Instruções

Arquitetura de Computadores

Emilio Francesquini
e.francesquini@ufabc.edu.br

2021.Q1

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de Arquitetura de Computadores na UFABC.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- O conteúdo destes slides foi baseado no conteúdo do livro *Computer Organization And Design: The Hardware/Software Interface*, 5th Edition.

Instruções - A linguagem do computador

Conjunto de instruções

UFABC

- O Conjunto de Instruções (*en: instruction set*) é o conjunto das instruções de um computador.
- Diferentes computadores têm diferentes conjuntos de instruções.
  - Mas todos tem muitos aspectos em comum!
- Os primeiros computadores tinham conjuntos de instruções bem limitados
  - Isto simplificava a implementação do hardware do computador
- Muitos computadores modernos continuam a ter conjuntos de instruções simples

O Conjunto de Instruções MIPS

UFABC

- É o conjunto com o qual mais trabalharemos até o fim da disciplina
- É um computador (Stanford MIPS) que é comercializado pela empresa MIPS Technologies (www.mips.com).
  - Atualmente muito usado em roteadores, NAS, impressoras, câmeras...
  - Também foi o processador usado no Nintendo 64, Playstation 1 e 2, e PSP
  - E também nos carros Tesla e Volvo
- Por que o MIPS?
  - Ele representa o desing típico das arquiteturas mais modernas como ARMv7 e ARMv8
  - O livro contém uma página com muitos outros detalhes sobre o MIPS

Instruções Aritméticas

UFABC

- Vamos começar com uma das operações mais utilizadas: adição
- As instruções em MIPS prezam pela regularidade
  - Quase todas trabalham com 3 operandos
- Princípio de Design 1 : Simplicidade favorece regularidade
  - Regularidade torna a implementação do hardware mais simples
  - Simplicidade permite um desempenho superior a um custo inferior

Soma - add

- `add` soma dois números em registradores e armazena o resultado em um terceiro registador

```
1 add a, b, c # a = b + c
```

- `sub` funciona de maneira equivalente para a subtração

Soma - Exemplo

O seguinte código em C...

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

... se transforma em:

```
add t0, g, h  # temp t0 = g + h
add t1, i, j  # temp t1 = i + j
sub f, t0, t1 # f = t0 - t1
```

- Esse exemplo ainda é fictício: não existem registradores f, g, h, i,...

Operandos em registradores

UFABC

- Instruções aritméticas operam apenas em valores já contidos nos registradores
- MIPS tem 32 registradores de 32 bits cada um
  - Registradores são rápidos e são usados para dados acessados frequentemente
  - São numerados de 0 a 31
  - Uma palavra (*en: word*) no MIPS são 32 bits

UFABC

- Nomenclatura
  - `$t0, $t1, …, $t9` utilizados para variáveis temporárias
  - `$s0, $s1, …, $s7` utilizadas para variáveis salvas
- Princípio de Design 2 : menor é mais rápido
  - Considere o tempo de acesso para milhares de posições e o tempo de acesso para apenas algumas dezenas

Exemplo: operandos em registradores

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

- f armazenado em $s0
- g armazenado em $s1
- h armazenado em $s2
- i armazenado em $s3
- j armazenado em $s4

Código MIPS:

```
1 add $t0, $s1, $s2
2 add $t1, $s3, $s4
3 sub $s0, $t0, $t1
```

Operadores na memória

UFABC

- A memória principal pode ser usado para dados maiores ou compostos
  - *Structs*, arrays, estruturas dinâmicas (listas ligadas, árvores, ...)
- Contudo, operações aritméticas trabalham apenas nos registradores
- Precisamos então ferramentas para carregar os valores da memória nos registradores
  - Operação conhecida como load
- Também precisamos uma maneira de levar os valores dos registradores de volta à memória
  - Capacidade e quantidade de registradores são limitadas!
  - Operação conhecida como store

Mais alguns detalhes...

UFABC

- A memória é endereçada (*en: addressed*) por bytes
  - Cada endereço identifica um byte de 8 bits
- Palavras são alinhadas na memória
  - Logo, como cada palavra tem 32 bits, cada endereço é um múltiplo de 4
- MIPS é Big Endian
  - Isso significa que o byte mais significativo fica localizado no menor endereço da palavra
  - Em contrapartida, arquiteturas litte endian armazenam o byte menos significativo no menor endereço

Exemplo 1: Lendo operadores na memória

UFABC

- O MIPS oferece a instrução `lw` (*load word*) que recebe:
  - registrador destino
  - deslocamento (*en:offset*)
  - registrador base (*en: base register*)

```
1  g = h + A[8];
```

Onde `g` está em `$s1`, `h` em `$s2`, e endereço base de `A` em `$s3`

Código MIPS:

```
1  lw $t0, 32($s3)   # Atenção ao deslocamento
2  add $s1, $s2, $t0
```

Exemplo 2: Escrevendo operadores na memória

UFABC

De maneira semelhante, temos a instrução **sw** (*store word*) que armazena o valor do segundo operando no registrador apontado pelo primeiro.

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

Onde h em $s2, e endereço base de A em $s3

Código MIPS:

```
1  lw  $t0, 32($s3)  # Atenção ao deslocamento
2  add $t1, $s2, $t0
3  sw  $t1, 48($s3)  # Atenção ao deslocamento
```

Registradores *vs.* memória

UFABC

- Registradores são de acesso mais rápido que a memória
- Operar na memória exige o uso de *loads* e *stores*
  - Mais instruções precisam ser executadas
- Compiladores se esforçam para usar os registradores o máximo possível
  - Eles apenas vazam (*en: spill*) os registradores contendo as variáveis menos frequentemente utilizadas
  - O estudo da otimização do uso de registradores é importantíssimo e foi por muitos anos tópico de muitas pesquisas

Operadores Imediatos

UFABC

- **Operandos imediatos** (*en: immediate operands*) são úteis pois, frequentemente, trabalhamos com constantes no nosso código.
  - Pense `i++`, `x = 0`, ...
  - Otimizando casos comuns podemos ganhar simplicidade e desempenho
- A operação `add` tem uma versão com operando imediato chamada `addi`

```
1 addi $s3, $s3, 4
```

- Não temos a versão com imediatos de `sub`, mas `addi` aceita valores negativos. Então:

```
1 addi $s2, $s1, -1
```

- Princípio de Design 3 : Torne os casos comuns rápidos
  - Constantes são comuns e o operando imediato evita uma instrução de load

A constante Zero

- A constante zero é tão importante e comum que o MIPS reserva um registrador que sempre contem este valor: `$zero`
  - Esse registrador não pode ser sobrescrito
- É muito útil para efetuar operações muito comuns, por exemplo, copiar um valor entre um registrador e outro
  - Tradicionalmente essa operação é chamada de *move* ainda que ela não elimine o valor original

```
1 add $t2, $s1, $zero
```

Representando números com e sem sinal

UFABC

- Dado um número x de n bits temos que:

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \ldots + x_1 2^1 + x_0 2^0$$

- Um número com n bits tem uma faixa de valores entre 0 e 2^{n-1}
- Exemplo $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
  $= 0 + \ldots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Se utilizarmos 32 bits
  - 0 até +4.294.967.295

Representando números com sinal

Sinal Magnitude	Complemento de 1	Complemento de 2
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- Considerações: número de 0s, facilidade de operações

Olhando a soma: Sinal magnitude

UFABC

- Sinal magnitude
  - (+3) + (-2): $011_2 + 110_2 = 001_2 = +1_{10}$ ✓
  - 0 + (-1): 000_2 ou $100_2 + 101_2$
    - $000_2 + 101_2 = 101_2 = -1_{10}$ ✓
    - $100_2 + 101_2 = 001_2 = +1_{10}$ ✗
  - (-1) + (-2): $101_2 + 110_2 = 011_2 = +3_{10}$ ✗

Olhando a soma: Complemento de 1

- **(+3) + (-2)**: $011_2 + 101_2 = 111_2 = -0$ ✗
- **0 + (-1)**: $0_{10} = 000_2$ ou 111_2 e $-1_{10} = 110_2$
  - $000_2 + 110_2 = 110_2 = -1_{10}$ ✓
  - $111_2 + 110_2 = 101_2 = -2_{10}$ ✗
- **(-1) + (-2)**: $110_2 + 101_2 = 011_2 = +3_{10}$ ✗

Olhando a soma: Complemento de 2

- **(+3) + (-2)**: $011_2 + 110_2 = 001_2 = +1$ ✓
- **0 + (-1)**: $0_{10} = 000_2 + 111_2 = 111_2 = -1$ ✓
- **(-1) + (-2)**: $111_2 + 110_2 = 101_2 = -3_{10}$ ✓

Representando números com sinal

UFABC

- Sinal magnitude
  - Usa um bit como indicador de sinal (de que lado?)
  - Checagem de sinal
  - 2 zeros
- Complemento de 1
  - 2 zeros
  - Passo adicional para efetuar operações aritméticas
- Complemento de 2
  - Somas são sempre a mesma operação!
  - Simplificam o circuito!

UFABC

- Dado um número x de n bits temos que:

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \ldots + x_1 2^1 + x_0 2^0$$

  - Um número com n bits tem uma faixa de valores entre -2^{n-1} e $2^{n-1} - 1$
  - Exemplo $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1 \times 2^{31} + 1 \times 2^{30} + \ldots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
    $= -2.147.483.648 + 2.147.483.644 = -4_{10}$
  - Se utilizarmos 32 bits
    - -2.147.483.648 até +2.147.483.647

Números com sinal - Complemento de 2

UFABC

- O bit 31 indica o sinal
  - 1 negativo
  - 0 positivo
- $-(-2^{n-1})$ não é representável
- Números inteiros positivos em complemento de 2 são exatamente iguais aos seus colegas sem sinal
- Alguns números específicos:
  - 0: $0000\ 0000 \ldots 0000$
  - -1: $1111\ 1111 \ldots 1111$
  - Limite negativo: $1000\ 0000 \ldots 0000$
  - Limite positivo: $0111\ 1111 \ldots 1111$

- Há um truque para inverter o sinal de um número em complemento de 2:
  - Complemente e adicione 1
  - Complementar significa inverter os bits, 1s viram 0s e 0s viram 1s
  - Curiosidade: algumas calculadoras mecânicas funcionavam com aritmética de complemento de 9 e 10!

Qual é a ideia por trás?

$$x + \overline{x} = 1111\ 1111 \ldots 1111_2 = -1_{10}$$

$$\overline{x} + 1 = -x$$

- Exemplo: negar $+2_{10}$

$$+2 = 0000\ 0000 \ldots 0010_2$$

$$-2 = 1111\ 1111 \ldots 1101_2 + 1$$

$$1111\ 1111 \ldots 1110_2$$

UFABC

Tip

O nome complemento de 2 vem do fato de que a soma de um número de n bits e o seu negativo pode ser interpretado como 2^n (se visto como um número sem sinal). Logo, $2^n - x$ é o número negado ou o "complemento de 2"de x.

$(+2) + (-2) = 010_2 + 110_2 = 1000_2 = 8 = 2^3$

Complemento de 1, por outro lado, é dado por $2^n - x - 1$.

Extensão de sinal

UFABC

- Às vezes é interessante trabalhar apenas com uma parte do valor (um byte, um mas os registradores do MIPS armazenam 32 bits)
  - Nesses casos é preciso estender o sinal preservando o valor original do número
- O conjunto de instruções do MIPS forence algumas instruções que podem nos auxiliar
  - `addi`: pode estender o valor do imediato
  - `lb` (*load byte*) e `lh` (*load half-word*): estendem um byte ou meia-palavra carregados da memória
  - `beq` (*branch if equal*) e `bne` (*branch if not equal*): estendem o deslocamento

Extensão de sinal

UFABC

- A extensão de sinal funcional duplicando o dígito mais à esquerda
  - com 0s no caso de números sem sinal
- Exemplos 8 bits para 16 bits:
  - $+2 : 0000\ 0010 \rightarrow 0000\ 0000\ 0000\ 0010$
  - $-2 : 1111\ 1110 \rightarrow 1111\ 1111\ 1111\ 1110$

Representando instruções no computador

Representação das instruções

UFABC

- As instruções que vimos até agora são, todas, codificadas como um número binário
  - Comumente chamado de código de máquina (*en: machine code*)
- Instruções MIPS
  - São codificadas como palavras de 32 bits
  - Têm um pequeno número de formatos, códigos de operação (***opcode***), números de registradores, ...
  - Promove regularidade!
- Os registradores são numerados:
  - `$t0 - $t7` são os registradores `8 - 15`
  - `$t8 - $t9` são os registradores `24 - 25`
  - `$s0 - $s7` são os registradores `16 - 23`

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

- Campos das instruções (*en: fields*)
  - `op`: código da operação (opcode)
  - `rs`: número do primeiro registrador fonte
  - `rt`: número do segundo registrador fonte
  - `rd`: número do registrador de destino
  - `shamt`: quantidade de deslocamento (00000 por equanto)
  - `funct`: código da função (extensão do opcode)

Exemplo: R-format

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

```
1 add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add

0	17	18	8	0	32

000000	10001	10010	01000	00000	100000

$$00000010001100100100000000100000_2 = 02324020_{16}$$

UFABC

- Base 16
  - Formato compacto e prático
  - 4 bits por dígito hexadecimal

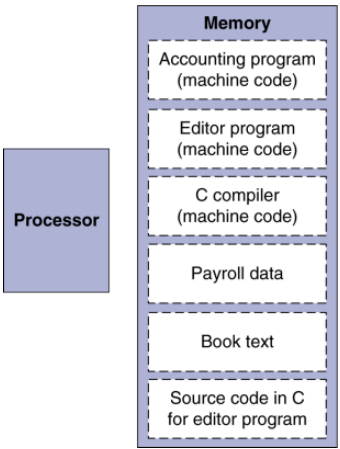
0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Exemplo: $eca8\ 6420$
  - $1110\ 1100\ 1010\ 1000\ 0110\ 0100\ 0010\ 0000$

op	rs	rt	constante ou endereço
6 bits	5 bits	5 bits	16 bits

- Utilizada para operações aritméticas com imediatos e para loads/stores
  - `rt`: número do registrador de origem ou destino
  - `constante`: -2^{15} até $2^{15} - 1$
  - `endereço`: offset adicionado ao registrador base em `rs`
- Design Principle 4: um bom design deve fazer bons compromissos
  - Formatos diferentes complicam o hardware, mas permitem que usemos apenas instruções de 32 bits
  - Mantém formatos tão próximos quanto possível

The BIG Picture



- Instruções são armazenadas como números binários na memória, assim como são os dados
- Programas podem operar em programas
  - Compiladores, linkers, ...
- Compatibilidade binária permite que programas executem em computadores diferentes
  - ISAs padronizadas

Operações lógicas

UFABC

- Instruções para manipulação de bits

Operação	C	Java	MIPS
Shift left	`<<`	`<<`	`sll`
Shift right	`>>`	`>>`	`srl`
Bitwise AND	`&`	`&`	`and`, `andi`
Bitwise OR	|	|	`or`, `ori`
Bitwise NOT	`~`	`~`	`nor`

- Muito úteis para máscaras para inserir ou retirar bits de uma palavra

UFABC

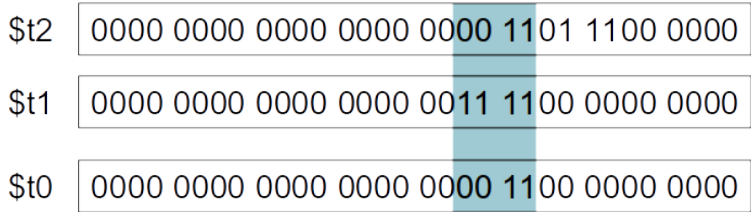
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- `shamt`: quantas posições devem ser deslocadas
- `sll`: *Shift left logical*
  - Desloca para a esquerda e completa com 0s
  - `sll` por *i* bits é o mesmo que multiplicar por 2^i
- `srl`: *shift right logical*
  - `srl` por *i* bits é o mesmo que dividir por 2^i. Atenção! Apenas para números sem sinal

Operações AND

- Útil para utilizar de máscaras
  - Seleciona alguns bits e limpa os demais

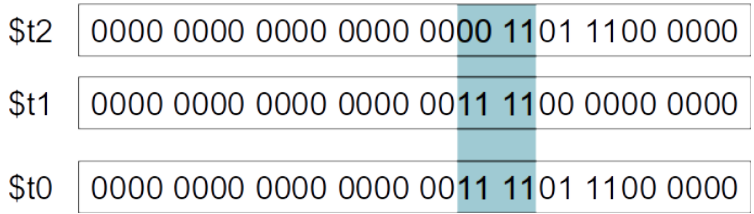
```
1 and $t0, $t1, $t2
```



Operações OR

- Útil para incluir alguns bits em uma palavra
  - Escreve 1 em alguns bits e mantém os demais intocados

```
1 or $t0, $t1, $t2
```



Operações NOT

- Útil para inverter os bits de uma palavra
  - Troca 0 por 1 e 1 por 0
- MIPS tem uma instrução **NOR** que leva 3 operandos (para manter regularidade) mas não tem uma operação NOT.
  - `a NOR b == NOT (a OR B)`

```
1 nor $t0, $t1, $zero # NOT
```

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

Operações condicionais

Operações condicionais

- Branch para uma instrução etiquetada (*en: labeled*) se a condição for verdadeira
  - Senão continua como se nada tivesse ocorrido
- `if (rs == rt)` branch para a instrução etiquetada `L1`

```
1 beq rs, rt, L1
```

- `if (rs != rt)` branch para a instrução etiquetada `L1`

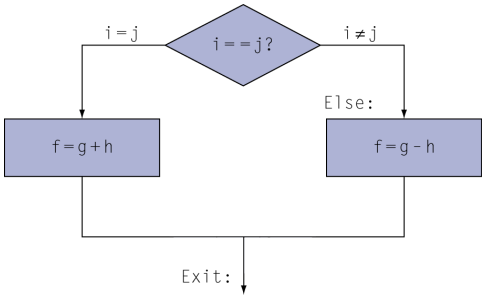
```
1 bne rs, rt, L1
```

- Para fazer um salto (*en: jump*) não condicinal

```
1 j L1
```

Compilando um if

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



Com `f, g, h, i, j` em `$s0, $s1, $s2, $s3, $s4` respectivamente.

Código MIPS compilado

```
1    # se (i != j) pula para Else
2    bne $s3, $s4, Else
3    add $s0, $s1, $s2
4    j Exit
5  Else:
6    sub $s0, $s1, $s2
7  Exit:
8    ...
```

Compilando laços

```
while (save[i] == k) i+= 1;
```

- Onde `i` está em `$s3`, `k` está em `$s5` e `save` está em `$s6`
- Código MIPS compilado

```
Loop:
    sll  $t1, $s3, 2
    add  $t1, $t1, $s6
    lw   $t0, 0($t1)
    bne  $t0, $s5, Exit
    addi $s3, $s3, 1
    j Loop
Exit: ...
```

Blocos básicos

UFABC

- Um bloco básico é uma sequência de instruções com
  - Nenhum branch (exceto no final)
  - Nenhum alvo de branches (exceto no início)
- Compiladores utilizam blocos básicos para otimização
- Um processador moderno é capaz de acelerar a execução de um bloco básico

Algumas operações condicionais a mais...

UFABC

- `slt`: (*set if less than*) escreve 1 no resultado caso a condição seja verdadeira
  - Escreve 0 caso contrário
- `if (rs < rt) rd = 1; else rd = 0;`

```
1 slt rd, rs, rt
```

- Há a versão com imediato `slti`
  - `if (rs < constante) rt = 1; else rt = 0;`

```
1 slti rt, rs, constant
```

- Útil para combinar com `beq` ou `bne`:

```
1 slt $t0, $s1, $s2 # if ($s1 < $s2)
2 bne $t0, $zero, L # branch para L
```

Sobre o design das instruções de branches

UFABC

- Por que não `blt`, `bge`, ...?
- O hardware para $<, \geq, \ldots$ é mais lento que para $=, \neq$
  - Combinado com branches, isso significa mais trabalho por instrução o que causaria um clock mais lento
  - Todas as demais instruções acabam sendo penalizadas
- `beq` e `bne` são os casos mais comuns.
  - Um bom compromisso entre complexidade e desempenho

Operações com sinal e sem sinal

- Comparações com sinal: `slt` e `slti`
- Comparações sem sinal: `sltu` e `sltui`
- Exemplo:

$$s0 = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111$$

$$s1 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001$$

```
1 slt $t0, $s0, $s1 # com sinal
```

- $-1 < +1 \rightarrow$ `$t0 = 1`

```
1 sltu $t0, $s0, $s1 # sem sinal
```

- $+4.294.967.295 > +1 \rightarrow$ `$t0 = 0`

Chamando procedimentos/funções

UFABC

- Passos necessários
  1. Colocar os parâmetros nos registradores
  2. Transferir o controle para o procedimento
  3. Obter espaço para o armazenamento das variáveis do procedimento
  4. Executar o procedimento
  5. Colocar o resultado em um registrador para devolver a quem chamou o procedimento
  6. Voltar para o local de onde o procedimento foi chamado

Padrão de uso dos registradores

UFABC

- `$a0 - $a3`: argumentos (registradores 4 - 7)
- `$v0, $v1`: valores de retorno (registradores 2 e 3)
- `$t0 - $t9`: temporárias
  - Podem ser sobrescritas pelo procedimento chamado
- `$s0 - $s7`: registradores salvos
  - Precisam ser salvos/restaurados pelo procedimento chamado
- `$gp`: (*global pointer*) ponteiro para a área de variáveis estáticas (globais) (registrador 28)
- `$sp`: (*stack pointer*) ponteiro para a pilha (registrador 29)
- `$fp`: (*frame pointer*) ponteiro para o quadro da pilha (registrador 30)
- `$ra`: endereço de retorno (registrador 31)

Instruções para chamada de procedimentos

- Chamada com *jump and link*: `jal`

```
1  jal ProcedureLabel
```

  - Endereço da instrução seguinte é colocada em `$ra`
  - Salta para o endereço do label

- Retornando da chamada de um procedimento `jr` (*jump register*)

```
1  jr $ra
```

  - Copia o `$ra` para o **program counter**
  - Também é útil para implementar funcionalidades como case/switch

Exemplo de uma função folha

```
1 int folha (int g, int h, int i, int j) {
2     int f;
3     f = (g + h) - (i + j);
4 }
```

- Argumentos passados em `$a0, $a1, $a2, $a3`
- `f` em `$s0` (logo é preciso salvar `$s0`)
- Resultado em `$v0$`

Exemplo de uma função folha

UFABC

Código MIPS:

```
1  folha:
2    addi $sp, $sp, -4 # salva $s0 na pilha
3    sw   $s0, 0($sp)
4    add  $t0, $a0, $a1 # Corpo do procedimento
5    add  $t1, $a2, $a3
6    sub  $s0, $t0, $t1
7    add  $v0, $s0, $zero # coloca o resultado
8    lw   $s0, 0($sp) # restaura o $s0
9    addi $sp, $sp, 4 # restaura pilha
10   jr $ra # retorna
```

Procedimentos não-folha

- São procedimentos que chamam outros procedimentos
- Para chamadas aninhadas, o ”chamador”precisa salvar na pilha
  - Seu endereço de retorno
  - Quaisquer argumentos e variáveis temporárias que ainda serão utilizadas
- Em seguida, após a chamada, restaurar a pilha

Exemplo de uma função não-folha

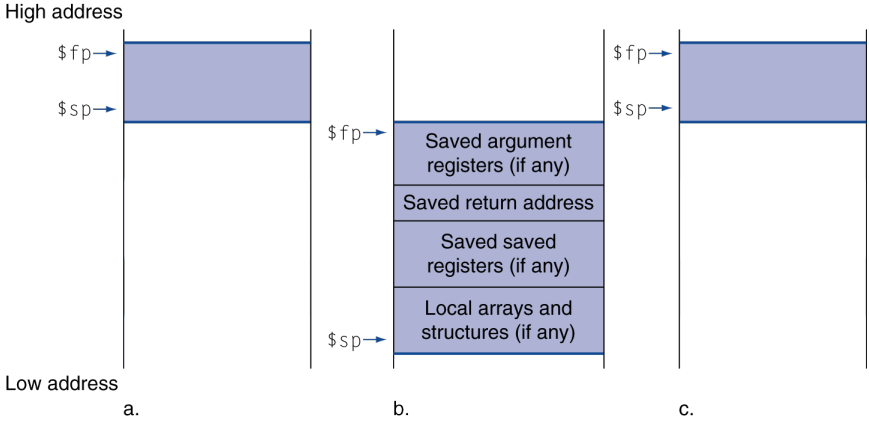
```
1 int fact (int n) {
2     if (n < 1)
3         return 1;
4     else
5         return n * fact (n - 1);
6 }
```

Fatorial - Versão MIPS

UFABC

```
1  fact:
2    addi $sp, $sp, -8   # Reserva espaço para 2 itens
3    sw   $ra, 4($sp)    # Salva o endereço de retorno
4    sw   $a0, 0($sp)    # Salva o argumento
5    slti $t0, $a0, 1    # Corpo da função, testa se n < 1
6    beq  $t0, $zero, L1
7    addi $v0, $zero, 1  # Se chegou aqui, resultado é 1
8    addi $sp, $sp, 8    # Restaura pilha, pop 2 itens
9    jr   $ra            # Retorna
10 L1:
11   addi $a0, $a0, -1   # decrementa n
12   jal  fact           # chamada recursiva
13   lw   $a0, 0($sp)    # restaura n original
14   lw   $ra, 4($sp)    # restaura $ra
15   addi $sp, $sp, 8    # Restaura pilha, pop 2 itens
16   mul  $v0, $a0, $v0  # muliplica pra pegar o resultado
17   jr   $ra            # retorna
```

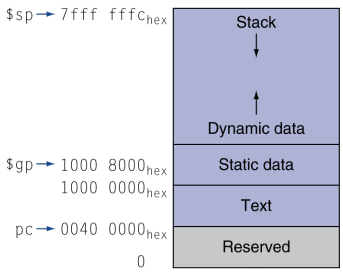
Dados locais na pilha



- Dados locais alocados pela função chamada
  - Por exemplo, variáveis automáticas em C
- *Procedure frame* (activation record) → Alguns compiladores utilizam para controlar o armazenamento de dados no stack

Leiaute da memória

- Seção de texto: código do programa
- Dados estáticos: variáveis static em C, constantes, etc.
- `$gp` inicializado e utilizado em conjunto com offsets
- Dados dinâmicos: heap
  - Usado por utilitários como `malloc/free` em C ou `new` em Java.
- Pilha: controle automático de armazenamento



Para seres humanos...

UFABC

- Para facilitar a comunicação com seres humanos definiu-se um padrão de mapeamento número $\rightarrow$ caracter
- Um dos padrões mais usados no mundo é o ASCII
  - 128 caracteres
    - 95 gráficos, 33 de controle
- Latin-1: 256 caracteres
  - ASCII + 96 caracteres gráficos
- Unicode: conjunto de caracteres de 32 bits
  - Usado em Java, C++, e quase todas as linguagens modernas
  - Contém a maior parte dos símbolos dos alfabetos/abugidas/... do mundo (incluindo símbolos importantíssimos como 🤦)
  - UTF-8 e UTF-16: maneiras de codificar com comprimento variável

UFABC

- Para usarmos caracteres sem desperdiçar boa parte da palavra do hardware, precisamos ser capazes de trabalhar com partes dela
- Poderíamos utilizar operações bitwise
- Mas como é muito comum, MIPS oferece algumas instruções especiais

Operações sobre Byte/Halfword

UFABC

- `lb` e `lh`

```
1 lb rt, offset(rs)
2 lh rt, offset(rs)
```

  - Efetua extensão de sinal para 32 bits em `rt`

- `lbu` e `lhu`

```
1 lbu rt, offset(rs)
2 lhu rt, offset(rs)
```

  - Efetua extensão com 0s para 32 bits em `rt`

- `sb` e `sh`
  - Guarda apenas o byte/halfword mais à direita de rt

```
1 sb rt, offset(rs)
2 sh rt, offset(rs)
```

UFABC

```
1  void strcpy (char x[], char y[]) {
2      int i;
3      i = 0;
4      while ((x[i]=y[i]) != '\0')
5          i += 1;
6  }
```

- Endereços de `x` e `y` em `$a0` e `$a1`
- `i` em `$s0`

String Copy - Versão MIPS

UFABC

```
1  strcpy:
2    addi $sp, $sp, -4       # Ajusta pilha para 1 item
3    sw   $s0, 0($sp)        # Salva $s0
4    add  $s0, $zero, $zero  # i = 0
5  L1:
6    add  $t1, $s0, $a1      # end de y[i] em $t1
7    lbu  $t2, 0($t1)        # $t2 = y[i]
8    add  $t3, $s0, $a0      # end de x[i] in $t3
9    sb   $t2, 0($t3)        # x[i] = y[i]
10   beq  $t2, $zero, L2     # sai do laço se y[i] == 0
11   addi $s0, $s0, 1        # i = i + 1
12   j    L1                 # itera
13 L2:
14   lw   $s0, 0($sp)        # restaura $s0
15   addi $sp, $sp, 4        # restaura pilha
16   jr   $ra                # retorna
```

Endereçamento no MIPS para imediatos de 32 bits e endereços

Constantes de 32-bits

UFABC

- A maior parte das constantes é pequena
  - 16 bits para armazenar o imediato é normalmente suficiente
- Para quando tivermos uma constante maior, de 32 bits

```
1  lui rt, constant
```

  - Copia 16 bits da constante para a parte esquerda de rt
  - Limpa 16 bits à direita de rt (preenche com zeros)

```
1  lui $s0, 61
2  ori $s0, $s0, 2304
```

Endereçamento de branches

- Instruções de branches especificam
  - Opcode, 2 registradores, endereço de destino
- A maior parte dos branches são próximos
  - Tanto para frente quanto para trás

op	rs	rt	constante ou endereço
6 bits	5 bits	5 bits	16 bits

- Endereço relativo ao PC
  - Endereço destino = PC + offset x 4
  - PC já é aumentando de 4 por esta linha

Endereçamento para saltos

- Saltos (`j` e `jal`) podem ter como destino qualquer local dentro do segmento de texto
  - Codifica o endereço completo na instrução

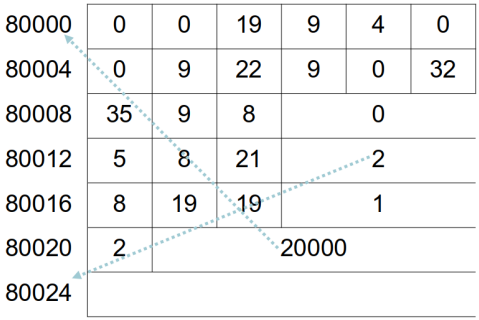
op	constante ou endereço
6 bits	26 bits

- Endereçamento de saltos (Pseudo)diretos
  - Endereço destino = $PC_{31...28}$: (endereço * 4)

Exemplo de destinos

- Código do exemplo anterior
- Assuma que **Loop** esteja na posição 80000

```
1  Loop: sll $t1, $s3, 2
2        add $t1, $t1, $s6
3        lw  $t0, 0($t1)
4        bne $t0, $s5, Exit
5        addi $s3, $s3, 1
6        j Loop
7  Exit: ...
```

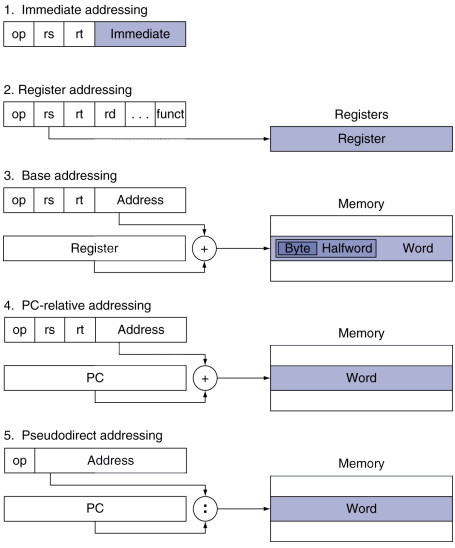


UFABC

- Se o branch estiver longe demais para codificar com 16 bits, o assembler reescreve o código
- Exemplo:

```
1     beq $s0,$s1, L1
2                       # Se transforma em
3     bne $s0,$s1, L2
4     j L1
5  L2:          ...
```

Resumo dos modos de endereçamento



Decodificando a linguagem de máquina

Exemplo

Queremos decodificar

0x00af8020

1. Convertemos de hexa para binário

```
0x 0    0     a    f    8    0    2    0
   0000 0000  1010 1111 1000 0000 0010 0000
   3322 2222  2222 1111 1111 11
   1098 7654  3210 9876 5432 1098 7654 3210
```

Tabela de decodificação

op(31:26)								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						

Veja a tabela completa no capítulo 2 do [PH]. Figura 2.19 da 5ª edição.

Formatos de instruções

UFABC

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

2. Agora que sabemos que a instrução é do tipo R rearranjamos para ver os campos da instrução mais claramente

```
opcode rs    rt    rd    shamt funct
000000 00101 01111 10000 00000 100000
332222 22222 21111 11111 1
109876 54321 09876 54321 09876 543210
```

O funct da nossa instrução é `100000`

op(31:26)=000000 (R-format), funct(5:0)								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

Exemplo

UFABC

```
opcode rs    rt    rd    shamt funct
000000 00101 01111 10000 00000 100000

É um ADD!

ADD rd rs rt
ADD 10000 00101 01111 # Binário
ADD 10    5     f     # Hexa
ADD 16    5     15    # Decimal
```

Usando a tabela de mapeamento de registradores

Procuramos os registradores 16, 5 e 15:

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

Que são os registradores `$s0 $a1, $t7`

Exemplo

A instrução é:

```
1 ADD  $s0, $a1, $t7
```

UFABC

MIPS Reference Data

CORE INSTRUCTION SET

NAME, MNEMONIC		FORMAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	$0 / 20_{hex}$
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		$0 / 21_{hex}$
And	and	R	R[rd] = R[rs] & R[rt]		$0 / 24_{hex}$
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}

Paralelismo e instruções: Sincronização

UFABC

- Dois processos que compartilham a mesma área de memória
  - P1 escreve e em seguida P2 lê
  - Condição de corrida se P1 e P2 não sincronizarem
    - Resultado depende da ordem em que ocorrem os acessos
- É necessário suporte de hardware
  - Operações atômicas de escrita/leitura
  - Garante que nenhum acesso é feito na posição de memória entre a leitura e a escrita
- Poderia ser uma simples instrução
  - Por exemplo, swap atômico entre registrador e memória
  - Ou um ”par atômico”de instruções

- `ll` *load linked*

```
1 ll rt, offset(rs)
```

- `sc` *store conditional*

```
1 sc rt, offset(rs)
```

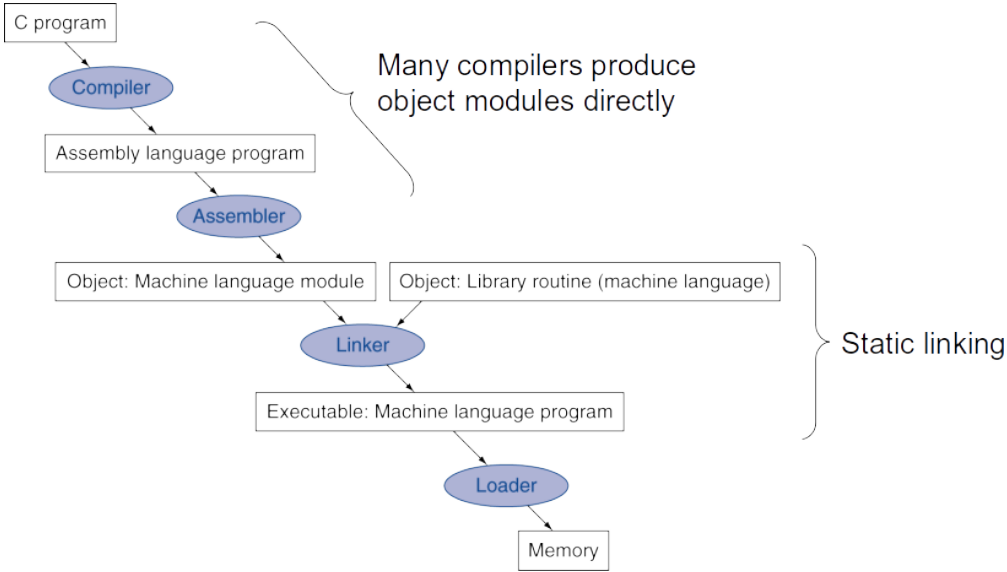
  - Apenas executa com sucesso caso a localização não tenha sido modificada desde o último `ll`
    - Devolve 1 no `rt`
  - Falha caso a posição tenha sido modificada
    - Devolve 0 no `rt`

UFABC

- Exemplo: troca atômica (test & set)

```
1  Try: add $t0, $zero, $s4 # Copia o endereço a ser trocado
2       ll  $t1, 0($s1)     # load linked
3       sc  $t0, 0($s1)     # store conditional
4       beq $t0, $zero, Try # branch se o store falhou
5       add $s4, $zero, $t1 # store do valor em $s4
```

Traduzindo e iniciando um programa


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
Many compilers produce object modules directly
Static linking

- O assembler pode fornecer algumas pseudo-instruções
- Normalmente há uma correspondência 1 para 1, nas pseudo-instruções a imaginação do assembler é o limite!

```
1 move $t0, $t1
2 blt $t0, $t1, L
```

```
1 add $t0, $zero, $t1
2 slt $at, $t0, $t1
3 bne $at, $zero, L
```

- `$at` (registrador 1): reservada para o assembler usar como variável temporária

Produzindo um Object Module

UFABC

- Assembler (ou o compilador) traduz o programa para instruções de máquina
- Fornece as informações para construir um programa completo a partir das peças
  - Cabeçalho: descreve o conteúdo do object module
  - Segmento de texto: instruções traduzidas
  - Segmento de dados estáticos: espaço de dados alocados que existem durante a execução completa do programa
  - Relocation info: usado para conteúdos que dependem da posição absoluta do programa (após carregado)
  - Symbol table: definições globais e referências externas
  - Debug info: informações para ligar endereços do objeto com o código fonte

Efetuando o linking de object modules

UFABC

- Produzem uma imagem executável
  1. Mescla segmentos
  2. Ajusta labels (calcula os seus endereços)
  3. Ajusta elementos que são dependentes de localização e referências externas
- Poder-se-ia deixar a correção dos elementos que dependem de localização para ser feito pelo *loader*
  - Contudo, o uso de memória virtual permite que adiantemos o serviço
    - Programa pode ser sempre carregado no mesmo endereço (virtual)

Carregando um programa

UFABC

- Carrega a imagem do programa do disco para a memória
  1. Lê header para determinar os tamanhos dos segmentos
  2. Cria espaço de memória virtual
  3. Copia texto e dados (pré) inicializados para a memória
     - Ou ajusta a tabela de páginas para que quando ocorra uma falha eles possam ser carregados
  4. Ajusta argumentos na pilha
  5. Inicializa registradores (incluindo `$sp`, `$fp`, `$gp`)
  6. Salta para a rotina de inicialização
     - Copia os argumentos para `$a0`, `$a1`, ...
     - Salta para `main`
     - Quando `main` retorna, chama *syscall* para indicar saída

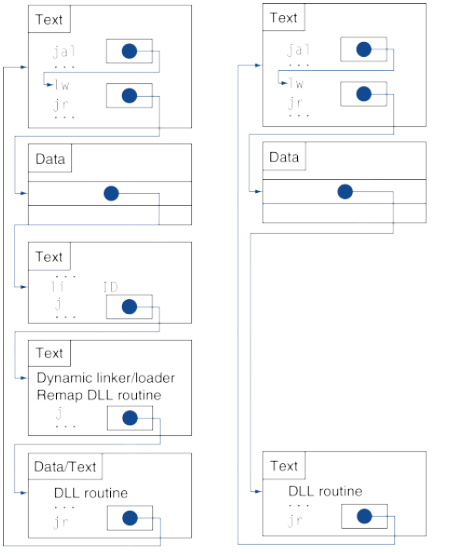
- Apenas fazer o procedimento de link/load quando a biblioteca é chamada
  - É preciso que o código do procedimento seja reposicionável
  - Evita o inchamento da imagem causada pelo linking estático de todas as bibliotecas utilizadas (e suas dependências também!)
  - Automaticamente seleciona versões mais novas das bibliotecas

Indirection table

Stub: Loads routine ID, Jump to linker/loader

Linker/loader code

Dynamically mapped code



a. First call to DLL routine

b. Subsequent calls to DLL routine

Exemplo: Bubble Sort - **swap**

```
1 void swap(int v[], int k) {
2     int temp;
3     temp = v[k];
4     v[k] = v[k+1];
5     v[k+1] = temp;
6 }
```

- v em $a0
- k em $a1
- temp em $t0

Exemplo: Bubble Sort - **swap**

```
1  swap: sll $t1, $a1, 2    # $t1 = k * 4
2        add $t1, $a0, $t1 # $t1 = v+(k*4)
3                          # (endereço de v[k])
4        lw $t0, 0($t1)    # $t0 (temp) = v[k]
5        lw $t2, 4($t1)    # $t2 = v[k+1]
6        sw $t2, 0($t1)    # v[k] = $t2 (v[k+1])
7        sw $t0, 4($t1)    # v[k+1] = $t0 (temp)
8        jr $ra            # retorna
```

Exemplo: Bubble Sort - **bsort**

```
1  void sort (int v[], int n) {
2    int i, j;
3    for (i = 0; i < n; i += 1) {
4      for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
5          swap(v,j);
6      }
7    }
8  }
```

- Função não folha (chama swap)
- v em $a0
- n in $a1
- i in $s0
- j in $s1

Exemplo: Bubble Sort - **bsort** - Corpo

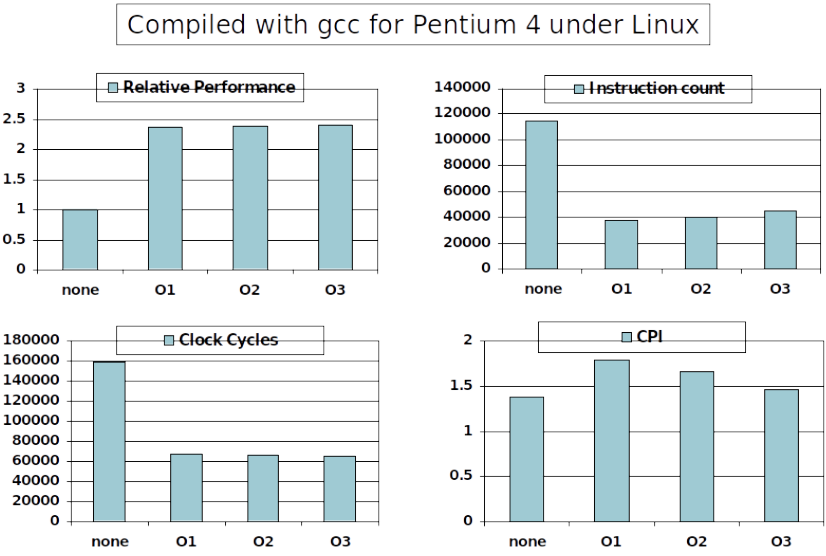
UFABC

```
1           # Move parâmetros
2           move $s2, $a0           # salva $a0 em $s2
3           move $s3, $a1           # salva $a1 em $s3
4           # Laço externo
5           move $s0, $zero         # i = 0
6   for1tst: slt  $t0, $s0, $s3     # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
7           # Laço interno
8           beq  $t0, $zero, exit1  # vai para exit1 se $s0 ≥ $s3 (i ≥ n)
9           addi $s1, $s0, -1       # j = i - 1
10  for2tst: slti $t0, $s1, 0       # $t0 = 1 if $s1 < 0 (j < 0)
11          bne  $t0, $zero, exit2  # vai p/ exit2 se $s1 < 0 (j < 0)
12          sll  $t1, $s1, 2        # $t1 = j * 4
13          add  $t2, $s2, $t1      # $t2 = v + (j * 4)
14          lw   $t3, 0($t2)        # $t3 = v[j]
15          lw   $t4, 4($t2)        # $t4 = v[j + 1]
16          slt  $t0, $t4, $t3      # $t0 = 0 if $t4 ≥ $t3
17          beq  $t0, $zero, exit2  # vai p/ exit2 se $t4 ≥ $t3
18          #Passagem de parâmetros e chamada à swap
19          move $a0, $s2           # 1o par de swap é v (era $a0)
20          move $a1, $s1           # 2o par de swap é j
21          jal  swap               # chama swap
22          # Laço interno
23          addi $s1, $s1, -1       # j -= 1
24          j    for2tst            # salta p/ teste do laço interno
25          # Laço externo
26  exit2:  addi $s0, $s0, 1        # i += 1
27          j    for1tst            # salta p/ teste do laço externo
```

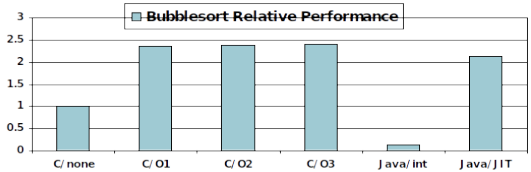
Exemplo: Bubble Sort - **bsort** - Casca

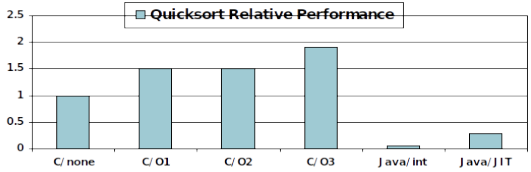
UFABC

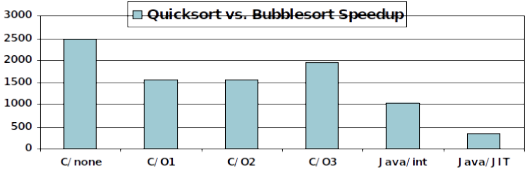
```
1  bsort: addi $sp,$sp, -20       # Reserva espaço na pilha para 5 regs
2         sw $ra, 16($sp)         # salva $ra na pilha
3         sw $s3,12($sp)          # salva $s3 na pilha
4         sw $s2, 8($sp)          # salva $s2 na pilha
5         sw $s1, 4($sp)          # salva $s1 na pilha
6         sw $s0, 0($sp)          # salva $s0 na pilha
7         ...
8         ...                       # corpo do bsort
9         ...
10        exit1: lw $s0, 0($sp)   # restaura $s0 da pilha
11        lw $s1, 4($sp)          # restaura $s1 da pilha
12        lw $s2, 8($sp)          # restaura $s2 da pilha
13        lw $s3,12($sp)          # restaura $s3 da pilha
14        lw $ra,16($sp)          # restaura $ra da pilha
15        addi $sp,$sp, 20        # restaura stack pointer
16        jr $ra                  # retorna
```


Compiled with gcc for Pentium 4 under Linux
Relative Performance
3
2.5
2
1.5
1
0.5
0
none
O1
O2
O3
Instruction count
140000
120000
100000
80000
60000
40000
20000
0
none
O1
O2
O3
Clock Cycles
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
none
O1
O2
O3
CPI
2
1.5
1
0.5
0
none
O1
O2
O3

Efeito da linguagem e do algoritmo







UFABC

- Número de instruções e CPI não são bons indicadores de desempenho se vistos isoladamente
- Otimizações do compilador são sensíveis ao algoritmo
- Java/JIT é bem mais rápido do que código interpretado
  - Comparável à C em alguns casos
- Nada conserta um algoritmo ruim!

Arrays *vs.* Ponteiros

- Indexação de arrays envolve
  - Multiplicação do índice pelo tamanho do elemento
  - Adição à base da array
- Ponteiros apontam diretamente para a posição de memória desejada
  - Podem evitar a complexidade envolvida com idexação

Exemplo: Limpando uma Array

```
1  clear1(int array[], int size) {
2    int i;
3    for (i = 0; i < size; i += 1)
4      array[i] = 0;
5  }
```

```
1    move $t0,$zero   # i = 0
2  loop1:
3    sll $t1,$t0,2    # $t1 = i * 4
4    add $t2,$a0,$t1  # $t2 =
5                     #   &array[i]
6    sw $zero, 0($t2) # array[i] = 0
7    addi $t0,$t0,1   # i = i + 1
8    slt $t3,$t0,$a1  # $t3 =
9                     #   (i < size)
10   bne $t3,$zero,loop1 # if (…)
11                       # goto loop1
```

```
1  clear2(int *array, int size) {
2    int *p;
3    for (p = &array[0]; p < &array[size]; p = p + 1)
4      *p = 0;
5  }
```

```
1    move $t0,$a0     # p = & array[0]
2    sll $t1,$a1,2    # $t1 = size * 4
3    add $t2,$a0,$t1  # $t2 =
4                     #   &array[size]
5  loop2:
6    sw $zero,0($t0) # Memory[p] = 0
7    addi $t0,$t0,4  # p = p + 4
8    slt $t3,$t0,$t2 # $t3 =
9                    #(p<&array[size])
10   bne $t3,$zero,loop2 # if (…)
11                       # goto loop2
```

UFABC

- Multiplicação acaba virando um shift
- A versão com array necessita que o shift esteja dentro do laço
  - Faz parte do cálculo do índice (ao incrementar o i)
  - ... *vs.* incrementar um ponteiro
- O compilador é capaz de otimizar o código e atingir o mesmo efeito que usando ponteiros diretamente
  - Elimina variáveis
  - O programador faz bem em se aproveitar para tornar o programa mais seguro e simples

Outras arquiteturas

Semelhanças entre ARM e MIPS

UFABC

- ARM é o processador embarcado mais popular
- Tem um conjunto de instruções parecido com MIPS

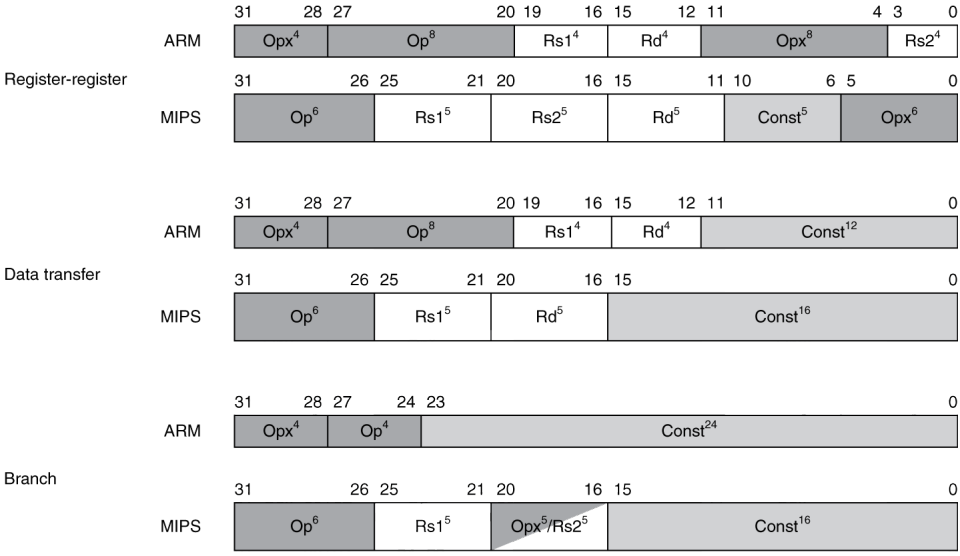
	ARM	MIPS
Ano	1985	1985
Tam. instrução	32 bits	32 bits
Espaço de End.	32 bits (flat)	32 bits (flat)
Alinhamento	Alinhado	Alinhado
Modos de End.	9	3
Registradores	15 x 32 bits	31 x 32 bits
E/S	Mapeado em memória	Mapeado em memória

Compare and Branch no ARM

UFABC

- Usa códigos de condição para resultados de operações lógico/aritméticas
  - Negativo, zero, carry, overflow
  - Compara instruções aos bits de condição sem manter o resultado
- Cada instrução pode ser condicional
  - 4 primeiros bits da instrução: condições para execução
  - Consegue evitar branches em uma única instrução

ARM - Formato das instruções



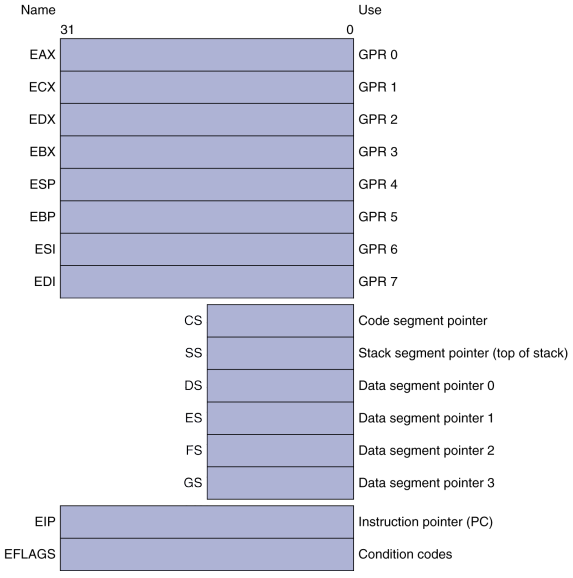
- Evolução mantendo a compatibilidade
  - 8080 (1974): microprocessador 8 bits
    - Acumulador, + 3 pares índice registrador
  - 8086 (1978): extensão 16 bits no 8080
    - Complex instruction set (CISC)
  - 8087 (1980): coprocessador de ponto flutuante
    - Adicionava suporte a instruções de ponto flutuante e pilha de registradores
  - 80286 (1982): endereços de 24 bits, MMU
    - Memória segmentada e protegida
  - 80386 (1985): extensão 32 bits (conhecida hoje como IA-32)
    - Modos adicionais de endereçamento e operação
    - Memória paginada e segmentos

- Continuou evoluindo...
  - i486 (1989): pipelined, on-chip caches e FPU
    - Competidores com compatíveis: AMD, Cyrix, ...
  - Pentium (1993): superescalar, datapath de 64 bits
    - Versões seguintes adicionaram a extensão MMX (Multi-Media eXtension)
    - Teve o infame bug FDIV
  - Pentium Pro (1995), Pentium II (1997)
    - Nova microarquitetura (veja Colwell, The Pentium Chronicles)
  - Pentium III (1999)
    - Adicionou SSE (Streaming SIMD Extensions) e novos registradores associados
  - Pentium 4 (2001)
    - Nova microarquitetura
    - Adicionou SSE2

UFABC

- Mais um pouquinho...
  - AMD64 (2003): estendeu a arquitetura para 64 bits
  - EM64T - Extended Memory 64 Technology (2004)
    - A Intel adota o AMD64 (com alguns refinamentos)
    - Adiciona SSE3
  - Intel Core (2006)
    - Adiciona SSE4, suporte a máquinas virtuais
  - AMD64 (2007): SSE5
    - Intel decidiu não seguir a AMD e acabou lançando..
  - Advanced Vector Extension (2008)
    - Registradores SSE mais longos e mais instruções
  - ...
- Se a Intel não estendesse a compatibilidade, seus competidores o fariam!
  - Elegância técnica ≠ sucesso no mercado

Registradores Intel x86 (Básicos)



UFABC

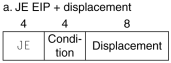
- Dois operandos por instrução

Destino	Origem
Registrador	Registrador
Registrador	Imediato
Registrador	Memória
Memória	Registrador
Memória	Imediato

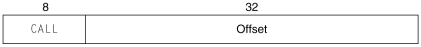
- Modos de endereçamento de memória
  - Endereço no registrador
  - Endereço = R_{Base} + Deslocamento
  - Endereço = $R_{Base} + 2^{escala} \times R_{índice}$ (escala = 0, 1, 2 ou 3)
  - Endereço = $R_{Base} + 2^{escala} \times R_{índice}$ + Deslocamento

Formato das instruções Intel x86

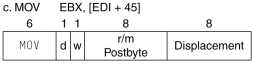
a. JE EIP + displacement



b. CALL



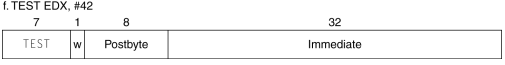
c. MOV EBX, [EDI + 45]



d. PUSH ESI

e. ADD EAX, #6765

f. TEST EDX, #42



- Formato de comprimento variável
  - Bytes sufixos indicam modo de endereçamento
  - Bytes prefixos modificam a operação
    - Comprimento do operando, repetição, travas, ...

UFABC

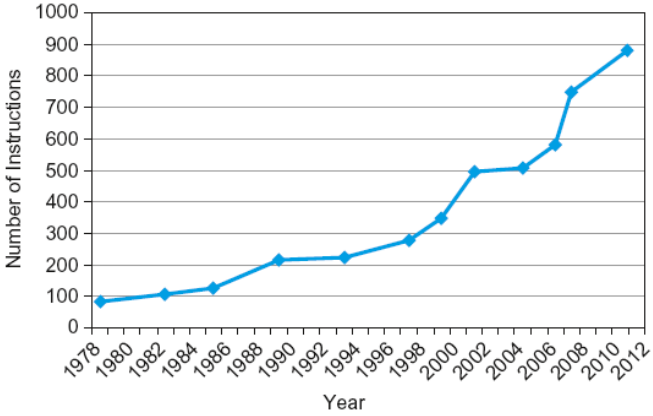
- Conjunto de instruções complexo dificulta a implementação
  - O hardware traduz instruções complexas para microoperações mais simples
    - Instruções simples 1-1
    - Instruções complexas 1-muitas
    - Mecanismo de execução parecido com RISC
    - A fatia de mercado da Intel possibilita que isso seja viável economicamente
  - Desempenho comparável com RISC
    - Compiladores acabam evitando instruções complexas

- Quando lançou a versão 64 bits, ARM fez uma reforma completa
- ARMv8 é muito parecido com MIPS
- Mudanças de ARMv7:
  - Não há campos de execução condicional
  - Campo de imediato é uma constante de 12 bits
  - Descartaram operações de load/store múltiplas
  - PC não é mais um GPR
  - 32 GPRs
  - Modos de endereçamento funcionam para todos os tamanhos de palavra
  - Divide instruction
  - Instruções `beq` e `bne`

UFABC

- Instrução poderosa → melhor desempenho
  - Precisa de menos instruções
  - Mas instruções complexas são difíceis de implementar
    - Podem atrasar outras instruções, incluindo as simples
  - Compiladores são bons em criar código rápido usando instruções simples
- Use código assembly para códigos mais rápidos
  - Mas compiladores modernos são melhores para lidar com processadores modernos
  - Mais linhas de código → mais erros e menos produtividade

- Compatibilidade reversa → conjunto de instruções não muda
  - mas ele acumula cada vez mais instruções!



x86 instruction set

- Palavras sequenciais não tem endereços sequenciais
  - Incrementam de 4 em 4, e não de 1 em 1
    - Cada palavra tem 4 bytes
- Manter um ponteiro para uma variável automática depois que um procedimento retorna
  - Por exemplo, passar um ponteiro de volta como um argumento
  - O ponteiro se torna inválido quando a desempilhamos a pilha

- Princípios de design
  - Simplicidade favorece a regularidade
  - Menor é mais rápido
  - Faça o caso mais comum ser rápido
  - Um bom design exige alguns bons compromissos
- Camadas de software/hardware
  - Compilador, assembler, hardware
- MIPS é um exemplo típico de ISAs RISC
  - Em contraste com Intel x86

UFABC

- Medida de instruções MIPS em benchmarks
  - Considere fazer o caso comum rápido
  - Considere os compromissos

Classe	Exemplos MIPS	SPEC2006 Int	SPEC2006 FP
Aritmética	`add`, `sub`, `addi`	16%	48%
Tranf. Dados	`lw`, `sw`, `lb`, `lbu`, `lh`, `lhu`, `sb`, `lui`	35%	36%
Lógicas	`and`, `or`, `nor`, `andi` `ori`, `sll`, `srl`	12%	4%
Branchs Cond.	`beq`, `bne`, `slt`, `slti`, `sltiu`	34%	8%
Saltos	`j`, `jr`, `jal`	2%	0%