

Instruction Set Architecture for MIPS Processors

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

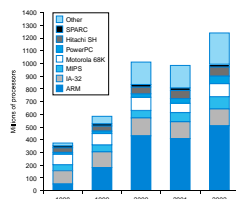
These slides are available at:
http://www.csc.lsu.edu/~durresi/CSC3501_05/



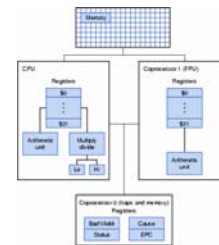
- Operations and Operands of the Computer Hardware
- Representing Instructions in Computer
- MIPS addressing
- An Introduction to Compilers
- IA-32 Instructions

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?

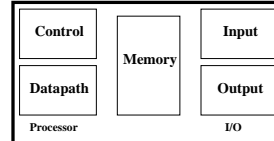
Compiling C Assignments into MIPS

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

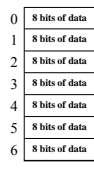
- CPU:
 - 32 32-bit general purpose registers - GPRs (r0 - r31);
 - r0 has fixed value of zero. Attempt to writing into r0 is not illegal, but its value will not change;
 - two 32-bit registers - Hi & Lo, hold results of integer multiply and divide
 - 32-bit program counter - PC;
- Floating Point Processor - 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 |
|-----------------------------------|--------------------------------------|
| <code>add \$s1, \$s2, \$s3</code> | <code>\$s1 = \$s2 + \$s3</code> |
| <code>sub \$s1, \$s2, \$s3</code> | <code>\$s1 = \$s2 - \$s3</code> |
| <code>lw \$s1, 100(\$s2)</code> | <code>\$s1 = Memory[\$s2+100]</code> |
| <code>sw \$s1, 100(\$s2)</code> | <code>Memory[\$s2+100] = \$s1</code> |

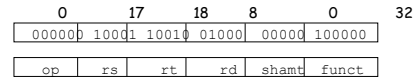
MIPS Instructions

- ❑ 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, $s2`
 - registers have numbers, `$t1=8, $s1=17, $s2=18`

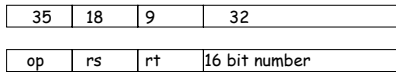
- ❑ Instruction Format:



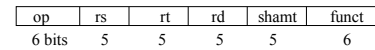
- ❑ Can you guess what the field names stand for?
- ❑ All MIPS instructions are 32 bits long

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)`



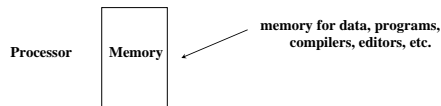
MIPS Fields



- ❑ 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

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?

So far:

- Instruction

| 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
else
    $t0 = 0
sit $t0, $s1, $s2
```

- 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
```

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
1010101010101010 | 0000000000000000
```

filled with zeros

- 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.

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 | 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:

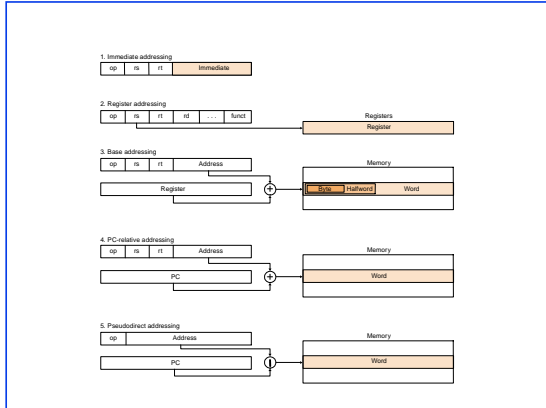
| | | | |
|----|----|----|----------------|
| op | rs | rt | 16 bit address |
|----|----|----|----------------|

- 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:

| Name | Example | MIPS operands | Comments |
|------------------------------|--|---------------|---|
| 32 registers | \$r0-\$r31, \$f0-\$f31, \$zero, \$a0-\$a31, \$v0-\$v31, \$gp, \$fp, \$sp, \$ra, \$k0, \$k1 | | Pair locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register \$zero always equals 0. Register \$a0 is reserved for the operating system to handle large constants. |
| 2 ¹⁶ memory words | Memory[0], Memory[4], Memory[256*7292] | | Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and control registers, such as those used in processor cache. |

| Category | Instruction | Example | Meaning | Comments |
|--------------------|---|-----------------------|--|--|
| Arithmetic | add | add \$a1, \$a2, \$a3 | \$a1 = \$a2 + \$a3 | These operands, data in registers |
| | addui | addui \$a1, \$a2, 100 | \$a1 = \$a2 + 100 | These operands; data in registers |
| | add immediate | addi \$a1, \$a2, 100 | \$a1 = \$a2 + 100 | Works in add constant |
| Data transfer | load word | lw \$a1, 100(\$a2) | \$a1 = Memory[\$a2 + 100] | Word from memory to register |
| | store word | sw \$a1, 100(\$a2) | Memory[\$a2 + 100] = \$a1 | Word from register to memory |
| | load byte | lb \$a1, 100(\$a2) | \$a1 = Memory[\$a2 + 100] | Byte from memory to register |
| Conditional branch | branch on equal | beq \$a1, \$a2, 25 | if(\$a1 == \$a2) go to PC = 4 + 100 | Equal test; PC-relative |
| | branch on not equal | bne \$a1, \$a2, 25 | if(\$a1 != \$a2) go to PC = 4 + 100 | Not equal test; PC-relative |
| | set on less than | slt \$a1, \$a2, \$a3 | if(\$a2 < \$a3) \$a1 = 1, else \$a1 = 0 | Compare less than; for bne, bne \$a1 = 0 |
| Jump | jump | j 2550 | go to 10000 | Jump to nearest address |
| | jump register | jr \$ra | go to \$ra | Go to register address; for exception call |
| Branch/Local jump | branch on less than immediate | blt \$a1, \$a2, 100 | if(\$a1 < \$a2) \$a1 = 1, else \$a1 = 0 | Compare less than constant |
| | branch on less than or equal | bltle \$a1, \$a2, 100 | if(\$a1 <= \$a2) \$a1 = 1, else \$a1 = 0 | Compare less than or equal constant |
| | branch on greater than or equal | bgtge \$a1, \$a2, 100 | if(\$a1 >= \$a2) \$a1 = 1, else \$a1 = 0 | Compare greater than or equal constant |
| | branch on greater than | bgt \$a1, \$a2, 100 | if(\$a1 > \$a2) \$a1 = 1, else \$a1 = 0 | Compare greater than constant |
| | branch on less than or equal immediate | bltle \$a1, \$a2, 100 | if(\$a1 <= \$a2) \$a1 = 1, else \$a1 = 0 | Compare less than or equal constant |
| | branch on greater than or equal immediate | bgtge \$a1, \$a2, 100 | if(\$a1 >= \$a2) \$a1 = 1, else \$a1 = 0 | Compare greater than or equal constant |
| | branch on less than immediate | blt \$a1, \$a2, 100 | if(\$a1 < \$a2) \$a1 = 1, else \$a1 = 0 | Compare less than constant |
| | branch on less than or equal immediate | bltle \$a1, \$a2, 100 | if(\$a1 <= \$a2) \$a1 = 1, else \$a1 = 0 | Compare less than or equal constant |
| | branch on greater than immediate | bgt \$a1, \$a2, 100 | if(\$a1 > \$a2) \$a1 = 1, else \$a1 = 0 | Compare greater than constant |
| | branch on greater than or equal immediate | bgtge \$a1, \$a2, 100 | if(\$a1 >= \$a2) \$a1 = 1, else \$a1 = 0 | Compare greater than or equal constant |



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 EIP or ESP | for R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31 |
| Base index with 8 or 32-bit displacement | Address is constant or base register plus displacement | not EIP or ESP | for R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31 |
| Base plus scaled index | The address is: Base + (CRP * index), where CRP is the value 0, 1, 2, or 3 | Base: any CRP; Index: not EIP | not applicable |
| Base plus scaled index with 8 or 32-bit displacement | The address is: Base + CRP * index + displacement, where CRP is the value 0, 1, 2, or 3 | Base: any CRP; Index: not EIP | not applicable |

FIGURE 3-42 IA-32 32-bit addressing modes with register restrictions and the equivalent MIPS mode. The three plus scaled index addressing modes, not listed in MIPS or the MIPS PC, are included to avoid the confusion for those modes that use 12 to store an index in a register into a data address (see Figures 3-76 and 3-20). A scale factor of 1 is used for the data, and a scale factor of 4 for the bit data. Scale factor of 4 means the address is not valid. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent mode would need two more conditions: a 1 to test the upper 16 bits of the displacement and an add to use the upper address with the base register R13. (Does anyone use all those modes to what is called base addressing modes—base and base+index—that they are essentially identical and we consider them here.)

IA-32 Typical Instructions

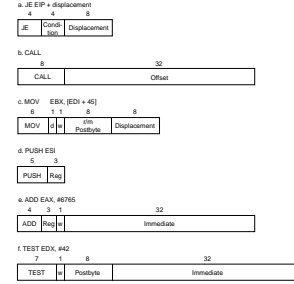
- Four major types of integer instructions:
 - Data movement including move, push, pop
 - Arithmetic and logical (destination register or memory)
 - Control flow (use of condition codes / flags)
 - String instructions, including string move and string compare

| Instruction | Operation |
|------------------|---|
| JE name | if (ZF=1) cond(1) then code 1; EIP=next I; else EIP = name + 1; EIP=I+5 |
| JEI name | EIP=code |
| CALL name | (EIP ← E, EIP ← 4 (EAX, EIP+name) |
| MOV EBX, 1101401 | EAX ← EBX ← 401 |
| PUSH ESI | (EIP ← 4, EIP ← 4) |
| POP ESI | (EIP ← 4, EIP ← 4) |
| ADD EAX, 80765 | EAX ← EAX + 80765 |
| TEST EDI, 8A3 | set condition codes (flags) with EDI and 8A3 |
| MOVB | (EIP ← 4, EIP ← 4) |

FIGURE 2.43 Some typical IA-32 instructions and their functions. A list of frequent operations appears in Figure 2.44. The CALL uses the EIP of the next instruction on the stack. EIP is the final 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!