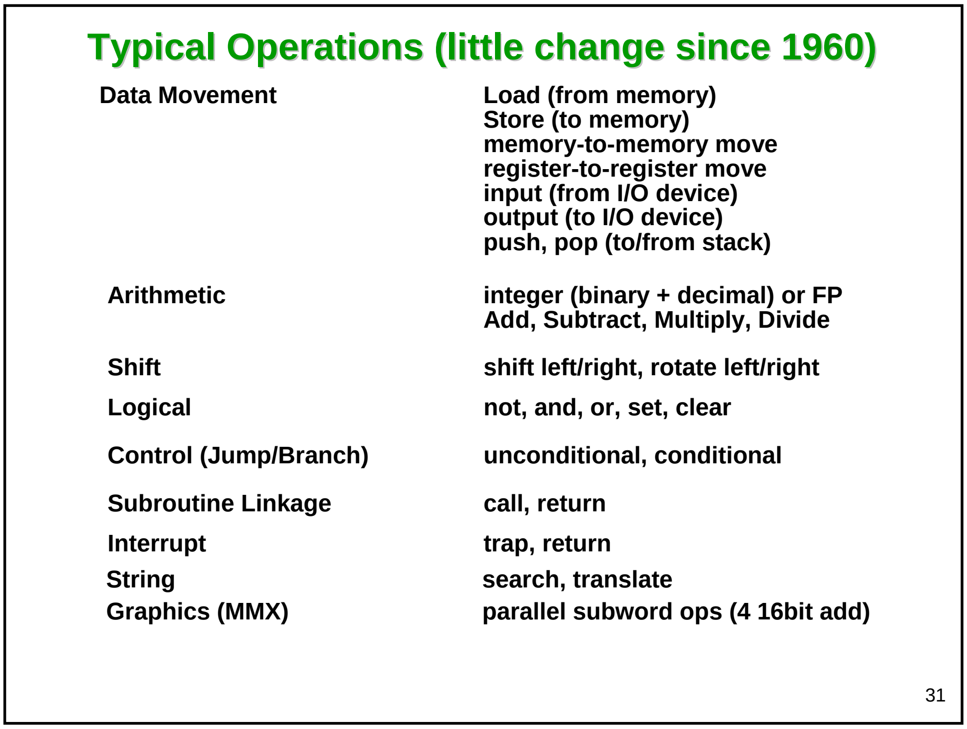
Operand Size Usage

Graph for sizes of objects referenced from memory
for SPEC CPU 2000 benchmarks



- a new 64-bit machine should support these data sizes and types:
  - 8-bit, 16-bit, 32-bit, 64-bit integers and
  - 32-bit and 64-bit IEEE 754 floating point numbers

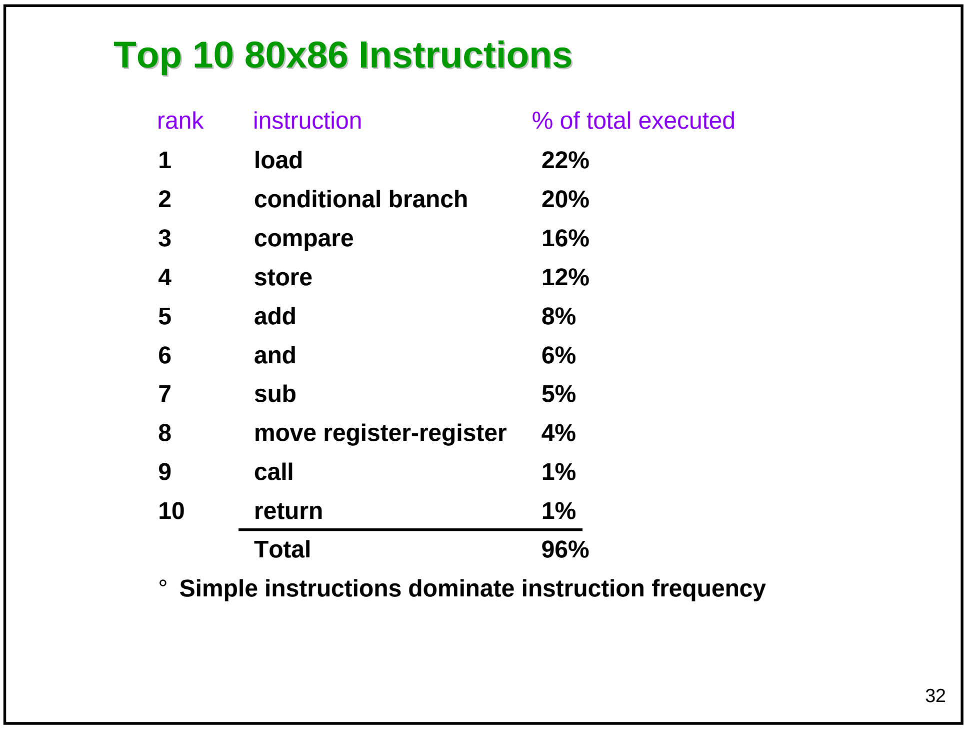
What Kind of Operations Should a New Computer Support?



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
String	search, translate
Graphics (MMX)	parallel subword ops (4 16bit add)

31

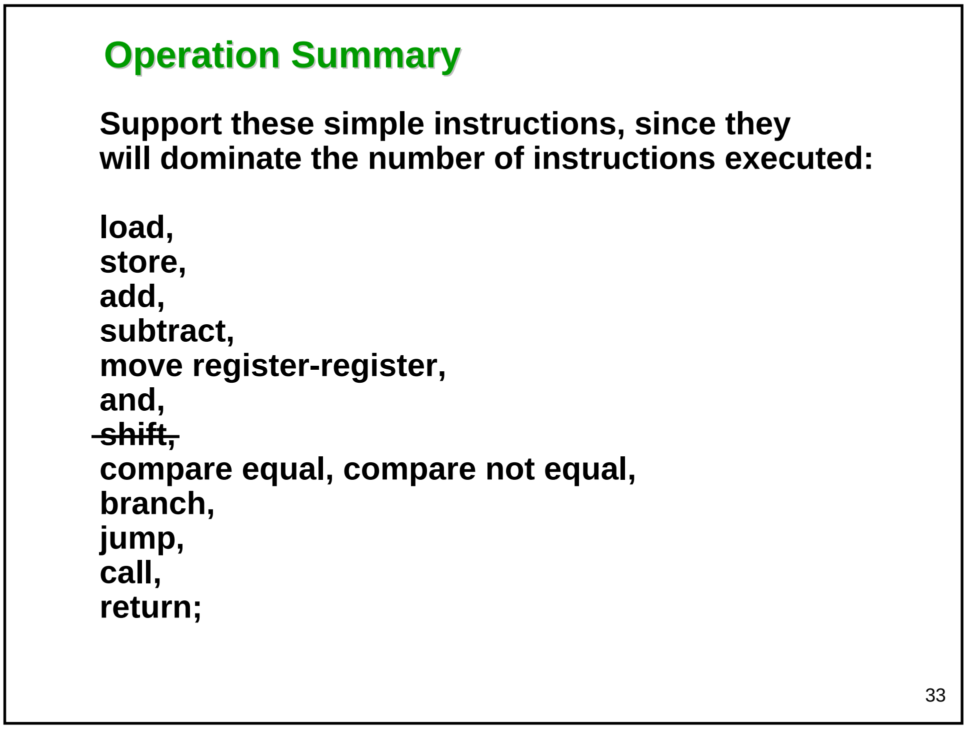


Top 10 80x86 Instructions

rank	instruction	% of 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

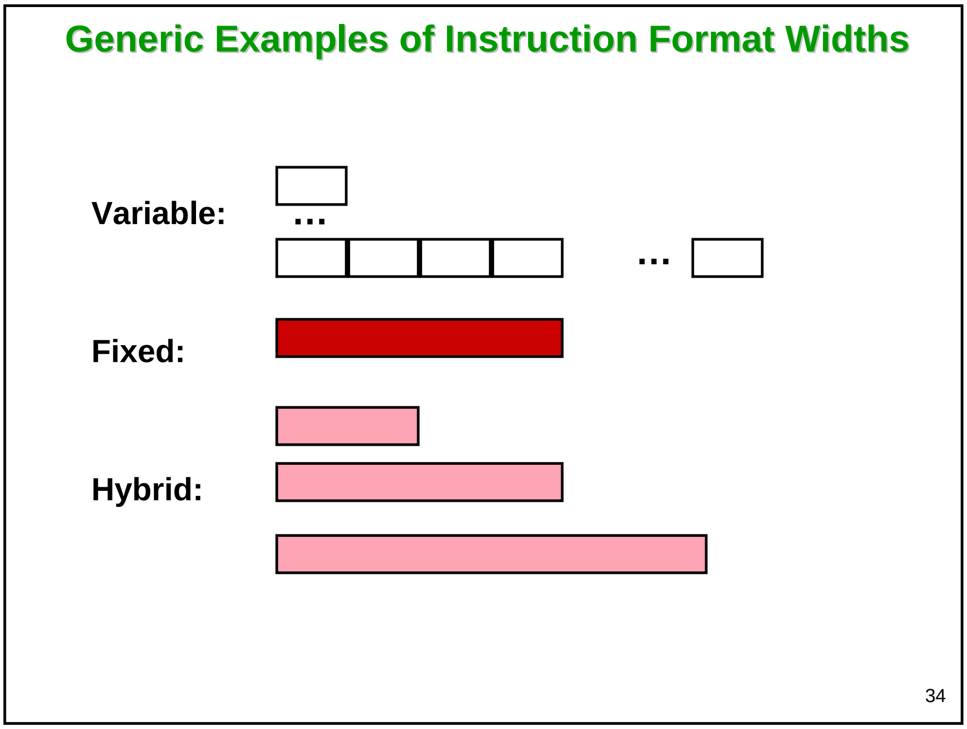
32



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;

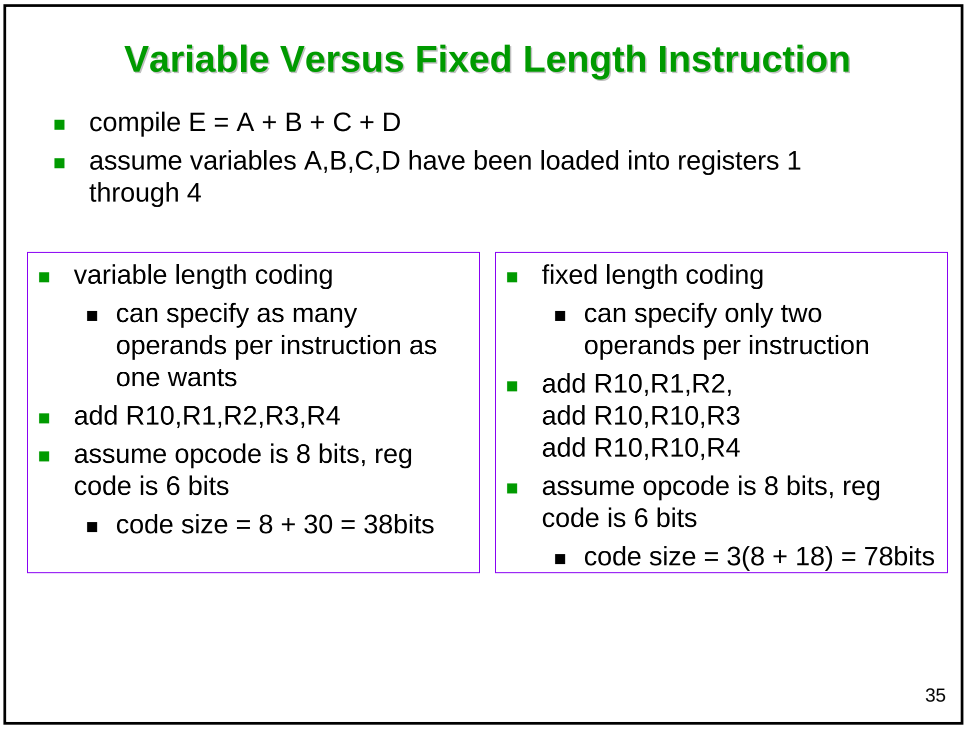


Generic Examples of Instruction Format Widths

Variable:

Fixed:

Hybrid:

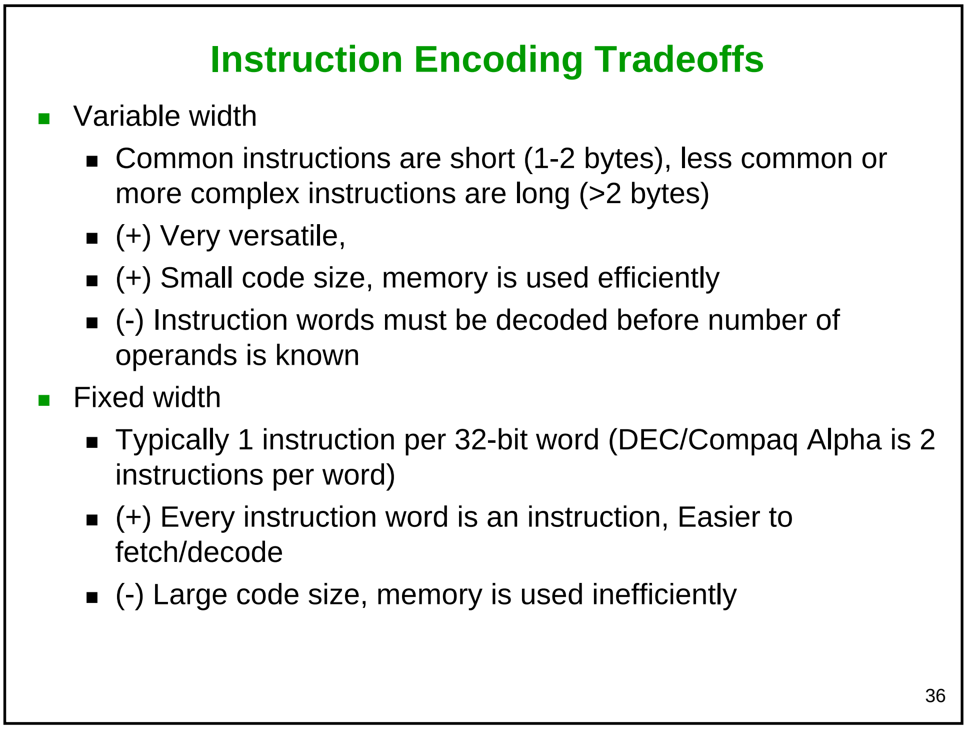


Variable Versus Fixed Length Instruction

- compile E = A + B + C + D
- assume variables A,B,C,D have been loaded into registers 1 through 4

- variable length coding
  - can specify as many operands per instruction as one wants
- add R10,R1,R2,R3,R4
- assume opcode is 8 bits, reg code is 6 bits
  - code size = 8 + 30 = 38bits

- fixed length coding
  - can specify only two operands per instruction
- add R10,R1,R2,
  add R10,R10,R3
  add R10,R10,R4
- assume opcode is 8 bits, reg code is 6 bits
  - code size = 3(8 + 18) = 78bits



Instruction Encoding Tradeoffs

- Variable width
  - Common instructions are short (1-2 bytes), less common or more complex instructions are long (>2 bytes)
  - (+) Very versatile,
  - (+) Small code size, memory is used efficiently
  - (-) Instruction words must be decoded before number of operands is known
- Fixed width
  - Typically 1 instruction per 32-bit word (DEC/Compaq Alpha is 2 instructions per word)
  - (+) Every instruction word is an instruction, Easier to fetch/decode
  - (-) Large code size, memory is used inefficiently

Instruction Formats

- If code size is more important, use variable length instructions
- If performance is more 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

Instruction Mode Encoding

- Two possibilities for specifiying addressing mode
  - Each operand has a “mode” field
    - Also called “address specifiers”
    - VAX, Motorola 68000
    - (+) Very versatile
    - (-) Encourages variable-width instructions (hard decode)
  - Opcode specifies addressing mode
    - Most RISCs (will see soon)
    - (+) Encourages fixed-width instructions (easy decode)
    - (+) “Natural” for a load/store ISA
    - (-) Limits what every instruction can do

Reduced Instruction Set Computing (RISC)

“The simplicity of all *instructions allows a quick and simple evaluation of status to begin execution… Notice that this applies to* all *instructions. Adding complications to a special operation, therefore, degrades all the others.”*

-- J. E. Thornton, at the introduction of the CDC 6600, 1963

“*I understand that in the laboratory developing this system there were only 34 people, including the janitor. .. Contrasting this modest effort with our own vast development activities, I fail to understand why we have lost our industry leadership by letting someone else offer the world’s most powerful computer”*

-- Thomas Watson, IBM president, in memo following 6600 announcement.

RISC Versus CISC

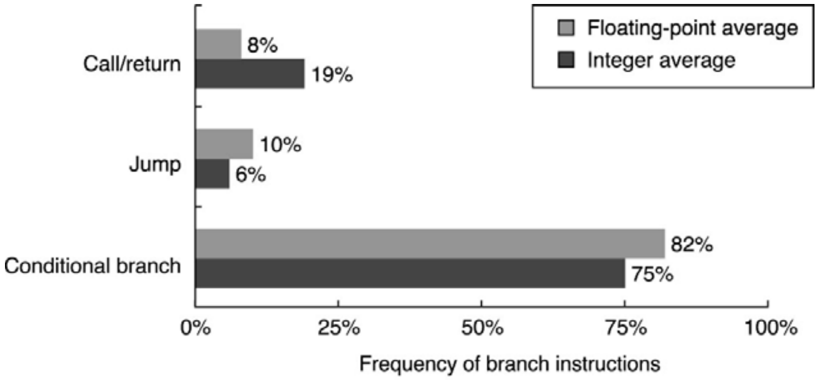
- small number of instruction sizes
  - preferably 1, e.g., MIPS
  - 6600 had two
- only load/stores access memory
  - e.g. add instr cannot

- variable instruction size
  - 80x86 instruction could be anywhere from 1 to 17 bytes
- any instruction could access memory
  - add mem[A] mem[B] R2

Control Instructions

- instructions that change the flow of a program by explicitly changing the program counter
- types
  - conditional branches
    - if a certain condition is true, the program branches to an instruction called the “branch target” and proceeds from thereon
  - jump
    - program unconditionally jumps to an instruction called the “jump target” and proceeds from thereon
  - procedure call
    - program unconditionally jumps to another instruction, executes a set of instructions called a “procedure,” and then returns to the original program to resume the original flow
  - procedure return
    - the instruction that returns flow of control from a procedure to the original program

Conditional Branches Dominant



How to Specify Target Address

- displacement from current PC to target specified in the instruction
  - called “PC-relative” branches
    - requires fewer bits
      - experiments suggest 8 bits would work
    - gives the code “position independence”

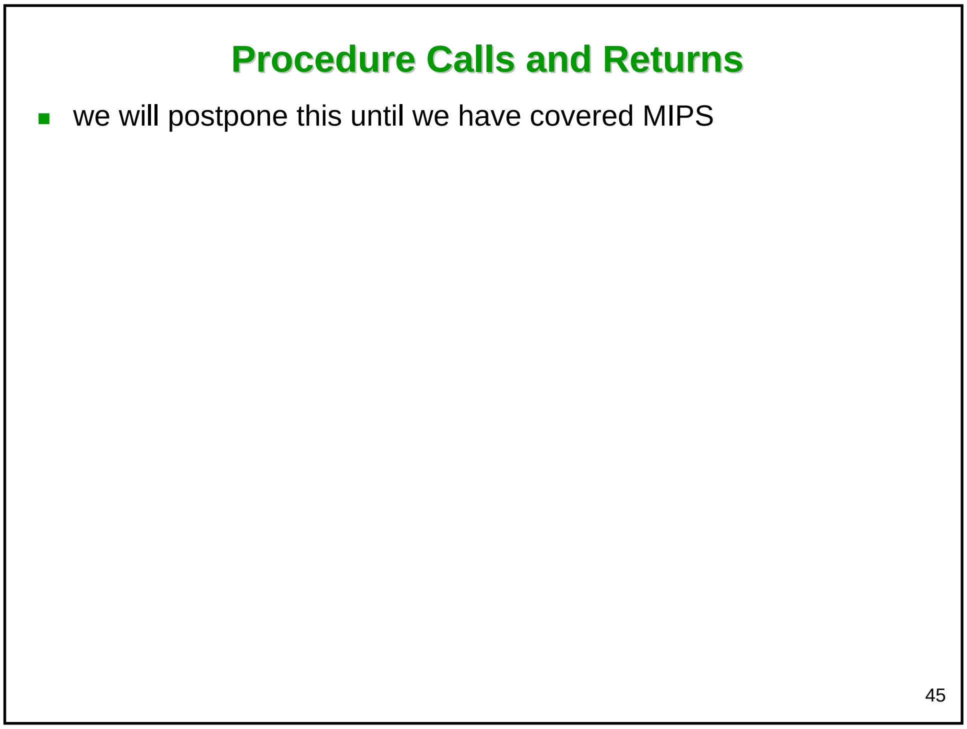
- target specified in a register
  - example: jump (R1), i.e., jump to the address contained in R1
  - very useful if the target address is not known at compile-time as in a case statement

Conditional Branch: When to Compare

- possibility 1: combine compare *calculation* and branch instructions
  - PC = PC + displacement if R2 > R3

- possibility 2: separate compare *calculation* and branch instructions
  - R1 = R2 – R3
    PC = PC + displacement if R1 > 0

  - a variation on possibility 2: don’t use up a general purpose register, instead provide and use a special register called CC (CC is like Program Status Word in 8051)
    - R1 = R2 – R3 (this op implicitly sets CC.NegFlag)
      PC = PC + displacement if CC.NegFlag == 0



Procedure Calls and Returns

- we will postpone this until we have covered MIPS

45