

# ECE 486/586

# Computer Architecture

Prof. Mark G. Faust

Maseeh College of Engineering  
and Computer Science



# Instruction Set Principles and Examples

- Reading:
  - Hennessy & Patterson: Appendix B
  - RISC-I paper (Patterson & Sequin)
- Homework:

# How do computers work?

- Need to understand abstractions such as:
  - Applications software
  - Systems software
  - Assembly Language
  - Machine Language
  - Architectural Issues: i.e., Caches, Virtual Memory, Pipelining
  - Sequential logic, finite state machines
  - Combinational logic, arithmetic circuits
  - Boolean logic, 1s and 0s
  - Transistors used to build logic gates (CMOS)
  - Semiconductors/Silicon used to build transistors
  - Properties of atoms, electrons, and quantum dynamics

# Instruction Set Architecture

- Instruction Set Architecture (ISA)
  - Traditional meaning of “computer architecture”
  - What the programmer/compiler writer “sees”
  - Independent of organization and implementation
    - ISA doesn't include caches and pipelines
  - Instructions, Operands, Addressing Modes

# ISA: Instruction Set Architecture

- **Compiler**
  - Input: High level language
  - Output: Assembly language for target ISA
  - Global, local optimization
  - Register allocation
- **Assembler**
  - Input: Assembly language
  - Output: Machine code (“object file”)
- **Linker**
  - Input(s): Object files, library files
  - Output: Executable program
  - (dynamically linked libraries)
- **Loader**
  - Reads executable from disk
  - Passes command line arguments
  - Optionally “fixes” absolute addresses

High-level  
language  
program  
(in C)

```
swap(int v[], int k)
{int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

Compiler

Assembly  
language  
program  
(for MIPS)

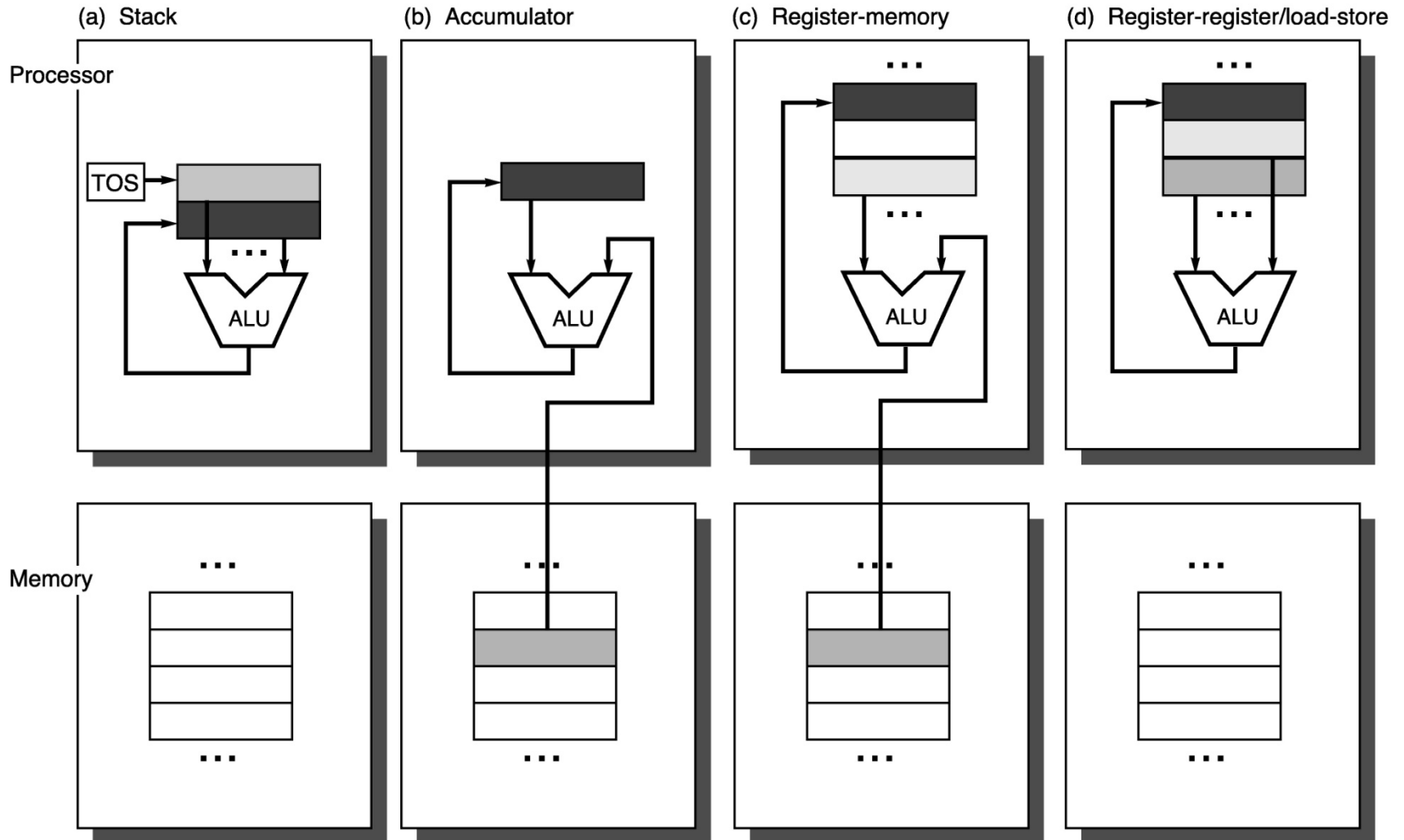
```
swap:
    muli $2, $5, 4
    add  $2, $4, $2
    lw   $15, 0($2)
    lw   $16, 4($2)
    sw   $16, 0($2)
    sw   $15, 4($2)
    jr   $31
```

Assembler

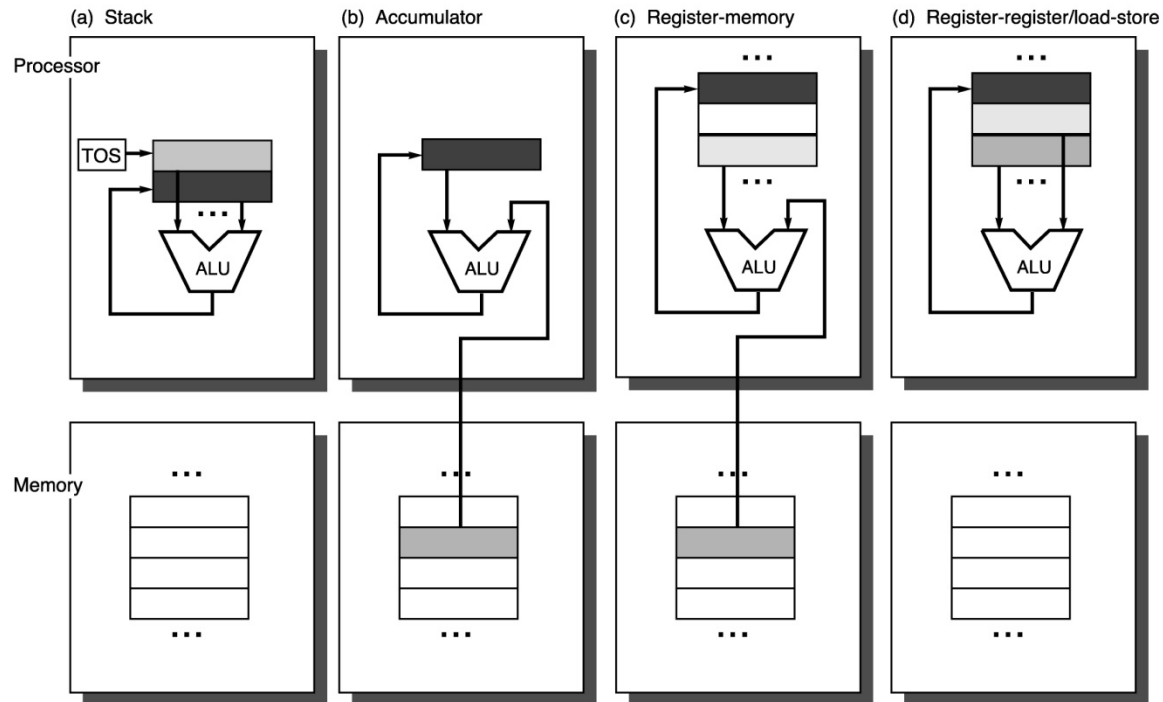
Binary machine  
language  
program  
(for MIPS)

```
000000001010000100000000000011000
000000000000110000001100000100001
100011000110001000000000000000000
100011001111001000000000000000100
101011001111001000000000000000000
101011000110001000000000000000100
00000011111000000000000000001000
```

# A Taxonomy (Classification)



# A Taxonomy (Classification)



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

A,B,C – memory references

R1,R2,R3 – registers

# Examples of each ISA

- Stack
  - HP calculator, Postscript
  - Pentium FP (x87 co-processor)
    - 8 registers organized as stack
- Accumulator
  - PDP-8
  - 8051 (MCU)
- Load/Store (Register/Register)
  - RISC: MIPS, Alpha, ARM, PowerPC, SPARC
  - Itanium
- Register/Memory
  - IA-32 (Intel X86), Motorola 68000, IBM 360
  - PDP-11
  - VAX (really Memory/Memory)



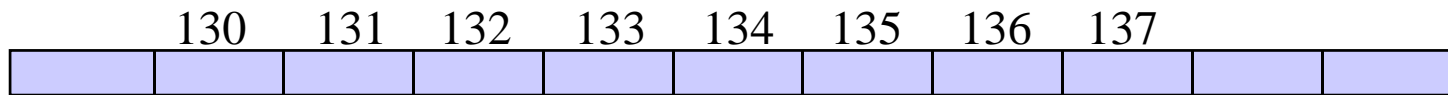
# Register and Memory Operands

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Register-register	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, Trimedia TM5200
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

Type	Advantages	Disadvantages
Register-register (0, 3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute (see App. A).	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density leads to larger programs.
Register-memory (1, 2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density.	Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location.
Memory-memory (2, 2) or (3, 3)	Most compact. Doesn't waste registers for temporaries.	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)

# Memory Addressing

- Byte Addressing



- Word Addressing
  - 16-bit half word (Intel: word)
  - 32-bit word (Intel: doubleword, dword)
  - 64-bit double word (Intel: quadword, qword)

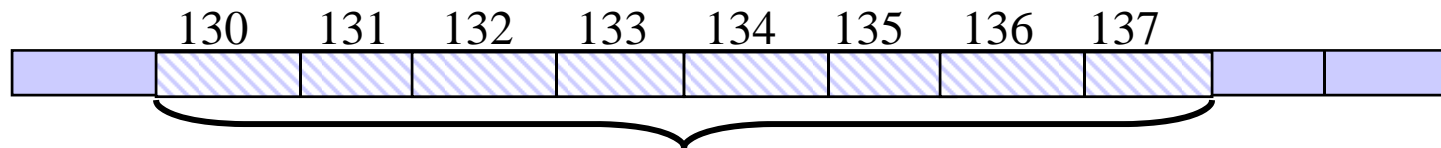
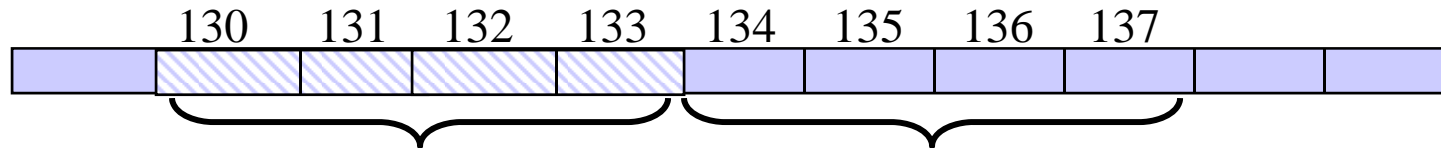
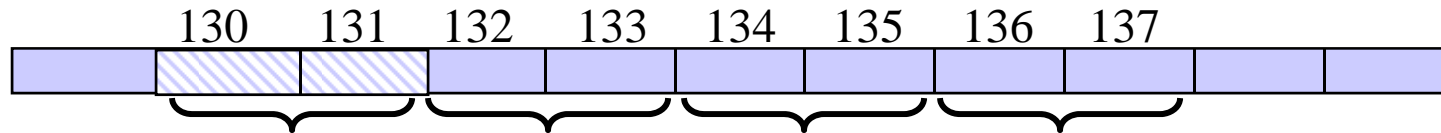
# What's in name?

Bytes	Intel IA-32	SUN SPARC
2	Word	Halfword
4	Doubleword (dword)	Word
8	Quadword (qword)	Doubleword
16	-----	Quadword



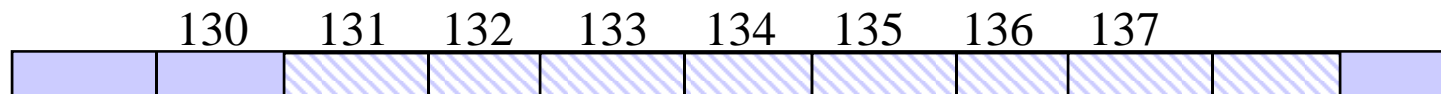
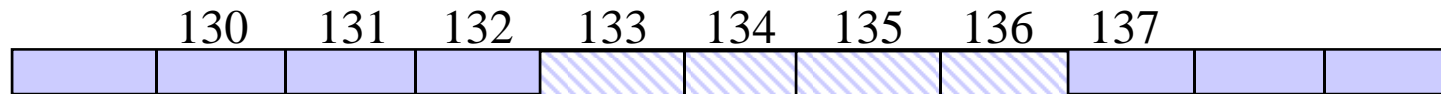
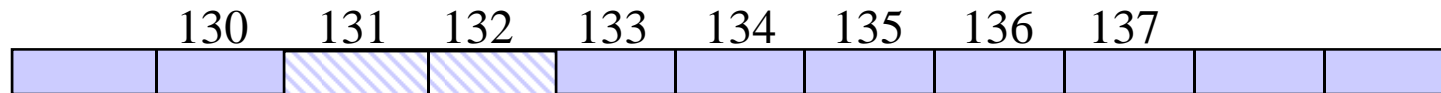
# Memory Addressing

- Alignment
  - Must words, dword, qwords begin on mod 2, mod 4, mod 8 boundaries?



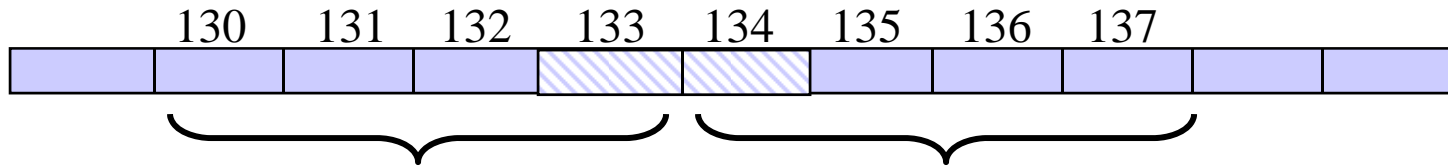
# Memory Addressing

- Alignment
  - Or are there no alignment restrictions?



# Alignment: Why Care?

- Non-aligned memory references may cause multiple memory accesses



- Consider a system in which memory reads return 4 bytes and a reference to a word spans a 4-byte boundary: two memory accesses are required
- Complicates memory and cache controller design
- Even in systems with no alignment restrictions, assemblers typically have directives to force alignment for efficiency

# Alignment

```
#define NEMPLOYEES 10000
```

```
struct employee {  
    char department;  
    float salary;  
    int ID;  
    .  
    .  
    .  
} employee_table[NEMPLOYEES];
```

aligned

char			
float			
int			

non-aligned

char	float byte1	float byte2	float byte3
float byte4	int byte1	int byte2	

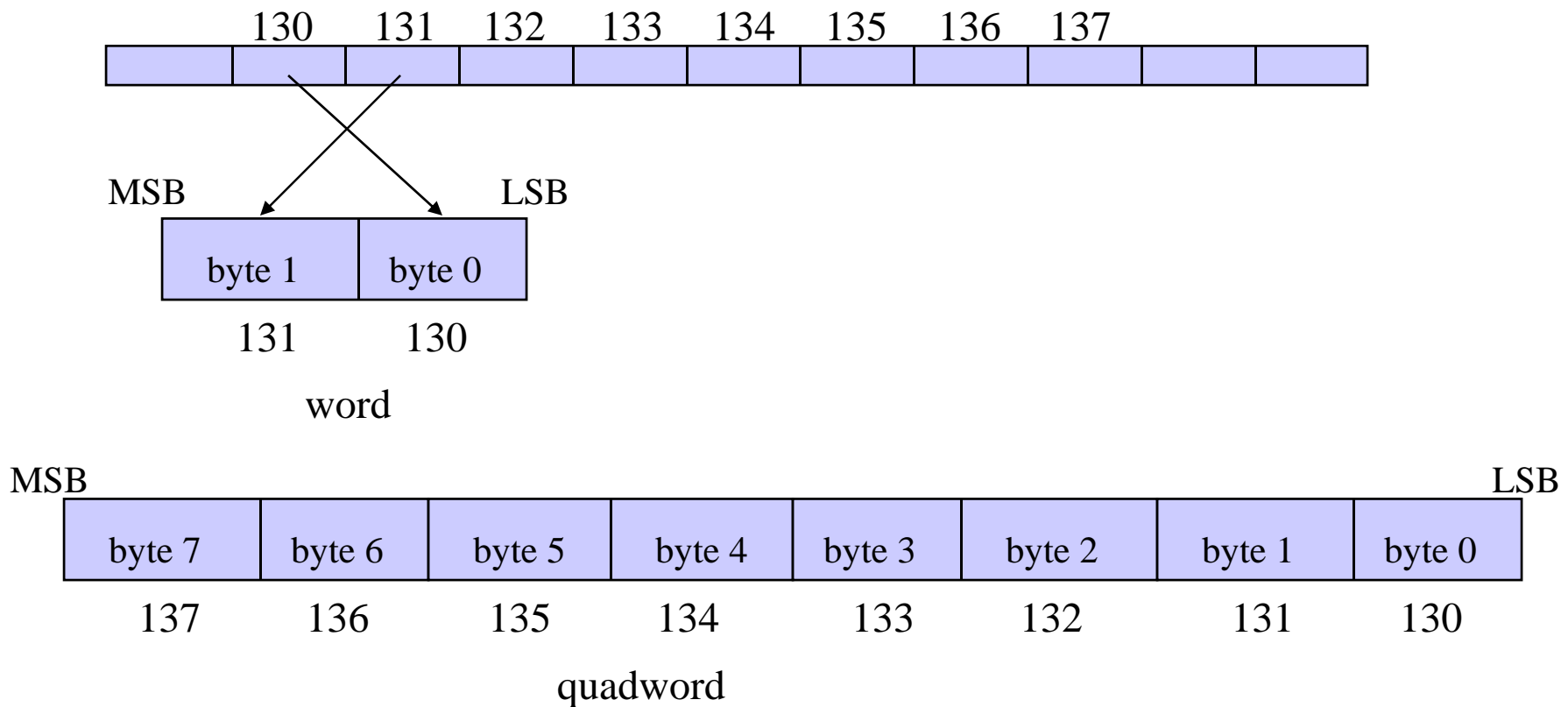


# Memory Alignment

Value of 3 low-order bits of byte address								
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						
8 bytes (double word)		Misaligned						

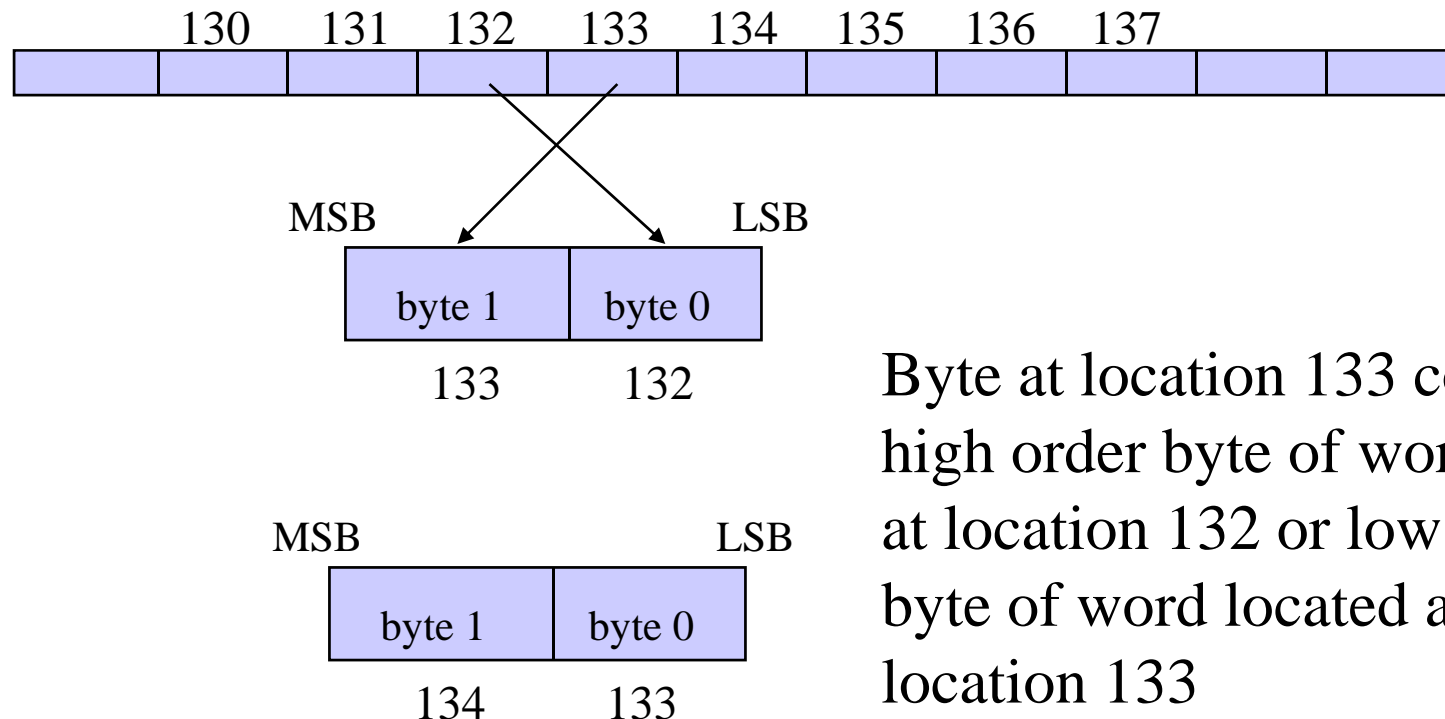
# Memory Addressing

- Byte order: Little Endian



# Memory Addressing

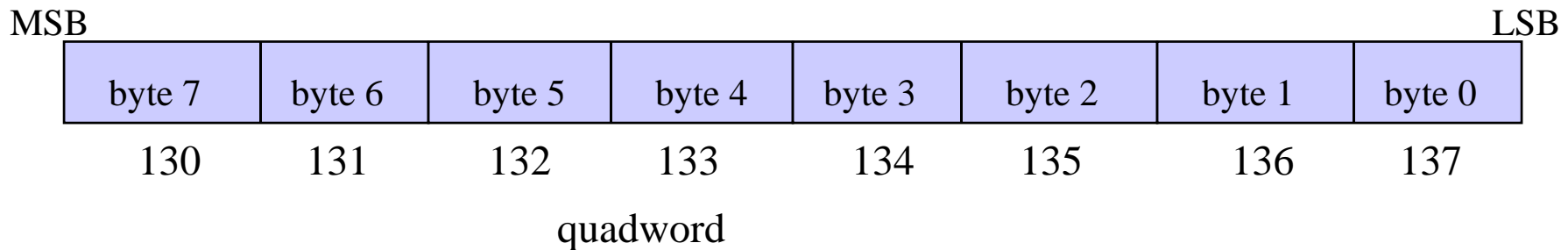
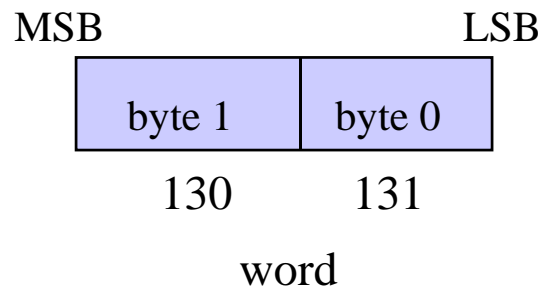
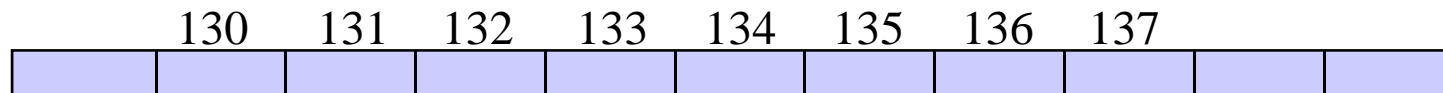
- Little Endian with no Alignment Restrictions



Byte at location 133 could be high order byte of word located at location 132 or low order byte of word located at location 133

# Memory Addressing

- Byte order: Big Endian



# Actual Systems

- Pros/Cons
  - Often exaggerated
  - Little Endian
    - Character strings will appear “backwards” in registers
    - Intuitive when incrementing from least to most significant bytes
    - Dumps and assembly listings possibly confusing
  - Big Endian: Motorola 680x0, Sun Sparc, PDP-11
  - Little Endian: VAX, Intel IA32
  - Configurable: MIPS, ARM

Opcode Address

FE	1300	LDX 1300
7E	1416	JMP 1416EA

Motorola 680x0 Program

Opcode Address

A1	0013	MOV AX,1300
1614	1214	JMP 1416,1412

Intel IA32 Program

# Addressing Modes

Addressing mode	Example instruction	Meaning	When used
Register	Add R4,R3	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Regs}[R3]$	When a value is in a register.
Immediate	Add R4,#3	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + 3$	For constants.
Displacement	Add R4,100(R1)	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Mem}[100 + \text{Regs}[R1]]$	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	Add R4,(R1)	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Mem}[\text{Regs}[R1]]$	Accessing using a pointer or a computed address.
Indexed	Add R3,(R1+R2)	$\text{Regs}[R3] \leftarrow \text{Regs}[R3] + \text{Mem}[\text{Regs}[R1] + \text{Regs}[R2]]$	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	Add R1,(1001)	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[1001]$	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	Add R1,@(R3)	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Mem}[\text{Regs}[R3]]]$	If R3 is the address of a pointer $p$ , then mode yields $*p$ .
Autoincrement	Add R1,(R2)+	$\begin{aligned} \text{Regs}[R1] &\leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]] \\ \text{Regs}[R2] &\leftarrow \text{Regs}[R2] + d \end{aligned}$	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)	$\begin{aligned} \text{Regs}[R2] &\leftarrow \text{Regs}[R2] - d \\ \text{Regs}[R1] &\leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]] \end{aligned}$	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	Add R1,100(R2)[R3]	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[100 + \text{Regs}[R2] + \text{Regs}[R3] * d]$	Used to index arrays. May be applied to any indexed addressing mode in some computers.

# Addressing Modes

- Addressing modes can reduce instruction count (code size) but at cost of added CPU design complexity
- Example: What if auto increment mode weren't available?

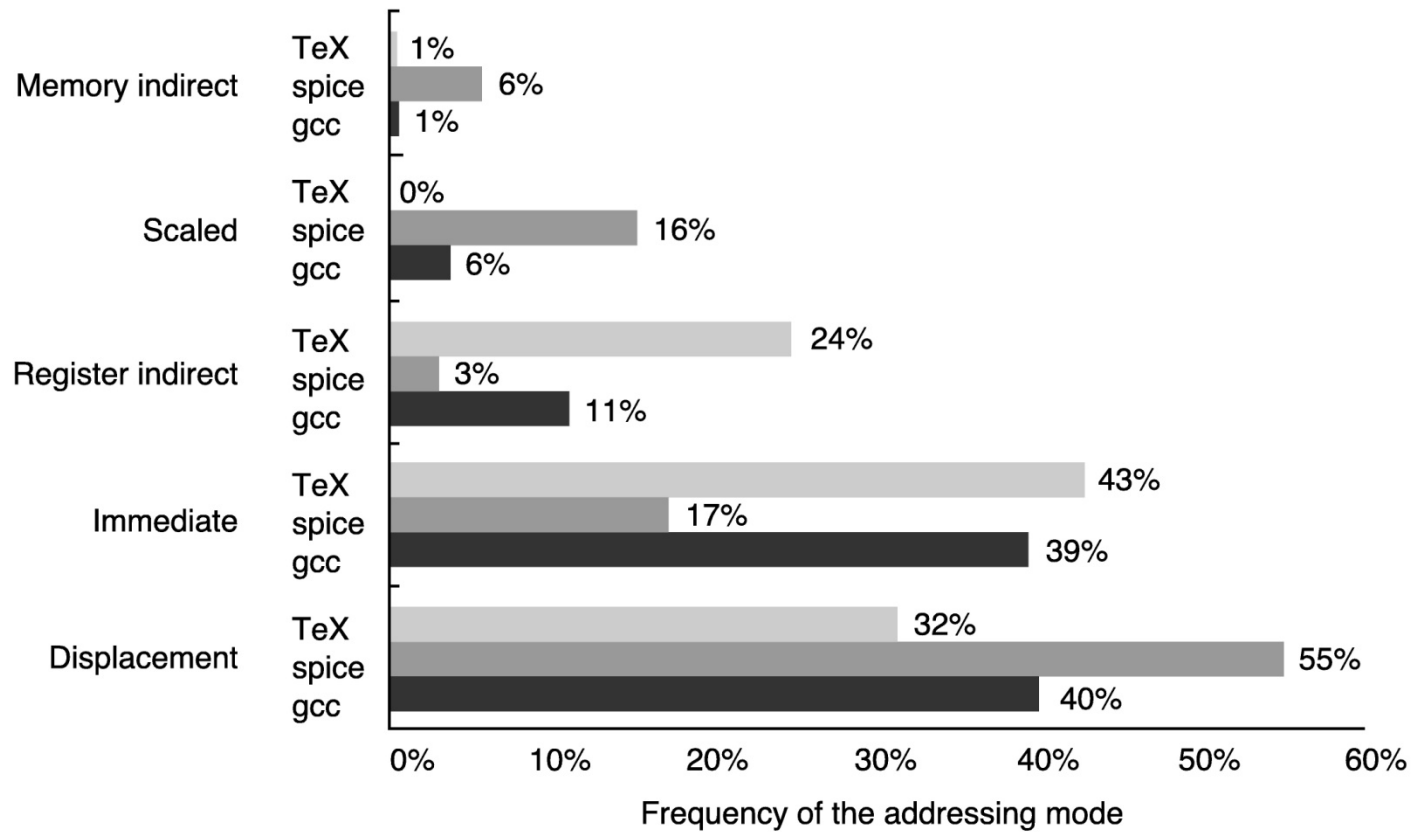
```
Add $R1, ($R2)+      # Regs[R1]=Regs[R1]+Mem[Regs[R2]]
                      # Regs[R2]=Regs[R2] + d;
    or
Add $R1, ($R2)
Add $R2, #d
```

- Example: What if displacement mode weren't available?

```
Add $R4, 100($R1)     Regs[R4]=Regs[R4]+Mem[100+Regs[R1]]
    or
Add $R1, #100
Add $R4, ($R1)
```

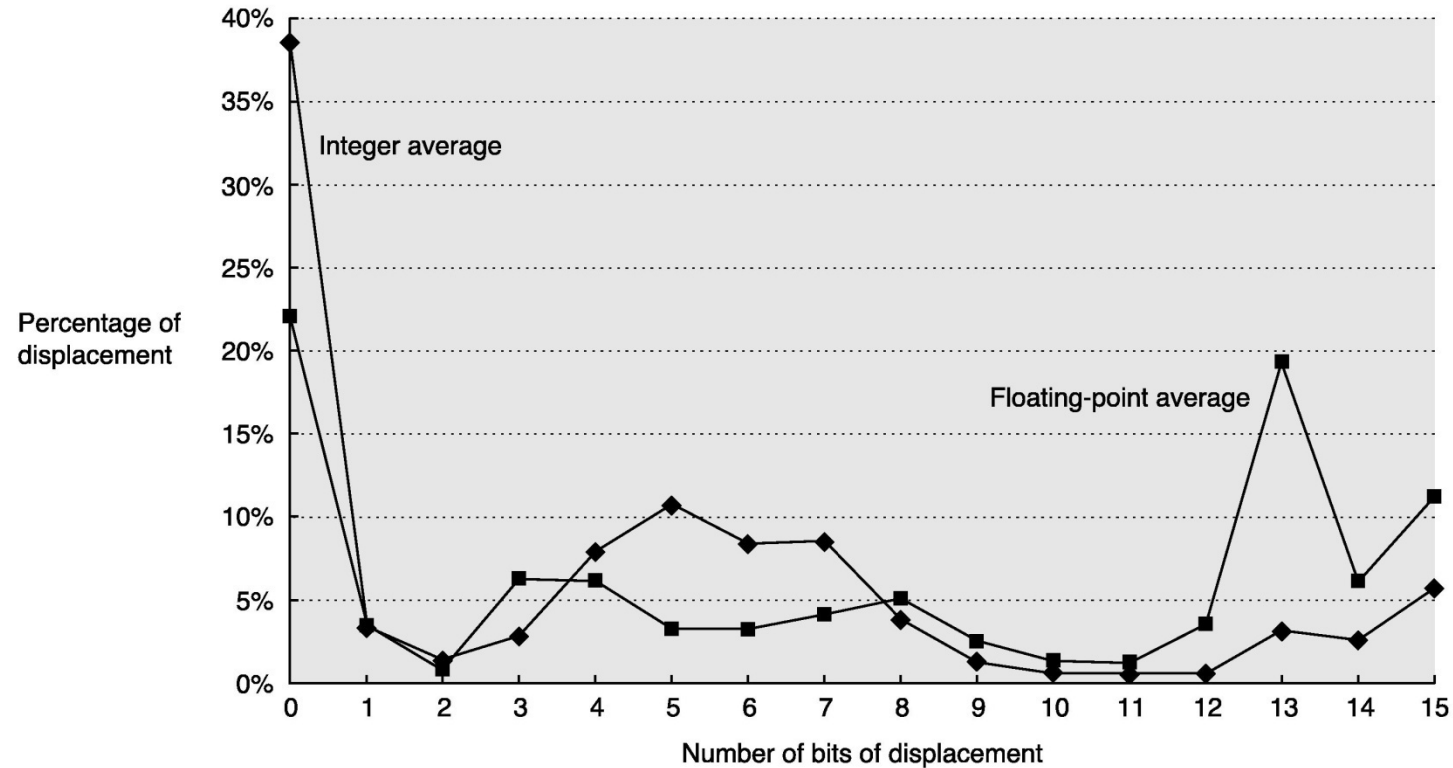
- Support most frequently used addressing modes (make the common case fast)

# Frequency of Addressing Modes

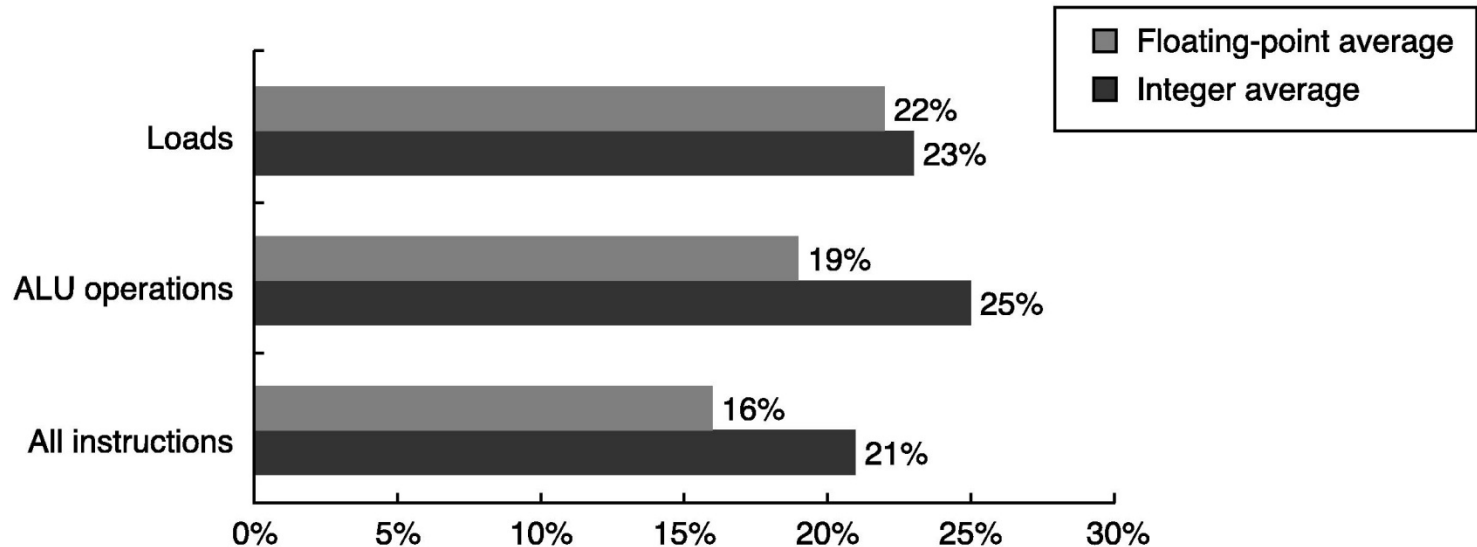




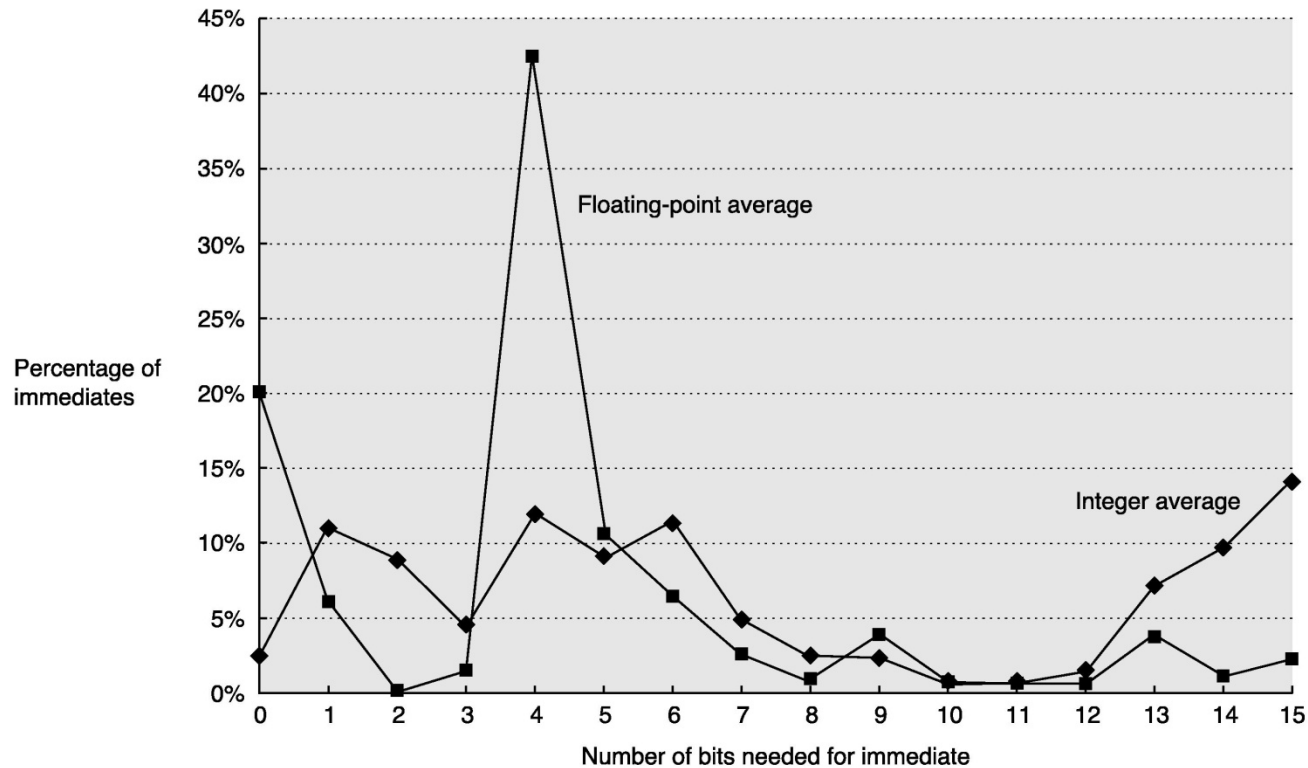
# Displacement values are widely distributed



# Immediate Operand Frequency



# Distribution of Immediate Values



# Type and Size of Operands

- Opcode specifies types of operands
  - MIPS load {LD, LW, LB, LBU, LH, L.S, L.D}
  - May not be different assembler name for opcode (Intel)
    - ADD AL, <immediate value> ; byte operands
    - ADD AX, <word address> ; 16-bit operands

Type	Size (bits)	Encoding
Character	8	ASCII
Character	16	Unicode
Integer	16, 32, 64	unsigned or twos complement
Floating point	32, 64	IEEE 754
Decimal	4	BCD
Strings	variable	Null-terminated

# Operations in the Instruction Set

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiple, 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

# Most Frequent 80x86 Instructions

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%

**Figure 2.16** The top 10 instructions for the 80x86. Simple instructions dominate this list and are responsible for 96% of the instructions executed. These percentages are the average of the five SPECint92 programs.

# Specialized Instructions and Operands

- Floating point
- String instructions
  - Operations
    - Copy, Compare
- Graphics/Multimedia
  - Data Types
    - Pixel
    - Frame Buffer
    - Vertex
  - Operations
    - BitBlt (“bit blit” – bit block transfer)
    - RasterOp
    - Intel: MMX (Multimedia Extension), SSE (SIMD Extension: 128-bit vectors)

# Special Purpose Operations

- General Purpose Processors
  - Exploiting existing hardware for special data types
  - Operands may not require full precision or data path width
    - Graphics vertex coordinates, audio samples
  - Perform operations in parallel
    - Internal 64-bit data path permits 4 16-bit quantities to be moved
    - “Partitioned Add”
      - 64-bit ALU permits 4 parallel operations on 16-bit data
      - Must break carry look-ahead chain
  - SIMD
    - “Single Instruction Multiple Data”



# RISC Multimedia Support for Desktops

Instruction category	Alpha MAX	HP PA-RISC MAX2	Intel Pentium MMX	PowerPC AltiVec	SPARC VIS
Add/subtract		4H	8B, 4H, 2W	16B, 8H, 4W	4H, 2W
Saturating add/sub		4H	8B, 4H	16B, 8H, 4W	
Multiply			4H	16B, 8H	
Compare	8B ( $\geq$ )		8B, 4H, 2W ( $=$ , $>$ )	16B, 8H, 4W ( $=$ , $>$ , $\geq$ , $<$ , $\leq$ )	4H, 2W ( $=$ , $\text{not} =$ , $>$ , $\leq$ )
Shift right/left		4H	4H, 2W	16B, 8H, 4W	
Shift right arithmetic		4H		16B, 8H, 4W	
Multiply and add				8H	
Shift and add (saturating)		4H			
And/or/xor	8B, 4H, 2W	8B, 4H, 2W	8B, 4H, 2W	16B, 8H, 4W	8B, 4H, 2W
Absolute difference	8B			16B, 8H, 4W	8B
Maximum/minimum	8B, 4W			16B, 8H, 4W	
Pack ( $2n$ bits $\rightarrow n$ bits)	2W $\rightarrow$ 2B, 4H $\rightarrow$ 4B	2*4H $\rightarrow$ 8B	4H $\rightarrow$ 4B, 2W $\rightarrow$ 2H	4W $\rightarrow$ 4B, 8H $\rightarrow$ 8B	2W $\rightarrow$ 2H, 2W $\rightarrow$ 2B, 4H $\rightarrow$ 4B
Unpack/merge	2B $\rightarrow$ 2W, 4B $\rightarrow$ 4H		2B $\rightarrow$ 2W, 4B $\rightarrow$ 4H	4B $\rightarrow$ 4W, 8B $\rightarrow$ 8H	4B $\rightarrow$ 4H, 2*4B $\rightarrow$ 8B
Permute/shuffle		4H		16B, 8H, 4W	

# Special Purpose Operations/Modes

- DSP
  - Multiply accumulate (MAC) instructions
    - $A = A + B * C$
    - Central operation in dot product
  - Saturating arithmetic
    - Real-time requirement precludes interrupt/trap for overflow detection
    - Replace with largest magnitude number
    - Accumulators much larger than operands
  - Special addressing modes
    - Circular addresses
    - Bit-reverse addressing

# DSP Addressing Modes

- Circular Addressing (Modulo Addressing)
  - Commonly used in DSP algorithms
    - filter convolution, buffer fill/re-fill
  - Index “wraps around” after auto-increment if end of array reached
- Bit Reverse Addressing
  - Common transformation in FFT algorithm
    - 0 (000)  $\rightarrow$  0 (000)
    - 1 (001)  $\rightarrow$  4 (100)
    - 2 (010)  $\rightarrow$  2 (010)
    - 3 (011)  $\rightarrow$  6 (110)
    - 4 (100)  $\rightarrow$  1 (001)
    - 5 (101)  $\rightarrow$  5 (101)
    - 6 (110)  $\rightarrow$  3 (011)
    - 7 (111)  $\rightarrow$  7 (111)

Very difficult to support in compiler

# Frequency of DSP Addressing Modes

Addressing mode	Assembly symbol	Percent
Immediate	#num	30.02%
Displacement	ARx(num)	10.82%
Register indirect	*ARx	17.42%
Direct	num	11.99%
Autoincrement, preincrement (increment register <i>before</i> using contents as address)	*+ARx	0
Autoincrement, postincrement (increment register <i>after</i> using contents as address)	*ARx+	18.84%
Autoincrement, preincrement with 16b immediate	*+ARx(num)	0.77%
Autoincrement, preincrement, with circular addressing	*+ARx%	0.08%
Autoincrement, postincrement with 16b immediate, with circular addressing	*ARx+(num)%	0
Autoincrement, postincrement by contents of AR0	*ARx+0	1.54%
Autoincrement, postincrement by contents of AR0, with circular addressing	*ARx+0%	2.15%
Autoincrement, postincrement by contents of AR0, with bit reverse addressing	*ARx+0B	0
Autodecrement, postdecrement (decrement register <i>after</i> using contents as address)	*ARx-	6.08%
Autodecrement, postdecrement, with circular addressing	*ARx-%	0.04%
Autodecrement, postdecrement by contents of AR0	*ARx-0	0.16%
Autodecrement, postdecrement by contents of AR0, with circular addressing	*ARx-0%	0.08%
Autodecrement, postdecrement by contents of AR0, with bit reverse addressing	*ARx-0B	0
<b>Total</b>		<b>100.00%</b>

**Figure 2.11 Frequency of addressing modes for TI TMS320C54x DSP.** The C54x has 17 data addressing modes, not counting register access, but the four found in MIPS account for 70% of the modes. Autoincrement and autodecrement, found in some RISC architectures, account for another 25% of the usage. These data were collected from a measurement of static instructions for the C-callable library of 54 DSP routines coded in assembly language. See [www.ti.com/sc/docs/products/dsp/c5000/c54x/54dsplib.htm](http://www.ti.com/sc/docs/products/dsp/c5000/c54x/54dsplib.htm).

# Encoding Instructions

- OpCode – Operation Code
  - The instruction (e.g. “add”, “load”)
  - Possibly variants (e.g. “load byte”, “load word”...)
- Source and Destination
  - Register or memory address
- Addressing Modes
  - Along with number of registers, impacts code size
  - Encode as part of opcode
  - Address specifier

# Encoding Instructions

Operation and no. of operands	Address specifier 1	Address field 1	...	Address specifier	Address field
----------------------------------	------------------------	--------------------	-----	----------------------	------------------

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

Operation	Address field 1	Address field 2	Address field 3
-----------	--------------------	--------------------	--------------------

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

Operation	Address specifier	Address field
-----------	----------------------	------------------

Operation	Address specifier 1	Address specifier 2	Address field
-----------	------------------------	------------------------	------------------

Operation	Address specifier	Address field 1	Address field 2
-----------	----------------------	--------------------	--------------------

(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)

# Fixed vs. Variable Length Encoding

- Fixed Length
  - Simple, easily decoded
  - Larger code size
- Variable Length
  - More complex
  - More compact, efficient use of memory
    - Fewer memory references
    - Advantage possibly mitigated by RISC use of cache
  - Complicated pipeline
- Pentium 4 Example
  - Instructions decoded to sequence of fixed length opcodes
  - Trace cache for instruction cache

# Structure of Compilers

## Dependencies

Language dependent;  
machine independent

Somewhat language dependent;  
largely machine independent

Small language dependencies;  
machine dependencies slight  
(e.g., register counts/types)

Highly machine dependent;  
language independent

Front end per  
language

*Intermediate  
representation*

High-level  
optimizations

Global  
optimizer

Code generator

## Function

Transform language to  
common intermediate form

For example, loop  
transformations and  
procedure inlining  
(also called  
procedure integration)

Including global and local  
optimizations + register  
allocation

Detailed instruction selection  
and machine-dependent  
optimizations; may include  
or be followed by assembler



# Compiler Optimizations

- High-level optimizations done on source
  - Procedure in-lining
    - Replace “expensive” procedure call with in-line code
- Local optimizations within a “basic block”
  - Common sub-expression elimination (CSE)
    - Don’t re-evaluate the same sub-expression; save result
  - Constant propagation
    - Replace all instances of variable containing a constant with the constant
    - Reduces memory accesses (e.g. immediate mode)

# Compiler Optimizations

- Global optimizations done across branches
  - Copy propagation
    - Replace all instances of variable assignments  $A = X$  with  $X$
  - Code motion
    - Remove invariant code from loop
- Register allocation
  - Allocate registers to
    - Evaluating expression
    - Passing parameters
    - Returning results
    - Storing variables
    - Stack, frame pointers
    - Return address

# Compiler Optimizations

- Processor dependent optimizations
  - Strength reduction
    - Replace/choose instructions with less “expensive” alternatives
      - Example:  $\times 2$  and  $/2 \rightarrow$  shift
  - Pipeline scheduling
    - Re-order instructions to improve pipeline performance

# How the Computer Architect Can Help the Compiler Writer

- Provide regularity
  - Make operators, data types, addressing modes “orthogonal” (independent)
- Provide primitives, not solutions
  - William Wulf, “semantic gap” and “semantic clash”
  - “...by giving too much semantic content to the instruction, the computer designer made it possible to use the instruction only in limited contexts.”
- Simplify trade-offs among alternatives
  - Make it easy for compiler writer to determine most effective implementation for a particular operation or sequence

# RISC vs. CISC Debate

- CISC (VAX)
  - Attempt to create instructions to support
    - High level languages
      - Procedure call
      - String processing
      - Loops, array accesses
    - Operating Systems
- High Level Language Computer Architecture
- RISC (IBM 801, later RISC I, II, MIPS)
  - Use simpler architecture
    - Simpler implementation → faster clock cycle
    - Make common case fast
    - Combinations of instructions for less frequent cases
    - Use die space saved by simpler design for on-chip cache
    - Later, simpler instructions made pipelining easier

# Introduction to MIPS

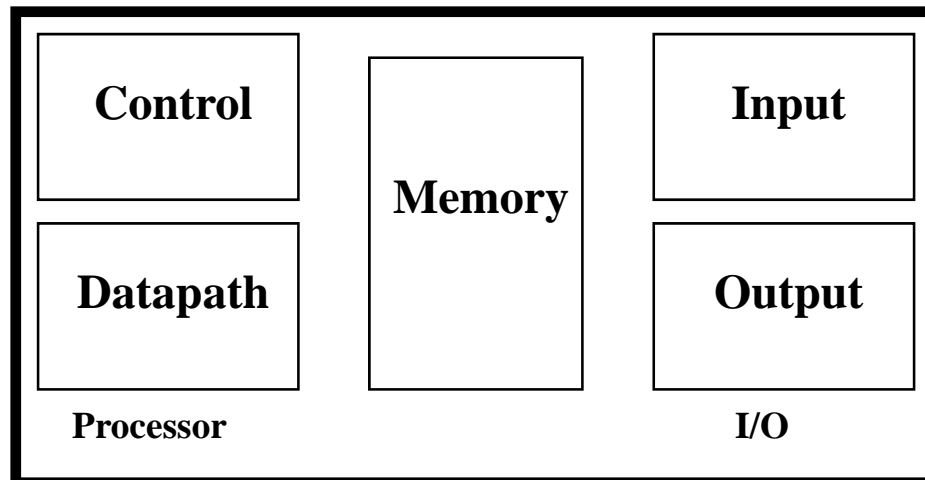
- Overview of MIPS instruction set architecture (ISA)
  - Assembly language
  - Machine code
- Objectives
  - Understand and create assembly language examples for remainder of course
  - Study an ISA (both the assembly code and machine code)
  - Examine a RISC ISA in depth, taking note of particular features and trade-offs
  - No need to become a MIPS assembly language programming expert

# The MIPS Architecture

- Use general purpose registers with a load-store architecture
  - *32 GPR, 32 FPR*
- Support most common addressing modes
  - *register, immediate, displacement*
- Support 8-, 16-, 32-, and 64-bit integers and 32- and 64-bit IEEE 754 floating point numbers
- Focus on most commonly executed instructions
  - *load, store, add, subtract, move, shift*
- Use small number of control instructions
  - *compare {equal, not equal}*
  - *branch PC-relative*
  - *jump, JAL (jump and link), JR (jump register/return)*
- Use fixed length instruction encoding

# Registers vs. Memory

- Arithmetic instructions operands must be registers,  
— only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables?





# Policy of Use Conventions

Name	Register number	Usage
\$zero	0	the constant value 0
\$v0-\$v1	2-3	values for results and expression evaluation
\$a0-\$a3	4-7	arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved
\$t8-\$t9	24-25	more temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

Register 1 (\$at) reserved for assembler, 26-27 for operating system

# Memory Organization

- Viewed as a large, single-dimension array, with an address
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data

...

# Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data

...

**Registers hold 32 bits of data**

- $2^{32}$  bytes with byte addresses from 0 to  $2^{32}-1$
- $2^{30}$  words with byte addresses 0, 4, 8, ...  $2^{32}-4$
- Words are aligned
  - What are the least 2 significant bits of a word address?

# MIPS Instruction Encoding

- All instructions are 32-bits
- Basic Instruction types
  - Arithmetic/Logic
  - Loads/Stores
  - Control
- Three instruction formats

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address/value		
J	op	26 bit address				

# ALU Instructions

- All ALU operations have three operands
- Operand order is fixed – destination first

```
a = b + c;
```

```
add $s1,$s2,$s3
```

- Keep intermediate results in registers if possible

```
a = b + c + d;
```

```
add $s1,$s2,$s3
```

```
add $s1,$s1,$s4
```

- opcode = 0, funct determines specific ALU operation
- Registers have numbers (\$s1 = 17, \$s2 = 18, \$s3 = 19)

R	op	rs	rt	rd	shamt	funct
		src1	src2	dst		
	000000	10010	10011	10001	00000	100000

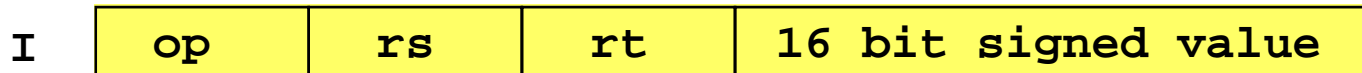
# ALU Instructions

- Immediate operands

```
a = b + 16;
```

```
addi $s1,$s2,16
```

- Immediate mode form of ALU operations
  - Opcode encodes specific operation (and indicates immediate)
  - 16-bit (signed) immediate value in low order 16-bits



# Load/Store Instructions

- Only instructions which access memory
- Displacement mode addressing

```
A[12] = h + A[8];
```

```
lw $t0, 32($s3)
add $t0, $s2, $t0
sw $t0, 48($s3)
```

- rs – base address register      rt – data register
  - rt -- source for Store; destination for Load

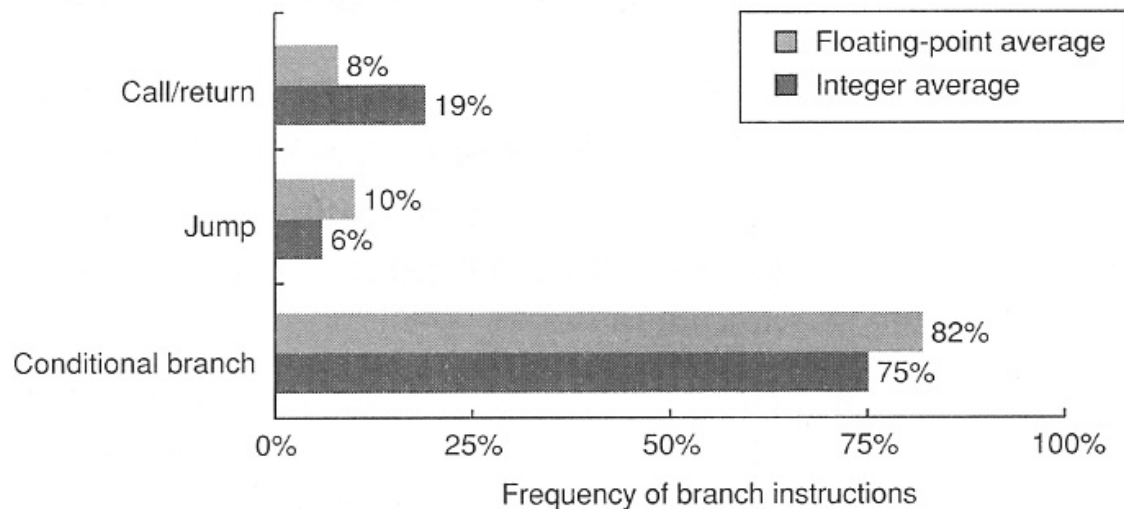
I	op	rs	rt	16 bit signed value
---	----	----	----	---------------------

- Not permitted

```
add 48($s3), $s2, 32($s3)
```

# Control Flow Instructions

- Conditional branches
- Jumps (unconditional branches)
- Procedure (function/subroutine) calls
- Procedure (function/subroutine) returns





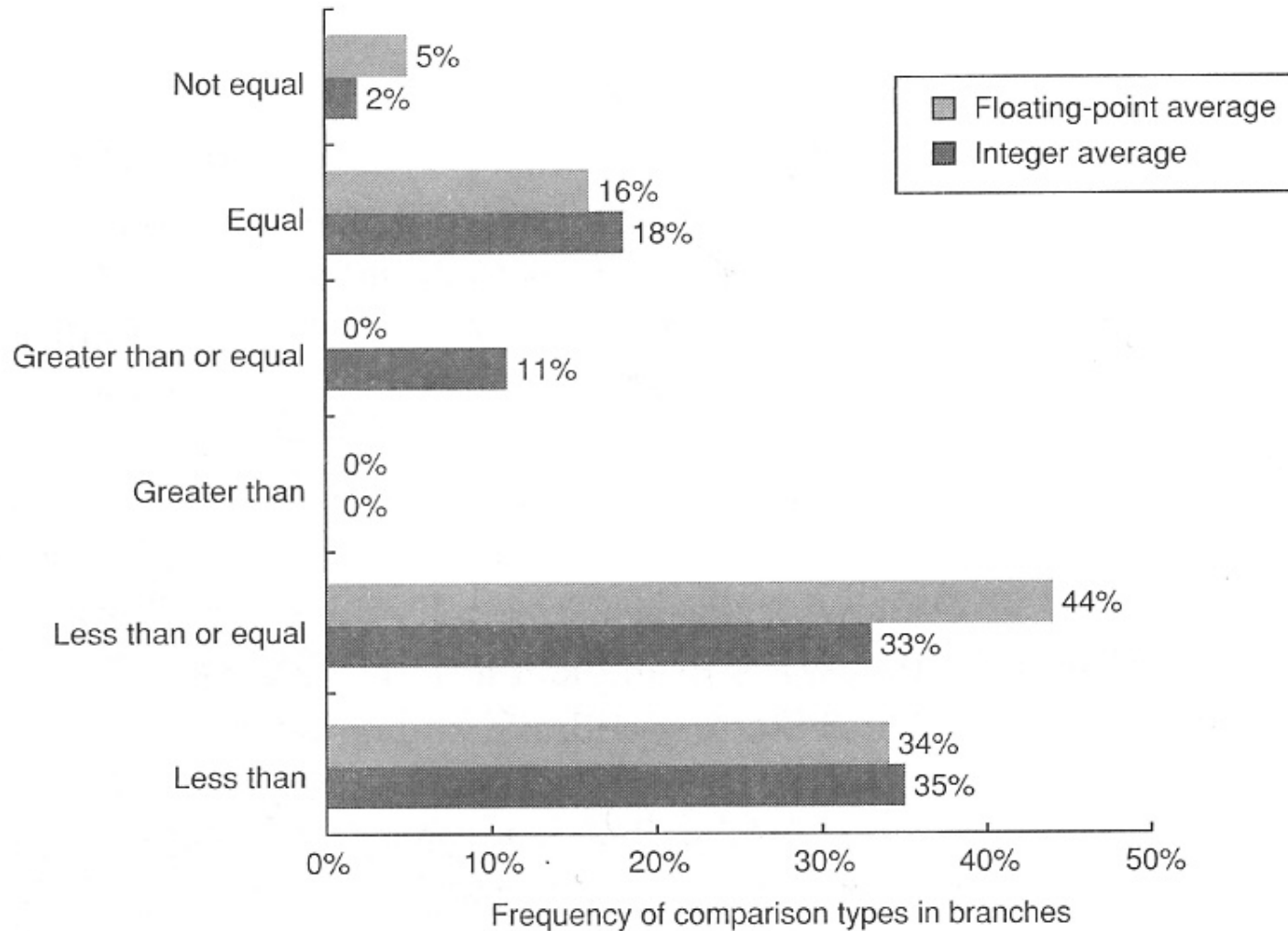
# Conditional Branch Options

- Conditional branches are said to be “taken” based upon a comparison or condition
  - Next instruction to be executed is the target
- If condition or comparison is not met, branch is said to be “not taken”
  - Next instruction executed is instruction following the branch instruction in the program
    - $PC + \langle \text{length of branch instruction in bytes} \rangle$

# Branch Condition Options

- Condition Codes
  - Special bits set as a consequence of ALU operations
    - Z – last result was zero
    - V – last result had over/under flow
    - C – last result had a carry
    - N – last result was negative
  - Constrain instruction order since information passed via CCs
- Condition register [MIPS]
  - Simple
  - Uses Register(s)
    - BEQZ R1, Target                      # Branch to target if R1 == 0
    - BEQ R1,R2,Target                    # Branch to target if R1 == R2
- Compare and branch [VAX]
  - BGEQ R1, R2, Target                # Branch on greater or equal

# Frequency of Compare Types in Conditional Branches

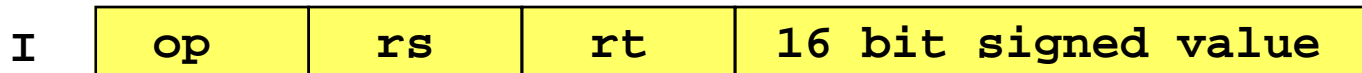


# Conditional Branch Instructions

- Condition is result of comparing two registers
- `bne`, `beq`

```
if (i==j)
    h = i + j;
```

```
bne $s0,$s1,Label
add $s4,$s0,$s1
Label: ...
```

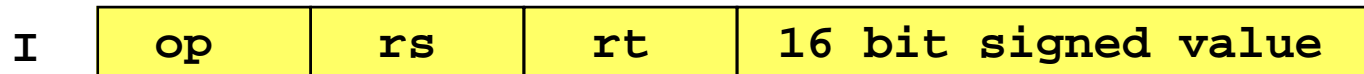


- Target address is PC-relative

# Conditional Branch Instructions

```
if (i==j)
    h = i + j;
```

```
bne $s0,$s1,Label
add $s4, $s0,$s1
Label: ...
```



- Target address is PC-relative
  - Exploits spatial locality
    - $\frac{1}{2}$  of branches in SPEC2000 < 16 instructions away
  - Saves bits of encoding (store offset, not absolute address)
  - Facilitates position independent code
  - Value is in words  $\rightarrow$  multiply by 4 and add to PC<sup>1</sup>
  - Provides range of  $+2^{15}-1$  to  $-2^{15}$  instructions from PC

<sup>1</sup>PC-4 since PC already incremented

# Other Conditional Branches

- Why not “branch if less than” `blt` instruction?
- Synthesize it from `slt` and `bne`

```
slt $t0, $s1, $s2
```

=

```
if $s1 < $s2 then
    $t0 = 1;
else
    $t0 = 0;
```

```
slt $t0, $s1, $s2
bne $t0, $zero, Label
```

Unlike condition codes, forces  
dependence between instructions to be  
through registers

- What about `beqz $s1, Label`?

```
beqz $s1, Label
```

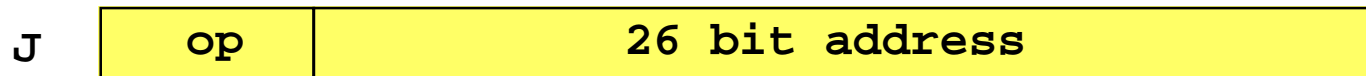
```
beq $s1, $zero, Label
```

# Unconditional Branch Instructions

- Jump instructions
- `j`, `jr`, `jal`

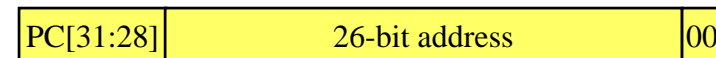
```
if (i != j)
    h = i + j;
else
    h = i - j;
```

```
beq $s4, $s5, L1
add $s3, $s4, $s5
j L2
L1: sub $s3, $s4, $s5
L2:
```



– Target address is pseudo-direct

- Value is in words
- Concatenate with PC[31:28] and 00
- Target must be in same 256 MB space as PC
  - not true PC-relative!



# Implementing Loops

```
while (save[i] == k)
    i++;
```

```
Loop: sll $t1,$s3,2      # $t1=4*i;
      add $t1,$t1,$s6    # addr of save[i]
      lw  $t0, 0($t1)    # $t0=save[i]
      bne $t0, $s5, Exit # Exit if k!=save[i]
      addi $s3,$s3,1     # i++
      j    Loop         # go to Loop
```

**Exit:** <next instructions>

- What if target is outside 256 MB space?
  - Jump Register instruction `jr $s0`

R	op	rs	rt	rd	shamt	funct
---	----	----	----	----	-------	-------



# Can you improve on this?

```
while (save[i] == k)
    i++;
```

```
        sll  $t1,$s3,2      # $t1 = 4*i;
        add  $t1,$t1,$s6    # ptr = save[i]
Loop:    lw   $t0, 0($t1)    # $t0=save[i]
        bne $t0, $s5, Exit  # Exit if k!=save[i]
        addi $s3,$s3,1      # i++
        addi $t1,$t1,4      # ptr++
        j     Loop         # go to Loop
```

```
Exit: <next instructions>
```

# Can you further improve on this?

```
while (save[i] == k)
    i++;
```

```
        sll  $t1,$s3,2      # $t1 = 4*i;
        add  $t1,$t1,$s6    # ptr = save[i]
        lw   $t0, 0($t1)    # *ptr
        bne  $t0, $s5, Exit # Exit if *ptr!=k
Loop:    addi $s3,$s3,1      # i++
        addi $t1,$t1,4      # ptr++
        lw   $t0, 0($t1)    # *ptr
        beq  $t0,$s5,Loop   # *ptr == k?
```

**Exit:** <next instructions>

# Procedure Invocation

- Save return address
  - Push onto stack or place in register
- Save register state
  - Responsibility of procedure being called
  - Responsibility of caller if register being used to store return PC
- Transmit arguments
  - Push onto stack
  - Place in registers
- Transfer control
  - PC for first instruction in procedure
- MIPS implementation “JAL – Jump And Link”
  - JAL <proc-name>
    - Store PC + 4 into R31
    - Jump to proc-name (PC  $\leftarrow$  proc-name)

# Returns from Procedure

- Set return value
- Restore state
  - At least register containing PC for return address
  - Any saved registers
- Transfer control to caller
- MIPS Implementation “JR – Jump Register”
  - JR R31
    - $PC \leftarrow R31$

# An Example

- Can we figure out the code?

v[] -- \$4

k -- \$5

temp -- \$15

our temporary (imposed by ISA) -- \$16

```
swap(int v[], int k);  
{ int temp;  
  temp = v[k]  
  v[k] = v[k+1];  
  v[k+1] = temp;  
}
```

multiply index k x4 index because ints are 4 bytes long ➡

add index to base address of v[]

load v[k] into \$15

load v[k+1] into \$16

store value of v[k+1] (\$16) into v[k]

store value of v[k] (\$15) into v[k+1]

return from procedure call

swap:

```
muli $2, $5, 4  
add $2, $4, $2  
lw $15, 0($2)  
lw $16, 4($2)  
sw $16, 0($2)  
sw $15, 4($2)  
jr $31
```

MIPS doesn't have register indirect; we use displacement instead

Could have used "strength reduction" on muli

Procedure invoked by "jal swap"

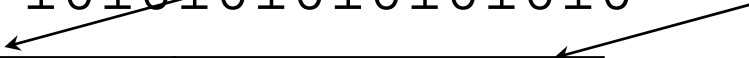
# Constants

- Small constants are used frequently (50% of operands)  
e.g.,  $A = A + 5;$   
 $B = B + 1;$   
 $C = C - 18;$
- Solutions? Why not?
  - create hard-wired registers (like \$zero) for constants like one
- MIPS instructions  
addi \$29, \$29, 4  
slti \$8, \$18, 10  
andi \$29, \$29, 6  
ori \$29, \$29, 4
- Design Principle: Make the common case fast.  
*Which format?*

# How about larger constants?

- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction

lui \$t0, 1010101010101010      filled with zeros



1010101010101010	0000000000000000
------------------	------------------

- Then must get the lower order bits right

ori \$t0, \$t0, 1010101010101010

ori	1010101010101010	0000000000000000
	0000000000000000	1010101010101010
<hr/>		
	1010101010101010	1010101010101010

# Assembly Language vs. Machine Language

- Assembly provides convenient symbolic representation
  - Much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudoinstructions'
  - `MOVE $t0, $t1`
    - Exists only in assembly
    - Implemented using `add $t0, $t1, $zero`
  - `LI` (load immediate)
    - Optional `LUI` (if 32-bit value) followed by `ORI`
  - `LA` (load address)
    - Implemented using `LUI` followed by `ORI`
    - `LA R1, Target`  
`JR R1`
- When considering performance you count real instructions



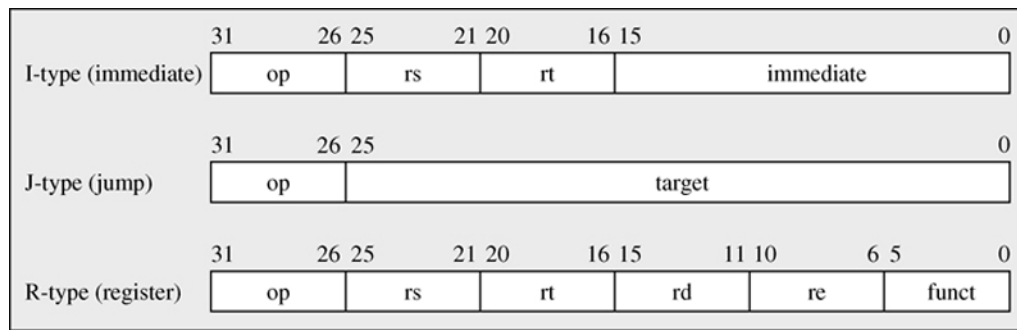
# MIPS Summary

## MIPS operands

Name	Example	Comments
32 registers	\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register \$zero always equals 0. Register \$at is reserved for the assembler to handle large constants.
2 <sup>30</sup> memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

## MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1, \$s2, \$s3	$\$s1 = \$s2 + \$s3$	Three operands; data in registers
	subtract	sub \$s1, \$s2, \$s3	$\$s1 = \$s2 - \$s3$	Three operands; data in registers
	add immediate	addi \$s1, \$s2, 100	$\$s1 = \$s2 + 100$	Used to add constants
Data transfer	load word	lw \$s1, 100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Word from memory to register
	store word	sw \$s1, 100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Word from register to memory
	load byte	lb \$s1, 100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Byte from memory to register
	store byte	sb \$s1, 100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Byte from register to memory
	load upper immediate	lui \$s1, 100	$\$s1 = 100 * 2^{16}$	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq \$s1, \$s2, 25	if ( $\$s1 == \$s2$ ) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1, \$s2, 25	if ( $\$s1 != \$s2$ ) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1, \$s2, \$s3	if ( $\$s2 < \$s3$ ) $\$s1 = 1$ ; else $\$s1 = 0$	Compare less than; for beq, bne
	set less than immediate	slti \$s1, \$s2, 100	if ( $\$s2 < 100$ ) $\$s1 = 1$ ; else $\$s1 = 0$	Compare less than constant
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	$\$ra = PC + 4$ ; go to 10000	For procedure call



(load, store, branch, ALU<sup>1</sup>)

(j, jal – pseudo-direct addressing)

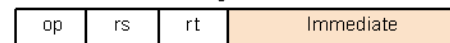
(ALU<sup>2</sup>, jr)

<sup>1</sup>if immediate operand

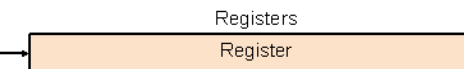
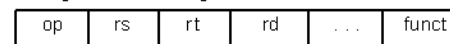
<sup>2</sup>if all register operands

Displacement mode

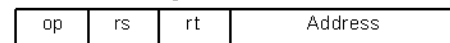
1. Immediate addressing



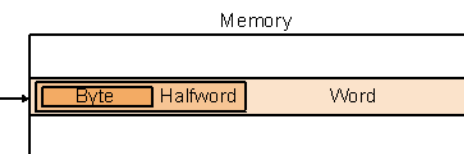
2. Register addressing



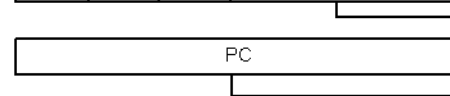
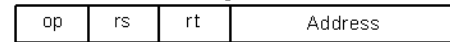
3. Base addressing



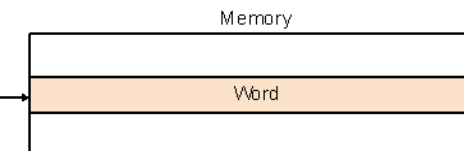
+



4. PC-relative addressing

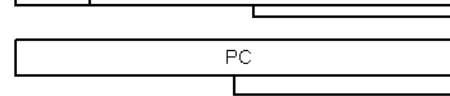
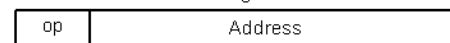


+

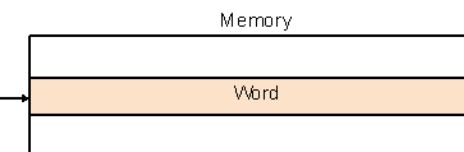


Multiply address x4, add to PC

5. Pseudodirect addressing



⊕



Multiply address x4, concatenate with high order 4-bits of PC

# Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI

*–“The path toward operation complexity is thus fraught with peril. To avoid these problems, designers have moved toward simpler instructions”*
- Let’s look (briefly) at IA-32

# IA - 32

- “This history illustrates the impact of the “golden handcuffs” of compatibility
- “adding new features as someone might add clothing to a packed bag”
- “an architecture that is difficult to explain and impossible to love”

# IA - 32

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: 57 new “MMX” instructions are added, Pentium II
  - Integer data
  - Used FP registers
  - Prohibited using FP and MMX extensions at same time
- 1999: The Pentium III added another 70 instructions (SSE)
  - Streaming SIMD Extensions
  - Added dedicated registers
- 2001: Another 144 instructions (SSE2)
- 2003: AMD extends the architecture to increase address space to 64 bits, widens all registers to 64 bits and other changes (AMD64)
- 2004: Intel capitulates and embraces AMD64 (calls it EM64T)
- 2004: Intel adds more media extensions, SSE3 (later SSSE3)

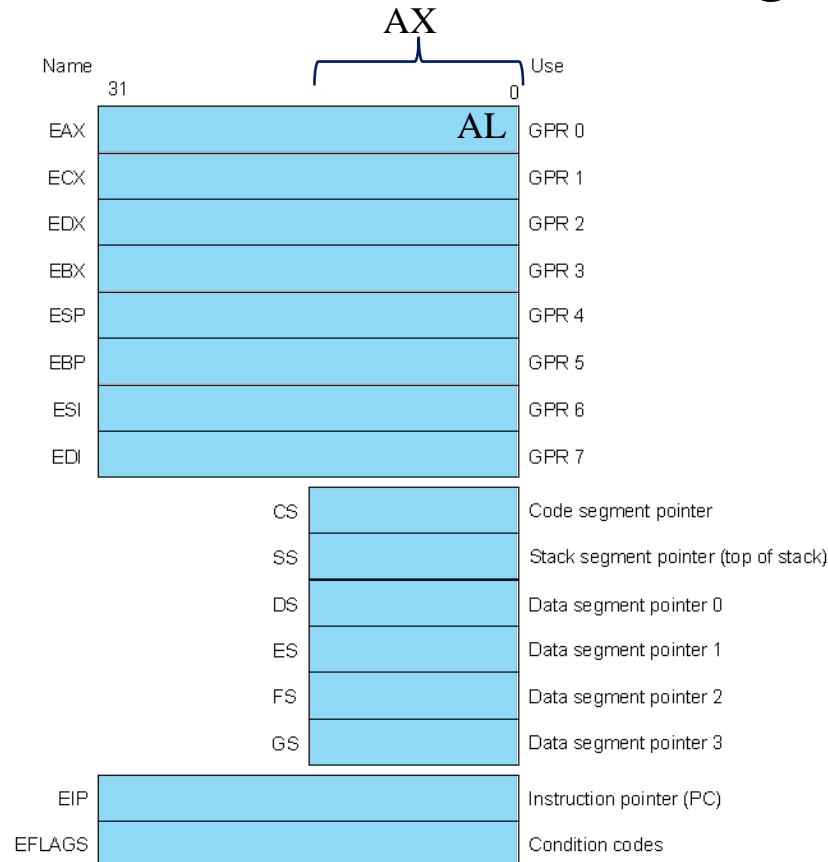
# IA-32 Overview

- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    - e.g., “base or scaled index with 8 or 32 bit displacement”
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

*“what the 80x86 lacks in style is made up in quantity,  
making it beautiful from the right perspective”*

# IA-32 Registers and Data Addressing

- Registers in the 32-bit subset that originated with 80386



# IA-32 Register Restrictions

- Registers are not “general purpose” – note the restrictions below

Mode	Description	Register restrictions	MIPS equivalent
Register Indirect	Address is in a register.	not ESP or EBP	lw \$s0,0(\$s1)
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	not ESP or EBP	lw \$s0,100(\$s1) # ≤16-bit displacement
Base plus scaled Index	The address is Base + (2 <sup>Scale</sup> × Index) where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	mul \$t0,\$s2,4 add \$t0,\$t0,\$s1 lw \$s0,0(\$t0)
Base plus scaled Index with 8- or 32-bit displacement	The address is Base + (2 <sup>Scale</sup> × Index) + displacement where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	mul \$t0,\$s2,4 add \$t0,\$t0,\$s1 lw \$s0,100(\$t0) # ≤16-bit displacement

**FIGURE 2.42 IA-32 32-bit addressing modes with register restrictions and the equivalent MIPS code.** The Base plus Scaled Index addressing mode, not found in MIPS or the PowerPC, is included to avoid the multiplies by four (scale factor of 2) to turn an index in a register into a byte address (see Figures 2.34 and 2.36). A scale factor of 1 is used for 16-bit data, and a scale factor of 3 for 64-bit data. Scale factor of 0 means the address is not scaled. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent mode would need two more instructions: a `lui` to load the upper 16 bits of the displacement and an `add` to sum the upper address with the base register `$s1`. (Intel gives two different names to what is called Based addressing mode—Based and Indexed—but they are essentially identical and we combine them here.)



# IA-32 Typical Instructions

- Four major types of integer instructions:
  - Data movement including move, push, pop
  - Arithmetic and logical (destination register or memory)
  - Control flow (use of condition codes / flags )
  - String instructions, including string move and string compare

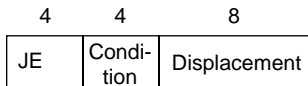
Instruction	Function
JE name	if equal(condition code) (EIP=name); EIP-128 ≤ name < EIP+128
JMP name	EIP=name
CALL name	SP=SP-4; M[SP]=EIP+5; EIP=name;
MOVW EBX,[EDI+45]	EBX=M[EDI+45]
PUSH ESI	SP=SP-4; M[SP]=ESI
POP EDI	EDI=M[SP]; SP=SP+4
ADD EAX,#6765	EAX= EAX+6765
TEST EDX,#42	Set condition code (flags) with EDX and 42
MOVSL	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

**FIGURE 2.43 Some typical IA-32 instructions and their functions.** A list of frequent operations appears in Figure 2.44. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)

# IA-32 instruction Formats

- Typical formats: (notice the different lengths)

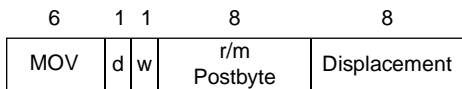
a. JE EIP + displacement



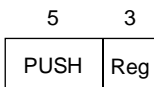
b. CALL



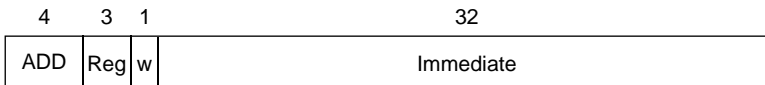
c. MOV EBX, [EDI + 45]



d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



# Summary

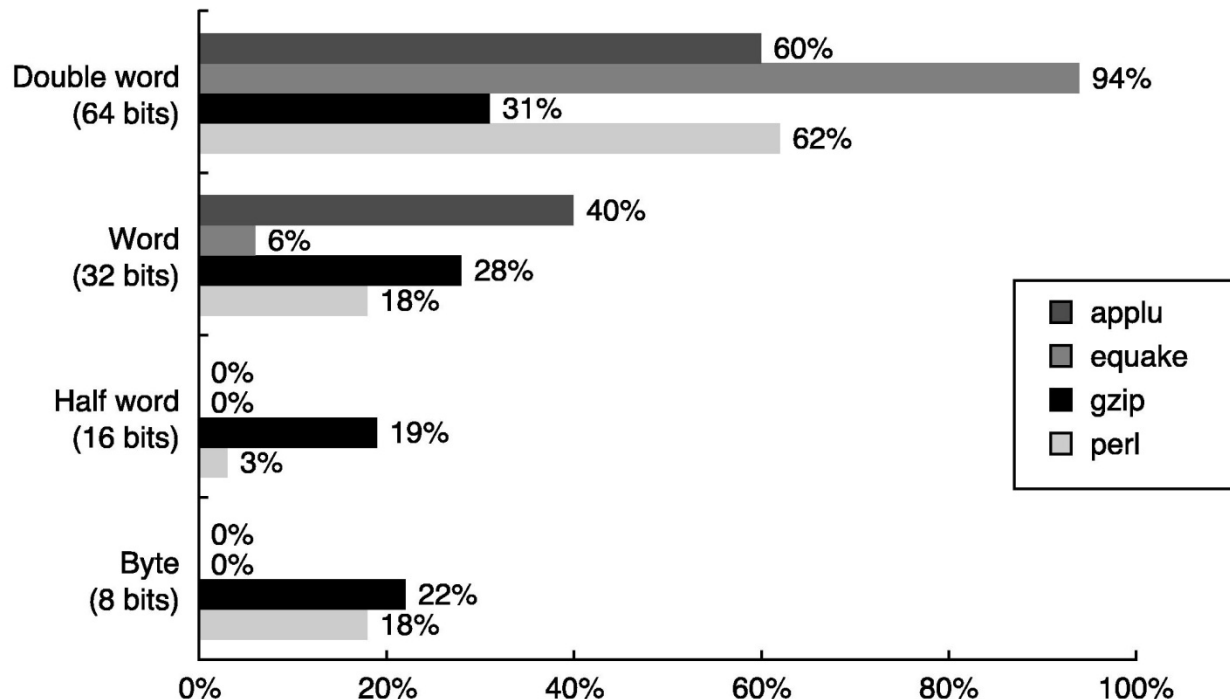
- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate
- Design Principles:
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast
- Instruction set architecture
  - a very important abstraction

# The Trimedia TM32 CPU

- Media Processor
  - Embedded processors dedicated to multimedia
    - Set-top boxes
  - Similar to DSPs but rely on compiler technology
    - C not hand-coded assembly language
  - VLIW (Very Long Instruction Word)
    - Parallelism
  - Permits 5 parallel operations
    - NOPs if unable to take advantage of all

# Fallacy

- There is such a thing as a typical program
  - Can't rely on single synthetic benchmark
  - Instruction frequency, addressing modes, operand types can vary widely between applications



# Fallacy

- An architecture with flaws cannot be successful
  - Intel X86 (IA32)
    - Segmentation vs. paging
    - Extended accumulators vs. GPRs
    - Stack for floating point
    - Enormously complicated ISA
  - Wildly successful in PC marketplace (85% share?)
    - Chosen by IBM for first PC
      - Importance of x86 binary compatibility
    - Moore's law permitted RISC “under the hood” to emulate CISC
    - High volume permits higher R&D costs needed to support complexity

# Fallacy

- You can design a flawless architecture
  - VAX designers in 1975 focused on
    - Code size
    - Bridging “semantic gap”
  - Didn’t anticipate effects (five years later) of
    - On-chip caches
    - Pipelining

# Pitfall

- Designing a “high-level” instruction set feature specifically oriented to supporting a high-level language structure
  - VAX CALLS instruction



# Pitfall

- Innovating at the instruction set architecture level to reduce code size without accounting for the compiler
  - >30-40% improvement a major achievement at ISA level
  - >2x variation attributable to compilers

Compiler	Apogee Software: Version 4.1	Green Hills: Multi2000 Version 2.0	Algorithmics SDE4.0B	IDT/c 7.2.1
Architecture	MIPS IV	MIPS IV	MIPS 32	MIPS 32
Processor	NEC VR5432	NEC VR5000	IDT 32334	IDT 79RC32364
Autocorrelation kernel	1.0	2.1	1.1	2.7
Convolutional encoder kernel	1.0	1.9	1.2	2.4
Fixed-point bit allocation kernel	1.0	2.0	1.2	2.3
Fixed-point complex FFT kernel	1.0	1.1	2.7	1.8
Viterbi GSM decoder kernel	1.0	1.7	0.8	1.1
Geometric mean of five kernels	1.0	1.7	1.4	2.0

Code size variation among compilers (same ISA)

# Pitfall

- Expecting to get good performance from a compiler for DSPs

TMS320C54 D ("C54") for DSPstone kernels	Ratio to assembly in execution time (> 1 means slower)	Ratio to assembly code space (> 1 means bigger)	TMS 320C6203 ("C62") for EEMBC Telecom kernels	Ratio to assembly in execution time (> 1 means slower)	Ratio to assembly code space (> 1 means bigger)
Convolution	11.8	16.5	Convolutional encoder	44.0	0.5
FIR	11.5	8.7	Fixed-point complex FFT	13.5	1.0
Matrix $1 \times 3$	7.7	8.1	Viterbi GSM decoder	13.0	0.7
FIR2dim	5.3	6.5	Fixed-point bit allocation	7.0	1.4
Dot product	5.2	14.1	Autocorrelation	1.8	0.7
LMS	5.1	0.7			
N real update	4.7	14.1			
IIR n biquad	2.4	8.6			
N complex update	2.4	9.8			
Matrix	1.2	5.1			
Complex update	1.2	8.7			
IIR one biquad	1.0	6.4			
Real update	0.8	15.6			
C54 geometric mean	3.2	7.8	C62 geometric mean	10.0	0.8

# Pitfall (not in text)

- Relying only on instruction frequency counts without understanding the underlying architecture
  - The ISA may “force” certain patterns of usage because others are unavailable to the compiler writer
    - OK if seeking ways to optimize current ISA
    - Not OK if looking for alternatives to the ISA
  - You may need to “go deeper” to understand why certain instructions are used and how
  - Example: MIPS use of \$R0 to always contain 0 may affect frequency counts and distribution of immediate mode values

# Perq Systems

- Xerox Alto origins
- One of earliest commercially available personal workstations
  - Perq
  - Apollo
  - Tektronix
  - Sun Microsystems
- Microcoded
  - RasterOp / BitBlt (Bit Block Transfer)
  - Pascal “byte codes”
  - High performance but must microcode!

