

Computer Architecture

Introduction

Instructions

Dr. Arjan Durresi
Louisiana State University
Baton Rouge, LA 70810
Durresi@Csc.LSU.Edu

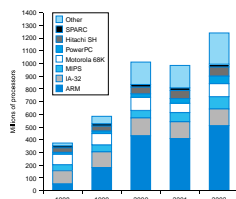
These slides are available at:
http://www.csc.lsu.edu/~durresi/CSC7080_05/



- Operation and Operands
- Representing Instructions
- Logical Operations
- MIPS addressing
- 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 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 code

- Compiling two C assignments

C code: $a = b + c;$
 $d = a - e;$

-

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

Compiling C code

- Compiling two C assignments

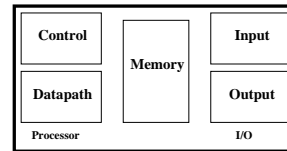
C code: `f = (g + h) - (i + j);`

-

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

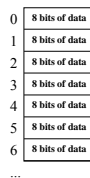
Registers vs. Memory

- Arithmetic instructions operands must be registers,
 - Only 32 registers provided
 - Register – special location built directly in hardware
- 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.



When an Operand is in Memory

- A is an array of 100 words and that the compiler has associated the variables `g` and `h` with the registers `$s1` and `$s2`. The starting address or base address of the array is in `$s3`
- C code: `g = h + A[8];`
- MIPS code: `lw $t0, 8($s3) # Temporary reg $t0 gets A[8]
add $s1, $s2, $t0 # g = h + A[8]`

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

Instructions

- Load and store instructions
- Example:
 - C code: `A[12] = h + A[8];`
 - MIPS code: `lw $t0, 32($s3) # Temp. reg $t0 gets [A8]
add $t0, $s2, $t0 # Temp. reg $t0 gets h + [A8]
sw $t0, 48($s3) # Stores h + A[8] into A[12]`

- 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)`

Registers vs. Memory

- Registers are smaller than memory \Rightarrow Data access is faster in registers.
- A MIPS arithmetic instruction can read two registers, operate on them, and write the result.
- A MIPS data transfer instruction only reads one operand or write one operand.
- Registers – take less time to access data and have a higher throughput than mamory

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;
}
    
```

\Rightarrow

```

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:

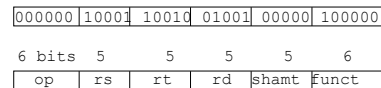
- MIPS
 - loading words but addressing bytes
 - arithmetic on registers only

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

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:



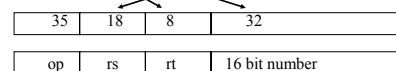
- Can you guess what the field names stand for?

MIPS Instructions

- op*: Basic operation of the instruction, *opcode*
- rs*: The first register source operand
- rt*: The second register source operand
- rd*: The register destination operand. It gets the result of the operation
- shamt*: Shift amount
- Funct*: Function. This field selects the specific variant of the operation in the *op* field and is called *function code*.

Machine Language

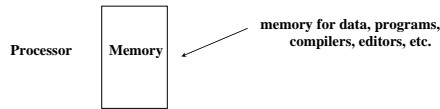
- Consider the load-word and store-word instructions,
 - What would the regularity principle have us do?
 - New principle: Good design demands a compromise
- Introduce a new type of instruction format
 - I-type for data transfer instructions (Immediate type)
 - other format was R-type for register (Register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



- 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

The Stored-program Concept



Control

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

```
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?

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.

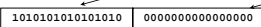
- MIPS Instructions:

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

## How about larger constants?

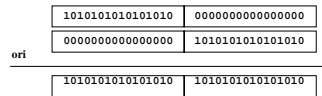
- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction

```
lui $t0, 1010101010101010 filled with zeros
```



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

```
ori $t0, $t0, 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

- Support for procedures
  - linkers, loaders, memory layout
  - stacks, frames, recursion
  - manipulating strings and pointers
  - interrupts and exceptions
  - system calls and conventions

## 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  $\$t4 \neq \$t5$
  - `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 | 26 bit address |    |    |                |
- Addresses are not 32 bits
  - How do we handle this with load and store instructions?

## Addresses in Branches

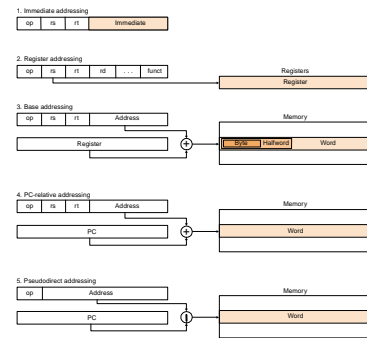
- Instructions:
  - `bne $t4, $t5, Label` Next instruction is at Label if  $\$t4 \neq \$t5$
  - `beq $t4, $t5, Label` Next instruction is at Label if  $\$t4 = \$t5$
- Formats:
 

|   |    |    |    |                |
|---|----|----|----|----------------|
| 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                 | <code>\$a0-\$a7, \$t0-\$t9, \$k0-\$k1, \$s0-\$s3, \$v0-\$v1, \$fp, \$ra</code> | Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register base always equals 0. Register file is reserved for the assembler to handle branch constants.                       |
| 2 <sup>30</sup> memory words | <code>Memory[0], Memory[4], ... Memory[4294967295]</code>                      | Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and global variables, such as those passed on procedure calls. |

| MIPS assembly language |                                   |                                   |                                                                                                                                  |
|------------------------|-----------------------------------|-----------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| Category               | Instruction                       | Example                           | Comments                                                                                                                         |
| Arithmetic             | <code>add</code>                  | <code>add \$a1, \$a2, \$a3</code> | Three operands; data in registers                                                                                                |
|                        | <code>subtract</code>             | <code>sub \$a1, \$a2, \$a3</code> | Three operands; data in registers                                                                                                |
|                        | <code>add immediate</code>        | <code>addi \$a1, \$a2, 100</code> | Used to add constants                                                                                                            |
| Data transfer          | <code>load word</code>            | <code>lw \$a1, 100(\$a2)</code>   | $\$a1 = \text{Memory}[\$a2 + 100]$ Word from memory to register                                                                  |
|                        | <code>store word</code>           | <code>sw \$a1, 100(\$a2)</code>   | $\text{Memory}[\$a2 + 100] = \$a1$ Word from register to memory                                                                  |
|                        | <code>load byte</code>            | <code>lb \$a1, 100(\$a2)</code>   | $\$a1 = \text{Memory}[\$a2 + 100]$ Byte from memory to register                                                                  |
|                        | <code>load upper immediate</code> | <code>lui \$a1, 100</code>        | $\$a1 = 100 \cdot 2^{20}$ Words constant in upper 16 bits                                                                        |
| Conditional branch     | <code>branch on equal</code>      | <code>beq \$a1, \$a2, 25</code>   | If $\$a1 == \$a2$ go to PC + 4 + 25<br>Equal test; PC-relative branch                                                            |
|                        | <code>branch on not equal</code>  | <code>bne \$a1, \$a2, 25</code>   | If $\$a1 \neq \$a2$ go to PC + 4 + 25<br>Not equal test; PC-relative                                                             |
|                        | <code>set on less than</code>     | <code>slt \$a1, \$a2, \$a3</code> | If $\$a2 < \$a3$ set $\$a1 = 1$ , else $\$a1 = 0$<br>Compare less than; for <code>beq</code> , <code>bne</code> false $\$a1 = 0$ |
| Unconditional          | <code>jump</code>                 | <code>j \$a1</code>               | Jump to target address                                                                                                           |
|                        | <code>jump register</code>        | <code>jr \$a1</code>              | Jump to register address                                                                                                         |
| Unconditional          | <code>return</code>               | <code>ret</code>                  | For <code>push</code> , procedure return                                                                                         |
| Unconditional          | <code>return and link</code>      | <code>rlw \$a1, \$a2</code>       | For <code>push</code> , 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”

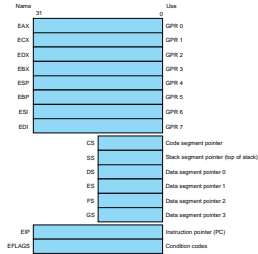
## IA-32 Overview

- 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 operand                                     | Address is in register                                                                       | not EIP or ESP                                                     | for EIP: R15(R13) |
| Base or scaled index with 8 or 32-bit displacement   | Address is constant if base register only; displacement                                      | not EIP or ESP for EIP: R15(R13) or R11(R10) for EIP: R11(R10)     |                   |
| Base plus scaled index                               | This registers to Base + C <sup>2</sup> * S * Index, where Base has the value 0, 1, 2, or 3. | Base: any GPR Index: not EIP for EIP: R15, R11, R10, R13, R12, R14 |                   |
| Base plus scaled index with 8 or 32-bit displacement | Base + C <sup>2</sup> * S * Index + displacement, where Base has the value 0, 1, 2, or 3.    | Base: any GPR Index: not EIP for EIP: R15, R11, R10, R13, R12, R14 |                   |

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 based on EIP or the EIP, is included to avoid the ambiguity for base factor of 2 to mean an index as a register into a base address (see Figure 2-36 and 2-38), a scale factor of 2 to mean for 16-bit data, and a scale factor of 2 to mean for 32-bit data. Scale factor of 2 means the address is not scaled. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent code would need two more instructions: a 16-bit load for register to hold the displacement and an EIP to use the register address with the base register R11. (Load gives the 48-bit base address to what is called base addressing mode—based and based+—that they are essentially identical and we combine them here.)

## IA-32 Typical Instructions

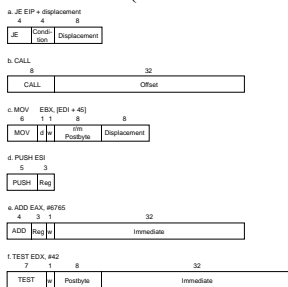
- Four major types of integer instructions:
  - Data movement including move, push, pop
  - Arithmetic and logical (destination register or memory)
  - Control flow (use of condition codes / flags)
  - String instructions, including string move and string compare

| Instruction                                                    | Function                                            |
|----------------------------------------------------------------|-----------------------------------------------------|
| CALL                                                           | Call procedure (pushes EIP on stack) (EIP = return) |
| CALL EBX                                                       | Call procedure (pushes EIP on stack) (EIP = return) |
| CALL EBX, 100h                                                 | Call procedure (pushes EIP on stack) (EIP = return) |
| CALL EBX, 100h, 100h                                           | Call procedure (pushes EIP on stack) (EIP = return) |
| CALL EBX, 100h, 100h, 100h                                     | Call procedure (pushes EIP on stack) (EIP = return) |
| CALL EBX, 100h, 100h, 100h, 100h                               | Call procedure (pushes EIP on stack) (EIP = return) |
| CALL EBX, 100h, 100h, 100h, 100h, 100h                         | Call procedure (pushes EIP on stack) (EIP = return) |
| CALL EBX, 100h, 100h, 100h, 100h, 100h, 100h                   | Call procedure (pushes EIP on stack) (EIP = return) |
| CALL EBX, 100h, 100h, 100h, 100h, 100h, 100h, 100h             | Call procedure (pushes EIP on stack) (EIP = return) |
| CALL EBX, 100h, 100h, 100h, 100h, 100h, 100h, 100h, 100h       | Call procedure (pushes EIP on stack) (EIP = return) |
| CALL EBX, 100h, 100h, 100h, 100h, 100h, 100h, 100h, 100h, 100h | Call procedure (pushes EIP on stack) (EIP = return) |

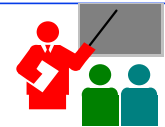
FIGURE 2-43 Some typical IA-32 instructions and their functions. (A list of Register operations appears in Figure 2-44. The CALL uses the EIP of the next instruction on the stack (EIP is the base PC.)

## 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!