Instruções - Parte 1

Arquitetura de Computadores

Emilio Francesquini
e.francesquini@ufabc.edu.br

2020.Q1

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

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

UFABC

- `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

UFABC

O seguinte código em C...

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

... se transforma em:

```
1  add t0, g, h  # temp t0 = g + h
2  add t1, i, j  # temp t1 = i + j
3  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

Registradores

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

UFABC

```
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 conhecido 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 menos 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.

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

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

Código MIPS:

```
lw  $t0, 32($s3)  # Atenção ao deslocamento
add $t1, $s2, $t0
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
```

UFABC

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

A constante Zero

UFABC

- 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
```

Números binários 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

Números com sinal - Complemento de 2

- 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: $0111\ 1111 \ldots 1111$

Negação

- 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 10!

Qual é a ideia por trás?

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

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

Negação: Exemplo

UFABC

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

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

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

$$1111\ 1111 \ldots 1110_2$$

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:
  - $+2 : 0000\ 0010 \rightarrow 0000\ 0000\ 0000\ 0010$
  - $-2 : 1111\ 1110 \rightarrow 1111\ 1111\ 1111\ 1110$

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`

UFABC

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 registro fonte
  - `rt`: número do segundo registro fonte
  - `rd`: número do registrador de destino
  - `shamt`: quantidade de deslocamento (00000 por equanto)
  - `funct`: código da função (extensão do opcode)

Exemplo: R-format

UFABC

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	010008	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

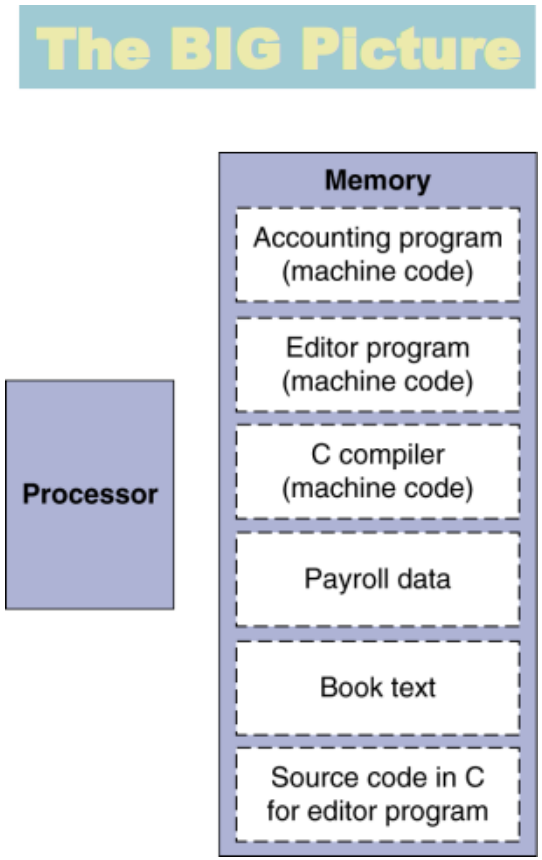
- Exemplo: $eca8\ 6420$
  - $1110\ 1100\ 1010\ 1000\ 0110\ 0100\ 0010\ 0000$

UFABC

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

Computadores com programa armazenado



- 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

Operações de shift

UFABC

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

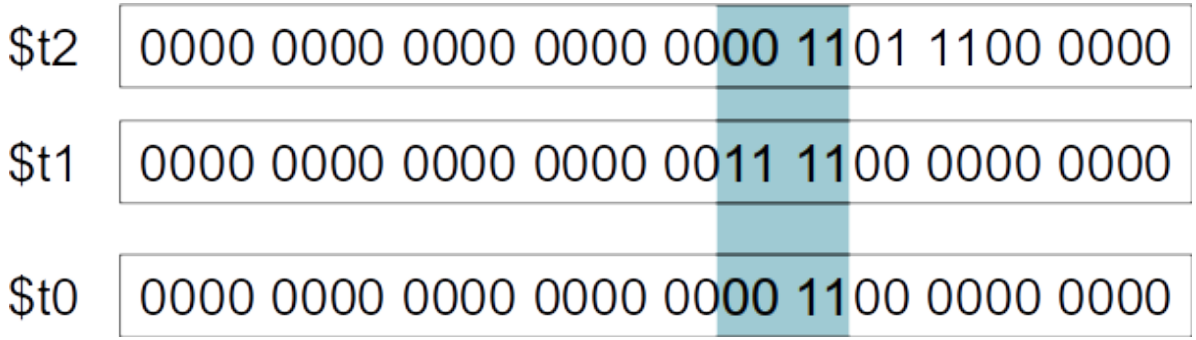
- `shamt`: quantas posições devem ser deslocadas
- `sll`: *Shift left logical*
  - Desloca para a esquerda e completa com 0s
  - `sll` por i bits é o mesmo que multiplicar por 2^i
- `srl`: *shift right logical*
  - `srl` por i bits é o mesmo que dividir por 2^i. Atenção! Apenas para números sem sinal

Operações AND

UFABC

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

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

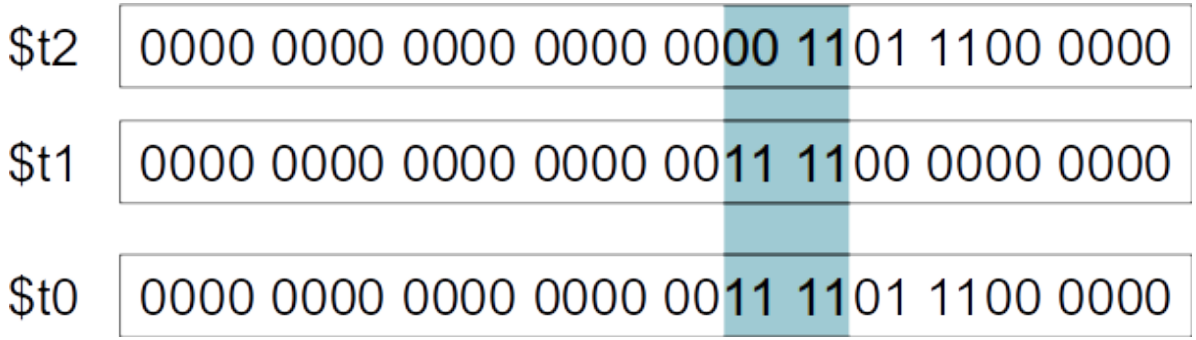


Operações OR

UFABC

- 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

UFABC

- 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
  - `a NOR b == NOT (a OR B)`

```
1  not $t0, $t1, $zero
```

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

Operações condicionais

UFABC

- 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`

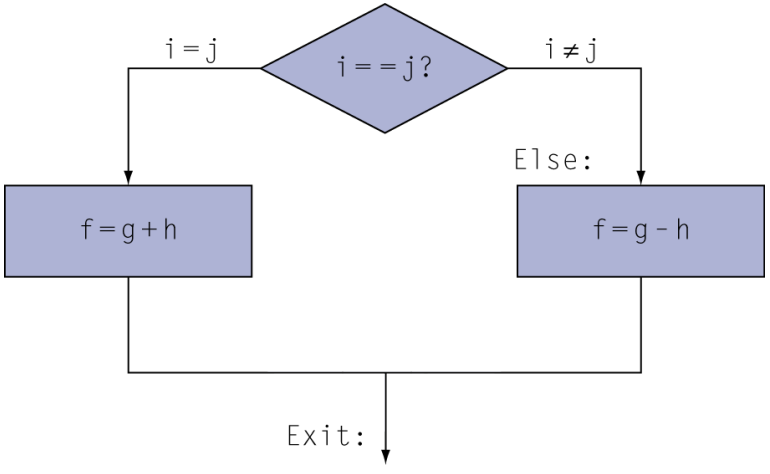
```
1  bne rs, rt, L1
```

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

```
1  j L1
```

Compilando um if

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



Com `f`, `g`, `...` em `$s0`, `$s1`, `...`

Código MIPS compilado

```
1      bne $s3, $s4, Else
2      add $s0, $s1, $s2
3      j Exit
4  Else:
5      sub $s0, $s1, $s2
6  Exit: ...
```

Compilando laços

UFABC

```
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

UFABC

- 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`

Chamadas de procedimentos

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

UFABC

- 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

UFABC

```
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

UFABC

- 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

UFABC

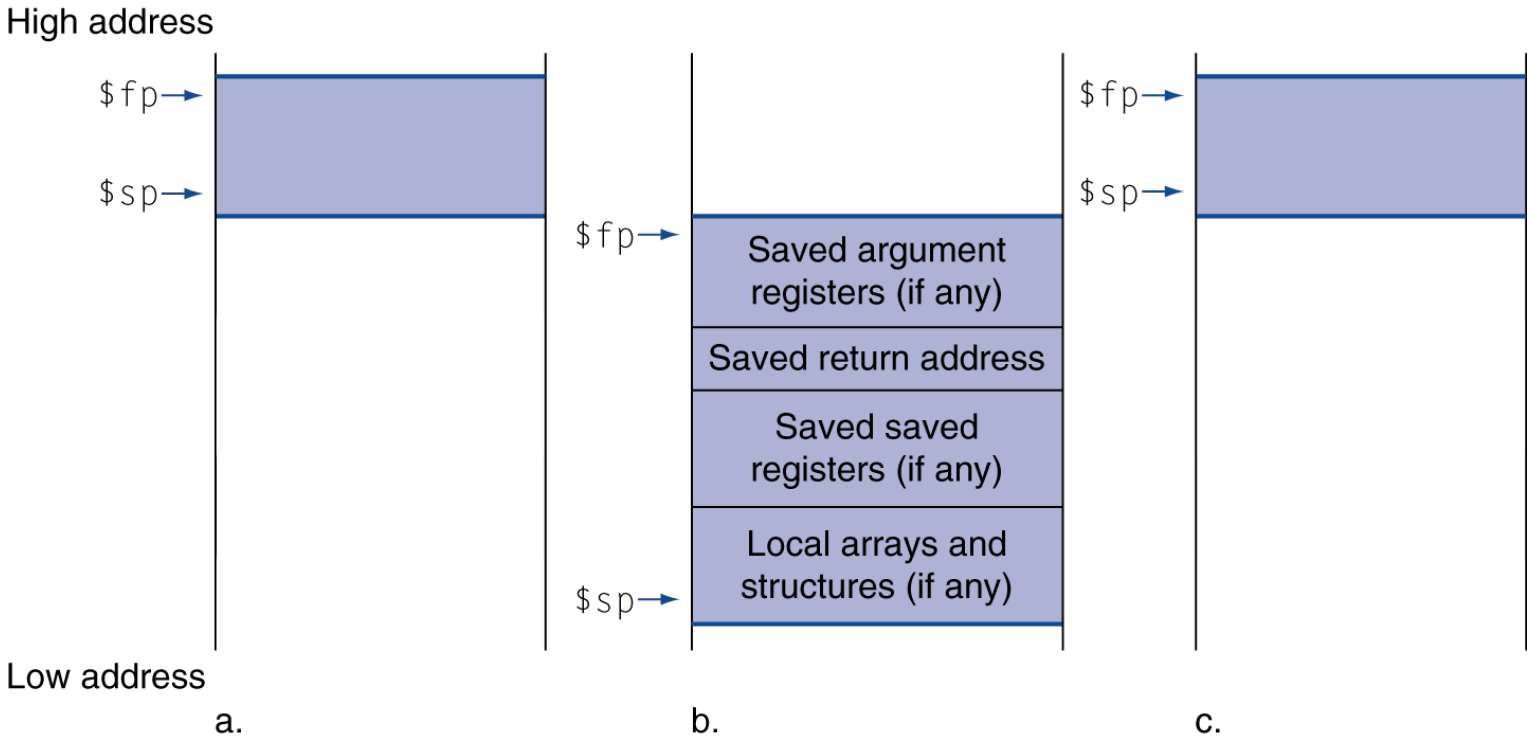
```
1  int fact (int n) {
2      if (n < 1)
3          return 1;
4      else
5          return n * fact (n - 1);
6  }
```

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
```

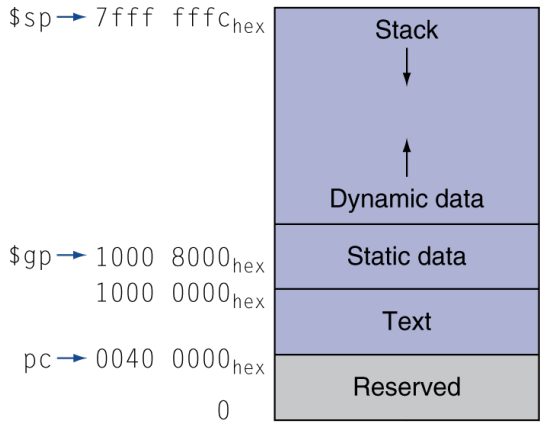
Dados locais na pilha



- 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 alfabetos do mundo (incluindo símbolos)
  - UTF-8 e UTF-16: maneiras de codificar com comprimento variável

Operações sobre Byte/Halfword

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 sb rt, offset(rs)
```

String Copy

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
```