CS4100: 計算機結構

Instruction Set Architecture

國立清華大學資訊工程學系
九十三學年度第一學期

Adapted from class notes of D. Patterson and W. Dally

Outline

- Instruction set architecture (using MIPS ISA as an example)
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Instruction format
- Operations
  - Arithmetic and logical
  - Decision making and branches
  - Jumps for procedures

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;

```
load    $r1, M[b]
load    $r2, 5
add     $r3, $r1, $r2
store   $r3, M[a]
```

Statement

Constant

Operands

Memory

Register

Operator (op code)

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 (next chapter)
- 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

$r0 - $r31
PC
HI
LO

3 Instruction Formats: all 32 bits wide

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

Outline

- Instruction set architecture (using MIPS ISA as an example)
- Operands
  - Register operands and their organization (Sec. 2.2)
  - Memory operands, data transfer
  - Immediate operands
- Instruction format
- Operations
  - Arithmetic and logical
  - Decision making and branches
  - Jumps for procedures

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? 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
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)

Fig. 2.18

MIPS R2000 Organization

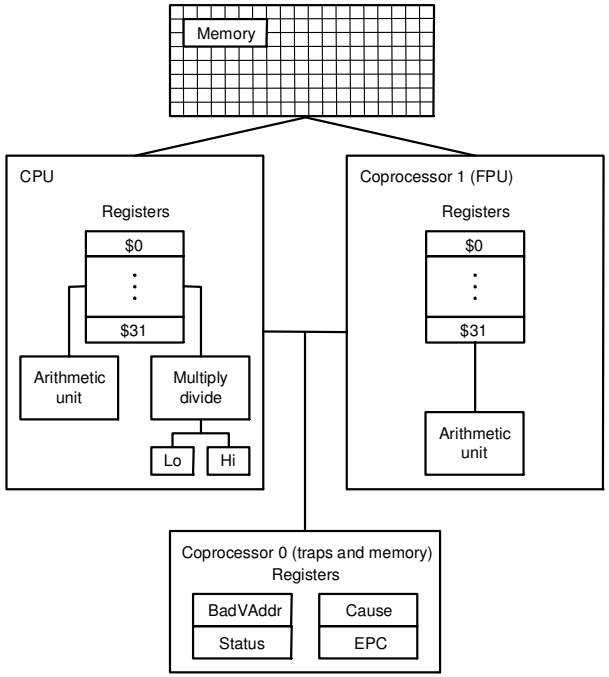


Fig. A.10.1

Operations of Hardware

- Syntax of basic MIPS arithmetic/logic instructions:

```
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

Example

- How to do the following C statement?

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

use intermediate temporary register `t0`

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


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 (using MIPS ISA as an example)
- Operands
  - Register operands and their organization
  - Memory operands, data transfer (Sec 2.3)
  - Immediate operands
- Instruction format
- Operations
  - Arithmetic and logical
  - Decision making and branches
  - Jumps for procedures

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:

```
lw $t0,12($s0)
```

  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

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 loaded 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

Compilation with Memory

- Compile by hand using registers:
  `g: $s1, h: $s2, $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]
```


Addressing: Byte versus Word

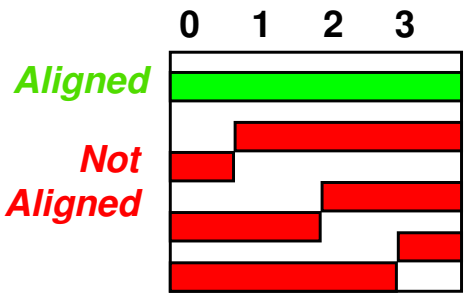
- Every word in memory has an address, similar to an index in an array
- Early computers numbered words like C numbers elements of an array:
  - Memory[0], Memory[1], Memory[2], …

Called the "**address**" of a word

- Computers needed 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

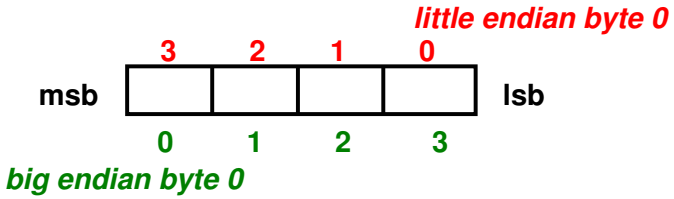
- MIPS requires that all words start at addresses that are multiples of 4 bytes



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

Another Note: Endianess

- Byte order: numbering of bytes within a word
- Big Endian: address of most significant byte = word address (xx00 = Big End of word)
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: address of least significant byte = word address (00xx = Little End of word)
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

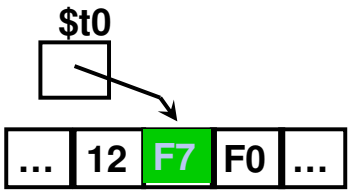


MIPS Data Transfer Instructions

Instruction		Comment
sw	$t3,500($t4)	Store word
sh	$t3,502($t2)	Store half
sb	$t2,41($t3)	Store byte
lw	$t1, 30($t2)	Load word
lh	$t1, 40($t3)	Load halfword
lhu	$t1, 40($t3)	Load halfword unsigned
lb	$t1, 40($t3)	Load byte
lbu	$t1, 40($t3)	Load byte unsigned
lui	$t1, 40	Load Upper Immediate (16 bits shifted left by 16)

What does it mean?

Load Byte Unsigned



lb $t1, 0($t0)

$t1: FFFFFF F7 — Sign-extended

lbu $t2, 0($t0)

$t2: 000000 F7 — Zero-extended

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 (using MIPS ISA as an example)
- Operands
  - Register operands and their organization
  - Memory operands, data transfer, and addressing
  - Immediate operands (Sec 2.3)
- Instruction format
- Operations
  - Arithmetic and logical
  - Decision making and branches
  - Jumps for procedures

Constants

- Small constants used frequently (50% of operands)

  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;
- Solutions? Why not?
  - put 'typical constants' in memory and load them
  - create hard-wired registers (like $zero) for constants
- MIPS Instructions:

  addi $29, $29, 4
  slti $8, $18, 10
  andi $29, $29, 6
  ori $29, $29, 4
- Design Principle: <u>Make the common case fast</u>
  Which format?

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
  - One particular immediate, the number zero (0), appears very often in code; so we define register zero (`$0` or `$zero`) to always 0
  - This is defined in hardware, so an instruction like `addi $0,$0,5` will not do anything

Outline

- Instruction set architecture (using MIPS ISA as an example)
- Operands
  - Register operands and their organization
  - Memory operands, data transfer
  - Immediate operands
- Instruction format (Sec. 2.4. 2.9)
- Operations
  - Arithmetic and logical
  - Decision making and branches
  - Jumps for procedures

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 (1/2)

- MIPS Instruction:
  `add  $8,$9,$10`
  - opcode = 0 (look up in table)
  - funct = 32 (look up in table)
  - rs = 9 (first operand)
  - rt = 10 (second operand)
  - rd = 8 (destination)
  - shamt = 0 (not a shift)

binary representation:

000000	01001	01010	01000	00000	100000

called a Machine Language Instruction

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, slitu, 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
- Key concept: Only one field is inconsistent with R-format. Most importantly, opcode is still in same location

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

I-Format Problem

What if immediate is too big to fit in immediate field?

- Load Upper Immediate:

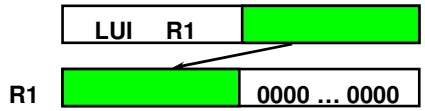
```
lui   register, immediate
```

  - puts 16-bit immediate in upper half (high order half) of the specified register, and sets lower half to 0s

```
addi   $t0,$t0, 0xABABCDCD
```

  becomes:

```
lui    $at, 0xABAB
ori    $at, $at, 0xCDCD
add    $t0,$t0,$at
```



Big Idea: Stored-Program Concept

- Computers built on 2 key principles:
  1) Instructions are represented as numbers
  2) Thus, entire programs can be stored in memory to be read or written just like numbers (data)
- One consequence: everything addressed
  - Everything has a memory address: instructions, data
    - both branches and jumps use these
  - One register keeps address of the instruction being executed: "Program Counter" (PC)
    - Basically a pointer to memory: Intel calls it Instruction Address Pointer, which is better
  - A register can hold any 32-bit value. That value can be a (signed) int, an unsigned int, a pointer (memory address), etc.

Outline

- Instruction set architecture (using MIPS ISA as an example)
- Operands
  - Register operands and their organization
  - Memory operands, data transfer, and addressing
  - Immediate operands
- Instruction format
- Operations
  - Arithmetic and logical (Sec 2.5)
  - Decision making and branches
  - Jumps for procedures

MIPS Arithmetic Instructions

Instruction	*Example*	*Meaning*	*Comments*
add	add $1,$2,$3	$1 = $2 + $3	3 operands;
subtract	sub $1,$2,$3	$1 = $2 - $3	3 operands;
add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant;

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:
  - Logical Operators
  - Shift Instructions

Logical Operators

- Logical instruction syntax:

```
or    $t0, $t1, $t2
```

  1) operation name
  2) register that will receive value
  3) first operand (register)
  4) second operand (register) or immediate (numerical constant)
- Instruction names:
  - `and`, `or`: expect the third argument to be a register
  - `andi`, `ori`: expect the third argument to be immediate
- MIPS Logical Operators are all bitwise, meaning that bit 0 of the output is produced by the respective bit 0's of the inputs, bit 1 by the bit 1's, etc.

Use for Logical Operator And

- `and` operator can be used to set certain portions of a bit-string to 0s, while leaving the rest alone => mask
- Example:

Mask: 1011 0110 1010 0100 0011 1101 1001 1010
0000 0000 0000 0000 0000 1111 1111 1111

- The result of anding these two is:
  0000 0000 0000 0000 0000 1101 1001 1010
- In MIPS assembly: `andi $t0,$t0,0xFFF`

Uses for Logical Operator Or

- `or` operator can be used to force certain bits of a string to 1s
- For example,
  `$t0 = 0x12345678`, then after

```
ori  $t0, $t0, 0xFFFF
```

  `$t0 = 0x1234FFFF`
  (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are set to 1s)

Shift Instructions (1/3)

- Shift Instruction Syntax:

```
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 (1/2)

- Suppose we want to get byte 1 (bit 15 to bit 8) of a word in `$t0`. We can use:

```
sll    $t0,$t0,16
srl    $t0,$t0,24
```

0001 0010 0011 0100 0101 0110 0111 1000

0101 0110 0111 1000 0000 0000 0000 0000

0000 0000 0000 0000 0000 0000 0101 0110

Uses for Shift Instructions (2/2)

- 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 2n 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)
```


MIPS Logical Instructions

Instruction	*Example*	*Meaning*	*Comment*
and	and $1,$2,$3	$1 = $2 & $3	3 reg. operands; Logical AND
or	or $1,$2,$3	$1 = $2 | $3	3 reg. operands; Logical OR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	3 reg. operands; Logical NOR
and immediate	andi $1,$2,10	$1 = $2 & 10	Logical AND reg, zero exten.
or immediate	ori $1,$2,10	$1 = $2 | 10	Logical OR reg, zero exten.
shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
shift right arithm.	sra $1,$2,10	$1 = $2 >> 10	Shift right (sign extend)

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 (using MIPS ISA as an example)
- Operands
  - Register operands and their organization
  - Memory operands, data transfer, and addressing
  - Immediate operands
- Instruction format
- Operations
  - Arithmetic and logical
  - Decision making and branches (Sec. 2.6, 2.9)
  - Jumps for procedures

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

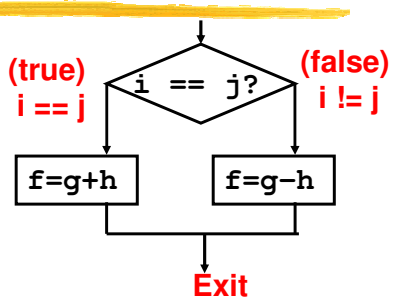
Compiling C `if` into MIPS

- Compile by hand

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

- Use this mapping:

f: $s0, g: $s1, h: $s2,
i: $s3, j: $s4



- Final compiled MIPS code:

```
        beq   $s3,$s4,True      # branch i==j
        sub   $s0,$s1,$s2 # f=g-h(false)
        j     Fin               # go to Fin
True:   add   $s0,$s1,$s2 # f=g+h (true)
Fin:
```

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

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   reg1,reg2,reg3
```

meaning :

```
if (reg2 < reg3)
reg1 = 1;          # set
else reg1 = 0;     # reset
```

- 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

Immediate in Inequalities

- There is also an immediate version of `slt` to test against constants: `slti`

C

```
if (g >= 1) goto Loop
Loop:    . . .
```

MIPS

```
slti $t0,$s0,1     # $t0 = 1 if $s0<1 (g<1)
beq  $t0,$0,Loop   # goto Loop if $t0==0
```

- Unsigned inequality: `sltu`, `sltiu`

  $s0 = FFFF FFFA$_{hex}$, $s1 = 0000 FFFA$_{hex}$

```
slt  $t0, $s0, $s1      => $t0 = ?
sltu $t1, $s0, $s1      => $t1 = ?
```


Branches: Instruction Format

- 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
    - Now we can branch +/- 2^{15} bytes from the PC ?

Branches: Instruction Format

- *`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 +/- 2^{15} words from the PC (or +/- 2^{17} 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` = 3

Branch Example

- MIPS Code:

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

decimal representation:

4	9	0	3

binary representation:

000100	01001	00000	0000000000000011

J-Format Instructions (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.

J-Format Instructions (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

J-Format Instructions (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: II 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

MIPS Jump, Branch, Compare

Instruction	*Example*	*Meaning*
branch on equal	beq $1,$2,25	if ($1 == $2) go to PC+4+100 *Equal test; PC relative branch*
branch on not eq.	bne $1,$2,25	if ($1!= $2) go to PC+4+100 *Not equal test; PC relative*
set on less than	slt $1,$2,$3	if ($2 < $3) $1=1; else $1=0 *Compare less than; 2's comp.*
set less than imm.	slti $1,$2,100	if ($2 < 100) $1=1; else $1=0 *Compare < constant; 2's comp..*
jump	j 10000	go to 10000 *26-bit+4-bit of PC*

Outline

- Instruction set architecture (using MIPS ISA as an example)
- Operands
  - Register operands and their organization
  - Immediate operands
  - Memory operands, data transfer, and addressing
- Instruction format
- Operations
  - Arithmetic and logical
  - Decision making and branches
  - Jumps for procedures (Sec. 2.7)

C Function Call Bookkeeping

```
... sum(a,b);... /* a:$s0; b:$s1 */
}

int sum(int x, int y) {
   return x+y;
}
```

- 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

Instruction Support for Functions

C

```
... sum(a,b);... /* a:$s0; b:$s1 */
}

int sum(int x, int y) { /* x:$a0; y:$a1 */
   return x+y;
}
```

MIPS

```
address
1000 add  $a0,$s0,$zero      # x <= a
1004 add  $a1,$s1,$zero      # y <= b
1008 addi $ra,$zero,1016     # $ra <= 1016
1012 j    sum                # jump to sum
1016 ...
2000 sum: add $v0,$a0,$a1
2004 jr   $ra                # new instruction
```

JAL and JR

- Single instruction to jump and save return address: jump and link (`jal`)
  - Replace:
    ```
    1008 addi $ra,$zero,1016      #$ra=1016
    1012 j sum                    #go to sum
    ```
    with:
    ```
    1012 jal sum            # $ra=1016,go to sum
    ```
  - Step 1 (link): Save address of *next* instruction into $ra
  - Step 2 (jump): Jump to the given label
  - Why have a `jal`? Make the common case fast: functions are very common
- jump register: `jr register`
  - `jr` provides a register that contains an address to jump to; usually used for procedure return

MIPS Jump, Branch, Compare

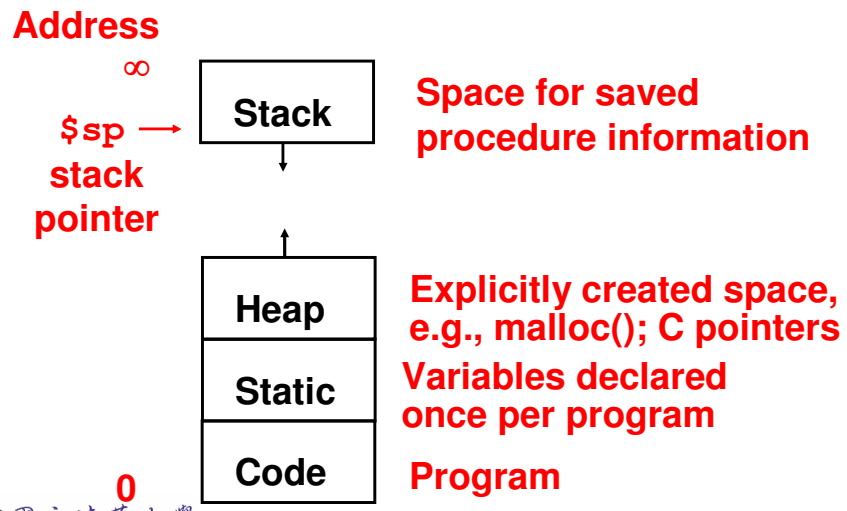
Instruction	*Example*	*Meaning*
branch on equal	beq $1,$2,25	if ($1 == $2) go to PC+4+100 *Equal test; PC relative branch*
branch on not eq.	bne $1,$2,25	if ($1!= $2) go to PC+4+100 *Not equal test; PC relative*
set on less than	slt $1,$2,$3	if ($2 < $3) $1=1; else $1=0 *Compare less than; 2's comp.*
set less than imm.	slti $1,$2,100	if ($2 < 100) $1=1; else $1=0 *Compare < constant; 2's comp..*
jump	j 10000	go to 10000 *26-bit+4-bit of PC*
jump register	jr $31	go to $31 *For switch, procedure return*
jump and link	jal 10000	$31 = PC + 4; go to 10000 *For procedure call*

Nested Procedures

```
int sumSquare(int x, int y) {
     return mult(x,x)+ y;
  }
```

- Need to save `sumSquare` return address (saved in `$ra`) before call to `mult`
  - In general, may need to save some other info in addition to `$ra`
- Recall 3 memory areas in a C program:
  - *Static*: variables declared once per program, cease to exist only after execution completes
  - *Heap*: variables declared dynamically
  - *Stack*: space used by procedure at execution
- Use the stack to save register values

C Memory Allocation



Compiling Nested C Function

C
```
int sumSquare(int x, int y) {
    return mult(x,x)+ y;}
```

```
sumSquare:
        subi $sp,$sp,12        # space on stack
        sw   $ra,8($sp)        # save ret addr
Prologue sw  $a0,0($sp)        # save x
        sw   $a1,4($sp)        # save y
        add  $a1,$a0,$zero     # mult(x,x)
Body    jal  mult              # call mult
        lw   $ra,8($sp)        # get ret addr
        lw   $a0,0($sp)        # restore x
        lw   $a1,4($sp)        # restore y
Epilogue addi $sp,$sp,12       # => stack space
        add  $v0,$v0,$a1       # mult()+y
        jr   $ra
```


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

- Callees' rights:
  - Right to use VAT registers freely
  - Right to assume arguments are passed correctly
- To ensure callees'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 S 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)

- 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`

Contract in Function Calls (2/2)

- 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`

An Example of Passing Arguments

```
int Doh(int i, int j, int k, int m,
  char c, int n)
{
  return i+j+n;
}
```

6th argument

```
Doh:    lw   $t0  20($sp)
        add  $a0, $a0, $a1
        add  $v0, $a0, $t0
        jr   $ra
```


Registers Conventions for MIPS

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)

Fig. 2.18

Addressing Modes

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

MIPS only supports these => simple is fast

MIPS Addressing Mode

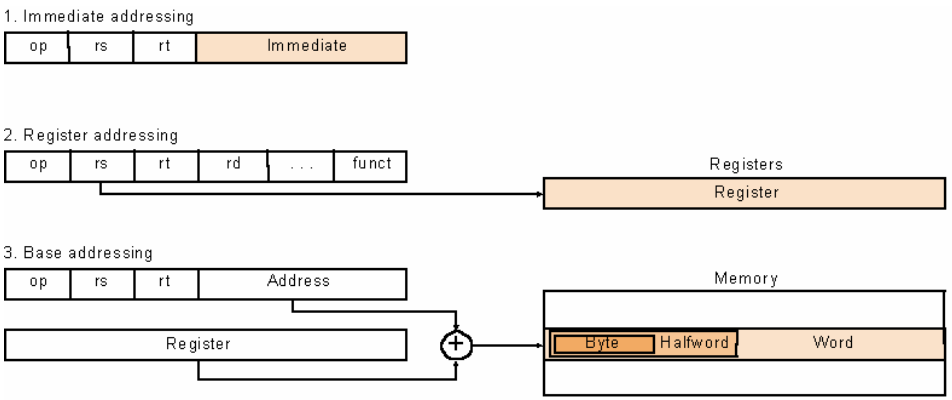


Fig. 2.24

MPIS Addressing Modes

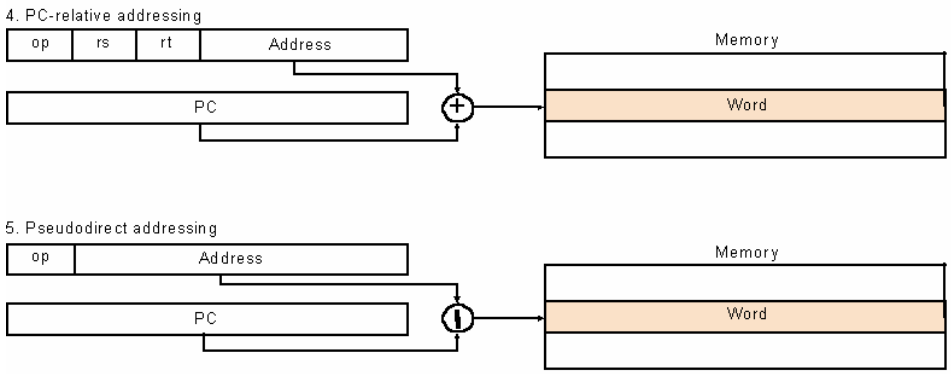


Fig. 2.24

To Summarize

MIPS operands

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

MIPS assembly language

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

Summary: MIPS ISA (1/2)

- 32-bit fixed format instructions (3 formats)
- 32 32-bit GPR (R0 = zero), 32 FP registers, (and HI LO)
  - partitioned by software convention
- 3-address, reg-reg arithmetic instructions
- Memory is byte-addressable with a single addressing mode: base+displacement
  - 16-bit immediate plus LUI
- Decision making with conditional branches: beq, bne
  - Often compare against zero or two registers for =
  - To help decisions with inequalities, use: "Set on Less Than"called slt, slti, sltu, sltui
- Jump and link puts return address PC+4 into link register (R31)
- Branches and Jumps were optimized to address to words, for greater branch distance

Summary: MIPS ISA (2/2)

- Immediates are extended as follows:
  - logical immediate: zero-extended to 32 bits
  - arithmetic immediate: sign-extended to 32 bits
  - Data loaded by lb and lh are similarly extended: lbu, lhu are zero extended; lb, lh are sign extended
- Simplifying MIPS: Define instructions to be same size as data (one word), so they can use same memory
- Stored Program Concept: Both data and actual code (instructions) are stored in the same memory
- Instructions formats are kept as similar as possible

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		
J	opcode	target address				

Alternative Architectures

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

  –*"The path toward operation complexity is thus fraught with peril. To avoid these problems, designers have moved toward simpler instructions"*

- Let's look (briefly) at IA-32

IA-32

- 1978: Intel 8086 is announced (16 bit architecture)
- 1980: 8087 floating point coprocessor is added
- 1982: 80286 increases address space to 24 bits, +instructions
- 1985: 80386 extends to 32 bits, new addressing modes
- 1989-1995: 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: 57 new "MMX" instructions are added, Pentium II
- 1999: Pentium III added another 70 instructions (SSE)
- 2001: Another 144 instructions (SSE2)
- 2003: AMD extends to increase address space to 64 bits, widens all registers to 64 bits and other changes (AMD64)
- 2004: Intel capitulates and embraces AMD64 (calls it EM64T) and adds more media extensions

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

IA-32 Overview

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

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

IA-32 Registers & Data Addressing

- Registers in 32-bit subset that originated with 80386

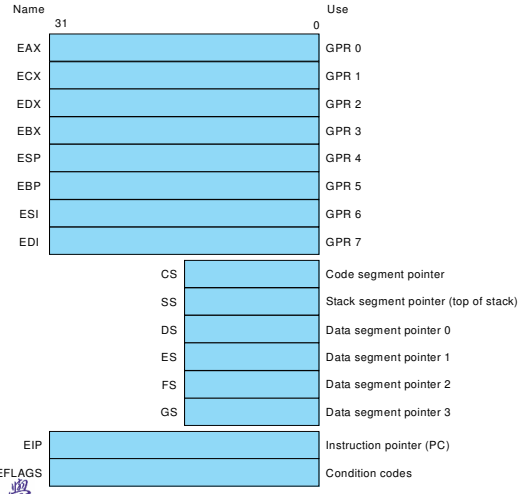


Fig. 2.40

IA-32 Register Restrictions

- Registers are not "general purpose" – note the restrictions below

Mode	Description	Register restrictions	MIPS equivalent
Register indirect	Address is in a register.	not ESP or EBP	`lw $s0,0($s1)`
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	not ESP or EBP	`lw $s0,100($s1)# ≤16-bit # displacement`

Base plus scaled index	The address is Base + (2^{Scale} x Index) where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,0($t0)`

Base plus scaled index with 8- or 32-bit displacement	The address is Base + (2^{Scale} x Index) + displacement where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,100($t0)# ≤16-bit # displacement`

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

Fig. 2.42

IA-32 Typical Instructions

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

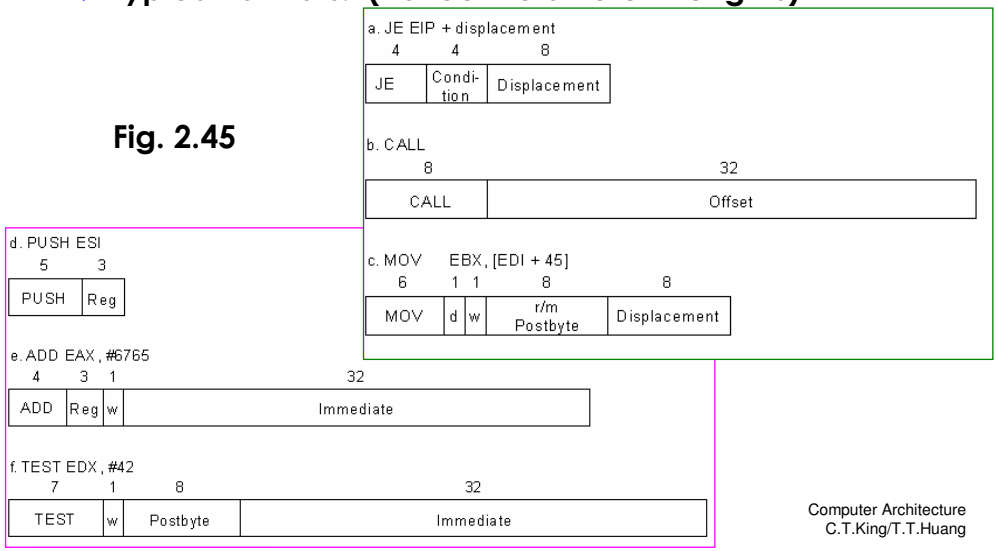
Instruction	Function
JE name	if equal(condition code) {EIP=name}; EIP-128 ≤ name < EIP+128
JMP name	EIP=name
CALL name	SP=SP-4; M[SP]=EIP+5; EIP=name;
MOVW EBX,[EDI+45]	EBX=M[EDI+45]
PUSH ESI	SP=SP-4; M[SP]=ESI
POP EDI	EDI=M[SP]; SP=SP+4
ADD EAX,#6765	EAX= EAX+6765
TEST EDX,#42	Set condition code (flags)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

Fig. 2.43

IA-32 instruction Formats

- Typical formats: (notice the different lengths)

Fig. 2.45



Summary

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