



Introdução

- Linguagem de Máquina
 - Conjunto de Instruções
 - Variáveis em Assembly: Registradores
 - Adição e Subtração em Assembly
 - Acesso à Memória em Assembly
- Objetivos
 - Facilitar a construção do hardware e compiladores
 - Maximizar a performance.
 - Minimizar o custo.
- Instruções: MIPS (NEC, Nintendo, Silicon Graphics, Sony).



Projeto de Assembly: Conceitos Chaves

- Linguagem Assembly é essencialmente suportada diretamente em hardware, portanto ...
- Princípio 1: Simplicidade favorece Regularidade.
 - Ela é mantida bem simples!
 - Limite nos tipos de operandos
 - Limite no conjunto de operações que podem ser feitas no mínimo absoluto
 - Se uma operação pode ser decomposta em uma mais simples, não a inclua (a complexa)



Todo computador: ops. Aritméticas

- MIPS: `add a,b,c # $a \leftarrow b + c$`
 - nota: os nomes reais dos operadores não são a, b e c. Serão vistos em Breve.
- Instruções são mais rígidas que em lin. de alto nível
 - MIPS: sempre 3 operandos.
- Exemplo: somar variáveis b, c, d, e, colocando a soma em a:

Todo computador: ops. Aritméticas

- MIPS: `add a,b,c # a ← b + c`
 - nota: os nomes reais dos operadores não são a, b e c. Serão vistos em Breve.
- Instruções são mais rígidas que em lin. de alto nível
 - MIPS: sempre 3 operandos.
- Exemplo: somar variáveis b, c, d, e, colocando a soma em a:

<code>add a,b,c</code>	<code># a = b+c</code>
<code>add a,a,d</code>	<code># a = b+c+d</code>
<code>add a,a,e</code>	<code># a = b+c+d+e</code>

Todo computador: ops. Aritméticas

- MIPS: `add a,b,c # a ← b + c`
 - nota: os nomes reais dos operadores não são a, b e c. Serão vistos em Breve.
- Instruções são mais rígidas que em lin. de alto nível
 - MIPS: sempre 3 operandos.
- Exemplo: somar variáveis b, c, d, e, colocando a soma em a:

<code>add a,b,c</code>	<code># a = b+c</code>
<code>add a,a,d</code>	<code># a = b+c+d</code>
<code>add a,a,e</code>	<code># a = b+c+d+e</code>
- Símbolo # → Comentário (até o fim da linha).
- Exemplo: C → Assembly.

<code>a = b + c;</code>
<code>d = a - e;</code>
- Em MIPS:

Todo computador: ops. Aritméticas

- MIPS: `add a,b,c # a ← b + c`
 - nota: os nomes reais dos operadores não são a, b e c. Serão vistos em Breve.
- Instruções são mais rígidas que em lin. de alto nível
 - MIPS: sempre 3 operandos.
- Exemplo: somar variáveis b, c, d, e, colocando a soma em a:

<code>add a,b,c</code>	<code># a = b+c</code>
<code>add a,a,d</code>	<code># a = b+c+d</code>
<code>add a,a,e</code>	<code># a = b+c+d+e</code>
- Símbolo # → Comentário (até o fim da linha).
- Exemplo: C → Assembly.

<code>a = b + c;</code>
<code>d = a - e;</code>
- Em MIPS:

<code>add a,b,c</code>	<code># a=b+c</code>
<code>sub d,a,e</code>	<code># d=a-e</code>



Variáveis Assembly: Registradores (1/3)

- Diferente de LAN, assembly não pode usar variáveis.
 - Por que não? Manter o Hardware simples
- Operandos Assembly são registradores
 - Número limitado de localizações especiais construídas diretamente no hardware
 - Operações podem somente ser realizadas nestes!
- Benefício: Como registradores estão diretamente no hardware, eles são muito rápidos.



Variáveis Assembly: Registradores (2/3)

- Desvantagem: Como registradores estão em hardware, existe um número predeterminado deles.
 - Solução: código MIPS deve ser muito cuidadosamente produzido para usar eficientemente os registradores.
- 32 registradores no MIPS
 - Por que 32?
 - Princípio 2: Menor é mais rápido ($>$ no. reg $\rightarrow >$ ciclo clock)
- Cada registrador MIPS tem 32 bits de largura
 - Grupos de 32 bits chamados uma palavra (word) no MIPS



Variáveis Assembly: Registradores (3/3)

- Registradores são numerados de 0 a 31
- Cada registrador pode ser referenciado por número ou nome.
 - Por convenção, cada registrador tem um nome para facilitar a codificação - nomes: iniciam em “\$”
- Por agora:

\$16 - \$22	\$s0 - \$s7	(corresponde a variáveis C)
\$8 - \$15	\$t0 - \$t7	(corresponde a registradores temporários)
- Em geral, utilize nomes de registradores para tornar o código mais fácil de ler.



Comentários em Assembly

- Outro modo de tornar o seu código mais claro: comente!
- Hash (#) é utilizado para comentários MIPS
 - Qualquer coisa da marca hash (#) ao final da linha é um comentário e será ignorado.
- Nota: Diferente do C.
 - Comentários em C tem a forma /* comentário */, de modo que podem ocupar várias linhas.



Instruções Assembly

- Em linguagem assembly, cada declaração (chamada uma Instruction), executa exatamente uma de uma lista pequena de comandos simples
- Diferente de C (e da maioria das outras linguagem de alto nível), onde cada linha pode representar múltiplas operações.



Adição e Subtração (1/3)

- Sintaxe de Instruções:

1 2,3,4

onde:

1) operação por nome

2) operando recebendo o resultado ("destino")

3) 1º operando da operação ("fonte 1")

4) 2º operando da operação ("fonte 2")

- Sintaxe é rígida:

- 1 operador, 3 operandos

- Por quê? Manter o Hardware simples via regularidade



Adição e Subtração

- em C: $f = (g + h)$
- Adição em Assembly (MIPS)



Adição e Subtração

- em C: $f = (g + h)$
- Adição em Assembly (MIPS)

Parte 1 – Função do compilador

Associar as variáveis aos registradores

f → **\$S0**

g → **\$S1**

h → **\$S2**

A escolha se dá de acordo com os registradores livres.
Procuraremos associar variáveis aos registradores do tipo S.



Adição e Subtração

- em C: $f = (g + h)$
- Adição em Assembly (MIPS)

f -> \$S0

g -> \$S1

h -> \$S2

Parte 2 – construir o programa

inicio

add \$S0, \$S1, \$S2 # f=g+h

fim



Adição e Subtração (2/3)

- em C: $f = (g + h) - (i + j);$
- Adição em Assembly (MIPS)

Adição e Subtração (2/3)

- em C: $f = (g + h) - (i + j);$
- Adição em Assembly (MIPS)
 - `add $t0, $s1, $s2` # $t0 = s1 + s2 = (g + h)$
 - `add $t1, $s3, $s4` # $t1 = s3 + s4 = (i + j)$
 - Reg. Temporários: `$t0, $t1`
 - Variáveis `$s1, $s2, $s3, $s4` estão associados com as variáveis `g, h, i, j`
- Subtração em Assembly
 - `sub $s0, $t0, $t1` # $s0 = t0 - t1 = (g + h) - (i + j)$
 - Reg. Temporários: `$t0, $t1`
 - Variável `$s0` está associada com a variável `f`



Adição e Subtração (3/3)

- Como fazer a seguinte declaração C?

`a = b + c + d - e;`

Adição e Subtração (3/3)

- Como fazer a seguinte declaração C?

$$a = b + c + d - e;$$

- Quebre em múltiplas instruções

`add $T0, $S1, $S2 # t0 = b + c`

`add $T1, $T0, $S3 # t1 = t0 + b`

`sub $S0, $T1, $S4 # a = t1 - c`

- Nota: Uma linha de C pode resultar em várias linhas de MIPS.
- Note: Qualquer coisa após a marca hash em cada linha é ignorado (comentários)

Exercício:

Compilar o seguinte código para MIPS:

a = 0;

b = 0;

c = a + b;

Exercício:

Compilar o seguinte código para MIPS:

a = 0;

b = 0;

b = 8 * a ;

Exercício:

Compilar o seguinte código para MIPS:

a = 1;



Imediatos

- Imediatos são constantes numéricas.
- Eles aparecem freqüentemente em código, logo existem instruções especiais para eles.

- Somar Imediato:

`addi $s0, $s1, 10` #`$s0=$s1+10` (em MIPS)

`f = g + 10` (em C)

- onde registradores `$s0, $s1` estão associados com as variáveis `f, g`
- Sintaxe similar à instrução `add` exceto que o último argumento é um número ao invés de um registrador.

Exercício:

Compilar o seguinte código para MIPS:

a = 10;

b = -1;

a = 4*a + 1;

c = a + b;

Exercício:

Compilar o seguinte código para MIPS:

a = 0x10;

b = 0x1ABC;

c = a + b;

Registrador Zero

- Um imediato particular, o número zero (0), aparece muito freqüentemente em código.
- Então nós definimos o registrador zero (`$0` ou `$zero`) para sempre ter o valor 0.
- Isto é definido em hardware, de modo que uma instrução como:
`addi $0, $0, 5` $\# \$0 = \$0 + 5 \rightarrow \$0 = 0$ (reg. `$0` = 0 sempre)
não vai fazer nada.
- Use este registrador, ele é muito prático!



Operações Bitwise (1/2)

- Até agora:
 - aritmética **add** e **sub** e **addi**
- Todas estas instruções vêem o **conteúdo** de um registrador como uma **única quantidade** (tal como um inteiro com sinal ou sem sinal).
- **Nova Perspectiva:** Ver o **conteúdo** do registrador **como 32 bits** ao invés de como um número de 32 bits.



Operações Bitwise (2/2)

- Como os registradores são compostos de 32 bits, nós podemos querer **acessar bits individuais** (ou grupos de bits) ao invés de todo ele.
- Temos então duas novas classes de operações:
 - Operadores Lógicos
 - Instruções Shift



Operadores Lógicos (1/4)

- Dois operadores lógicos básicos:
 - **AND**: saída 1 somente se ambas as entradas são 1
 - **OR**: saída 1 se pelo menos uma entrada é 1
- Em geral, podemos defini-los para aceitar >2 entradas:
 - assembly MIPS: ambos aceitam exatamente **2 entradas e produzem 1 saída**.
 - Novamente, **sintaxe rígida**, hardware mais simples

Operadores Lógicos (2/4)

- Tabela Verdade: tabela padrão listando todas as possíveis combinações de entradas e saídas resultantes para cada um.
- Tabela Verdade para AND e OR

A	B	AND	OR
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1



Operadores Lógicos (3/4)

- Sintaxe da Instrução Lógica:

1 2,3,4

- onde

1) nome da operação

2) registrador que recebe o resultado

3) primeiro operando (registrador)

4) segundo operando (registrador) ou imediato (constante numérica).

Operadores Lógicos (4/4)

- Nomes das Instruções:
 - `and`, `or`: Ambas esperam o **terceiro argumento** ser um **registrador**.
`or $T0, $S1, $S2` # $t0 = s1 \parallel s2$
 - `andi`, `ori`: Ambas esperam o **terceiro argumento** ser um **imediato**.
`ori $T0, $S1, 3` # $t0 = s1 \parallel 3$
- Operadores Lógicos MIPS são todos **bitwise**:
 - o bit 0 da saída é produzido pelo respectivos bits 0's da entrada
 - o bit 1 pelos respectivos bits 1, etc.

Uso dos Operadores Lógicos (1/3)

- **anding** um bit com **0** produz **0** na saída
- **anding** um bit com **1** produz o **bit original**.
- Isto pode ser utilizado para criar uma **máscara**.

- Exemplo:

1011 0110 1010 0100 0011 1101 1001 1010 ← **\$T0**

0000 0000 0000 0000 0000 1111 1111 1111 ← **0xFFFF**

- O resultado de **anding** estes dois é:

0000 0000 0000 0000 0000 1101 1001 1010 ← **\$T0 && 0xFFFF**

Usos dos Operadores Lógicos (2/3)

- A segunda bitstring: chamada de **máscara**.
 - Função: **isolar** os 12 **bits** mais à direita da bitstring mascarando os outros (fazendo-os todos igual a 0s).
- Operador `and`:
 - setar em 0s certas partes de uma bitstring
 - deixar os outros como estão, intactos.
 - Em particular, se a primeira string de bits do exemplo acima estivesse em `$t0`, então a **instrução** a seguir iria mascará-la:

```
andi    $t0,$t0,0xFFF
```

Uso dos Operadores Lógicos (3/3)

- Similarmente:
 - `or` um bit com **1** produz **1** na saída
 - `or` um bit com **0** produz o **bit original**.
- Função: **forçar** certos **bits** de uma string **em 1s**.
 - Por exemplo, se `$t0` contém **0x12345678**, então após esta instrução:

```
ori    $t0, $t0, 0xFFFF
```

 - `$t0` contém **0x1234FFFF**
 - (i.e. Os 16 bits de ordem mais alta são intocados, enquanto que os 16 bits de ordem mais baixa são forçados a 1s).

Instruções Shift (1/4)

- Move (shift) todos os bits na palavra para a esquerda ou direita um certo número de bits.

- Exemplo: **shift right** por 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

- Exemplo: **shift left** por 8 bits

001 0010 0011 0100 0101 0110 0111 1000

0011 0100 0101 0110 0111 1000 0000 0000

Instruções Shift (2/4)

- Sintaxe das Instruções Shift:

1 2,3,4

- onde

1) nome da operação

2) registrador que receberá o valor

3) primeiro operando (registrador)

4) quantidade de deslocamento - shift amount
(constante ≤ 32)

Exemplo:

sll \$S1, \$S2, 8 # s1 = s2 << 8

Instruções Shift (3/4)

- Instruções shift MIPS :

1. `sll` (shift left logical): desloca para **esquerda** e completa os bits esvaziados com 0s.

■ Exemplo: **shift left** por 8 bits `sll $S1, $S2, 8` # $s1 = s2 \ll 8$

001 0010 0011 0100 0101 0110 0111 1000

0011 0100 0101 0110 0111 1000 0000 0000

Instruções Shift (3/4)

- Instruções shift MIPS :

1. `sll` (shift left logical): desloca para **esquerda** e completa os bits esvaziados com 0s.
2. `srl` (shift right logical): desloca para a **direita** e preenche os bits esvaziados com 0s.

- Exemplo: **shift right** por 8 bits `srl $S1,$S2,8` # $s1 = s2 \gg 8$

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110



Instruções Shift (3/4)

- Instruções shift MIPS :

1. `sll` (shift left logical): desloca para **esquerda** e completa os bits esvaziados com 0s.
2. `srl` (shift right logical): desloca para a **direita** e preenche os bits esvaziados com 0s.
3. `sra` (shift right arithmetic): desloca para a **direita** e preenche os bits esvaziados **estendendo o sinal**.

`sra $S1,$S2,8 # s1 = s2>>8`

Instruções Shift (4/4)

- Exemplo: **shift right arith** por 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

- Exemplo: **shift right arith** por 8 bits

1001 0010 0011 0100 0101 0110 0111 1000

1111 1111 1001 0010 0011 0100 0101 0110

Uso das Instruções Shift (1/5)

- **Isolar o byte 0** (8 bits mais à direita) de uma palavra em \$t0. Simplesmente use: xxxx xxxx xxxx xxxx xxxx xxxx **xxxx xxxx**

```
andi $t0, $t0, 0xFF
```

Uso das Instruções Shift (1/5)

- **Isolar o byte 0** (8 bits mais à direita) de uma palavra em \$t0. Simplesmente use: xxxx xxxx xxxx xxxx xxxx xxxx **xxxx xxxx**

```
andi $t0, $t0, 0xFF
```

- **Isolar o byte 1** (bit 15 ao bit 8) da palavra em \$t0.

```
xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
```

Uso das Instruções Shift (1/5)

- **Isolar o byte 0** (8 bits mais à direita) de uma palavra em \$t0. Simplesmente use: `XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX`

```
andi $t0, $t0, 0xFF
```

- **Isolar o byte 1** (bit 15 ao bit 8) da palavra em \$t0. Nós podemos usar: `XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX`

```
andi $t0, $t0, 0xFF00
```

mas então nós ainda **precisamos deslocar** para a direita por 8 bits ...

```
srl $t0, $t0, 8 XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
```

Uso das Instruções Shift (2/5)

- Ao invés nós poderíamos usar:

```
sll $t0,$t0,16
```

```
srl $t0,$t0,24
```

0001 0010 0011 0100 **0101 0110** 0111 1000

0101 0110 0111 1000 0000 0000 0000 0000

0000 0000 0000 0000 0000 0000 **0101 0110**

Uso das Instruções Shift (3/5)

- Em decimal:
 - Multiplicando por 10 é o mesmo que deslocar para a esquerda por 1:
 - $714_{10} \times 10_{10} = 7140_{10}$
 - $56_{10} \times 10_{10} = 560_{10}$
 - Multiplicando por 100 é o mesmo que deslocar para esquerda por 2:
 - $714_{10} \times 100_{10} = 71400_{10}$
 - $56_{10} \times 100_{10} = 5600_{10}$
 - Multiplicando por 10^n é o mesmo que deslocar para a esquerda por n

Uso das Instruções Shift (4/5)

- Em binário:
 - Multiplicar por 2 é o mesmo que deslocar para a esquerda por 1:
 - $11_2 \times 10_2 = 110_2$
 - $1010_2 \times 10_2 = 10100_2$
 - Multiplicar por 4 é o mesmo que deslocar para a esquerda por 2:
 - $11_2 \times 100_2 = 1100_2$
 - $1010_2 \times 100_2 = 101000_2$
 - Multiplicar por 2^n é o mesmo que deslocar para a esquerda por n

Uso das Instruções Shift (5/5)

- Como **deslocar** pode ser **mais rápido** que multiplicar, um bom compilador usualmente percebe quando o código C **multiplica por uma potência de 2** e compila como uma **instrução shift**:

`a *= 8; (em C)`

seria compilado como:

`sll $s0, $s0, 3 (em MIPS)`

- Da mesma forma, **desloque para a direita** para **dividir por potências de 2**
 - Lembre-se de usar `sra` (manter o sinal)



Coisas para se Lembrar

- Instruções **Lógicas e Shift**: operam em bits **individualmente**
- **Aritméticas**: operam em uma **palavra** toda.
- Use Instruções Lógicas e Shift para **isolar campos**, ou **mascarando** ou **deslocando** para um lado ou para outro.
- Novas Instruções:

`and, andi, or, ori`

`sll, srl, sra`



Instruções Shift

Exercício 1:

Passar para MIPS o seguinte código:

x = 3;

y = x * 4 ;

usando add

usando sll



Instruções Shift

Exercício 1:

Passar para MIPS o seguinte código:

```
x = 3;  
y = x * 4 ;
```

Exercício 1b:

Passar para MIPS o seguinte código:

```
x = 3;  
y = x * 1025 ;
```



Instruções Shift

Exercício 2:

Passar para MIPS o seguinte código:

$x = 3;$

$y = x / 4 ;$



Instruções Shift

Exercício 3:

Passar para MIPS o seguinte código:

```
x = 305419896;
```



Instruções Shift

Exercício 3:

Passar para MIPS o seguinte código:

$x = 305419896;$

Uma dica:

Se passarmos a ver como a máquina, 305410996 está na base 10, mas se convertermos para Hexa, temos 12345678₍₁₆₎



Instruções Shift

Exercício 4:

Passar para MIPS o seguinte código:

$x = -1;$

$y = x / 32 ;$



Instruções Shift

Exercício 5:

Considere que a primeira linha de um programa seja:

ori \$S0, \$zero, 0x01

Utilizando apenas:

- instruções de deslocamento,**
- instruções reg-reg lógicas (não usar instruções com imediatos),**
- a menor quantidade possível de instruções.**

Obter \$S0 = 0xFFFFFFFF



Operandos Assembly: Memória

- Variáveis C mapeiam em registradores; e como ficam as grandes estruturas de dados, como arrays/vetores?
- A memória contém tais estruturas.
- Mas as instruções aritméticas MIPS somente operam sobre registradores, nunca diretamente sobre a memória.



Operandos Assembly: Memória

- Instruções para transferência de dados transferem dados entre os registradores e a memória:
 - Memória para registrador → Load
 - Registrador para memória → Store



Transferência de Dados: Memória para Reg. (1/4)

- Para transferir uma palavra de dados, nós devemos especificar duas coisas:
 - Registrador: especifique este pelo número (0 - 31)
 - Endereço da memória: mais difícil
 - Pense a memória como um array único uni-dimensional, de modo que nós podemos endereçá-la simplesmente fornecendo um ponteiro para um endereço da memória.
 - Outras vezes, nós queremos ser capazes de deslocar a partir deste ponteiro.



Transferência de Dados: Memória para Reg (2/4)

- Para especificar um endereço de memória para copiar dele, especifique duas coisas:
 - Um registrador que contém um ponteiro para a memória.
 - Um deslocamento numérico (em bytes)
- O endereço de memória desejado é a soma destes dois valores.
- Exemplo: `8 ($t0)`
 - Especifica o endereço de memória apontado pelo valor em `$t0`, mais 8 bytes

Transferência de Dados: Memória para Reg (3/4)

- Sintaxe da instrução de carga (load):

1 2,3(4) **Exemplo : lw \$t0, 12 (\$s0)**

- onde

1) nome da operação (instrução) **lw**

2) registrador que receberá o valor **\$t0**

3) deslocamento numérico em bytes. **12**

4) registrador contendo o ponteiro para a memória **\$s0**

- Nome da Instrução:

- **lw** (significa Load Word, logo 32 bits ou uma palavra é carregada por vez)

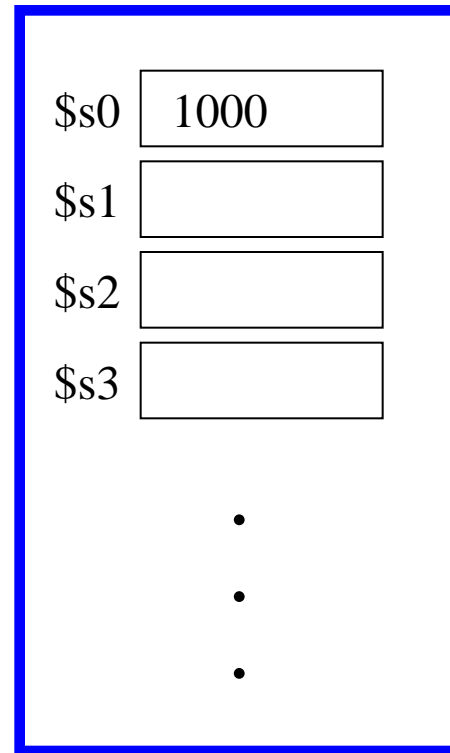


Transferência de Dados: Memória para Reg.(4/4)

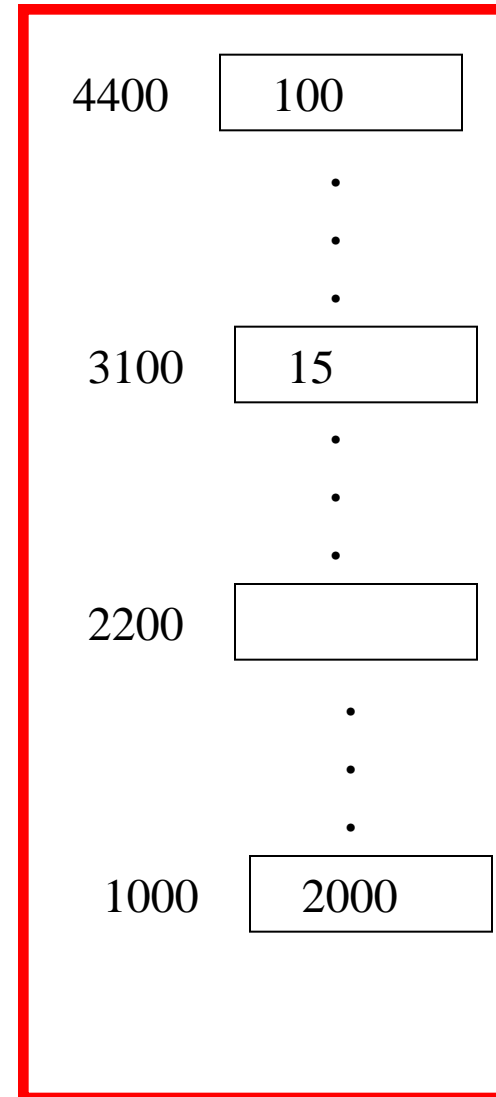
- Exemplo: `lw $t0, 12($s0)`
 - Esta instrução pegará o ponteiro em `$s0`, soma 12 bytes a ele, e então carrega o valor da memória apontado por esta soma calculada no registrador `$t0`
- Notas:
 - `$s0` é chamado **registrador base**
 - 12 é chamado **deslocamento (offset)**
 - Deslocamento é geralmente utilizado no acesso de elementos de array ou estruturas: reg base aponta para o início do array ou estrutura.

Exemplo lw:

Registradores



Memória



lw __, __ (__) # s__ = MEM [____ + s__]

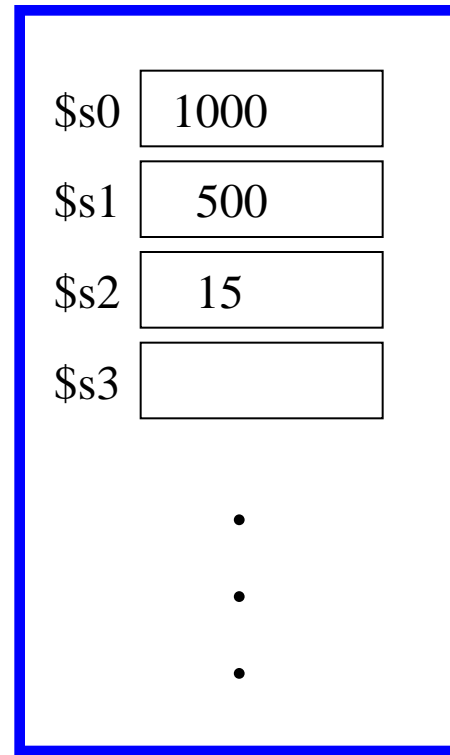


Transferência de Dados: Reg para Memória

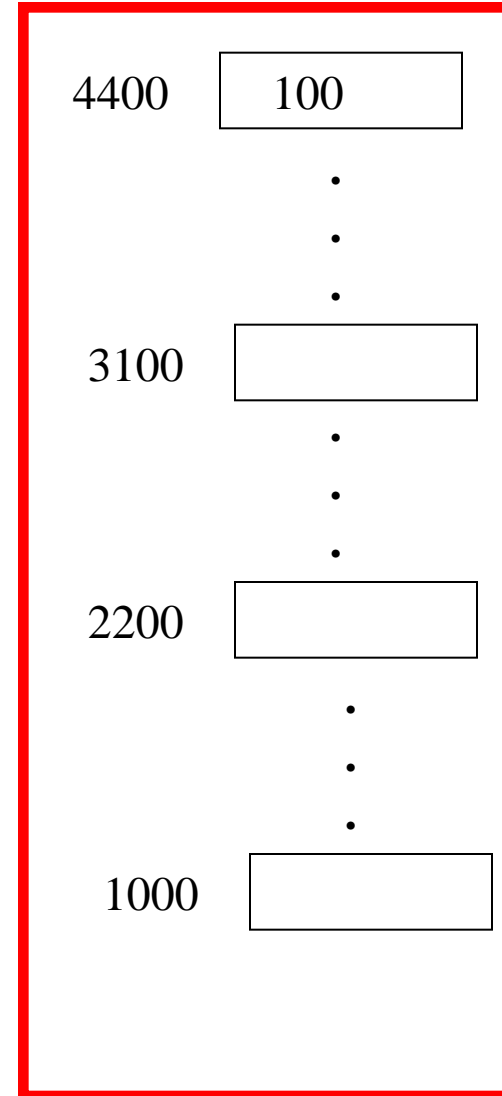
- Também queremos armazenar um valor do registrador na memória.
- Sintaxe da instrução store é idêntica à da instrução load.
- Nome da Instrução:
 - `sw` (significa Store Word, logo 32 bits ou uma palavra será carregada por vez)
- Exemplo: `sw $t0, 12($s0)`
 - Esta instrução tomará o ponteiro em `$s0`, somará 12 bytes a ele, e então armazenará o valor do registrador `$t0` no endereço de memória apontado pela soma calculada.

Exemplo sw:

Registradores



Memória



SW __, __ (__) # **MEM** [__ + s__] = s__

Memória e vetores

Compilar :

h = A[2];

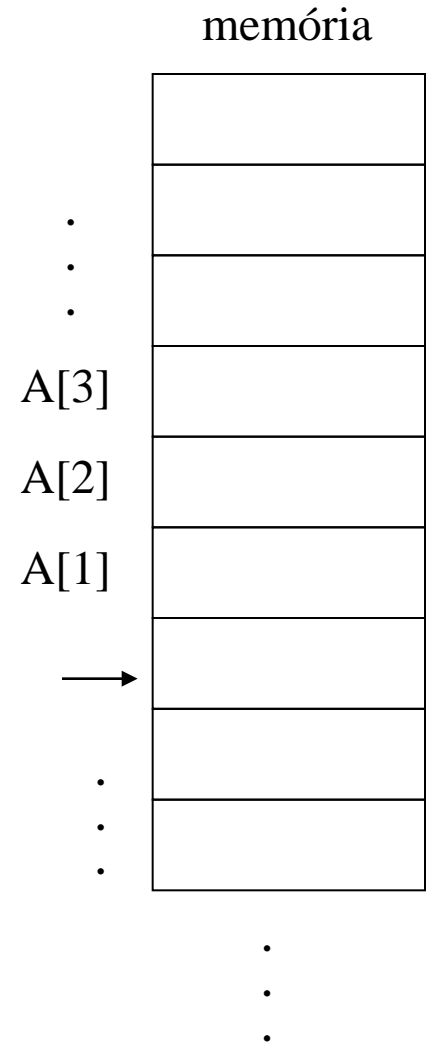
1) Vamos mapear **h** em **\$s0**

2) Para o vetor **A []**, devemos
Inicialmente mapear o endereço base
ou **A[0]** em um registrador, seja **\$s1**.

O programa fica:

lw \$s0, 2 (\$s1) # h = MEM [s1 + 2]

Endereço base ou A[0] →



Memória e vetores

Compilar :

A [12] = h + A [8];

Endereçamento: Byte vs. palavra (1/2)

- Cada palavra na memória tem um endereço, similar a um índice em um array.
- Primeiros computadores numeravam palavras como elementos de um array C:

■ `Memory[0], Memory[1], Memory[2], ...`

Chamado o "endereço" de uma palavra

- Computadores precisam acessar bytes (8-bits) bem como palavras (4 bytes/palavra)
 - Máquinas de hoje endereçam memória como bytes, portanto endereços de palavra diferem por 4
- `Memory[0], Memory[4], Memory[8], ...`

No Mips os endereços são incrementados de 4 em 4 pois indicamos o primeiro byte da palavra e podemos acessar cada byte individualmente.

0x 8	8	9	10	11
0x 4	4	5	6	7
0x 0	0	1	2	3

Memória e vetores com a correção

Compilar :

h = A[2];

1) Vamos mapear **h** em **\$s0**

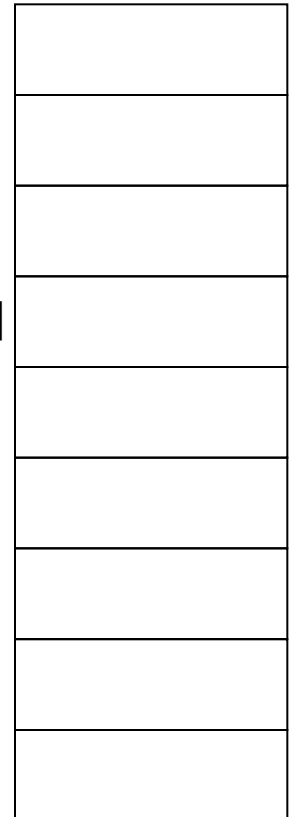
2) Para o vetor **A []**, devemos
Inicialmente mapear o endereço base
ou **A[0]** em um registrador, seja **\$s1**.

O programa fica:

lw \$s0, 8 (\$s1) # h = MEM [s1 + 8]

memória

Endereço base ou A[0] →



Compilação com Memória

- Qual o offset em 1_w para selecionar $A[8]$ em C?
 - $4 \times 8 = 32$ para selecionar $A[8]$: byte vs. palavra
- Compile manualmente usando registradores:
 $g = h + A[8];$
 - $g: \$s1, h: \$s2, \$s3$: endereço base de A

Compilação com Memória

- Qual o offset em $1w$ para selecionar $A[8]$ em C?
 - $4 \times 8 = 32$ para selecionar $A[8]$: byte vs. palavra
- Compile manualmente usando registradores:
 $g = h + A[8];$
 - $g: \$s1, h: \$s2, \$s3$: endereço base de A
- 1º transfere da memória para registrador:
 - $lw \quad \$t0, 32(\$s3) \quad \# \quad \$t0 = A[8]$
 - Some 32 a $\$s3$ para selecionar $A[8]$, põe em $\$t0$
- A seguir, some-o a h e coloque em g
 $add \quad \$s1, \$s2, \$t0 \quad \# \quad \$s1 = h + A[8] = \$s2 + \$t0$

Exemplo: Compilar ...

- Compile manualmente usando registradores:

$A[12] = h + A[8];$

- $h: \$s2, \$s3$: endereço base de A

Exemplo: Compilar ...

- Compile manualmente usando registradores:

$A[12] = h + A[8];$

- $h: \$s2, \$s3$: endereço base de A

- 1º transfere da memória para registrador \$t0:

`lw $t0, 32($s3) # $t0 = A[8]`

- 2º some-o a h e coloque em \$t0

`add $t0, $s2, $t0 # $t0 = h + A[8]`

- 3º transfere do reg. \$t0 para a memória :

`sw $t0, 48($s3) # A[12] = $t0`



Exemplo: Compilar ...

1) **$h = k + A[i];$**



Exemplo: Compilar ...

2) **$A[j] = h + A[i];$**



Exemplo: Compilar ...

3) **$h = A[i];$**
 $A[i] = A[i + 1];$
 $A[i + 1] = h;$

Ponteiros vs. Valores

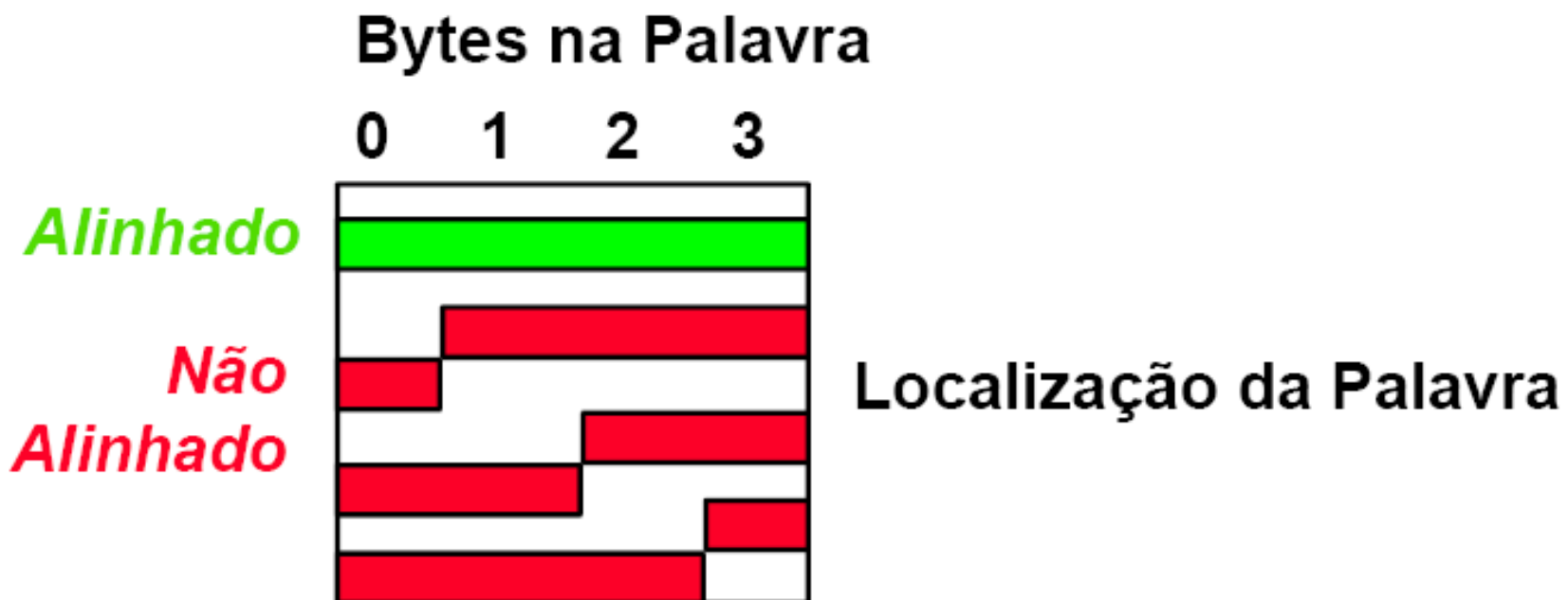
- **Conceito Chave:** Um registrador pode conter qualquer valor de 32 bits. Este valor pode ser um `int` (signed), um `unsigned int`, um ponteiro (endereço de memória), etc.
- Se você escreve `lw $t2, 0($t0)` então é melhor que `$t0` contenha um ponteiro.
- E se você escrever `add $t2, $t1, $t0` então `$t0` e `$t1` devem conter o quê?

Notas a cerca da Memória

- Falha: Esquecer que endereços seqüenciais de palavras em máquinas com endereçamento de byte não diferem por 1.
 - Muitos erros são cometidos por programadores de linguagem assembly por assumirem que o endereço da próxima palavra pode ser achado incrementando-se o endereço em um registrador por 1 ao invés do tamanho da palavra em bytes.
 - Logo, lembre-se que tanto para `lw` e `sw`, a soma do endereço base e o offset deve ser um múltiplo de 4 (para ser **alinhado em palavra**)

Mais Notas acerca da Memória: Alinhamento

- MIPS requer que todas as palavras comecem em endereços que são múltiplos de 4 bytes.



- Chamado Alinhamento: objetos devem cair em endereços que são múltiplos do seu tamanho.



"Em conclusão ..." (1/2)

- Em linguagem Assembly MIPS:
 - Registradores substituem variáveis C
 - Uma instrução (operação simples) por linha
 - Mais Simples é Melhor
 - Menor é Mais Rápido
- Memória é endereçada por **byte**, mas `lw` e `sw` acessam uma **palavra** de cada vez.
- Um ponteiro (usado por `lw` e `sw`) é simplesmente um endereço de memória, logo nós podemos somar a ele ou subtrair dele (usando offset).



"E em conclusão..." (2/2)

- Novas Instruções:
 - `add, addi,`
 - `sub`
 - `lw, sw`
- Novos registradores:
 - Variáveis C: `$s0 - $s7`
 - Variáveis Temporárias: `$t0 - $t9`
 - Zero: `$zero`

Linguagem de Máquina

- Instruções, como registradores e palavras, são de 32 bits
 - Exemplo: `add $t0, $s1, $s2`
 - registradores tem números `t0=reg.8`, `$s1=reg.17`, `$s2=reg.18`
- Formato de Instrução de soma com registradores (R-tipo):

0	17	18	8	0	32
000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Primeiro campo: tipo de operação (soma).
- Último campo: modo da operação (soma com 2 registradores).
- Segundo campo: primeira fonte (17=\$s1).
- Terceiro campo: segunda fonte (18=\$s2).
- Quarto campo: registrador de destino (8=\$t0).

Linguagem de Máquina

- Novo princípio: Bom projeto exige um bom compromisso
 - Vários tipos de instruções (tipo-R, tipo-I)
 - Múltiplos formatos: complicam o hardware.
 - Manter os formatos similares (3 primeiros campos iguais).
- Considere as instruções load-word e store-word,
 - I-tipo para instruções de transferência de dados (lw,sw)
- Exemplo: `lw $t0, 32($s2)`

35	18	8	32
op	rs	rt	16 bit - offset

- Registrador de base: rs `$s2`.
- Registrador de fonte ou origem: rt `$t0`.

Codificação das instruções vistas até agora

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34	n.a.
lw (load word)	I	35	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43	reg	reg	n.a.	n.a.	n.a.	address

- Notar: add e sub:
 - Mesmo opcode: 0.
 - Diferente função: 32 e 34.



Panorama

- Decisões C/Assembly : `if`, `if-else`
- Laços (loops) C/Assembly: `while`, `do while`,
`for`
- Desigualdades
- Declaração Switch C



Até agora...

- Todas as instruções nos permitiram manipular dados.
- Assim, construímos uma calculadora.
- Para construirmos um computador, precisamos da habilidade de tomar decisões...

Decisões em C: Declaração `if`

- 2 tipos de declaração `if` em C

- `if (condição) cláusula`
- `if (condição) cláusula1 else cláusula2`

- Rearranje o 2º `if` do seguinte modo:

```
    if (condição) goto L1;  
    cláusula2;  
go to L2;  
L1:  cláusula1;  
L2:
```

- Não é tão elegante como `if-else`, mas com o mesmo significado



Instruções de Decisão MIPS

■ Desvios condicionais

- `beq register1, register2, L1`
- `beq` é "bbranch if (registers are) equal"

O mesmo que (usando C): `if (register1==register2)`
`goto L1`

Exemplo:

```
beq $s1, $s2, fim      # se o conteúdo de s1 for igual ao de s2,  
                        # vá para a linha marcada como fim
```

...

fim: ...



Instruções de Decisão MIPS

■ Desvios condicionais

- `bne register1, register2, L1`
- `bne` é "bbranch if (registers are) not equal••

O mesmo que (usando C): `if (register1!=register2)`
`goto L1`

Exemplo:

```
bne $s1, $s2, fim      # se o conteúdo de s1 for diferente de s2,  
                        # vá para a linha marcada como fim
```

...

fim: ...

Instrução Goto MIPS

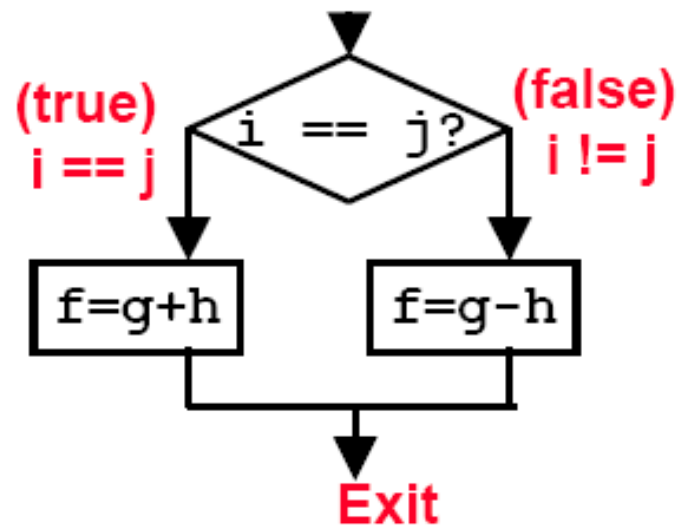
- Além dos desvios condicionais, MIPS tem um **desvio incondicional**:
 - `J label`
- Chamada instrução Pulo (Jump): pule (ou desvie) diretamente para a marca dada sem precisar satisfazer qualquer condição
- Mesmo significado (usando C):
 - `goto label`
- Tecnicamente, é o mesmo que:
 - `beq $0, $0, label`
 - já que sempre vai satisfazer a condição.

Compilando `if` C em MIPS (1/2)

- Compile manualmente

`if (i == j) f = g+h;`

`else f = g-h;`



- Use este mapeamento:

`f: $s0, g: $s1, h: $s2, i: $s3, j: $s4`

Compilando `if` C em MIPS (2/2)

- Código MIPS final compilado:

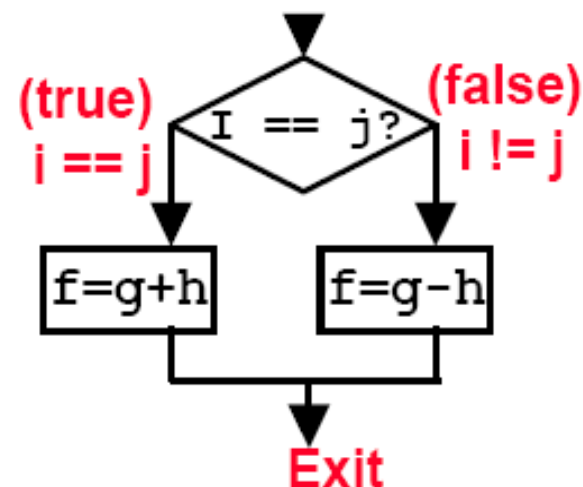
```
beq $s3,$s4,True    # branch  $i==j$ 
```

```
sub $s0,$s1,$s2     # (false)  $f=g-h$ 
```

```
j    Fim            # go to Fim
```

```
True: add $s0,$s1,$s2 # (true)  $f=g+h$ 
```

```
Fim:
```

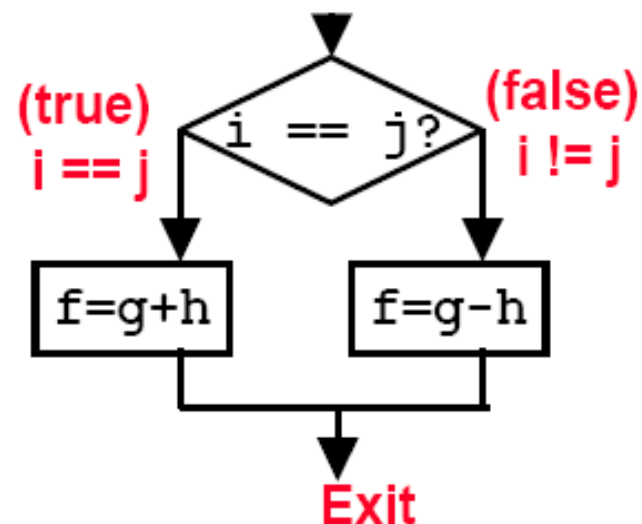


- Nota:

- Compilador automaticamente cria labels para tratar decisões (desvios) apropriadamente. Geralmente não são encontrados no código da Linguagem de Alto Nível

Compilando `if` C em MIPS (2/2)

- Código final MIPS compilado
(*preencha o espaço em branco*):





Laços (loops) em C/Assembly (1/3)

- Laço (loop) simples em C

```
j = 0;  
i = 10;  
do  
  {  
    j = j + 1;  
  }  
while ( j != i );
```

Laços (loops) em C/Assembly (1/3)

- Laço (loop) simples em C

```
j = 0;  
i = 10;  
do  
{  
    j = j + 1;  
}  
while ( j != i );
```

Reescrevendo:

```
j = 0;  
i = 10;  
loop:  
    j = j + 1;  
    if ( j != i ) goto loop;
```

Compilar:

```
i - $s0  
j - $s1
```


Laços (loops) em C/Assembly (1/3)

- Laço (loop) simples em C

```
do
{
    g = g + A[i];
    i = i + j;
} while (i != h);
```

Laços (loops) em C/Assembly (1/3)

- Laço (loop) simples em C

```
do
{
    g = g + A[i];
    i = i + j;
} while (i != h);
```

- Reescreva isto como:

```
Loop:  g = g + A[i];
        i = i + j;
        if (i != h) goto Loop;
```

- Use este mapeamento:

- g: \$s1, h: \$s2, i: \$s3, j: \$s4, base de A:\$s5



Loops em C/Assembly (2/3)

- Código MIPS final compilado:
(preencha o espaço em branco):

Laços (loops) em C/Assembly (2/3)

- Código MIPS final compilado:

Loop:

```
add $t1,$s3,$s3    # $t1 = 2*i
add $t1,$t1,$t1     # $t1 = 4*i
add $t1,$t1,$s5     # $t1=end(A+4*i)
lw  $t1,0($t1)      # $t1=A[i]
add $s1,$s1,$t1     # g=g+A[i]
add $s3,$s3,$s4     # i=i+j
bne $s3,$s2,Loop    # goto Loop if i!=h
```



Laços (loops) em C/Assembly (3/3)

- Existem três tipos de laços em C:
 - `while`
 - `do • •while`
 - `for`
- Cada um pode ser rescrito como um dos outros dois, de modo que o método utilizado no exemplo anterior, pode ser aplicado a laços `while` e `for` igualmente.
- **Conceito Chave:** Apesar de haver muitas maneiras de se escrever um loop em MIPS, desvio condicional é a chave para se tomar decisões.

Laço com while em C/Assembly (1/2)

- Laço (loop) simples em C

```
while (save[i] == k)
    i = i + j;
```

- Reescreva isto como:

```
Loop: if (save[i] != k) goto Exit;
      i = i + j;
      goto Loop;
Exit:
```

- Use este mapeamento:

- i: \$s3, j: \$s4, k: \$s5, base de save :\$s6

Laços (loops) em C/Assembly (2/2)

- Código MIPS final compilado:

Loop:

```
add $t1,$s3,$s3    # $t1 = 2*i
add $t1,$t1,$t1    # $t1 = 4*i
add $t1,$t1,$s6    # $t1=end(save+4*i)
lw  $t1,0($t1)     # $t1=save[i]
bne $t1,$s5,Exit   # goto Exit if save[i]!=k
add $s3,$s3,$s4    # i=i+j
j   Loop           # goto Loop
```

Exit:

Desigualdades em MIPS (1/5)

- Até agora, nós testamos apenas igualdades (`==` e `!=`).
- Programas gerais precisam testar `>` e `<` também.
- Criar uma Instrução de Desigualdade em MIPS:
 - "Set on Less Than"
 - Sintaxe: `slt reg1, reg2, reg3`
 - Significado:

```
if (reg2 < reg3) reg1 = 1;
else reg1 = 0;
```
 - Em computadores, "set" significa "set to 1", "reset" significa "set to 0".



Desigualdades em MIPS (2/5)

- Como nós utilizamos isto?
- Compile manualmente:
`if (g < h) goto Less;`
- Use este mapeamento:
`g: $s0, h: $s1`

Desigualdades em MIPS (3/5)

- Código final MIPS compilado:

```
slt $t0,$s0,$s1 # $t0 = 1 if g<h
bne $t0,$0,Less # goto Less if $t0!=0
Less:           # (if (g<h))
```

- Desvie se $\$t0 \neq 0$ • • $(g < h)$
 - Registrador \$0 sempre contém o valor 0, assim `bne` e `beq` freqüentemente utilizam-no para comparação após uma instrução `slt`.

Desigualdades em MIPS (4/5)

- Agora, nós podemos implementar $<$, mas como implementamos $>$, \leq e \geq ?
- Poderíamos adicionar mais 3 instruções mas:
 - Meta MIPS: Mais simples é Melhor
- Nós podemos implementar \leq em um ou mais instruções utilizando apenas `slt` e os desvios?
- $E > ?$
- $E \geq ?$
- 4 combinações de `slt` e `beq/bne`

Desigualdades em MIPS (5/5)

- 4 combinações de slt e beq/bne:

`slt $t0,$s0,$s1` *# \$t0 = 1 if g<h*

`bne $t0,$0,Less` *# if(g<h) goto Less*

Desigualdades em MIPS (5/5)

- 4 combinações de slt e beq/bne:

`slt $t0,$s0,$s1` *# \$t0 = 1 if g<h*

`bne $t0,$0,Less` *# if(g<h) goto Less*

`slt $t0,$s0,$s1` *# \$t0 = 1 if g<h*

`beq $t0,$0,Gteq` *# if(g>=**h**) goto **Gteq***

Desigualdades em MIPS (5/5)

- 4 combinações de slt e beq/bne:

```
slt $t0,$s1,$s0    # $t0 = 1 if g>h
```

```
bne $t0,$0,Greater # if(g>h) goto Greater
```

Desigualdades em MIPS (5/5)

- 4 combinações de slt e beq/bne:

```
slt $t0,$s1,$s0      # $t0 = 1 if g>h  
bne $t0,$0,Greater    # if(g>h) goto Greater
```

```
slt $t0,$s1,$s0      # $t0 = 1 if g>h  
beq $t0,$0,Lteq     # if(g<=h) goto Lteq
```

Desigualdades em MIPS (5/5)

- 4 combinações de slt e beq/bne:

slt \$t0,\$s0,\$s1 # \$t0 = 1 if $g < h$

bne \$t0,\$0,Less # if ($g < h$) goto Less

slt \$t0,\$s1,\$s0 # \$t0 = 1 if $g > h$

bne \$t0,\$0,Greater # if ($g > h$) goto Greater

slt \$t0,\$s0,\$s1 # \$t0 = 1 if $g < h$

beq \$t0,\$0,Gteq # if ($g \geq h$) goto Gteq

slt \$t0,\$s1,\$s0 # \$t0 = 1 if $g > h$

beq \$t0,\$0,Lteq # if ($g \leq h$) goto Lteq

Imediatos em Desigualdades

- Existe também uma versão com imediatos de `slt` para testar contra constantes: `slti`
 - Útil em laços (loops) `for`
 - `if (g >= 1) goto Loop`

C

M
I
P
S



Comando Switch/Case

- Novo instrumento: jr (jump register):
 - Salto incondicional.
 - Pula para o endereço especificado pelo registrador.
 - Geralmente é usada juntamente com uma tabela.
- Para casa: estudar a instrução Switch/Case.