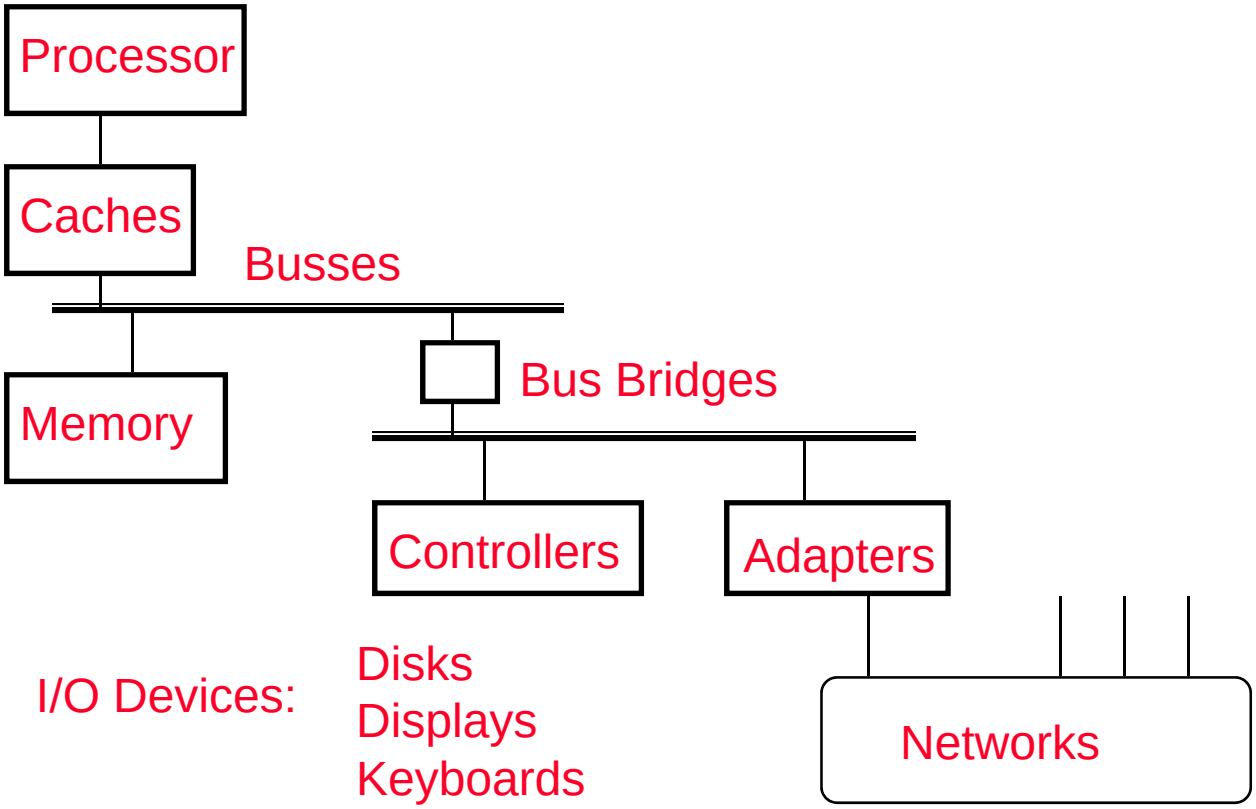
2 Instruction Set Architecture

2.1 ISA Design Considerations Example: MIPS I ISA

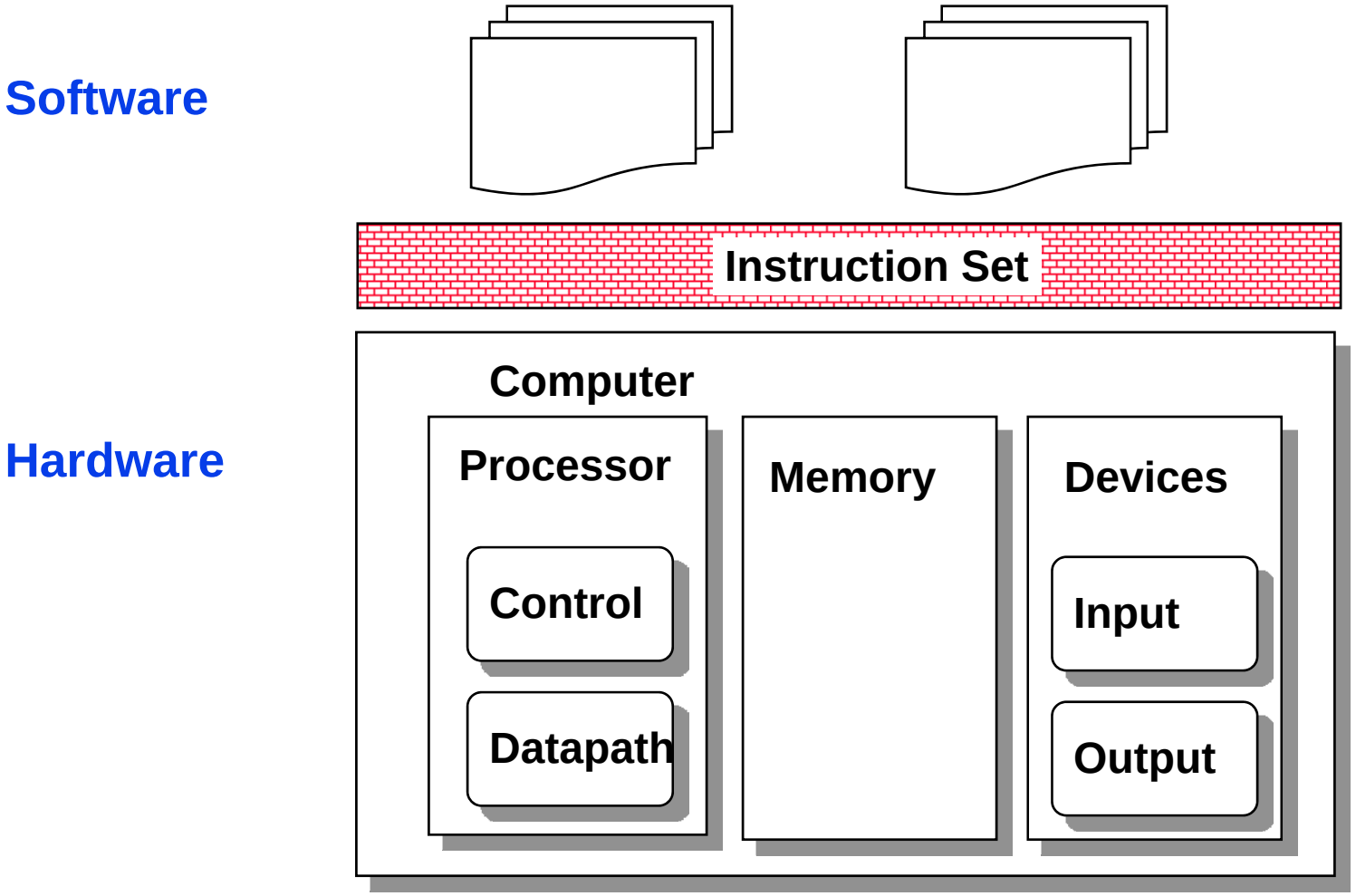
Adapted from course material by D.A. Patterson, J. Kubiatowicz, and R.H. Katz
Course CS 152 („Computer Architecture and Engineering“)
University of California at Berkeley, Fall 1997–2002

Review: Computer System Components



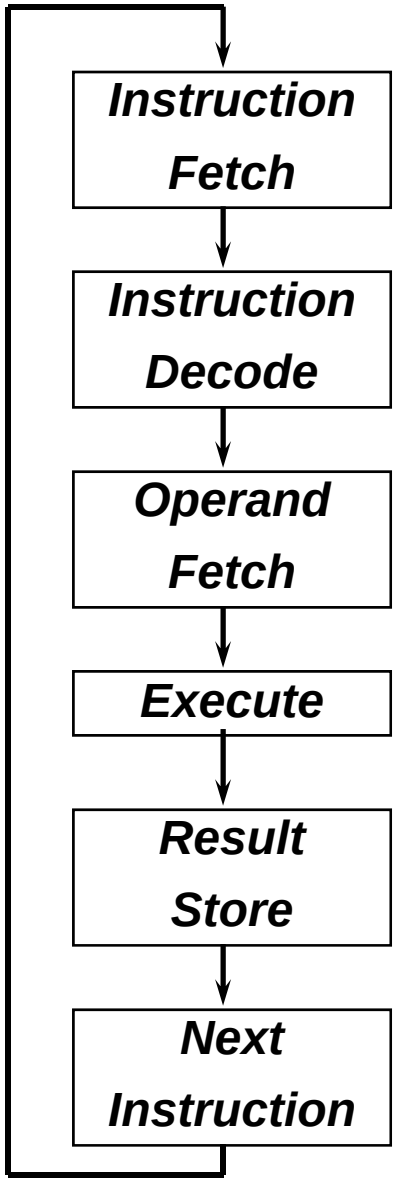
All components have interfaces and organizations.

Review: Instruction Set Design



Which is easier to design / change - interface or organization ?

Instruction Set Architecture: What Must be Specified?



- Instruction format or encoding
  - How is it decoded?
- Location of instructions, operands and result
  - Where other than memory? Registers?
  - How are they addressed?
  - How many explicit operands?
  - How are memory operands located?
  - Which can or cannot be in memory?
- Data type
- Data size
- Operations
  - Which ones are supported?
- Successor instruction
  - Jumps, conditions, branches
  - *Fetch-decode-execute is implicit!*

ISA: Essential Characteristics

- Instruction and data formats
- Instruction types
- Number of instructions
- Number of addresses and memory operands per instruction
- Addressing modes

[Design principles for ISA and hardware:

1. Simplicity favors regularity
2. Smaller is faster
3. Good design demands good compromises
4. Make the common case fast]

Basic ISA Classes: Number of Addresses in Instruction

1. Stack machine (0-address instructions):

0 addresses	`add`	tos ← tos + next	**Top of Stack**

2. Accumulator machine (1-address instructions):

1 address	`add A`	acc ← acc + mem[A]
1+x address	`addx A`	acc ← acc + mem[A + x]

3. General Purpose Register- (GPR-) based machine (2/3-address instructions; can be memory/memory):

2 addresses	`add A B`	EA[A] ← EA[A] + EA[B]	**Effective Addr.**
3 addresses	`add A B C`	EA[A] ← EA[B] + EA[C]	

4. Load/Store-based machine (3-address ALU instructions):

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

Most real machines are hybrids of these.

Basic ISA Classes: Comparison

Comparison by what?

- Bytes per instruction? → Code size
- Number of instructions? → Code size, run time
- Cycles per instruction? → Run time

Example:

Comparison in terms of number of instructions on code sequence C = A + B:

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

Basic ISA Classes: Status and Example

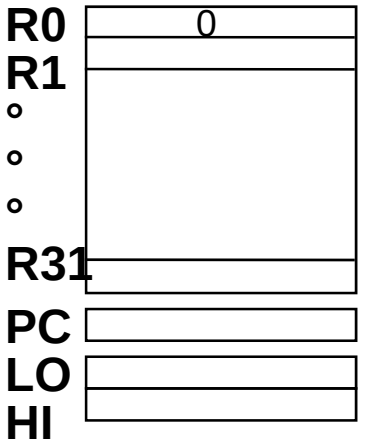
1975 – today: Most real machines use general purpose registers.

Advantages of registers:

- **Registers are faster than memory.**
- **Registers are easier for a compiler to use, e.g. compared to stack:**
  - **Example: (A*B) – (C*D) – (E*F):**
    MULTs can be done in any order.
- **Registers can hold variables:**
  - **Memory traffic is reduced, so program is sped up (since registers are faster than memory).**
  - **Code density improves (since register named with fewer bits than memory location).**

Example: MIPS Registers / Programmable Storage

- 2^{32} bytes of memory
- 31 x 32-bit general-purpose (GP) regs. (R0 = 0)
- 32 x 32-bit FP regs. (paired: double precision)
- PC; HI, LO
- GP register naming convention (for now):



0	zero	Constant 0
1	at	Reserved for assembler
2	v0	Expression evaluation
3	v1	& subrout. return values
4	a0	
5	a1	Argument registers
6	a2	
7	a3	
8	t0	
. . .		Temporary registers
15	t7	
16	s0	
. . .		Regs. (e.g. for vars.)
23	s7	
24	t8	Temp. regs. (cont'd.)
25	t9	
26	k0	Reserved for OS kernel
27	k1	
28	gp	Pointer to global area
29	sp	Stack pointer
30	fp	Frame pointer
31	ra	Return address (HW)

MIPS Assembly Example: Using Registers

C assignment statement:

```
f = (g + h) – (i + j);
```

MIPS assembly code (portion):

```
add   $t0, $s1, $s2      # assume reg. $s1 contains g, $s2 h
                         # add: temp. reg. $t0 gets sum g+h
add   $t1, $s3, $s4      # assume $s3 contains i, $s4 j
                         # add: $t1 gets sum i+j
sub   $s0, $t0, $t1      # sub(tract): $s0 gets result $t0–$t1
```

Memory Addressing

Since 1980 almost every machine uses addresses to level of 8-bits: byte addressing

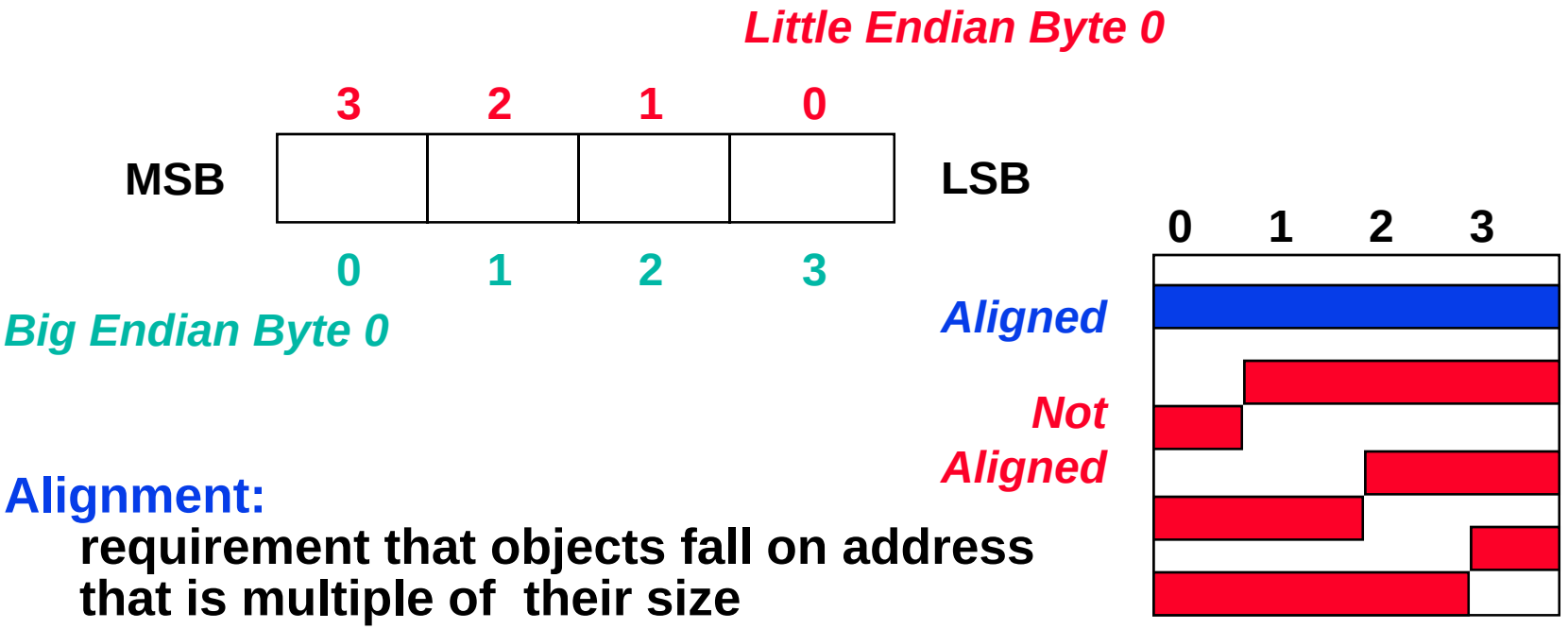
2 questions for design of ISA:

- **Since a 32-bit word could be read 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?**

Memory Addressing: Endianess and Alignment

- **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, DEC Alpha (Windows NT)



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

Addressing Modes

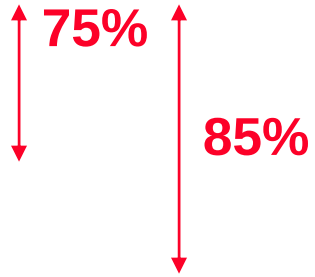
Addressing mode	Example	Meaning
Register	Add R4,R3	R4← R4+R3
Immediate	Add R4,#3	R4 ← R4+3
Displacement	Add R4,100(R1)	R4 ← R4+Mem[100+R1]
Register indirect	Add R4,(R1)	R4 ← R4+Mem[R1]
Indexed / Base	Add R3,(R1+R2)	R3 ← R3+Mem[R1+R2]
Direct or absolute	Add R1,(1001)	R1 ← R1+Mem[1001]
Memory indirect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

Addressing Modes: Usage?

Usage of addressing modes:

3 programs measured on machine with all addressing modes (DEC VAX), register mode ignored:

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



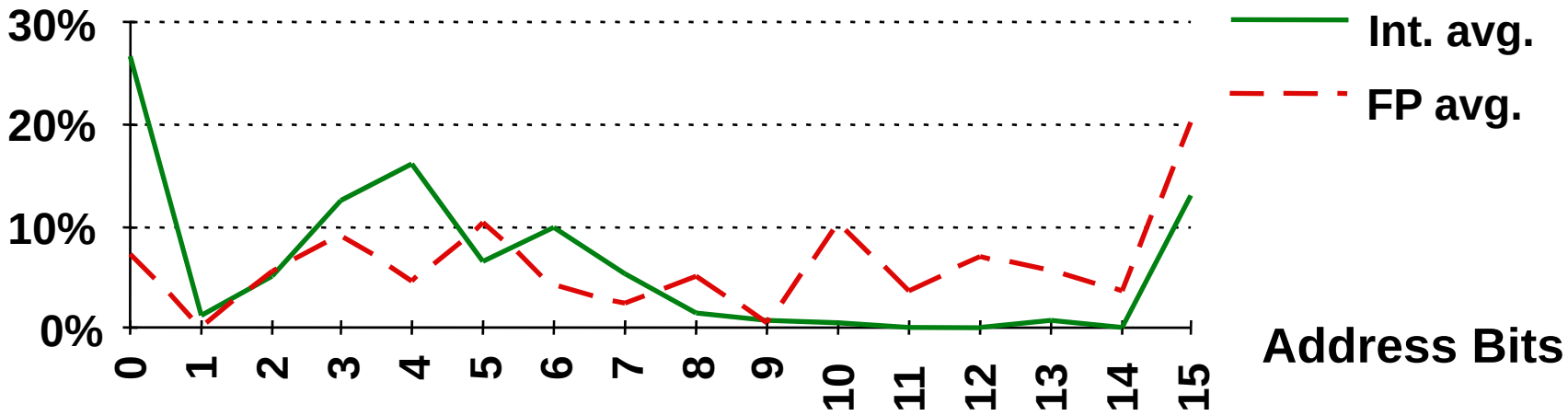
Summary:

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

Addressing Modes: Displacement and Immediate Size?

Size of displacement address:

5 SPECint92 programs, 5 SPECfp92 programs measured



- 1% of addresses > 16 bits
- 12 - 16 bits of displacement needed

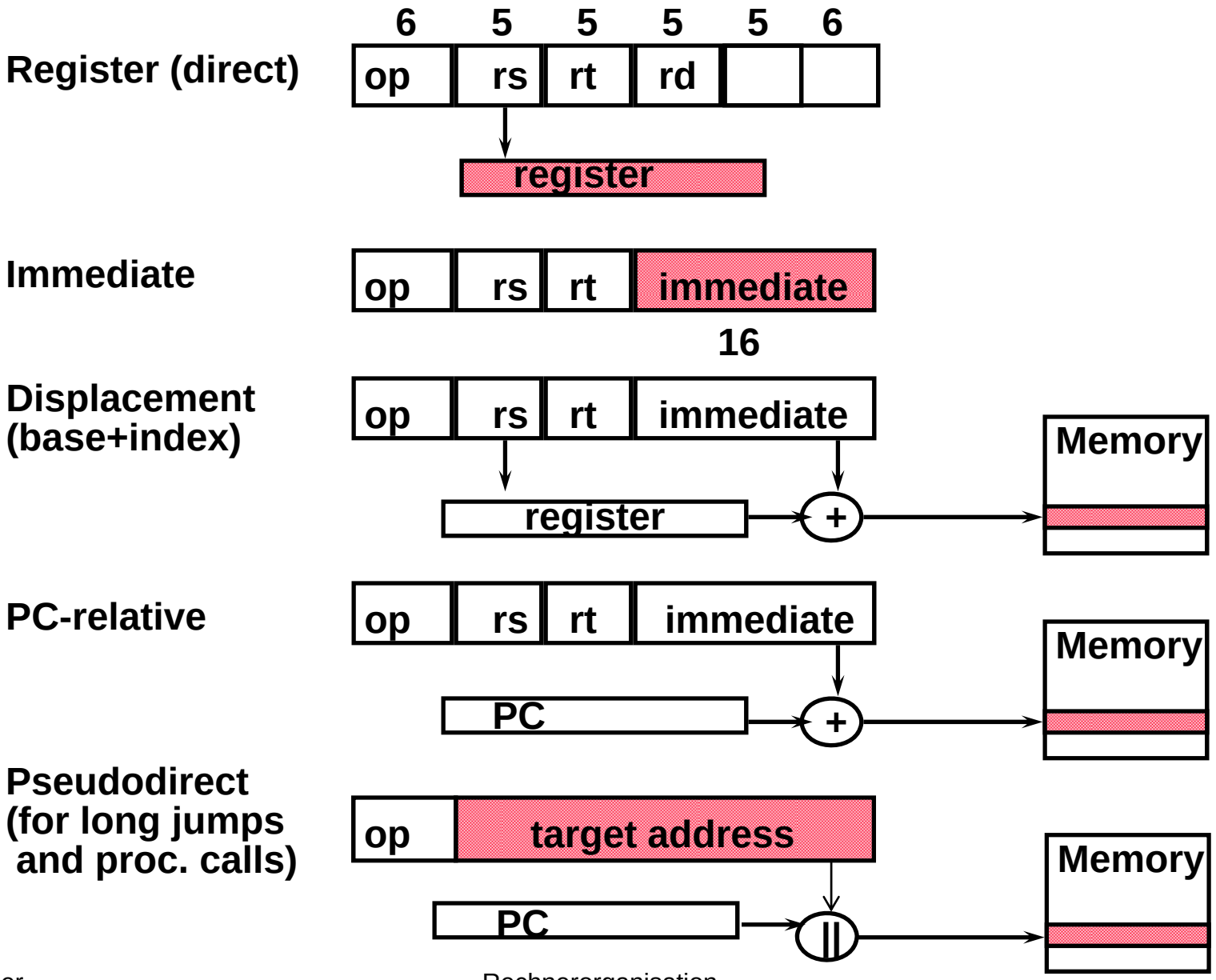
Size of immediate data:

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

Addressing Modes: Summary

- **Data addressing modes that are important:**
  - **Displacement**
  - **Immediate**
  - **Register Indirect**
- **Displacement size should be 12 to 16 bits**
- **Immediate size should be 8 to 16 bits**

Example: MIPS Addressing Modes



MIPS Assembly Example: Addressing Memory

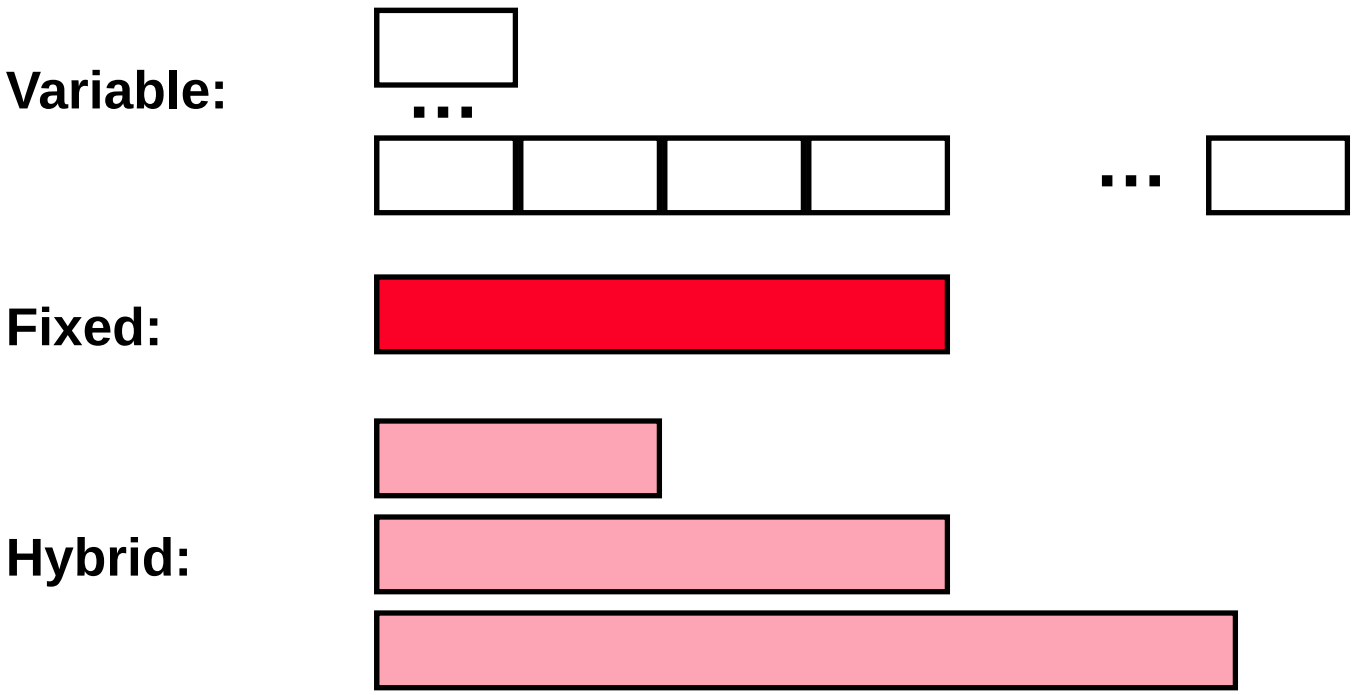
C assignment including loading and storing array elements:

```
a[12] = h + a[8];        /* assume h and a[] are integers (4 bytes) */
```

MIPS assembly code (portion):

```
lw    $t0, 32($s3)     # assume $s3 contains base addr. of a (a[0])
                       # load word: $t0 gets a[8]   (displacement a.)
add   $t0, $s2, $t0    # assume $s2 contains h
                       # add: $t0 gets h+a[8]
sw    $t0, 48($s3)     # store word: result into a[12]
```

Instruction Formats: Instruction Width Considerations



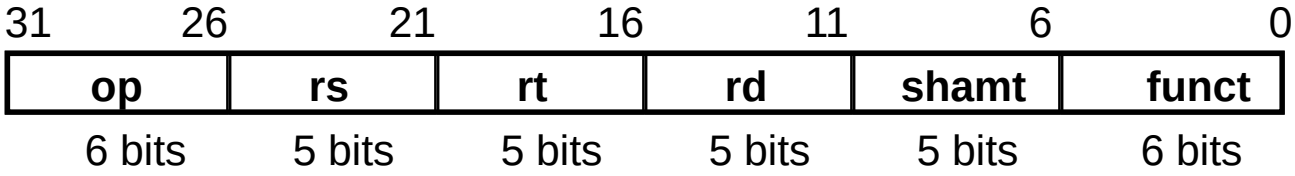
- If **code size** is most important,
  variable length instructions should be used

- If **performance** is most important,
  fixed length instructions should be used

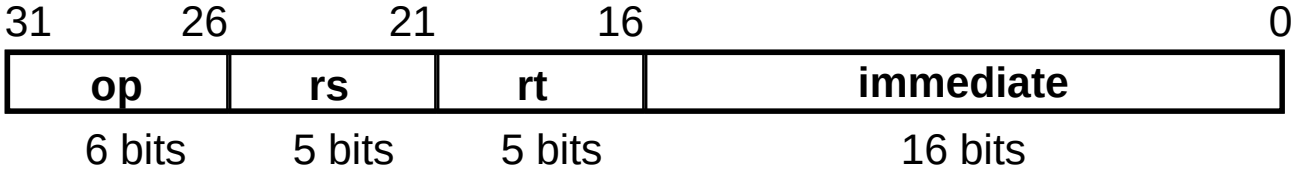
Example: MIPS Instruction Formats

All instructions are 32 bits wide. Three instruction formats:

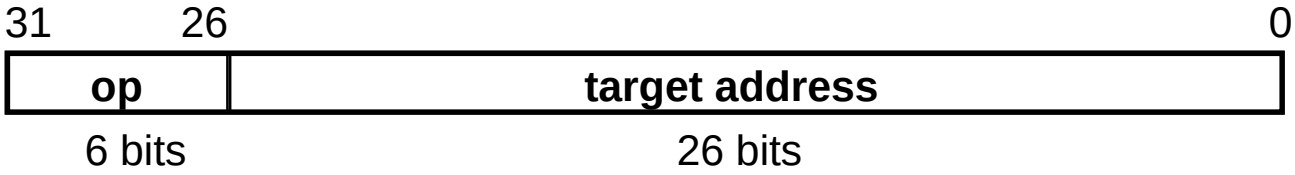
- **R-type**



- **I-type**



- **J-type**



Different fields:

- **op: operation of the instruction *(opcode)***
- **rs, rt, rd: the source and destination register specifiers**
- **shamt: shift amount**
- **funct: selects the variant of the operation in the “op” field**
- **address / immediate: address offset or immediate value**
- **target address: target address of the jump instruction**

Instruction Types: Typical Operations

Data Movement	**load (from memory)** **store (to memory)** **memory-to-memory move** **register-to-register move** **input (from I/O device)** **output (to I/O device)** **push, pop (to/from stack)**
Arithmetic	**integer (binary + decimal) or FP** **add, subtract, multiply, divide**
Shift	**shift left/right, rotate left/right**
Logical	**not, and, or, set, clear**
Control (Jump/Branch)	**unconditional, conditional**
Subroutine Linkage	**call, return**
Interrupt	**trap, return**
Synchronization	**test&set (atomic read-modify-write)**
String	**search, translate**
Graphics (MMX)	**parallel subword ops (4 16bit add)**

Instruction Types: Top Ten 80x86 Instructions

Rank	Instruction	Ratio (of instructions 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.

Support these simple instructions (and a few others, e.g., shift, branch)!

MIPS Assembly Example: Conditional Branch (1)

C if statement:

```
        if (i == j) go to L1;
        f = g + h;                  /* if  (i ≠ j)  f = g+h; */
L1:     f = f – i;
```

MIPS assembly code (portion):

```
        beq   $s3, $s4, L1      # assume $s0 .. $s4 contain vars. f, g, h, i, j
                                # branch if equal (i and j)
        add   $s0, $s1, $s2     # if  (i ≠ j)  f = g+h;
L1:  sub   $s0, $s0, $s3        # f = f – i; (always executed)
```

MIPS Assembly Example: Conditional Branch (2)

C if-then-else statement:

```
if (i == j)
  f = g + h;
  else  f = g – h;
```

MIPS assembly code (portion):

```
      bne   $s3, $s4, Else    # go to else branch if i ≠ j
                              # branch if not equal
      add   $s0, $s1, $s2     # ‘then’ branch:  f = g + h
      j     Exit              # leave ‘then’ branch unconditionally
                              # jump
Else: sub   $s0, $s1, $s2     # ‘else’ branch: f = g – h
Exit:
```

Data Types

Bit: 0, 1

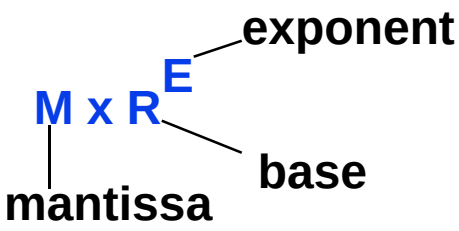
Bit String: sequence of bits of a particular length
- 4 bits is a nibble
- 8 bits is a byte
- 16 bits is a half-word
- 32 bits is a word
- 64 bits is a double-word

Character:
- ASCII 7-bit code
- UNICODE 16-bit code

Decimal:
- Digits 0-9 encoded as 0000b – 1001b
- Two decimal digits packed per 8-bit byte

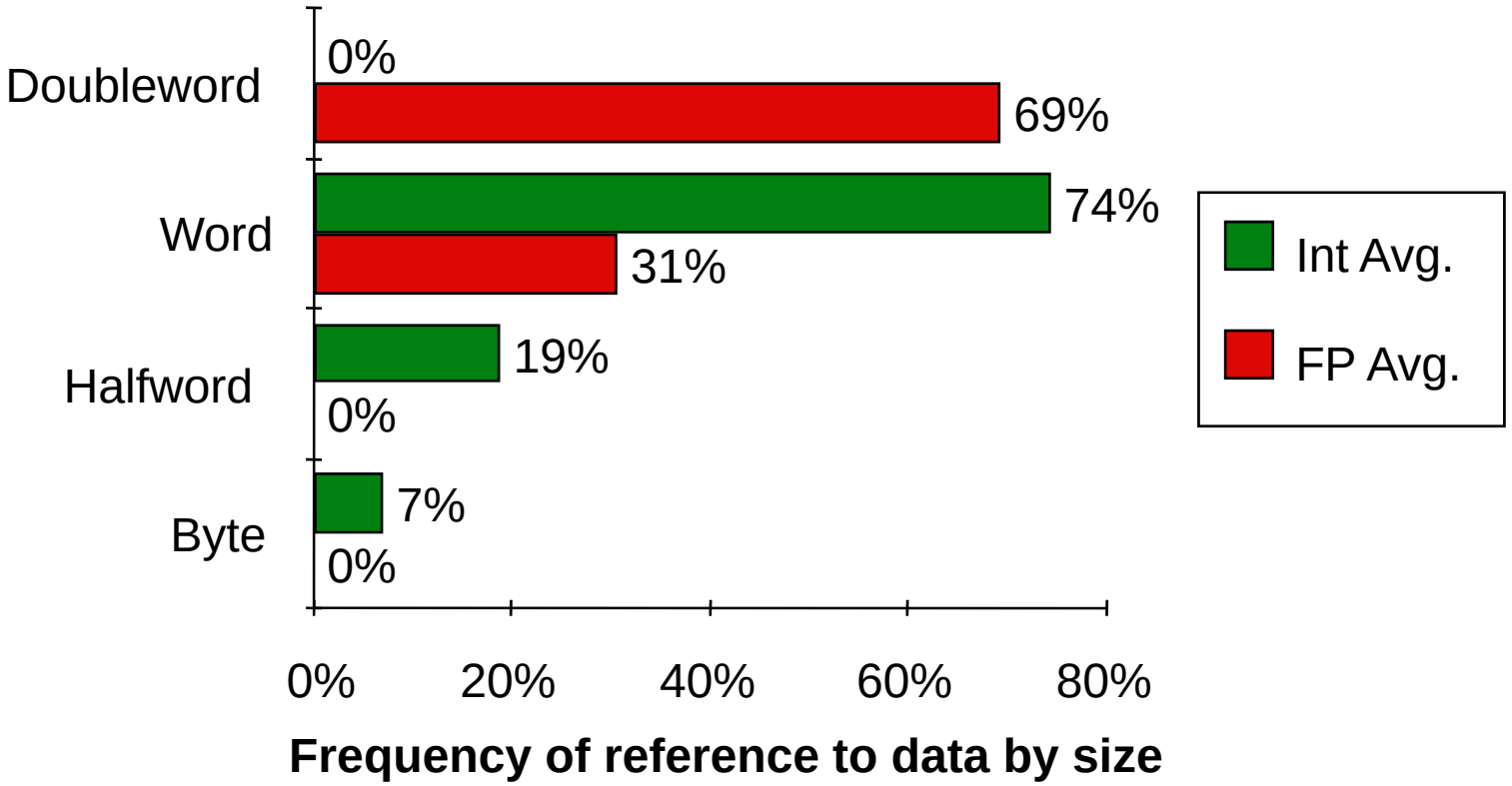
Integers:
- 2's complement

Floating Point:
- Single Precision
- Double Precision
- Extended Precision



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

Operand Size Usage



Support these data sizes and types:

- **8-bit, 16-bit, and 32-bit integers**
- **32-bit and 64-bit IEEE 754 floating-point (FP) numbers**

Instruction Set Architecture: Compiler Considerations

Support 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**
- **Simplicity: resource needs easily determined**

Support register assignment:

- **Easier if many registers**

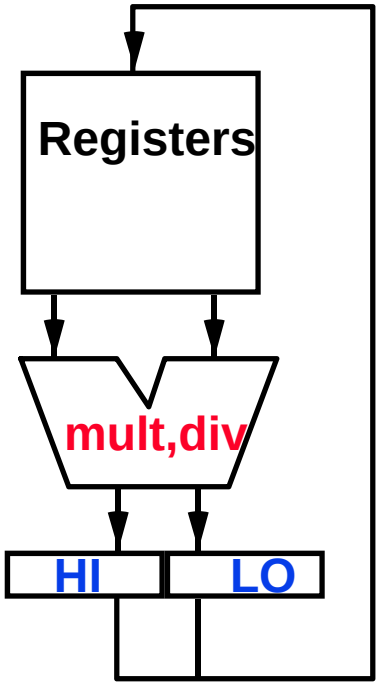
→ Provide at least 16 general purpose registers, plus separate floating-point registers

→ Make sure all addressing modes apply to all data transfer instructions

→ Aim for a minimalist instruction set

Example: MIPS Operation Overview (I)

- **Arithmetic & Logical Operations:**
  - **Arithmetic ops:** add, addu, sub, subu; addi, addiu; mult, multu, div, divu
  - **Logical ops:** and, or, xor, nor; andi, ori, xori
  - **Shift ops:** sll, srl, sra, sllv, srlv, srav
- **Memory Access:**
  - **Loads:** lb, lbu, lh, lhu, lw, lwl, lwr; lwcz
  - **Stores:** sb, sh, sw, swl, swr; swcz
- **Data Transfer:**
  - mfhi, mflo; mthi, mtlo; mfcz, mtcz
- **Constant Manipulation:**
  - lui



Example: MIPS Arithmetic Instructions

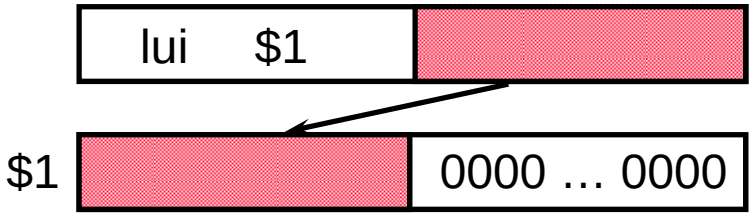
Instruction	Example	Meaning	Comments
add	add $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands; exception possible
add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; no exceptions
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 x $3	64-bit signed product
multiply unsigned	multu $2,$3	Hi, Lo = $2 x $3	64-bit unsigned product
divide	div $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Lo = quotient, Hi = remainder
divide unsigned	divu $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Unsigned quotient & remainder
move from Hi	mfhi $1	$1 = Hi	Used to get copy of Hi
move from Lo	mflo $1	$1 = Lo	Used to get copy of Lo

Example: MIPS Logical Instructions

Instruction	Example	Meaning	Comment_
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 ⊕ $3	3 reg. operands; Logical XOR
nor	nor $1,$2,$3	$1 = ~($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 = ~$2 &~10	Logical XOR reg, constant
shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
shift right arithm.	sra $1,$2,10	$1 = $2 >> 10	Shift right (sign extend)
shift left logical	sllv $1,$2,$3	$1 = $2 << $3	Shift left by variable
shift right logical	srlv $1,$2, $3	$1 = $2 >> $3	Shift right by variable
shift right arithm.	srav $1,$2, $3	$1 = $2 >> $3	Shift right arith. by variable

Example: MIPS Data Transfer Instructions

Instruction	Comment
sw $3, 500($4)	Store word
sh $3, 502($2)	Store halfword
sb $2, 41($3)	Store byte
lw $1, 30($2)	Load word
lh $1, 40($3)	Load halfword
lhu $1, 40($3)	Load halfword unsigned
lb $1, 40($3)	Load byte
lbu $1, 40($3)	Load byte unsigned
lui $1, 40	Load Upper Immediate (16 bits shifted left by 16)



Control Transfer: Methods of Testing Conditions

- **Condition Codes**

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

  Example:
  ```
  add r1, r2, r3
  bz label
  ```

- **Condition Register**

  Register is set as a result of explicit compare or test instruction.

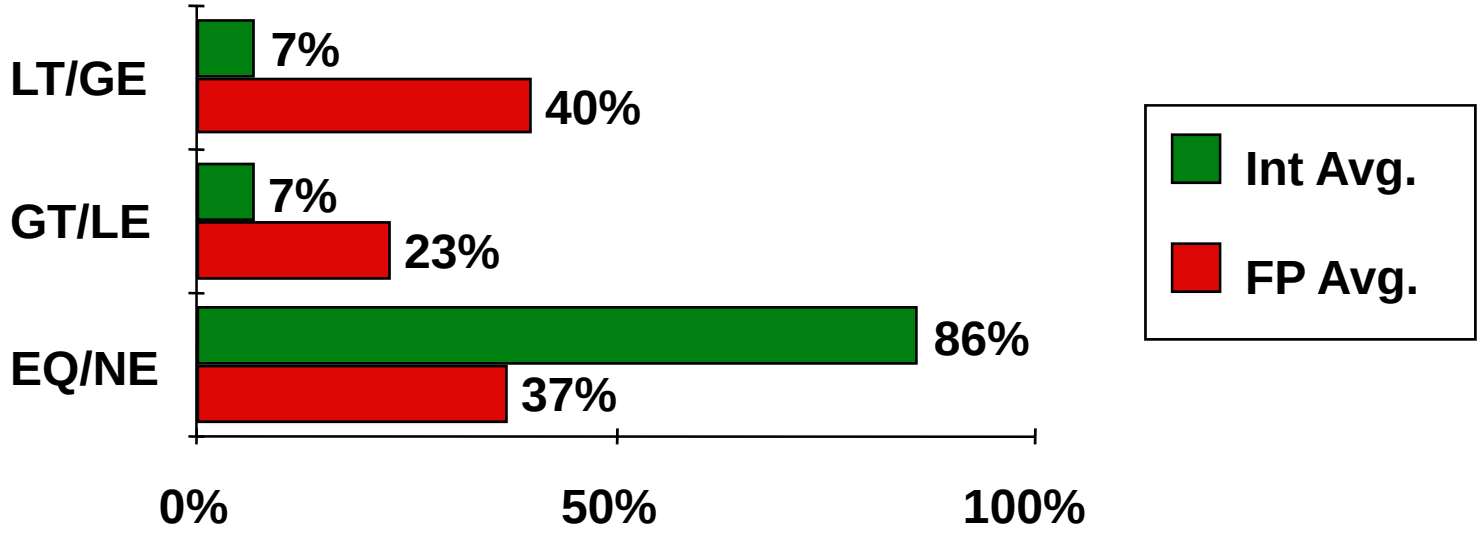
  Example:
  ```
  cmp r1, r2, r3
  bgt r1, label
  ```

- **Compare and Branch**

  Separate compare/test & branch instructions for various conditions.

  Example:
  ```
  bgt r1, r2, label
  ```

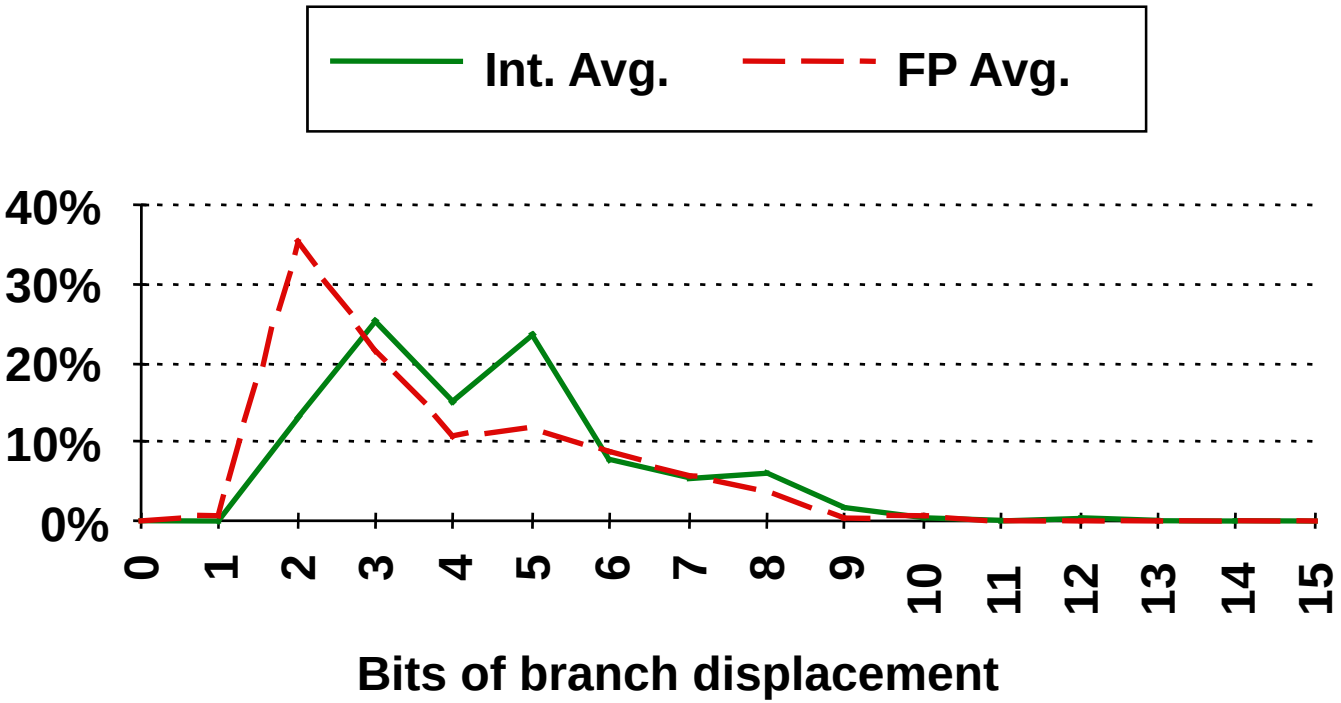
Control Transfer: Comparison Types



Frequency of comparison types in branches

- Compare Equal / Not Equal most important for integer programs (86%)
- Almost all comparisons are against zero

Control Transfer: Conditional Branch Distance



- Branches should be taken PC-relative since most are relatively close to the current PC
- ≥ 8 bits for branch displacement suggested (± 128 instructions)

Example: MIPS Operation Overview (II)

- **Compare and Branch:** beq, bne
- **Compare Coprocessor Condition Flag and Branch:** bczt, bczf
- **Compare to Zero and Branch:**
  - Cond. branch: blez, bltz; bgez, bgtz
  - Cond. branch and save return address ("link"): bltzal, bgezal

(Remaining set of compare and branch ops take two instructions.)

- **Comparison:** slt, sltu; slti, sltiu ("set on less than")
- **Jump:**
  - Jump: j, jr
  - Jump and save return address ("link"): jal, jalr
- **Floating Point ("Coprocessor") Instructions**

Example: MIPS Compare, Branch, Jump Instructions

Instruction	Example	Meaning
branch on equal	beq $1,$2,100	if ($1 == $2) go to PC+4+100 *Equal test; PC relative branch*
..... (bne, bczt, bczf)		
branch on less than or equal zero	blez $1,100	if ($1<= 0) go to PC+4+100 *Compare to 0; PC relative branch*
..... (bltz, bgez, bgtz)		
set on less than	slt $1,$2,$3	if ($2 < $3) $1=1; else $1=0 *Compare less than; 2's complement*
set less than uns.	sltu $1,$2,$3	if ($2 < $3) $1=1; else $1=0 *Compare less than; natural numbers*
..... (slti, sltiu)		
jump	j 10000	go to 10000 *Jump to target address*
jump register	jr $31	go to $31 *For switch/case and procedure return*
jump and link	jal 10000	$31 = PC + 4; go to 10000 *For procedure call*
..... (jalr, bltzal, bgezal)		

MIPS Assembly Example: Loop with Array Index

C loop with array index:

```
Loop:   g = g + a[i];
        i = i + j;
        if (i != h) goto Loop;
```

MIPS assembly code (portion):

```
                                # assume $s1 .. $s4 contain vars. g, h, i, j
Loop:   add $t1, $s3, $s3       # compute array offset: $t1 = 2 * i
        add $t1, $t1, $t1       # $t1 = 4 * i
        add $t1, $t1, $s5       # $t1 gets address of a[i]  ($s5: base)
        lw  $t0, 0($t1)         # $t0 gets a[i]
        add $s1, $s1, $t0       # g = g + a[i]
        add $s3, $s3, $s4       # i = i + j
        bne $s3, $s2, Loop      # go to Loop if i ≠ h
```

MIPS Assembly Example: While Loop

C while loop:

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

MIPS assembly code (portion):

```
                                # assume $s3, $s4, $s5 contain vars. i, j, k
Loop:   add $t1, $s3, $s3       # $t1 = 2 * i
        add $t1, $t1, $t1       # $t1 = 4 * i
        add $t1, $t1, $s6       # $t1: address of save[i]  ($s6: base)
        lw  $t0, 0($t1)         # $t0 gets save[i]
        bne $t0, $s5, Exit      # go to Exit if save[i] ≠ k
        add $s3, $s3, $s4       # i = i + j
        j   Loop                # go to Loop
Exit:
```

or: sll $t1, $s3, 2

MIPS Assembly Example: Less-Than Test

C less than test:

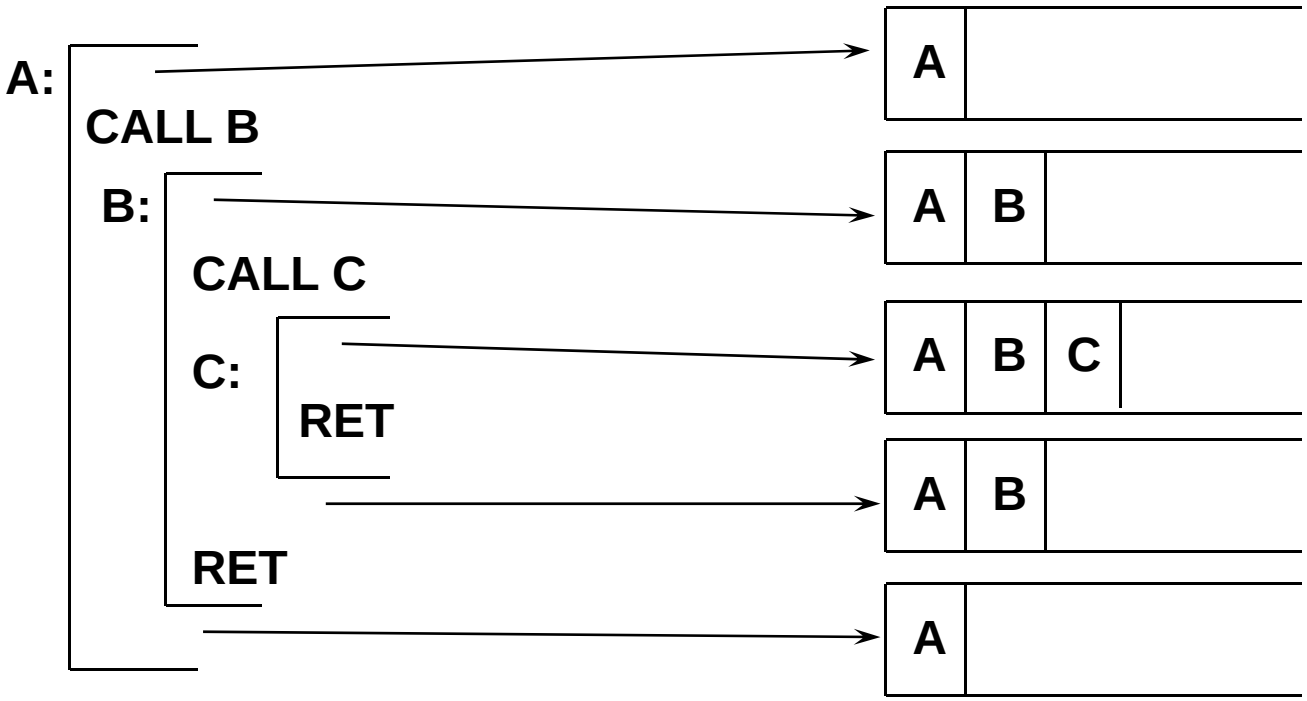
```
if (a < b)
    ..... ;      /* 'less' branch */
```

MIPS assembly code (portion):

```
                                # assume $s0, $s1 contain a, b, respect.
        slt   $t0, $s0, $s1     # $t0 gets 1 if $s0 < $s1 (a < b)
                                # set on less than
        bne $t0, $zero, Less    # go to Less if $t0 ≠ 0 (that is, if a < b)
        .....
Less: .....                     # 'less' branch
```

Subroutine Calls and Stacks

Stacking of Subroutine Calls & Returns and Environments:



Support for procedures in hardware, by registers and memory stacks:

- **Some machines provide a memory stack as part of the architecture (e.g., DEC VAX)**
- **Sometimes stacks are implemented via software convention (e.g., MIPS)**

Subroutine Execution

Steps:

Caller

- Place subroutine parameters (arguments) in a place where the subroutine can access them (registers or/and memory).
- Transfer control to the subroutine.

Callee

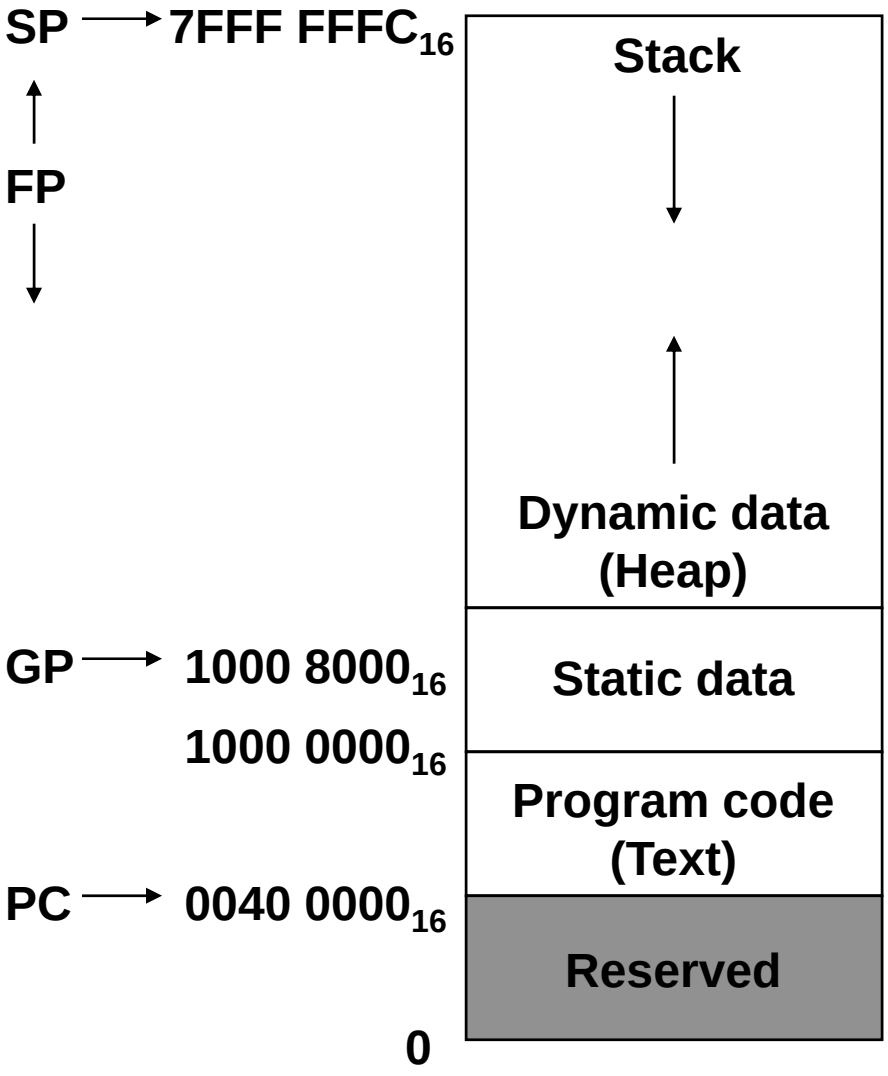
- Acquire the storage resources needed for the subroutine.
- Perform the desired task.
- Place the result value(s) in a place where the calling program can access it (them).
- Return control to the point of origin.

Subroutine Execution

Required:

- **Calling conventions for caller and callee, adhered to by:**
  - **Compiler or**
  - **Assembly language programmer**
- **Support by the hardware:**
  - **Instructions for subroutine call and return**
  - **Memory (registers, stack) and possibly instructions**
    - **to save caller‘s environment,**
    - **to place parameters, return address, result values,**
    - **to store callee‘s local variables**
  - **Registers to help organize the subroutine execution**

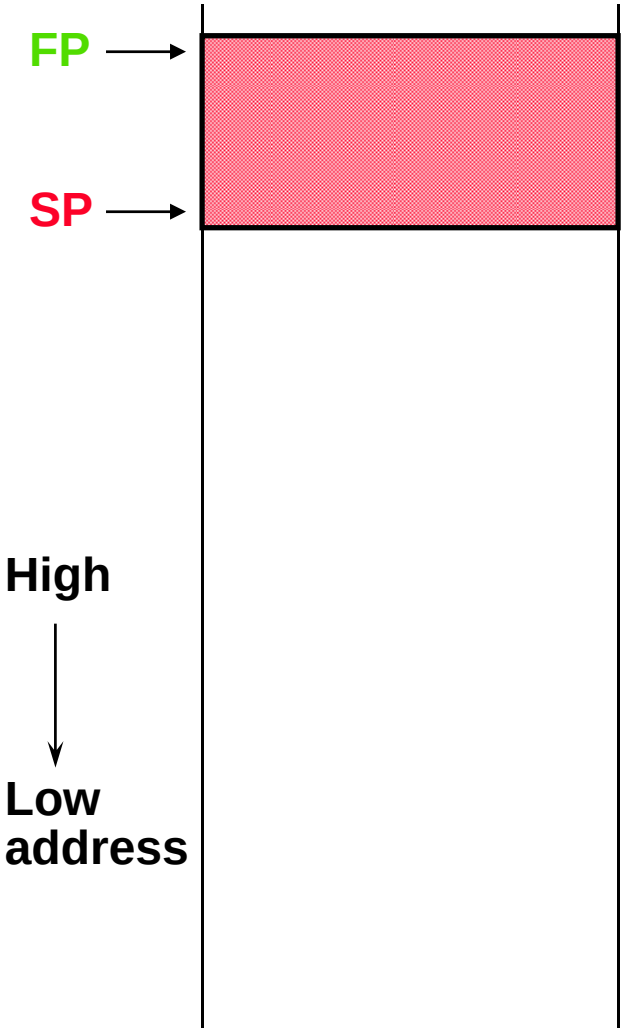
Example: MIPS Memory Allocation



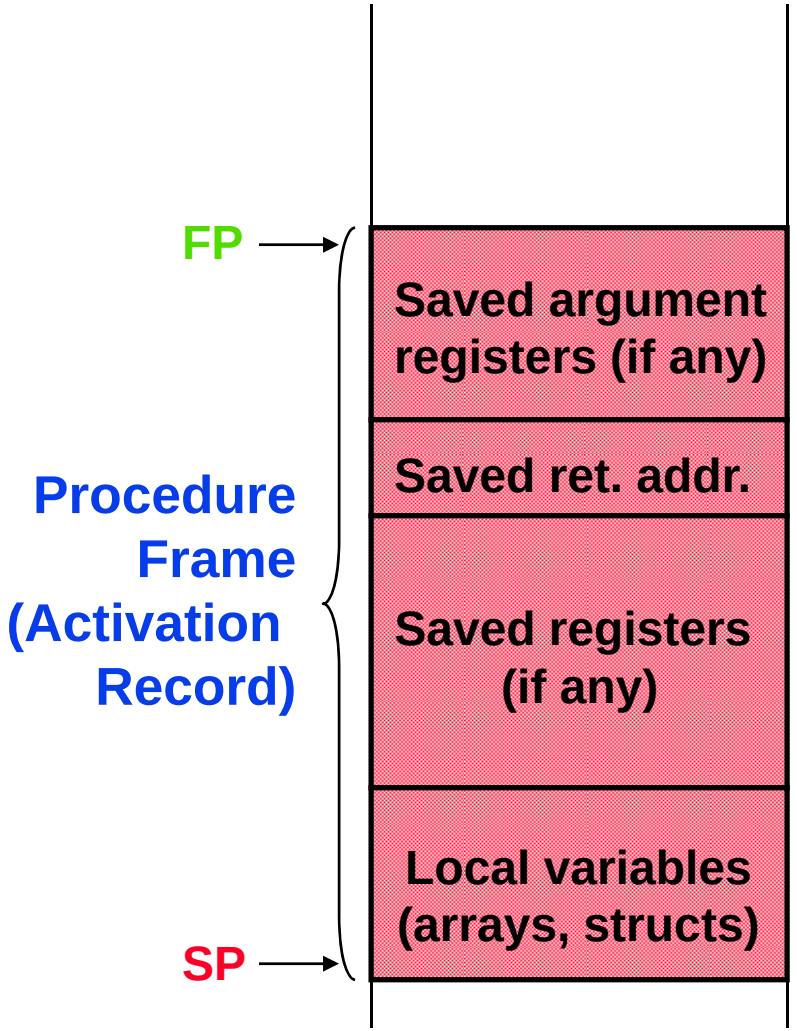
Like in MIPS,
the memory stack
in many machines
grows from high to
low addresses.

Example: MIPS Stack Allocation

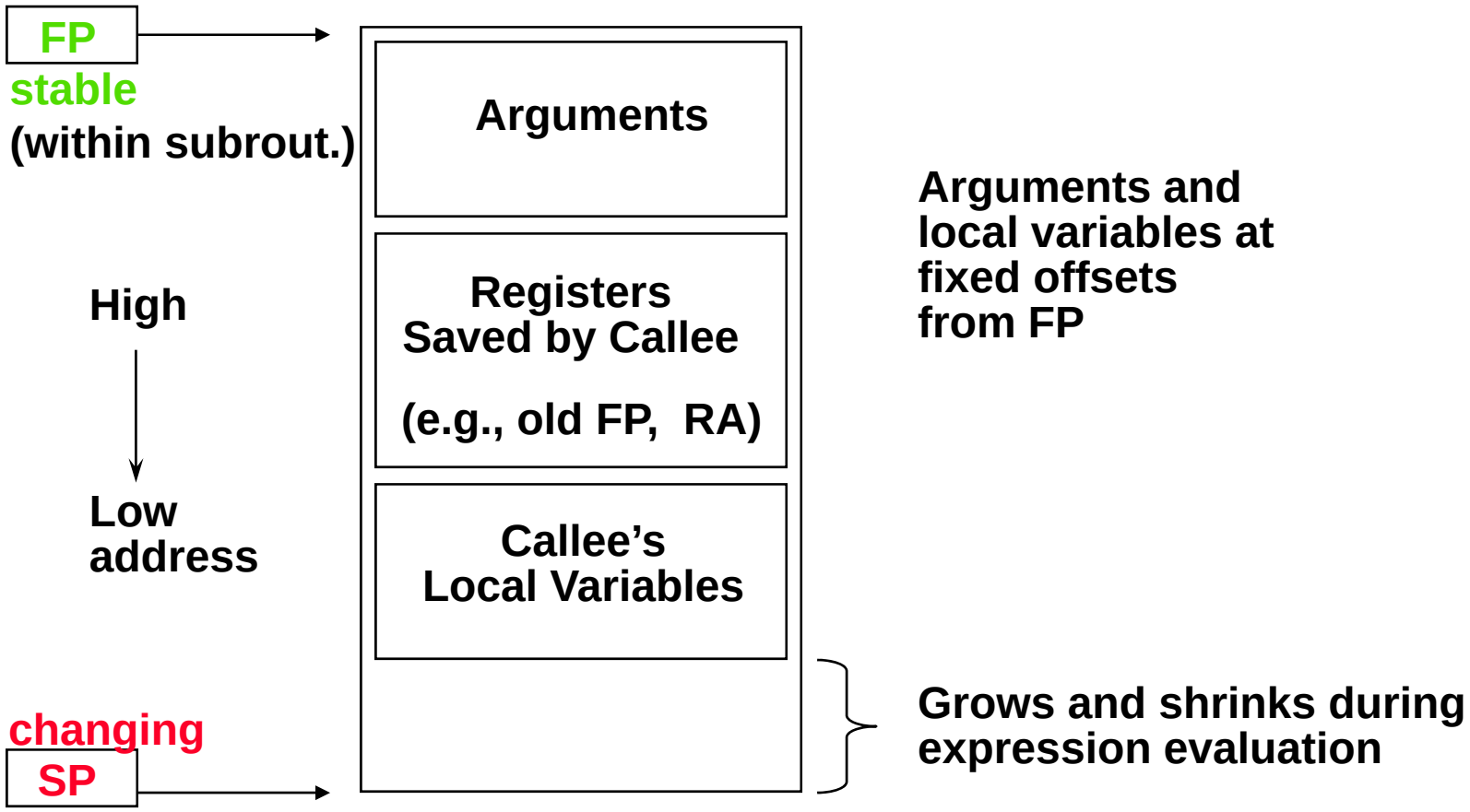
Before and after subroutine call



During subroutine execution



Example: MIPS Stack Frames – Call-Return Linkage



- Variations on stacks possible (e.g., up/down, elem. pointed to by SP)
- Compilers normally keep scalar variables in registers, not memory.

Example: MIPS Software Conventions for Registers

0	zero	Constant 0
1	at	Reserved for assembler
2	v0	Expression evaluation
3	v1	& subrout. return values
4	a0	Argument registers
5	a1	to pass 4 parameters
6	a2	to subroutine
7	a3	
8	t0	Temporary registers,
. . .		saved by caller
15	t7	
16	s0	Registers, must be
. . .		saved by callee
23	s7	
24	t8	Temp. regs. (cont'd)
25	t9	
26	k0	Reserved for OS kernel
27	k1	
28	gp	Pointer to global area
29	sp	Stack pointer
30	fp	Frame pointer
31	ra	Return address (HW)

Example: MIPS Calling Conventions

What is preserved across a subroutine call?

Preserved	Not preserved
Saved regs: $s0–$s7	Temporary regs: $t0–$t9
Stack pointer reg: $sp	Argument regs: $a0–$a3
Return address reg: $ra	Return value regs: $v0–$v1
Frame pointer reg: $fp	
Stack above the SP	Stack below the SP

Example: MIPS Compare, Branch, Jump Instructions

Instruction	Example	Meaning
branch on equal	beq $1,$2,100	if ($1 == $2) go to PC+4+100 *Equal test; PC relative branch*
..... (bne, bczt, bczf)		
branch on less than or equal zero	blez $1,100	if ($1<= 0) go to PC+4+100 *Compare to 0; PC relative branch*
..... (bltz, bgez, bgtz)		
set on less than	slt $1,$2,$3	if ($2 < $3) $1=1; else $1=0 *Compare less than; 2's complement*
set less than uns.	sltu $1,$2,$3	if ($2 < $3) $1=1; else $1=0 *Compare less than; natural numbers*
..... (slti, sltiu)		
jump	j 10000	go to 10000 *Jump to target address*
jump register	**jr $31**	**go to $31 (ra)** ***For switch/case and procedure return***
jump and link	**jal 10000**	**$31 (ra) = PC + 4; go to 10000** ***For procedure call***
..... (jalr, bltzal, bgezal)		

MIPS Assembly Example: Procedure Call (1)

C procedure example: string copy

```
int strcpy (char x[], char y[])      /* copy string y to x using */
{                                    /* null byte termination of C */
    int i = 0;
    while ((x[i] = y[i]) != 0)       /* copy and test byte */
        i = i + 1;
}
```

MIPS assembly code for ‘strcpy‘ subroutine (portion):

```
strcpy:                              # assume base addresses of x and y
                                     # are passed in $a0 and $a1, resp.
        addi  $sp, $sp, –4           # adjust stack for 1 more item
        sw    $s0, 0($sp)            # save $s0, used for i in this subroutine
        add   $s0, $zero, $zero      # i = 0 + 0
        …..
```

MIPS Assembly Example: Procedure Call (2)

MIPS assembly code for ‘strcpy‘ subroutine (cont‘d.):

```
        .....
Loop:   add     $t1, $a1, $s0     # put address of y[i] into $t1 (byte addr.!)
        lb      $t2, 0($t1)       # load byte: $t2 = y[i]
        add     $t3, $a0, $s0     # put address of x[i] into $t3
        sb      $t2, 0($t3)       # store byte: x[i] = y[i]
        addi    $s0, $s0, 1       # i = i + 1
        bne     $t2, $zero, Loop        # if y[i] ≠ 0, go to Loop
                                  # case y[i] == 0: end of string
        lw      $s0, 0($sp)       # restore $s0 from the stack
        addi    $sp, $sp, 4       # pop 1 item off the stack
        jr      $ra               # return to the caller
```

MIPS Assembly Example: Recursive Procedure Call (1)

C recursive procedure example: factorial

```
int fact (int n)     /* Compute factorial */
{
    if (n < 1)
        return (1);
        else return (n * fact(n–1));
}
```

MIPS assembly code for ‘fact‘ subroutine (portion):

```
fact:                              # assume argument n is passed in $a0
        addi  $sp, $sp, –8         # adjust stack for 2 items
        sw    $ra, 4($sp)          # save the return address
        sw    $a0, 0($sp)          # save the argument n
        slti  $t0, $a0, 1          # test for n < 1
        beq   $t0, $zero, L1       # if n ≥ 1, go to L1
        …..
```

MIPS Assembly Example: Recursive Procedure Call (2)

MIPS assembly code for ‘fact‘ subroutine (cont‘d.):

```
        .....
        addi    $v0, $zero, 1     # case n < 1: return 1
        addi    $sp, $sp, 8       # pop 2 items off the stack (in case n<1
                                  # only virtually, since regs. unchanged)
        jr      $ra               # return (to line after jal)
L1:     addi    $a0, $a0, –1      # case n ≥ 1: argument gets (n–1)
        jal     fact              # call ‘fact’ with (n–1)
        lw      $a0, 0($sp)       # return from subr.: restore argument n
        lw      $ra, 4($sp)       # restore the return address
        addi    $sp, $sp, 8       # adjust stack pointer to pop 2 items
        mul     $v0, $a0, $v0     # return n * fact (n–1)
        jr      $ra               # return to the caller
```

Larger MIPS Assembly Example: Array Sort (1)

C array sort example:

```
sort (int v[],  int n)     /* Sort array of integers */
{
    int i, j;
    for (i=0; i<n; i=i+1)
        for (j=i–1; j>=0 && v[j]>v[j+1]; j=j–1)
            swap(v,j);
}

swap (int v[], int k)     /* Swap two array elements */
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Larger MIPS Assembly Example: Array Sort (2)

MIPS assembly code for ‘swap’:

```
                                    # procedure body
swap:  add     $t1, $a1, $a1        # $a1 ... k; $t1 = k*2
       add     $t1, $t1, $t1        # $t1 = k*4
       add     $t1, $a0, $t1        # $a0 ... base address of v[]; $t1 = v + k*4;
                                    # $t1 ... address of v[k]
       lw      $t0, 0($t1)          # $t0 ... temp; $t0 = v[k]
       lw      $t2, 4($t1)          # $t2 = v[k+1]
       sw      $t2, 0($t1)          # v[k] = $t2
       sw      $t0, 4($t1)          # v[k+1] = $t0
                                    # procedure return
       jr      $ra                  # return to calling routine
```

No need to save registers and return address,
allocate local variables on stack, ...

Larger MIPS Assembly Example: Array Sort (3)

MIPS assembly code for 'sort':

```
                                    # saving registers
sort:   addi    $sp, $sp, –20       # make room on stack for 5 regs.
        sw      $ra, 16($sp)        # save $ra on stack
        sw      $s3, 12($sp)        # save $s3
        sw      $s2, 8($sp)         # save $s2
        sw      $s1, 4($sp)         # save $s1
        sw      $s0, 0($sp)         # save $s0

                                    # procedure body
        move    $s2, $a0            # save $a0 (... v)  (move is a
        move    $s3, $a1            # save $a1 (... n)   pseudo-instr.)
        move    $s0, $zero          # i = 0
for1:   slt     $t0, $s0, $s3       # $t0 = 0  if  $s0 ≥ $s3 (i ≥ n)
        beq     $t0, $zero, exit1   # exit outer loop if  i ≥ n
```

Move params

Outer loop

Larger MIPS Assembly Example: Array Sort (4)

```
        addi    $s1, $s0, –1      # j = i–1
for2:   slti    $t0, $s1, 0       # $t0 = 1  if  $s1 < 0 (j < 0)
        bne     $t0, $zero, exit2    # exit inner loop if j < 0
        add     $t1, $s1, $s1     # $t1 = j*2
        add     $t1, $t1, $t1     # $t1 = j*4
        add     $t2, $s2, $t1     # $t2 = v + j*4
        lw      $t3, 0($t2)       # $t3 = v[j]
        lw      $t4, 4($t2)       # $t4 = v[j+1]
        slt     $t0, $t4, $t3     # $t0 = 0  if v[j+1] ≥ v[j]
        beq     $t0, $zero, exit2    # exit inner loop if v[j+1] ≥ v[j]
        move    $a0, $s2          # 1st param of swap is v (old $a0)
        move    $a1, $s1          # 2nd param of swap is j
        jal     swap              # call swap
        addi    $s1, $s1, –1      # j = j–1
        j       for2              # jump to test of inner loop
exit2:  addi    $s0, $s0, 1       # i = i+1
        j       for1              # jump to test of outer loop
```

Inner loop

Pass params & call

Inner loop

Outer loop

Larger MIPS Assembly Example: Array Sort (5)

```
                                    # restoring registers
exit1:  lw      $s0, 0($sp)         # restore $s0 from stack
        lw      $s1, 4($sp)         # restore $s1
        lw      $s2, 8($sp)         # restore $s2
        lw      $s3, 12($sp)        # restore $s3
        lw      $ra, 16($sp)        # restore $ra
        addi    $sp, $sp, 20        # restore stack pointer

                                    # procedure return
        jr      $ra                 # return to calling routine
```

Example: MIPS Operation Overview (III): Miscellaneous

▪ **break**	**A breakpoint trap occurs, transfers control to exception handler**
▪ **syscall**	**A system trap occurs, transfers control to exception handler; e.g., to handle I/O**
▪ **coprocessor instrs.**	**Support for floating point**
▪ **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**

Summary: Salient Features of MIPS I

- **32-bit fixed format instructions** (3 formats)
- **31 32-bit GP regs.** (R0 contains zero) and **32 32-bit FP regs.**
  (and HI+LO); partitioned by software convention
- **3-address, register-register arithmetic and logical instructions**
- **Simple, few instructions**
- **Simple data types and sizes**
- **Single address mode for loads and stores:**
  base+displacement; no indirection, scaled, etc.
- **16-bit immediate data/address plus LUI instruction**
- **Simple branch conditions:**
  - Compare against zero or two registers for =,≠
  - No integer condition codes

Summary: Instruction Set Design (MIPS I)

- Use general purpose registers with a load-store architecture: YES
- Provide at least 16 general purpose registers plus separate floating-point registers: 31 GPRs & 32 FPRs
- Support basic addressing modes: displacement (with address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register deferred: YES: 16 bits for immediate, displacement (displacement=0 => register indirect)
- All addressing modes apply to all data transfer instructions: YES
- Use fixed instruction encoding if interested in performance and use variable instruction encoding if interested in code size: Fixed
- Support these data sizes and types: 8-bit, 16-bit, 32-bit integers and 32-bit and 64-bit IEEE 754 floating point numbers: YES
- Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch (with a PC-relative address at least 8-bits long), jump, call, and return: YES, 16 bits
- Aim for a minimalist instruction set: YES

Summary: Principles for Instruction Set & HW Design

1. Simplicity favors regularity

Regularity motivates many features of the MIPS instruction set, and enables efficient implementation.

2. Smaller is faster

The desire for speed is the reason that MIPS has 31 GPRs rather than more.

3. Good design demands good compromises

MIPS example: compromise between providing for larger addresses and constants in instructions and keeping all instructions the same length.

4. Make the common case fast

MIPS examples: fast PC-relative addressing for conditional branches and immediate addressing for constant operands.