Instruction Set Architecture for MIPS Processors

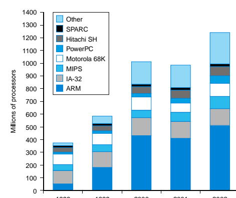
Dr. Arjan Durresi
Louisiana State University
Baton Rouge, LA 70803
durresi@csc.lsu.edu

These slides are available at:
http://www.csc.lsu.edu/~durresi/CSC3501_07/

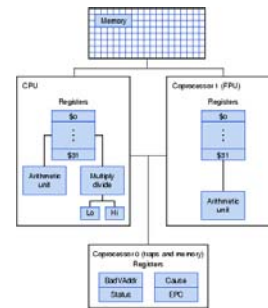
- Operations and Operands of the Computer Hardware
- Representing Instructions in Computer
- MIPS addressing
- An Introduction to Compilers
- IA-32 Instructions

Instructions:

- Language of the Machine
- We'll be working with the MIPS instruction set architecture
  - similar to other architectures developed since the 1980's
  - Almost 100 million MIPS processors manufactured in 2002
  - used by NEC, Nintendo, Cisco, Silicon Graphics, Sony, …



MIPS Processor



MIPS arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code: `a = b + c`

MIPS 'code': `add a, b, c`

(we'll talk about registers in a bit)

"The natural number of operands for an operation like addition is three…requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple"

MIPS arithmetic

- Design Principle: Simplicity favors regularity.
- Of course this complicates some things...

C code: `a = b + c + d;`

MIPS code:
```
add a, b, c
add a, a, d
```

- Operands must be registers, only 32 registers provided
- Each register contains 32 bits

- Design Principle: smaller is faster. Why?

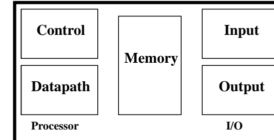
Compiling C Assignments into MIPS

C code: f = (g+h) - (i+j);

MIPS code:
```
add t0, g, h
add t1, i, j
sub f, t0, t1
```


Registers vs. Memory

- Arithmetic instructions operands must be registers,
  — only 32 registers provided
  - Registers are primitives used in hardware design that are visible to the programmer
- Compiler associates variables with registers
- What about programs with lots of variables



MIPS Registers

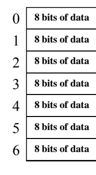
- CPU:
  - 32 32-bit general purpose registers - GPRs (r0 – r31);
  - r0 has fixed value of zero. Attempt to writing into r0 is not illegal, but its value will not change;
  - two 32-bit registers – Hi & Lo, hold results of integer multiply and divide
  - 32-bit program counter – PC;
- Floating Point Processor – FPU (Coprocessor 1 – CP1):
  - 32 32-bit floating point registers –FPRs (f0 –f31)
  - Five control registers

MIPS Data Types

- MIPS operates on:
  - - 32-bit (unsigned or 2's complement) integers,
  - - 32-bit (single precision floating point) real numbers,
  - - 64-bit (double precision floating point) real numbers;
- bytes and half words loaded into GPRs are either zero or sign bit expanded to fill the 32 bits;
- only 32-bit units can be loaded into FPRs; 32-bit real numbers are stored in even numbered FPRs.
- 64-bit real numbers are stored in two consecutive FPRs, starting with even-numbered register.

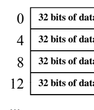
Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.



Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.



Registers hold 32 bits of data

- 2^{32} bytes with byte addresses from 0 to 2^{32}-1
- 2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4
- Words are aligned

MIPS Addressing Modes

- register addressing;
- immediate addressing;
- only one memory data addressing:
  - – register content plus offset (register indexed);
- since r0 always contains value 0:
  - – r0 + offset → absolute addressing;
- offset = 0 → register indirect;
- MIPS supports byte addressability:
  - – it means that a byte is the smallest unit with its address;
- MIPS supports 32-bit addresses:
  - – it means that an address is given as 32-bit unsigned integer;

MIPS Alignment

- MIPS restricts memory accesses to be aligned as follows:
  - 32-bit word has to start at byte address that is multiple of 4;
  - 32-bit word at address 4n includes four bytes with addresses 4n, 4n+1, 4n+2, and 4n+3.
  - 16-bit half word has to start at byte address that is multiple of 2;
  - 16-bit word at address 2n includes two bytes with addresses 2n and 2n+1.

MIPS Instructions

- 32-bit fixed format instruction and 3 formats;
- Register – register and register-immediate computational instructions;
- Single address mode for load/store instructions:
  - register content + offset (called base addressing);
- Simple branch conditions;
  - branch instructions use PC relative addressing;
  - branch address = [PC] + 4 + 4×offset
- Jump instructions with:
  - 28-bit addresses (jumps inside 256 megabyte regions),

  or
  - absolute 32-bit addresses.

Instructions

- Load and store instructions
- Example:

C code: `A[12] = h + A[8];`

MIPS code:
```
lw $t0, 32($s3)
add $t0, $s2, $t0
sw $t0, 48($s3)
```

- Can refer to registers by name (e.g., $s2, $t2) instead of number
- Store word has destination last
- Remember arithmetic operands are registers, not memory!

Can't write: `add 48($s3), $s2, 32($s3)`

Our First Example

- Can we figure out the code?

```
swap(int v[], int k);
{ int temp;
    temp = v[k]
    v[k] = v[k+1];
    v[k+1] = temp;
}
```
➡
```
swap:
    muli $2, $5, 4
    add $2, $4, $2
    lw $15, 0($2)
    lw $16, 4($2)
    sw $16, 0($2)
    sw $15, 4($2)
    jr $31
```


So far we've learned:

- MIPS
  - — loading words but addressing bytes
  - — arithmetic on registers only

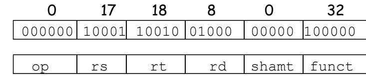
Instruction	Meaning
`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`
`sub $s1, $s2, $s3`	`$s1 = $s2 – $s3`
`lw $s1, 100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1, 100($s2)`	`Memory[$s2+100] = $s1`

MIPS Instructions

- *Instructions that move data:*
  - *load to register from memory,*
  - *store from register to memory,*
  - *move between registers in same and different coprocessors*
- *ALU integer instructions,*
- *Floating point instructions,*
- *Control-related instructions,*
- *Special control-related instructions.*

Machine Language

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

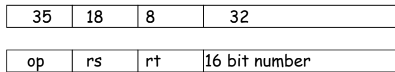


0	17	18	8	0	32
000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

- *Can you guess what the field names stand for?*
- All MIPS instructions are 32 bits long

Machine Language

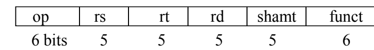
- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: Good design demands a compromise
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register
- Example: `lw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



35	18	8	32
op	rs	rt	16 bit number

Design Principle: Good design demands good compromises.

MIPS Fields

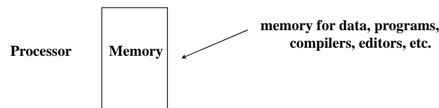


op	rs	rt	rd	shamt	funct
6 bits	5	5	5	5	6

- op: - Basic operation of the instruction, traditionally called opcode
- rs: - The first register source operand
- rt: - The second register source operand
- rd: - The Register destination operand
- shamt: Shift amount
- funct: Function

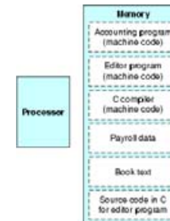
Stored Program Concept

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



- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions
  - Fetch the "next" instruction and continue

Stored Program Concept



Example

```
A[300] = h + A[300]:

lw      $t0, 1200($t1)    #Temporary reg $t0 gets A[300]

add     $t0,$s2,$t0       # Temporary reg $t0 gets h + A[300]

sw      $t0,1200($t1)     #Stores h + A[300]
```

- $t1 has the base array for A and $s2 corresponds to h

op	rs	rt	rd	Address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

100011	01001	0100	0000 0100 1011 0000		
000000	10010	0100	0100	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Hexadecimal

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

Logical Operations

Logical operation	MIPS
Shift left	sll
Shift right	srl
Bit-by-bit AND	and, andi
Bit-by-bit OR	or, ori
Bit-by-bit NOT	nor

- It is useful to be able to operate on bits

Shift

0000 0000 0000 0000 0000 0000 0000 1001 = 9_{ten}

0000 0000 0000 0000 0000 0000 1001 0000 = 144_{ten}

```
Sll     $t2,$s0,4    # reg $t2 = reg $s0 << 4 bits
```

op	rs	rt	rd	shamt	funct
0	0	16	10	4	4

- Shifting left by *i* bits gives the same result as multiplying by *2i*

AND, OR , NOR

```
0000 0000 0000 0000 0000 0000 1101 0000 0000

0000 0000 0000 0000 0000 0011 1100 0000 0000

add     $t0,$t1,$t2    # reg $t0 = reg $t1 & reg $t2
0000 0000 0000 0000 0000 0000 1100 0000 0000


or      $t0,$t1,$t2    # reg $t0 = reg $t1 | reg $t2
0000 0000 0000 0000 0000 0011 1101 0000 0000

nor     $t0,$t1,$t2    # reg $t0 = ~(reg $t1 | reg $t2)
1111 1111 1111 1111 1111 1100 0010 1111 1111
```

- AND is called a mask
- NOR - NOT OR

Control

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

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

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

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

Control

- MIPS unconditional branch instructions:

```
j  label
```

- Example:

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

- *Can you build a simple for loop?*

Loop

- Loop

```
    while (save[i]==k)
          i + = 1;
i -> $s3,  k -> $s5, base of save in $s6

Loop: sll   $t1,$s3,2       #Temp reg $t1 = 4 * i
      add   $t1,$t1,$s6     # $t1 = address of save[i]
      lw    $t0,0($t1)      # Temp reg $t0 = save[i]
      bne   $t0,$s5, Exit   # Got to Exit if save[i]≠k
     add    $s3,$s3,1       # i = i +1
        j   Loop
     Exit
```


So far:

Instruction	Meaning
`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`
`sub $s1,$s2,$s3`	`$s1 = $s2 – $s3`
`lw $s1,100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1,100($s2)`	`Memory[$s2+100] = $s1`
`bne $s4,$s5,L`	Next instr. is at Label if $s4 ≠ $s5
`beq $s4,$s5,L`	Next instr. is at Label if $s4 = $s5
`j Label`	Next instr. is at Label

- Formats:

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

Control Flow

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

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

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

Policy of Use Conventions

Name	Register number	Usage
$zero	0	the constant value 0
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

Register 1 ($at) reserved for assembler, 26-27 for operating system

Constants

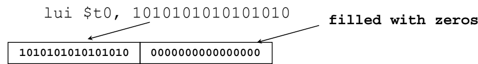
- Small constants are used quite frequently (50% of operands)
  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;
- Solutions? Why not?
  - put 'typical constants' in memory and load them.
  - create hard-wired registers (like $zero) for constants like one.
- MIPS Instructions:

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

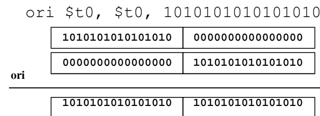
- Design Principle: Make the common case fast. *Which format?*

How about larger constants?

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



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



Assembly Language vs. Machine Language

- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudoinstructions'
  - e.g., "move $t0, $t1" exists only in Assembly
  - would be implemented using "add $t0,$t1,$zero"
- When considering performance you should count real instructions

Other Issues

- Support for procedures linkers, loaders, memory layout stacks, frames, recursion manipulating strings and pointers interrupts and exceptions system calls and conventions
- Some of these we'll talk more about later
- We'll talk about compiler optimizations when we hit chapter 4.

Supporting Procedures

- Procedure – a stored subroutine that performs a specific task based on the parameters which it is provided
- A program should:
  - Place parameters in a place where the procedure can access them
  - Transfer control to procedures
  - Acquired the storage resources needed for the procedure
  - Perform the desired task
  - Place the result value in a place where the calling program can access it
  - Return control to the point of origin, since a procedure can be called from several points in a program

Supporting Procedures

- $a0 - $a3 : four argument registers in which to pass parameters
- $v0 - $v1 : two value registers in which to return values
- $ra: one return address register to return to the point of origin
- jal - jump-and-link instruction: It jumps to an address and simultaneously saves the address of the following instruction to $ra

jal ProcedureAddress

- jal saves PC+4 to $ra
- jr – jump register: unconditional jump to the address specified in a register
- jr $ra
- The caller places the parameter values in $a0-$a3 and uses jal ProcAddress; the callee places the results in $v0-$v1 and return control to the caller using jr $ra

Supporting Procedures

- If more registers are needed:
  - Spill registers to memory
  - Why not use other registers?
- The best data structure for spilling registers in a stack – a last-in-first-out queue
  - Needs a pointer to the most recently allocated address
  - Push : Placing data onto the stack
  - Pop : Removing data from the stack
  - $sp – the stack pointer

Example

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

The parameter variables g,h,i,j corresponds to the argument registers $a0,$a1,$a2,$a3 and f corresponds to $s0

We need to save three registers $s0, $t0, $t1. We push the old values onto the stack by creating space for three words:

```
addi $sp,$sp,-12 #adjust stack to make room for 3 items
sw   $t1,8($sp)  # save register $t1 for use afterwards
sw   $t0,4($sp)  # save register $t0 for use afterwards
sw   $s0,0($sp)  # save register $s0 for use afterwards
```


Stack

High address

Low address a. b. c.

The value of the stack pointer and the stack
a) before, b) during and c) after the procedure call

Example

```
add $t0, $a0, $a1  # register $t0 contains g + h
add $t1, $a2, $a3  # register $t1 contains i + j
sub $s0, $t0, $t1  # f = $t0 - $t1
```

To return the value of f, we copy it into a return value register:

```
add $v0, $s0, $zero # returns f ($v0 = $s0 +0)
```

Before returning, we restore the three old values of the register we saved by "popping" them from the stack:

```
lw  $s0, 0($sp)  # restore register $s0 for caller
lw  $t0, 4($sp)  # restore register $t0 for caller
lw  $t1, 8($sp)  # restore register $t1 for caller
addi $sp, $sp, 12 # adjust stack to delete 3 items
The procedure ends with a jump register using the return
   address
jr $ra # jump back to calling routine
```


Supporting Procedures

- To reduce saving and restoring registers, MIPS:
  - $t0 - $t9: 10 temporary registers that are not preserved by the callee
  - $s0-S7 : 8 saved registers that must be preserved on a procedure call
- What about nested procedures?

Characters

```
lb $t0,0($sp) # Read byte from source
sb $t0,0($sp) # Write byte to destination

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


Overview of MIPS

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

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

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

Addresses in Branches and Jumps

- Instructions:

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

- Formats:

I	op	rs	rt	16 bit address
J	op	26 bit address		

- Addresses are not 32 bits
  — How do we handle this with load and store instructions?

Addresses in Branches

- Instructions:

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

- Formats:

op	rs	rt	16 bit address

- Could specify a register (like lw and sw) and add it to address
  - use Instruction Address Register (PC = program counter)
  - most branches are local (principle of locality)
- Jump instructions just use high order bits of PC
  - address boundaries of 256 MB

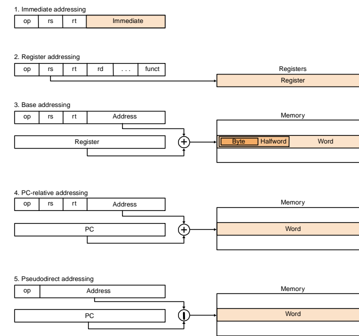
To summarize:

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



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: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: 57 new "MMX" instructions are added, Pentium II
- 1999: The Pentium III added another 70 instructions (SSE)
- 2001: Another 144 instructions (SSE2)
- 2003: AMD extends the architecture 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"

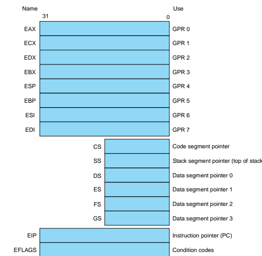
IA-32 Overview

- 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 and Data Addressing

- Registers in the 32-bit subset that originated with 80386



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. [illegible]

IA-32 Typical Instructions

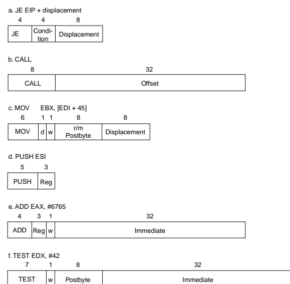
- Four major types of integer instructions:
  - Data movement including move, push, pop
  - Arithmetic and logical (destination register or memory)
  - Control flow (use of condition codes / flags)
  - String instructions, including string move and string compare

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

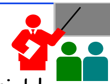
FIGURE 2.43 Some typical IA-32 instructions and their functions. A list of frequent operations appears in Figure 2.44. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)

IA-32 instruction Formats

- Typical formats: (notice the different lengths)



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!