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 ← 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:

```
add a,b,c        # a → b+c
add a,a,d        # a → b+c+d
add a,a,e        # a → b+c+d+e
```

- Símbolo # → Comentário (até o fim da linha).
- Exemplo: C → Assembly.

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

- Em MIPS:

```
add a,b,c        # a=b+c
sub d,a,e        # d=a-e
```

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 → > 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;
```

- 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)

Panorama

- Instruções Lógicas
- Shifts - Deslocamentos

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.

Imediatos

Exercício:
Compilar o seguinte código para MIPS:

```
a = 10;
b = -1;
a = a + 1;
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` → `$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 || s2**
  - `andi`, `ori`: Ambas esperam o **terceiro argumento** ser um **imediato**. **ori $T0, $S1, 3 # t0 = s1 || 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)

- **and**ing um bit com **0** produz **0** na saída
- **and**ing 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 ← **0xFFF**

  - O resultado de `and`ing estes dois é:

0000 0000 0000 0000 0000 1101 1001 1010 ← **$T0 && 0xFFF**

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`ing um bit com **1** produz **1** na saída
  - `or`ing 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 << 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>>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
```

- **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 4:
Passar para MIPS o seguinte código:

```
x = -1;
y = x / 32 ;
```

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**

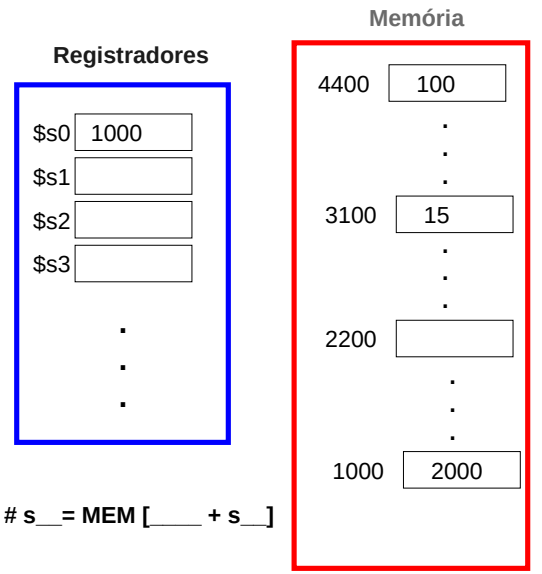
  3) deslocamento numérico em bytes. **12**

  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:

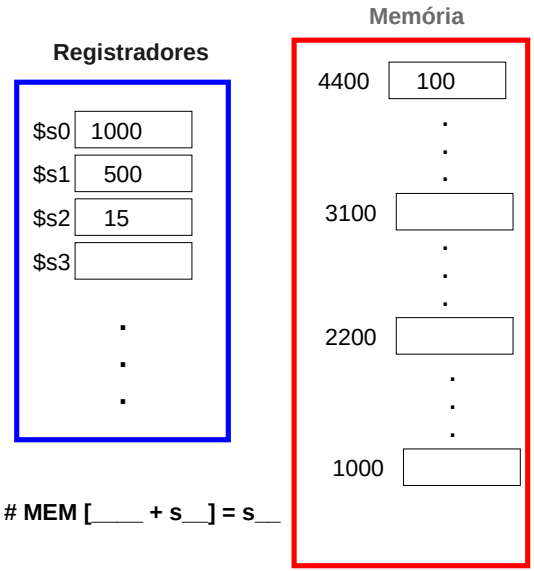


lw ___,____ (__) # s__= MEM [____ + s__]

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:



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

Memória e vetores

Compilar :

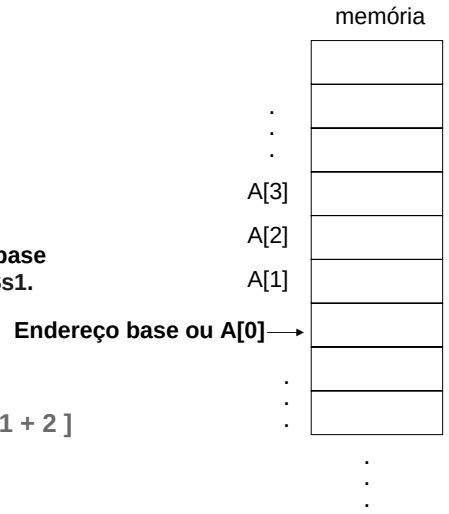
h = A[2];

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]**



Memória e vetores

Compilar :

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

Endereçamento: Byte vs. palavra (1/2)

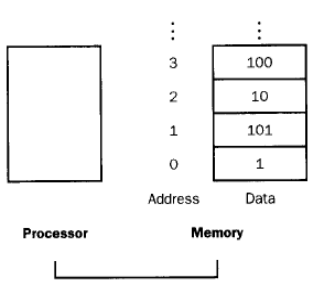
- Cada palavra na memória tem um endereço, similar a um índice em um array.
- Primeiros computadores numeravam palavras como elementos de um array C:
  - `Memory[0], Memory[1], Memory[2], . . .`

Chamado o "endereço" de uma palavra

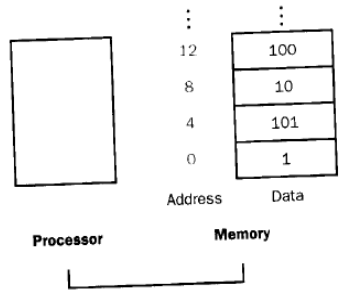
- Computadores precisam acessar bytes (8-bits) bem como palavras (4 bytes/palavra)
- Máquinas de hoje endereçam memória como bytes, portanto endereços de palavra diferem por 4
  - `Memory[0], Memory[4], Memory[8], . . .`

Endereçamento: Byte vs. palavra (2/2)

Endereçamento Byte



Endereçamento Palavra



Memória e vetores com a correção

Compilar :

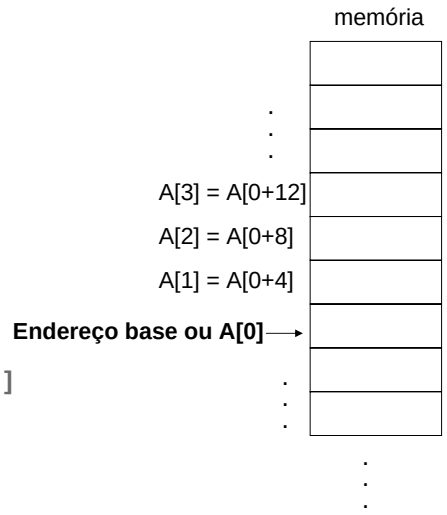
`h = A[2];`

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]



Compilação com Memória

- Qual o offset em `lw` para selecionar `A[8]` em C?
  - 4x8=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 `lw` para selecionar `A[8]` em C?
  - 4x8=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 $t0,32($s3)  # $t0 = A[8]`
  - Some 32 a `$s3` para selecionar `A[8]`, põe em `$t0`
- A seguir, some-o a h e coloque em g

```
add $s1,$s2,$t0      # $s1 = h+A[8]=$s2+$t0
```

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 ...

- 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

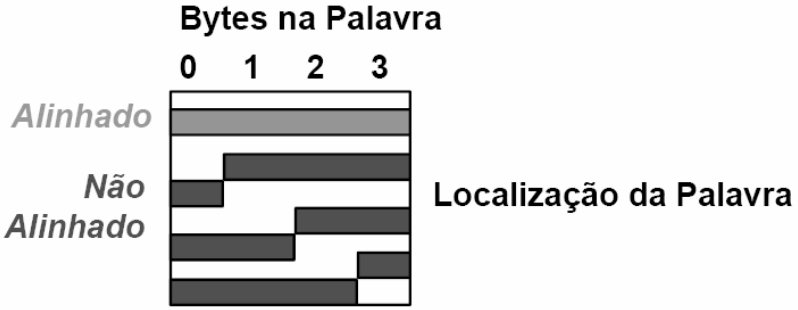
- Conceito Chave: Um registrador pode conter qualquer valor de 32 bits. Este valor pode ser um `int` (signed), um `unsigned int`, um ponteiro (endereço de memória), etc.
- Se você escreve `lw $t2,0($t0)` então é melhor que `$t0` contenha um ponteiro.
- E se você escrever `add $t2,$t1,$t0` então `$t0`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

Papel dos Registradores vs. Memória

- E se temos mais variáveis do que registradores?
  - Compilador tenta manter as variáveis mais freqüentemente utilizadas nos registradores.
  - Escrevendo as menos comuns na memória: spilling
- Por que não manter todas as variáveis na memória?
  - Menor é mais rápido: registradores são mais rápidos que a memória
  - Registradores são mais versáteis:
    - Instruções aritméticas MIPS pode ler 2, operar sobre eles e escrever 1 por instrução
    - Transferência de dados MIPS somente lê ou grava 1 operando por instrução, sem nenhuma operação.

"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.

Exemplo

- Tradução de C para assembly e linguagem de máquina.
  - C: A[300] = h + A[300]
  - Assembly:

```
lw   $t0,1200($t1)    # $t1=end. base=A
add  $t0,$s2, $t0     # $t0= A[300]=conteúdo(1200+$t1)
sw   $t0,1200($t1)    #          # $t0=$t0+$s2=A[300]+h
                      # A[300]=$t0
```

  - Linguagem de máquina:

"simplicidade favorece regularidade"

op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

1 bit de diferença

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Resumo da linguagem até agora

MIPS operands

Name	Example	Comments
32 registers	$s0, $s1, ..., $s7 $t0, $t1, ..., $t7	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers $s0–$s7 map to 16–23 and $t0–$t7 map to 8–15.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions in MIPS. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers.

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
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Data from register to memory

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s4
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

Revisão (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 por vez
- Um ponteiro (usado por `lw` e `sw`) é simplesmente um endereço de memória, de modo que podemos adicionar a ele ou subtrair dele (utilizando offset).

Revisão (2/2)

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

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, construimos 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 "branch 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 "branch 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:    ...
```

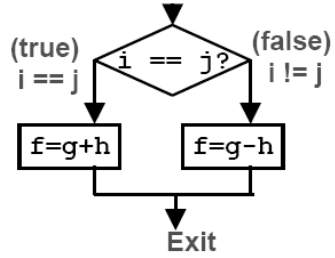
Instrução Goto MIPS

- 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;
```



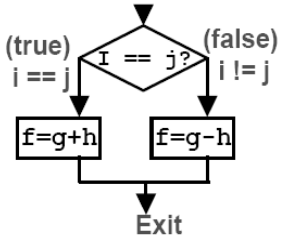
- Use este mapeamento:

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

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

- Código MIPS final compilado:

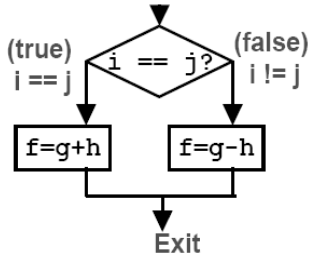
```
        beq  $s3,$s4,True     # branch i==j
        sub  $s0,$s1,$s2      # (false) f=g-h
        j    Fim              # go to Fim
True:   add  $s0,$s1,$s2      # (true) f=g+h
Fim:
```



- 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` != 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 >, <= e >=?
- Poderíamos adicionar mais 3 instruções mas:
  - Meta MIPS: Mais simples é Melhor
- Nós podemos implementar <= em um ou mais instruções utilizando apenas `slt` e os desvios?
- E >?
- E >=?
- 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
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>=h) goto Gteq
slt $t0,$s1,$s0     # $t0 = 1 if g>h
beq $t0,$0,Lteq     # if(g<=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

MIPS

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.

Procedimentos

- Estruturação de programas.
- Mais fácil de entender.
- Reutilização.
- Dividir em partes menores.

Panorama

- Funções em C
- Instruções MIPS para Funções
- A pilha (Stack)
- Convenções de Registradores
- Exemplo

Funções em C

```
main() {
 int i,j,k,m;
 i = mult(j,k); ... ;
 m = mult(i,i); ...
}
/* really dumb mult function */
int mult (int mcand, int mlier){
 int product;
 product = 0;
 while (mlier > 0)  {
  product = product + mcand;
  mlier = mlier -1; }
 return product;
 }
```

De quais informações o compilador/programador deve manter registro?

Quais instruções podem realizar isto?

Contabilidade de Chamada de Função

- Registradores tem papel fundamental para manter registro de informações nas chamadas de funções.
- Convenção de Registradores:
  - Endereço de retorno `$ra`
  - Argumentos `$a0, $a1, $a2, $a3`
  - Valor de Retorno `$v0, $v1`
  - Variáveis locais `$s0, $s1, • •, $s7`
- A pilha também é utilizada.

Instruções de Suporte para Funções (1/4)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

MIPS

```
endereço                                  # programa principal
1000      add  $a0,$s0,$zero    # x = a
1004      add  $a1,$s1,$zero    # y = b
1008      addi $ra,$zero,1016   #$ra=1016
1012      j    sum              #jump to sum
1016 ...
2000 sum: add $v0,$a0,$a1       # função sum: x+y
2004      jr   $ra              # new instruction
```

Instruções de Suporte para Funções (2/4)

- Instrução única para **pular e salvar o endereço de retorno**: jump and link (**`jal`**)
- Antes:

```
1008 addi $ra,$zero,1016 #$ra=1016
1012 j sum               #go to sum
```

- Depois:

```
1008 jal sum  # $ra=1016,go to sum
```

- Por que ter uma `jal`?
  - Torne o **caso comum rápido**: funções são muito comuns.

Instruções de Suporte para Funções (3/4)

- Sintaxe de **`jal`** (jump and link) é a mesma de `j` (jump):
  - `jal label`
- `jal` deveria na verdade ser chamada `laj` de link and jump:
  - Passo 1 (link): **Salva o endereço da próxima instrução** em $ra (Por que a próxima instrução? Por que não a corrente?)
  - Passo 2 (jump): **Pule para o label** dado

Instruções de Suporte para Funções (4/4)

- Sintaxe de **`jr`** (jump register):
  - `jr  register`
- Ao invés de prover um label para pular para, a instrução `jr` provê um **registrador** que **contém** um **endereço** para onde pular.
- Útil somente se nós sabemos o **endereço exato** para onde pular: raramente aplicável.
- Muito útil para **chamadas de funções**:
  - **`jal` guarda** o **endereço de retorno** no registrador ($ra) (**chamada de uma função**)
  - **`jr` pula** de volta **para** aquele **endereço (retorno da função)**

Funções Aninhadas (1/2)

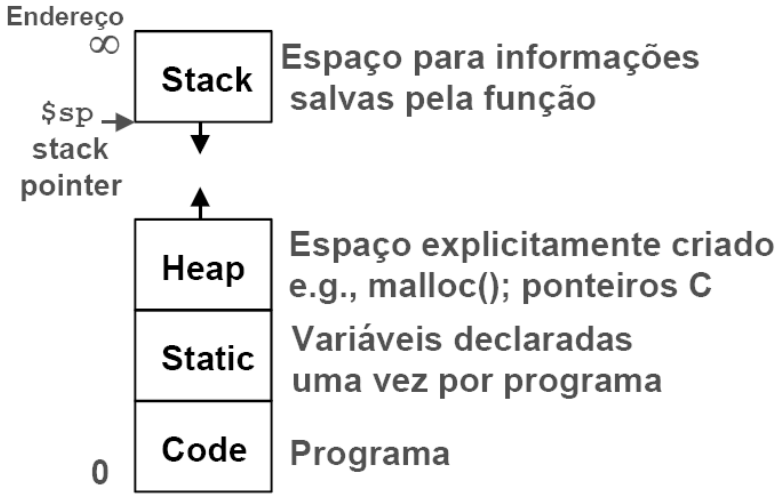
```
int sumSquare(int x, int y) {
     return mult(x,x)+ y;
  }
```

- Alguma coisa chamada **`sumSquare`**, que **chama `mult`**.
- Há um valor em **$ra** que `sumSquare` quer pular de volta, mas **será sobrescrito** pela chamada a `mult`.
- Precisa **salvar o endereço de retorno** de `sumSquare` antes de chamar `mult`.

Funções Aninhadas (2/2)

- Em geral, pode ser necessário **salvar** algum outro **registrador** além de $ra.
- Quando um programa C está rodando, existem 3 **importantes áreas de memória** que são alocadas:
  - **Static** (alocação estática): Variáveis declaradas uma vez por programa, deixam de existir somente quando a execução termina.
  - **Heap** (alocação dinâmica): Variáveis declaradas dinamicamente.
  - **Stack** (pilha): Espaço a ser utilizado pela função durante sua execução; é aqui que podemos **salvar** os valores dos **registradores**.

Alocação de Memória em C



Usando a Pilha (1/2)

- Nós temos um registrador **$sp** que **sempre aponta** para o **último espaço utilizado na pilha**.
- Para utilizar a pilha:
  - 1º **decrementamos** o ponteiro **$sp** de 4 bytes
  - 2º **preenchemos** os 4 bytes da pilha com a informação.
- Então, como compilamos isto?

```
int sumSquare(int x, int y) {
  return mult(x,x)+ y;}
```

Usando a Pilha (2/2)

- Compile manualmente

Função SumSquare

```
int sumSquare(int x, int y)
{
return mult(x,x)+ y;}
```

```
sumSquare:
(salva end. retorno e argumento de SumSquare)
        addi    $sp,$sp,-8          # reserva espaço na pilha
        sw      $ra, 4($sp)         # salva reg. end. retorno
        sw      $a1, 0($sp)         # salvar argumento y
(Transfere arg. de mult e chama função mult)
        add     $a1,$a0,$zero       # transfere arg.x de mult(x,x)
        jal     mult                # chama mult
(restaura arg. De SumSquare e executa operação)
        lw      $a1, 0($sp)         # restaura arg. y
        add     $v0,$v0,$a1         # mult()+y
(restaura end. retorno de SumSquare e a pilha)
        lw      $ra, 4($sp)         # restaura end. retorno
        addi    $sp,$sp,8           # restaura pilha
        jr      $ra                 # retorna para prog. principal
```

Passos para fazer uma chamada de função

1) **Salvar** os valores necessários na pilha.
2) **Atribuir** argumento(s), se existir(em).
3) **Chamar** a função `jal`
4) **Restaurar** os valores da pilha.

Regras para Funções

- **Chamada** com uma instrução **`jal`** **retorna** com uma **`jr $ra`**
- Aceita **até 4 argumentos** em $a0, $a1, $a2 e $a3
- **Valor** de **retorno** sempre está **em $v0** (e se necessário em $v1)
- Deve seguir as convenções de registradores (mesmo em funções que somente você vai chamar)! Então, quais são elas?

Registradores MIPS

A constante 0	$0	$zero
Valores de Retorno	$2-$3	$v0-$v1
Argumentos	$4-$7	$a0-$a3
Temporários	$8-$15	$t0-$t7
Salvos	$16-$23	$s0-$s7
Mais Temporários	$24-$25	$t8-$t9
Stack Pointer	$29	$sp
Return Address	$31	$ra

- Em geral, você pode utilizar ou o nome ou o número. Os nomes deixam o código mais fácil de se ler.

Convenções de Registradores (1/5)

- Caller (*chamador*): a função que faz a chamada
- Callee (*função*): a função sendo chamada
- Quando a *função* retorna da execução, o *chamador* precisa saber quais **registradores podem ter mudado e quais não mudaram**.
- **Convenções** de Registradores: Um conjunto geralmente aceito de regras de quais registradores não mudam após uma chamada de função (`jal`) e quais podem ter sido mudados.

Convenções de Registradores (2/5)

- $0: Nunca muda. Sempre 0.
- $v0-$v1: Muda. Estes são esperados conter novos valores.
- $a0-$a3: Muda. Estes são registradores de argumentos voláteis.
- $t0-$t9: Muda. Por isso eles são chamados temporários: qualquer função pode mudá-los a qualquer momento.

Convenções de Registradores (3/5)

- $s0-$s7: Sem mudança. Muito importante, por isso eles são chamados registradores salvos. Se **a função chamada** (*calle*) muda estes registradores de algum modo, ela **deve restaurar os valores originais antes de retornar**.
- $sp: Sem mudança. O ponteiro da pilha deve apontar para o mesmo lugar antes e depois de uma chamada de jal ou então a função chamadora (*caller*) não será capaz de restaurar os valores da pilha. **A função chamada deve restaurar os valores originais antes de retornar**
- $ra: Muda. A chamada a `jal` vai mudar este registrador por si mesma.

Convenções de Registradores (4/5)

- O que estas convenções significam?
  - Se a função A chama B
    - **a função A deve salvar qualquer registrador temporário** que esteja utilizando na pilha antes de fazer uma chamada `jal`.
    - **A função B deve salvar qualquer registrador S (salvos - sp, $s0-$s7)** que ela pretende utilizar antes de modificar seus valores.
  - Lembre-se: *Caller/callee* precisam salvar somente os registradores temporários que eles estejam utilizando, não todos os registradores.

Convenções de Registradores (5/5)

- Note que, se ***callee*** vai utilizar algum registrador S, ela **deve**:
  - **Salvar** aqueles **registradores S** na pilha.
  - Utilizar os registradores
  - Restaurar os registradores S da pilha.
  - `jr $ra`
- Com os **registradores temporários**, a *calle* **não precisa salvar** na pilha.
- Portanto, a ***caller*** deve **salvar** aqueles **registradores temporários que quer preservar** através da chamada.

Exemplo: Compile isto (1/5)

```
main() {
 int i,j,k,m; /* i-m:$s0-$s3 */
 i = mult(j,k);...;
 m = mult(i,i);...
}
int mult (int mcand, int mlier){
 int product;
 product = 0;
 while (mlier > 0)  {
     product += mcand;
     mlier -= 1; }
 return product;
}
```

Exemplo: Compile isto (2/5)

Programa Principal

```
main() {
int i,j,k,m; /* i-m:$s0-$s3 */
i = mult(j,k);...;
m = mult(i,i);...}
```

```
start:
Transfere argumentos e chama função
      add $a0,$s1,$0      # arg0 = j
      add $a1,$s2,$0      # arg1 = k
      jal mult            # call mult
Salva valor de retorno na variável estática $s0
      add $s0,$v0,$0      # i = mult()
Transfere argumentos e chama função
      add $a0,$s0,$0      # arg0 = i
      add $a1,$s0,$0      # arg1 = i
      jal mult            # call mult
Salva valor de retorno na variável estática $s3
      add $s3,$v0,$0      # m = mult()
Done:
...
```

Exemplo: Compile isto (3/5)

- Notas:
  - **função `main`** (principal) termina com `done`, não com `jr $ra`, assim **não é necessário salvar $ra na pilha**.
  - todas as **variáveis** utilizadas **na função** `main` são **registradores salvos** (**estáticos**), assim não é necessário salvá-las na pilha.

Exemplo: Compile isto (4/5)

Função Multiplicação

```
mult:
     add  $t0,$0,$0      # prod ($t0)=0

Loop:
     slt  $t1,$0,$a1     # mlr($a1) > 0?
     beq  $t1,$0,Fim     # não => vá para Fim
     add  $t0,$t0,$a0    # sim: prod += mc($a0)
     addi $a1,$a1,-1     # mlr -= 1
     j    Loop           # goto Loop

Fim:
     add  $v0,$t0,$0     # $v0 = prod
     jr   $ra            # retorna
```

Mlr:multiplicador
Mc:multiplicando

Exemplo: Compile isto (5/5)

- Notas:
  - **Função não aninhada**: nenhuma chamada `jal` é feita de `mult`
    - não utilizamos qualquer registrador salvo, logo não precisamos salvar nada na pilha.
    - poderíamos ter utilizados **registradores S**, mas **teríamos que salvar** na pilha.
  - **registradores temporários** são utilizados para cálculos intermediários (**não precisam ser salvos**)
  - **$a1** é modificado diretamente (ao invés de copiá-lo em um registrador temporário) já que nós somos **livres para mudá-lo**.
  - o **resultado** é posto **em $v0** antes de retornar.

Exemplo: Compile isto (1/2)

Função recursiva

- Fatorial

```
Int  fact (int n)
{
     if (n < 1) return (1);
     else return (n * fact(n – 1));
}
```

- Colocar na **pilha**, **todos** os **registradores** que **necessitam ser preservados**:
  - $a0-$a3, $t0-$t9, $ra, $s0, ajustar $sp.
  - No retorno: restaurar regs, restaurar $sp.

Exemplo: Compile isto (2/2)

```
fact:
        addi  $sp, $sp,-8      # reserva espaço na pilha (2 palavras)
        sw    $ra, 4($sp)      # salva end. retorno na pilha
        sw    $a0, 0($sp)      # salva argumento (n)
        slt   $t0, $a0, 1      # n<1?
        beq   $t0, $zero, L1   # não: (n>=1) vá para L1

        addi  $v0, $zero, 1    # n=0: $v0=1
        addi  $sp, $sp, 8      # restaura a pilha
        jr    $ra              # retorna da função
L1:     addi  $a0, $a0,-1      # n=n-1 =>$a0=$a0-1
        jal   fact             # chama fact(n-1)=> $v0=fact($a0)
        lw    $a0, 0($sp)      # restaura argumento n ($a0)
        lw    $ra, 4($sp)      # restaura end. retorno
        addi  $sp, $sp, 8      # restaura pilha
        mult  $v0, $a0, $v0    # $v0=n*fact(n-1)
        jr    $ra              # retorna da função
```

Coisas para Lembrar (1/2)

- Funções são chamadas com `jal`, e retornam com `jr $ra`.
- A **pilha** é sua amiga: use-a para **salvar** qualquer coisa que precise. Apenas assegure-se de **deixá-la como a achou**.
- Convenções de Registradores: Cada registrador tem um propósito e limites no seu uso. Aprenda-os e siga-os.

Coisas para Lembrar (2/2)

- Instruções que nós conhecemos até agora:

  Aritmética: `add, addi, sub`

  Memória: `lw, sw`

  Decisão: `beq, bne, slt`

  Desvios incondicionais (pulos): `j, jal, jr`

- Registradores que nós conhecemos até agora:
  - $zero, $v0-$v1, $a0-$a3, $t0-$t7, $s0-$s7, $t8-$t9, $sp, $ra