



香港中文大學
計算機科學與工程學系

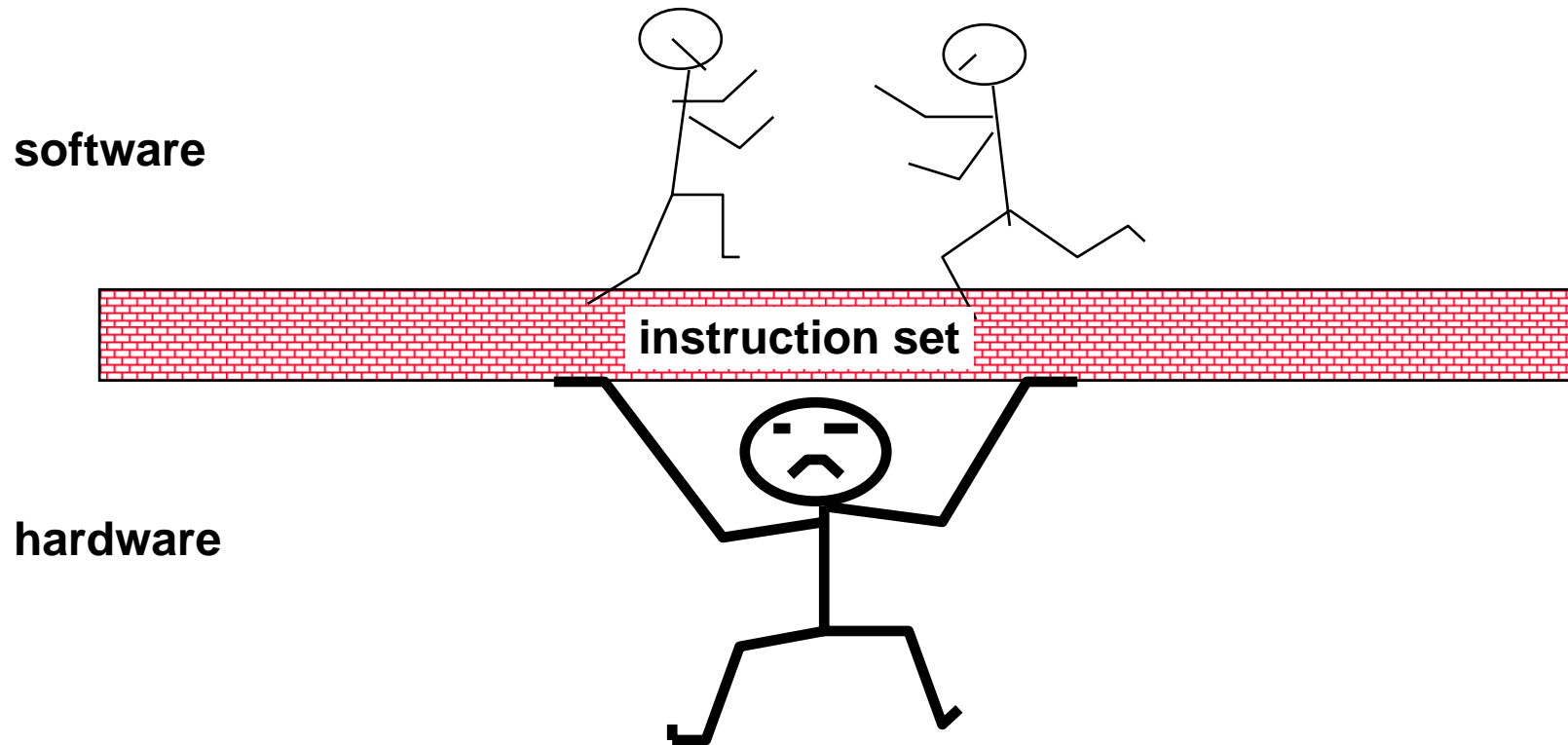
CSC 3420

Computer Systems Architecture

Chapter 3

Instructions : Language of the Machine

Instruction Set Design



Instructions:

- **Vocabulary** of Instructions is called an **Instruction Set**
- **Language of the Machine**
- **More primitive than higher level languages**
e.g., no sophisticated control flow like “Do loop”; “If Then Else”
- **Very restrictive**
e.g., MIPS Arithmetic Instructions
- **Machine Languages are quite similar, more like regional dialects** than like independent languages.
- **We’ll be working with the MIPS instruction set architecture**
 - similar to other architectures developed since the 1980's
 - used by NEC, Nintendo, Silicon Graphics, Sony
- *Design goals: maximise performance, minimise cost and reduce design time*

Instruction Set Architecture

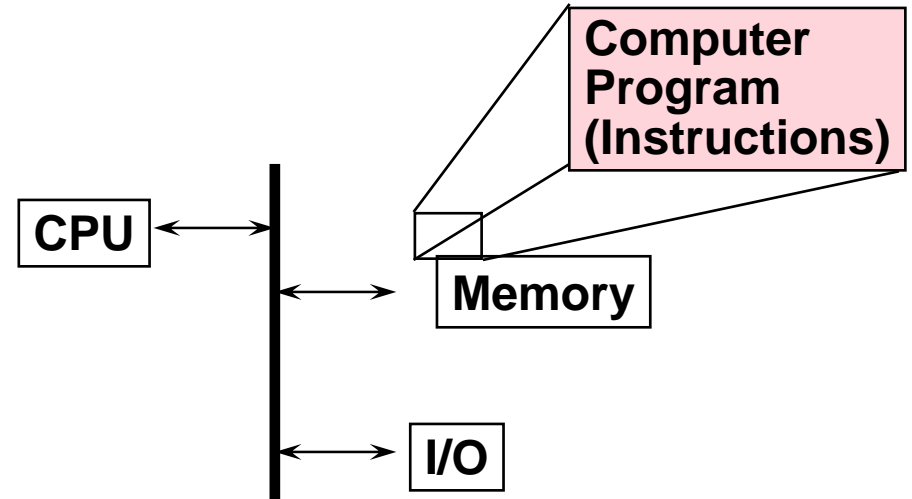
Programmer's View

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

Computer's View, Set if ON/OFF signals

Princeton (Von Neumann) Architecture

- Data and Instructions mixed in same memory ("**stored program computer**")
- Program as data (**dubious advantage**)
- Storage utilisation
- Single memory interface

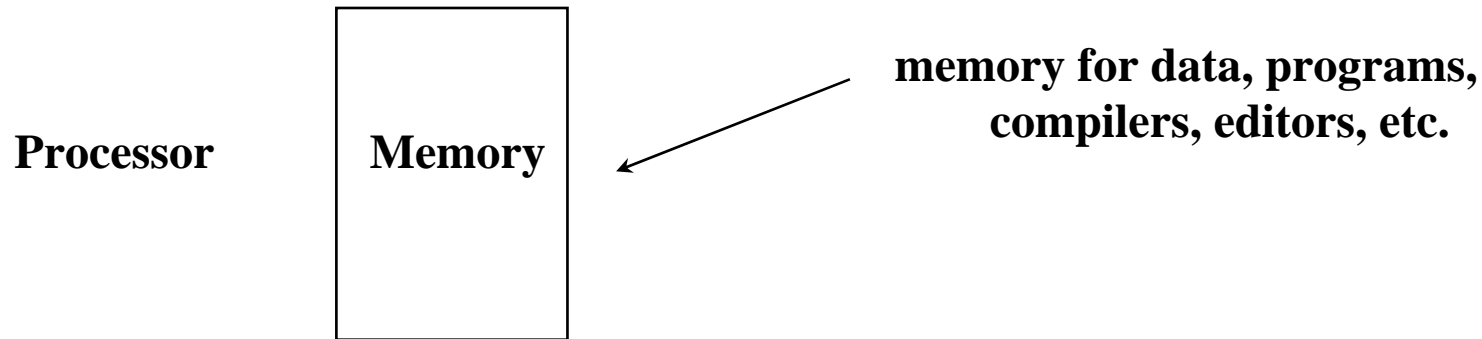


Harvard Architecture

- Data & Instructions in separate memories
- Has advantages in certain high performance implementations

Stored Program Concept

- **Instructions are bits**
- **Programs are stored in memory**
 - to be read or written just like data



Fetch & Execute Cycle

- **Instructions are fetched and put into a special register**
 - E.g. Instruction Register
- **Bits in the register "control" the subsequent actions**
- **Fetch the “next” instruction and continue**

Basic Issues in Instruction Set Design

- **What operations** (and how many) should be provided ?

LD/ST/INC/BRN sufficient to encode any computation but not useful because programs too long!

- **How** (and how many) **operands are specified**

Most operations are dyadic (e.g., $A \leftarrow B + C$)
Some are monadic (e.g., $A \leftarrow \sim B$)

- **How to encode** these into consistent instruction formats?
Assignment of bit patterns

- Should Instructions be **multiples of basic data/address widths?**

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 ($A := B + C$)

Challenge: encode operations in a small number of bits!

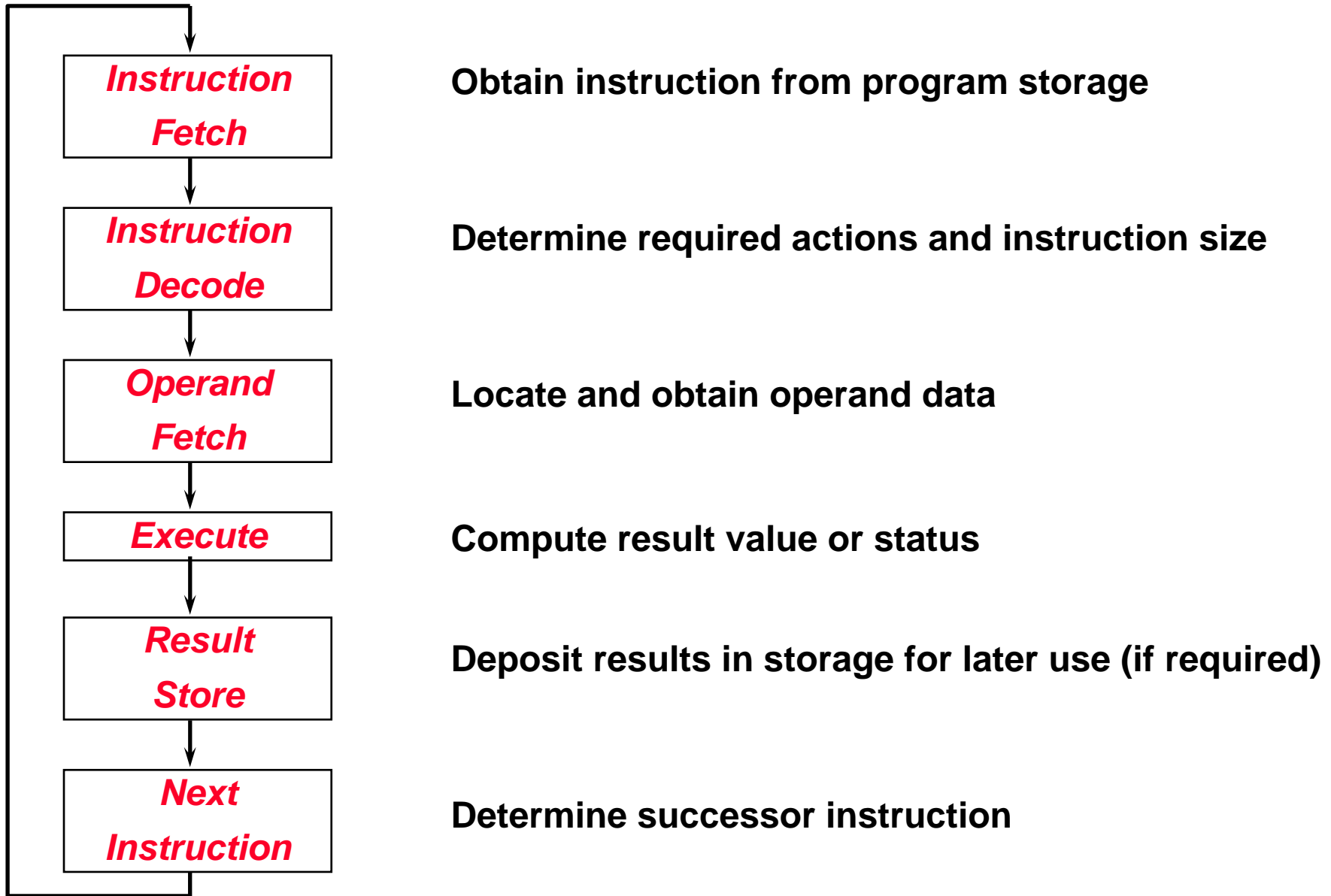
Design Principles :

1. Simplicity Favours regularity (e.g. 3 address arithmetic instructions)
2. Smaller is faster (e.g. use register instead of memory)
3. Good design demands good compromises
(e.g. one instruction size and multiple instruction formats)
4. Make the common case fast (immediate operands)

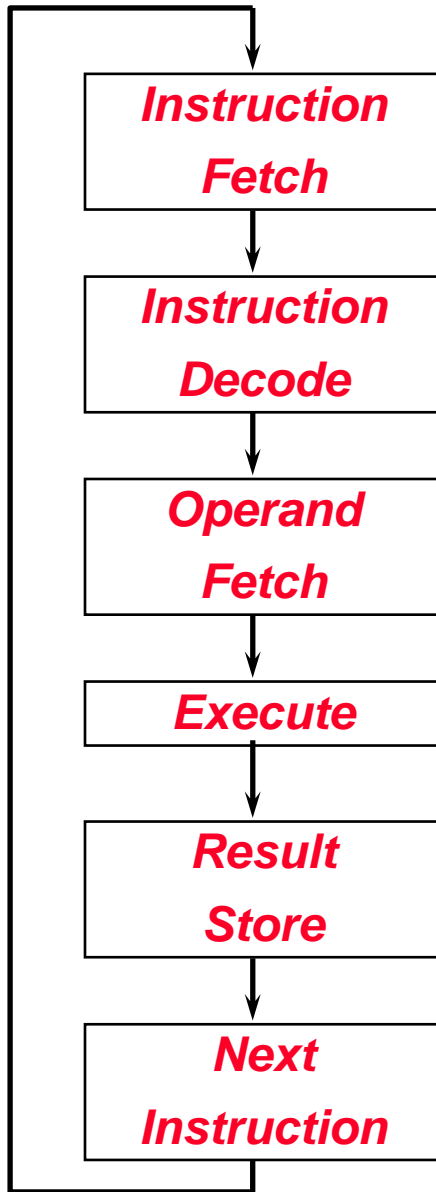
Assembly Language vs. Machine Language

- Assembly **provides convenient symbolic representation**
 - much easier than writing down (binary or hex) numbers
 - e.g., variable name instead of address
- **Machine language is the underlying reality**
 - e.g., destination is no longer first
- Assembly can **provide 'pseudo-instructions'**
 - e.g., “move \$t0, \$t1” exists only in Assembly
 - would be implemented using “add \$t0,\$t1,\$zero”
- When considering performance you should count real (machine) instructions

Typical Execution Cycle



What Must be Specified?



fetch-decode-execute is implicit!

Basic ISA Classes

Accumulator:

1 address add A $\text{acc} \leftarrow \text{acc} + \text{mem} [A]$

1+x address addx A $\text{acc} \leftarrow \text{acc} + \text{mem} [A + x]$

Stack:

0 address add $\text{tos} \leftarrow \text{tos} + \text{next}$

General Purpose Register (GPR):

2 address add A B $\text{EA}(A) \leftarrow \text{EA}(A) + \text{EA}(B)$

3 address add A B C $\text{EA}(A) \leftarrow \text{EA}(B) + \text{EA}(C)$

Load/Store: (a modified form of GPR design)

3 address load Ra Rb $\text{Ra} \leftarrow \text{mem} [\text{Rb}]$

 add Ra Rb Rc $\text{Ra} \leftarrow \text{Rb} + \text{Rc}$

 store Ra Rb $\text{mem} [\text{Rb}] \leftarrow \text{Ra}$

How to compare?

Bytes per instruction? Number of Instructions? Cycles per instruction?

Comparing Number of Instructions

- Code sequence for $C = A + B$ for four classes of instruction sets:

Stack

Accumulator

Register

Register

(register-memory)

(load-store)

Stack machine: no general purpose registers - all operations are performed using the stack

Load-store machine: only load and store instructions reference memory
All ALU operations are performed using registers only.

Comparing Number of Bytes

Assumptions: 1 byte OP code; 4 byte memory address; 1 byte Reg. No.

Stack

Accumulator

Register

(register-memory)

Register

(load-store)

Comparing Number of Memory Access

Assumptions: 1 memory access for OP code; 1 access per memory address

Stack

Accumulator

Register

(register-memory)

Register

(load-store)

Now repeat the exercises (coding and comparisons) for a longer sequence:

$$A = (A + B * C) / (B^2 + C^2)$$

General Purpose Registers Dominate

- * Since 1975 all machines use general purpose registers
- * **Advantages of registers**
 - * registers are **faster** than memory
 - * registers are **easier for a compiler** to use
 - e.g., $(A*B) - (C*D) - (E*F)$ can do multiplies in any order
 - Stack is the most restrictive one
 - * registers can **hold variables**
 - **memory traffic is reduced**, so program is sped up (registers are also faster than memory)
 - **code density improves** (since register named with fewer bits than memory location)
- * **Stack machines: Burroughs B55/5700 mainframe, HP3000, Transputer, HP pocket calculators. With new Java machines, stack machines may make a comeback.**

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 (Load & Store)
1	2	Intel 80x86, Motorola 68000 (Register-memory)
2	2	VAX (also has 3-operand formats)
3	3	VAX (also has 2-operand formats)

Summary on Instruction Classes

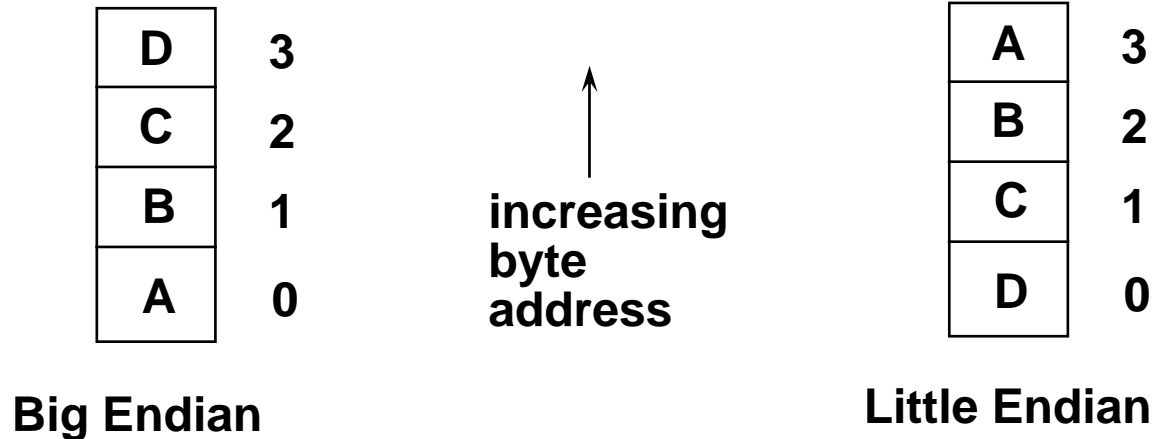
- Expect new instruction set architecture to use general purpose register
- Pipelining => Expect it to use load store variant of **GPR** (general purpose register) ISA

Memory Addressing

- Since 1980 almost every machine uses addresses to level of 8-bits (byte) (**Why?**)
- Three questions for design of ISA:
 - Since ISA could read a 32-bit word as four loads of bytes from sequential byte addresses or 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?
 - How to generate the memory addresses?

Addressing modes

Addressing Objects



Word A B C D
Word address 0

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

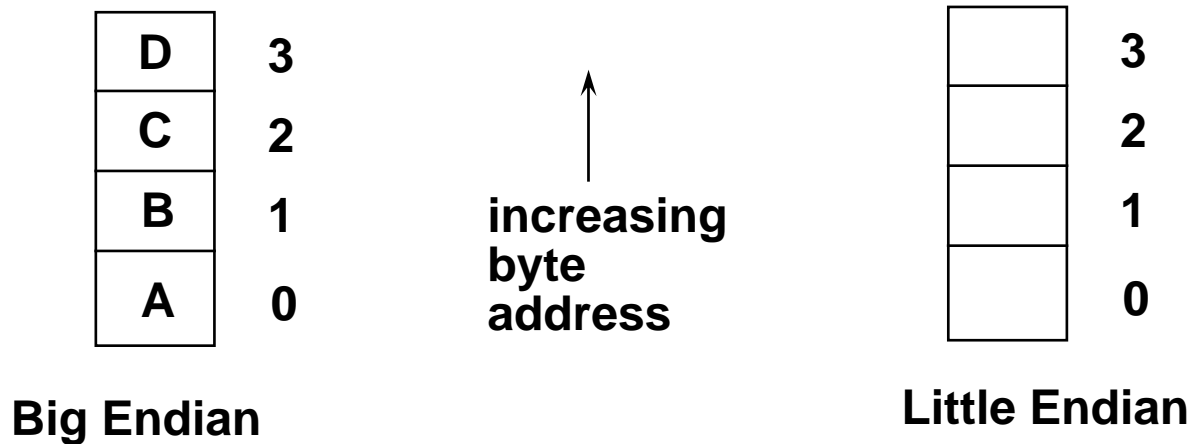
- IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

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

- Intel 80x86, DEC Vax

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

- **Each system is self-consistent**, but causes problems when they need to communicate!
- When words are transferred between Big Endian and Little Endian machines, you must permute the bytes to successfully copy the data



Word A B C D
Word address 0

Addressing Modes: how data is accessed?

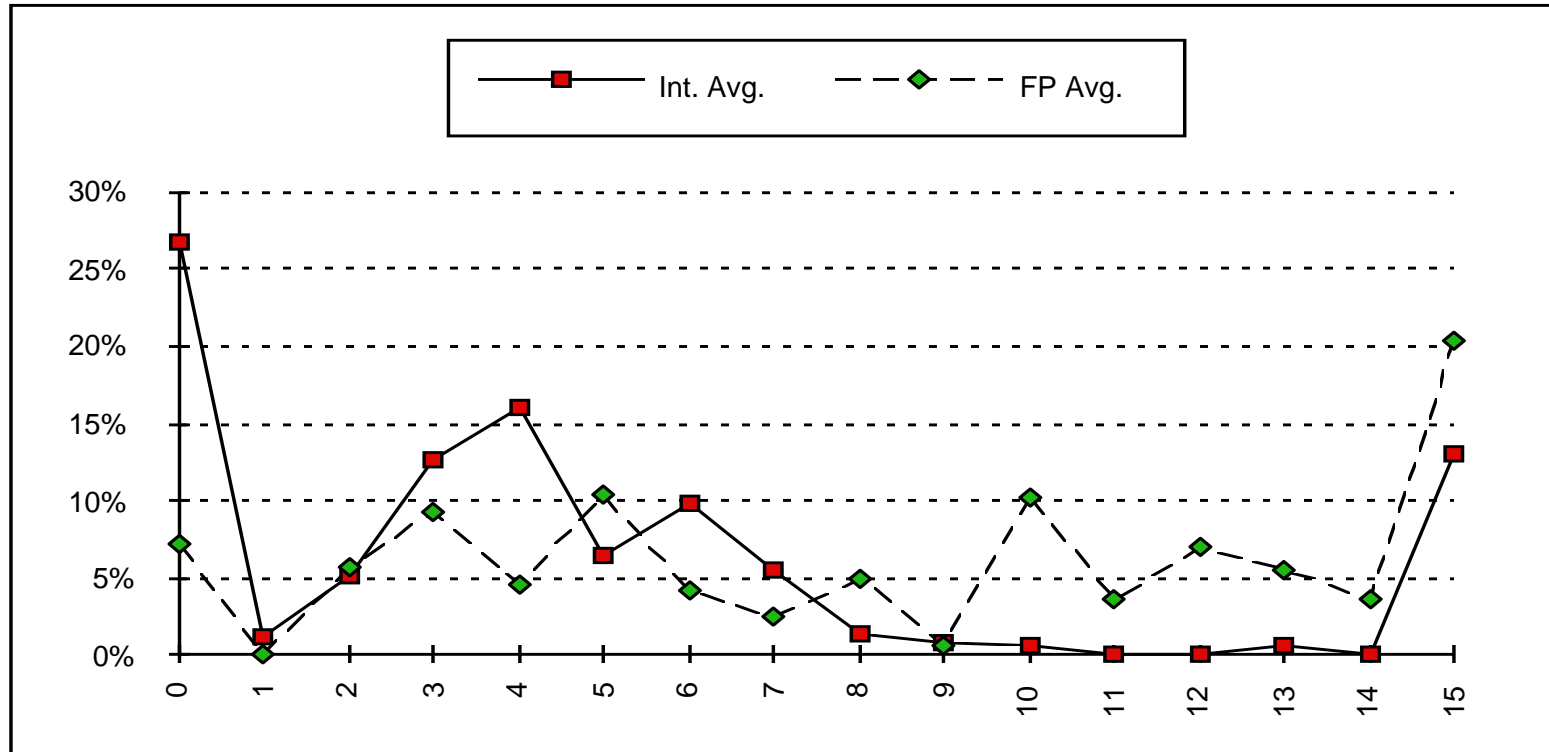
Addressing mode	Example	Meaning
Register	Add R4,R3	R4 ←
Immediate	Add R4,#3	R4 ←
Register indirect	Add R4,(R1)	R4 ←
Displacement	Add R4,100(R1)	R4 ←
Indexed	Add R3,(R1+R2)	R3 ←
Direct or absolute	Add R1,(100)	R1 ←
Memory indirect	Add R1,@(R3)	R1 ←
Auto-increment	Add R1,(R2)+	R1 ←
Auto-decrement	Add R1,-(R2)	R2 ←
Scaled	Add R1,d,100(R2)[R3]	R1 ←

Addressing Mode Usage

3 programs measured on machine with all address modes (VAX)

Displacement:	42% avg, 32% to 55%
Immediate:	33% avg, 17% to 43%
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%

Displacement Address Size



Average of 5 programs from SPECint92 and Average of 5 programs from SPECfp92

X-axis is in powers of 2: (indication of number of address bits)

X= 4 means addresses $> 2^3$ (8) and $\leq 2^4$ (16)

1% of addresses > 16-bits

Immediate (constant) Size

- 50% to 60% fit within 8 bits ($0 \leq X < 255$)
- 75% to 80% fit within 16 bits ($256 < X \leq 65535$)

Addressing: Points to remember

- 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

What to do if

Displacement $> 2^{16}$

Immediate $> 2^{16}$

Typical Operations

Data Movement	Load (from memory) memory-to-memory move input (from I/O device) push, pop (to/from stack)	Store (to memory) register-to-register move output (to I/O device)
Arithmetic	Data Types: Operations:	
Logical		
Shift		
Control (Jump/Branch)		
Subroutine Linkage		
Interrupt		
Synchronisation		
String		

Top 10 80x86 Instructions

Rank	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%

- Simple instructions dominate instruction frequency

Methods of Testing Condition

- **Condition Codes**

Processor status bits are set as a **side-effect of arithmetic instructions** (possibly on Moves too) **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**

Branches

Conditional control transfers

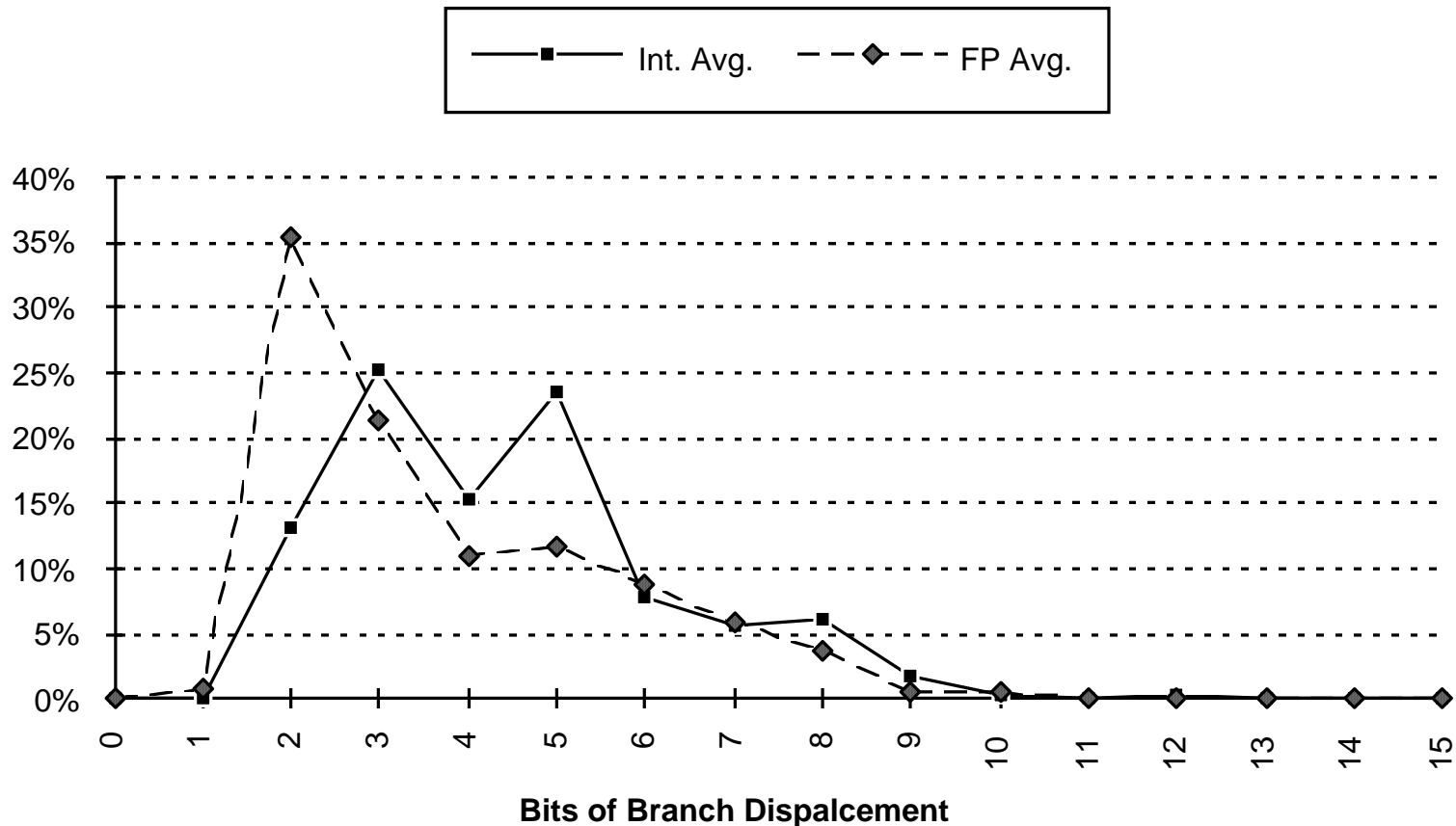
Four basic conditions:

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

Sixteen example combinations of the basic four conditions:

Always	Unconditional
Never	NOP
Not Equal	$\sim Z$
Equal	Z
Greater	$\sim[Z + (N \text{ NEQ } V)]$
Less or Equal	$Z + (N \text{ NEQ } V)$
Greater or Equal	$\sim(N \text{ NEQ } V)$
Less	$N \text{ NEQ } V$
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

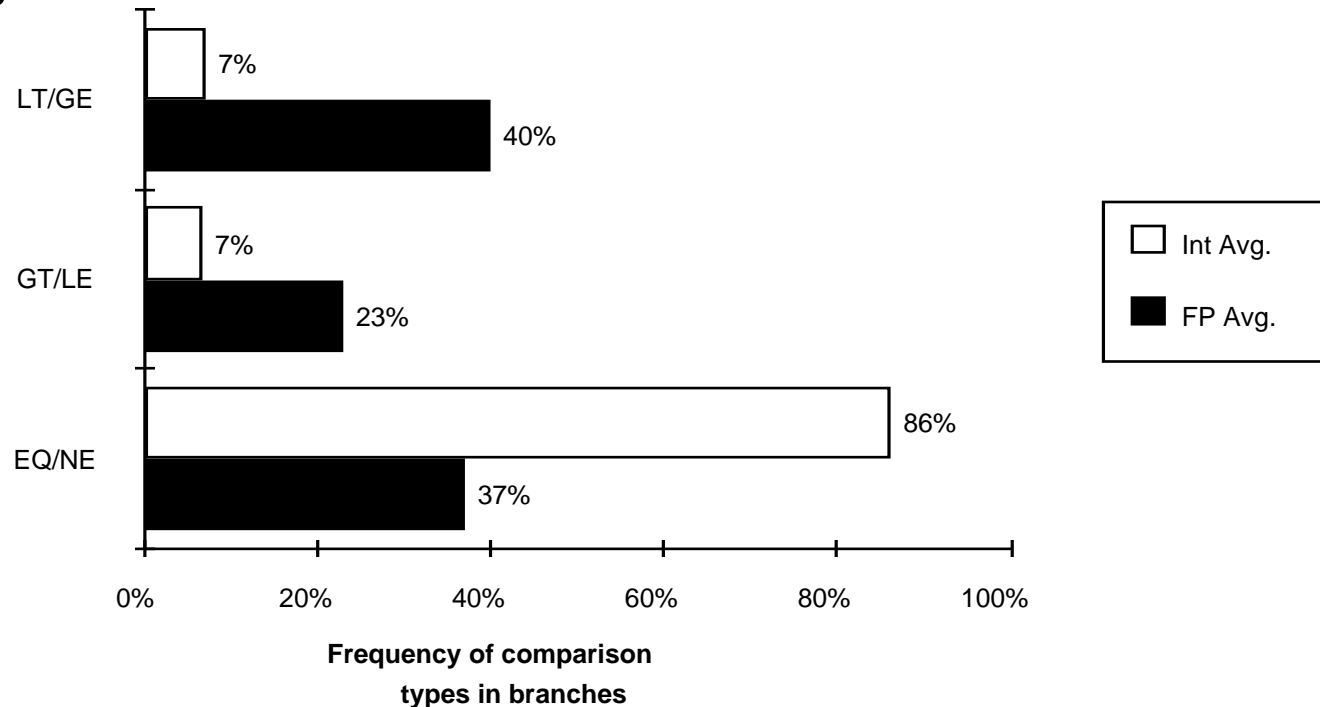
Conditional Branch Distance



- Distance from branch in instructions 2^i means $2^{i-1} - 1 \leq |\text{branch offset}| < 2^i - 1$ ($i > 0$)
- 25% of integer branches are between 2^2 and 2^4 (OR -2^2 and -2^4)

Conditional Branch Addressing

- **PC-relative** since most branches are relatively close to the current PC address
- At least **8 bits** suggested (± 128 instructions)
- **Compare Equal/Not Equal** most important for integer programs



Operation Summary

- **Support & accelerate 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 (with a PC-relative address at least 8-bits long),
 - jump, call, return;

Data Types

Bit: 0, 1

Bit String: sequence of bits of a particular length

4 bits is a nibble

16 bits is a half-word (VAX: word)

8 bits is a byte

32 bits is a word (VAX: long word)

Character:

ASCII 7 bit code

EBCDIC 8 bit code

ISO 10646 16 bit code (or more)

Decimal: (4/6/8/10 bytes, variable byte string)

digits 0-9 encoded as 0000b through 1001b

two decimal digits packed per 8 bit byte

Integers: (8/16/32/48/64/80/128 bits)

Sign & Magnitude: 0X vs. 1X

1's Complement: 0X vs. 1(~X)

2's Complement: 0X vs. (1's comp) + 1

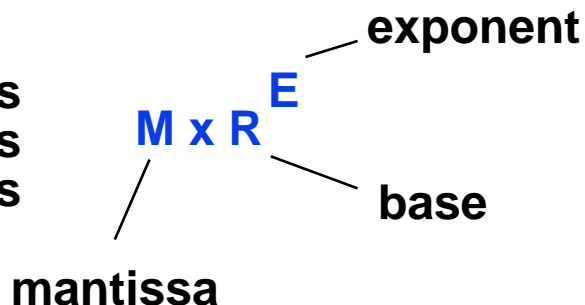
Positive #'s same in all
First 2 have two zeros
2's Comp. usually chosen

Floating Point:

Single Precision 32bits

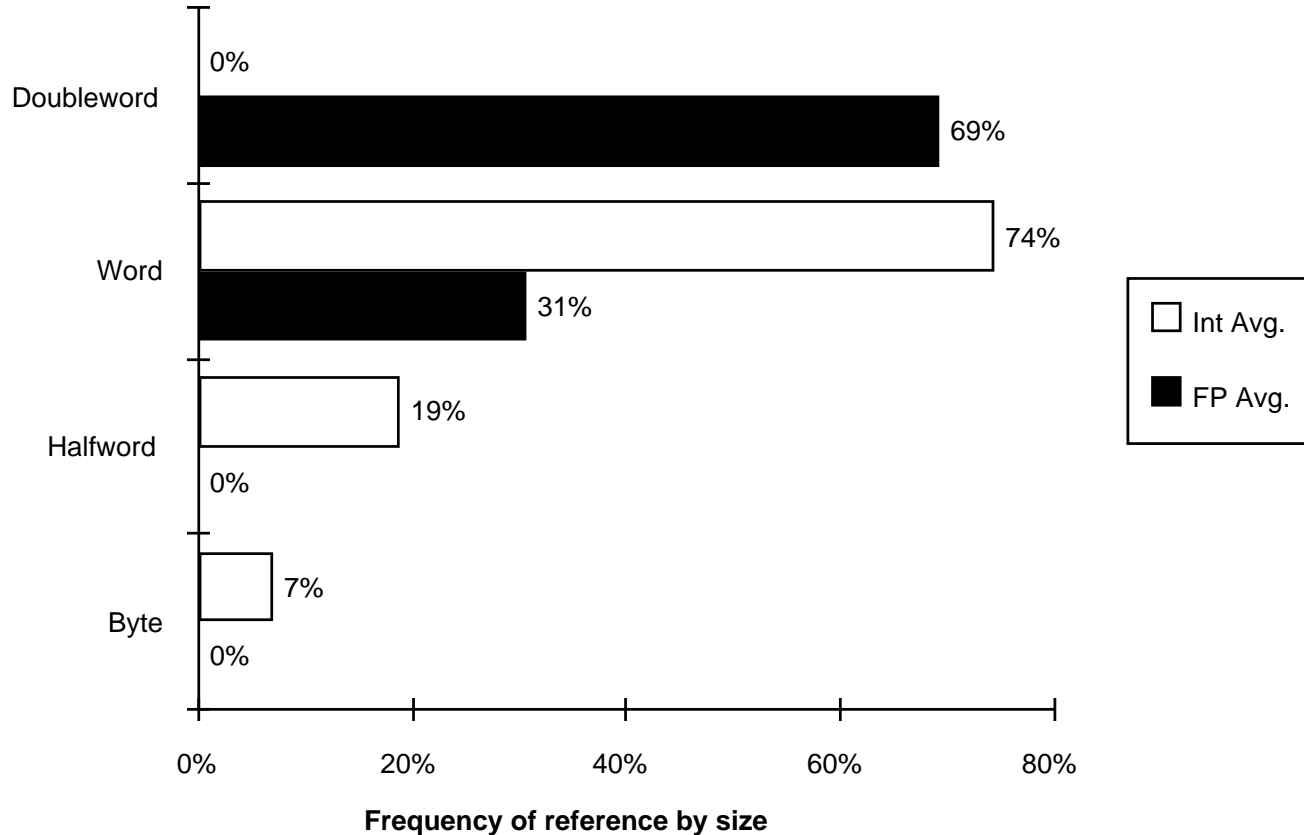
Double Precision 64bits

Extended Precision 80bits



How many +/- numbers.?
Where is decimal point?
How are +/- exponents represented?

Operand Size Usage



Support these data sizes and types:

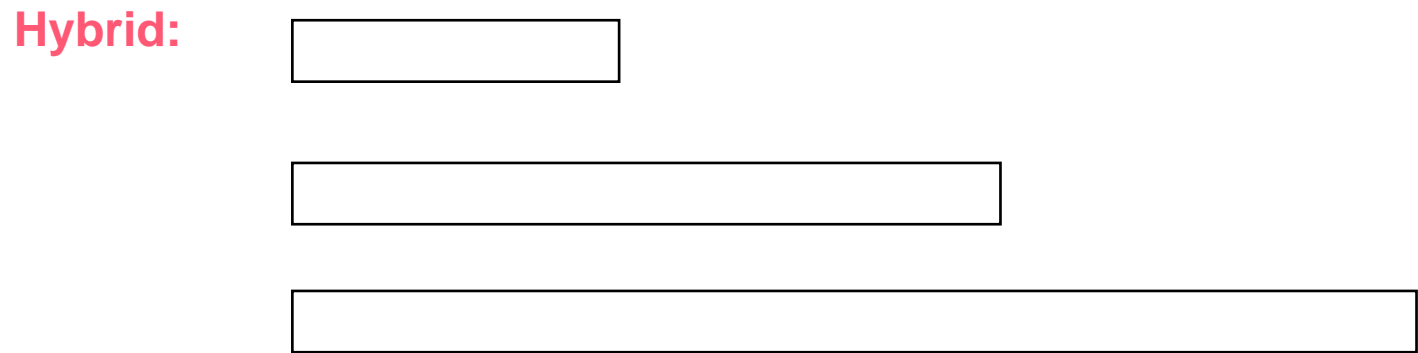
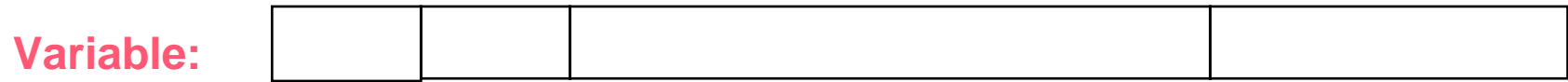
8-bit, 16-bit, 32-bit integers and

32-bit and 64-bit IEEE 754 floating point numbers

Instruction Format

- **If you have many memory operands per instructions and many addressing modes, need an Address Specifier per operand**
- **If you have load-store machine with 1 address per instruction and one or two addressing modes, then just encode addressing mode in the opcode**

Generic Examples of Instruction Formats



Summary of Instruction Formats

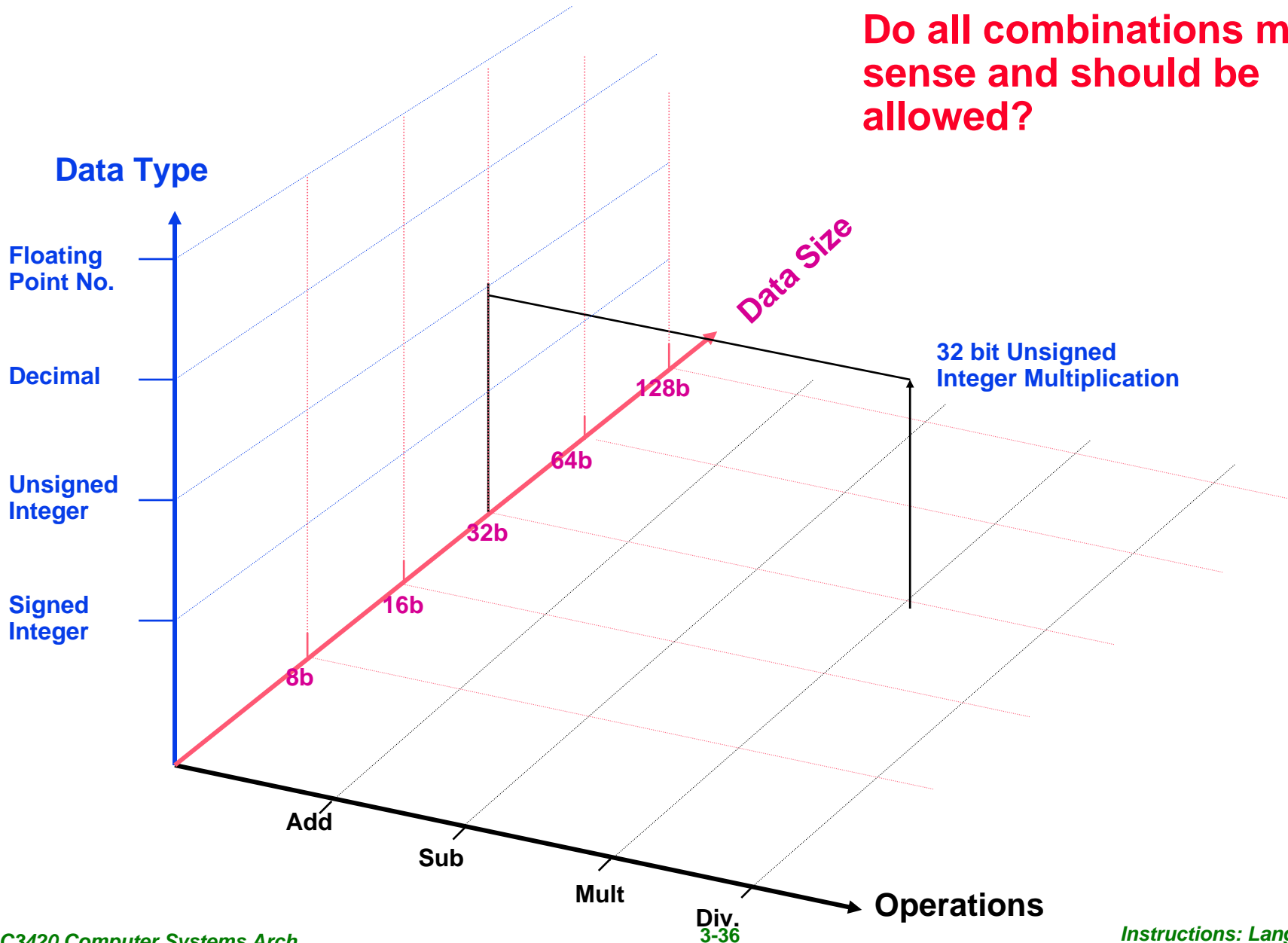
- If **code size is most important**, use variable length instructions
- If **performance is most important**, use fixed length instructions

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**

Ideal Design (orthogonal & completeness)

Do all combinations make sense and should be allowed?



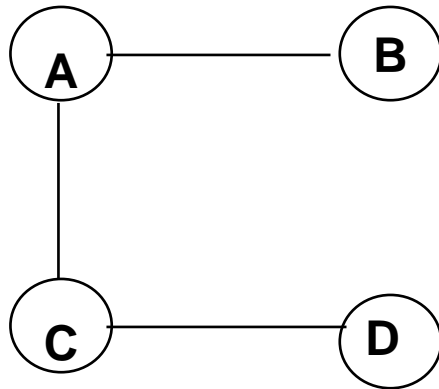
Modern Register Assignment

- Keep arguments and local variables in registers
 - unless they are shared by other programs
- Assign registers by "graph colouring"
- Works well if at least 16 registers

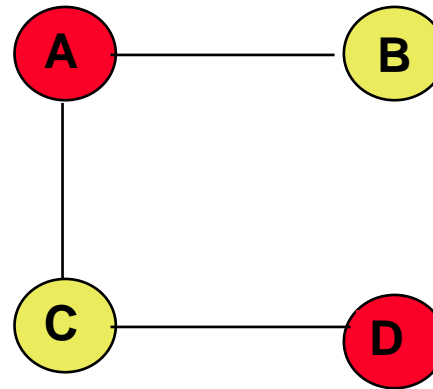
Program Fragment

```
A=  
B=  
.....  
... B ...  
C =  
... A ...  
.....  
D = ...  
... D ...  
....C...
```

Interference graph



Coloured graph



Register Allocated Code

```
R1=  
R2=  
.....  
... R2 ...  
R2 =  
... R1 ...  
.....  
R1 = ...  
... R1 ...  
....R2...
```

Summary of Compiler Considerations

- 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 minimal instruction set.

Instruction Set Metrics

Design-time metrics:

- Can it be implemented, in how long, at what cost?
- Can it be programmed? Ease of compilation?

Static Metrics:

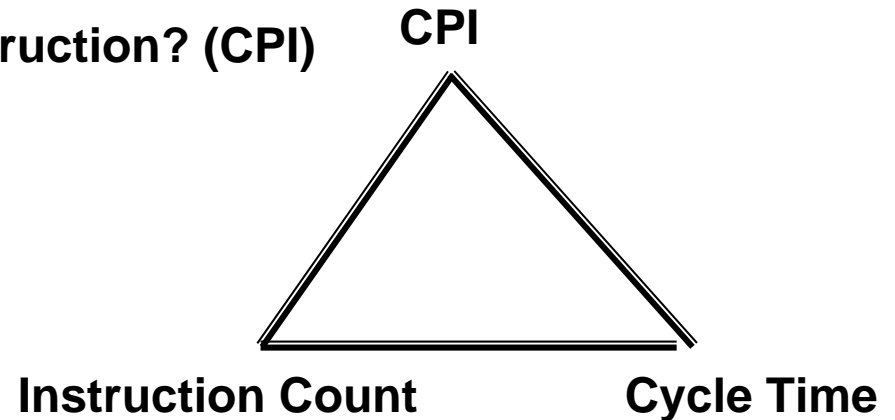
- How many bytes does the program occupy in memory?

Dynamic Metrics:

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

Best Metric: Time to execute the program!

NOTE: this depends on instructions set, processor organisation, and compilation techniques.



ISA: Recap

- Use **general purpose registers** with a **load-store architecture**;
- Support these addressing modes: **displacement** (with an address offset size of 12 to 16 bits), **immediate** (size 8 to 16 bits), and **register deferred**;
- Support and **make** these **simple instructions efficient**, since they will dominate the number of instructions executed:
 - load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch (with a PC-relative address at least 8-bits long), jump, call, and return;
- Support these **data sizes and types**: **8-bit, 16-bit, 32-bit integers** and 64-bit IEEE 754 floating point numbers;
- Use **fixed instruction encoding** if interested in performance and use variable instruction encoding if interested in code size;
- Provide **at least 16 general purpose registers plus separate floating-point registers**, be sure all addressing modes apply to all data transfer instructions, and aim for a minimalist instruction set.

Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

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

- rely on compiler to achieve performance
 - Q. what are the compiler's goals?
- help compiler where we can

MIPS arithmetic

- All instructions have 3 operands
- Operands must be registers, only 32 registers provided
- Operand order in assembly code is fixed (destination first)

Example:

C code: $A = B + C$

MIPS code: `add $s0, $s1, $s2`

$(\$s0 = \$s1 + \$s2)$

- Compiler associates variables with registers
- What about programs with lots of variables?

Memory Organisation

- 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 Organisation

- 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?

Instructions

- Load and store instructions
- Example:

C code: `A[8] = h + A[8];`

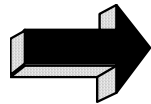
MIPS code: `lw $t0, 32($s3);` `$s3` points to `A`
 `add $t0, $s2, $t0;` `h` in `$s2`
 `sw $t0, 32($s3)`

- Store word has destination last
- Remember **arithmetic operands are registers, not memory !**

Our First Example

- Can we figure out the code?

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



swap:

```
    muli  $2,  $5, 4;   index k in $5  
    add   $2,  $4, $2;  $4 points to v  
    lw   $15, 0($2)  
    lw   $16, 4($2)  
    sw   $16, 0($2)  
    sw   $15, 4($2)  
    jr   $31
```

Machine Language : 32-bit instructions

- **Instructions**, like registers and words of data, are also **32 bits long**
 - **Example:** add **\$t0**, **\$s1**, **\$s2**
- **Registers have numbers**, **\$t0=8**, **\$s1=17**, **\$s2=18**
- **Instruction Format:**

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

- *Not difficult to guess what the field names stand for.*

Machine Language : Load/store

- Consider the load-word and store-word instructions,
 - What would the regularity principle have us do?
 - New principle: Good design demands a compromise
- Introduce a new type of instruction format
 - **I-type** for data transfer instructions
 - other format was **R-type** for register
 - Example: lw **\$t0**, **32**(**\$s2**)



- *Where's the compromise?*

Machine Language : Control (conditional)

- Decision making instructions
 - alter the control flow,
 - i.e., change the "next" instruction to be executed
- MIPS conditional branch instructions:

```
bne $t0, $t1, Label  
beq $t0, $t1, Label
```

- Example: if (i==j) h = i + j;

```
          bne $s0, $s1, Label  
          add $s3, $s0, $s1  
Label: . . . .
```

Machine Language : Control (unconditional)

- MIPS unconditional branch instructions:

```
j label
```

- Example:

```
if (i!=j)                beq $s4, $s5, Lab1
    h=i+j;                add $s3, $s4, $s5
else                       j Lab2
    h=i-j;                Lab1: sub $s3, $s4, $s5
                           Lab2: ...
```

- *Can you build a simple “FOR LOOP” ?*

Addresses in Branches and Jumps

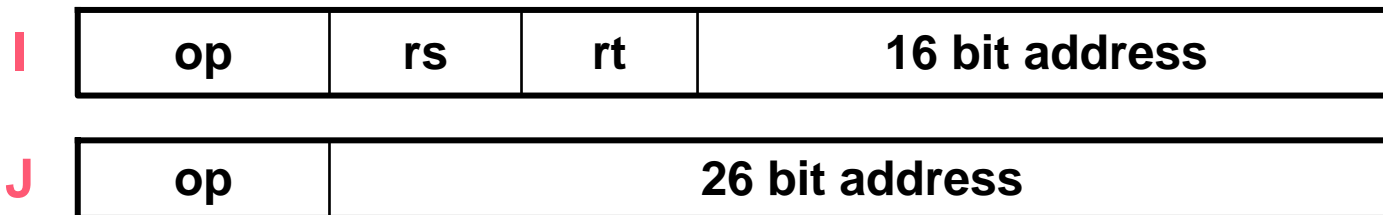
- **Instructions:**

`bne $t4,$t5,Label` Next instruction is at Label if `$t4 <> $t5`

`beq $t4,$t5,Label` Next instruction is at Label if `$t4 = $t5`

`j Label` Next instruction is at Label

- **Formats:**



- **Addresses are not 32 bits**

— How do we handle this with load and store instructions?

Addresses in Branches

- **Instructions:**

`bne $t4,$t5,Label`

Next instruction is at Label if $\$t4 \neq \$t5$

`beq $t4,$t5,Label`

Next instruction is at Label if $\$t4 = \$t5$

- **Formats:**



- **Word address**

- **Could specify a register (like lw and sw) and add it to address**

- use **Instruction Address Register (same as PC = program counter)**
- most **branches are local** (principle of locality)

- **Jump instructions (J format) just use high order bits of PC**

- **26 bit word address => address boundaries of 256 MB**

So far:

- Instruction Meaning

`add $s1,$s2,$s3` `$s1 = $s2 + $s3`

`sub $s1,$s2,$s3` `$s1 = $s2 - $s3`

`lw $s1,100($s2)` `$s1 = Memory[$s2+100]`

`sw $s1,100($s2)` `Memory[$s2+100] = $s1`

`bne $s4,$s5,L` Next instr. is at Label if `$s4 <> $s5`

`beq $s4,$s5,L` Next instr. is at Label if `$s4 = $s5`

`j Label` Next instr. is at Label

- **Formats:**



Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:

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

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

- Can use this instruction to build "b1t \$s1, \$s2, Label"
— can now build general control structures
- Note that the assembler needs a register to do this,
— there are **policy of “use conventions” for registers**

Register Use Conventions

Name	Register number	Usage
\$zero	00	
\$at	01	
\$v0-\$v1	02, 03	
\$a0-\$a3	04 - 07	
\$t0-\$t7	08 - 15	
\$s0-\$s7	16 - 23	
\$t8-\$t9	24, 25	
\$k0-\$k1	26, 27	
\$gp	28	
\$sp	29	

Constants

- Small constants are used quite frequently (50% of operands)

e.g., A = A + 5;
 B = B + 1;
 C = C - 18;

- Solutions? Why not?
 - put 'typical constants' in memory and load them.
 - create hard-wired registers for constants like \$zero (\$one).

- MIPS Instructions:

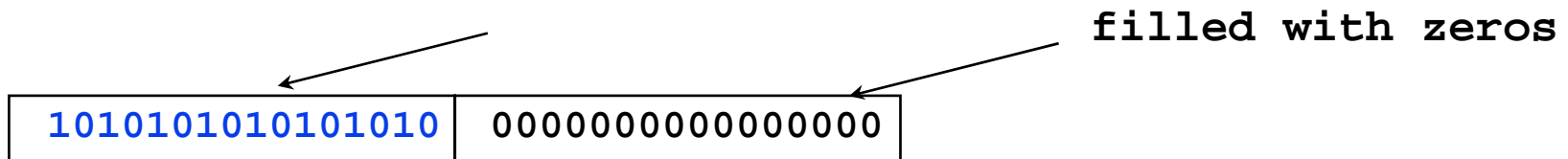
```
addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori  $29, $29, 4
```

- How do we make this work?

How about larger constants?

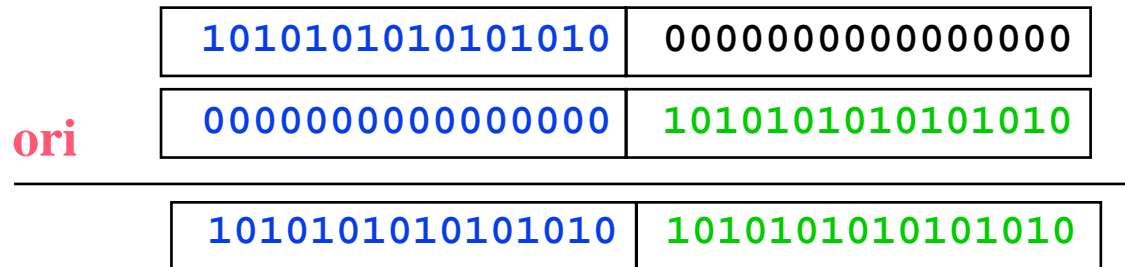
- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```



- Then must get the lower order bits right, i.e.,

```
ori $t0, $t0, 1010101010101010
```



MIPS ISA Summary: MIPS operands

MIPS operands

Name	Example	Comments
32 registers	<code>\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at</code>	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register <code>\$zero</code> always equals 0. Register <code>\$at</code> is reserved for the assembler to handle large constants.
2^{30} memory words	<code>Memory[0], Memory[4], ..., Memory[4294967292]</code>	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 ISA Summary: Assembly language

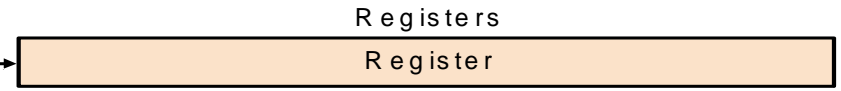
MIPS assembly language

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 ($\$s1 \neq \$s2$) 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	$\$ra = \text{PC} + 4$; go to 10000	For procedure call

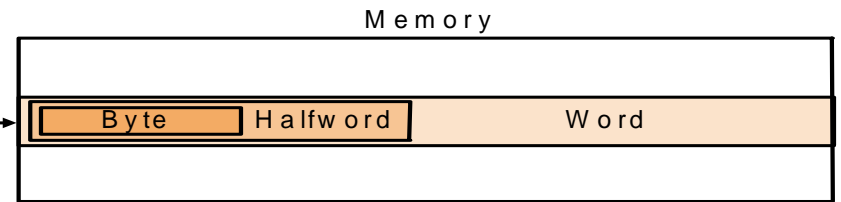
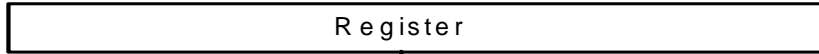
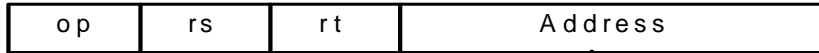
1. Immediate addressing



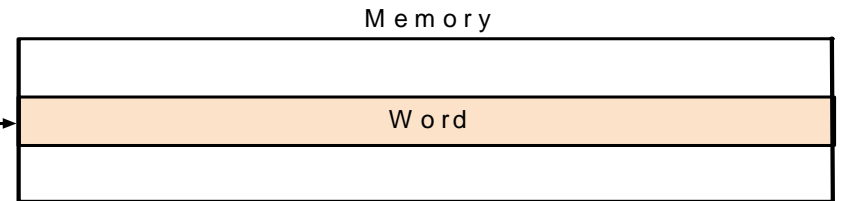
2. Register addressing



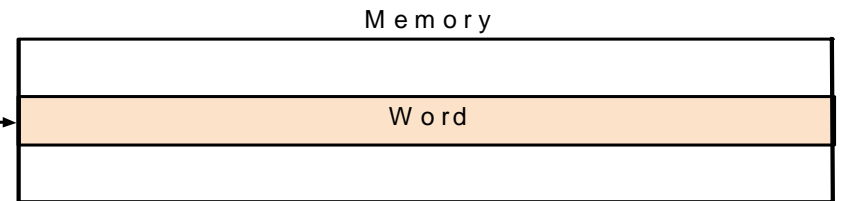
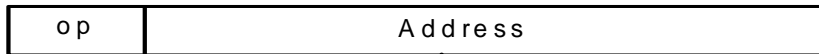
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing



MIPS ISA Summary: data transfer instructions

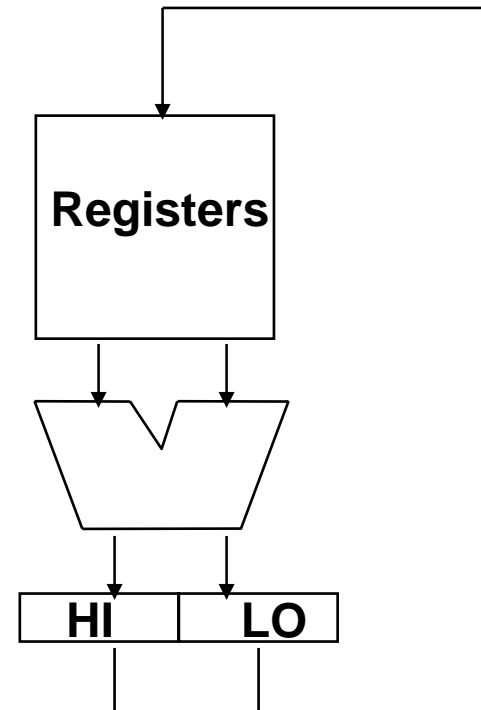
<u><i>Instruction</i></u>	<u><i>Comment</i></u>
SW 500(R4), R3	Store word
SH 502(R2), R3	Store half
SB 41(R3), R2	Store byte
LW R1, 30(R2)	Load word
LH R1, 40(R3)	Load halfword
LHU R1, 40(R3)	Load halfword unsigned
LB R1, 40(R3)	Load byte
LBU R1, 40(R3)	Load byte unsigned
LUI R1, 40	Load Upper Immediate (16 bits shifted left by 16)

MIPS ISA Summary: arithmetic instructions

<i>Instruction</i>	<i>Example</i>	<i>Meaning</i>	<i>Comments</i>
add	add \$1,\$2,\$3	$\$1 = \$2 + \$3$	3 operands; exception possible
subtract	sub \$1,\$2,\$3	$\$1 = \$2 - \$3$	3 operands; exception possible
add immediate	addi \$1,\$2,100	$\$1 = \$2 + 100$	+ constant; exception possible
add unsigned	addu \$1,\$2,\$3	$\$1 = \$2 + \$3$	3 operands; no exceptions
subtract unsigned	subu \$1,\$2,\$3	$\$1 = \$2 - \$3$	3 operands; no exceptions
add imm. unsign.	addiu \$1,\$2,100	$\$1 = \$2 + 100$	+ constant; no exceptions
multiply	mult \$2,\$3	Hi, Lo = $\$2 \times \3	64-bit signed product
multiply unsigned	multu \$2,\$3	Hi, Lo = $\$2 \times \3	64-bit unsigned product
divide	div \$2,\$3	Lo = $\$2 \div \3 , Hi = $\$2 \bmod \3	Lo = quotient, Hi = remainder
divide unsigned	divu \$2,\$3	Lo = $\$2 \div \3 , Hi = $\$2 \bmod \3	Unsigned quotient & remainder
Move from Hi	mfhi \$1	$\$1 = \text{Hi}$	Used to get copy of Hi
Move from Lo	mflo \$1	$\$1 = \text{Lo}$	Used to get copy of Lo

MIPS ISA Summary: Multiply / Divide

- **To perform multiply, divide**
 - MULT rs, rt
 - MULTU rs, rt
 - DIV rs, rt
 - DIVU rs, rt
- **Move result from multiply, divide**
 - MFHI rd
 - MFLO rd
- **Move to HI or LO**
 - MTHI rd
 - MTLO rd



MIPS ISA Summary: logical instructions

<i>Instruction</i>	<i>Example</i>	<i>Meaning</i>	<i>Comment</i>
and	and \$1,\$2,\$3	$\$1 = \$2 \& \$3$	3 reg. operands; Logical AND
or	or \$1,\$2,\$3	$\$1 = \$2 \$3$	3 reg. operands; Logical OR
xor	xor \$1,\$2,\$3	$\$1 = \$2 \oplus \$3$	3 reg. operands; Logical XOR
nor	nor \$1,\$2,\$3	$\$1 = \sim(\$2 \$3)$	3 reg. operands; Logical NOR
and immediate	andi \$1,\$2,10	$\$1 = \$2 \& 10$	Logical AND reg, constant
or immediate	ori \$1,\$2,10	$\$1 = \$2 10$	Logical OR reg, constant
xor immediate	xori \$1, \$2,10	$\$1 = \sim\$2 \& \sim 10$	Logical XOR reg, constant
shift left logical	sll \$1,\$2,10	$\$1 = \$2 \ll 10$	Shift left by constant
shift right logical	srl \$1,\$2,10	$\$1 = \$2 \gg 10$	Shift right by constant
shift right arithm.	sra \$1,\$2,10	$\$1 = \$2 \gg 10$	Shift right (sign extend)
shift left logical	sllv \$1,\$2,\$3	$\$1 = \$2 \ll \$3$	Shift left by variable
shift right logical	srlv \$1,\$2, \$3	$\$1 = \$2 \gg \$3$	Shift right by variable
shift right arithm.	srav \$1,\$2, \$3	$\$1 = \$2 \gg \$3$	Shift right arith. by variable

MIPS ISA Summary: Compare and Branch Formats

- Compare and Branch

- PC-relative branch **BEQ** *rs, rt, offset* if $R[rs] == R[rt]$ then
- **BNE** *rs, rt, offset* \neq

- Compare to zero and Branch

- **BLEZ** *rs, offset* if $R[rs] \leq 0$ then PC-relative branch
- **BGTZ** *rs, offset* $>$
- **BLT** $<$
- **BGEZ** \geq
- **BLTZAL** *rs, offset* if $R[rs] < 0$ then branch and link (into R 31)
- **BGEZAL** \geq

- Remaining set of compare and branch take two instructions

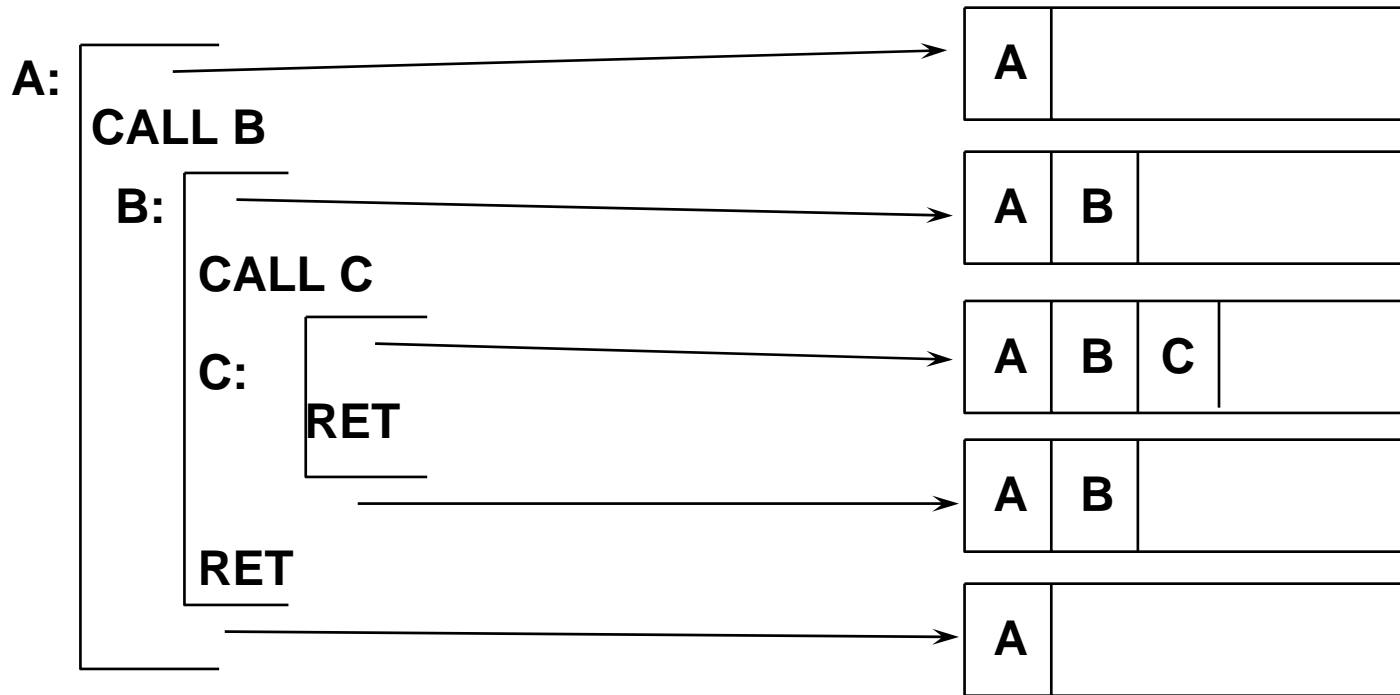
- Almost all comparisons are against zero!

MIPS ISA Summary: jump, branch, compare instructions

<i>Instruction</i>	<i>Example</i>	<i>Meaning</i>
branch on equal	<code>beq \$1,\$2,100</code>	if ($\$1 == \2) go to $PC+4+100$ <i>Equal test; PC relative branch</i>
branch on not eq.	<code>bne \$1,\$2,100</code>	if ($\$1 \neq \2) go to $PC+4+100$ <i>Not equal test; PC relative</i>
set on less than	<code>slt \$1,\$2,\$3</code>	if ($\$2 < \3) $\$1=1$; else $\$1=0$ <i>Compare less than; 2's comp.</i>
set less than imm.	<code>slti \$1,\$2,100</code>	if ($\$2 < 100$) $\$1=1$; else $\$1=0$ <i>Compare < constant; 2's comp.</i>
set less than uns.	<code>sltu \$1,\$2,\$3</code>	if ($\$2 < \3) $\$1=1$; else $\$1=0$ <i>Compare less than; natural no.</i>
set l. t. imm. uns.	<code>sltiu \$1,\$2,100</code>	if ($\$2 < 100$) $\$1=1$; else $\$1=0$ <i>Compare < constant; natural</i>
jump	<code>j 10000</code>	go to 10000 <i>Jump to target address</i>
jump register	<code>jr \$31</code>	go to \$31 <i>For switch, procedure return</i>
jump and link	<code>jal 10000</code>	$\$31 = PC + 4$; go to 10000 <i>For procedure call</i>

MIPS ISA Summary: Stack

Stacking of Subroutine Calls & Returns and Environments:



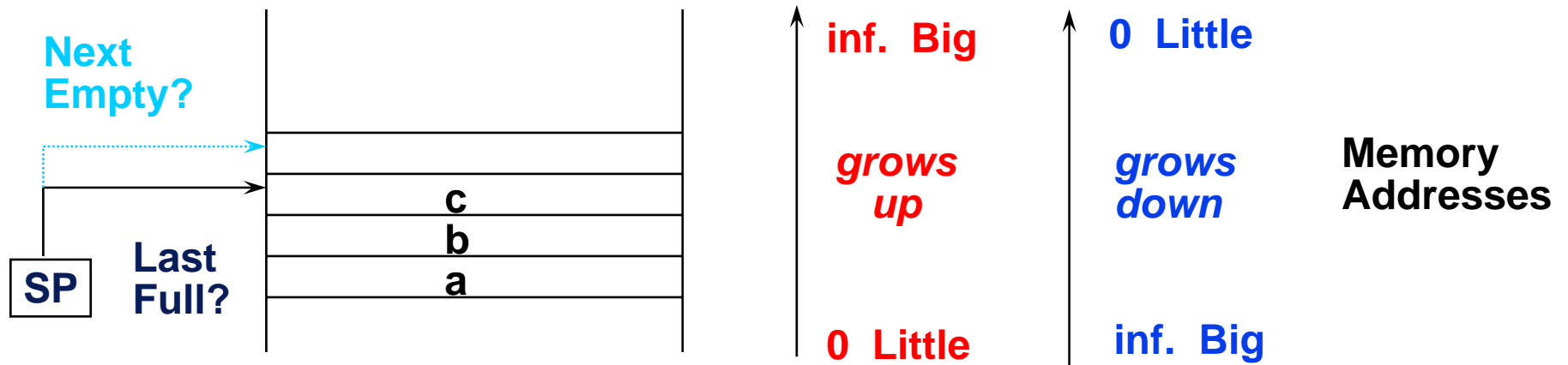
Some machines provide a memory stack as part of the architecture (e.g., the VAX)

Sometimes stacks are implemented via software convention (eg MIPS)

MIPS ISA Summary: Memory Stacks

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

Stacks that Grow Up VERSUS **Stacks that Grow Down:**



Little --> Big /Last Full

POP:

PUSH:

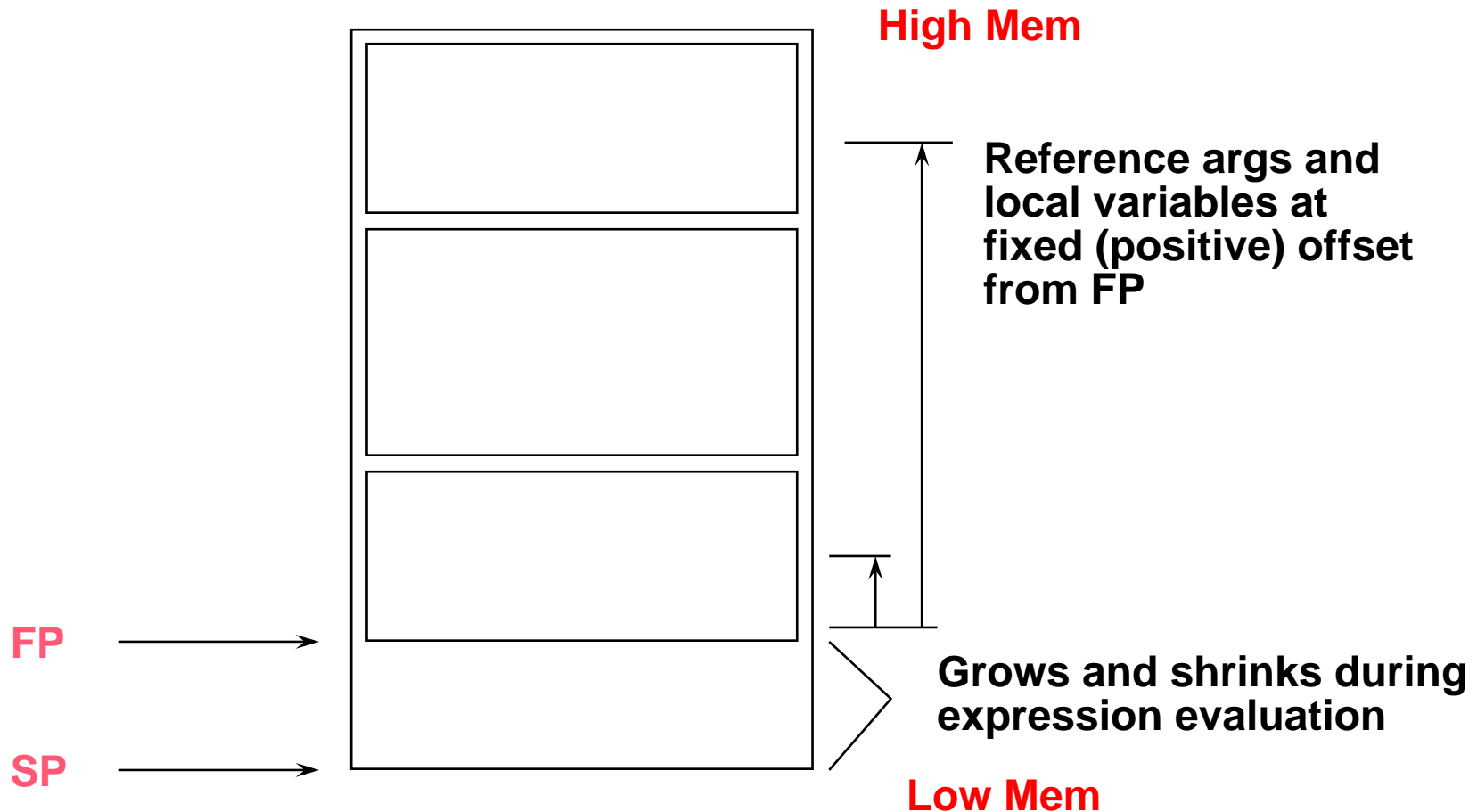
Little --> Big /Next Empty

POP:

PUSH:

How is empty stack represented?

MIPS ISA Summary: Call-Return Linkage: Stack Frames



- Many variations on stacks possible (up/down, last pushed / next)
- Block structured languages contain link to lexically enclosing frame.

MIPS ISA Summary: MIPS / GCC Calling Conventions

fact:

```
addiu $sp, $sp, -32
```

```
sw    $ra, 20($sp)
```

```
sw    $fp, 16($sp)
```

```
addu  $fp, $sp, 32
```

...

```
sw    $a0, 0($fp)
```

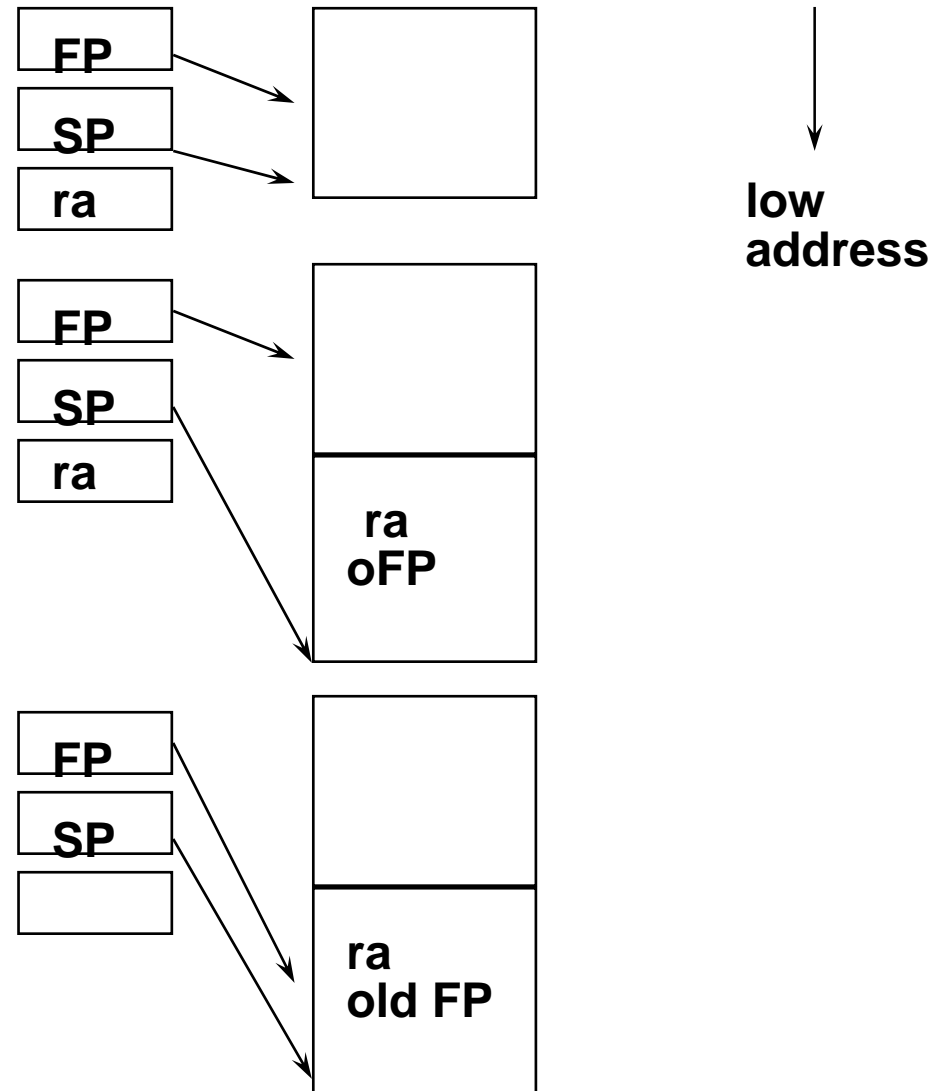
...

```
lw    $31, 20($sp)
```

```
lw    $fp, 16($sp)
```

```
addiu $sp, $sp, 32
```

```
jr    $31
```



First four args passed in registers.

MIPS ISA Summary: Miscellaneous MIPS instructions

- **break** **A breakpoint trap occurs, transfers control to exception handler**
- **syscall** **A system trap occurs, transfers control to exception handler**
- **coprocessor instrs.** **Support for floating point: discussed later**
- **TLB instructions** **Support for virtual memory: discussed later**
- **restore from exception kernel/user** **Restores previous interrupt mask & mode bits into status register**
- **load word left/right** **Supports misaligned word loads**
- **store word left/right** **Supports misaligned word stores**

MIPS ISA Summary: other information

- Register zero always has the value zero (even if you try to write it)
- Branch and jump instructions put the return address PC+4 into the link register
- All instructions change all 32 bits of the destination register (including lui, lb, lh) and all read all 32 bits of sources (add, sub, and, or, ...)
- Immediate arithmetic and logical instructions are extended as follows:
 - logical immediates are zero extended to 32 bits
 - arithmetic immediates are sign extended to 32 bits
- The data loaded by the instructions lb and lh are extended as follows:
 - lbu, lhu are zero extended
 - lb, lh are sign extended
- Overflow can occur in these arithmetic instructions:
 - add, sub, addi
 - overflow cannot occur in addu, subu, addiu, and, or, xor, nor, shifts, mult, multu, div, divu

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**
- Sometimes referred to as **“RISC vs. CISC”**
 - virtually all new instruction sets since 1982 have been RISC
 - VAX: minimise code size, make assembly language easy
instructions from 1 to 54 bytes long!
- We'll talk **very briefly** about other ISAs

Machine Examples: Address & Registers

Intel 8086

2^{20} x 8 bit bytes
AX, BX, CX, DX
SP, BP, SI, DI
CS, SS, DS
IP, Flags

acc, index, count, quot
stack, string
code, stack, data segment

VAX 11

2^{32} x 8 bit bytes
16 x 32 bit GPRs

r15-- program counter
r14-- stack pointer
r13-- frame pointer
r12-- argument ptr

MC 68000

2^{24} x 8 bit bytes
8 x 32 bit GPRs
7 x 32 bit addr reg
1 x 32 bit SP
1 x 32 bit PC

MIPS

2^{32} x 8 bit bytes
32 x 32 bit GPRs
32 x 32 bit FPRs
HI, LO, PC

PowerPC

- **Indexed addressing**

- example: `lw $t1,$a0+$s3` $\#\$t1=Memory[\$a0+\$s3]$
- What do we have to do in MIPS?

- **Update addressing**

- update a register as part of load (for marching through arrays)
- example: `lwu $t0,4($s3)`
 $\#\$t0=Memory[\$s3+4];\$s3=\$s3+4$
- What do we have to do in MIPS?

- **Others:**

- load multiple/store multiple
- a special counter register “bc Loop”

decrement counter, if not 0 goto loop

80x86

- 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: MMX is added
- 2000: Pentium 4; 90 nm technology; L2 cache – 512K to 2 M bytes; up to 3.8 Ghz; System bus: 400, 533, 800 or 1066 MHz

“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”

A dominant architecture: 80x86

- See your textbook for a more detailed description
- **Complexity:**
 - Instructions from 1 to 17 bytes long
 - one operand must act as both a source and destination (2 address instructions)
 - 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 efficient from the right perspective”

VAX11

- **Efficient instruction encoding**
- **Powerful addressing modes**
- **Powerful instructions**
- **Few registers**
- **Simple compiler & small code size**
- **Product of a single talented architect**
- **Announced 1977**
- **Very often quoted as the benchmark**

Concluding remarks

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

Other Issues

- Things we are **not going to cover**
 - support for procedures
 - linkers, loaders, memory layout
 - stacks, frames, recursion
 - manipulating strings and pointers
 - interrupts and exceptions
 - system calls and conventions(To understand more, **read compiler & OS** related materials)
- Some of these we'll talk about later
- We've focused on architectural issues
 - basics of MIPS assembly language and machine code
 - we'll build a processor to execute these instructions.

References:

- **Computer Architecture Case Studies**

Robert J. Baron, Lee Higbie

Addison Wesley

- **Computer Structures: Principles and Examples**

Daniel P. Siewiorek, Gordon Bell, Allen Newell

McGraw Hill, International Student Edition

More Assembly Code Challenge/analysis

LF = 0 ; Given parameter X

For (i=2, i < X, i++)

For (j=i, i*j ≤ X, j++)

If i*j = X then LF = j

- **What is LF ? (with given parameter X)**
- **If LF = 0, what do you know about X?**
- **How to modify this program to find the largest prime number from 0 to limit?**
- **Do you have a more clever algorithm?**
- **Can you write assemble codes for the above routine?**
- **How do the zero, one, two and three (Load & store) architecture assembly codes look like?**