



Arquitectura dos Computadores

Aula 2 – Introdução ao MIPS

Professor: Dr. Christophe Soares



Pensamento...

"I've found that the key difference between mediocre and excellent programmers is whether or not they know assembly language" Slashdot.org

Assembler

Função do CPU: executar MUITAS instruções

Instruções: Operações básicas que o CPU sabe executar

Diferentes CPUs implementam diferentes conjuntos de instruções

Conjunto de instruções designado por “Instruction Set Architecture (ISA)”

- Intel x86,
- ARM,
- MIPS, ...

Conjuntos de Instruções

CISC (*Complex Instruction Set Computing*)

- Inicialmente a ideia passava por aumentar o conjunto de instruções com operações complexas em hardware
- VAX tinha até uma instrução para multiplicar polinómios!

RISC (*Reduced Instruction Set Computing*)

- Manter o conjunto de instruções pequeno e simples o que ajuda a construir hardware mais rápido
- Operações complexas são decompostas em várias instruções/operações simples

Arquitetura MIPS

MIPS – Companhia que criou uma das primeiras implementações comerciais da arquitetura RISC

<https://www.imaginationtech.com/>

Porquê estudar MIPS em vez de Intel 80x86 ou ARM?

- MIPS é simples e elegante.
- MIPS implementa o conceito RISC
- Conhecendo MIPS é simples perceber ARM



Variáveis em Assembler: Registros (1/4)

Ao contrário das linguagens de alto nível como C ou Java em assembler não há variáveis

- Porque não? Manter o hardware simples!

Os operandos em assembler são os registos

- Número limitado de localizações especiais criadas diretamente no hardware.
- Todas as operações são executadas nos registos!

Benefício: Os registos são muito rápidos! (menos 1e-9 segundo)

Variáveis em Assembler: Registros (2/4)

Inconveniente: os registos são implementados em hardware, logo o seu nº é limitado

Solução: o código MIPS tem que fazer um uso eficiente ds registos.

- 32 registos no MIPS
- Porquê 32? “*Smaller is faster*”

Cada registo MIPS tem 32 bits

- Um grupo de 32 bits, é designado por: palavra ou word

Variáveis em Assembler: Registros (3/4)

Os registos estão numerados de 0 a 31

Cada registo pode ser referenciado pelo número

Referência numérica:

- **\$0, \$1, \$2, ... \$30, \$31**

Variáveis em Assembler: Registros (4/4)

Por convenção, cada registo pode também ser referenciado pelo seu nome

Por agora:

\$16 - \$23 → \$s0 - \$s7 (8 variáveis “preservadas” → C)

\$8 - \$15 → \$t0 - \$t7 (8 varáveis temporárias)

Os restantes 16 nomes vêm mais tarde...

Em geral preferimos usar nomes para tornar o código mais legível

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7ffffefffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

Registros vs. Variáveis em C ou Java

Em C, por exemplo, as variáveis são declaradas como sendo de um tipo.

- **Exemplo:** `int fahr, celsius;`
 `char a, b, c, d, e;`

Cada variável só pode contar dados desse mesmo tipo!

Em assembler os registos não têm tipo

A operação a executar é que vai determinar o significado do seu conteúdo.

Comentários em Assembler

Outra forma de tornar o código mais legível: Comentários!

O cardinal (#) é usado para comentários no MIPS

Qualquer coisa depois de um # e até ao fim da linha é ignorado

Nota: Diferente do C!

**Comentários em C podem ter o formato /* comentário */
portanto podem ocupar várias linhas.**

Adição e subtração no MIPS (1/4)

Sintaxe das instruções:

1 2, 3, 4

- 1) Nome da operação**
- 2) Operando que recebe o resultado (“destino”)**
- 3) 1º operando (“origem1”)**
- 4) 2º operando (“origem2”)**

A sintaxe é rígida:

1 operador, 3 operandos

Porquê? Manter o hardware simples pela regularidade.

Adição e subtração no MIPS (2/4)

Adição em assembler

int a = b + c // C → add \$s0, \$s1, \$s2 # MIPS

\$s0 – a
\$s1 – b
\$s2 – c

Subtração em assembler

int d = e - f // C → sub \$s3, \$s4, \$s5 # MIPS

\$s3 – d
\$s4 – e
\$s5 – f

Adição e subtração no MIPS (3/4)

Como fica em assembler a declaração seguinte?

```
int a = b + c + d - e; // C
```

Decomposta em várias instruções...

add \$t0, \$s1, \$s2 # temp = b + c	\$s0 – a
add \$t0, \$t0, \$s3 # temp = temp + d	\$s1 – b
sub \$s0, \$t0, \$s4 # a = temp - e	\$s2 – c
	\$s3 – d
	\$s4 – e

Nota: uma linha em C pode ser convertida em diversas linhas de código MIPS

Adição e subtração no MIPS (4/4)

Como é possível codificar em assembler do MIPS a declaração seguinte?

```
int f = (g + h) - (i + j); // C
```

Usando registos temporários intermédios

add \$t0,\$s1,\$s2 # temp	= g + h	\$s0 - f
add \$t1,\$s3,\$s4 # temp1	= i + j	\$s1 - g
sub \$s0,\$t0,\$t1 # f	= (g+h)-(i+j)	\$s2 - h
		\$s3 - i
		\$s4 - j

Registo Zero

Existe uma constante (“*immediate*”), o zero (0), que aparece frequentemente no código.

Portanto definimos o registo zero (\$0 ou \$zero) que tem sempre o valor zero.

- Exemplo:

int f = g // C → add \$s0,\$s1,\$zero # MIPS

\$s0 – f
\$s1 – g

\$0 é definido em hardware, portanto a instrução:

add \$zero,\$zero,\$s0 # MIPS

não faz nada!

Constantes (1/2)

Na bibliografia (“*immediates*”)

“Add Immediate”:

```
int f = g + 10 // C    →    addi $s0, $s1, 10 # MIPS      $s0 - f  
                                $s1 - g
```

A sintaxe é similar à da instrução add só que o último operando é uma constante e não um registo.

Constantes (2/2)

Não existe “Subtract Immediate” no MIPS: Porquê?

Limitar os tipos de operações que se podem executar

Se uma operação pode ser decomposta noutra mais simples então não deve ser incluída!

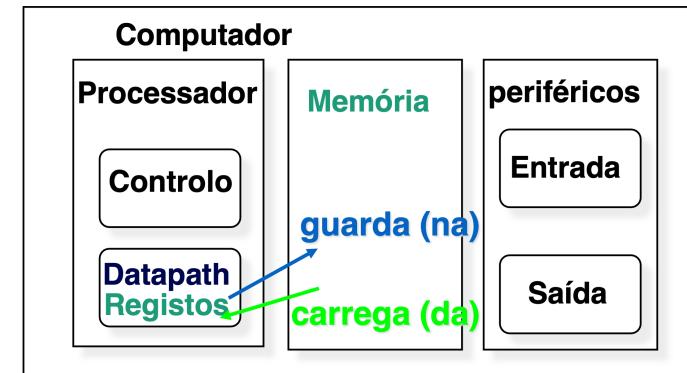
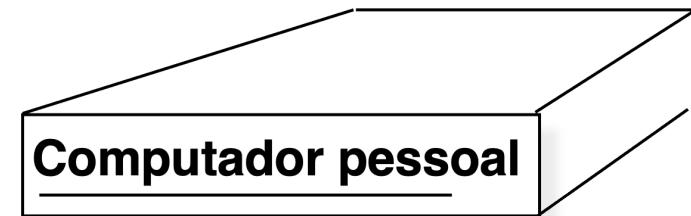
int f = g - 10 // C → addi \$s0,\$s1,-10 # EXISTE EM MIPS

Onde os registos MIPS \$s0,\$s1 estão associados ás variáveis f, g

Interacção com a memória

Os registos encontram-se no barramento de dados do processador;

Se os operandos se encontram na memória é necessário transferi-los para os registos para executar as operações, e, no final transferi-los de volta para a memória.



Transferência de dados: Memória >> Registos (1/4)

- Para transferir uma palavra da memória para os registos temos que indicar:
Registo: indicando o registo (\$0 - \$31) ou o nome (\$s0,..., \$t7)
- Endereço de memória, é na teoria mais difícil!
 - Temos que imaginar a memória como um vetor em que cada posição possui um endereço
 - queremos endereçar uma posição de memória a partir deste endereço (“base address” mais “offset”)

Importante: “*Load FROM memory*”

Transferência de dados: Memória >> Registros (2/4)

Para especificar um endereço de memória a partir do qual copiar, são necessárias duas coisas:

- Um registo contendo um apontador para a memória
- Um deslocamento numérico em bytes (“*offset*”)

O endereço de memória desejado é a soma destes dois valores.

Exemplo: 8 (\$t0)

Especifica o endereço de memória apontado por \$t0 mais 8 bytes.

Transferência de dados: Memória >> Registos (3/4)

- Sintaxe da instrução “Load”:
- 1 2, 3 (4)**
- 1) Nome da operação**
 - 2) Registo que vai receber o valor**
 - 3) O deslocamento (“offset”) em bytes**
 - 4) Registo contendo um apontador para a memória.**

Nome da instrução no MIPS:

lw : “Load Word”, portanto 32 bits - ou uma palavra - são carregados de cada vez

Transferência de dados: Memória >> Registros (4/4)



Exemplo: **lw \$t0, 12 (\$s0) # MIPS**

Esta instrução pega no apontador em \$s0, soma-lhe 12 bytes, e depois carrega o conteúdo da posição de memória apontado por esta soma no registo \$t0

Notas:

\$s0 é o registo base (“*base register*”)

12 é o deslocamento (“*offset*”)

O deslocamento é geralmente usado para aceder aos elementos de um “*array*”: o registo base aponta para o início do “*array*”

Transferência de dados: Registros >> Memória

Os dados também têm de passar dos registos para a memória. A sintaxe do “Store” é idêntica à do “Load”

Nome no MIPS :

sw - “Store Word”, portanto 32 bits (ou uma palavra) passam do registo para a memória

Exemplo: `sw $t0, 12 ($s0) # MIPS`

Esta instrução pega no apontador em \$s0, soma-lhe 12 bytes, e depois carrega o conteúdo do registo \$t0 na posição de memória apontada por esta soma.

Importante: “*Store INTO memory*”

Apontadores vs. Valores

Importante:

Um registo pode conter um qualquer valor de 32 bits. Esse valor pode ser um inteiro, com ou sem sinal, um apontador (endereço de memória), etc.

Se escrever

add \$t2, \$t1, \$t0

**então \$t0 e \$t1
devem conter NÚMEROS**

Se escrever

lw \$t2, 0 (\$t0)

**Então \$t0 deve conter
um ENDEREÇO**

Não confundir!

Exercício

Memória		Registos	
Endereço	Conteúdo	Nome	Conteúdo
.....
0000AAA0	00000003	\$0	0000AAA4
0000AAA4	0000000A	\$1	0000AAA8
0000AAA8	00000002	\$2	00000002
0000AAAC	00000001	\$3	00000000
.....

Qual o conteúdo de \$S3 depois de:

lw \$t0, 0 (\$s0)
lw \$t1, 4 (\$s0)
add \$t2, \$t0, \$t1
add \$s3, \$s3, \$s2
add \$s3, \$s3, \$t2

Qual o conteúdo da memória depois de:

add \$t1, \$zero, \$s2
addi \$t1, \$t1, 12
lw \$t0, 4 (\$s1)
add \$t1, \$t1, \$t0
sw \$t1, 0 (\$s1)

Conclusão...

- RISC vs CISC
- ADD – SUB
- Constantes & *Immediates*
- Primeiros passos na programação do MIPS
- http://en.wikipedia.org/wiki/MIPS_processor