

Chapter 2

Instructions: Language of the Computer

Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
 - But with many aspects in common
- Early computers had very simple instruction sets
 - Simplified implementation
- Many modern computers also have simple instruction sets

Typical Operations (little change since 1960)

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

Top 10 80x86 Instructions

Rank	instruction	Integer Average Percent total executed
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	<hr/> 96%

- Simple instructions dominate instruction frequency

Operation Summary

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

load

store

add

subtract

move register-register

and

shift

compare equal, compare not equal

branch

jump

call

return

Instructions

- Language of the Machine
- More primitive than higher level languages
e.g., no sophisticated control flow
- Very restrictive
e.g., MIPS Arithmetic Instructions
- 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: maximize performance and minimize cost, reduce design time

Basic ISA Classes

Accumulator (1 register): (Example: most old mainframes)

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: (Example: old HP postfix calculator)

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

General Purpose Register: (Example: some old mainframes)

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:

3 address	add Ra Rb Rc	$\text{Ra} \leftarrow \text{Rb} + \text{Rc}$
	load Ra Rb	$\text{Ra} \leftarrow \text{mem}[\text{Rb}]$
	store Ra Rb	$\text{mem}[\text{Rb}] \leftarrow \text{Ra}$

MIPS
& other
RISC
processors

General Purpose Registers Dominate

- 1975-1995 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 vs. stack
 - registers can hold variables
 - memory traffic is reduced, so program is sped up (since registers are faster than memory)
 - code density improves (since register named with fewer bits than memory location)

RISC - Reduced Instruction Set Computer

- RISC philosophy
 - fixed instruction lengths
 - load-store instruction sets
 - limited addressing modes
 - limited operations
- MIPS, Sun SPARC, HP PA-RISC, IBM PowerPC, Intel (Compaq) Alpha, ...
- Instruction sets are measured by how well compilers use them as opposed to how well assembly language programmers use them

Design goals: speed, cost (design, fabrication, test, packaging), size, power consumption, reliability, memory space (embedded systems)

The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies
- Large share of embedded core market
 - Applications in consumer electronics, network/storage equipment, cameras, printers, ...
- Typical of many modern ISAs
 - See MIPS Reference Data tear-out card (green card).

MIPS Instruction Set Architecture

- | ■ Six instruction categories | Examples |
|---------------------------------|----------------------|
| ■ arithmetic: | add \$17, \$18, \$19 |
| ■ logical: | or \$17, \$18, \$19 |
| ■ data transfer to/from memory: | lw \$17, 100(\$18) |
| ■ conditional branch: | beq \$17, \$18, 25 |
| ■ unconditional jump: | jr \$31 |
| ■ floating point: | mult \$18, \$19 |
- The examples all address **REGISTERS**! Notation: \$# refers to register # (i.e., \$17 is register 17).
 - Some of them also address “main memory” (e.g., lw)
 - We’ll talk more about this later.

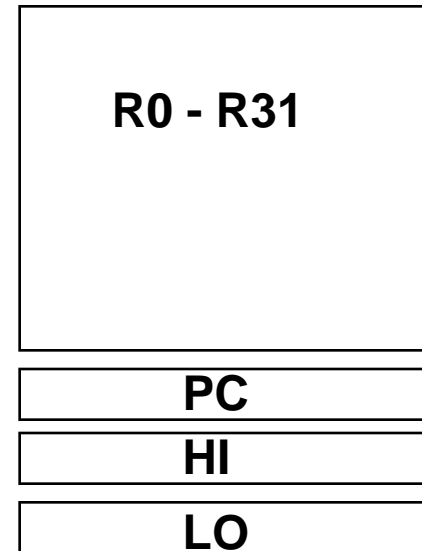
MIPS Instruction Set Architecture

■ Instruction Categories

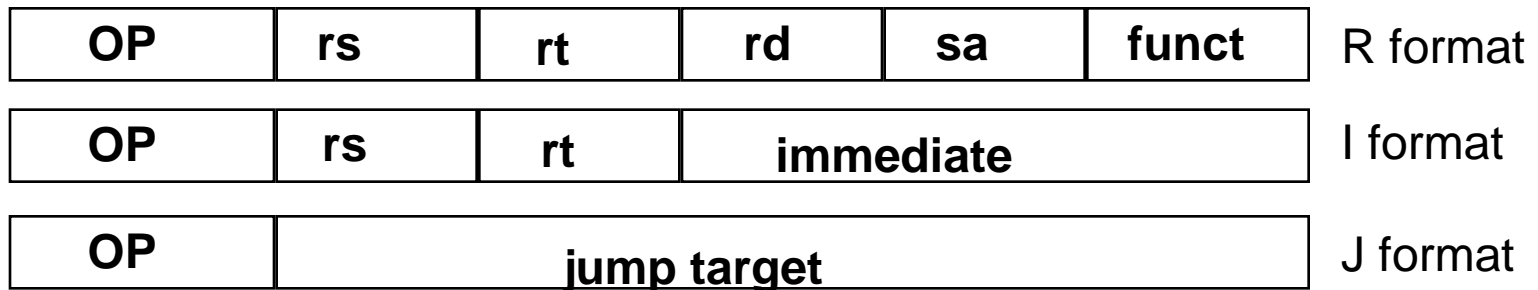
So
called
integer
instructions

- Computational
- Load/Store
- Jump and Branch
- Floating Point
 - Coprocessor (separate hw)
- Memory Management
- Special

Registers



3 Instruction Formats: all 32 bits wide



Arithmetic Operations

- Add and subtract, three operands
 - Two sources and one destination

add a, b, c # a gets b + c
- All arithmetic operations have this form
- *Design Principle 1*: Simplicity favors regularity
 - Regularity makes implementation simpler
 - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

$f = (g + h) - (i + j);$

- Compiled pseudo-MIPS code:

```
add t0, g, h    # temp t0 = g + h
add t1, i, j    # temp t1 = i + j
sub f, t0, t1   # f = t0 - t1
```

Register Operands

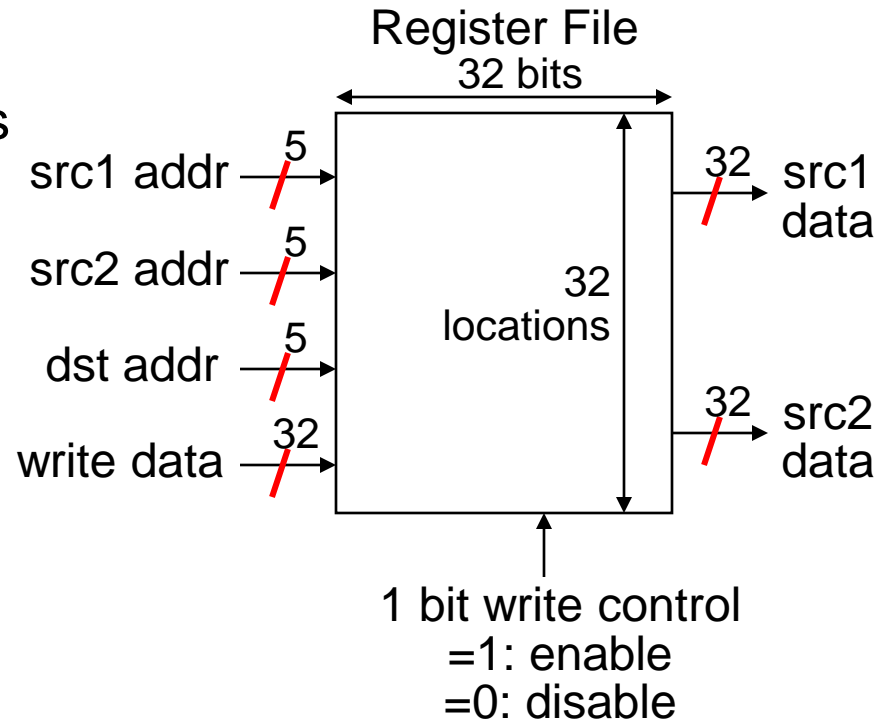
- Arithmetic instructions use register operands
- MIPS has a 32×32 -bit register file
 - Use for frequently accessed data
 - Numbered 0 to 31
 - 32-bit data called a “word” (equal to 4 bytes)
- Assembler names
 - \$t0, \$t1, ..., \$t9 for temporary values
 - \$s0, \$s1, ..., \$s7 for saved variables
 - Can also address registers by just their numbers: \$0, \$1, \$2, etc.
- *Design Principle 2*: Smaller is faster: registers
 - vs. main memory: millions of locations

Register naming convention

Symbolic Name	Register Number
\$zero	0
\$at	1
\$v0 - \$v1	2-3
\$a0 - \$a3	4-7
\$t0 - \$t7	8-15
\$s0 - \$s7	16-23
\$t8 - \$t9	24-25
\$gp	28
\$sp	29
\$fp	30
\$ra	31

MIPS Register File

- Holds thirty-two 32-bit registers
 - Two read ports and
 - One write port



Register Operand Example

- C code:

$f = (g + h) - (i + j);$

- f, \dots, j in $\$s0, \dots, \$s4$

- Compiled MIPS code:

```
add $t0, $s1, $s2    # t0 ← g+h
add $t1, $s3, $s4    # t1 ← i+j
sub $s0, $t0, $t1    # f ← t0-t1
```

Memory Operands

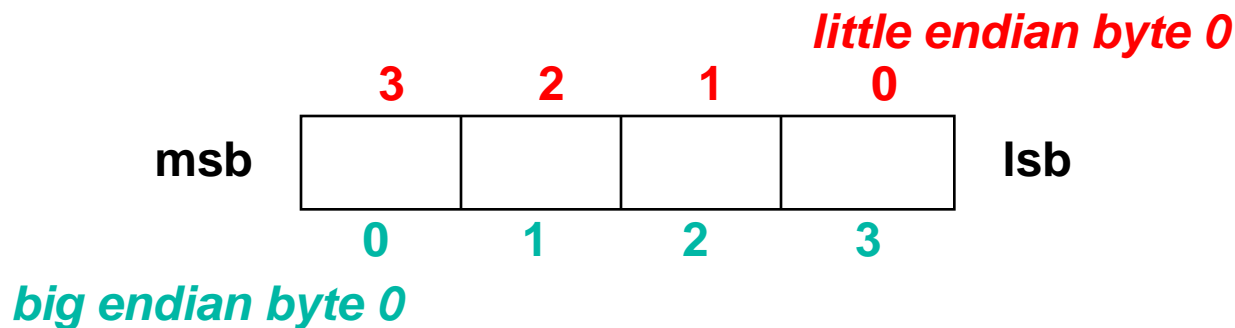
- Main memory used for composite data
 - Arrays, structures, dynamic data
- To apply arithmetic operations
 - Load values from memory into registers
 - Store result from register to memory
- Memory is byte addressed
 - Each address identifies an 8-bit byte
- Words are aligned in memory
 - Address must be a multiple of 4
- MIPS is Big Endian
 - Most-significant byte at least address of a word
 - *c.f.* Little Endian: least-significant byte at least address

Memory Addressing

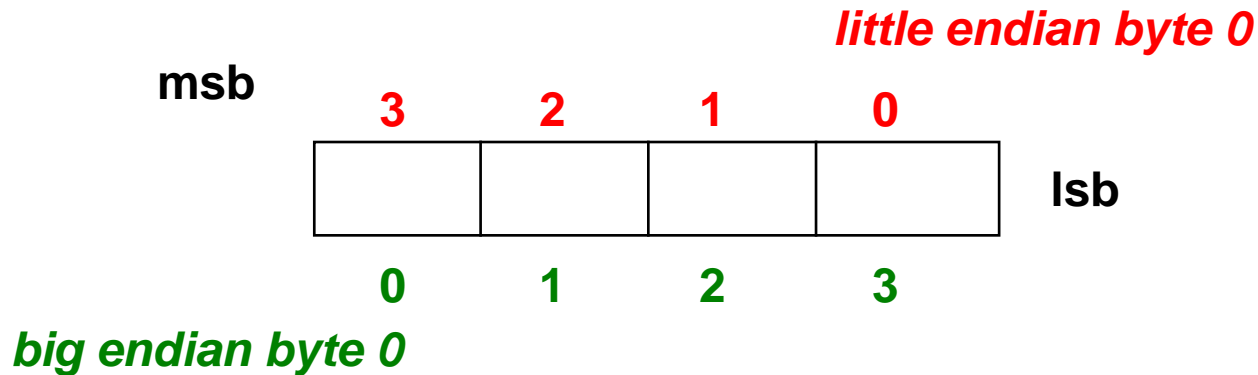
- Since 1980 almost every machine uses addresses to level of 8-bits (byte)
- 2 questions for design of ISA:
 - Q1: Since one could read a 32-bit word as four loads of bytes from sequential byte addresses or as one load word from a single byte address, how do byte addresses map onto word addresses?
 - Q2: Can a word be placed on any byte boundary?

Q1: Byte Addresses & Endianness

- Since 8-bit bytes are so useful, most architectures address individual **bytes** in memory
 - The memory address of a **word** must be a multiple of 4 (**alignment restriction**)
- **Big Endian**: leftmost (most significant) byte is word address
IBM 360/370, Motorola 68k, **MIPS**, Sparc, HP PA
- **Little Endian**: rightmost (least significant) byte is word address
Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

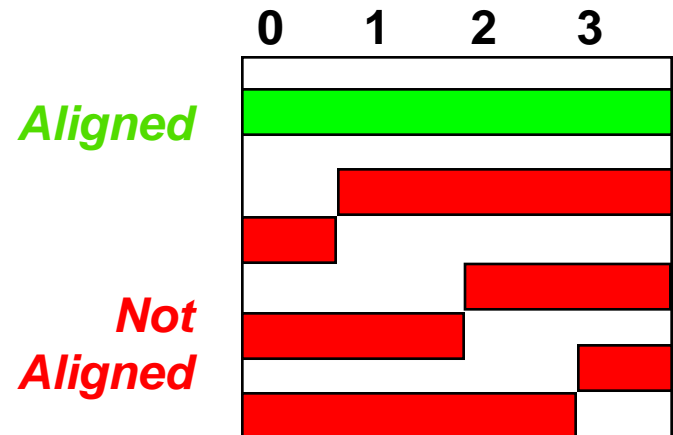


Q2: Addressing Objects: Endianness and Alignment



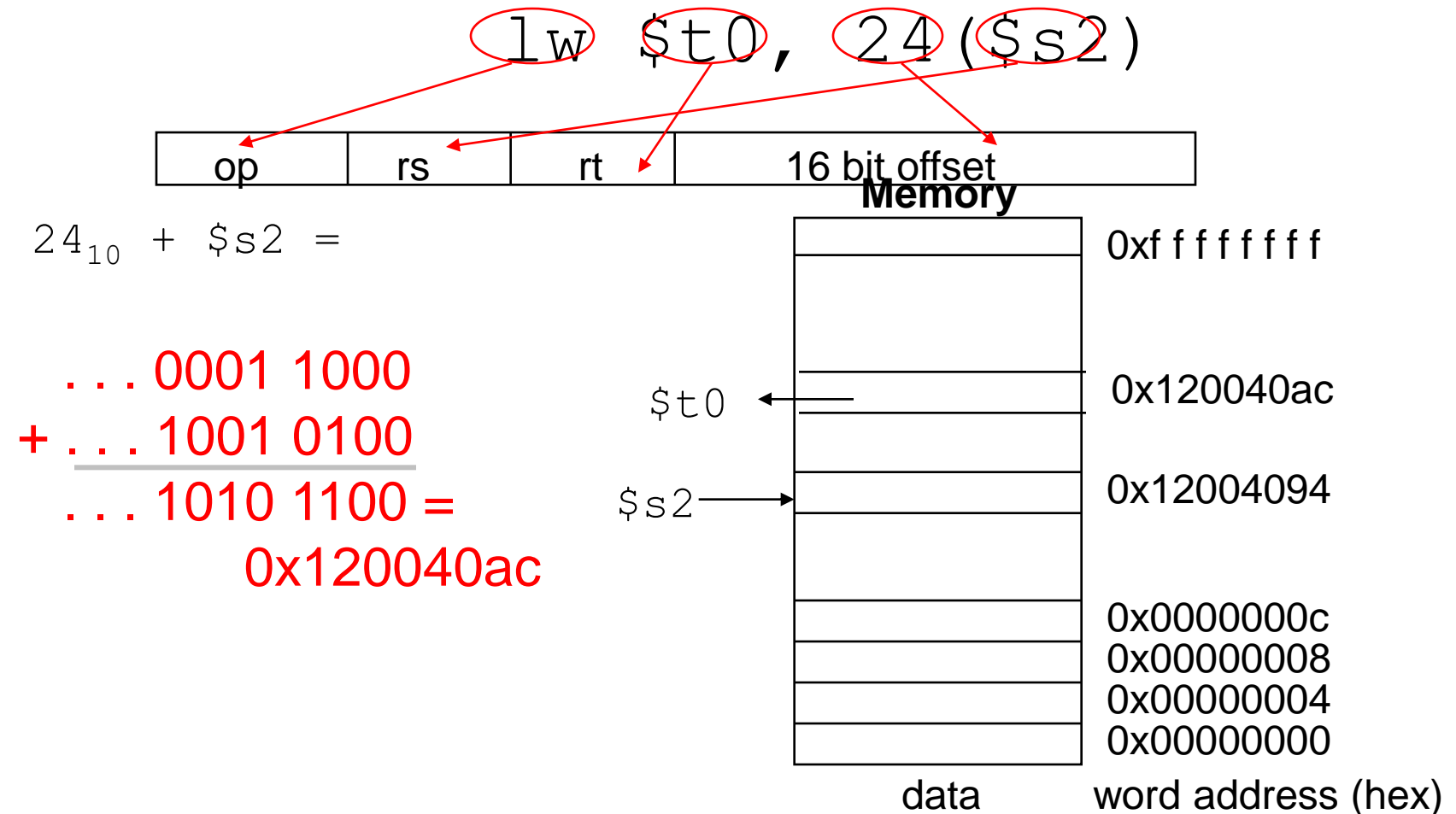
Alignment: require that objects fall on address that is multiple of their size.

Words: must be aligned with 4 byte addresses: 0, 4, 8, 12, etc.



Machine Language - Load Instruction

■ Load/Store Instruction Format (I format):



Memory Operand Example 1


- C code (assume int variables: 4 bytes):
g = h + A[8];
 - g in \$s1, h in \$s2, base address of A in \$s3
- Compiled MIPS code:
 - Index 8 requires offset of 32 (=8 x 4 bytes)
 - 4 bytes per word

```
lw    $t0, 32($s3)    # load word
add   $s1, $s2, $t0
```

offset



base register



Memory Operand Example 2

- C code:

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

- `h` in `$s2`, base address of `A` in `$s3`

- Compiled MIPS code:

- Index 8 requires offset of $32 = 8 \times 4$
- Index 12 requires offset of $48 = 12 \times 4$

```
lw    $t0, 32($s3)    # load word
add   $t0, $s2, $t0
sw    $t0, 48($s3)    # store word
```

Immediate Operands

- Constant data specified in an instruction
`addi $s3, $s3, 4`
- No subtract immediate instruction
 - Just use a negative constant
`addi $s2, $s1, -1`
- *Design Principle 3*: Make the common case fast
 - Small constants are common
 - Immediate operand avoids a load (memory) instruction

The Constant Zero

- MIPS register 0 (\$zero) is the constant 0
 - Cannot be overwritten
- Useful for common operations
 - E.g., copy between registers
`add $t2, $s1, $zero`



$t2 \leftarrow s1$

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: 0 to $+2^n - 1$

- Example

- $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
 $= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
 $= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$

- Using 32 bits

- 0 to +4,294,967,295

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$

- Example

- $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
 $= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
 $= -2,147,483,648 + 2,147,483,644 = -4_{10}$

- Using 32 bits

- $-2,147,483,648$ to $+2,147,483,647$

2s-Complement Signed Integers

- Bit 31 is sign bit
 - 1 for negative numbers
 - 0 for non-negative numbers
- $-(-2^n - 1)$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
 - 0: 0000 0000 ... 0000
 - -1: 1111 1111 ... 1111
 - Most-negative: 1000 0000 ... 0000
 - Most-positive: 0111 1111 ... 1111

Sign Extension

- Representing a number using more bits
 - Preserve the numeric value
 - Example: 16 bit immediate field to 32 bit register.
- In MIPS instruction set
 - addi: extend immediate value
 - lb, lh: extend loaded byte/halfword
 - beq, bne: extend the displacement
- Replicate the sign bit to the left
 - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
 - +2: 0000 0010 => 0000 0000 0000 0010
 - -2: 1111 1110 => 1111 1111 1111 1110

Representing Instructions

- Instructions are encoded in binary
 - Called machine code
- MIPS instructions
 - Encoded as 32-bit instruction words
 - Small number of formats encoding operation code (opcode), register numbers, ...
 - Regularity!
- Register numbers
 - \$t0 – \$t7 are reg's 8 – 15
 - \$t8 – \$t9 are reg's 24 – 25
 - \$s0 – \$s7 are reg's 16 – 23

MIPS R-format Instructions



■ Instruction fields

- op: operation code (opcode)
- rs: first source register number
- rt: second source register number
- rd: destination register number
- shamt: shift amount (00000 for now)
- funct: function code (extends opcode)

R-format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

add \$t0, \$s1, \$s2

special	\$s1	\$s2	\$t0	0	add
---------	------	------	------	---	-----

Opcode = 0
R-type
instruction

0	17	18	8	0	32
---	----	----	---	---	----

000000	10001	10010	01000	00000	100000
--------	-------	-------	-------	-------	--------

$00000010001100100100000000100000_2 = 02324020_{16}$

MIPS I-format Instructions



- Immediate arithmetic and load/store instructions
 - rt: destination or source register number
 - Constant: -2^{15} to $+2^{15} - 1$
 - Address: offset added to base address in rs
- *Design Principle 4*: Good design demands good compromises
 - Different formats complicate decoding, but allow 32-bit instructions uniformly
 - Keep formats as similar as possible

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR			or, ori
Bitwise NOT	~	~	nor

- Useful for extracting and inserting groups of bits in a word

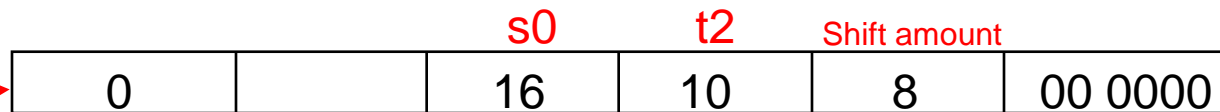
MIPS Shift Operations

- Need operations to **pack** and **unpack** 8-bit characters into 32-bit words
- Shifts move all the bits in a word left or right

```
sll $t2, $s0, 8 # $t2 = $s0 << 8 bits
```

```
srl $t2, $s0, 8 # $t2 = $s0 >> 8 bits
```

- Instruction Format (**R** format)



Function code for sll

- Such shifts are called logical because they fill with zeros
 - Notice that a 5-bit shamt field is enough to shift a 32-bit value $2^5 - 1$ or 31 bit positions

MIPS Logical Operations

- There are a number of **bit-wise** logical operations in the MIPS ISA

`and $t0, $t1, $t2` `#$t0 = $t1 & $t2`

`or $t0, $t1, $t2` `#$t0 = $t1 | $t2`

`nor $t0, $t1, $t2` `#$t0 = not($t1 | $t2)`

- Instruction Format (**R** format)

0	9	10	8	0	0x24
---	---	----	---	---	------

`andi $t0, $t1, 0xFF00` `#$t0 = $t1 & ff00`

`ori $t0, $t1, 0xFF00` `#$t0 = $t1 | ff00`

- Instruction Format (**I** format)

0x0D	9	8	0xFF00
------	---	---	--------

MIPS Control Flow Instructions

- MIPS **conditional branch** instructions:

bne \$s0, \$s1, Lbl1 #go to Lbl1 if \$s0≠\$s1

beq \$s0, \$s1, Lbl1 #go to Lbl1 if \$s0=\$s1

- Ex: if (i==j) **h = i + j;** C statement

 bne \$s0, \$s1, Lbl1

 add \$s3, \$s0, \$s1

Lbl1:

...

} MIPS code

- Instruction Format (I format):

0x05	16	17	16 bit offset
------	----	----	---------------

- How is the branch destination address specified?

Compiling If Statements

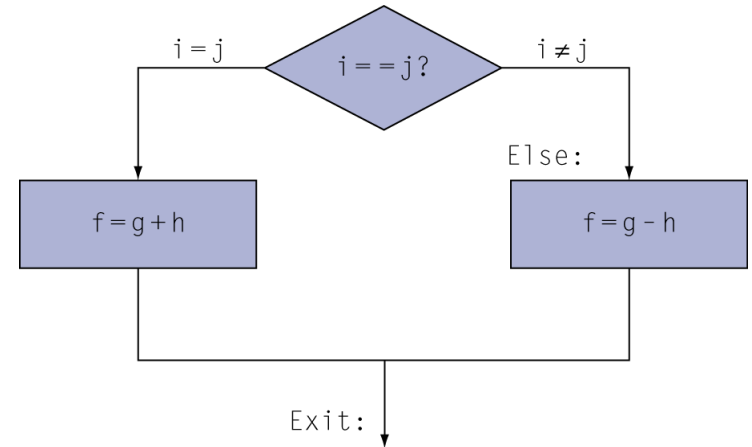
- C code:

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

- f, g, ... in \$s0, \$s1, ...

- Compiled MIPS code:

```
        bne $s3, $s4, Else  
        add $s0, $s1, $s2  
        j   Exit  
Else:   sub $s0, $s1, $s2  
Exit:   ...
```

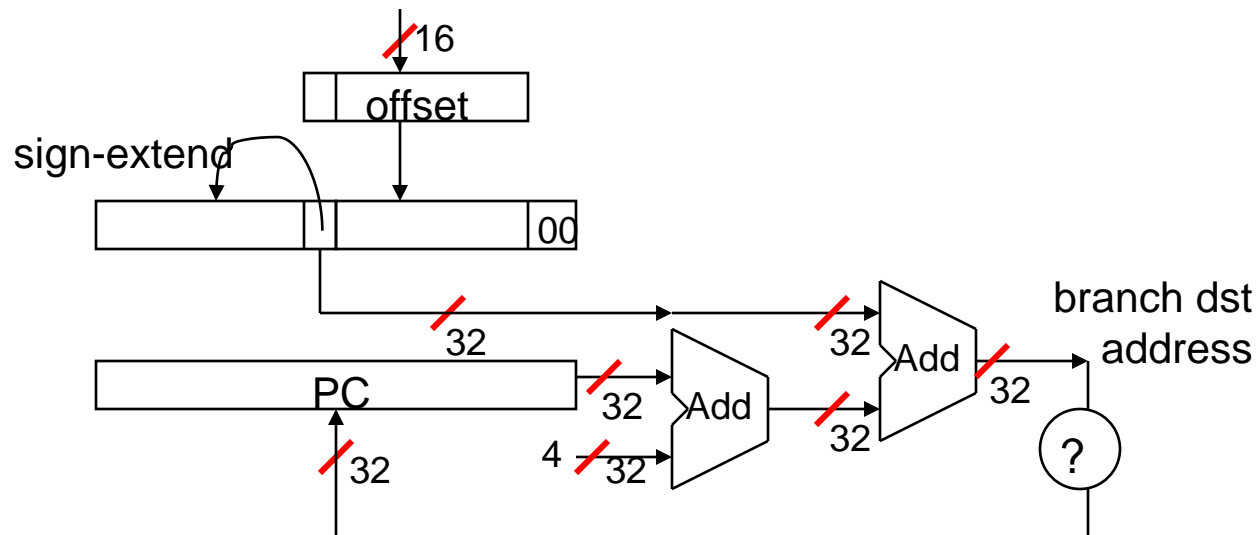


Assembler calculates addresses

Specifying Branch Destinations

- Use a register (like in lw and sw) added to the 16-bit offset
 - which register? Instruction Address Register (the **PC**)
 - its use is automatically **implied** by instruction
 - PC gets updated (PC+4) during the **fetch** cycle so that it holds the address of the next instruction
 - limits the branch distance to **-2^{15} to $+2^{15}-1$** (word) instructions from the (instruction after the) branch instruction, but most branches are local anyway

from the low order 16 bits of the branch instruction



Aside: Branching Far Away

- What if the branch destination is further away than can be captured in 16 bits?
- The assembler comes to the rescue – it inserts an unconditional jump to the branch target and inverts the condition

```
beq    $s0, $s1, L1
```

becomes

```
bne    $s0, $s1, L2
```

```
j      L1
```

```
L2:
```

In Support of Branch Instructions

- We have `beq`, `bne`, but what about other kinds of branches (e.g., branch-if-less-than)? For this, we need yet another instruction, `slt`

- Set on less than instruction:

```
slt $t0, $s0, $s1    # if $s0 < $s1    then
                      # $t0 = 1         else
                      # $t0 = 0
```

- Instruction format (**R** format):

0	16	17	8		0x24
---	----	----	---	--	------

- Alternate versions of `slt`


```
slti $t0, $s0, 25    # if $s0 < 25 then $t0=1 ...
```

```
sltu $t0, $s0, $s1    # if $s0 < $s1 then $t0=1 ...
```

```
sltiu $t0, $s0, 25    # if $s0 < 25 then $t0=1 ...
```

Aside: More Branch Instructions

- Can use `slt`, `beq`, `bne`, and the fixed value of 0 in register `$zero` to **create** other conditions Pseudo instruction

- less than `blt $s1, $s2, Label` 

Equivalent MIPS sequence { `slt $at, $s1, $s2` `#$at set to 1 if`
`bne $at, $zero, Label` `#$s1 < $s2`

- less than or equal to `ble $s1, $s2, Label`
- greater than `bgt $s1, $s2, Label`
- great than or equal to `bge $s1, $s2, Label`

- Such branches are included in the instruction set as pseudo instructions - recognized (and expanded) by the assembler

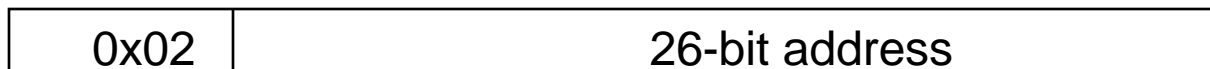
- Its why the assembler needs a reserved register (`$at`)

Other Control Flow Instructions

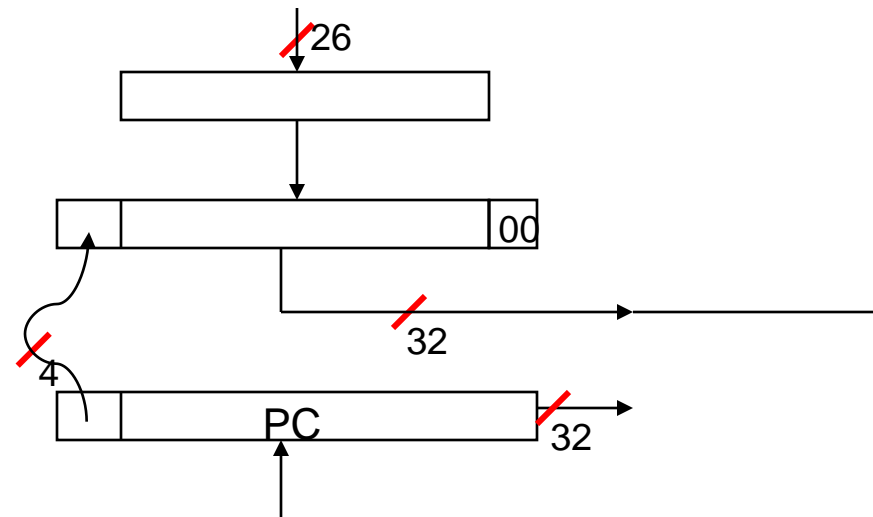
- MIPS also has an unconditional branch instruction or **jump** instruction:

```
j    label           #go to label
```

- Instruction Format (J Format):



from the low order 26 bits of the jump instruction



Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

- i in \$s3, k in \$s5, address of save in \$s6

- Compiled MIPS code:

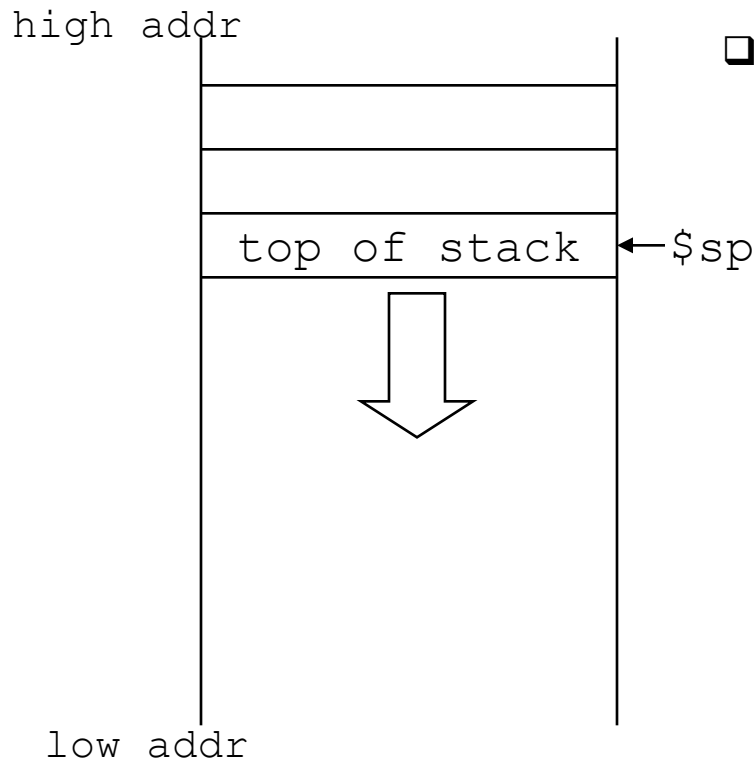
```
Loop:  sll    $t1, $s3, 2
        add   $t1, $t1, $s6
        lw    $t0, 0($t1)
        bne   $t0, $s5, Exit
        addi  $s3, $s3, 1
        j     Loop
Exit:  ...
```

Six Steps in Execution of a Procedure

1. Main routine (**caller**) places parameters in a place where the procedure (**callee**) can access them
 - `$a0 - $a3`: four **argument** registers
2. **Caller** transfers control to the **callee**
3. **Callee** acquires the storage resources needed
4. **Callee** performs the desired task
5. **Callee** places the result value in a place where the **caller** can access it
 - `$v0 - $v1`: two **value** registers for result values
6. **Callee** returns control to the **caller**
 - `$ra`: one **return address** register to return to the point of origin

Aside: Spilling Registers

- What if the **callee** needs to use more registers than allocated to argument and return values?
 - **callee** uses a **stack** – a last-in-first-out queue



- One of the general registers, $\$sp$ ($\$29$), is used to address the stack (which “grows” downward from high address to low address)

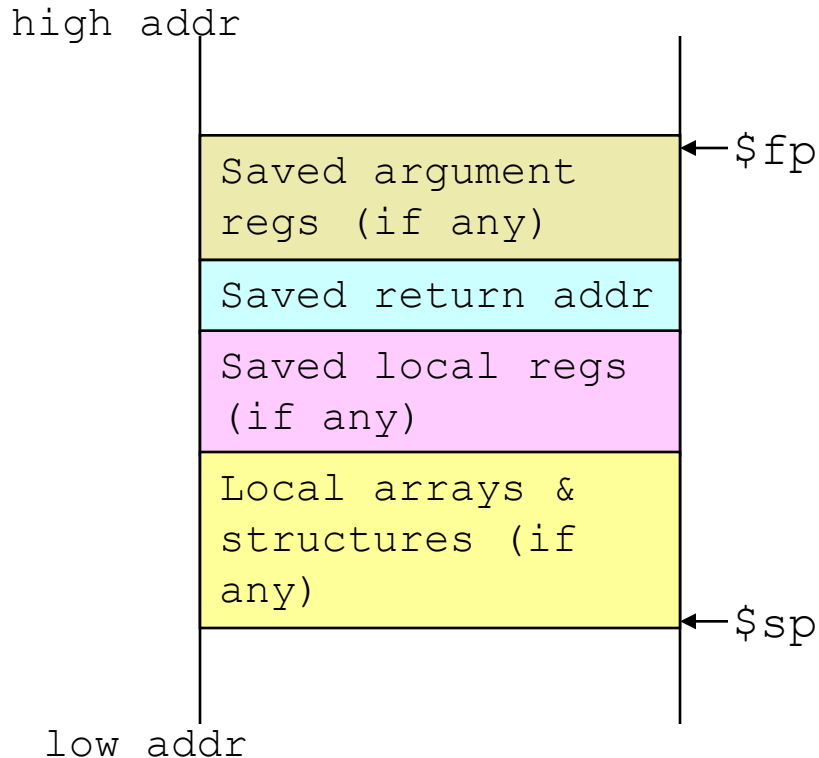
- add data onto the stack – **push**

$\$sp = \$sp - 4$
data on stack at new $\$sp$

- remove data from the stack – **pop**

data from stack at $\$sp$
 $\$sp = \$sp + 4$

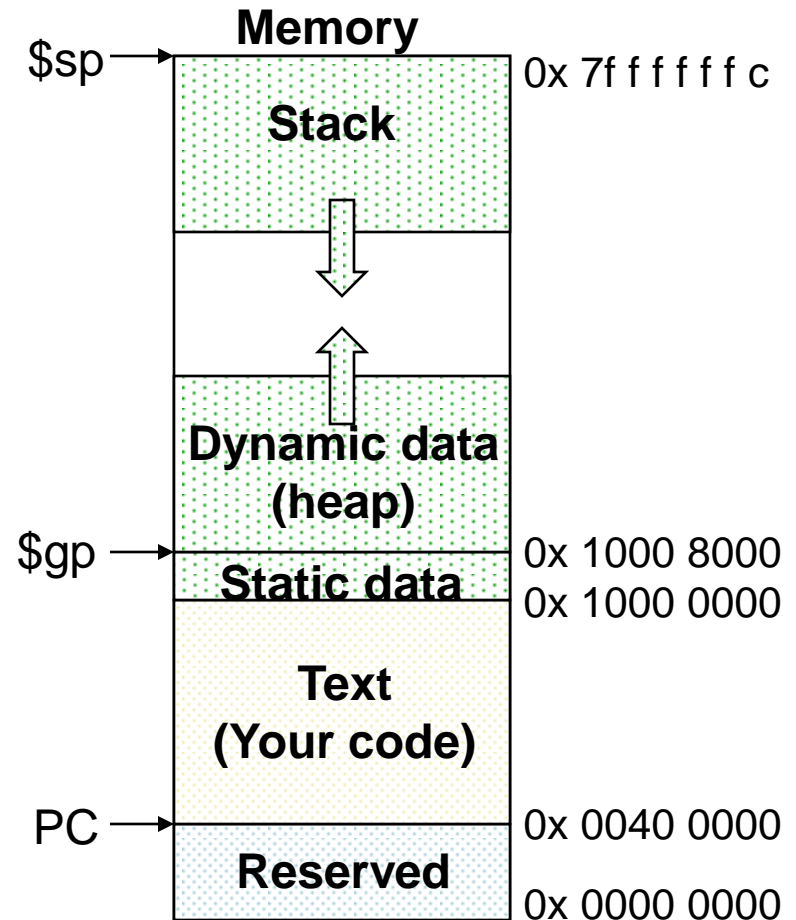
Aside: Allocating Space on the Stack



- The segment of the stack containing a procedure's saved registers and local variables is its **procedure frame** (aka **activation record**)
 - The frame pointer ($\$fp$) points to the first word of the frame of a procedure – providing a stable “base” register for the procedure
 - $\$fp$ is initialized using $\$sp$ on a call and $\$sp$ is restored using $\$fp$ on a return

Aside: Allocating Space on the Heap

- Static data segment for constants and other static variables (e.g., arrays)
- Dynamic data segment (aka **heap**) for structures that grow and shrink (e.g., linked lists)
 - Allocate space on the heap with `malloc()` and free it with `free()` in C



MIPS: Software conventions for Registers

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
...		(callee can clobber)
15	t7	
16	s0	callee saves
...		(caller can clobber)
23	s7	
24	t8	temporary (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)

Procedure Call Instructions

- Procedure call: jump and link
`jal ProcedureLabel`
 - Address of following instruction put in `$ra`
 - Jumps to target address
- Procedure return: jump register
`jr $ra`
 - Copies `$ra` to program counter
 - Can also be used for computed jumps
 - e.g., for case/switch statements

Leaf Procedure Example

- C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

- Arguments g, ..., j in \$a0, ..., \$a3
- f (local variable) in \$s0 (hence, need to save \$s0 on stack)
- Result in \$v0

Leaf Procedure Example

- MIPS code:

leaf_example:			
addi	\$sp,	\$sp, -4	Save \$s0 on stack
sw	\$s0,	0(\$sp)	
add	\$t0,	\$a0, \$a1	Procedure body
add	\$t1,	\$a2, \$a3	
sub	\$s0,	\$t0, \$t1	
add	\$v0,	\$s0, \$zero	Result
lw	\$s0,	0(\$sp)	Restore \$s0
addi	\$sp,	\$sp, 4	
jr	\$ra		Return

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
 - Its return address
 - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

```
int fact (int n)
{
    if (n < 1) return 1;
    else return n * fact(n - 1);
}
```

- Argument n in \$a0
- Result in \$v0

Non-Leaf Procedure Example

- MIPS code:

fact:		
addi	\$sp, \$sp, -8	# adjust stack for 2 items
sw	\$ra, 4(\$sp)	# save return address
sw	\$a0, 0(\$sp)	# save argument
slti	\$t0, \$a0, 1	# test for n < 1
beq	\$t0, \$zero, L1	
addi	\$v0, \$zero, 1	# if so, result is 1
addi	\$sp, \$sp, 8	# pop 2 items from stack
jr	\$ra	# and return
L1:	addi \$a0, \$a0, -1	# else decrement n
	jal fact	# recursive call
lw	\$a0, 0(\$sp)	# restore original n
lw	\$ra, 4(\$sp)	# and return address
addi	\$sp, \$sp, 8	# pop 2 items from stack
mul	\$v0, \$a0, \$v0	# multiply to get result
jr	\$ra	# and return

Character Data

- Byte-encoded character sets
 - ASCII: 128 characters
 - 95 graphic, 33 control
 - Latin-1: 256 characters
 - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
 - Used in Java, C++ wide characters, ...
 - Most of the world's alphabets, plus symbols
 - UTF-8, UTF-16: variable-length encodings

Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
 - String processing is a common case

`lb rt, offset(rs)` `lh rt, offset(rs)`

- Sign extend to 32 bits in `rt`

`lbu rt, offset(rs)` `lhu rt, offset(rs)`

- Zero extend to 32 bits in `rt`

`sb rt, offset(rs)` `sh rt, offset(rs)`

- Store just rightmost byte/halfword

String Copy Example

- C code (naïve):

- Null-terminated string

```
void strcpy (char x[], char y[])  
{ int i;  
  i = 0;  
  while ((x[i]=y[i])!='\0')  
    i += 1;  
}
```

- Addresses of x, y in \$a0, \$a1
- i in \$s0

String Copy Example

- MIPS code:

strcpy:		
	addi \$sp, \$sp, -4	# adjust stack for 1 item
	sw \$s0, 0(\$sp)	# save \$s0
	add \$s0, \$zero, \$zero	# i = 0
L1:	add \$t1, \$s0, \$a1	# addr of y[i] in \$t1
	lbu \$t2, 0(\$t1)	# \$t2 = y[i]
	add \$t3, \$s0, \$a0	# addr of x[i] in \$t3
	sb \$t2, 0(\$t3)	# x[i] = y[i]
	beq \$t2, \$zero, L2	# exit loop if y[i] == 0
	addi \$s0, \$s0, 1	# i = i + 1
	j L1	# next iteration of loop
L2:	lw \$s0, 0(\$sp)	# restore saved \$s0
	addi \$sp, \$sp, 4	# pop 1 item from stack
	jr \$ra	# and return

Example MIPS program (strlen)

```
# $s0 contains address of string whose length we compute
# use register $s1 for count variable
```

```
add    $s1,$0,$0      # count <- 0
# top of while loop
```

loop:

```
add    $t0,$s0,$s1    # $t0 <- address of teststr[count]
lb     $t1,0($t0)      # $t1 <- character stored
                        # at teststr[count] (load byte)
beq    $t1,$0,done     # done with loop if character is
                        # a null (0)
addi   $s1,$s1,1       # count++
beq    $0,$0,loop      # always branch back to loop label
```

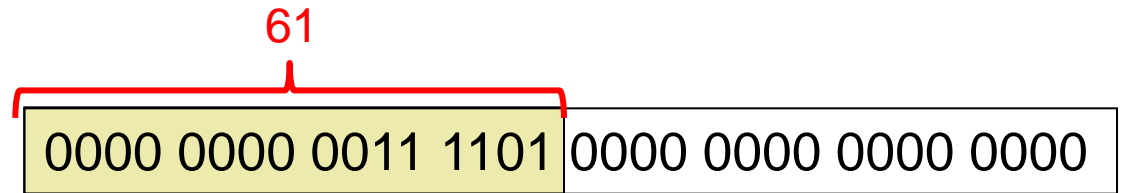
done:

```
jr     $ra             # exit main
```

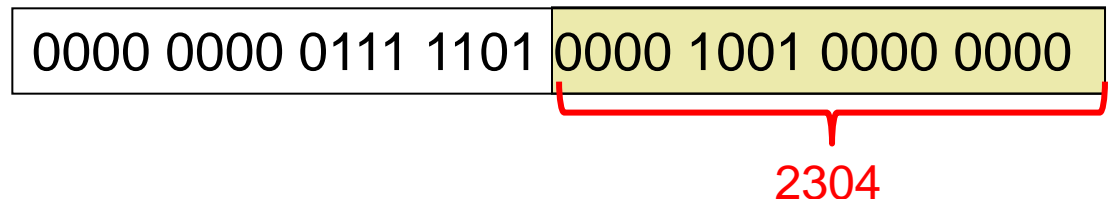
32-bit Constants

- Most constants are small
 - 16-bit immediate is sufficient
- For the occasional 32-bit constant
 - Copies 16-bit constant to left 16 bits of rt
 - Clears right 16 bits of rt to 0

`lui $s0, 61`



`ori $s0, $s0, 2304`



Addressing Mode Summary

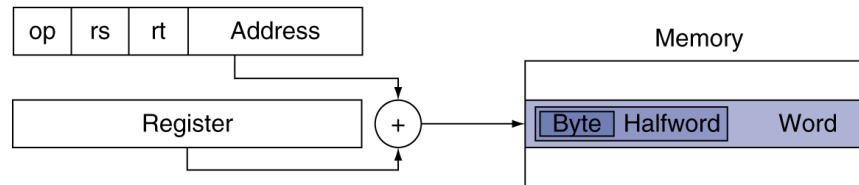
1. Immediate addressing



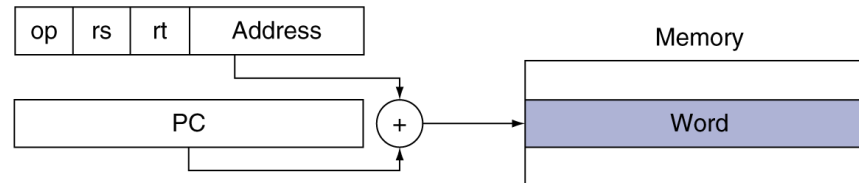
2. Register addressing



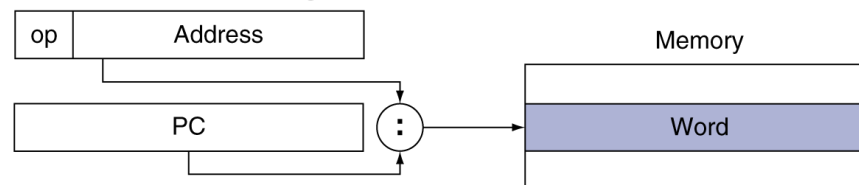
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing



FP Instructions in MIPS

- FP hardware is coprocessor 1
 - Adjunct processor that extends the ISA
- Separate FP registers
 - 32 single-precision: \$f0, \$f1, ... \$f31
 - Paired for double-precision: \$f0/\$f1, \$f2/\$f3, ...
 - Release 2 of MIPS ISA supports 32 × 64-bit FP reg's
- FP instructions operate only on FP registers
 - Programs generally don't do integer ops on FP data, or vice versa
 - More registers with minimal code-size impact
- FP load and store instructions
 - lwc1, ldc1, swc1, sdc1
 - e.g., ldc1 \$f8, 32(\$sp)

FP Instructions in MIPS

- Single-precision arithmetic
 - `add.s`, `sub.s`, `mul.s`, `div.s`
 - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
 - `add.d`, `sub.d`, `mul.d`, `div.d`
 - e.g., `mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
 - `c.xx.s`, `c.xx.d` (`xx` is `eq`, `lt`, `le`, ...)
 - Sets or clears FP condition-code bit
 - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false
 - `bc1t`, `bc1f`
 - e.g., `bc1t TargetLabel`

FP Example: °F to °C

- C code:

```
float f2c (float fahr) {  
    return ((5.0/9.0)*(fahr - 32.0));  
}
```

- fahr in \$f12, result in \$f0, literals in global memory space

- Compiled MIPS code:

```
f2c: lwc1    $f16, const5($gp)  
     lwc2    $f18, const9($gp)  
     div.s   $f16, $f16, $f18  
     lwc1    $f18, const32($gp)  
     sub.s   $f18, $f12, $f18  
     mul.s   $f0, $f16, $f18  
     jr      $ra
```

FP Example: Array Multiplication

- $X = X + Y \times Z$
 - All 32×32 matrices, 64-bit double-precision elements

- C code:

```
void mm (double x[][],  
         double y[][], double z[][]) {  
    int i, j, k;  
    for (i = 0; i != 32; i = i + 1)  
        for (j = 0; j != 32; j = j + 1)  
            for (k = 0; k != 32; k = k + 1)  
                x[i][j] = x[i][j]  
                    + y[i][k] * z[k][j];  
}
```

- Addresses of x, y, z in \$a0, \$a1, \$a2, and
i, j, k in \$s0, \$s1, \$s2

FP Example: Array Multiplication

■ MIPS code:

	li	\$t1, 32	# \$t1 = 32 (row size/loop end)
	li	\$s0, 0	# i = 0; initialize 1st for loop
L1:	li	\$s1, 0	# j = 0; restart 2nd for loop
L2:	li	\$s2, 0	# k = 0; restart 3rd for loop
	sll	\$t2, \$s0, 5	# \$t2 = i * 32 (size of row of x)
	addu	\$t2, \$t2, \$s1	# \$t2 = i * size(row) + j
	sll	\$t2, \$t2, 3	# \$t2 = byte offset of [i][j]
	addu	\$t2, \$a0, \$t2	# \$t2 = byte address of x[i][j]
	l.d	\$f4, 0(\$t2)	# \$f4 = 8 bytes of x[i][j]
L3:	sll	\$t0, \$s2, 5	# \$t0 = k * 32 (size of row of z)
	addu	\$t0, \$t0, \$s1	# \$t0 = k * size(row) + j
	sll	\$t0, \$t0, 3	# \$t0 = byte offset of [k][j]
	addu	\$t0, \$a2, \$t0	# \$t0 = byte address of z[k][j]
	l.d	\$f16, 0(\$t0)	# \$f16 = 8 bytes of z[k][j]

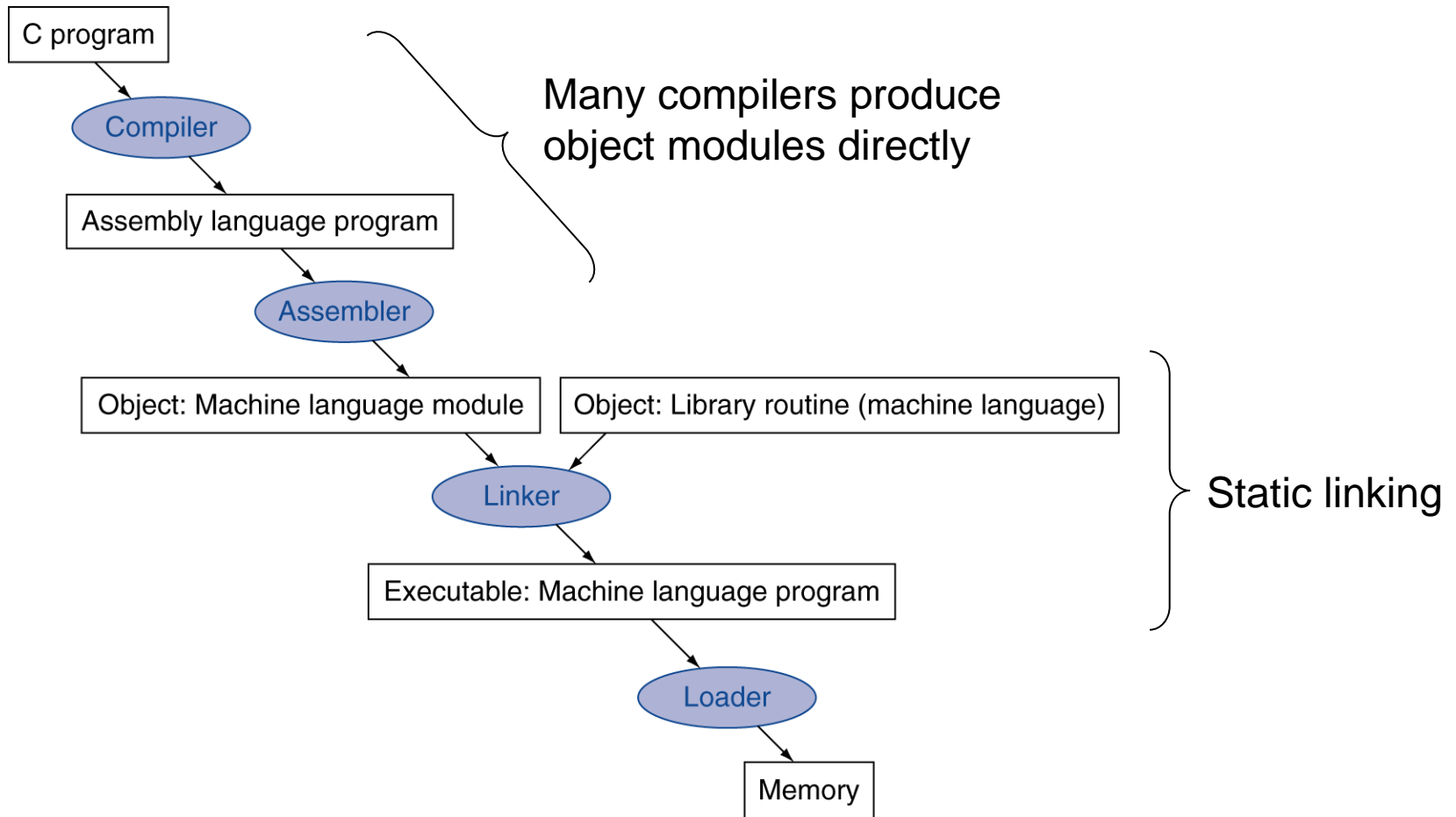
...

FP Example: Array Multiplication

...

sll	\$t0, \$s0, 5	# \$t0 = i*32 (size of row of y)
addu	\$t0, \$t0, \$s2	# \$t0 = i*size(row) + k
sll	\$t0, \$t0, 3	# \$t0 = byte offset of [i][k]
addu	\$t0, \$a1, \$t0	# \$t0 = byte address of y[i][k]
l.d	\$f18, 0(\$t0)	# \$f18 = 8 bytes of y[i][k]
mul.d	\$f16, \$f18, \$f16	# \$f16 = y[i][k] * z[k][j]
add.d	\$f4, \$f4, \$f16	# f4=x[i][j] + y[i][k]*z[k][j]
addiu	\$s2, \$s2, 1	# \$k k + 1
bne	\$s2, \$t1, L3	# if (k != 32) go to L3
s.d	\$f4, 0(\$t2)	# x[i][j] = \$f4
addiu	\$s1, \$s1, 1	# \$j = j + 1
bne	\$s1, \$t1, L2	# if (j != 32) go to L2
addiu	\$s0, \$s0, 1	# \$i = i + 1
bne	\$s0, \$t1, L1	# if (i != 32) go to L1

Translation and Startup



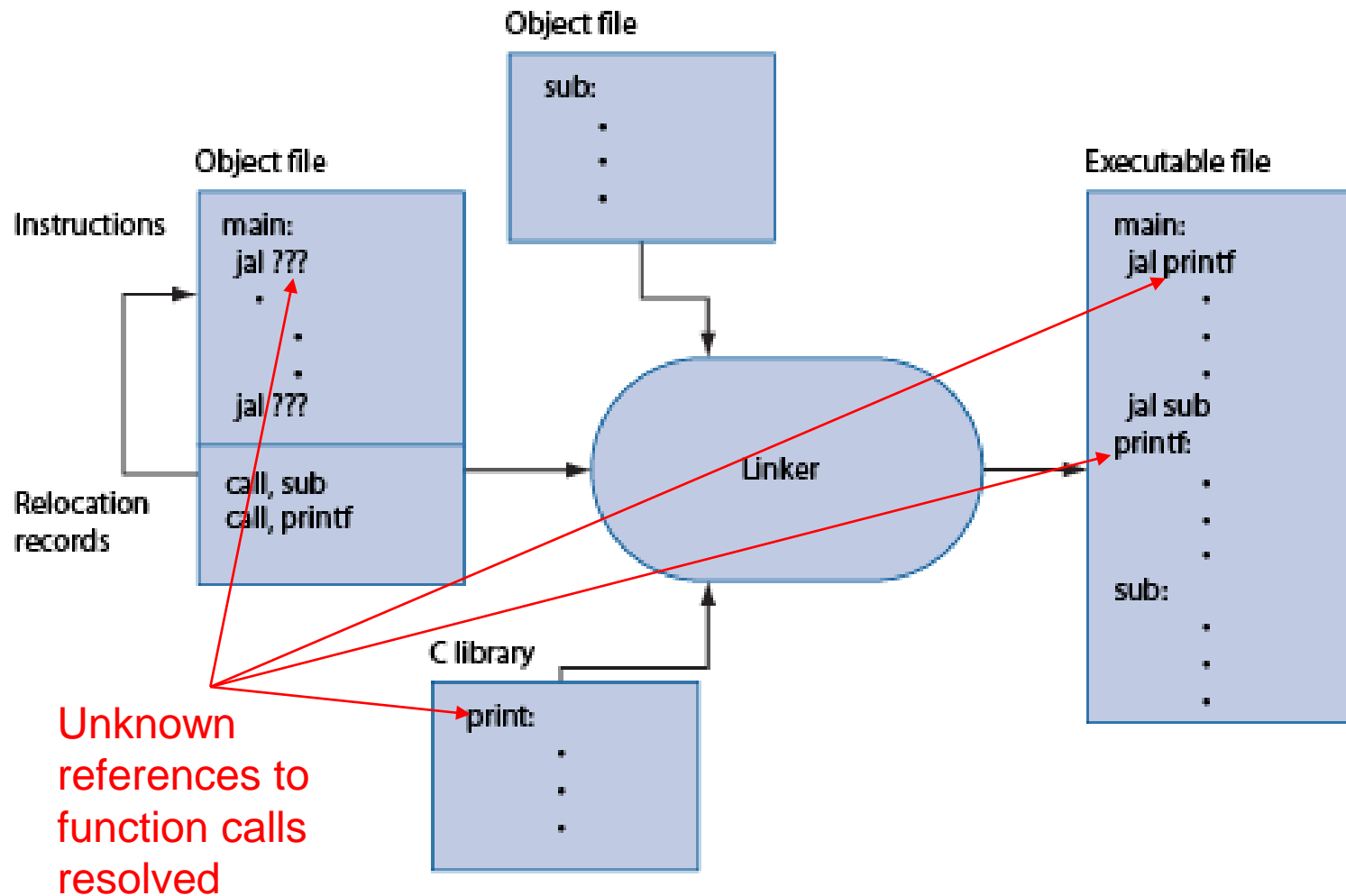
Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
 - Header: described contents of object module
 - Text segment: translated instructions
 - Static data segment: data allocated for the life of the program
 - Relocation info: for contents that depend on absolute location of loaded program
 - Symbol table: global definitions and external refs
 - Debug info: for associating with source code

Linking Object Modules

- Produces an executable image
 1. Merges segments
 2. Resolve labels (determine their addresses)
 3. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
 - But with virtual memory, no need to do this (will see later in memory architecture)
 - Program can be loaded into absolute location in virtual memory space

Linking example



Loading a Program

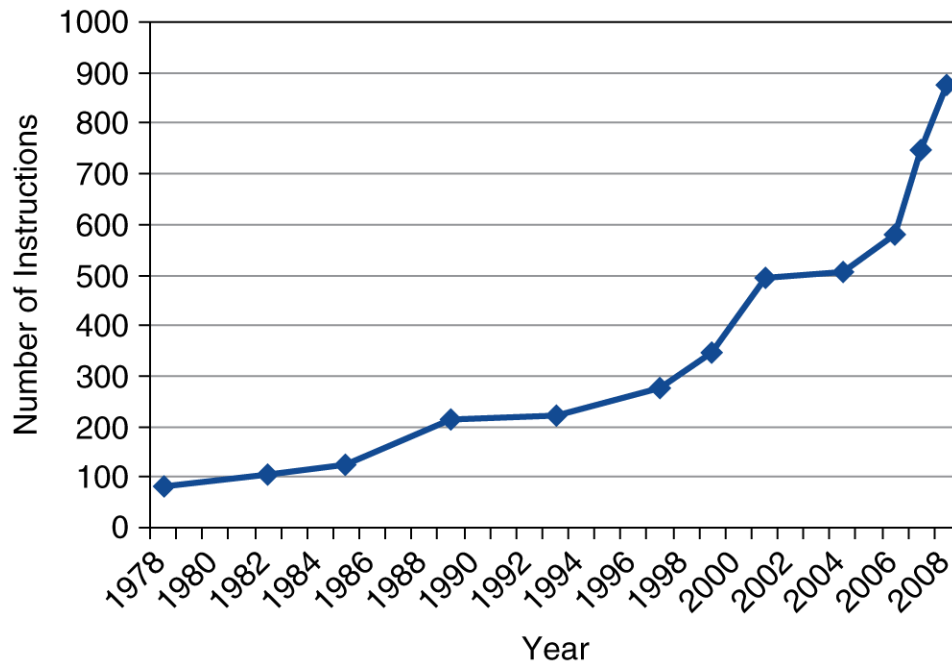
- Load from image file on disk into memory
 1. Read header to determine segment sizes
 2. Create virtual address space
 3. Copy text and initialized data into memory
 - Or set page table entries so they can be faulted in
 4. Set up arguments on stack
 5. Initialize registers (including `$sp`, `$fp`, `$gp`)
 6. Jump to startup routine
 - Copies arguments to `$a0`, ... and calls `main`
 - When `main` returns, do `exit` syscall

Fallacies

- Powerful instruction \Rightarrow higher performance
 - Fewer instructions required
 - But complex instructions are hard to implement
 - May slow down all instructions, including simple ones
 - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
 - But modern compilers are better at dealing with modern processors
 - More lines of code \Rightarrow more errors and less productivity

Fallacies

- Backward compatibility \Rightarrow instruction set doesn't change
 - But they do accrete more instructions



x86 instruction set

Pitfalls

- Sequential words are not at sequential addresses
 - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
 - e.g., passing pointer back via an argument
 - Pointer becomes invalid when stack popped

Concluding Remarks

- Design principles
 1. Simplicity favors regularity
 2. Smaller is faster
 3. Make the common case fast
 4. Good design demands good compromises
- Layers of software/hardware
 - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
 - c.f. x86

Concluding Remarks

- Measure MIPS instruction executions in benchmark programs
 - Consider making the common case fast
 - Consider compromises

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	add, sub, addi	16%	48%
Data transfer	lw, sw, lb, lbu, lh, lhu, sb, lui	35%	36%
Logical	and, or, nor, andi, ori, sll, srl	12%	4%
Cond. Branch	beq, bne, slt, slti, sltiu	34%	8%
Jump	j, jr, jal	2%	0%