

Introdução ao MIPS

- Linguagem Assembly e Operações Aritméticas -



Sistemas de Microprocessadores 2021/2022

Linguagem Assembly

- ◆ Tarefa principal do CPU: Executar muitas *instruções*.
- ◆ As instruções definem as ações/operações básicas que a CPU é capaz de levar a cabo.
- ◆ Diferentes CPUs implementam diferentes conjuntos de instruções. O conjunto de instruções implementado por uma determinada CPU designa-se por *Instruction Set Arquitecture (ISA)*.
 - Exemplos: Intel 80x86 (Pentium 4), IBM/Motorola PowerPC (Macintosh), MIPS, Intel IA64, ...

Instruction Set Architectures

- ◆ Inicialmente a filosofia de desenvolvimento consistia em adicionar mais instruções aos novos processadores para realizar tarefas cada vez mais complexas
 - A arquitetura VAX tinha instruções para a multiplicação de polinómios!
 - Estes eram os processadores CISC (Complete Instruction Set Computing)
- ◆ A partir da década de 80 a filosofia RISC - Reduced Instruction Set Computing - começou a impor-se
 - Manter um "instruction set" pequeno e simples facilita o desenho de hardware mais rápido (smaller is faster).
 - As operações complicadas são feitas pelo software através da composição de várias instruções simples.

Arquitetura do MIPS

- ◆ MIPS – companhia de semicondutores que construiu uma das primeiras arquiteturas comerciais RISC.
- ◆ Nesta disciplina iremos estudar a arquitetura do MIPS em detalhe.
- ◆ Porquê o MIPS e não o Intel 80x86?
 - MIPS é simples e elegante. O design da Intel é mais tortuoso devido à necessidade de manter compatibilidade com versões anteriores (legacy issues).
 - MIPS é mais usado que Intel em aplicações embebidas. E há mais computadores embebidos que PCs.



Most HP LaserJet
workgroup printers are
driven by MIPS-based™
64-bit processors.

"Variáveis" em Assembly: Registros (1/3)

- ◆ Ao contrário de Linguagens de Alto Nível, como o C e o Python, o assembly não pode usar variáveis
 - Porque não? "Keep the hardware simple"
- ◆ Os operandos em assembly são os registos
 - Pequeno número de locais de armazenamento construídos diretamente em hardware
 - As operações só podem ser realizadas sobre os registos
- ◆ Benefício: Como os registos são construídos diretamente em hardware, são muito rápidos (uma mudança num registo é feita em menos de um nano-segundo)

"Variáveis" em Assembly: Registros (2/3)

- ◆ Desvantagem: Como os registos são construídos em hardware, existe um número pré-determinado que não pode ser aumentado.
 - Solução: O código do MIPS tem que ser feito com cuidado de forma a usar eficientemente os recursos disponíveis.
- ◆ O MIPS tem 32 registos ... e o x86 ainda tem menos!
 - Porquê 32? **Smaller is faster**
- ◆ Os registos no MIPS têm todos 32 bits
 - Os grupos de 32 bits chamam-se uma word na arquitetura do MIPS
 - Atenção que a dimensão de uma word muda entre diferentes arquiteturas

"Variáveis" em Assembly: Registros (3/3)

- ◆ Os registos estão numerados de 0 a 31
- ◆ Os registos tanto podem ser referenciados por um número como por um nome:
 - Referência por número :
\$0, \$1, \$2, ... \$30, \$31
 - Referência por nome :
 - **Semelhante às variáveis em C**
 $\$16 - \$23 \rightarrow \$s0 - \$s7$
 - **Variáveis temporárias**
 $\$8 - \$15 \rightarrow \$t0 - \$t7$
 - Mais à frente falaremos dos nomes dos 16 registos que faltam.
- ◆ Utilize preferencialmente nomes para tornar o seu código mais legível

QUIZ

Para pensar:

- Quais são os programas compilados que ocupam mais espaço em memória? Os programas para uma arquitetura CISC ou RISC?
- Em que medida o aumento no tamanho das memórias disponíveis terá ajudado à mudança de CISC para RISC?

C, Java, variáveis vs. registos

Sistemas de Microprocessadores 2021/2022

Comentários em Assembly

- ◆ Utilizar comentários também ajuda a tornar o código mais legível!
- ◆ Em MIPS para comentar uma linha utilize o símbolo cardinal (#)
- ◆ Nota: Diferente do C
 - Os comentários em C têm a forma

```
/* comment */
```

e podem ter múltiplas linhas

Instruções em Assembly

- ◆ Em assembly, cada linha de código (designada por Instrução), executa uma, e uma só, ação de uma lista de comandos simples pré-estabelecidos
- ◆ Ao contrário do que acontece no C, cada linha contém no máximo uma instrução para o processador.
- ◆ As instruções em assembly são equivalentes às operações (=, +, -, *, /) em C ou Java.
- ◆ OK, chega de conversa introdutória ... vamos começar a controlar o MIPS!

Adição e Subtração no MIPS (1/4)

♦ Sintaxe:

1 2, 3, 4

Onde :

- 1) nome da operação
- 2) operando que recebe o resultado (“destination”)
- 3) 1º operando (“source1”)
- 4) 2º operando (“source2”)

♦ A sintaxe é rígida:

- 1 operador + 3 operandos
- Porquê? Regularidade para manter o hardware simples

Adição e Subtração no MIPS (2/4)

♦ Adição em assembly

□ Exemplo: add \$s0, \$s1, \$s2 (MIPS)

Equivalente a: $a = b + c$ (em C)

onde os registos do MIPS \$s0, \$s1, \$s2 estão associados às variáveis do C a, b, c

♦ Subtração em assembly

□ Exemplo: sub \$s3, \$s4, \$s5 (MIPS)

Equivalente a: $d = e - f$ (em C)

onde os registos do MIPS \$s3, \$s4, \$s5 estão associados às variáveis do C d, e, f

Adição e Subtração no MIPS (3/4)

- ◆ Qual é o equivalente à seguinte instrução em C?

a = b + c + d - e;

- ◆ Dividir em múltiplas instruções

```
add $t0, $s1, $s2 # temp = b + c
```

```
add $t0, $t0, $s3 # temp = temp + d
```

```
sub $s0, $t0, $s4 # a = temp - e
```

- ◆ Nota: Uma única linha em C pode dar origem a várias linhas em assembly do MIPS.

- ◆ Nota: Tudo aquilo que estiver depois do cardinal é ignorado (comentários)

Adição e Subtração no MIPS (4/4)

- ◆ Qual é o equivalente da seguinte instrução?

$$f = (g + h) - (i + j);$$

- ◆ Temos que utilizar registos temporários

```
add $t0,$s1,$s2      # temp1 = g + h  
add $t1,$s3,$s4      # temp2 = i + j  
sub $s0,$t0,$t1      # f= (g+h) - (i+j)
```

Registo Zero

- ◆ O número zero (0) é um "constante" que aparece muito frequentemente no código.
- ◆ Definimos um registo zero (\$0 ou *\$zero*) para termos o valor 0 sempre à mão. Exemplo:

add \$s0, \$s1, *\$zero* (MIPS)

f = g (C)

onde os registos do MIPS \$s0, \$s1 estão associados às variáveis do C f, g

- ◆ O registo *\$zero* está definido no hardware, e a instrução
add \$zero, \$zero, \$s0
não faz nada

Valores Imediatos (1/2)

- ◆ As constantes numéricas designam-se por "immediatos".
- ◆ Os "immediatos" aparecem frequentemente no código. Sempre que aparecem valores constantes temos que usar instruções específicas (Porquê?)
- ◆ Adição com imediatos:
 $\text{addi } \$s0, \$s1, 10 \text{ (MIPS)}$
 $f = g + 10 \text{ (C)}$
Onde os registos $\$s0, \$s1$ estão associados às variáveis do C f, g
- ◆ Sintaxe semelhante à instrução add, exceto no facto de que o último argumento é um número em vez de um registo

Valores Imediatos (2/2)

- ◆ Não existe uma instrução no MIPS para subtração com imediatos: Porquê?
- ◆ O conjunto de instruções elementares deve ter a menor dimensão possível de forma a simplificar o hardware.
 - Se uma operação pode ser decomposta em instruções mais simples, então não faz sentido inclui-la no "instruction set"
 - addi ..., -X é o mesmo que subi ..., X portanto não há subi
- ◆ $\text{addi } \$s0, \$s1, -10 \text{ (MIPS)}$
 $f = g - 10 \text{ (C)}$
onde os registos $\$s0, \$s1$ estão associados com as variáveis do C
 f, g

QUIZ

- A. Os Tipos são algo característico das declarações em C, que se reflete nas instruções (operadores) do MIPS.
- B. Assumindo os 16 registos que vimos, como só existem 8 variáveis locais (\$s) e 8 variáveis temporárias (\$t), nós não podemos escrever em assembly do MIPS expressões em C que contenham/envolvam mais do que 16 variáveis.
- C. Se a variável p (armazenada no registo \$s0) for um ponteiro para um array de ints, então a instrução em C `p++;` corresponde a `addi $s0 $s0 1`

ABC	
1 :	FFF
2 :	FFT
3 :	FTF
4 :	FTT
5 :	TFF
6 :	TFT
7 :	TTF
8 :	TTT

Concluindo ...

- ◆ Na linguagem Assembly do MIPS:

- Os registos substituem as variáveis em C
 - Existe uma instrução elementar por linha
 - "Simpler is Better"
 - "Smaller is Faster"

- ◆ Novas instruções que aprendemos:

- add, addi, sub

- ◆ Novos registos:

- Variáveis género C: \$s0 - \$s7

- Variáveis temporárias: \$t0 - \$t9

- Zero: \$zero

Introdução à Linguagem Assembly

- Load & Store -

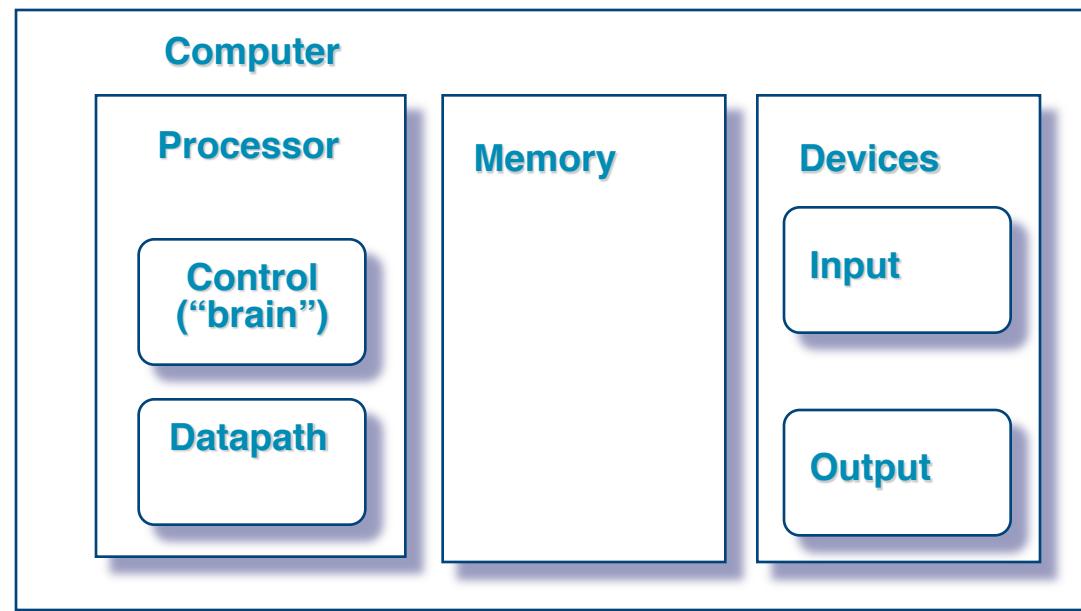
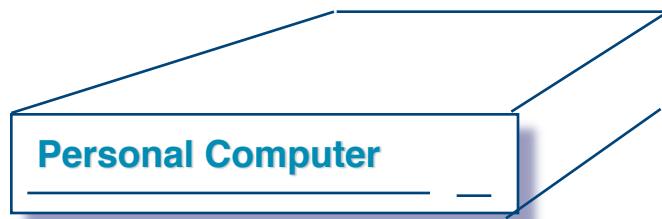


Sistemas de Microprocessadores 2021/2022

A Memória

- ◆ Até aqui mapeámos as variáveis do C em registos do processador; *o que fazer com estruturas de dados de maiores dimensões como as tabelas/arrays?*
- ◆ As estruturas de dados são guardadas na *memória*, que é *1 dos 5 componentes fundamentais do computador*
- ◆ As instruções aritméticas e lógicas do MIPS só operam sobre registos, e nunca sobre a memória.
- ◆ As *instruções de transferência de dados* permitem transferir dados entre os registos e a memória:
 - Da memória para um registo
 - De um registo para a memória

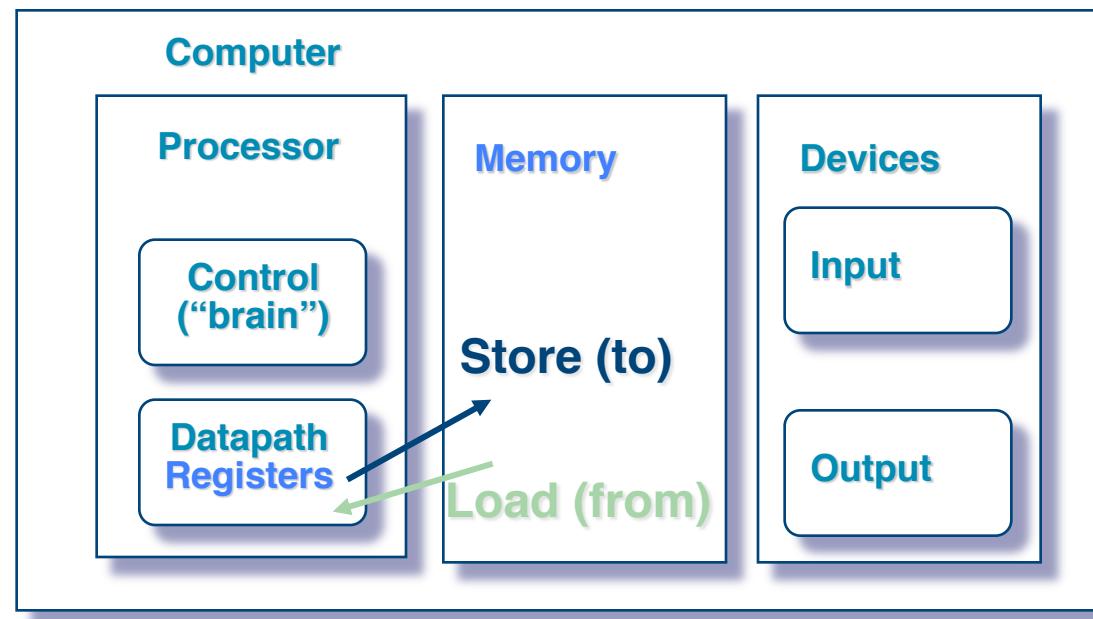
Anatomia: os 5 componentes de um Computador



Anatomia: os 5 componentes de um Computador

- ◆ Os registos estão no "datapath" do processador.
- ◆ Se os operandos estiverem em memória, então:

1. Os dados são transferidos para os registos,
2. a ação é realizada,
3. o resultado é colocado de volta na memória.



Estas são as instruções para “data transfer” ...

Data Transfer: Memória para Reg. (1/4)

- ◆ Para transferir uma "word" de dados precisamos de especificar duas coisas:
 - Registo: especifica-se usando o # de referência (\$0 - \$31) ou o nome simbólico (\$s0, ..., \$t0, ...)
 - Endereço de memória: mais difícil
 - **Pense na memória como sendo uma grande tabela unidimensional. Cada elemento dessa tabela é referenciado por um ponteiro que corresponde ao endereço de uma célula do array (char = 1 byte) .**
 - **Muitas vezes iremos querer incrementar esse ponteiro/endereço**
- ◆ Lembre-se:
 - “Load FROM memory”

Data Transfer: Memória para Reg. (2/4)

- ◆ Para especificar um endereço de memória de onde quer copiar precisa de duas coisas:
 - Um registo contendo um ponteiro para memória
 - Um deslocamento (offset) numérico (sempre **bytes** pois em assembly não existem tipos)
- ◆ O endereço de memória pretendido é a soma destes dois elementos.
- ◆ Exemplo: 8 (\$t0)
 - Especifica o endereço de memória apontado pelo valor no registo \$t0, mais 8 bytes

Data Transfer: Memória para Reg. (3/4)

- ◆ Sintaxe da instrução Load :

1 2, 3 (4)

Em que

- 1) nome da operação
- 2) registo que recebe o valor
- 3) deslocamento **em bytes** (offset)
- 4) registo contendo o endereço base (ponteiro) para a memória

- ◆ Nome da Operação:

- **l_w** (que significa Load Word, ou seja transferir 32 bits (1 word) de cada vez)

Data Transfer: Memória para Reg. (4/4)



- ♦ Exemplo: `lw $t0,12($s0)`

Esta instrução agarra no valor que está no registo `$s0` (ponteiro base), adiciona-lhe um deslocamento de 12 bytes para obter o endereço de memória, e transfere para `$t0` o conteúdo das 4 células de memória apontadas por esse endereço.

- ♦ Notas:

- `$s0` é chamado o registro base
- `12` é chamado o offset
- O offset é geralmente usado para aceder aos elementos de um array ou estrutura: o registo base aponta para o início desse array ou estrutura (nota: o offset é sempre uma constante).

Data Transfer: Registo para Memória

- ◆ Queremos agora transferir do registo para a memória
 - A instrução **store** tem uma sintaxe semelhante ao **load**
- ◆ MIPS Instruction Name:

sw (significa Store Word, ou seja transferir 32 bits (1 word) de cada vez)



- ◆ Exemplo: **sw \$t0, 10(\$s0)**

Esta instrução agarra no ponteiro em **\$s0**, adiciona-lhe 10 bytes, e depois guarda o valor do registo **\$t0** no endereço de memória assim calculado

- ◆ Lembre-se: “Store INTO memory”

Ponteiro vs. Valor

- ◆ Conceito Chave: Um registo guarda sempre um valor de 32 bits. Esse valor pode ser um int, um unsigned int, um ponteiro (endereço de memória), etc. O "tipo" é implicitamente definido pela operação sobre os dados
- ◆ Se fizer add \$t2,\$t1,\$t0
então \$t0 e \$t1 contêm valores/parcelas
- ◆ Se fizer lw \$t2,0(\$t0)
então \$t0 deve conter um ponteiro
- ◆ Não fazer confusão com isto!

Endereçamento: Byte vs. word

- ◆ Todas as words em memória têm um endereço.
- ◆ Os primeiros computadores referenciavam as words da mesma forma que o C numera elementos num array:
 - Memory[0], Memory[1], Memory[2], ...

- ◆ No entanto os computadores precisam de referenciar simultaneamente bytes e words (4 bytes/word)
- ◆ Hoje em dia todas as arquiteturas endereçam a memória em bytes (i.e., “Byte Addressed”). Assim para aceder a words de 32-bits os endereços têm que dar saltos de 4 bytes
 - Memory[0], Memory[4], Memory[8] , ...

Compilação de Acessos à Memória

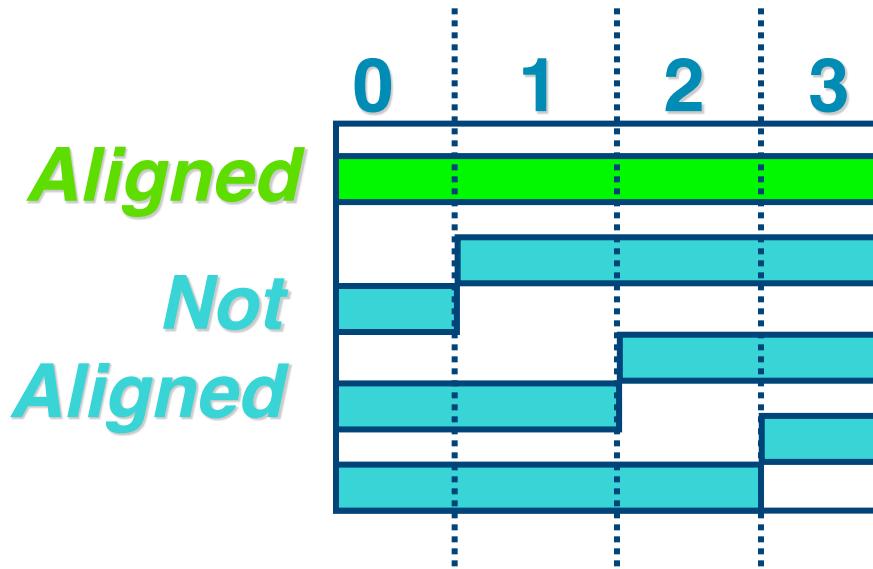
- ◆ Qual o offset que devemos usar com `lw` para aceder a `A[5]`, sendo `A` uma tabela de `int` em `C`?
 - Para selecionar `A[5]` temos que calcular $4 \times 5 = 20$: byte v. word
- ◆ Desafio: Compile a instrução à mão usando registos:
 - `g = h + A[5]` com `g: $s1`, `h: $s2`, endereço base de `A: $s3`
 - Transfira da memória para o registo:
`lw $t0, 20($s3) # $t0 gets A[5]`
 - Adicione 20 a `$s3` para selecionar `A[5]` e coloque em `$t0`
 - Adicione o resultado a `h` e coloque em `g`
`add $s1, $s2, $t0 # $s1 = h+A[5]`

Notas sobre a memória

- ◆ Erro Frequentes: Esquecermo-nos de que os endereços de words sucessivas numa máquina com “Byte Addressing” diferem em mais do que 1.
 - Muitos programadores de assembly cometem erros por assumirem que o endereço da próxima word pode ser obtido incrementando o registo em 1 unidade em vez de adicionarem o número de bytes da word (diferente do C).
 - Ao contrário do que acontece no C, em assembly não existe a noção de tipo, e é impossível o computador saber o tamanho de uma word fazendo o ajuste implícito do incremento dos ponteiros.
 - Lembre-se também que no `lw` e `sw`, a soma do endereço de base com o offset deve ser sempre um múltiplo de 4 (word aligned memory)

Alinhamento de Memória

- ♦ No MIPS as words e objetos são guardados em memória em bytes cujo endereço é sempre múltiplo de 4.



O último dígito hexa do endereço é:

0, 4, 8, or C_{hex}

1, 5, 9, or D_{hex}

2, 6, A, or E_{hex}

3, 7, B, or F_{hex}

- ♦ **Alinhamento de Memória:** os objetos começam sempre em endereços que são múltiplos do seu tamanho
 - Lembram-se do “Bus Error”?

Registros vs Memória

- ◆ O que acontece se houver mais variáveis do que registos?
 - O compilador tenta manter as variáveis mais utilizadas nos registos
 - As variáveis menos usadas são armazenadas em memória: spilling
 - Consulte o comando `register` no C
- ◆ Por que não manter todas as variáveis em memória?
 - Smaller is faster: os registos são mais rápidos do que a memória
 - Os registos são mais versáteis:
 - **Cada instrução aritmética do MIPS pode ler 2 registos, fazer uma operação sobre os dados, e escrever o resultado num registo**
 - **Uma instrução de transferência de dados só pode ler ou escrever 1 operando.**

QUIZ

Queremos traduzir a instrução $*x = *y$ para assembly do MIPS
(x , y são ponteiros armazenados em: $\$s0$ $\$s1$)

- A: add \$s0, \$s1, \$0
- B: add \$s1, \$s0, \$0
- C: lw \$s0, 0(\$s1)
- D: lw \$s1, 0(\$s0)
- E: lw \$t0, 0(\$s1)
- F: sw \$t0, 0(\$s0)
- G: lw \$s0, 0(\$t0)
- H: sw \$s1, 0(\$t0)

- 0 : A
- 1 : B
- 2 : C
- 3 : D
- 4 : E → F
- 5 : E → G
- 6 : F → E
- 7 : F → H
- 8 : H → G
- 9 : G → H

E concluindo ...

- ◆ A memória é endereçada em bytes, mas as instruções `lw` e `sw` acedem a uma word (4 bytes) de cada vez.
- ◆ Um ponteiro (usado em `lw` e `sw`) é só um endereço de memória. Podemos adicionar ou subtrair valores ao endereço base (usando um offset).
- ◆ Novas instruções que vimos:
`lw`, `sw`

Leitura e escrita de bytes (1/2)

- ◆ Para além da transferência de “words” (4 bytes usando `lw` e `sw`), o MIPS permite também a transferência de bytes:
 - load byte: `lb`
 - store byte: `sb`
- ◆ O formato das instruções é semelhante ao `lw`, `sw`
E.g., `lb $s0, 3($s1)`

o byte de memória com endereço = “3” + “conteúdo do registo s1” é copiado para o byte menos significativo do registo s0.

Leitura e escrita de bytes (2/2)

- ◆ O que é que acontece com os outros 24 bits do registo de 32 bits?

□ **lb**: extensão de sinal para preencher os 24 bits mais significativos (relembra que a representação em complementos de 2 assume um número fixo de bits)



- No caso de leitura de “chars” nós não queremos que haja extensão de sinal!
- Neste caso devemos usar a seguinte instrução

load byte unsigned: 1bu

Overflow Aritmético (1/2)

- ◆ Relembrar: O overflow acontece quando existe um erro numa operação aritmética devido à precisão limitada dos computadores (número fixo de bits por registo)
- ◆ Exemplo (números de 4-bits sem sinal):

$$\begin{array}{r} +15 \\ \underline{+3} \\ +18 \end{array} \qquad \begin{array}{r} 1111 \\ 0011 \\ \hline 1\ 0010 \end{array}$$

- Não há espaço para o 5º bit da soma, assim a solução seria 0010, que é +2 em decimal, e portanto está errada.

Overflow Aritmético (2/2)

- ◆ Algumas linguagens detetam o overflow (Ada), enquanto outras não (C)
- ◆ No MIPS existem 2 tipos de instruções:
 - add (add), add immediate (addi) e subtract (sub) em que o overflow é detetado
 - add unsigned (addu), add immediate unsigned (addiu) e subtract unsigned (subu) que não fazem deteção de overflow (no caso de ocorrer é ignorado)
- ◆ O compilador utiliza a aritmética conveniente
 - O compilador de C para o MIPS utiliza addu, addiu, subu

Instruções “Lógicas”

- ◆ Shift Left: `sll $s1,$s2,2 #s1=s2<<2`
 - Guarda em `$s1` o valor de `$s2` deslocado 2 bits para a esquerda, colocando 0's nos bits da direita que ficam “livres”; (`<<` em C)
 - Antes: `00 00 00 02`_{hex}
`0000 0000 0000 0000 0000 0000 0000 0010`_{two}
 - Depois: `00 00 00 08`_{hex}
`0000 0000 0000 0000 0000 0000 0000 1000`_{two}
 - QUIZ: Qual é o efeito aritmético do `sll`?
- ◆ Shift Right: `srl` é o deslocamento no sentido oposto;
`>>`

Concluindo

- ◆ A memória é endereçada em **bytes**, mas as instruções **lw** e **sw** acedem a uma **word (4 bytes)** de cada vez.
- ◆ Um ponteiro (usado em **lw** e **sw**) é só um endereço de memória. Podemos adicionar ou subtrair valores ao endereço base (usando um offset).
- ◆ Para carregar e armazenar bytes devemos utilizar as instruções **lb/sb** (signed) e **lbu/sbu** (unsigned)
- ◆ As instruções **addu/subu/addiu** não causam **overflow**
- ◆ Novas instruções que vimos:

lw, sw, sll, srl, addu, addiu, subu, lb, sb

O que vimos até agora ...

- ◆ As instruções que vimos até agora só manipulam informação (operações aritméticas e transferência de dados) ...
- ◆ Para construir um computador precisamos de tomar decisões e alterar a sequência de execução durante o “runtime” ... imagine como seria fazer um programa se não existissem instruções “if”, “while”, “for”, etc!
- ◆ O C (e o MIPS) permitem usar labels como suporte ao comando “goto”.
 - C: o uso de “breaks” e “goto” é deselegante e altamente desaconselhado;
 - MIPS: A utilização de “goto” é a única forma de modificar o fluxo sequencial de execução!

Para saber mais ...

- ◆ P&H - Capítulos 2.1, 2.2, 2.3 e 2.6
- ◆ P&H - Capítulo 2.9 páginas 95 e 96



Introdução ao MIPS

- Instruções de Decisão -



Sistemas de Microprocessadores 2021/2022

O que vimos até agora ...

- ◆ As instruções que vimos até agora só manipulam informação (operações aritméticas e transferência de dados) ...
- ◆ Para construir um computador precisamos de tomar decisões e alterar a sequência de execução durante o “runtime” ... imagine como seria fazer um programa se não existissem instruções “if”, “while”, “for”, etc!
- ◆ O C (e o MIPS) permitem usar labels como suporte ao comando “goto”.
 - C: o uso de “breaks” e “goto” é deselegante e altamente desaconselhado;
 - MIPS: A utilização de “goto” é a única forma de modificar o fluxo sequencial de execução!

Decisões em C: o comando if

- ♦ Existem dois tipos de “if statements” em C

`if (condition) clause`

`if (condition) clause1 else clause2`

- ♦ Rearrange the second if statement into the following form:

```
if (condition) goto L1;  
    clause2;  
    goto L2;  
L1: clause1;  
L2:
```

- ♦ It's not as elegant as if-else, but it does the same thing

Instruções de decisão no MIPS

- ◆ Instrução de decisão no MIPS:

beq register1, register2, L1

beq significa “Branch if (registers are) equal”

A tradução em C seria:

if (register1==register2) goto L1

- ◆ Instrução de decisão complementar

bne register1, register2, L1

bne significa “Branch if (registers are) NOT equal”

A tradução em C seria :

if (register1!=register2) goto L1

- ◆ Estas instruções são os “conditional branches” (saltos condicionais)

Instrução “goto” no MIPS

- ◆ Para além dos saltos condicionais, o MIPS tem ainda o salto incondicional (unconditional branch):

j label

- O salto na execução é feito diretamente para o sítio referenciado por “label” sem ser necessário satisfazer uma condição

- ◆ Equivalente em C a:

goto label

- ◆ Tecnicamente tem o mesmo efeito que :

beq \$0, \$0, label

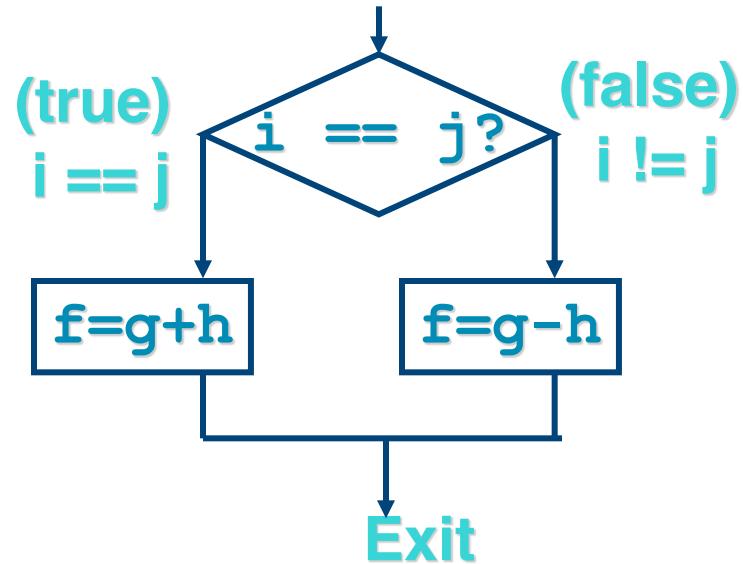
Compilação de um `if` em C (1/2)

♦ Compile à mão

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

♦ Assumindo o seguinte mapeamento variável-registo:

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



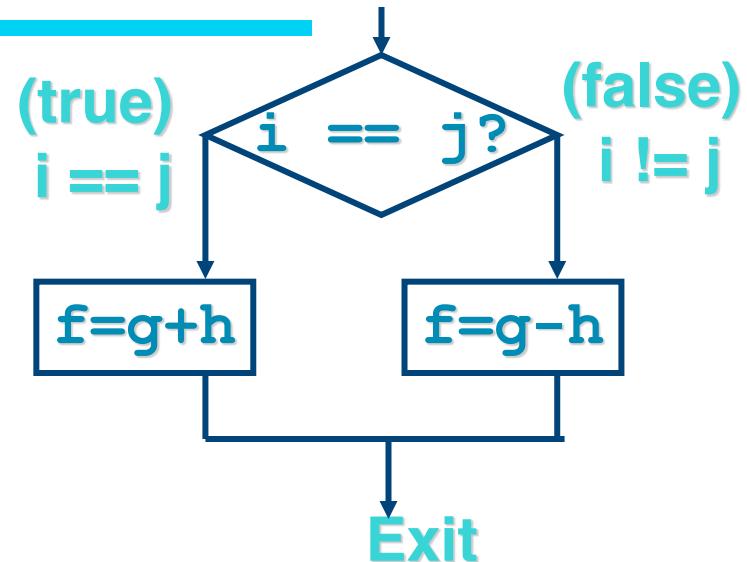
Compilação de um if em C (2/2)

- ♦ Compile à mão

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

- ♦ Código em assembly para MIPS:

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



Nota: O compilador cria automaticamente labels quando aparecem instruções de decisão (branches).

Ciclos (Loops) em C/Assembly (1/3)

- ◆ Ciclo simples em C; A[] é um array de ints

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

- ◆ Re-escrevendo de uma forma deselegante:

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

- ◆ Assumindo agora o seguinte mapeamento variável-registo:

g, h, i, j, base of A
\$s1, \$s2, \$s3, \$s4, \$s5

Ciclos (Loops) em C/Assembly (2/3)

- ◆ Código compilado para MIPS:

```
Loop:    sll $t1,$s3,2          #$t1= 4*i
          add $t1,$t1,$s5        #$t1=addr A
          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
```

- ◆ Código original (guia):

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

Ciclos/Loops em C/Assembly (3/3)

- ◆ Existem três tipos diferentes de ciclos em C:
 - while
 - do... while
 - for
- ◆ Cada um destes ciclos pode ser reescrito usando um dos outros dois. Assim o método utilizado para o do... while pode ser também usado para implementar o while e for.
- ◆ Ideia Chave: Apesar de existirem diferentes formas de construir um ciclo em MIPS, todos eles passam por tomar uma decisão com um **conditional branch**

Desigualdades no MIPS (1/4)

- ◆ Até agora só trabalhámos com igualdades ($==$ e $!=$ no C). No entanto um programa também trabalha com desigualdades ($<$ e $>$ no C).
- ◆ Instruções de desigualdade no MIPS :
 - “Set on Less Than”
 - Sintaxe: `slt reg1, reg2, reg3`
 - Significado:

```
if (reg2 < reg3)
    reg1 = 1;
else
    reg1 = 0;
```

“set” significa “set to 1”,
“reset” significa “set to 0”.

Desigualdades no MIPS (2/4)

- ◆ Compile “à mão” o seguinte código

```
if (g < h) goto Less; # assuma g:$s0, h:$s1
```

- ◆ O resultado em assembly para o MIPS é ...

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

- ♦ O registo \$0 contém sempre o valor 0, e por isso é frequentemente utilizado com bne e beq depois de uma instrução slt.
 - ♦ O par de instruções slt → bne significa if (... < ...) goto...

Desigualdades no MIPS (3/4)

- ◆ Com o `slt` podemos implementar “`<`” ! Mas como será que podemos implementar o `>`, `≤` e `≥` ?
- ◆ Poderia haver mais três instruções similares, mas:
 - Filosofia do MIPS: **Simpler is Better, Smaller is faster**
- ◆ Será que podemos implementar o `≥` usando unicamente o `slt` e “branches”?
- ◆ E quanto ao `>?`
- ◆ E ao `≤?`

Desigualdades no MIPS (4/4)

```
          # a:$s0, b:$s1
slt $t0,$s0,$s1      # $t0 = 1 if a<b
beq $t0,$0,skip      # skip if a >= b
<stuff>              # do if a<b

skip:
```

Existem sempre duas variações:

Usar `slt $t0,$s1,$s0` em vez de `slt $t0,$s0,$s1`

Usar `bne` em vez de `beq`

Desigualdades e Imediatos

- ♦ Existe também uma versão do `slt` para trabalhar com argumentos imediatos (constantes) : `slti`

- útil em ciclos `for`

```
if (g >= 1) goto Loop
```

C Loop: . . .

M slti \$t0,\$s0,1 # \$t0 = 1 if
I # \$s0<1 (g<1)
P beq \$t0,\$0,Loop # goto Loop
S # if \$t0==0
(if (g>=1)

O par `slt` → `beq` significa em C `if (... ≥ ...) goto...`

E quanto aos números sem sinal?

- ◆ Existe também uma instrução de desigualdade para trabalhar com números sem sinal (**unsigned**) :

sltu, sltiu

...que coloca o registo de output a 1 (set) ou 0 (reset) em função de uma comparação sem sinal

- ◆ Qual é o valor de \$t0 e \$t1?

(\$s0 = FFFF FFFA_{hex}, \$s1 = 0000 FFFA_{hex})

slt \$t0, \$s0, \$s1

sltu \$t1, \$s0, \$s1

Signed/Unsigned tem diferentes significados!

- ◆ Os termos *signed/unsigned* estão “sobre utilizados”. É preciso ter cuidado com os seus múltiplos significados
 - Faz / Não faz extensão de sinal
(lb, lbu)
 - Não deteta overflow
(addu, addiu, subu, multu, divu)
 - Faz comparação com/sem sinal
(slt, slti/sltu, sltiu)

Exemplo: O Switch do C (1/3)

- ◆ Escolha entre quatro alternativas diferentes, em função de k ter os valores 0, 1, 2 ou 3. Compile “à mão” o seguinte código em C:

```
switch (k) {  
    case 0: f=i+j; break; /* k=0 */  
    case 1: f=g+h; break; /* k=1 */  
    case 2: f=g-h; break; /* k=2 */  
    case 3: f=i-j; break; /* k=3 */  
}
```

Exemplo: O switch do C (2/3)

- ◆ Isto é um ciclo complicado, portanto o primeiro passo é **simplificar**.
- ◆ Escreva o ciclo como uma cadeia de declarações if-else, as quais já sabemos compilar:

```
if (k==0) f=i+j;  
else if (k==1) f=g+h;  
else if (k==2) f=g-h;  
else if (k==3) f=i-j;
```

- ◆ Assumindo o seguinte mapeamento:

f:\$s0, g:\$s1, h:\$s2,
i:\$s3, j:\$s4, k:\$s5

Exemplo: O switch do C (3/3)

- ◆ O código compilado é:

```
bne    $s5,$0,L1      # branch k!=0
add    $s0,$s3,$s4    #k==0 so f=i+j
j      Exit           # end of case so Exit
L1:   addi   $t0,$s5,-1  # $t0=k-1
      bne   $t0,$0,L2      # branch k!=1
      add    $s0,$s1,$s2    #k==1 so f=g+h
      j      Exit           # end of case so Exit
L2:   addi   $t0,$s5,-2  # $t0=k-2
      bne   $t0,$0,L3      # branch k!=2
      sub    $s0,$s1,$s2    #k==2 so f=g-h
      j      Exit           # end of case so Exit
L3:   addi   $t0,$s5,-3  # $t0=k-3
      bne   $t0,$0,Exit     # branch k!=3
      sub    $s0,$s3,$s4    #k==3 so f=i-j
```

Exit:

QUIZ

Indique o que deveria estar na zona com os pontos de interrogação!

```

Loop: addi $s0,$s0,-1    # i = i - 1
        slti $t0,$s1,2    # $t0 =(j < 2)
        beq  $t0,$0 ,Loop  # goto Loop if $t0 == 0
        slt   $t0,$s1,$s0   # $t0 =(j < i)
        bne  $t0,$0 ,Loop  # goto Loop if $t0 != 0

```

(\$s0=i, \$s1=j)

```
do {i--;} while(???);
```

Concluindo

- ◆ Os branches permitem tomar a decisão do que vai ser executado em “runtime” em vez de “compile time”.
- ◆ As decisões em C são feitas usando conditional statements como o `if`, `while`, `do while`, `for`.
- ◆ As decisões em MIPS são feitas usando conditional branches: `beq` e `bne`.
- ◆ Para complementar os conditional branches em decisões que envolvam desigualdades, vimos as instruções “Set on Less Than”: `slt`, `slti`, `sltu`, `sltiu`
- ◆ Novas instruções que vimos:
`beq`, `bne`, `j`, `slt`, `slti`, `sltu`, `sltiu`

Para saber mais ...

- ◆ P&H - Capítulos 2.1, 2.2, 2.3, 2.5 e 2.6
- ◆ P&H - Capítulo 3.3
- ◆ Resolver a ficha de trabalho



Revisão

- ◆ Os branches permitem tomar a decisão do que vai ser executado em “runtime” em vez de “compile time”.
- ◆ As decisões em C são feitas usando conditional statements como o if, while, do while, for.
- ◆ As decisões em MIPS são feitas usando conditional branches: beq, bne e blt.
- ◆ Para complementar os conditional branches em decisões que envolvam desigualdades, vimos as instruções “Set on Less Than”: slt, slti, sltu, sltiu

MIPS instruction set

MIPS32® Instruction Set Quick Reference

Rd	— DESTINATION REGISTER
Rs, Rt	— SOURCE OPERAND REGISTERS
RA	— RETURN ADDRESS REGISTER (R31)
PC	— PROGRAM COUNTER
ACC	— 64-BIT ACCUMULATOR
Lo, Hi	— ACCUMULATOR LOW (ACC _{31:0}) AND HIGH (ACC _{63:32}) PARTS
±	— SIGNED OPERAND OR SIGN EXTENSION
∅	— UNSIGNED OPERAND OR ZERO EXTENSION
::	— CONCATENATION OF BIT FIELDS
R2	— MIPS32 RELEASE 2 INSTRUCTION
DOTTED	— ASSEMBLER PSEUDO-INSTRUCTION

PLEASE REFER TO "MIPS32 ARCHITECTURE FOR PROGRAMMERS VOLUME II: THE MIPS32 INSTRUCTION SET" FOR COMPLETE INSTRUCTION SET INFORMATION.

ARITHMETIC OPERATIONS		
ADD	Rd, Rs, Rt	Rd = Rs + Rt (OVERFLOW TRAP)
ADDI	Rd, Rs, const16	Rd = Rs + const16* (OVERFLOW TRAP)
ADDIU	Rd, Rs, const16	Rd = Rs + const16*
ADDU	Rd, Rs, Rt	Rd = Rs + Rt
CLO	Rd, Rs	Rd = COUNTLEADINGONES(Rs)
CLZ	Rd, Rs	Rd = COUNTLEADINGZEROS(Rs)
L.A.	Rd, LABEL	Rd = ADDRESS(LABEL)
LI	Rd, IMM32	Rd = IMM32
LUI	Rd, const16	Rd = const16 << 16
MOVE	Rd, Rs	Rd = Rs
NEGU	Rd, Rs	Rd = -Rs
SEB ^{R2}	Rd, Rs	Rd = Rs _{7:0} *
SEH ^{R2}	Rd, Rs	Rd = Rs _{15:0} *
SUB	Rd, Rs, Rt	Rd = Rs - Rt (OVERFLOW TRAP)
SUBU	Rd, Rs, Rt	Rd = Rs - Rt

SHIFT AND ROTATE OPERATIONS		
ROTR ^{R2}	Rd, Rs, bits5	Rd = RS _{BITSS-1:0} :: RS _{31:BITS5}
ROTRV ^{R2}	Rd, Rs, Rt	Rd = RS _{RT40-1:0} :: RS _{31:RT40}
SLL	Rd, Rs, shift5	Rd = Rs << shift5
SLLV	Rd, Rs, Rt	Rd = Rs << RT ₄₀
SRA	Rd, Rs, shift5	Rd = Rs [±] >> shift5
SRAV	Rd, Rs, Rt	Rd = Rs [±] >> RT ₄₀
SRL	Rd, Rs, shift5	Rd = Rs [∅] >> shift5
SRLV	Rd, Rs, Rt	Rd = Rs [∅] >> RT ₄₀

Copyright © 2008 MIPS Technologies, Inc. All rights reserved.

LOGICAL AND BIT-FIELD OPERATIONS		
AND	Rd, Rs, Rt	Rd = Rs & Rt
ANDI	Rd, Rs, const16	Rd = Rs & const16 [∅]
EXT ^{R2}	Rd, Rs, P, S	Rs = RS _{P:S-1:P}
INS ^{R2}	Rd, Rs, P, S	RD _{P:S-1:P} = RS _{S-1:0}
NOP		No-op
NOR	Rd, Rs, Rt	Rd = ~ (Rs Rt)
NOT	Rd, Rs	Rd = ~Rs
OR	Rd, Rs, Rt	Rd = Rs Rt
ORI	Rd, Rs, const16	Rd = Rs const16 [∅]
WSBH ^{R2}	Rd, Rs	Rd = RS _{23:16} :: RS _{31:24} :: RS _{7:0} :: RS _{15:8}
XOR	Rd, Rs, Rt	Rd = Rs ⊕ Rt
XORI	Rd, Rs, const16	Rd = Rs ⊕ const16 [∅]

CONDITION TESTING AND CONDITIONAL MOVE OPERATIONS		
MOVN	Rd, Rs, Rt	If Rt ≠ 0, Rd = Rs
MOVZ	Rd, Rs, Rt	If Rt = 0, Rd = Rs
SLT	Rd, Rs, Rt	Rd = (Rs [±] < Rt [±]) ? 1 : 0
SLTI	Rd, Rs, const16	Rd = (Rs [∅] < const16 [∅]) ? 1 : 0
SLTIU	Rd, Rs, const16	Rd = (Rs [∅] < const16 [∅]) ? 1 : 0
SLTU	Rd, Rs, Rt	Rd = (Rs [∅] < Rt [∅]) ? 1 : 0

MULTIPLY AND DIVIDE OPERATIONS		
DIV	Rs, Rt	Lo = Rs [±] / Rt [±] ; Hi = Rs [±] MOD Rt [±]
DIVU	Rs, Rt	Lo = Rs [∅] / Rt [∅] ; Hi = Rs [∅] MOD Rt [∅]
MADD	Rs, Rt	Acc += Rs [±] × Rt [±]
MADDU	Rs, Rt	Acc += Rs [∅] × Rt [∅]
MSUB	Rs, Rt	Acc -= Rs [±] × Rt [±]
MSUBU	Rs, Rt	Acc -= Rs [∅] × Rt [∅]
MUL	Rd, Rs, Rt	Rd = Rs [±] × Rt [±]
MULT	Rs, Rt	Acc = Rs [±] × Rt [±]
MULTU	Rs, Rt	Acc = Rs [∅] × Rt [∅]

ACCUMULATOR ACCESS OPERATIONS		
MFHI	Rd	Rs = Hi
MFLO	Rd	Rs = Lo
MTHI	Rs	Hi = Rs
MTLO	Rs	Lo = Rs

JUMPS AND BRANCHES (NOTE: ONE DELAY SLOT)		
B	OFF18	PC += OFF18 [±]
BAL	OFF18	RA = PC + 8; PC += OFF18 [±]
BEQ	Rs, Rt, OFF18	If Rs = Rt, PC += OFF18 [±]
BEQZ	Rs, OFF18	If Rs = 0, PC += OFF18 [±]
BGEZ	Rs, OFF18	If Rs ≥ 0, PC += OFF18 [±]
BGEZAL	Rs, OFF18	RA = PC + 8; If Rs ≥ 0, PC += OFF18 [±]
BGTZ	Rs, OFF18	If Rs > 0, PC += OFF18 [±]
BLEZ	Rs, OFF18	If Rs ≤ 0, PC += OFF18 [±]
BLTZ	Rs, OFF18	If Rs < 0, PC += OFF18 [±]
BLTZAL	Rs, OFF18	RA = PC + 8; If Rs < 0, PC += OFF18 [±]
BNE	Rs, Rt, OFF18	If Rs ≠ Rt, PC += OFF18 [±]
BNEZ	Rs, OFF18	If Rs ≠ 0, PC += OFF18 [±]
J	ADDR28	PC = PC _{31:28} :: ADDR28 [±]
JAL	ADDR28	RA = PC + 8; PC = PC _{31:28} :: ADDR28 [±]
JALR	Rd, Rs	Rd = PC + 8; PC = Rs
JR	Rs	PC = Rs

LOAD AND STORE OPERATIONS		
LB	Rd, OFF16(Rs)	Rd = MEM8(Rs + OFF16 [±]) [*]
LBU	Rd, OFF16(Rs)	Rd = MEM8(Rs + OFF16 [±]) [∅]
LH	Rd, OFF16(Rs)	Rd = MEM16(Rs + OFF16 [±]) [*]
LHU	Rd, OFF16(Rs)	Rd = MEM16(Rs + OFF16 [±]) [∅]
LW	Rd, OFF16(Rs)	Rd = MEM32(Rs + OFF16 [±])
LWL	Rd, OFF16(Rs)	Rd = LOADWORDLEFT(Rs + OFF16 [±])
LWR	Rd, OFF16(Rs)	Rd = LOADWORDRIGHT(Rs + OFF16 [±])
SB	Rs, OFF16(Rt)	MEM8(Rt + OFF16 [±]) = RS _{7:0}
SH	Rs, OFF16(Rt)	MEM16(Rt + OFF16 [±]) = RS _{15:0}
SW	Rs, OFF16(Rt)	MEM32(Rt + OFF16 [±]) = Rs
SWL	Rs, OFF16(Rt)	STOREWORDLEFT(Rt + OFF16 [±] , Rs)
SWR	Rs, OFF16(Rt)	STOREWORDRIGHT(Rt + OFF16 [±] , Rs)
ULW	Rd, OFF16(Rs)	Rd = UNALIGNED_MEMORY32(Rs + OFF16 [±])
USW	Rs, OFF16(Rt)	UNALIGNED_MEMORY32(Rt + OFF16 [±]) = Rs

ATOMIC READ-MODIFY-WRITE OPERATIONS		
LL	Rd, OFF16(Rs)	Rd = MEM32(Rs + OFF16 [±]); LINK
SC	Rd, OFF16(Rs)	If ATOMIC, MEM32(Rs + OFF16 [±]) = Rd; Rd = ATOMIC ? 1 : 0

MD00565 Revision 01.01

Breves notas adicionais

- ◆ Pseudo-instruções (e.g. move, branches)
- ◆ Operandos imediatos de 32 bits (instruções **la** e **li**)
- ◆ Chamadas ao sistema: **syscall**
- ◆ Diferença entre **srl** e **sra**

Introdução ao MIPS

- Funções e Procedimentos -



Sistemas de Microprocessadores 2021/2022

Funções em C

```
main() {  
    int i,j,k,m;  
    ...  
    i = mult(j,k); ...  
    m = mult(i,i); ...  
}
```

Numa chamada a uma função, que informação é que o compilador/programador precisa de registar ?

Que instruções permitem fazer isto?

```
/* forma burra de implementar mult */  
  
int mult (int a, int b){  
    int product;  
    product = 0;  
    while (b > 0) {  
        product = product + a;  
        b = b -1; }  
    return product;  
}
```

Chamada de funções – Book-keeping

- ◆ No MIPS os registos são fundamentais para guardar a informação necessária à chamada de funções.
- ◆ Convenção de utilização de registos:
 - Endereço de retorno. \$ra
 - Argumentos / Parâmetros: \$a0, \$a1, \$a2, \$a3
 - Retorno de valores: \$v0, \$v1
 - Variáveis locais: \$s0, \$s1, ... , \$s7
- ◆ Veremos mais tarde que a stack também é utilizada.

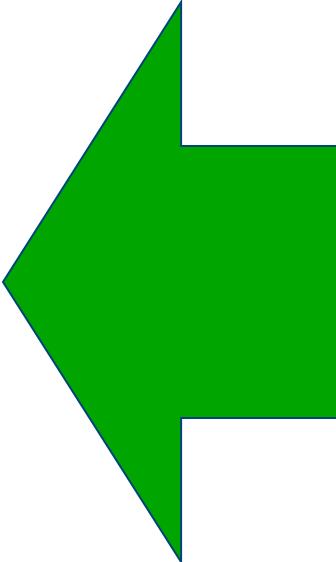
Instruções de suporte a funções (1/6)

C

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

MIPS

address
1000
1004
1008
1012
1016
2000
2004



No MIPS todas as instruções têm 4 bytes e são armazenadas em memória de forma semelhante aos dados. Estes são os endereços onde o programa está armazenado.

Instruções de suporte a funções (2/6)

C

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

M address

I 1000 add \$a0,\$s0,\$zero # $x = a$
P 1004 add \$a1,\$s1,\$zero # $y = b$
S 1008 addi \$ra,\$zero,1016 # $$ra=1016$
1012 j sum # jump to sum
1016 ...
2000 sum: add \$v0,\$a0,\$a1
2004 jr \$ra # nova instrução - salta

para o endereço apontado pelo registo

Instruções de suporte a funções (3/6)

C

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

M
I
P
S

- Pergunta: Por quê utilizar `jr`? Porque não `j`?
- Resposta: A função `sum` pode ser chamada de muitos sítios diferentes. Assim, não podemos regressar para um endereço fixo pré-definido. É preciso disponibilizar um mecanismo para dizer “regressa aqui” !

```
2000 sum: add $v0,$a0,$a1  
2004 jr    $ra # nova instrução
```

Instruções de suporte a funções (4/6)

- ◆ Instrução para simultaneamente saltar e fazer a salvaguarda do endereço de retorno: jump and link (`jal`)

- ◆ Sem `jal`:

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

- ◆ Com `jal`:

```
1008 jal sum                  #$ra=1012,goto sum
```

- ◆ Será que `jal` é imprescindível?

- “Make the common case fast”: a chamada a funções é uma operação muito frequente.
- Para além disso com `jal` o programador não precisa de saber onde é que o código vai ser carregado.

Instruções de suporte a funções (5/6)

- ◆ A sintaxe do `jal` (jump and link) é semelhante à do `j` (jump):

```
jal label
```

- ◆ Na verdade o `jal` deveria ser chamado `la j` (link and jump):
 - Passo 1 (link) - Guarda o endereço da próxima instrução em `$ra`
 - Passo 2 (jump) - Salta para a instrução assinalada por `label`
- ◆ Por que é guardado o endereço da instrução seguinte em vez do da instrução corrente?

Instrução de Suporte a Funções (6/6)

- ◆ Sintaxe do `jr` (jump register):

`jr register`

- ◆ Em vez de darmos um “label” ao jump, passamos um registo que contém o endereço para onde queremos saltar.
- ◆ Estas duas instruções são muito úteis para chamada de funções:
 - `jal` guarda o endereço de retorno no registo (`$ra`)
 - `jr $ra` salta de volta para o sítio onde a função foi chamada (se entretanto não alterarmos o conteúdo do registo)

Nested Procedures (1/2)

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

- ◆ Alguém chamou `sumSquare`, e agora `sumSquare` está a chamar `mult`.
- ◆ Assim o endereço que está em `$ra` é o endereço para onde `sumSquare` vai ter que regressar. No entanto o registo vai ser alterado pela chamada a `mult`.
- ◆ Vamos ter que guardar o endereço de retorno de `sumSquare` antes de fazer a chamada a `mult`.

Nested Procedures (2/2)

- ◆ Iremos ver para a frente que normalmente precisamos de guardar outras informações para além do conteúdo de \$ra.
- ◆ Onde será que podemos guardar essa informação?
- ◆ Quando um programa em C está a correr existem 3 zonas diferentes de memória:
 - Static**: Variáveis declaradas uma única vez no início do programa. Esta zona só é desalocada quando o programa termina.
 - Heap**: Variáveis declaradas de forma dinâmica
 - Stack**: Espaço para ser utilizado pelas funções/procedimentos durante a execução. Este é a zona onde fazemos a salvaguarda de contexto!

Revisão da alocação de memória em C

Address
 ∞
\$sp →
**stack
pointer**



**Espaço para guardar
informação dos
procedimentos**

**Espaço criado explicitamente,
e.g., `malloc()` ; ponteiros do C**

Variáveis Globais
Programa

Utilização da Pilha (1/2)

- ◆ O registo **\$sp** contém sempre o endereço da última zona de memória que está a ser ocupada pela stack (topo da pilha ... ou melhor fundo da pilha!).
- ◆ Para utilizar a pilha, devemos decrementar o ponteiro **\$sp** pelo número de bytes de que vamos necessitar para guardar a informação.
- ◆ Como é que devemos então compilar o programa?

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

Utilização da Pilha (2/2)

- ♦ Compile “à mão” `int sumSquare(int x, int y) { return mult(x,x)+ y; }`

x e y estão em \$a0 e \$a1

sumSquare:

addi \$sp,\$sp,-8 # espaço na stack 2 words
sw \$ra, 4(\$sp) # guardar ret addr
sw \$a1, 0(\$sp) # guardar y

“push”

add \$a1,\$a0,\$zero # mult(x,x)
jal mult # chamar mult

lw \$a1, 0(\$sp) # restaurar y
add \$v0,\$v0,\$a1 # mult() +y

“pop”

lw \$ra, 4(\$sp) # obter ret addr
addi \$sp,\$sp,8 # libertar a stack

jr \$ra

mult: ...

Passos na chamada de uma função

- 1) Salvaguardar a informação necessária na pilha (e.g. endereço de retorno em `$ra`).
- 2) Fazer a passagem de parâmetro(s), se houver
- 3) Saltar para a função chamada usando `jal`
- 4) Restabelecer valores a partir da pilha.

Regras a respeitar pela função chamada

- ◆ A função é chamada através da instrução `jal`, e regressa usando `jr $ra`
- ◆ Aceita um máximo de 4 parâmetros passados através dos registos `$a0`, `$a1`, `$a2` e `$a3`
- ◆ O retorno de valores é sempre feito através de `$v0` (e se necessário de `$v1`)
- ◆ Tem de obedecer às convenções de registos
O que será isto?

Estrutura básica de uma função

Prólogo

```
entry_label:  
    addi    $sp,$sp, -framesize  
    sw      $ra, framesize-4($sp)  # guarda $ra  
    (salvaguarda outros registos se necessário)
```

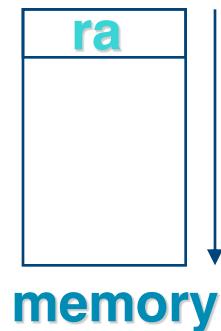
Corpo

... **(chama outras funções...)**

Epílogo

(recupera outros registos)

```
lw      $ra, framesize-4($sp)  # recupera $ra  
addi   $sp,$sp, framesize  
jr     $ra
```



Registros Gerais do MIPS

Constante 0	\$0	\$zero
Reservado para o Assembler	\$1	\$at
Retorno de Valores	\$2-\$3	\$v0-\$v1
Parâmetros	\$4-\$7	\$a0-\$a3
Variáveis Temporárias	\$8-\$15	\$t0-\$t7
Variáveis (saved)	\$16-\$23	\$s0-\$s7
Mais variáveis temporárias	\$24-\$25	\$t8-\$t9
Reservado para o Kernel	\$26-27	\$k0-\$k1
Ponteiro Global	\$28	\$gp
Ponteiro da Pilha	\$29	\$sp
Ponteiro de “frame”	\$30	\$fp
Endereço de Retorno	\$31	\$ra

Existem ainda: Registros reservados (e.g. PC), e registos de vírgula flutuante

Registos desconhecidos

- ◆ `$at`: pode ser utilizado pelo assembler em qualquer altura; não é seguro utilizar
- ◆ `$k0-$k1`: podem ser usados pelo SO em qualquer altura; não é seguro utilizar.
- ◆ `$gp`, `$fp`: vamos ignorar estes registos. Podem ler sobre eles no apêndice A do livro, mas vamos passar sem eles na escrita dos nossos códigos.

Convenção dos Registos (1/4)

- ◆ “Chamante” ou CalleR: a função que chama
- ◆ Chamada ou CalleE: a função chamada
- ◆ Quando a função chamada regressa, a função “chamante” precisa de saber que registos foram alterados e que registos mantiveram o valor.
- ◆ Convenção de registos: Conjunto de regras ou convenções, a ser respeitadas pelo programador/compilador, que define quais os registos que podem ser alterados depois da chamada a `jal`, e quais têm de ser preservados no regresso.

Convenção dos Registos (2/4) - SAVED

- ◆ \$0: Não Altera. Sempre 0.
- ◆ \$s0-\$s7: Repor se modificado. É por isso que são chamados “saved registers”. Se a função chamada alterar estes registos deverá restaurá-los antes de regressar à função chamante.
- ◆ \$sp: Repor se modificado. O stack pointer deverá apontar para o mesmo endereço de memória antes e depois da instrução `jal` que passa a execução para a função chamada.
- ◆ DICA -- Todos os registos “saved” começam por S!

Convenção dos Registos (3/4) - VOLÁTEIS

- ◆ \$ra: Pode ser alterado. A própria instrução ja1 modifica este registo. A função chamante tem a obrigação de o salvaguardar na pilha antes de passar a execução a outra função.
- ◆ \$v0-\$v1: Podem ser alterados. Este registo contêm os valores de retorno
- ◆ \$a0-\$a3: Podem ser alterados. Servem para passar parâmetros à função chamada. A função chamante tem que os salvaguardar se precisar de manter estes valores depois da função chamada regressar.
- ◆ \$t0-\$t9: Podem ser alterados. Por alguma coisa são chamados temporários ...

Convenção de Registos (4/4)

- ◆ Se R é a função chamante, e E é a função chamada, temos em resumo que ...
 - A função R, antes de fazer o `jal` para E, tem que guardar na pilha todos os registos temporários que tencione usar mais tarde (isto para além de `$ra`)
 - A função E tem que guardar na pilha todos os registos S (`saved`) que pretende utilizar, de forma a poder repor os seus valores antes de regressar com `jr`
 - Atenção: caller/callee só precisam de guardar os registos temporários/`saved` que precisem/utilizem, e não todos os registos.

Concluindo

- ◆ As funções são chamadas com `jal`, e regressam com `jr $ra`.
- ◆ “The stack is your friend!”. Utilize-a para guardar tudo aquilo que precisa ... Só tem de ter o cuidado de a deixar como a encontrou.
- ◆ As instruções que já aprendemos
 - Aritmética: `add, addi, sub, addu, addiu, subu`
 - Memória: `lw, sw, lb, sb, lbu, sbu`
 - Decisão: `beq, bne, slt, slti, sltu, sltiu`
 - Saltos incondicionais: `j, jal, jr`
- ◆ Os registos que já conhecemos
 - Todos !

Exemplo: Séries de Fibonacci (1/4)

- ◆ Os números de Fibonacci definem-se da seguinte forma:

$$F(n) = F(n - 1) + F(n - 2),$$

$F(0)$ e $F(1)$ são sempre 0 e 1, respetivamente

- ◆ Assim a série de Fibonacci para $n=9$ é:

$F(0)=0;$	$F(3)=2;$	$F(6)=8;$	$F(9)=34;$
$F(1)=1;$	$F(4)=3;$	$F(7)=13;$	
$F(2)=1;$	$F(5)=5;$	$F(8)=21;$	

- ◆ E o código recursivo em C é

```
int fib(int n) {  
    if(n == 0) { return 0; }  
    if(n == 1) { return 1; }  
    return (fib(n - 1) + fib(n - 2));  
}
```

Exemplo: Séries de Fibonacci (2/4)

◆ Vamos compilar “à mão”

◆ Argumento de entrada => \$a0

◆ Passagem de resultado => \$v0

◆ Precisamos de guardar 3 words na pilha:

- \$ra (a função chama outras funções)
- Um registo para acumular o resultado (e.g. \$s0)
- Guardar o valor de “n” para passar corretamente o parâmetro na segunda chamada

◆ Durante a resolução use o seu espírito crítico para ver se conseguiria resolver o problema guardando menos de 3 words na pilha

```
int fib(int n) {  
    if(n == 0) { return 0; }  
    if(n == 1) { return 1; }  
    return (fib(n - 1) + fib(n - 2));  
}
```

Exemplo: Série de Fibonacci (3/4)

Prólogo

fib:

```
addi $sp, $sp, -12  
sw $ra, 8($sp)  
sw $s0, 4($sp)
```

...

```
int fib(int n) {  
    if(n == 0) { return 0; }  
    if(n == 1) { return 1; }  
    return (fib(n - 1) + fib(n - 2));  
}  
# Espaço para 3 words  
# Guardar endereço de retorno  
# Salvaguardar $s0
```

Epílogo

fim:

```
lw $s0, 4($sp)  
lw $ra, 8($sp)  
addi $sp, $sp, 12  
jr $ra
```

```
#Repor $s0  
#Repor o endereço de retorno  
#Colocar a pilha como foi recebida  
#Regressar à função chamante
```

Exemplo: Série de Fibonacci (4/4)

Corpo

```
# Retornar 0 ou 1 quando $a0 é 0 ou 1
addi      $v0, $zero, $zero
beq      $a0, $zero, fim      #Preparar para sair ($a0=0)
addiu    $v0, $zero, 1
addiu    $t0, $zero, 1      #Será que podíamos não sujar $t0?
beq      $a0, $t0, fim      #Preparar para sair ($a0=1)

addiu    $a0, $a0, -1      #Preparar argumento 1ª chamada
sw       $a0, 0($sp)      #Salvaguardar para a 2ª chamada
jal     fib                #fib(n-1)
addi    $s0, $v0, $zero      #salvaguardar o result preliminar
lw       $a0, 0($sp)      #Preparar argumento 2ª chamada
addiu    $a0, $a0, -1
jal     fib                #fib(n-2)

add      $v0, $v0, $s0      #resultado final
```

```
int fib(int n) {
    if(n == 0) { return 0; }
    if(n == 1) { return 1; }
    return (fib(n - 1) + fib(n - 2));
}
```

Exemplo B - Faça a Compilação (1/3)

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

Exemplo B - Faça a Compilação (2/3)

main:

...

```
add $a0,$s1,$0  
add $a1,$s2,$0  
jal mult  
add $s0,$v0,$0
```

...

```
add $a0,$s0,$0  
add $a1,$s0,$0  
jal mult  
add $s3,$v0,$0
```

...

```
main() {  
    int i,j,k,m; /* i-m:$s0-$s3 */  
    i = mult(j,k); ...  
    m = mult(i,i); ... }  
  
    # arg0 = j  
    # arg1 = k  
    # call mult  
    # i = mult()  
  
    # arg0 = i  
    # arg1 = i  
    # call mult  
    # m = mult()
```

- Nota: todas as variáveis a ser preservadas na função main estão em registos “saved” e portanto não precisam de ser salvaguardadas na pilha.

Exemplo B - Faça a Compilação (3/3)

mult:

```
add $t0,$0,$0      # prod=0
```

```
int mult (int mcand, int mlier){  
    int product = 0;  
    while (mlier > 0) {  
        product += mcand;  
        mlier -= 1; }  
    return product;  
}
```

Loop:

```
slt $t1,$0,$a1      # b > 0?  
beq $t1,$0,Fim      # no -> goto Fim  
add $t0,$t0,$a0      # prod+=a  
addi $a1,$a1,-1      # b=b-1  
j Loop                # goto Loop
```

Fim:

```
add $v0,$t0,$0      # $v0=prod  
jr $ra                # return
```

◆ Notas:

- Não há chamadas a jal feitas dentro do mult, assim não é preciso fazer a salvaguarda de \$ra
- Também não são usados saved registers o que significa que não há contexto a ser guardado na pilha

QUIZ

Na tradução para MIPS ...

- A. Podemos COPIAR \$a0 para \$a1 (e depois não guardar \$a0 ou \$a1 na pilha) para guardar o n em chamadas sucessivas.
- B. Temos SEMPRE que salvaguardar o \$a0 na pilha dado que é alterado.
- C. Temos sempre que salvaguardar o \$ra na pilha dado que precisamos de saber para onde retornar ...

```
int factorial(int n) {  
    if(n == 0)  
        return 1;  
    else  
        return (n*fatorial (n-1)) ; }
```

ABC
0 : FFF
1 : FFT
2 : FTF
3 : FTT
4 : TFF
5 : TFT
6 : TT F
7 : TTT

Para saber mais ...

- ◆ P&H - Capítulos 2.6 e 2.7
- ◆ P&H - Capítulo 2.9 páginas 95 e 96
- ◆ Anexo A-6 do livro



Introdução ao MIPS

- Operações Lógicas -



Sistemas de Microprocessadores 2021/2022

Multiplicação Inteira (1/2)

- ◆ No MIPS, se multiplicarmos 2 registos de 32 bits temos um resultado que em geral ocupa 64 bits:

- 32-bit value x 32-bit value = 64-bit value

- ◆ Sintaxe da multiplicação (com sinal):

```
mult      register1, register2
```

- O resultado de 64 bits é guardado em dois registos especiais:

- A word mais significativa do produto é guardada no registo HI
 - e a word menos significativa no registo LO

- HI e LO são 2 registos especiais separados dos 32 registos “general purpose”

- Use `mfhi register` & `mflo register` para mover o conteúdos de HI, LO para outro registo

Multiplicação Inteira (2/2)

- ◆ Exemplo:

 - em C: $a = b * c;$

 - em MIPS:

 - considere b:c em \$s2:\$s3; e assuma que a ocupa \$s0 e \$s1

```
mult $s2,$s3          # b*c
mfhi $s0              # upper half of
                      # product into $s0

mflo $s1              # lower half of
                      # product into $s1
```

- ◆ Nota: Muitas vezes só nos importamos com a word menos significativa.

Voltando ao fatorial

```
int factorial(int n){  
    if(n == 0)  
        return 1;  
    else  
        return (n*fatorial(n-1));}
```

fatorial:

```
    addu $v0, $zero, 1
```

Loop:

```
    beq $a0, $zero, fim
```

```
    addiu $a0, $a0, -1
```

```
    mult $v0, $a0
```

```
    mflo $v0
```

```
    j Loop
```

fim:

Operações Bitwise

- ◆ Até agora fizemos operações aritméticas (add, sub, addi), acessos a memória (lw e sw), “branches” e saltos.
- ◆ Em todos estes casos o registo é visto como um todo, representando um número com ou sem sinal.
- ◆ Nova perspetiva: ver o registo como um conjunto de 32 bits não relacionados, em vez de um número único representado por 32 bits.
- ◆ Neste contexto podemos querer aceder a bits individuais (ou grupos de bits).
- ◆ Para isso vamos precisar de duas novas classes de operações:
 - Operações lógicas
 - Shifts/Deslocamentos (já vimos)

Operações Lógicas

- ◆ As duas operações lógicas fundamentais são:
 - AND: saída 1 se, e só se, ambas as entradas são 1
 - OR: saída 0 se, e só se, ambas as entradas forem 0
- ◆ Sintaxe semelhante ao add, addi, etc
 - **OP** \$destino, \$fonte1, \$fonte2/imediato
- ◆ Nome das instruções:
 - and, or Neste caso o terceiro argumento é um registo
 - andi, ori Neste caso o terceiro argumento é um imediato
- ◆ Os operadores lógicos do MIPS são sempre **bitwise**, significando que o bit 0 da saída depende dos bits 0's das entradas, o bit 1 dos bits 1's, etc.
 - C: Bitwise AND é & (e.g., $z = x \& y;$)
 - C: Bitwise OR é | (e.g., $z = x | y;$)

Utilidade das Operações Lógicas (1/2)

- ◆ Note que fazer o and de um bit desconhecido com 0 produz sempre 0. Por outro lado o resultado do and com 1 produz sempre o bit original.
- ◆ Isto é extremamente útil para criar máscaras

□ Exemplo:

mask: 0000 0000 0000 0000 0000 1101 1001 1010
1011 0110 1010 0100 0011 1111 1111 1111

□ O resultado deste AND é:

0000 0000 0000 0000 0000 1101 1001 1010

mask dos últimos 12 bits

Utilidade das Operações Lógicas (2/2)

- ◆ A segunda sequência de bits do exemplo é chamada uma **máscara**, e serve para isolar os últimos 12 bits da direita **mascarando** o resto da “bitstring” original.
- ◆ Usando a instrução `andi`, e assumindo que a sequência original estava no registo `$t0`, teríamos:

```
andi      $t0, $t0, 0xFFFF
```

- ◆ De forma semelhante repare que fazer o `or` de um bit desconhecido com 1 produz sempre 1, e com 0 produz o bit original.
- ◆ Esta propriedade pode ser utilizada para forçar (mascarar) certos bits da string a ser 1s.
 - Se `$t0` contém `0x12345678`, então depois da instrução:
`ori $t0, $t0, 0xFFFF`
 - ... `$t0` contém `0x1234FFFF`.

Instruções de Deslocamento (revisão) (1/3)

- ◆ Sintaxe

OP \$destino, \$fonte, imediato

- ◆ O valor imediato especifica o número de bits que são deslocados (<32)
- ◆ MIPS shift instructions:
 - **sll** (shift left logical): desloca para a esquerda e preenche os bits vazios com 0's
 - **srl** (shift right logical): desloca para a direita e preenche os bits vazios com 0's
 - **sra** (shift right arithmetic): desloca para a direita e preenche os bits vazios com a extensão de sinal

Instruções de Deslocamento (revisão) (2/3)

- ◆ Deslocamentos lógicos para a esquerda e direita
 - Exemplo: shift right de 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

- Exemplo: shift left de 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0011 0100 0101 0110 0111 1000 0000 0000

- ◆ Um bom compilador de C deteta quando existem multiplicações por potências de 2 e usa a instrução sll

a *= 8; (em C)

Compila como:

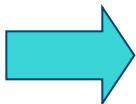
sll \$s0,\$s0,3 (em MIPS)

Instruções de deslocamento (3/3)

- ◆ Deslocamento aritmético

- Exemplo: shift right arith de 8 bits

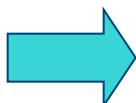
0001 0010 0011 0100 0101 0110 0111 1000



0000 0000 0001 0010 0011 0100 0101 0110

- Exemplo: shift right arith de 8 bits

1001 0010 0011 0100 0101 0110 0111 1000



1111 1111 1001 0010 0011 0100 0101 0110

- ◆ A instrução sra é utilizada para fazer divisões com sinal por potências de 2

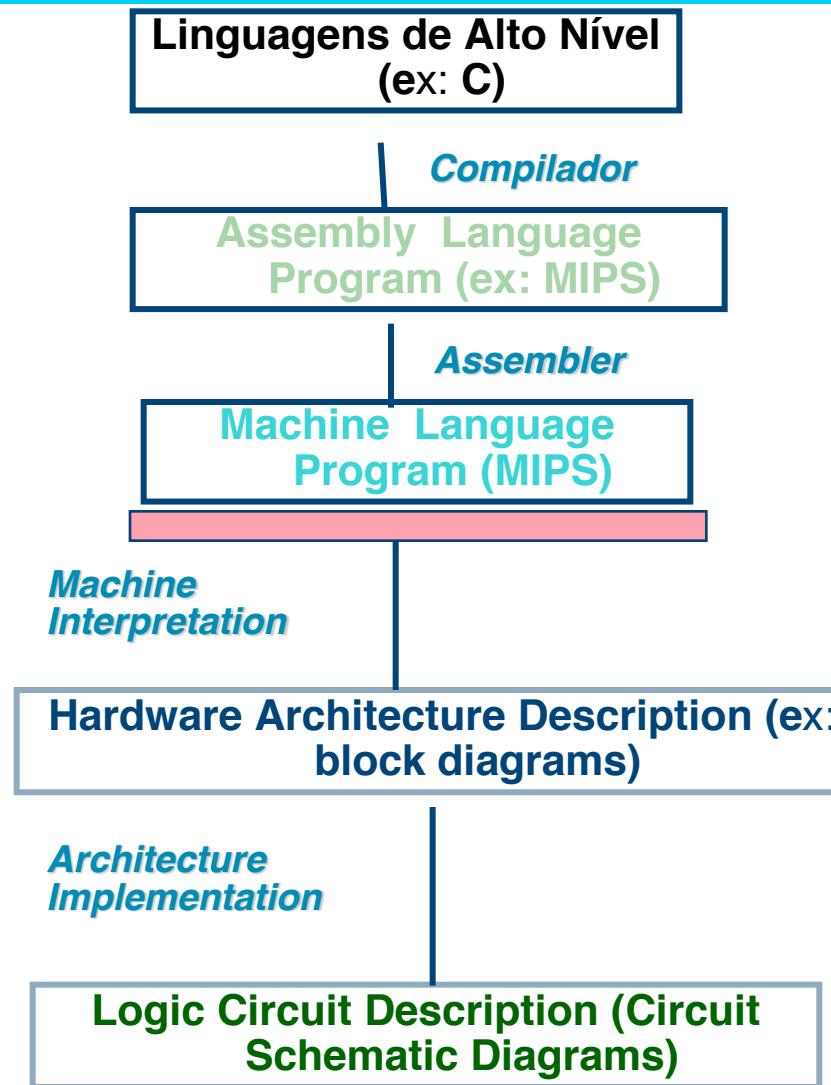
Introdução ao MIPS

- Representação de Instruções -



Sistemas de Microprocessadores 2021/2022

Níveis de representação num computador

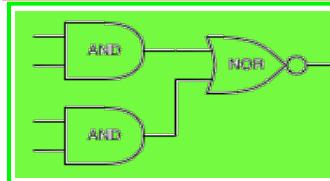
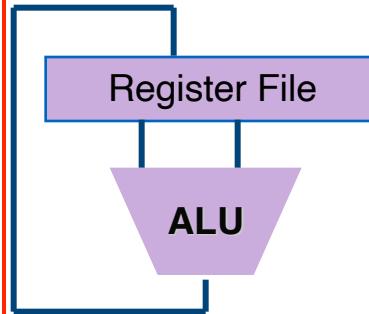


temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;

PC/EDA

Iw \$t0, 0(\$2)
Iw \$t1, 4(\$2)
Sw \$t1, 0(\$2)
Sw \$t0, 4(\$2)
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111

SMP



LSD

Ideia brilhante: O conceito de Stored-Program

- ◆ Os computadores baseiam-se em 2 princípios chave:
 - 1) As instruções são representadas através de “bitstrings”/ padrões de bits - podemos pensar nas instruções como números.
 - 2) Assim, programas inteiros podem ser armazenados em memória para serem lidos ou escritos de forma semelhante ao que acontece com os dados.
- ◆ VANTAGEM: Simplifica o SW/HW dos computadores:
 - A tecnologia de memória para dados é usada também para programas

Consequência 1: tudo funciona por endereços

- ◆ Como tanto as instruções como os dados são armazenados em memória, tudo é referenciado por endereços: instruções, dados, words, etc.
- ◆ Os ponteiros do C são simplesmente endereços de memória
 - isto permite-nos apontar para qualquer coisa, o que pode conduzir a bugs difíceis de depurar
- ◆ O MIPS tem um registo, o “Program Counter” (PC), que indica a próxima instrução a ser executada.
- ◆ Os “branches” e os “jumps” modificam a sequência de execução através de escritas no PC

Consequência 2: Binary Compatibility

- ◆ Os programas são normalmente distribuídos em binário por questões de simplicidade de instalação e proteção da propriedade intelectual:
 - O programa fica vinculado a um determinado instruction set
 - Diferentes versões para diferentes arquiteturas (Macintoshes, PCs)
 - A comunidade “open source” muitas vezes disponibiliza as fontes
- ◆ As novas máquinas querem simultaneamente correr velhos programas (“binaries”) bem como novos programas compilados com novas instruções
- ◆ Isto obriga a “backward compatible” dos instruction sets (ex: Intel)

As instruções como números binários (1/2)

- ◆ No MIPS a manipulação de dados é feita com base em words (blocos de 32-bits):
 - Cada registo é uma word
 - Tanto `lw` como `sw` transacionam com a memória uma word de cada vez.
- ◆ Então como será que devemos representar instruções em binário?
 - A filosofia do MIPS (RISC) é baseada na simplicidade: assim, se os dados estão em words, é conveniente colocar as instruções também em words.
- ◆ 1 instrução => 1 word em memória

As instruções como números binários (2/2)

- ♦ Como uma word tem 32 bits, dividimos a word que representa uma instrução em partes chamadas “campos”.
 - ♦ Cada “campo” diz ao processador algo sobre a instrução em causa.
 - ♦ Podíamos definir “campos” diferentes para instruções diferentes, no entanto isto contraria a filosofia do MIPS de simplicidade e “standardização”.
 - ♦ O MIPS tem somente três tipos de instruções, obedecendo cada tipo à mesma organização em termos de “campos”.
 - formato I: usado para codificar instruções com imediatos (exceto os deslocamentos – sll, ...), os lw e sw (em que o offset conta como um imediato), e os “branches” (beq e bne),
 - formato J: usado para o j e jal
 - formato R: usado para todas as outras instruções

Instruções formato R (1/3)

- ◆ Tem seis “campos” distintos com o seguinte número de bits: $6 + 5 + 5 + 5 + 5 + 6 = 32$

6	5	5	5	5	6
---	---	---	---	---	---

- ◆ Cada campo tem um nome/sigla:

opcode	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------

- ◆ Os campos “r” normalmente especificam registos
 - rs (Source Register): especifica o primeiro operando
 - rt (Target Register): especifica o segundo operando
 - rd (Destination Register): especifica o registo que recebe o resultado

Nota: Cada campo tem 5 bits permitindo distinguir 32 entidades (bate certo?)

Instruções formato R (2/3)



- ◆ O campo opcode especifica parcialmente qual é a instrução.
- ◆ O campo funct é combinado com opcode para definir exatamente a instrução (um add, sub, etc)
- ◆ No caso das instruções R o campo opcode é sempre zero. Assim a instrução é definida unicamente pelo conteúdo de funct.

Instruções formato R (3/3)

opcode	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------



Questões Pertinente:

- Por que é que opcode e funct não são contíguos formando um único campo de 12 bits?

- Porque é que as instruções de tipo R têm campo opcode?

□ **Resposta: Vamos ver isto melhor mais à frente ... mas a razão é mais uma vez simplicidade e uniformidade da arquitetura.**

- ◆ O campo shamt indica o deslocamento a ser feito pelas instruções sll, srl e sra. Este campo está a 0 em todas as instruções R que não sejam shift's.
- ◆ Repare que os campos rs, rt, rd e shamt só têm 5 bits, o que significa que só podem representar números inteiros entre 0 e 31.
 - Será isto suficiente?

Exemplo formato R (1/2)

- ◆ Instrução MIPS:

add \$8, \$9, \$10

opcode = 0 (veja a tabela no livro)

funct = 32 (veja a tabela no livro)

rd = 8 (destino)

rs = 9 (primeiro *operando*)

rt = 10 (segundo *operando*)

shamt = 0 (não é um shift)

Exemplo formato R (2/2)

- ◆ Instrução MIPS:

add \$8, \$9, \$10

Representação em decimal do valor de cada campo:

0	9	10	8	0	32
---	---	----	---	---	----

Representação em binário:

0000000	01001	01010	01000	00000	1000000
---------	-------	-------	-------	-------	---------

hex

Representação em hexa: 012A 4020_{hex}

Representação em decimal: 19,546,144_{ten}

- Isto é uma Instrução em Linguagem Máquina (Machine Language Instruction)

Instruções formato I (1/4)

- ◆ E quanto às instruções com valores imediatos (constantes)?
 - Um campo de 5-bits só pode representar valores entre 0 e 31: normalmente os valores imediatos são bastante maiores do que 31
 - Idealmente o MIPS só teria um formato de instrução, mas infelizmente isso não é possível. Assim temos que fazer compromissos (é por isso que somos engenheiros ;-))
- ◆ Vamos tentar definir um novo formato que permita representar imediatos e seja o mais consistente possível com o formato R:
 - Repare que as instruções com imediatos envolvem no máximo 2 registos (e nunca 3).

Instruções formato I (2/4)

- ◆ Vamos definir uma divisão em “campos” com o seguinte número de bits: $6 + 5 + 5 + 16 = 32$ bits



- ## ◆ O nome dos campos é:



- ◆ Ideia Chave: Repare que só o último campo é inconsistente com o formato R. E ainda mais importante: o opcode, que define a instrução, está ainda no mesmo sítio.
 - Começa a perceber agora o por quê dos campos opcode e funct nas instruções R?

Instruções formato I (3/4)



- ◆ O que significam estes campos

- opcode: o mesmo que vimos para as instruções R com a exceção de que agora não existe um campo funct. O campo opcode define sozinho de que instrução se trata.
- Isto também esclarece o facto de as instruções R terem dois campos de 6-bits para identificar a instrução, em vez de um único campo de 12-bits. É a forma de manter a coerência entre diferentes formatos, deixando 16 bits contíguos para acomodar imediatos no caso das instruções I.
- rs: especifica um registo operando (no caso de existir)
- rt: especifica o registo que vai receber o resultado (target register).

Instruções formato I (4/4)

- ◆ O campo imediato:
 - O campo imediato tem 16bits e pode representar 2^{16} valores diferentes
 - Esta gama é suficientemente ampla para armazenar o deslocamento típico em instruções `lw` e `sw`, bem como a maioria dos valores usados com a instrução `slti`.
 - Nas instruções `addi`, `slti`, `sltiu`, o sinal do resultado é estendido para 32 bits e guardado no registo `rt`. Assim o imediato é interpretado como um inteiro com sinal (complementos de 2).
 - Veremos à frente o que fazer quando o número imediato é demasiado grande para ser representado só com 16 bits...

Exemplo formato I (1/2)

- ◆ Instrução MIPS:

addi \$21, \$22, -50

opcode = 8 (ver tabela no livro)

rs = 22 (registro operando)

rt = 21 (resgisto alvo/destino)

immediate = -50 (valor passado)

Exemplo formato I (2/2)

- ◆ MIPS Instruction:

addi \$21, \$22, -50

Representação de campos decimal:

8	22	21	-50
---	----	----	-----

Representação de campos binária:

001000	10110	10101	1111111111001110
--------	-------	-------	------------------

Representação hexadecimal : 22D5 FFCE_{hex}

Representação decimal: 584,449,998_{ten}

Limitação do formato I (1/3)

◆ Problema:

- Na maior parte das situações instruções como addi, lw, sw e slti têm imediatos que são suficientemente pequenos para caberem num campo de 16 bits.
- Isto valida a opção de usar instruções I que ocupam uma word (make the common case faster)
- ...no entanto o que fazer quando o imediato não couber no campo de 16 bits?
- Precisamos de ter uma estratégia para lidar com imediatos de 32 bits.

Limitação do formato I (2/3)

◆ Solução:

- Resolver com software + nova instrução de suporte
- Em vez de criarmos um conjunto de novas instruções, vamos manter aquelas que já vimos que serão coadjuvadas por nova instrução adicional.

◆ Nova instrução:

lui register, immediate

- lui significa Load Upper Immediate
- A instrução agarra nos 16-bits mais significativos do imediato e coloca-os na metade de cima do registo destino
- A metade menos significativa do registo fica com 0s

Limitação do formato I (3/3)

- ◆ Solução do problema:
 - Como é que lui nos pode ajudar?

- Exemplo:

```
addi    $t0,$t0, 0xABABCD
```

É codificado:

```
lui     $at, 0xABAB
ori     $at, $at, 0xCD
add    $t0,$t0,$at
```

Lembra-se do registo \$at ?
É o registo “assembler temporary”

- As instruções de formato I ori e addi têm um imediato de 16-bits.
- *Era bom que o assembler fizesse este desdobramento de forma automática ...*

Quiz

Que instrução é representado por $35_{(10)}$?

1. add \$0, \$0, \$0
2. subu \$s0,\$s0,\$s0
3. lw \$0, 0(\$0)
4. addi \$0, \$0, 35
5. subu \$0, \$0, \$0

opcode	rs	rt	rd	shamt	funct
opcode	rs	rt	rd	shamt	funct
opcode	rs	rt		offset	
opcode	rs	rt		immediate	
opcode	rs	rt	rd	shamt	funct

Números e nomes dos registos:

0: \$0, .. 8: \$t0, 9:\$t1, ..15: \$t7, 16: \$s0, 17: \$s1, .. 23: \$s7

Opcodes e campos

add: opcode = 0, funct = 32

subu: opcode = 0, funct = 35

addi: opcode = 8

lw: opcode = 35

Pseudo-Instruções (1/4)

◆ **Pseudo-Instrução:** É um comando para o MIPS que não é diretamente mapeado numa instrução linguagem máquina.

- Em vez de ser codificada em hardware, a pseudo-instrução é convertida pelo assemblador numa sequência de instruções de linguagem máquina.

◆ **Exemplos:**

- *Register move*

move reg2, reg1

É desdobrado em:

add reg2, \$zero, reg1

Pseudo-Instruções (2/4)

◆ Exemplos:

□ Load Immediate

li reg,value

Se o imediato couber em 16 bits:

addi reg,\$zero,value

Caso contrário:

lui reg,upper 16 bits of value
ori reg,reg,lower 16 bits

Nota: Repare que o assemblador tem que fazer a avaliação em “compile time”

Pseudo-Instruções (3/4)

◆ Exemplo:

- Load Address: Coloca o endereço de uma instrução ou variável global num registo

la reg,label

Se o valor couber em 16 bits:

addi reg,\$zero,label_value

Se não:

lui reg,upper 16 bits of value
ori reg,reg,lower 16 bits

Pseudo-Instruções (4/4)

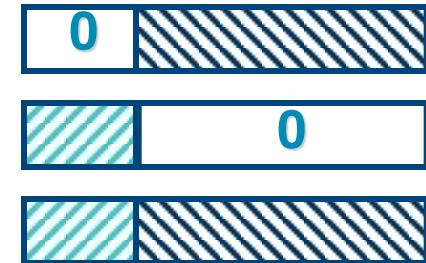
Exemplo

Rotate Right Instruction

ror reg, value

Fica como:

srl \$at, reg, value
sll reg, reg, 32-value
or reg, reg, \$at



- ◆ O registo \$at é utilizado pelo assemblador como registo auxiliar para implementar as pseudo-instruções. Por isso não dever ser utilizado diretamente pelo programador

True Assembly Language (1/2)

- ◆ **MAL** (MIPS Assembly Language): conjunto de instruções que o programador pode utilizar para fazer código para o MIPS; isto inclui as pseudo-instruções.
- ◆ **TAL** (True Assembly Language): conjunto de instruções que são traduzidas diretamente para uma instrução em linguagem máquina de 32 bits
- ◆ Um programa tem de ser convertido de MAL para TAL antes de ser traduzido em linguagem máquina (“1s e 0s”).

True Assembly Language (2/2)

- ◆ Como é que o assemblador do MIPS reconhece uma pseudo-instrução?
 - Verifica se a instrução está na lista oficial de pseudo-instruções (caso do `ror` e `move`)
 - Também existem situações em que a instrução tem um sinónimo TAL mas os operandos estão incorretos (tipicamente existe um imediato com mais de 16 bits). Neste caso faz o desdobramento ...

```
addi    $t0, $s0, 0x0ABC3EF1
```

O imediato tem mais do que 16 bits. Assim de MAL para TAL temos ..

```
lui    $at, 0x0ABC
ori    $at,$at,0x3EF1
add    $t0,$s0,$at
```

Branches e endereçamento relativo (1/5)

- ◆ Considere o formato I para codificar a instrução `beq` ou `bne`



- ◆ opcode especifica beq ou bne
 - ◆ rs e rt especificam os registos a ser comparados
 - ◆ O que é que o campo immediate especifica?
 - Immediate só tem 16 bits
 - PC (Program Counter) tem o endereço da instrução que está a ser executada. É um ponteiro para memória com 32-bits.
 - ◆ Assim o immediate não pode especificar o endereço completo para onde queremos saltar com o branch.

Branches e endereçamento relativo (2/5)

- ◆ Como é que tipicamente se usam branches (“check the common case”)?
 - Resposta: ciclos if-else, while, for
 - Os Loops são normalmente pequenos: tipicamente até 50 instruções
 - As chamadas de funções e os saltos incondicionais são feitos com instruções `j` e `jal`, e não branches.
- ◆ Conclusão: potencialmente um “branch” pode mover a execução para qualquer ponto da memória, mas, na maior parte dos casos, o branch só precisa de alterar o PC numa pequena quantidade de bytes.

Branches e endereçamento relativo (3/5)

- ◆ Solução para os “branches” serem codificados numa instrução de 32-bits: PC-Relative Addressing
- ◆ O campo `immediate` de 16 bits é interpretado como um inteiro com sinal em complementos de 2. Este valor é *adicionado* ao PC no caso de se verificar o salto (endereçamento relativo à posição atual)
- ◆ Com este mecanismo é possível fazer saltos de $\pm 2^{15}$ words com relação ao valor corrente do registo PC. Isto é suficiente para a maior parte dos loops!
- ◆ Ideias para otimizar isto ainda mais?

Branches e endereçamento relativo (4/5)

- ◆ Lembre-se que as instruções são words, e que as words são guardadas de forma alinhada na memória (o “byte address” de uma instrução é sempre um múltiplo de 4, o que significa que termina sempre em 00 em binário).
 - Assim o número de bytes a adicionar ao PC é sempre um múltiplo de 4 de forma a respeitar o alinhamento.
 - Então podemos especificar o `immediate` em termos de words.
 - ◆ Com este ajuste passamos a poder dar saltos de $\pm 2^{15}$ words a partir do PC (ou $\pm 2^{17}$ bytes), sendo possível lidar com loops 4 vezes maiores.

Branches e endereçamento relativo (5/5)

- ◆ Cálculo de saltos em Branches :

- Se não houver salto:

$$PC = PC + 4$$

$PC+4$ = “byte address” da próxima instrução

- Se houver salto:

$$PC = (PC + 4) + (\text{immediate} * 4)$$

- Observações

- Immediate especifica o número de words a saltar, o que é o mesmo que dizer o número de instruções.
 - Immediate pode ser um número positivo ou negativo.

Exemplo de Branch

- ◆ Código MIPS:

```
Loop:    beq    $9, $0, End
          add    $8, $8, $10
          addi   $9, $9, -1
          j      Loop
```

End:

- ◆ beq branch tem formato I:

opcode = 4

rs = 9

rt = 0

immediate = 3 (número de instruções a saltar)

Cuidado: o que aconteceria se tivesse

addi \$9, \$9, 0xFFFF ?

Questões PC-addressing

- ◆ O valor no campo do branch altera se movermos o código?
- ◆ O que acontece se o destino estiver a mais de 2^{15} instruções do branch?
- ◆ Por que razão necessitamos de modos diferentes de endereçamento (diferentes formas de formar o endereço de memória)? Por que não apenas um?

Instruções formato J (1/4)

- ◆ No caso dos branches, partimos do princípio de que o salto nunca seria muito distante. Isto permitiu a codificação em instruções formato I usando endereçamento relativo a partir do valor corrente de PC.
- ◆ No entanto, no caso de saltos incondicionais (`j` e `jal`), podemos querer saltar para qualquer lugar na memória.
- ◆ Neste caso deveríamos ser capazes de especificar um endereço de 32 bits.
- ◆ Infelizmente é impossível colocar numa instrução com o tamanho de uma word um opcode de 6 bits e um endereço de 32 bits.

Instruções formato J (2/4)

- Este tipo de instruções tem dois “campos” com o seguinte tamanho:

6 bits

26 bits

- Os nomes dos campos são:

opcode

target address

- Ideia chave**

- Manter o campo de **opcode** idêntico ao formato R e formato I por razões de consistência.
- Colapsar todos os outros campos para arranjar o máximo de espaço possível para colocar o endereço.

Instruções formato J (3/4)

- ◆ Para já conseguimos acomodar 26 bits de um endereço de 32-bits.
- ◆ Otimização:
 - Como a memória está alinhada podemos usar o mesmo truque que usámos para as instruções I: o campo é interpretado em termos de número de words em vez de bytes.
 - Desta forma conseguimos “cobrir” uma região de 2^{28} bytes de memória.

Instruções formato J (4/4)

- ◆ Assim conseguimos especificar 28 bits do endereço de 32-bits
- ◆ O que fazer quanto aos 4 bits que faltam?
 - Na prática consideramos que os 4 bits mais significativos de PC se mantêm, e a instrução só especifica os 28 menos significativos.
 - Tecnicamente isto significa que não podemos saltar para qualquer sítio da memória. No entanto esta solução permite resolver 99.999...% das situações reais
 - **Repare que conseguimos lidar com blocos de memória até 256 MB**
- ◆ Nos casos em que é necessário especificar um endereço de 32 bits temos que o colocar num registo e usar a instrução `j r` (este jump é uma instrução de tipo R)

Concluindo ...

- ◆ MIPS Machine Language Instruction:
cada instrução é representada por uma word de 32 bits

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt		immediate	
J	opcode		target address			

- ◆ Os branches usam endereçamento relativo a partir do valor atual de PC; os jumps usam endereçamento absoluto.
- ◆ A *desassemblagem* é possível se começarmos por fazer a descodificação do campo opcode (a ver).

QUIZ

Imagine que tem dois ficheiros com código fonte em C. Compile-os independentemente e depois faça a linkagem dos códigos objetos para gerar um executável.

- A. As instruções Jump não são alteradas na linkagem.
- B. As instruções Branch não são alteradas na linkagem.
- C. Nós já temos todas as ferramentas necessárias para sermos capazes de gerar o código C original a partir do binário!

	ABC
0 :	FFF
1 :	FFT
2 :	FTF
3 :	FTT
4 :	TFF
5 :	TFT
6 :	TTF
7 :	TTT

QUIZ

◆ Quais das seguintes instruções são **MAL** e quais são **TAL**?

- i. addi \$t0, \$t1, 40000
- ii. beq \$s0, \$s1, Exit
- iii. sub \$t0, \$t1, 1

	ABC
1 :	MMM
2 :	MMT
3 :	MTM
4 :	MTT
5 :	TMM
6 :	TMT
7 :	TTM
8 :	TTT

Para saber mais ...

- ◆ P&H - Capítulos 2.4, 2.9 e 2.10
- ◆ Anexo A.17



Introdução ao MIPS

- Correr um Programa -



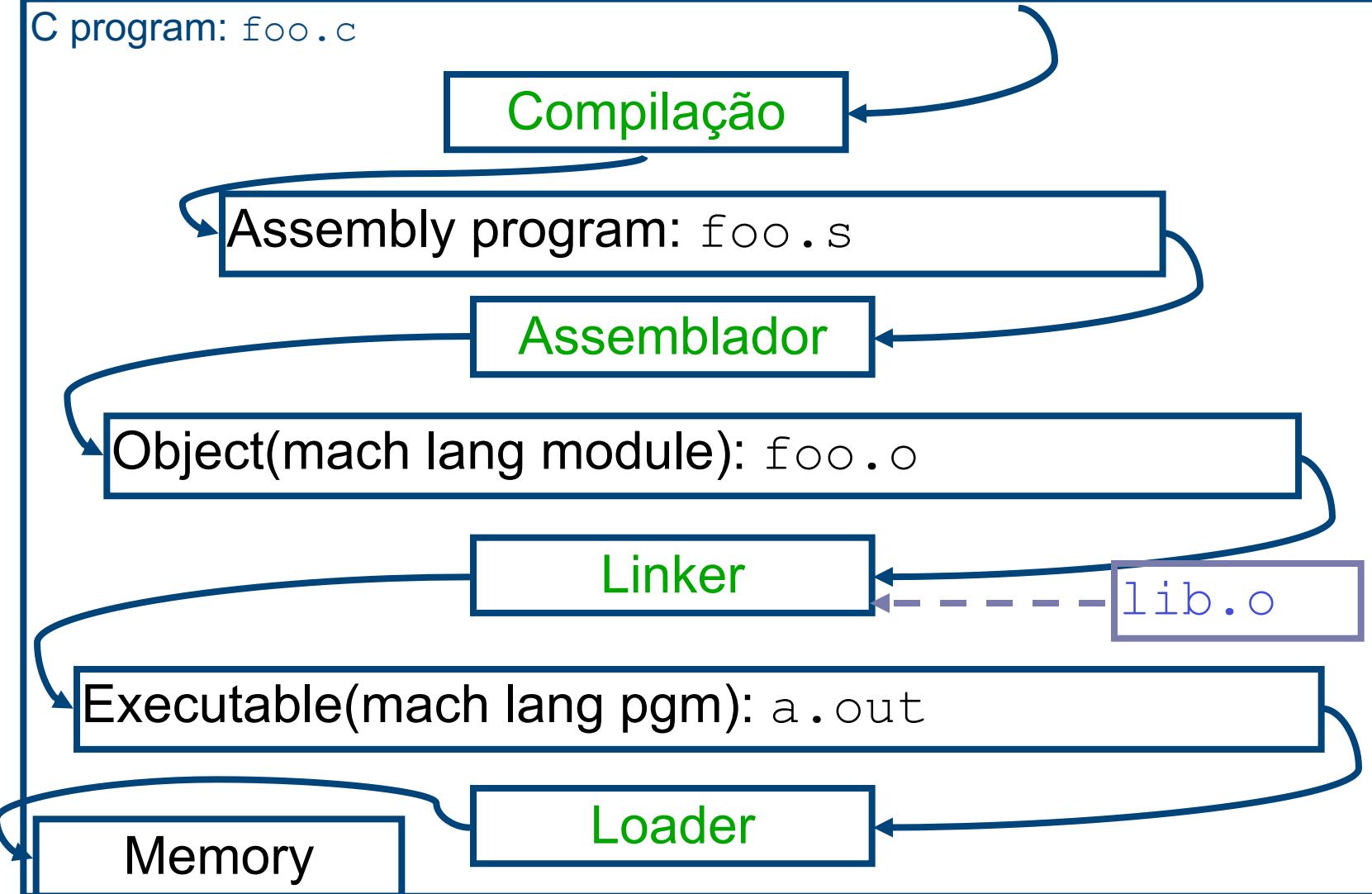
Compilação, Assemblagem,
Linkagem e Carregamento

Sistemas de Microprocessadores 2021/2022

Revisão

- ◆ Podemos fazer a desassemblagem de instruções máquina começando por interpretar o campo de opcode.
 - Depois de sabermos a instrução (add, lw, etc), passamos a conhecer o seu formato e podemos facilmente decompô-la nos seus campos.
 - Será que é possível gerar o código C a partir do binário?
- ◆ O Assemblador expande o conjunto de instruções máquina (TAL) com pseudo-instruções (MAL)
 - Só o TAL é que tem um paralelo em binário
 - A tarefa do assemblador é traduzir de MAL para TAL, e depois de TAL para binário
 - O assemblador utiliza o registo reservado \$at
 - O MAL torna muito mais fácil a tarefa do programador de escrever código MIPS.

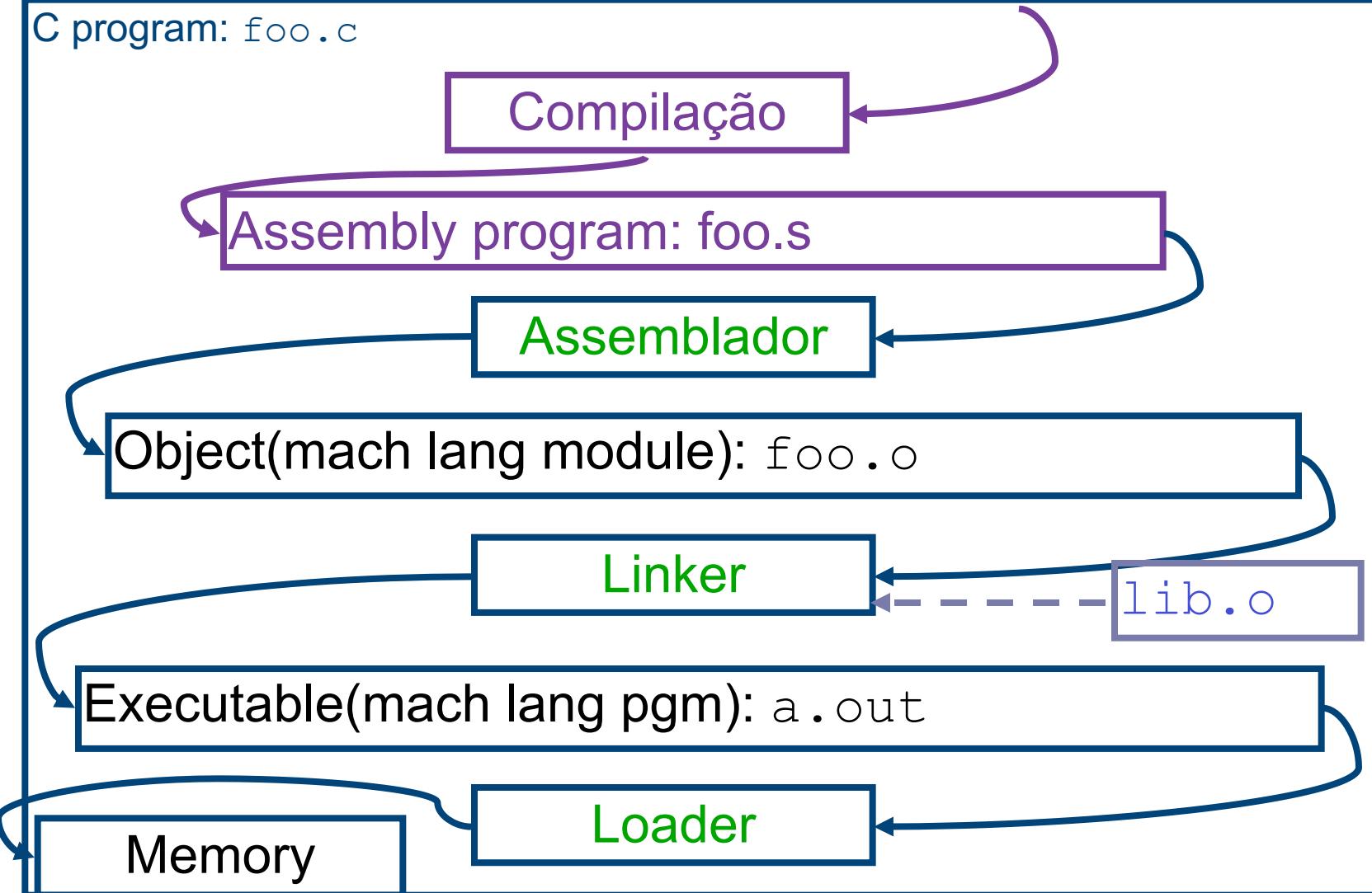
Tradução: Do código fonte ao executável



Compilação

- ◆ Input: Código fonte escrito numa linguagem de alto nível
(e.g., C, Java como `foo.c`)
 - ◆ Output: Código em linguagem assembly
(e.g., `foo.s` para o MIPS)
 - ◆ Nota: O output *pode* conter pseudo-instruções
 - ◆ Pseudo-instruções: instruções que o assemblador comprehende mas que não fazem parte do “instruction set” do processador. Por exemplo
 - `move $s1, $s2` ⇒ `add $s1, $s2, $zero`

Em que etapa estamos?



Assemblagem

- ◆ Input: Código em linguagem assembly
(e.g., `foo.s` para o MIPS)
- ◆ Output: Código objeto, tabelas
(e.g., `foo.o` para o MIPS)
- ◆ Lê e utiliza Diretivas
- ◆ Substitui Pseudo-instruções (MAL para TAL)
- ◆ Produz código máquina
- ◆ Cria Ficheiro de Código Objeto

Diretivas do Assemblador (p. A-51 a A-53)

- ◆ Dá indicações ao assemblador, mas não é traduzido em instruções máquina
 - .text: Colocar o que vem a seguir no segmento de texto do utilizador (a ser traduzido em código máquina)
 - .data: Colocar o que vem a seguir no segmento de dados do utilizador
 - .globl sym: declarar `sym` como “label” global que pode ser referenciado a partir de outros ficheiros
 - .asciiz str: Armazenar a string `str` em memória terminada por null
 - .word w1...wn: Armazenar as n quantidades de 32-bit em words sucessivas de memória

Substituição de Pseudo-Instruções

- ♦ O assemblador não só considera como pseudo-instruções, instruções que manifestamente não fazem parte do IS, como retifica variações cujo sentido é claro.

Pseudo:

subu \$sp,\$sp,32

sd \$a0, 32(\$sp)

mul \$t7,\$t6,\$t5

addu \$t0,\$t6,1

ble \$t0,100,loop

la \$a0, str

Real:

addiu \$sp,\$sp,-32

sw \$a0, 32(\$sp)

sw \$a1, 36(\$sp)

mult \$t6,\$t5

mflo \$t7

addiu \$t0,\$t6,1

slti \$at,\$t0,101

bne \$at,\$0,loop

lui \$at, left(str)

ori \$a0,\$at,right(str)

Geração de Código Máquina (1/3)

- ◆ Casos Simples
 - Instruções aritméticas e lógicas (add, sub, sll, or, etc)
 - Toda a informação necessária está codificada na própria instrução
- ◆ E quanto aos “branches” condicionais?
 - Salto relativo ao valor do PC
 - Só podemos saber o tamanho real do salto relativo, depois de as pseudo-instruções terem sido substituídas
- ◆ No caso dos “branches” a assemblagem requer duas passagens

Geração de Código Máquina (2/3)

“Forward Reference” problem

- As instruções de “branch” podem fazer referência a “labels” que estão à frente ou atrás no código

```
          or      $v0,$0,$0  
L1:      slt     $t0,$0,$a1  
          beq     $t0,$0,L2  
          addi    $a1,$a1,-1  
          j       L1  
  
L2:      add     $t1,$a0,$a1
```

- A tradução para código máquina da instrução “beq” é feita em 2 passagens
 - A primeira passagem determina a posição do label
 - A segunda passagem usa a posição do label para fazer a tradução

Geração de Linguagem Máquina (3/3)

- ◆ E quanto aos jumps (`j` e `jal`)?
 - Os jumps funcionam em termos de **endereços absolutos**.
 - Só é possível gerar a instrução máquina depois de se saber a posição do label em memória (o salto não é relativo)
 - Isto só pode ser resolvido depois da linkagem
- ◆ E quanto às referências a dados?
 - `la` é desdobrado num `lui` e `ori`
 - Estes precisam de saber o endereço de 32 bits dos dados ... (mesmo problema que os jumps)
- ◆ Como estes endereços só se sabem depois da assemblagem, precisamos de criar duas tabelas ...

Tabelas

◆ Tabela de Símbolos

- Lista os “items” do “ficheiro .o” que podem ser referenciados deste ou de outros “ficheiros .o”.
- Que items são estes?
 - **Labels:** e.g. chamada de funções
 - **Dados:** qualquer entidade da secção .data; variáveis que podem ser acedidas a partir de outros ficheiros

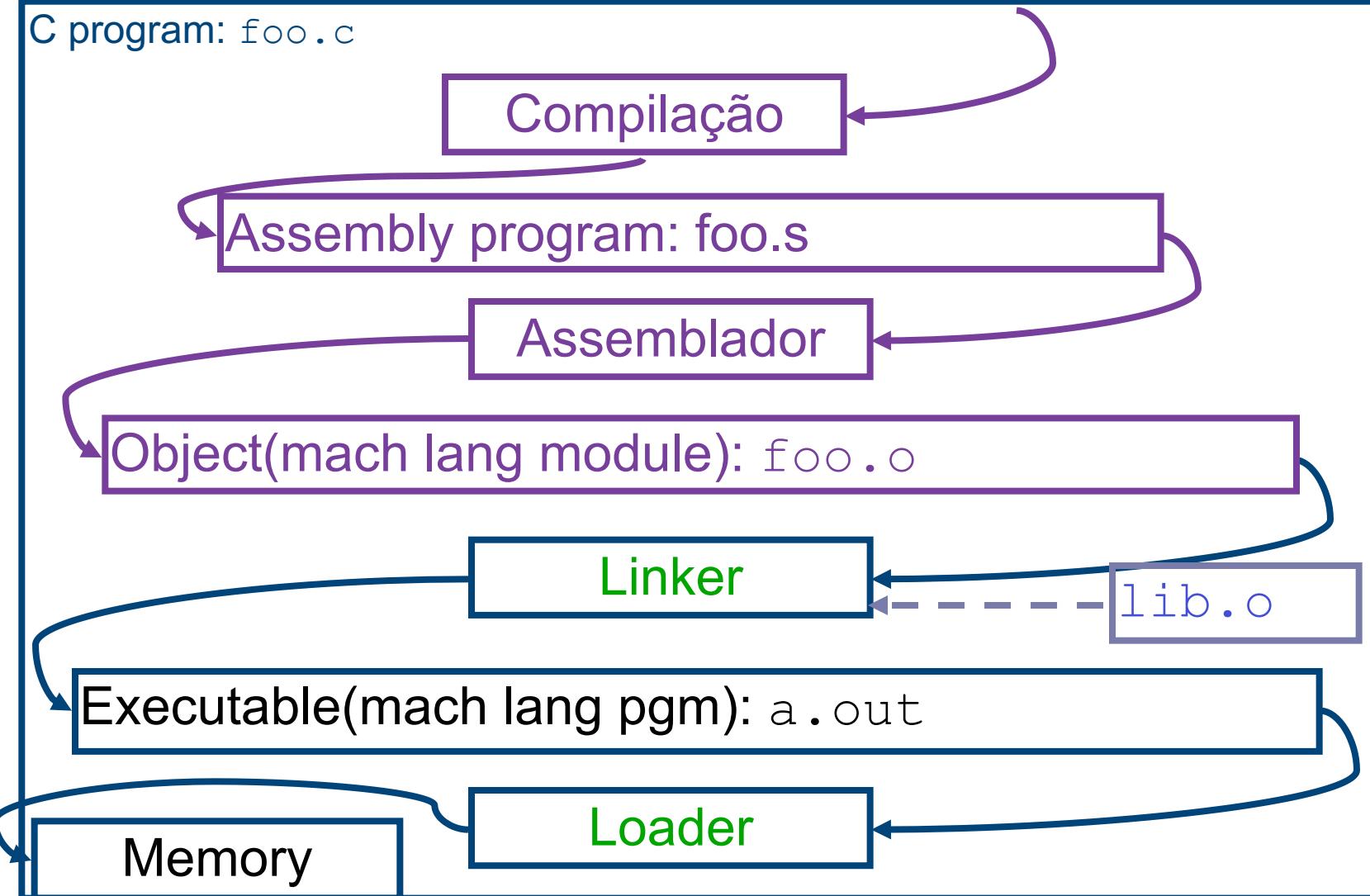
◆ Tabela de Realocação

- Lista de “items” que o “ficheiro .o” referencia e do qual não tem o endereço porque são externos (estão noutra ficheiro) ou serão resolvidos em “runtime”.
 - **Os “labels” usados nos j ou ja1**
 - internos
 - externos (incluindo ficheiros .lib)
 - **Dados**
 - Por exemplo a instrução la

Formato dos ficheiros .o (código objeto)

- ◆ **Cabeçalho:** posição e tamanho dos diferentes componentes do ficheiro objeto.
- ◆ **Segmento de texto:** código máquina
- ◆ **Segmento de dados:** representação binária dos dados e estruturas declarados no código fonte (normalmente declarações globais)
- ◆ **Tabela de realocação:** identifica as linhas de código onde há endereços a ser resolvidos
- ◆ **Tabela de símbolos:** lista de “labels” internos que podem ser referenciados, quer a partir do ficheiro, quer a partir de ficheiros externos.
- ◆ **Informação de debug:** (lembre-se da flag do gcc)
- ◆ Um formato standard é o ELF
http://www.skyfree.org/linux/references/ELF_Format.pdf

Em que etapa estamos?



Linker (1/3)

- ◆ Input: ficheiros código objeto, tabelas (e.g., `foo.o`, `libc.o` para o MIPS)
- ◆ Output: Código executável (e.g., `a.out` para MIPS)
- ◆ Combina vários ficheiros (`.o`) num único executável (“linking”)
- ◆ A técnica permite a compilação separada de diferentes ficheiros
 - Alterações num ficheiro fonte não requerem a recompilação de todo o programa (lembra-se do makefile?)
 - O código fonte do Windows NT tem > 40 M linhas de código!

Linker (2/3)

.o file 1

text 1

data 1

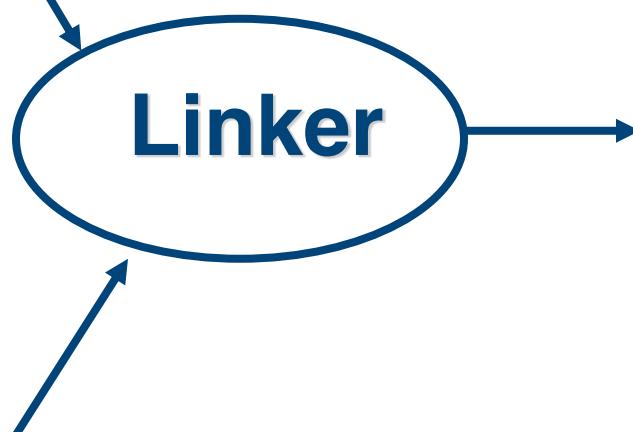
info 1

.o file 2

text 2

data 2

info 2



a.out

Relocated text 1

Relocated text 2

Relocated data 1

Relocated data 2

Linker (3/3)

- ◆ Passo 1: Concatenação dos segmentos de texto de cada ficheiro .o
- ◆ Passo 2: Juntar os segmentos de dados de cada ficheiro .o e concatená-los com o segmento de texto
- ◆ Passo 3: Resolver as referências
 - Ver as tabelas de realocação e resolver cada entrada
 - Definir os endereços absolutos em relação ao início do programa

Tipos de Endereçamento

- ◆ Endereçamento em relação ao PC (beq, bne): não é usada realocação
- ◆ Endereçamento absoluto (j, jal): realocação sempre
- ◆ Referências externas (normalmente jal): realocação sempre
- ◆ Referência a dados (normalmente lui e ori): realocação sempre

Endereçamento Absoluto no MIPS

- ◆ Quais as instruções que precisam de realocação de endereços?
 - J-format: jump, jump and link

j/jal	xxxxx
-------	-------

- Loads e stores de variáveis na zona estática, referenciadas em relação ao global pointer

lw/sw	\$gp	\$x	address
-------	------	-----	---------

- ◆ E quanto aos branches condicionais?

beq/bne	\$rs	\$rt	address (label)
---------	------	------	-----------------

- Como o endereçamento é feito em relação ao PC, as referências relativas mantêm-se mesmo que o código mude de sítio

Resolver Referências (1/2)

- ◆ O Linker *assume* que a primeira palavra do primeiro segmento de texto está no endereço 0x00000000.
(Quando estudarem o mecanismo de memória virtual voltarão a falar disto)
- ◆ O Linker sabe:
 - O tamanho do segmento de texto e dados
 - A ordem e posição dos segmentos de texto e dados
- ◆ O Linker calcula com base nisto:
 - O endereço absoluto de cada label associado aos jumps (internos e externos) bem como cada bloco de dados que é referenciado

Resolver Referências (2/2)

- ◆ Para resolver as referências:
 - Procurar a referência (dados ou label) na tabela de símbolos
 - Se a referência não for encontrada, procurar nos ficheiros das bibliotecas (e.g. stdio.h / printf)
 - Assim que o endereço absoluto for encontrado, preencher o código máquina de forma appropriada
- ◆ Output do linker: ficheiro executável contendo o segmento de texto, o segmento de dados, e o cabeçalho a ser lido pelo “loader” (ver a seguir)

Bibliotecas estáticas e dinâmicas

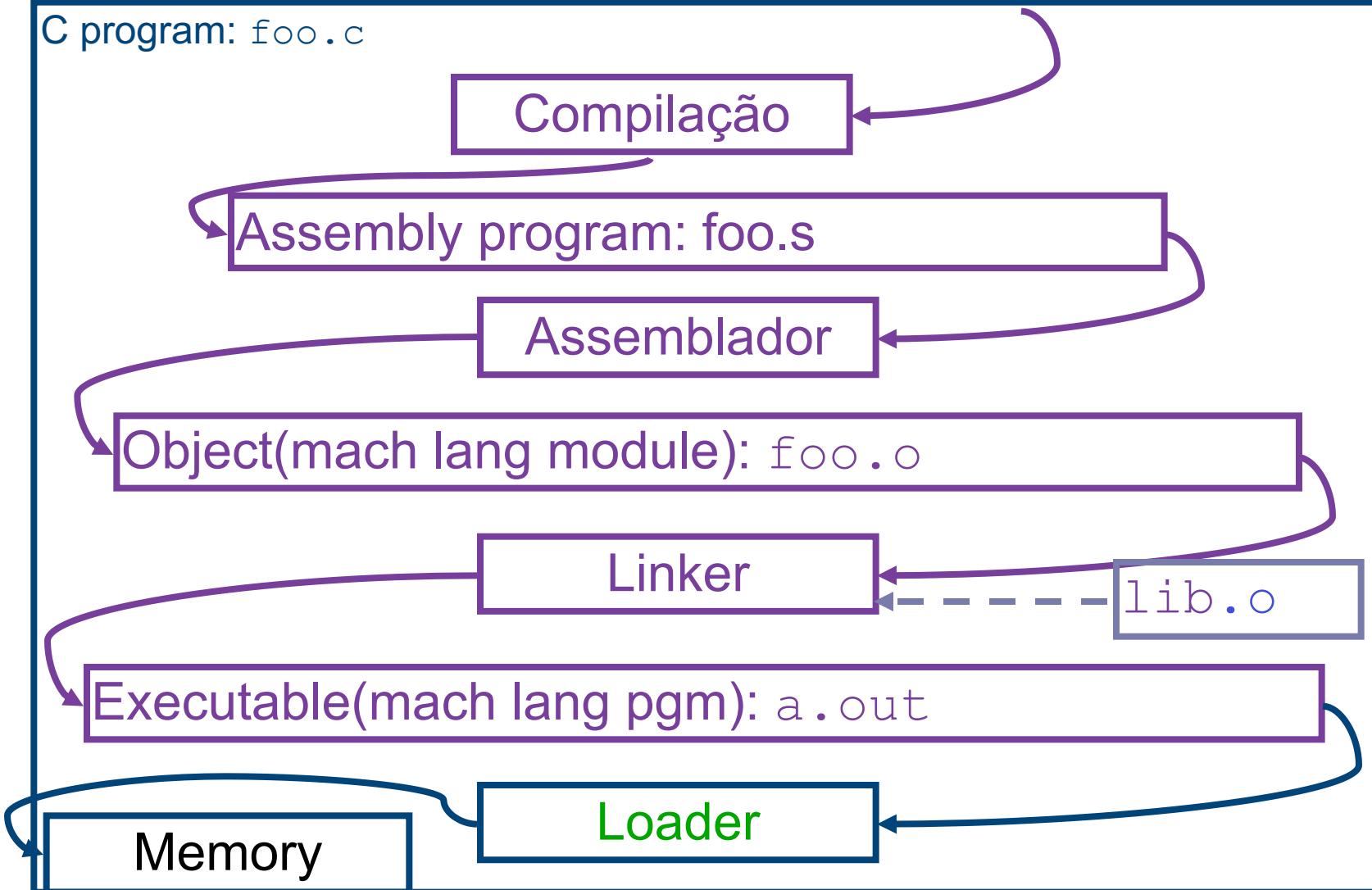
- ◆ Aquilo que descrevemos é a forma tradicional de fazer “linkagem”, normalmente conhecida por “linkagem estática”
 - No final a biblioteca é parte do executável. Assim, se posteriormente houver atualizações da biblioteca, o código criado não irá beneficiar das melhorias (teria que ser re-compilado a partir das fontes)
 - O executável inclui toda a biblioteca, mesmo que só uma pequena parte tenha sido utilizada (e.g. só a função `printf()`)
 - O executável é auto-contido.
- ◆ Uma alternativa é usar “bibliotecas dinâmicas” (DLL-dynamically linked libraries), que são muito comuns no Windows & UNIX

Dynamically linked libraries

en.wikipedia.org/wiki/Dynamic_linking

- ◆ Espaço em Disco / Tempo de Execução
 - + O executável requer menos espaço em disco
 - + Como o executável é mais pequeno, o seu envio/partilha é feito de forma mais rápida
 - + A execução de dois programas que partilhem a mesma biblioteca é mais rápida (ver o que é código re-entrante)
 - – Existe um “overhead” em runtime para ser feita a linkagem
- ◆ Upgrades
 - + Substituindo um ficheiro (libXYZ.so) faz o upgrade de todos os programas que usem XYZ.
 - – O executável não é auto-contido

Em que etapa estamos?



Loader (1/2)

- ◆ Input: código executável (e.g., a.out para MIPS)
- ◆ Output: (programa a correr)
- ◆ Os ficheiros executáveis estão armazenados em disco.
- ◆ Quando o executável é chamado, o “loader” tem a tarefa de o carregar em memória e iniciar a execução.
- ◆ Normalmente o “loader” é o próprio sistema operativo
 - O carregamento de programas é uma das tarefas do SO

Loader (2/2)

- ◆ O que é que o “loader” faz?
 - Lê o cabeçalho dos executáveis para determinar o tamanho e posição dos segmentos de texto e dados
 - Cria um espaço de endereçamento para o programa capaz de receber o texto, dados e pilha (e eventualmente “heap”)
 - Copia os dados e instruções do executável para o espaço de endereçamento criado
 - Copia os argumentos de chamada para a pilha (lembre-se do argc e argv no C)
 - Inicializa os registos do processador
 - A maioria dos registos são colocados a 0, mas o “stack pointer” fica a apontar para a 1^a frame livre
 - Salta para a rotina de “start-up” (ainda SO) que copia os argumentos do programa e faz o set do PC
 - Se a rotina principal (`main()`) regressar, a rotina de “startup” termina o programa com uma chamada a exit.

Exemplo: C \Rightarrow Asm \Rightarrow Obj \Rightarrow Exe \Rightarrow Run

Código fonte do programa em C : prog.c

```
#include <stdio.h>

int main (int argc, char *argv[] ) {
    int i, sum = 0;
    for (i = 0; i <= 100; i++)
        sum = sum + i * i;
    printf ("The sum of sq from 0 .. 100 is %d\n", sum);
}
```

“printf” está em “libc”

Compilação: MAL

```
.text
.align      2
.globl      main
main:
    subu $sp,$sp,32
    sw   $ra, 20($sp)
    sd   $a0, 32($sp)
    sw   $0, 24($sp)
    sw   $0, 28($sp)
loop:
    lw   $t6, 28($sp)
    mul $t7, $t6,$t6
    lw   $t8, 24($sp)
    addu $t9,$t8,$t7
    sw   $t9, 24($sp)
    ...
```

```
addu $t0, $t6, 1
sw   $t0, 28($sp)
ble $t0,100, loop
la   $a0, str
lw   $a1, 24($sp)
jal printf
move $v0, $0
lw   $ra, 20($sp)
addiu $sp,$sp,32
jr $ra
.data
.align      0
str:
    .asciiiz "The sum of
    sq from 0 .. 100 is
    %d\n"
```

Onde estão as 7 pseudo-instruções?

Compilação: MAL

```
.text
.align      2
.globl      main
main:
    subu $sp,$sp,32
    sw   $ra, 20($sp)
    sd   $a0, 32($sp)
    sw   $0, 24($sp)
    sw   $0, 28($sp)
loop:
    lw   $t6, 28($sp)
    mul $t7, $t6,$t6
    lw   $t8, 24($sp)
    addu $t9,$t8,$t7
    sw   $t9, 24($sp)
    ...
    ...
```

```
addu $t0, $t6, 1
sw   $t0, 28($sp)
ble $t0,100, loop
la   $a0, str
lw   $a1, 24($sp)
jal printf
move $v0, $0
lw   $ra, 20($sp)
addiu $sp,$sp,32
jr $ra
.data
.align      0
str:
    .asciiz "The sum of
    sq from 0 .. 100 is
    %d\n"
```

Assemblagem: Passo 1

Substituir pseudo-instruções, atribuir endereços

```
00 addiu $29,$29,-32
04 sw $31,20($29)
08 sw $4, 32($29)
0c sw $5, 36($29)
10 sw $0, 24($29)
14 sw $0, 28($29)
18 lw $14, 28($29)
1c multu $14, $14
20 mflo $15
24 lw $24, 24($29)
28 addu $25,$24,$15
2c sw $25, 24($29)
```

```
30 addiu $8,$14, 1
34 sw $8,28($29)
38 slti $1,$8, 101
3c bne $1,$0, loop
40 lui $4, 1.str
44 ori $4,$4,r.str
48 lw $5,24($29)
4c jal printf
50 add $2, $0, $0
54 lw $31,20($29)
58 addiu $29,$29,32
5c jr $31
```

Assemblagem: Passo 2

Criar tabelas de símbolos e realocação

◆ Tabela de símbolos

Label	address (in module)	type
main:	0x00000000	global text
loop:	0x00000018	local text
str:	0x00000000	local data

◆ Tabela de realocação

Address	Instr.	type	Dependency
0x00000040	lui		l.str
0x00000044	ori		r.str
0x0000004c	jal		printf

Assemblagem: Passo 3

Resolução de labels locais relativos a PC

```
00 addiu $29,$29,-32
04 sw $31,20($29)
08 sw $4, 32($29)
0c sw $5, 36($29)
10 sw      $0, 24($29)
14 sw      $0, 28($29)
18 lw       $14, 28($29)
1c multu   $14, $14
20 mflo    $15
24 lw       $24, 24($29)
28 addu   $25,$24,$15
2c sw       $25, 24($29)
```

```
30 addiu $8,$14, 1
34 sw $8,28($29)
38 slti     $1,$8, 101
3c bne     $1,$0, -10
40 lui      $4, 1.str
44 ori      $4,$4, r.str
48 lw       $5,24($29)
4c jal      printf
50 add     $2, $0, $0
54 lw       $31,20($29)
58 addiu   $29,$29,32
5c jr      $31
```

Assemblagem: Passo 4

- ◆ Gerar ficheiro código objeto (.o):
 - Representação binária
 - Segmento de texto (instruções),
 - Segmento de dados,
 - Tabelas de símbolos e realocação.
 - Utiliza endereços “dummy” para referências não resolvidas (endereços absolutos e items externos).

Segmento de Texto no ficheiro .o

0x000000	0010011110111101111111111111111100000
0x000004	101011111011111100000000000010100
0x000008	1010111110100100000000000000100000
0x00000c	1010111110100101000000000000100100
0x000010	1010111110100000000000000000110000
0x000014	1010111110100000000000000000111000
0x000018	1000111110101110000000000000111000
0x00001c	1000111110111000000000000000110000
0x000020	0000000111001100000000000000110001
0x000024	0010010111001000000000000000000001
0x000028	001010010000000100000000000011000101
0x00002c	1010111110101000000000000000111000
0x000030	000000000000000000001111000000100100
0x000034	00000011000011111100100000100001
0x000038	0001010000100000111111111110111
0x00003c	1010111110111001000000000000110000
0x000040	0011110000000100000000000000000000
0x000044	1000111110100101000000000000000000
0x000048	000011000001000000000000000011101100
0x00004c	0010010000000000000000000000000000
0x000050	1000111110111111000000000000101000
0x000054	0010011110111110100000000000100000
0x000058	000000111110000000000000000000001000
0x00005c	000000000000000000000000000000001000001

**Entradas
na
Tabela de
relocação**

Link passo 1: combina prog.o, libc.o

- ♦ Junta os segmentos de texto/dados
- ♦ Cria endereços absolutos de memória (o início do programa é 0x00000000)
- ♦ Modifica e concatena as tabelas de símbolos e realocação
- ♦ Tabela de símbolos

Label	Address
main:	0x00000000
loop:	0x00000018
str:	0x10000430
printf:	0x000003b0 ...

- ♦ Informação de realocação

Address	Instr.	Type	Dependency
0x00000040	lui		l.str
0x00000044	ori		r.str
0x0000004c	jal		printf ...

Link passo 2:

Edita endereços da tabela de realocação

(mostrado em TAL por razões de clareza, mas feito em binário)

```
00 addiu $29,$29,-32
04 sw $31,20($29)
08 sw $4, 32($29)
0c sw $5, 36($29)
10 sw      $0, 24($29)
14 sw      $0, 28($29)
18 lw       $14, 28($29)
1c multu   $14, $14
20 mflo    $15
24 lw       $24, 24($29)
28 addu   $25,$24,$15
2c sw       $25, 24($29)
```

```
30 addiu $8,$14, 1
34 sw $8,28($29)
38 slti     $1,$8, 101
3c bne     $1,$0, -10
40 lui      $4, 4096
44 ori      $4,$4,1072
48 lw       $5,24($29)
4c jal      812
50 add     $2, $0, $0
54 lw       $31,20($29)
58 addiu   $29,$29,32
5c jr      $31
```

Link passo 3:

◆ Executável

- Um único segmento de texto
 - Um único segmento de dados
 - Cabeçalho com informação da posição e tamanho de cada segmento (informação para o loader)

Para saber mais ...

- ◆ P&H - Capítulo 2.10
 - ◆ Ver anexos A.1 a A.4

