Universidade Federal do Rio Grande do Sul
Instituto de Informática

Organização de Computadores

Aula 3

Arquitetura do processador MIPS

Instructions

- Language of the Machine
- More primitive than higher level languages
  e.g., no sophisticated control flow
- Very restrictive e.g., MIPS Arithmetic Instructions
  We'll be working with the MIPS instruction set architecture
  - similar to other architectures developed since the 1980's
  - used by NEC, Nintendo, Silicon Graphics, Sony

Design goals: maximize performance and minimize cost, reduce design time

Basic ISA Classes

Accumulator (1 register):

1 address	add A	acc ← acc + mem[A]
1+x address	addx A	acc ← acc + mem[A + x]

Stack:

0 address	add	tos ← tos + next

General Purpose Register:

2 address	add A B	EA(A) ← EA(A) + EA(B)
3 address	add A B C	EA(A) ← EA(B) + EA(C)

Load/Store:

load Ra Rb	Ra ← mem[Rb]
store Ra Rb	mem[Rb] ← Ra

Memory to Memory:

All operands and destinations can be memory addresses.

Comparing Number of Instructions

Comparison: Bytes per instruction? Number of Instructions? Cycles per instruction?

° Code sequence for C = A + B for four classes of instruction sets:

Stack	Accumulator	Register (register-memory)	Register (load-store)
Pop A	Load A	Load R1,A	Load R1,A
Pop B	Add B	Add R1,B	Load R2,B
Add	Store C	Store C, R1	Add R3,R1,R2
Push C			Store C,R3

RISC machines (like MIPS) have only load-store instns. So, they are also called load-store machines. CISC machines may even have memory-memory instrns, like mem (A) = mem (B) + mem (C) Advantages/disadvantages?

General Purpose Registers Dominate

Advantages of registers:

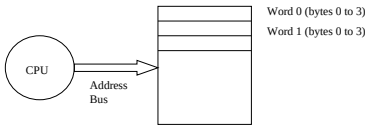
1. registers are faster than memory
2. registers are easier for a compiler to use
   e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order
3. registers can hold variables
   memory traffic is reduced, so program is sped up (since registers are faster than memory)
   code density improves (since register named with fewer bits than memory location)

MIPS Registers:

31 x 32-bit GPRs, (R0 =0), 32 x 32-bit FP regs (paired DP)

Memory Addressing

- Since 1980 almost every machine uses addresses to level of 8-bits (byte)
- 2 questions for design of ISA:
  - Read a 32-bit word as four loads of bytes from sequential byte addresses or as one load word from a single byte address, how do byte addresses map onto words?
  - Can a word be placed on any byte boundary?



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
...	

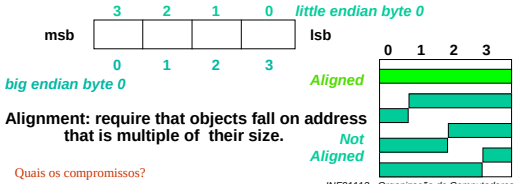
Addressing: Byte vs. word

- Every word in memory has an address, similar to an index in an array
- Early computers numbered words like C numbers elements of an array:
  - `Memory[0]`, `Memory[1]`, `Memory[2]`, …
- Today machines address memory as bytes, hence word addresses differ by 4
  - `Memory[0]`, `Memory[4]`, `Memory[8]`, …
    Called the "address" of a word
  - Computers needed to access 8-bit bytes as well as words (4 bytes/word)

Called "byte addressing"

Addressing Objects: Endianess and Alignment

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



Alignment: require that objects fall on address that is multiple of their size.

Quais os compromissos?

Addressing Modes

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

Why Auto-increment/decrement? Scaled?

Addressing Mode Usage? (ignore register mode)

3 programs measured on machine with all address modes (VAX)

--- Displacement: 42% avg, 32% to 55%

--- Immediate: 33% avg, 17% to 43%

75%

85%

--- Register deferred (indirect): 13% avg, 3% to 24%

--- Scaled: 7% avg, 0% to 16%

--- Memory indirect: 3% avg, 1% to 6%

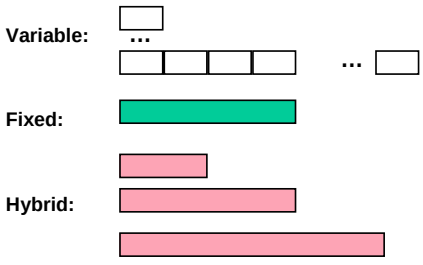
--- Misc: 2% avg, 0% to 3%

75% displacement & immediate
88% displacement, immediate & register indirect
Immediate Size:

50% to 60% fit within 8 bits

75% to 80% fit within 16 bits

Generic Examples of Instruction Formats



Summary of Instruction Formats

- If code size is most important, use variable length instructions:
  (1)Difficult control design to compute next address
  (2) complex operations, so use microprogramming
  (3) Slow due to several memory accesses
- If performance is over is most important, use fixed length instructions
  (1) Simple to decode, so use hardware
  (2) Wastes code space because of simple operations
  (3) Works well with pipelining
- Recent embedded machines (ARM, MIPS) added optional mode to execute subset of 16-bit wide instructions (Thumb, MIPS16); per procedure decide performance or density

Typical Operations (little change since 1960)

Data Movement	Load (from memory) Store (to memory) memory-to-memory move register-to-register move input (from I/O device) output (to I/O device) push, pop (to/from stack)
Arithmetic	integer (binary + decimal) or FP Add, Subtract, Multiply, Divide
Shift	shift left/right, rotate left/right
Logical	not, and, or, set, clear
Control (Jump/Branch)	unconditional, conditional
Subroutine Linkage	call, return
Interrupt	trap, return
Synchronization	test & set (atomic r-m-w)
String	search, translate
Graphics (MMX)	parallel subword ops (4 16bit add)

Top 10 80x86 Instructions

° Rank	instruction	Integer Average Percent total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

° Simple instructions dominate instruction frequency

Summary

While theoretically we can talk about complicated addressing modes and instructions, the ones we actually use in programs are the simple ones

=> RISC philosophy

Summary: Instruction set design (MIPS)

- Use general purpose registers with a load-store architecture: YES
- Provide at least 16 general purpose registers plus separate floating-point registers: 31 GPR & 32 FPR
- Support basic addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register deferred; : YES: 16 bits for immediate, displacement (disp=0 => register deferred)
- All addressing modes apply to all data transfer instructions : YES
- Use fixed instruction encoding if interested in performance and use variable instruction encoding if interested in code size : Fixed
- Support these data sizes and types: 8-bit, 16-bit, 32-bit integers and 32-bit and 64-bit IEEE 754 floating point numbers: YES
- Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch (with a PC-relative address at least 8-bits long), jump, call, and return: YES, 16b
- Aim for a minimalist instruction set: YES

Instruction Format Field Names

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

- Fields have names:
- op: basic operation of instruction, "opcode"
- rs: 1st register source operand
- rt: 2nd register source operand
- rd: register destination operand, gets the result
- shamt: shift amount (use later, so 0 for now)
- funct: function; selects the specific variant of the operation in the op field; sometimes called the function code

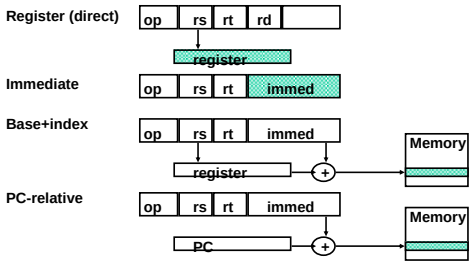
Notes about Register and Imm. Formats

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R:	op	rs	rt	rd	shamt	funct
I:	op	rs	rt	address		
	6 bits	5 bits	5 bits	16 bits		

- To make it easier for hardware (HW), 1st 3 fields same in R-format and I-format
- Alas, `rt` field meaning changed
  - R-format: `rt` is 2nd source operand
  - I-format: `rt` can be destination operand
- How does HW know which format is which?
  - Distinct values in 1st field (op) tell whether last 16 bits are 3 fields (R-format) or 1 field (I-format)

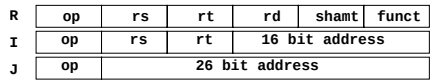
MIPS Addressing Modes/Instruction Formats

- All instructions 32 bits wide



MIPS Instruction Formats

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats
- rely on compiler to achieve performance — what are the compiler's goals?
- help compiler where we can



Arquitetura do processador MIPS

1. Registradores
2. Tipos de dados
3. Modos de endereçamento
4. Formatos das instruções
5. Tipos de instruções

1. Registradores

- 32 registradores de propósitos gerais de 32 bits
  - $0, $1, …, $31
  - operações inteiras
  - endereçamento
- $0 tem sempre valor 0
- $31 guarda endereço de retorno de sub-rotina
- 32 registradores de ponto flutuante de 32 bits (precisão simples) $f0, $f1, …, $f31
  - podem ser usados em pares para precisão dupla
- registradores Hi e Lo para uso em multiplicação e divisão

2. Tipos de dados

- dados inteiros disponíveis em instruções load e store
  - bytes
  - meias-palavras de 16 bits
  - palavras de 32 bits
- dados inteiros disponíveis em instruções aritméticas e lógicas
  - meias-palavras de 16 bits (estendidos para 32 bits por que?)
  - palavras de 32 bits
- dados em ponto flutuante
  - precisão simples em 32 bits (expoente: 8 bits, magnitude: 24 bits)
  - precisão dupla em 64 bits (expoente: 11 bits, magnitude: 53 bits)

3. Modos de endereçamento

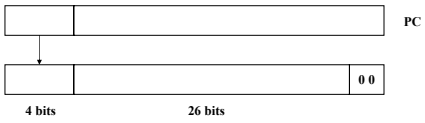
- acessos à memória devem ser alinhados
  - dados de 32 bits devem estar em endereços múltiplos de 4
  - dados de 16 bits devem estar em endereços múltiplos de 2 por que?
- modo registrador
  - para instruções aritméticas e lógicas: dado está em registrador
  - para instruções de desvio incondicional: endereço está em registrador
- modo base e deslocamento
  - para instruções load e store
  - base é registrador inteiro de 32 bits
  - deslocamento de 16 bits contido na própria instrução
- modo relativo ao PC
  - para instruções de branch condicional
  - endereço é a soma do PC com deslocamento contido na instrução
  - deslocamento é dado em palavras e precisa ser multiplicado por 4

Modos de endereçamento

- modo imediato
  - para instruções aritméticas e lógicas
  - dado imediato de 16 bits contido na própria instrução
  - dado é estendido para 32 bits
    - extensão com sinal nas instruções aritméticas
    - extensão sem sinal nas instruções lógicas
- para que se possa especificar constantes de 32 bits
  - instrução lui (load upper immediate)
  - carrega 16 bits imediatos na parte superior do registrador
  - parte inferior do registrador é zerada
  - instrução seguinte precisa fazer soma imediata do registrador com 16 bits da parte inferior

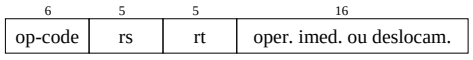
Modos de endereçamento

- modo absoluto
  - para instruções de desvio incondicional
  - instrução tem campo com endereço de palavra com 26 bits
  - endereço de byte obtido com dois bits menos significativos iguais a 0
  - 4 bits mais significativos obtidos do PC
  - só permite desvios dentro de uma área de 256 Mbytes



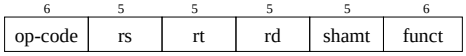
4. Formatos das instruções

- todas as instruções têm 32 bits
  - todas têm op-code de 6 bits
  - modo de endereçamento é codificado juntamente com o op-code
- instruções de tipo I
  - loads, stores
  - operações aritméticas e lógicas com operando imediato
  - desvios condicionais (“branches”)
  - desvios incondicionais para endereço em registrador

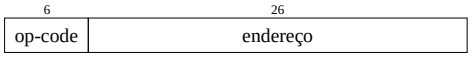


Formatos das instruções

- **instruções de tipo R**
  - **instruções aritméticas e lógicas**
  - **instruções de movimentação entre registradores**
  - **"shamt" (shift amount) é usado em instruções de deslocamento**
  - **"funct" é a operação a ser feita pela ALU**

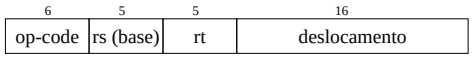


- **instruções de tipo J**
  - **desvios com endereçamento absoluto**
  - **chamada de sub-rotina**



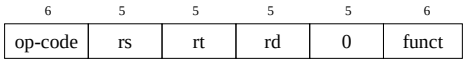
5. Tipos de instruções

- **instruções load / store**
  - **são sempre tipo I**
  - **qualquer registrador de propósitos gerais pode ser carregado ou armazenado da / na memória**
  - **pode-se carregar ou armazenar bytes, meias palavras, palavras**
  - **endereçamento sempre por base e deslocamento**
- **lb, lh, lw – load byte, halfword, word**
  - **sinal é estendido em lb e lh**
- **lbu, lhu – load byte, halfword sem extensão de sinal**
- **sb, sh, sw – store byte, halfword, word**

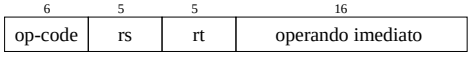


Instruções aritméticas e lógicas

- **operação entre 2 registradores, resultado num terceiro registrador**
  - **tipo R**
- **add, sub, and, or, nor, xor**
- **comparação slt – compara dois registradores e coloca valor 1 ou 0 em registrador destino**



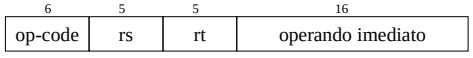
- **existem versões com operando imediato, de tipo I**
  - **addi, andi, ori, xori, slti**



Instruções aritméticas e lógicas

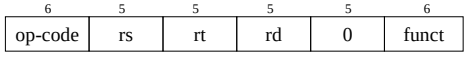
- **$0 usado para sintetizar operações populares**
- **carga de constante = soma imediata onde $0 é um dos operandos**

```
addi   $5, $0, 10
```



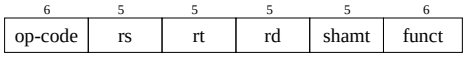
- **mover de registrador para registrador = soma com $0**

```
add    $6, $2, $0
```



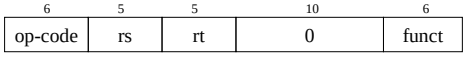
Instruções aritméticas e lógicas

- **instruções de deslocamento variável**
  - **tipo R**
  - **sllv, srlv – shift lógico (entra 0 na extremidade)**
  - **srav – shift aritmético (duplica sinal)**
  - **desloca registrador rt pela distância especificada no registrador rs e coloca resultado no registrador rd**
- **instruções de deslocamento constante**
  - **tipo R**
  - **sll, sra, srl**
  - **desloca registrador rt pela distância especificada no campo "shamt" e coloca resultado no registrador rd**



Instruções aritméticas e lógicas

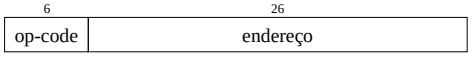
- **instrução de multiplicação: mul**
  - **multiplica registradores rs e rt**
  - **resultado colocado em hi (32 msb) e lo (32 lsb)**
- **instrução de divisão: div**
  - **divide registrador rs pelo registrador rt**
  - **quociente colocado em lo**
  - **resto colocado em hi**



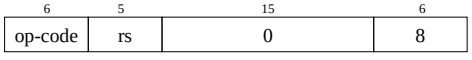
- **instruções de movimentação permitem transferir dados entre hi e lo e os demais registradores**
  - **mfhi Rd, mflo Rd**
  - **mthi Rs, mtlo Rs**

Instruções de desvio incondicional

- **instrução j**
  - **tipo J**
  - **endereço destino = concatenação dos 4 msb do PC com endereço imediato de 28 bits**

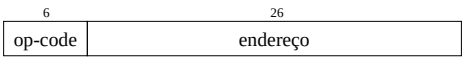


- **instrução jr**
  - **tipo I: endereço destino contido em registrador**
  - **também serve para retornar de sub-rotina**

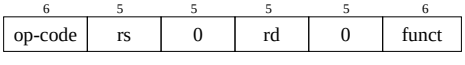


Instruções de desvio incondicional

- **instrução jal (jump and link)**
  - **tipo J**
  - **desvio para sub-rotina, endereço especificado na instrução**
  - **endereço de retorno salvo em $31**



- **instrução jalr (jump and link register)**
  - **tipo R**
  - **desvio para sub-rotina, endereço especificado em rs**
  - **endereço de retorno salvo em rd**



Instruções de desvio condicional

- **são sempre tipo I**
- **endereço destino = soma do PC com offset imediato de 16 bits**
- **instruções que testam um único registrador**
  - **bgez, bgtz, blez, bltz – desvia se registrador é $\geq, >, \leq, <$ zero**
  - **bgezal, bltzal – como bgez e bltz, mas salva endereço de retorno em $31**
- **instruções que comparam dois registradores**
  - **beq, bne – desvia se registradores são iguais (ou diferentes)**

6	5	5	16
op-code	rs	rt	offset

Overflow em instruções aritméticas e lógicas

- **instruções "unsigned" não geram overflow**
  - **addu, addiu, divu, multu, subu, sltu, sltiu**
- **instruções "não-unsigned" geram overflow**
  - **endereço da instrução que causou overflow é colocado no registrador EPC = registrador 0 de um co-processador**
  - **instrução "mfc0" copia valor do EPC para outro registrador qualquer**
  - **instrução "jr" (jump para endereço especificado por registrador) é usada para desviar para rotina de atendimento**

Instruções de ponto flutuante

- **mover operandos de precisão simples ou dupla entre registradores**
  - **mov.d, mov.s**
- **soma, subtração, negação, multiplicação, divisão em precisão simples e dupla**
  - **add.s, sub.s, neg.s, mul.s, div.s**
  - **add.d, sub.d, neg.d, mul.d, div.d**
- **comparações em precisão simples e dupla**
  - **c.eq.s, c.le.s, c.lt.s,**
  - **c.eq.d, c.le.d, c.lt.d,**
- **conversão para ponto flutuante de precisão simples (ou dupla)**
  - **cvt.s.d, cvt.s.w – converter FP double (ou inteiro) para FP single**
  - **cvt.d.s, cvt.d.w – converter FP single (ou inteiro) para FP double**
- **conversão de ponto flutuante para inteiro**
  - **cvt.w.s, cvt.w.d – converter FP double (ou single) para inteiro**

MIPS Instructions:

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