

CS4100: 計算機結構

Instruction Set Architecture

國立清華大學資訊工程學系
一零零學年度第二學期

Outline

- ◆ Instruction set architecture (Sec 2.1)
- ◆ Operands
 - Register operands and their organization
 - Memory operands, data transfer
 - Immediate operands
- ◆ Signed and unsigned numbers
- ◆ Representing instructions
- ◆ Operations
 - Logical
 - Decision making and branches
- ◆ Supporting procedures in hardware
- ◆ Communicating with people
- ◆ Addressing for 32-bit immediate and addresses
- ◆ Translating and starting a program
- ◆ A sort example
- ◆ Arrays versus pointers
- ◆ ARM and x86 instruction sets

What Is Computer Architecture?

Computer Architecture =
Instruction Set Architecture
+ Machine Organization

- ◆ “... the attributes of a [computing] system as seen by the [assembly] language] programmer, i.e. the conceptual structure and functional behavior ...”

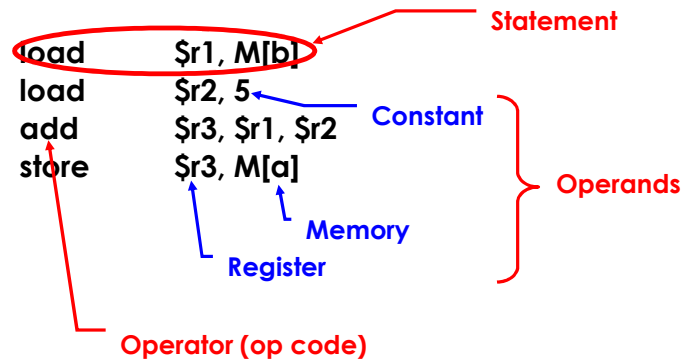
What are specified?

Recall in C Language

- ◆ Operators: +, -, *, /, % (mod), ...
 - $7/4=1$, $7\%4=3$
- ◆ Operands:
 - Variables: lower, upper, fahr, celsius
 - Constants: 0, 1000, -17, 15.4
- ◆ Assignment statement:
variable = expression
 - Expressions consist of operators operating on operands, e.g.,
`celsius = 5*(fahr-32)/9;`
`a = b+c+d-e;`

When Translating to Assembly ...

$a = b + 5;$



Components of an ISA

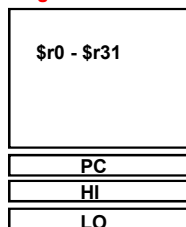
- ♦ Organization of programmable storage
 - registers
 - memory: flat, segmented
 - modes of addressing and accessing data items and instructions
- ♦ Data types and data structures
 - encoding and representation
- ♦ Instruction formats
- ♦ Instruction set (or operation code)
 - ALU, control transfer, exceptional handling

MIPS ISA as an Example

♦ Instruction categories:

- Load/Store
- Computational
- Jump and Branch
- Floating Point
- Memory Management
- Special

Registers



3 Instruction Formats: all 32 bits wide

OP	\$rs	\$rt	\$rd	sa	funct
OP	\$rs	\$rt	immediate		
OP	jump target				

Outline

- ♦ Instruction set architecture
- ♦ Operands (Sec 2.2, 2.3)
 - Register operands and their organization
 - Memory operands, data transfer
 - Immediate operands
- ♦ Signed and unsigned numbers
- ♦ Representing instructions
- ♦ Operations
 - Logical
 - Decision making and branches
- ♦ Supporting procedures in hardware
- ♦ Communicating with people
- ♦ Addressing for 32-bit immediate and addresses
- ♦ Translating and starting a program
- ♦ A sort example
- ♦ Arrays versus pointers
- ♦ ARM and x86 instruction sets

Operations of Hardware

- ♦ Syntax of basic MIPS arithmetic/logic instructions:

```
      1      2      3      4
add $s0,$s1,$s2    # f = g + h
```

- 1) operation by name
- 2) operand getting result ("destination")
- 3) 1st operand for operation ("source1")
- 4) 2nd operand for operation ("source2")

- ♦ Each instruction is **32** bits
- ♦ Syntax is rigid: 1 operator, 3 operands
 - Why? Keep hardware simple via regularity
- ♦ Design Principle 1: Simplicity favors regularity
 - Regularity makes implementation simpler
 - Simplicity enables higher performance at lower cost

Example

- ♦ How to do the following C statement?

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

Compiled 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
```

Operands and Registers

- ♦ Unlike high-level language, assembly don't use variables
=> assembly operands are registers
 - Limited number of special locations built directly into the hardware
 - Operations are performed on these
- ♦ Benefits:
 - Registers in hardware => faster than memory
 - Registers are easier for a compiler to use
 - e.g., as a place for temporary storage
 - Registers can hold variables to reduce memory traffic and improve code density (since register named with fewer bits than memory location)

MIPS Registers

- ♦ 32 registers, each is 32 bits wide
 - Why 32? Design Principle 2: smaller is faster
 - Groups of 32 bits called a **word** in MIPS
 - Registers are numbered from 0 to 31
 - Each can be referred to by number or name
 - Number references:
\$0, \$1, \$2, ... \$30, \$31
 - By convention, each register also has a name to make it easier to code, e.g.,
\$16 - \$22 → \$s0 - \$s7 (C variables)
\$8 - \$15 → \$t0 - \$t7 (temporary)
- ♦ 32 x 32-bit FP registers (paired DP)
- ♦ Others: HI, LO, PC

Registers Conventions for MIPS

0	zero	constant 0	16	s0	callee saves
1	at	reserved for assembler	...		(caller can clobber)
2	v0	expression evaluation &	23	s7	
3	v1	function results	24	t8	temporary (cont'd)
4	a0	arguments	25	t9	
5	a1		26	k0	reserved for OS kernel
6	a2		27	k1	
7	a3		28	gp	pointer to global area
8	t0	temporary: caller saves	29	sp	stack pointer
...		(callee can clobber)	30	fp	frame pointer
15	t7		31	ra	return address (HW)

Fig. 2.18

MIPS R2000 Organization

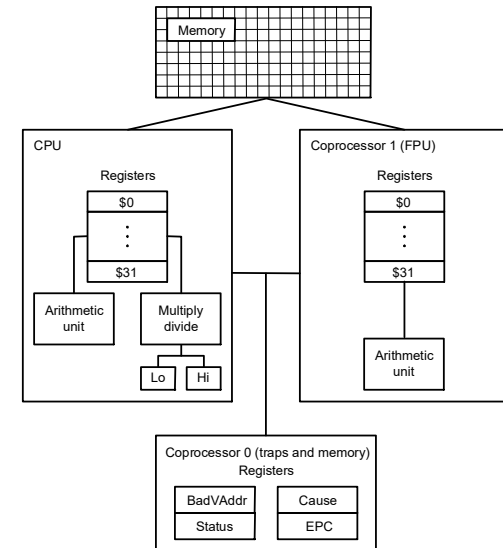


Fig. A.10.1

Example

- How to do the following C statement?

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

- f, ..., j in \$s0, ..., \$s4
- use intermediate temporary register t0, t1

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

Operations of Hardware

- Syntax of basic MIPS arithmetic/logic instructions:

```
      1   2   3   4
add $s0,$s1,$s2    # f = g + h
```

- operation by name
 - operand getting result ("destination")
 - 1st operand for operation ("source1")
 - 2nd operand for operation ("source2")
- Each instruction is **32** bits
 - Syntax is rigid: 1 operator, 3 operands
 - Why? Keep hardware simple via regularity
 - Design Principle 1: Simplicity favors regularity**
 - Regularity makes implementation simpler
 - Simplicity enables higher performance at lower cost

Register Architecture

◆ Accumulator (1 register):

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

◆ Stack:

0 address: `add tos ← tos + next`

◆ General Purpose Register:

2 address: `add A,B EA(A) ← EA(A) + EA(B)`
 3 address: `add A,B,C EA(A) ← EA(B) + EA(C)`

◆ Load/Store: (a special case of GPR)

3 address: `add $ra,$rb,$rc $ra ← $rb + $rc`
`load $ra,$rb $ra ← mem[$rb]`
`store $ra,$rb mem[$rb] ← $ra`

Register Organization Affects Programming

Code for $C = A + B$ for four register organizations:

Stack	Accumulator	Register (reg-mem)	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

=> Register organization is an attribute of ISA!

Comparison: Byte per instruction? Number of instructions? Cycles per instruction?

Since 1975 all machines use GPRs

Outline

- ◆ Instruction set architecture
- ◆ Operands (Sec 2.2, 2.3)
 - Register operands and their organization
 - Memory operands, data transfer
 - Immediate operands
- ◆ Signed and unsigned numbers
- ◆ Representing instructions
- ◆ Operations
 - Logical
 - Decision making and branches
- ◆ Supporting procedures in hardware
- ◆ Communicating with people
- ◆ Addressing for 32-bit immediate and addresses
- ◆ Translating and starting a program
- ◆ A sort example
- ◆ Arrays versus pointers
- ◆ ARM and x86 instruction sets

Memory Operands

- ◆ C variables map onto registers; what about large data structures like arrays?
 - Memory contains such data structures
- ◆ But MIPS arithmetic instructions operate on registers, not directly on memory
 - Data transfer instructions (`lw`, `sw`, ...) to transfer between memory and register
 - A way to address memory operands

Data Transfer: Memory to Register (1/2)

- ◆ To transfer a word of data, need to specify two things:
 - Register: specify this by number (0 - 31)
 - Memory address: more difficult
 - Think of memory as a 1D array
 - Address it by supplying a pointer to a memory address
 - Offset (in bytes) from this pointer
 - The desired memory address is the sum of these two values, e.g., 8 (\$t0)
 - Specifies the memory address pointed to by the value in \$t0, plus 8 bytes (why "bytes", not "words"?)
 - Each address is **32** bits

Data Transfer: Memory to Register (2/2)

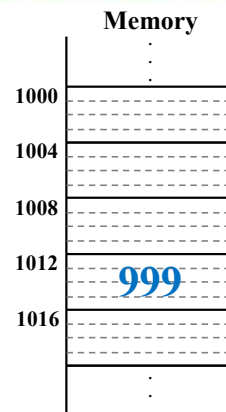
- ◆ Load Instruction Syntax:

1	2	3	4
<code>lw \$t0, 12(\$s0)</code>			

 - 1) operation name
 - 2) register that will receive value
 - 3) numerical offset in bytes
 - 4) register containing pointer to memory
- ◆ Example: `lw $t0, 12($s0)`
 - lw (Load Word, so a word (32 bits) is loaded at a time)
 - Take the pointer in \$s0, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register \$t0
- ◆ Notes:
 - \$s0 is called the *base register*, 12 is called the *offset*
 - Offset is generally used in accessing elements of array: base register points to the beginning of the array

Example

- ◆ \$s0 = 1000
- ◆ `lw $t0, 12($s0)`
- ◆ \$t0 = ?

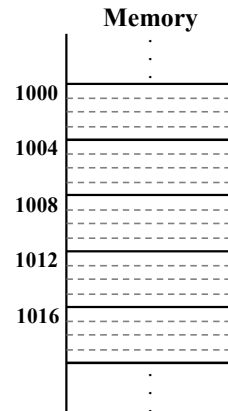


Data Transfer: Register to Memory

- ◆ Also want to store value from a register into memory
- ◆ Store instruction syntax is identical to Load instruction syntax
- ◆ Example: `sw $t0, 12($s0)`
 - sw (meaning Store Word, so 32 bits or one word are **stored** at a time)
 - This instruction will take the pointer in \$s0, add 12 bytes to it, and then store the value from register \$t0 into the memory address pointed to by the calculated sum

Example

- ◆ \$s0 = 1000
- ◆ \$t0 = 25
- ◆ sw \$t0, 12(\$s0)
- ◆ M[?] = 25



Compilation with Memory

- ◆ Compile by hand using registers:
\$s1:g, \$s2:h, \$s3:base address of A
 $g = h + A[8];$
- ◆ What offset in lw to select an array element A[8] in a C program?
 - 4x8=32 bytes to select A[8]
 - 1st transfer from memory to register:
lw \$t0, 32(\$s3) # \$t0 gets A[8]
 - Add 32 to \$s3 to select A[8], put into \$t0
- ◆ Next add it to h and place in g
add \$s1, \$s2, \$t0 # \$s1 = h+A[8]

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

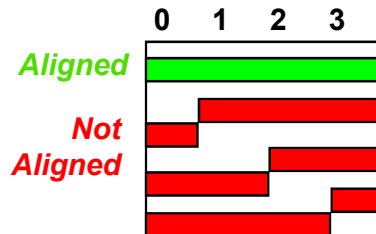
## Addressing: Byte versus Word

- ◆ Every word in memory has an address, similar to an index in an array
- ◆ Early computers numbered words like C numbers elements of an array:
  - Memory[0], Memory[1], Memory[2], ...

Called the "address" of a word
- ◆ Computers need to access 8-bit **bytes** as well as words (4 bytes/word)
- ◆ Today, machines address memory as bytes, hence word addresses differ by 4
  - Memory[0], Memory[4], Memory[8], ...
  - This is also why lw and sw use bytes in offset

## A Note about Memory: Alignment

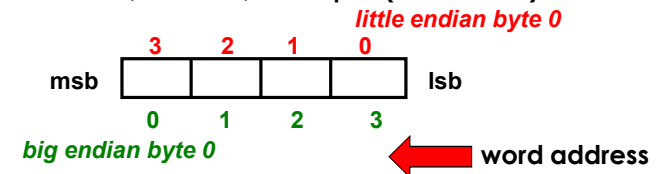
- ♦ MIPS requires that all words start at addresses that are multiples of 4 bytes



- ♦ Called **Alignment**: objects must fall on address that is multiple of their size

## Another Note: Endianness

- ♦ Byte order: numbering of bytes within a word
- ♦ Big Endian: address of most significant byte at least address of a word
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- ♦ Little Endian: address of least significant byte at least address
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



## Role of Registers vs. Memory

- ♦ What if more variables than registers?
  - Compiler tries to keep most frequently used variables in registers
  - Writes less common variables to memory: **spilling**
- ♦ Why not keep all variables in memory?
  - Smaller is faster: registers are faster than memory
  - Registers more versatile:
    - MIPS arithmetic instructions can read 2 registers, operate on them, and write 1 per instruction
    - MIPS data transfers only read or write 1 operand per instruction, and no operation

## Outline

- ♦ Instruction set architecture
- ♦ Operands (**Sec 2.2, 2.3**)
  - Register operands and their organization
  - Memory operands, data transfer
  - **Immediate operands**
- ♦ Signed and unsigned numbers
- ♦ Representing instructions
- ♦ Operations
  - Logical
  - Decision making and branches
- ♦ Supporting procedures in hardware
- ♦ Communicating with people
- ♦ Addressing for 32-bit immediate and addresses
- ♦ Translating and starting a program
- ♦ A sort example
- ♦ Arrays versus pointers
- ♦ ARM and x86 instruction sets



## Constants

- ♦ Small constants used frequently (50% of operands)  
e.g.,  $A = A + 5;$   
 $B = B + 1;$   
 $C = C - 18;$
- ♦ Put 'typical constants' in memory and load them
- ♦ Constant data specified in an instruction:  
`addi $29, $29, 4`  
`slli $8, $18, 10`  
`andi $29, $29, 6`  
`ori $29, $29, 4`
- ♦ **Design Principle 3:** Make the common case fast

## Immediate Operands

- ♦ Immediate: numerical *constants*
  - Often appear in code, so there are special instructions for them
  - **Add Immediate:**  
 $f = g + 10$  (in C)  
`addi $s0, $s1, 10` (in MIPS)  
where  $\$s0, \$s1$  are associated with  $f, g$
  - Syntax similar to add instruction, except that last argument is a number instead of a register
  - No subtract immediate instruction
    - Just use a negative constant  
`addi $s2, $s1, -1`

## The Constant Zero

- ♦ The number zero (0), appears very often in code; so we define register zero
- ♦ MIPS register 0 (\$zero) is the constant 0
  - Cannot be overwritten
  - This is defined in hardware, so an instruction like  
`addi $0, $0, 5` will not do anything
- ♦ Useful for common operations
  - E.g., move between registers  
`add $t2, $s1, $zero`

## Outline

- ♦ Instruction set architecture
- ♦ Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- ♦ **Signed and unsigned numbers (Sec 2.4, read by students)**
- ♦ Representing instructions
- ♦ Operations
  - Logical
  - Decision making and branches
- ♦ Supporting procedures in hardware
- ♦ Communicating with people
- ♦ Addressing for 32-bit immediate and addresses
- ♦ Translating and starting a program
- ♦ A sort example
- ♦ Arrays versus pointers
- ♦ ARM and x86 instruction sets

## Outline

- ◆ Instruction set architecture
- ◆ Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- ◆ Signed and unsigned numbers
- ◆ Representing instructions (Sec 2.5)
- ◆ Operations
  - Logical
  - Decision making and branches
- ◆ Supporting procedures in hardware
- ◆ Communicating with people
- ◆ Addressing for 32-bit immediate and addresses
- ◆ Translating and starting a program
- ◆ A sort example
- ◆ Arrays versus pointers
- ◆ ARM and x86 instruction sets

## Instructions as Numbers

- ◆ Currently we only work with words (32-bit blocks):
  - Each register is a word
  - `lw` and `sw` both access memory one word at a time
- ◆ So how do we represent instructions?
  - Remember: Computer only understands 1s and 0s, so “`add $t0, $0, $0`” is meaningless to hardware
  - MIPS wants simplicity: since data is in words, make instructions be words...

## MIPS Instruction Format

- ◆ One instruction is 32 bits  
=> divide instruction word into “fields”
  - Each field tells computer something about instruction
- ◆ We could define different fields for each instruction, but MIPS is based on simplicity, so define 3 basic types of instruction formats:
  - *R-format*: for register
  - *I-format*: for immediate, and `lw` and `sw` (since the offset counts as an immediate)
  - *J-format*: for jump

## R-Format Instructions (1/2)

- ◆ Define the following “fields”:

|        |    |    |    |       |       |
|--------|----|----|----|-------|-------|
| 6      | 5  | 5  | 5  | 5     | 6     |
| opcode | rs | rt | rd | shamt | funct |

- **opcode**: partially specifies what instruction it is (Note: 0 for all R-Format instructions)
- **funct**: combined with opcode to specify the instruction  
Question: Why aren't opcode and funct a single 12-bit field?
- **rs** (Source Register): generally used to specify register containing first operand
- **rt** (Target Register): generally used to specify register containing second operand
- **rd** (Destination Register): generally used to specify register which will receive result of computation

## R-Format Instructions (2/2)

### ◆ Notes about register fields:

- Each register field is exactly 5 bits, which means that it can specify any unsigned integer in the range 0-31. Each of these fields specifies one of the 32 registers by number.

### ◆ Final field:

- shamt:** contains the amount a shift instruction will shift by. Shifting a 32-bit word by more than 31 is useless, so this field is only 5 bits
- This field is set to 0 in all but the shift instructions

## 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    |
|---------|-------|-------|-------|-------|--------|
| 0       | 17    | 18    | 8     | 0     | 32     |
| 000000  | 10001 | 10010 | 01000 | 00000 | 100000 |

$00000010001100100100000000100000_2 = 02324020_{16}$

## Hexadecimal

### ◆ Base 16

- Compact representation of bit strings
- 4 bits per hex digit

|   |      |   |      |   |      |   |      |
|---|------|---|------|---|------|---|------|
| 0 | 0000 | 4 | 0100 | 8 | 1000 | c | 1100 |
| 1 | 0001 | 5 | 0101 | 9 | 1001 | d | 1101 |
| 2 | 0010 | 6 | 0110 | a | 1010 | e | 1110 |
| 3 | 0011 | 7 | 0111 | b | 1011 | f | 1111 |

### ◆ Example: eca8 6420

- 1110 1100 1010 1000 0110 0100 0010 0000

## I-Format Instructions

### ◆ Define the following "fields":

| 6      | 5  | 5  | 16        |
|--------|----|----|-----------|
| opcode | rs | rt | immediate |

- opcode:** uniquely specifies an I-format instruction
- rs:** specifies the *only* register operand
- rt:** specifies register which will receive result of computation (*target register*)
- addi, slti, immediate** is **sign-extended** to 32 bits, and treated as a signed integer
- 16 bits** → can be used to represent immediate up to  $2^{16}$  different values

## MIPS I-format Instructions

- ◆ **Design Principle 4: Good design demands good compromises**
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

## I-Format Example 1

- ◆ **MIPS Instruction:**  
addi \$21, \$22, -50
  - opcode = 8 (look up in table)
  - rs = 22 (register containing operand)
  - rt = 21 (target register)
  - immediate = -50 (by default, this is decimal)

decimal representation:

|   |    |    |     |
|---|----|----|-----|
| 8 | 22 | 21 | -50 |
|---|----|----|-----|

binary representation:

|        |       |       |                  |
|--------|-------|-------|------------------|
| 001000 | 10110 | 10101 | 1111111111001110 |
|--------|-------|-------|------------------|

## I-Format Example 2

- ◆ **MIPS Instruction:**  
lw \$t0, 1200(\$t1)
  - opcode = 35 (look up in table)
  - rs = 9 (base register)
  - rt = 8 (destination register)
  - immediate = 1200 (offset)

decimal representation:

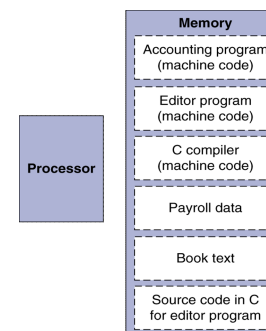
|    |   |   |      |
|----|---|---|------|
| 35 | 9 | 8 | 1200 |
|----|---|---|------|

binary representation:

|        |       |       |                  |
|--------|-------|-------|------------------|
| 100011 | 01001 | 01000 | 0000010010110000 |
|--------|-------|-------|------------------|

## Stored Program Computers

### The BIG Picture



- ◆ Instructions represented in binary, just like data
- ◆ Instructions and data stored in memory
- ◆ Programs can operate on programs
  - e.g., compilers, linkers, ...
- ◆ Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

## Outline

- ◆ Instruction set architecture
- ◆ Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- ◆ Signed and unsigned numbers
- ◆ Representing instructions
- ◆ Operations
  - Logical (Sec 2.6)
  - Decision making and branches
- ◆ Supporting procedures in hardware
- ◆ Communicating with people
- ◆ Addressing for 32-bit immediate and addresses
- ◆ Translating and starting a program
- ◆ A sort example
- ◆ Arrays versus pointers
- ◆ ARM and x86 instruction sets

## Bitwise Operations

- ◆ Up until now, we've done arithmetic (add, sub, addi) and memory access (lw and sw)
- ◆ All of these instructions view contents of register as a single quantity (such as a signed or unsigned integer)
- ◆ **New perspective:** View contents of register as 32 bits rather than as a single 32-bit number
- ◆ Since registers are composed of 32 bits, we may want to access individual bits rather than the whole.
- ◆ Introduce two new classes of instructions:
  - Shift instructions
  - Logical operators

## 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

## Shift Operations

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

- ◆ shamt: how many positions to shift
- ◆ Shift left logical
  - Shift left and fill with 0 bits
  - sll by  $i$  bits multiplies by  $2^i$
- ◆ Shift right logical
  - Shift right and fill with 0 bits
  - srl by  $i$  bits divides by  $2^i$  (unsigned only)

## Shift Instructions (1/3)

### ◆ Shift Instruction Syntax:

1      2      3      4  
sll    \$t2, \$s0, 4

- 1) operation name
- 2) register that will receive value
- 3) first operand (register)
- 4) shift amount (constant)

### ◆ MIPS has three shift instructions:

- sll (shift left logical): shifts left, fills empties with 0s
- srl (shift right logical): shifts right, fills empties with 0s
- sra (shift right arithmetic): shifts right, fills empties by sign extending

## Shift Instructions (2/3)

### ◆ Move (shift) all the bits in a word to the left or right by a number of bits, filling the emptied bits with 0s.

### ◆ Example: shift right by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

### ◆ Example: shift left by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0011 0100 0101 0110 0111 1000 0000 0000

## Shift Instructions (3/3)

### ◆ Example: shift right arithmetic by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

### ◆ Example: shift right arithmetic by 8 bits

1001 0010 0011 0100 0101 0110 0111 1000

1111 1111 1001 0010 0011 0100 0101 0110

## Uses for Shift Instructions

### ◆ Shift for multiplication: in binary

- Multiplying by 4 is same as shifting left by 2:

$$11_2 \times 100_2 = 1100_2$$

$$1010_2 \times 100_2 = 101000_2$$

- Multiplying by  $2^n$  is same as shifting left by n

### ◆ Since shifting is so much faster than multiplication (you can imagine how complicated multiplication is), a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:

a \*= 8;                      (in C)

would compile to:

sll    \$s0, \$s0, 3              (in MIPS)

## AND Operations

- ♦ Useful to mask bits in a word
    - Select some bits, clear others to 0
- and \$t0, \$t1, \$t2

|      |                                         |
|------|-----------------------------------------|
| \$t2 | 0000 0000 0000 0000 0000 1101 1100 0000 |
| \$t1 | 0000 0000 0000 0000 0011 1100 0000 0000 |
| \$t0 | 0000 0000 0000 0000 0000 1100 0000 0000 |

## OR Operations

- ♦ Useful to include bits in a word
    - Set some bits to 1, leave others unchanged
- or \$t0, \$t1, \$t2

|      |                                         |
|------|-----------------------------------------|
| \$t2 | 0000 0000 0000 0000 0000 1101 1100 0000 |
| \$t1 | 0000 0000 0000 0000 0011 1100 0000 0000 |
| \$t0 | 0000 0000 0000 0000 0011 1101 1100 0000 |

## NOT Operations

- ♦ Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- ♦ MIPS has NOR 3-operand instruction
  - $a \text{ NOR } b == \text{NOT}(a \text{ OR } b)$

nor \$t0, \$t1, \$zero

← Register 0: always read as zero

|      |                                         |
|------|-----------------------------------------|
| \$t1 | 0000 0000 0000 0000 0011 1100 0000 0000 |
| \$t0 | 1111 1111 1111 1111 1100 0011 1111 1111 |

## So Far...

- ♦ All instructions have allowed us to manipulate data.
- ♦ So we've built a calculator.
- ♦ In order to build a computer, we need ability to make decisions...

## Outline

- ◆ Instruction set architecture
- ◆ Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- ◆ Signed and unsigned numbers
- ◆ Representing instructions
- ◆ Operations
  - Logical
  - **Decision making and branches (Sec 2.7)**
- ◆ Supporting procedures in hardware
- ◆ Communicating with people
- ◆ Addressing for 32-bit immediate and addresses
- ◆ Translating and starting a program
- ◆ A sort example
- ◆ Arrays versus pointers
- ◆ ARM and x86 instruction sets

## MIPS Decision Instructions

`beq register1, register2, L1`

- ◆ Decision instruction in MIPS:
 

```
beq register1, register2, L1
```

“Branch if (registers are) equal”  
meaning:  
`if (register1==register2) goto L1`
- ◆ Complementary MIPS decision instruction
 

```
bne register1, register2, L1
```

“Branch if (registers are) not equal”  
meaning:  
`if (register1!=register2) goto L1`
- ◆ These are called **conditional branches**

## MIPS Goto Instruction

`j label`

- ◆ MIPS has an **unconditional branch**:
 

```
j label
```

  - Called a Jump Instruction: jump directly to the given label without testing any condition
  - meaning:  
`goto label`
- ◆ Technically, it's the same as:
 

```
beq $0, $0, label
```

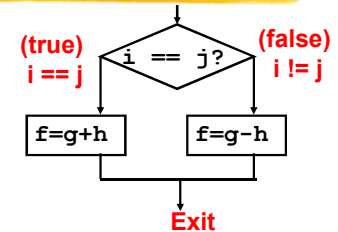
since it always satisfies the condition
- ◆ It has the j-type instruction format

## Compiling C if into MIPS

- ◆ Compile by hand
 

```
if (i == j) f=g+h;
else f=g-h;
```
- ◆ Use this mapping:
 

```
f, g, ..., j : $s0, $s1, $s2,
 $s3, $s4
```



- ◆ Final compiled MIPS code:
 

|                                                                                                                    |                                                                                            |
|--------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| <pre> bne \$s3, \$s4, Else add \$s0, \$s1, \$s2 j Exit Else: sub \$s0, \$s1, \$s2 Exit:                     </pre> | <pre> # branch i!=j # f=g+h (true) # go to Exit # f=g-h (false)                     </pre> |
|--------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|

**Note:** Compiler automatically creates labels to handle decisions (branches) appropriately



## 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:

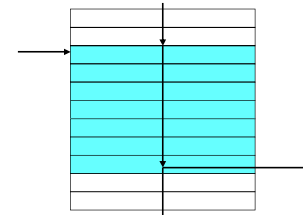
```
Loop: sll $t1, $s3, 2 # $t1 = i x 4
 add $t1, $t1, $s6 # $t1 = addr of save[i]
 lw $t0, 0($t1) # $t0 = save[i]
 bne $t0, $s5, Exit # if save[i] != k goto Exit
 addi $s3, $s3, 1 # i = i + 1
 j Loop # goto Loop

Exit: ...
```

## Basic Blocks

### ♦ A basic block is a sequence of instructions with

- No embedded branches (except at end)
- No branch targets (except at beginning)



### ♦ A compiler identifies basic blocks for optimization

### ♦ An advanced processor can accelerate execution of basic blocks

## Inequalities in MIPS

### ♦ Until now, we've only tested equalities (== and != in C), but general programs need to test < and >

### ♦ Set on Less Than:

- slt rd, rs, rt
  - if (rs < rt) rd = 1; else rd = 0;
- slti rt, rs, constant
  - if (rs < constant) rt = 1; else rt = 0;

Compile by hand: if (g < h) goto Less;  
Let g: \$s0, h: \$s1

```
slt $t0, $s0, $s1 # $t0 = 1 if g < h
bne $t0, $0, Less # goto Less if $t0 != 0
```

### ♦ MIPS has no "branch on less than" => too complex

## Branch Instruction Design

### ♦ Why not blt, bge, etc?

### ♦ Hardware for <, ≥, ... slower than =, ≠

- Combining with branch involves more work per instruction, requiring a slower clock
- All instructions penalized!

### ♦ beq and bne are the common case

### ♦ This is a good design compromise

## Signed vs. Unsigned

- ♦ Signed comparison: `slt`, `slti`
- ♦ Unsigned comparison: `sltu`, `sltui`
- ♦ Example
  - `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
  - `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`
  - `slt $t0, $s0, $s1 # signed`
    - $-1 < +1 \Rightarrow \$t0 = 1$
  - `sltu $t0, $s0, $s1 # unsigned`
    - $+4,294,967,295 > +1 \Rightarrow \$t0 = 0$

## Outline

- ♦ Instruction set architecture (using MIPS ISA as an example)
- ♦ Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- ♦ Signed and unsigned numbers
- ♦ Representing instructions
- ♦ Operations
  - Logical
  - Decision making and branches
- ♦ Supporting procedures in hardware (Sec. 2.8)
- ♦ Communicating with people
- ♦ Addressing for 32-bit immediate and addresses
- ♦ Translating and starting a program
- ♦ A sort example
- ♦ Arrays versus pointers
- ♦ ARM and x86 instruction sets

## Procedure Calling

- ♦ Steps required

Caller:

1. Place parameters in registers
2. Transfer control to procedure

Callee:

3. Acquire storage for procedure
4. Perform procedure's operations
5. Place result in register for caller
6. Return to place of call

## C Function Call Bookkeeping

```
sum = leaf_example(a,b,c,d) . . .
```

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

- ♦ Return address      `$ra`
- ♦ Procedure address    `Labels`
- ♦ Arguments            `$a0, $a1, $a2, $a3`
- ♦ Return value         `$v0, $v1`
- ♦ Local variables      `$s0, $s1, ..., $s7`

Note the use of **register conventions**

## Registers Conventions for MIPS

|     |      |                         |     |    |                        |
|-----|------|-------------------------|-----|----|------------------------|
| 0   | zero | constant 0              | 16  | s0 | callee saves           |
| 1   | at   | reserved for assembler  | ... |    | (caller can clobber)   |
| 2   | v0   | expression evaluation & | 23  | s7 |                        |
| 3   | v1   | function results        | 24  | t8 | temporary (cont'd)     |
| 4   | a0   | arguments               | 25  | t9 |                        |
| 5   | a1   |                         | 26  | k0 | reserved for OS kernel |
| 6   | a2   |                         | 27  | k1 |                        |
| 7   | a3   |                         | 28  | gp | pointer to global area |
| 8   | t0   | temporary: caller saves | 29  | sp | stack pointer          |
| ... |      | (callee can clobber)    | 30  | fp | frame pointer          |
| 15  | t7   |                         | 31  | ra | return address (HW)    |

Fig. 2.18

## Procedure Call Instructions

- ◆ Procedure call: jump and link  
`jal ProcedureLabel`
  - Address of following instruction put in \$ra
  - Jumps to target address (i.e., ProcedureLabel)
- ◆ Procedure return: jump register  
`jr $ra`
  - Copies \$ra to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements
      - ✱ Jump table is an array of addresses corresponding to labels in codes
      - ✱ Load appropriate entry to register
      - ✱ Jump register

## Caller's Code

- ◆ ...  
`sum = leaf_example(a,b,c,d)`  
...
- ◆ MIPS code: a, ..., d in \$s0, ..., \$s3, and sum in \$s4

```

:
add $a0, $0, $s0
add $a1, $0, $s1
add $a2, $0, $s2
add $a3, $0, $s3
jal leaf_example
add $s4, $0, $v0
:

```

Move a,b,c,d to a0..a3

Jump to leaf\_example

Move result in v0 to sum

## 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 in \$s0 (hence, need to save \$s0 on stack)
  - Save \$t1 and \$t2
  - Result in \$v0

## Leaf Procedure Example

♦ MIPS code:

leaf\_example:

addi \$sp, \$sp, -12

sw \$s0, 0(\$sp)

sw \$t0, 4(\$sp)

sw \$t1, 8(\$sp)

Save \$s0 \$t0 \$t1 on stack

add \$t0, \$a0, \$a1

add \$t1, \$a2, \$a3

sub \$s0, \$t0, \$t1

Procedure body

add \$v0, \$s0, \$zero

Result

lw \$s0, 0(\$sp)

lw \$t0, 4(\$sp)

lw \$t1, 8(\$sp)

Restore \$s0 \$t0 \$t1

addi \$sp, \$sp, 12

jr \$ra

Return

## 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 in \$s0 (hence, need to save \$s0 on stack)
- \$t1 and \$t2 are not saved on stack
- Result in \$v0

## Leaf Procedure Example

♦ MIPS code:

leaf\_example:

addi \$sp, \$sp, -4

sw \$s0, 0(\$sp)

Save \$s0 on stack

add \$t0, \$a0, \$a1

add \$t1, \$a2, \$a3

sub \$s0, \$t0, \$t1

Procedure body

add \$v0, \$s0, \$zero

Result

lw \$s0, 0(\$sp)

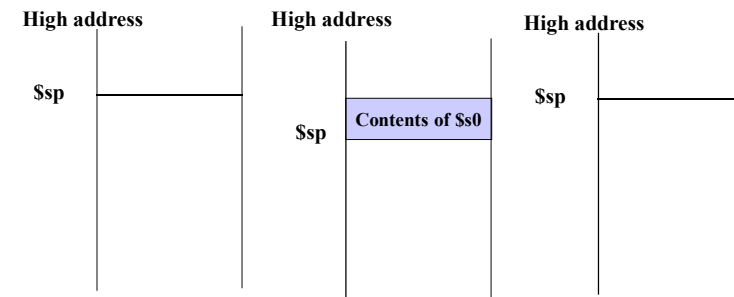
Restore \$s0

addi \$sp, \$sp, 4

jr \$ra

Return

## Local Data on the Stack



## 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 (because callee will not save them)
- ◆ 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);
}
```

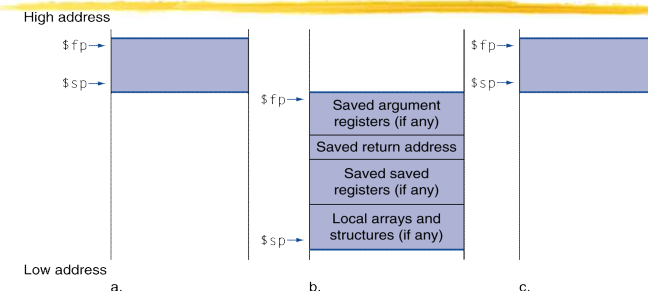
  - Argument n in \$a0
  - Result in \$v0

## 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               |

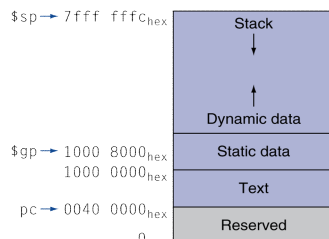
## Local Data on the Stack



- ◆ Local data allocated by callee
  - e.g., C automatic variables
- ◆ Procedure frame (activation record)
  - Used by some compilers to manage stack storage

## Memory Layout

- ◆ Text: program code
- ◆ Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - \$gp initialized to address allowing  $\pm$ offsets into this segment
- ◆ Dynamic data: heap
  - E.g., malloc in C, new in Java
- ◆ Stack: automatic storage



## Why Procedure Conventions?

- ◆ Definitions
  - Caller: function making the call, using jal
  - Callee: function being called
- ◆ Procedure conventions as a contract between the Caller and the Callee
- ◆ If both the Caller and Callee obey the procedure conventions, there are significant benefits
  - People who have never seen or even communicated with each other can write functions that work together
  - Recursion functions work correctly

## Caller's Rights, Callee's Rights

- ◆ Callee's rights:
  - Right to use VAT registers freely
  - Right to assume arguments are passed correctly
- ◆ To ensure callee's right, caller saves registers:
  - Return address      \$ra
  - Arguments            \$a0, \$a1, \$a2, \$a3
  - Return value        \$v0, \$v1
  - \$t Registers        \$t0 - \$t9
- ◆ Callers' rights:
  - Right to use \$ registers without fear of being overwritten by callee
  - Right to assume return value will be returned correctly
- ◆ To ensure caller's right, callee saves registers:
  - \$s Registers        \$s0 - \$s7

## Contract in Function Calls (1/2)

- ◆ Caller's responsibilities (how to call a function)
  - Slide \$sp down to reserve memory: e.g., addi \$sp, \$sp, -28
  - Save \$ra on stack because jal clobbers it: e.g., sw \$ra, 24(\$sp)
  - If still need their values after function call, save \$v, \$a, \$t on stack or copy to \$s registers
  - Put first 4 words of arguments in \$a0-3, additional arguments go on stack: "a4" is 16(\$sp)
  - jal to the desired function
  - Receive return values in \$v0, \$v1
  - Undo first steps: e.g. lw \$t0, 20(\$sp) lw \$ra, 24(\$sp) addi \$sp, \$sp, 28

## Contract in Function Calls (2/2)

- ◆ Callee's responsibilities (i.e. how to write a function)
  - If using `$s` or big local structures, slide `$sp` down to reserve memory: e.g., `addi $sp, $sp, -48`
  - If using `$s`, save before using: e.g.,  
`sw $s0, 44($sp)`
  - Receive arguments in `$a0-3`, additional arguments on stack
  - Run the procedure body
  - If not void, put return values in `$v0, 1`
  - If applicable, undo first two steps: e.g.,  
`lw $s0, 44($sp)      addi $sp, $sp, 48`
  - `jr $ra`

## Outline

- ◆ Instruction set architecture (using MIPS ISA as an example)
- ◆ Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- ◆ Signed and unsigned numbers
- ◆ Representing instructions
- ◆ Operations
  - Logical
  - Decision making and branches
- ◆ Supporting procedures in hardware
- ◆ Communicating with people (Sec. 2.9)
- ◆ Addressing for 32-bit immediate and addresses
- ◆ Translating and starting a program
- ◆ A sort example
- ◆ Arrays versus pointers
- ◆ ARM and x86 instruction sets

## 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

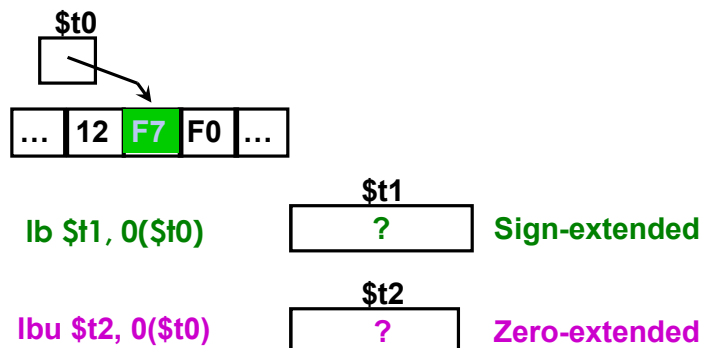
## Byte/Halfword Operations

- ◆ 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

## Load Byte Signed/Unsigned



## 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

Outline

- ♦ Instruction set architecture (using MIPS ISA as an example)
- ♦ Operands
 - Register operands and their organization
 - Memory operands, data transfer
 - Immediate operands
- ♦ Signed and unsigned numbers
- ♦ Representing instructions
- ♦ Operations
 - Logical
 - Decision making and branches
- ♦ Supporting procedures in hardware
- ♦ Communicating with people
- ♦ Addressing for 32-bit immediate and addresses (Sec. 2.10)
- ♦ Translating and starting a program
- ♦ A sort example
- ♦ Arrays versus pointers
- ♦ ARM and x86 instruction sets

32-bit Constants

- ♦ Most constants are small
 - 16-bit immediate is sufficient
- ♦ For the occasional 32-bit constant
- ♦ Load Upper Immediate:
 - lui rt, constant
 - Copies 16-bit constant to left 16 bits of rt
 - Clears right 16 bits of rt to 0

machine version of lui 001111 00000 10000 0000 0000 0011 1101

lui \$s0, 61 0000 0000 0011 1101 0000 0000 0000 0000 \$s0

ori \$s0, \$s0, 2304 0000 0000 0011 1101 0000 1001 0000 0000 \$s0

Branch Addressing (1)

- ♦ Use I-format:

opcode	rs	rt	immediate
--------	----	----	-----------

- opcode specifies beq or bne
- Rs and Rt specify registers to compare
- ♦ What can *immediate* specify? **PC-relative addressing**
 - *Immediate* is only 16 bits, but PC is 32-bit
=> *immediate* cannot specify entire address
 - Loops are generally small: < 50 instructions
 - Though we want to branch to anywhere in memory, a single branch only need to change **PC** by a small amount
 - How to use PC-relative addressing
 - 16-bit *immediate* as a signed two's complement integer to be added to the PC if branch taken

Branch Addressing (2)

- ♦ *Immediate* specifies **word address**
 - Instructions are word aligned (byte address is always a multiple of 4, i.e., it ends with 00 in binary)
 - The number of bytes to add to the PC will always be a multiple of 4
 - Specify the *immediate* in words (confusing?)
 - Now, we can branch +/- 2¹⁵ **words** from the PC (or +/- 2¹⁷ bytes), handle loops 4 times as large
- ♦ *Immediate* specifies **PC + 4**
 - Due to hardware, add *immediate* to (PC+4), not to PC
 - If branch not taken: PC = PC + 4
 - If branch taken: PC = (PC+4) + (*immediate**4)

Branch Example

- ♦ MIPS Code:

```

Loop: beq    $9,$0,End
      add    $8,$8,$10
      addi   $9,$9,-1
      j      Loop

End:
    
```

- ♦ Branch is I-Format:

opcode	rs	rt	immediate
--------	----	----	-----------

opcode = 4 (look up in table)
rs = 9 (first operand)
rt = 0 (second operand)
immediate = ???

- Number of instructions to add to (or subtract from) the PC, starting at the instruction **following** the branch
=> *immediate* = ?

Branch Example

♦ MIPS Code:

```

Loop: beq    $9,$0,End
      add    $8,$8,$10
      addi   $9,$9,-1
      j      Loop

```

End:

decimal representation:

4	9	0	3
---	---	---	---

binary representation:

000100	01001	00000	00000000000000011
--------	-------	-------	-------------------

Jump Addressing (1/3)

- ♦ For branches, we assumed that we won't want to branch too far, so we can specify change in PC.
- ♦ For general jumps (j and jal), we may jump to anywhere in memory.
- ♦ Ideally, we could specify a 32-bit memory address to jump to.
- ♦ Unfortunately, we can't fit both a 6-bit opcode and a 32-bit address into a single 32-bit word, so we compromise.

Jump Addressing (2/3)

- ♦ Define "fields" of the following number of bits each:

6 bits	26 bits
--------	---------

- ♦ As usual, each field has a name:

opcode	target address
--------	----------------

♦ Key concepts:

- Keep opcode field identical to R-format and I-format for consistency
- Combine other fields to make room for target address

♦ Optimization:

- Jumps only jump to word aligned addresses
 - last two bits are always 00 (in binary)
 - specify 28 bits of the 32-bit bit address

Jump Addressing (3/3)

- ♦ Where do we get the other 4 bits?
 - Take the 4 highest order bits from the PC
 - Technically, this means that we cannot jump to anywhere in memory, but it's adequate 99.9999...% of the time, since programs aren't that long
 - Linker and loader avoid placing a program across an address boundary of 256 MB
- ♦ Summary:
 - New PC = PC[31..28] || target address (26 bits) || 00
 - Note: || means concatenation
4 bits || 26 bits || 2 bits = 32-bit address
- ♦ If we absolutely need to specify a 32-bit address:
 - Use jr \$ra # jump to the address specified by \$ra

Target Addressing Example

- Loop code from earlier example
 - Assume Loop at location 80000

Loop: sll \$t1, \$s3, 2	80000	0	0	19	9	4	0
add \$t1, \$t1, \$s6	80004	0	9	22	9	0	32
lw \$t0, 0(\$t1)	80008	35	9	8			0
bne \$t0, \$s5, Exit	80012	5	8	21			2
addi \$s3, \$s3, 1	80016	8	19	19			1
j Loop	80020	2					20000
Exit: ...	80024						

- $80016 + 2 \times 4 = 80024$
- $20000 \times 4 = 80000$

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code

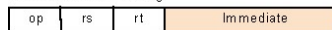
- Example

```

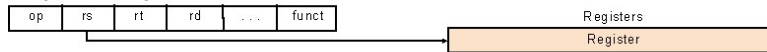
beq $s0,$s1, L1
      ↓
bne $s0,$s1, L2
j L1
L2: ...
    
```

MIPS Addressing Mode

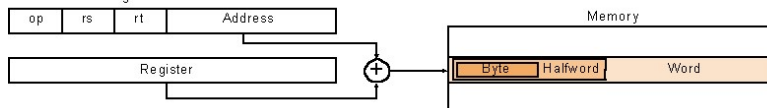
1. Immediate addressing



2. Register addressing

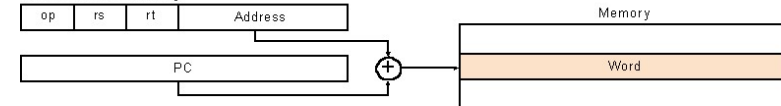


3. Base addressing

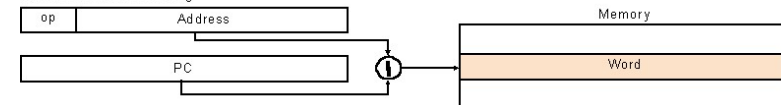


MPIS Addressing Modes

4. PC-relative addressing



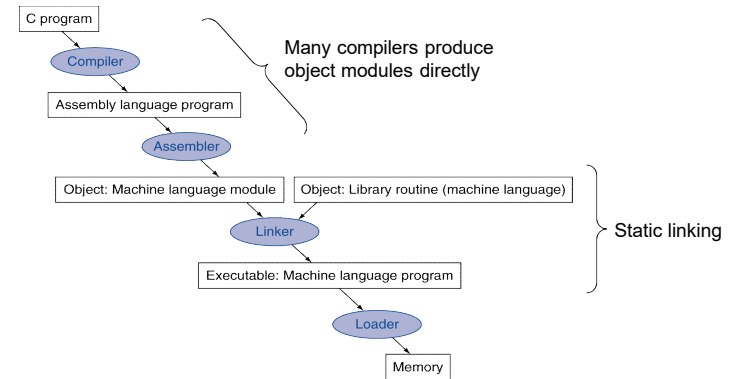
5. Pseudodirect addressing



Outline

- ◆ Instruction set architecture (using MIPS ISA as an example)
- ◆ Operands
 - Register operands and their organization
 - Memory operands, data transfer
 - Immediate operands
- ◆ Signed and unsigned numbers
- ◆ Representing instructions
- ◆ Operations
 - Logical
 - Decision making and branches
- ◆ Supporting procedures in hardware
- ◆ Communicating with people
- ◆ Addressing for 32-bit immediate and addresses
- ◆ Translating and starting a program (Sec. 2.12)
- ◆ A sort example
- ◆ Arrays versus pointers
- ◆ ARM and x86 instruction sets

Translation and Startup



Assembler Pseudoinstructions

- ◆ Most assembler instructions represent machine instructions one-to-one
- ◆ Pseudo instructions: figments of the assembler's imagination
 - move \$t0, \$t1 → add \$t0, \$zero, \$t1
 - blt \$t0, \$t1, L → slt \$at, \$t0, \$t1
 - bne \$at, \$zero, L
 - \$at (register 1): assembler temporary

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
 - Program can be loaded into absolute location in virtual memory space

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

Outline

- ◆ Instruction set architecture (using MIPS ISA as an example)
- ◆ Operands
 - Register operands and their organization
 - Memory operands, data transfer
 - Immediate operands
- ◆ Signed and unsigned numbers
- ◆ Representing instructions
- ◆ Operations
 - Logical
 - Decision making and branches
- ◆ Supporting procedures in hardware
- ◆ Communicating with people
- ◆ Addressing for 32-bit immediate and addresses
- ◆ Translating and starting a program
- ◆ A sort example (Sec. 2.13)
- ◆ Arrays versus pointers
- ◆ ARM and x86 instruction sets

C Sort Example

- ◆ Illustrates use of assembly instructions for a C bubble sort function
- ◆ Swap procedure (leaf)

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

 - v in \$a0, k in \$a1, temp in \$t0

The Procedure Swap

```

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

```

The Sort Procedure in C

♦ Non-leaf (calls swap)

```

void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1;
              j >= 0 && v[j] > v[j + 1];
              j -= 1) {
            swap(v, j);
        }
    }
}

```

• v in \$a0, k in \$a1, i in \$s0, j in \$s1

The Procedure Body

move \$s2, \$a0	# save \$a0 into \$s2	Move
move \$s3, \$a1	# save \$a1 into \$s3	params
move \$s0, \$zero	# i = 0	Outer loop
for1tst: slt \$t0, \$s0, \$s3	# \$t0 = 0 if \$s0 ≥ \$s3 (i ≥ n)	
beq \$t0, \$zero, exit1	# go to exit1 if \$s0 ≥ \$s3 (i ≥ n)	
addi \$s1, \$s0, -1	# j = i - 1	
for2tst: slti \$t0, \$s1, 0	# \$t0 = 1 if \$s1 < 0 (j < 0)	
bne \$t0, \$zero, exit2	# go to exit2 if \$s1 < 0 (j < 0)	
sll \$t1, \$s1, 2	# \$t1 = j * 4	Inner loop
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 \$t4 ≥ \$t3	
beq \$t0, \$zero, exit2	# go to exit2 if \$t4 ≥ \$t3	
move \$a0, \$s2	# 1st param of swap is v (old \$a0)	Pass
move \$a1, \$s1	# 2nd param of swap is j	params
jal swap	# call swap procedure	& call
addi \$s1, \$s1, -1	# j -- 1	Inner loop
j for2tst	# jump to test of inner loop	
exit2: addi \$s0, \$s0, 1	# i += 1	Outer loop
j for1tst	# jump to test of outer loop	

The Full Procedure

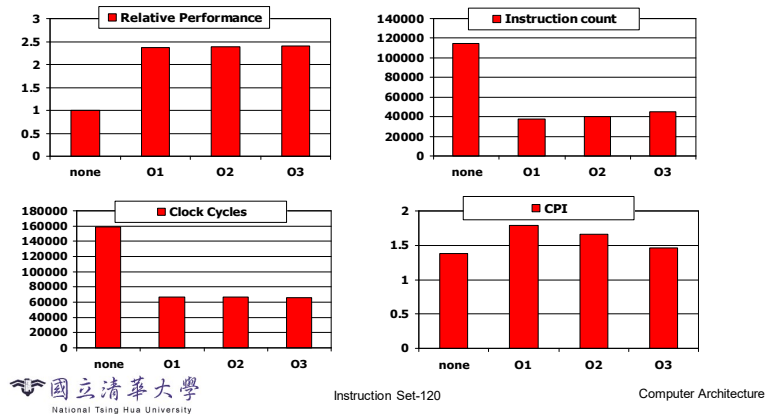
```

sort:  addi $sp,$sp, -20    # make room on stack for 5 registers
      sw $ra, 16($sp)     # save $ra on stack
      sw $s3, 12($sp)     # save $s3 on stack
      sw $s2, 8($sp)      # save $s2 on stack
      sw $s1, 4($sp)      # save $s1 on stack
      sw $s0, 0($sp)      # save $s0 on stack
      ...                 # procedure body
      ...
exit1: lw $s0, 0($sp)      # restore $s0 from stack
      lw $s1, 4($sp)      # restore $s1 from stack
      lw $s2, 8($sp)      # restore $s2 from stack
      lw $s3, 12($sp)     # restore $s3 from stack
      lw $ra, 16($sp)     # restore $ra from stack
      addi $sp,$sp, 20    # restore stack pointer
      jr $ra              # return to calling routine

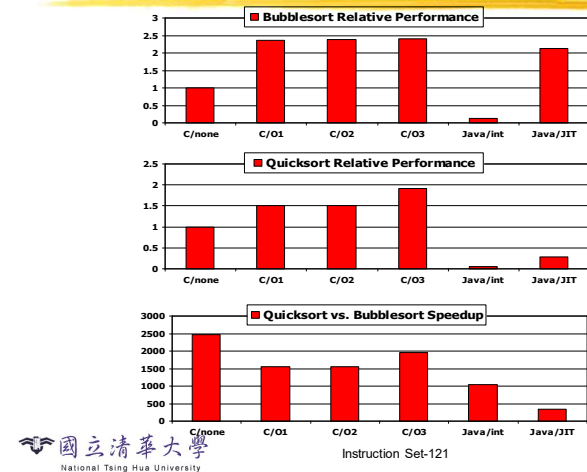
```

Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux



Effect of Language and Algorithm



Lessons Learnt

- ◆ Instruction count and CPI are not good performance indicators in isolation
- ◆ Compiler optimizations are sensitive to the algorithm
- ◆ Nothing can fix a dumb algorithm!

Outline

- ◆ Instruction set architecture (using MIPS ISA as an example)
- ◆ Operands
 - Register operands and their organization
 - Memory operands, data transfer
 - Immediate operands
- ◆ Signed and unsigned numbers
- ◆ Representing instructions
- ◆ Operations
 - Logical
 - Decision making and branches
- ◆ Supporting procedures in hardware
- ◆ Communicating with people
- ◆ Addressing for 32-bit immediate and addresses
- ◆ Translating and starting a program (Sec. 2.12)
- ◆ A sort example
- ◆ Arrays versus pointers (Sec. 2.14)
- ◆ ARM and x86 instruction sets

Arrays vs. Pointers

- ♦ Array indexing involves
 - Multiplying index by element size
 - Adding to array base address
- ♦ Pointers correspond directly to memory addresses
 - Can avoid indexing complexity

Example: Clearing an Array

<pre>clear1(int array[], int size) { int i; for (i = 0; i < size; i += 1) array[i] = 0; }</pre>	<pre>clear2(int *array, int size) { int *p; for (p = &array[0]; p < &array[size]; p = p + 1) *p = 0; }</pre>
<pre>loop1: move \$t0,\$zero # i = 0 sll \$t1,\$t0,2 # \$t1 = i * 4 add \$t2,\$a0,\$t1 # \$t2 = # &array[i] sw \$zero, 0(\$t2) # array[i] = 0 addi \$t0,\$t0,1 # i = i + 1 slt \$t3,\$t0,\$a1 # \$t3 = # (i < size) bne \$t3,\$zero,loop1 # if (...) # goto loop1</pre>	<pre> move \$t0, \$a0 # p = & array[0] sll \$t1, \$a1, 2 # \$t1 = size * 4 add \$t2,\$a0,\$t1 # \$t2 = # &array[size] loop2: sw \$zero,0(\$t0) # Memory[p] = 0 addi \$t0,\$t0,4 # p = p + 4 slt \$t3,\$t0,\$t2 # \$t3 = # (p<&array[size]) bne \$t3,\$zero,loop2 # if (...) # goto loop2</pre>

Comparison of Array vs. Ptr

- ♦ Multiply “strength reduced” to shift (strength reduction)
- ♦ Array version requires shift to be inside loop
 - Part of index calculation for incremented i
 - c.f. incrementing pointer
- ♦ Compiler can achieve same effect as manual use of pointers
 - Eliminating array address calculations within loop (induction variable elimination): 6 instructions reduced to 4 in loop
 - Better to make program clearer and safer

Outline

- ♦ Instruction set architecture (using MIPS ISA as an example)
- ♦ Operands
 - Register operands and their organization
 - Memory operands, data transfer
 - Immediate operands
- ♦ Signed and unsigned numbers
- ♦ Representing instructions
- ♦ Operations
 - Logical
 - Decision making and branches
- ♦ Supporting procedures in hardware
- ♦ Communicating with people
- ♦ Addressing for 32-bit immediate and addresses
- ♦ Translating and starting a program
- ♦ A sort example
- ♦ Arrays versus pointers
- ♦ ARM and x86 instruction sets (Sec. 2.15, 2.16)

ARM & MIPS Similarities

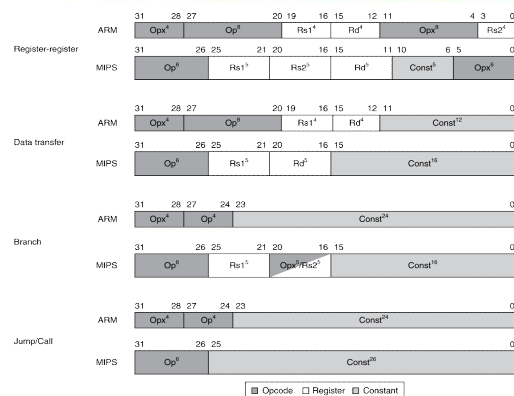
- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 x 32-bit	31 x 32-bit
Input/output	Memory mapped	Memory mapped

Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
 - Negative, zero, carry, overflow
 - Compare instructions to set condition codes without keeping the result
- Each instruction can be conditional
 - Top 4 bits of instruction word: condition value
 - Can avoid branches over single instructions

Instruction Encoding



The Intel x86 ISA

- Evolution with backward compatibility
 - 8080 (1974): 8-bit microprocessor
 - Accumulator, plus 3 index-register pairs
 - 8086 (1978): 16-bit extension to 8080
 - Complex instruction set (CISC)
 - 8087 (1980): floating-point coprocessor
 - Adds FP instructions and register stack
 - 80286 (1982): 24-bit addresses, MMU
 - Segmented memory mapping and protection
 - 80386 (1985): 32-bit extension (now IA-32)
 - Additional addressing modes and operations
 - Paged memory mapping as well as segments

The Intel x86 ISA

- ◆ Further evolution...
 - i486 (1989): pipelined, on-chip caches and FPU
 - Compatible competitors: AMD, Cyrix, ...
 - Pentium (1993): superscalar, 64-bit datapath
 - Later versions added MMX (Multi-Media eXtension) instructions
 - The infamous FDIV bug
 - Pentium Pro (1995), Pentium II (1997)
 - New microarchitecture (see Colwell, *The Pentium Chronicles*)
 - Pentium III (1999)
 - Added SSE (Streaming SIMD Extensions) and associated registers
 - Pentium 4 (2001)
 - New microarchitecture
 - Added SSE2 instructions

The Intel x86 ISA

- ◆ And further...
 - AMD64 (2003): extended architecture to 64 bits
 - EM64T – Extended Memory 64 Technology (2004)
 - AMD64 adopted by Intel (with refinements)
 - Added SSE3 instructions
 - Intel Core (2006)
 - Added SSE4 instructions, virtual machine support
 - AMD64 (announced 2007): SSE5 instructions
 - Intel declined to follow, instead...
 - Advanced Vector Extension (announced 2008)
 - Longer SSE registers, more instructions
- ◆ If Intel didn't extend with compatibility, its competitors would!
 - Technical elegance ≠ market success

Basic x86 Registers

Name	31	0	Use
EAX			GPR 0
ECX			GPR 1
EDX			GPR 2
EBX			GPR 3
ESP			GPR 4
EBP			GPR 5
ESI			GPR 6
EDI			GPR 7
		CS	Code segment pointer
		SS	Stack segment pointer (top of stack)
		DS	Data segment pointer 0
		ES	Data segment pointer 1
		FS	Data segment pointer 2
		GS	Data segment pointer 3
EIP			Instruction pointer (PC)
EFLAGS			Condition codes

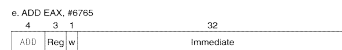
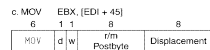
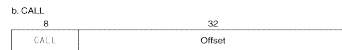
Basic x86 Addressing Modes

- ◆ Two operands per instruction

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- ◆ Memory addressing modes
 - Address in register
 - Address = $R_{\text{base}} + \text{displacement}$
 - Address = $R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}}$ (scale = 0, 1, 2, or 3)
 - Address = $R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}} + \text{displacement}$

x86 Instruction Encoding



Variable length encoding

- Postfix bytes specify addressing mode
- Prefix bytes modify operation
- Operand length, repetition, locking, ...

Implementing IA-32

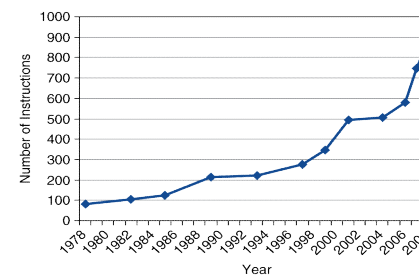
- Complex instruction set makes implementation difficult
 - Hardware translates instructions to simpler microoperations
 - Simple instructions: 1-1
 - Complex instructions: 1-many
 - Microengine similar to RISC
 - Market share makes this economically viable
- Comparable performance to RISC
 - Compilers avoid complex instructions

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%