Chapter 2

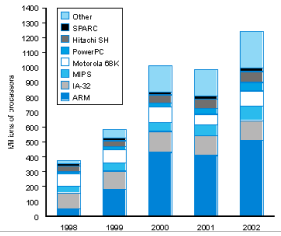
Conjunto de Instruções → Assembly

MIPS

x86

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 arithmetic

add | a | b | c | 32 bits

add ax, bx [100]

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

Example:

C code: `a = b + c;` (dest, src)

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"

RISC

MIPS arithmetic

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

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

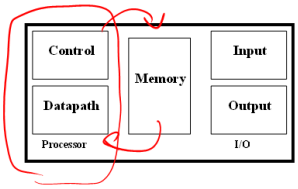
MIPS code:
```
add a, b, c
add a, a, d
```

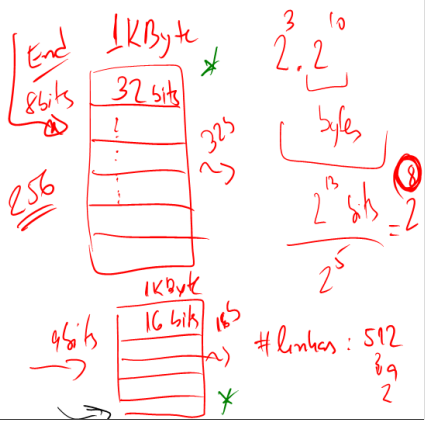
Banco 32 0..31

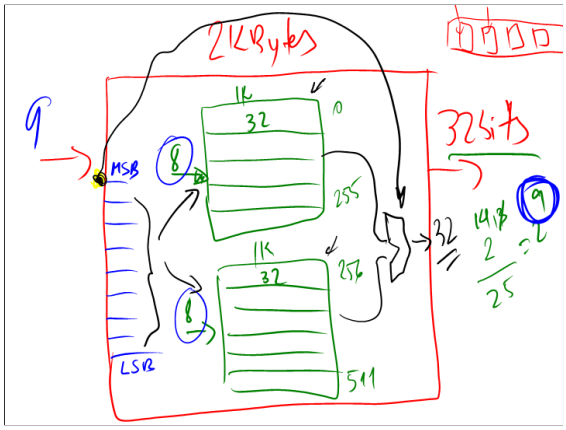
- Operands must be registers, only 32 registers provided
- Each register contains 32 bits
- Design Principle: smaller is faster. Why?

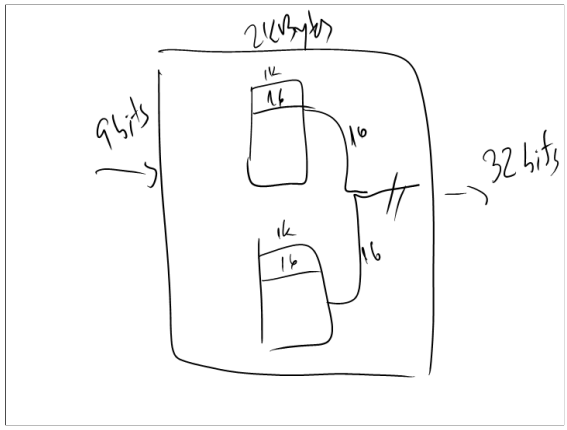
Registers vs. Memory

- Arithmetic instructions operands must be registers,
  — only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables









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.

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data
...	

 6

Memory Organization

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

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data
...	

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
  i.e., what are the least 2 significant bits of a word address?

 7

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!

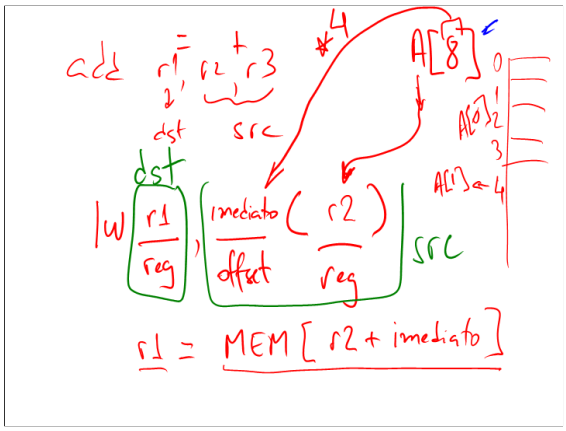
Can't write: `add 48($s3), $s2, 32($s3)`

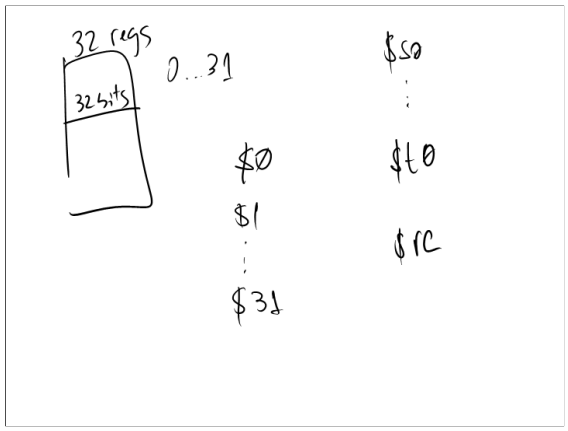
 8

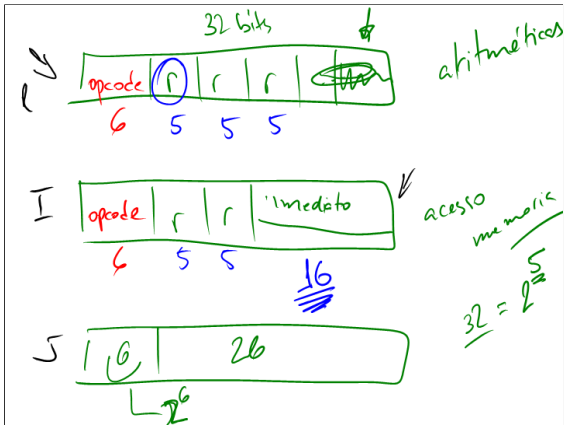
sw $t0, 48($s3)

MEM[$s3+48] = $t0

A[12]







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
```


 9

So far we've learned:

- MIPS
  - loading words but addressing bytes
  - arithmetic on registers only
- Instruction / Meaning

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`

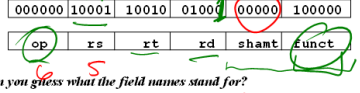
addi $2, $5, 1

-32

 10

Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - Example: `add $t1, $s1, $s2`
  - registers have numbers, `$t1=9, $s1=17, $s2=18`
- Instruction Format:



- *Can you guess what the field names stand for?*

add rd, rs, rt

32 bits

 11

Machine Language

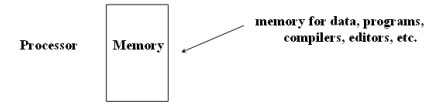
- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: Good design demands a compromise
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register
- Example: `lw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

35	18	9	32
op	rs	rt	16 bit number

- Where's the compromise?

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

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)
    h=i+j;
else
    h=i-j;
```

```
beq $s4, $s5, Lab1
add $s3, $s4, $s5
j Lab2
Lab1: sub $s3, $s4, $s5
Lab2: ...
```

- *Can you build a simple for loop?*

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:

```
slt $t0, $s1, $s2
```

```
if $s1 < $s2 then
    $t0 = 1
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

```
lui $t0, 1010101010101010
```

filled with zeros

1010101010101010	0000000000000000

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

```
ori $t0, $t0, 1010101010101010
```

	1010101010101010	0000000000000000
	0000000000000000	1010101010101010
ori		
	1010101010101010	1010101010101010

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

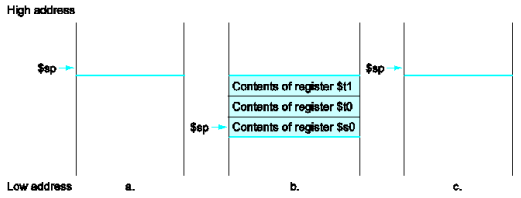
- Discussed in your assembly language programming lab:
  - support for procedures
  - linkers, loaders, memory layout
  - stacks, frames, recursion
  - manipulating strings and pointers
  - interrupts and exceptions
  - system calls and conventions
- Some of these we'll talk more about later
- We'll talk about compiler optimizations when we hit chapter 4.

Procedimentos

- Passos a serem executados pelo programa:
  1. Colocar os parâmetros em um lugar onde o procedimento consegue recebe-los
  2. Transferir o controle para o procedimento
  3. Alocar recursos para o procedimento
  4. Executar o procedimento
  5. Colocar os resultados em um lugar onde o programa possa acessar
  6. Retornar ao ponto seguinte da chamada do procedimento
- Novidades
  - Instrução jal chama o procedimento
  - Instrução jr retorna do procedimento
  - Registrador $ra contém o valor de retorno

Pilha

- Utiliza o registrador $sp e $fp (em alguns casos)
- Cresce do endereço alto para o endereço baixo
- Utilizada para guardar valores, variáveis locais e passagem de parâmetros extras



Exemplo

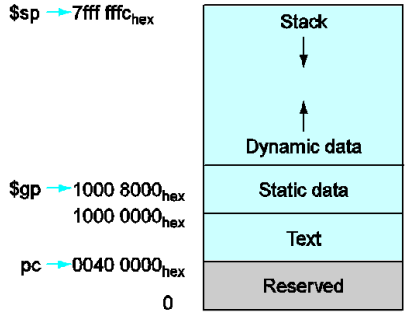
```
int fact(int n)
{
  if (n < 1)
     return 1;
  else
     return (n * fact(n - 1));
}
```


Exercício 2.29

- Coloque comentários no código MIPS abaixo e descreva em uma frase o que ele computa. $a0 e $a1 são usados como entrada e contém, respectivamente, os inteiros a e b. $v0 é usado como saída.

```
        add    $t0, $zero, $zero
loop:   beq    $a1, $zero, finish
        add    $t0, $t0, $a0
        sub    $a1, $a1, 1
        j      loop
finish: addi   $t0, $t0, 100
        add    $v0, $t0, $zero
```


Organização da Memória



Caracteres

- Representação ASCII x Unicode
- Representações para strings
  1. Primeiro caracter indica o tamanho da string
  2. Uma variável acompanha a string indicando seu tamanho (como numa estrutura)
  3. A string termina com um caracter reservado
- Instruções
  - Byte: lb e sb (8 bits)
  - Halfword: lh e sh (16 bits)

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:**

I	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**

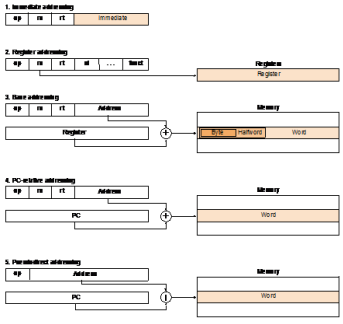
To summarize:

MIPS operands

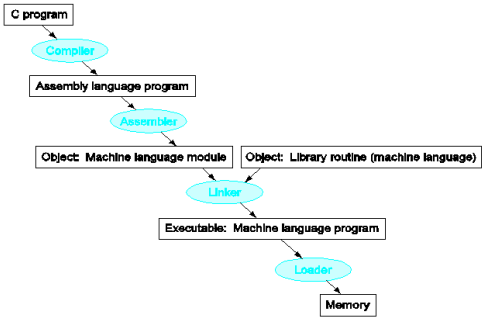
Name	Example	Comments
32 registers	$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory 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
Uncondi-tional 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



Passos para criar um programa



Exemplos de código

```
if (x == 10)
  a = 5
else
  b = 7
```

```
while (x > 7)
  x --;
```

```
switch (x) {
  case 0: a = 5;
          break;
  case 1: b = 7;
  case 2: c = 8;
          break;
  default: a = 9;
}
```

```
for (i = 0; i < 100; i +=2)
  j += i;
```

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"

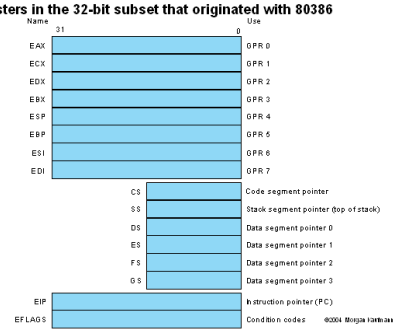
IA-32 Overview

- **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. 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.)

IA-32 Typical Instructions

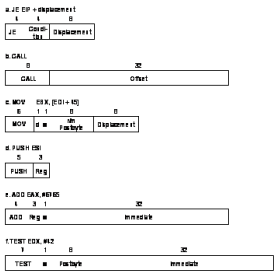
- **Four major types of integer instructions:**
  - **Data movement including move, push, pop**
  - **Arithmetic and logical (destination register or memory)**
  - **Control flow (use of condition codes / flags)**
  - **String instructions, including string move and string compare**

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!**

