Arquitetura de Computadores

Instruction Set Architecture (ISA)

UFMT

Fevereiro de 2025

Agenda

- Introdução.
- Programa Armazenado e Modelo de Von Newman.
- Arquiteturas CISC *versus* RISC.
- MIPS:
    1. Visão geral.
    2. Assembly *vs* Linguagem de Máquina.
    3. Conjunto de Instruções.

Parte 1

    4. Programação.
    5. Modos de Endereçamento.
    6. Compilação, Montagem, Ligação e Carga.
    7. Pseudo-instruções.
    8. Exceções.
    9. Números em Ponto Flutuante.

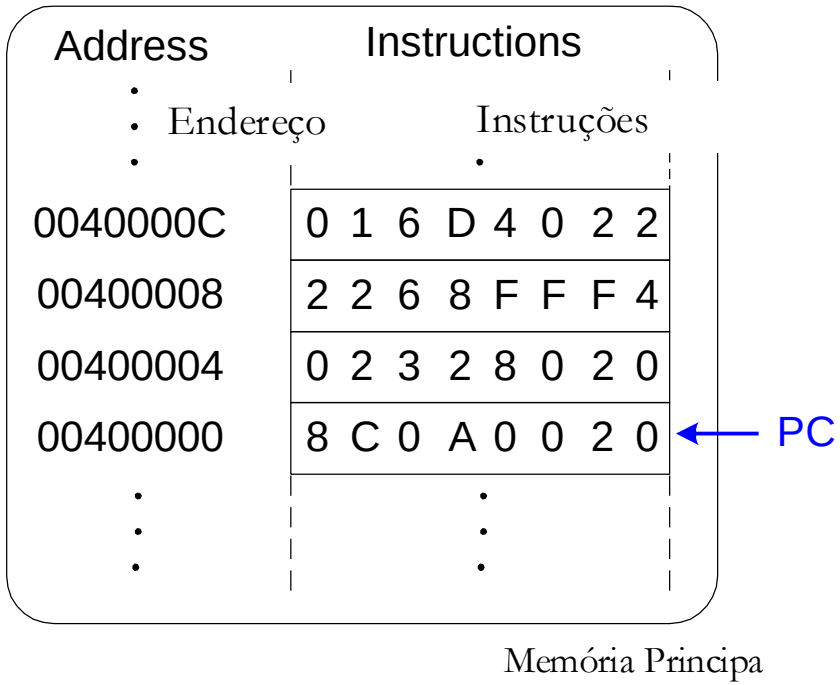
Parte 2

MIPS
Construções de Programação

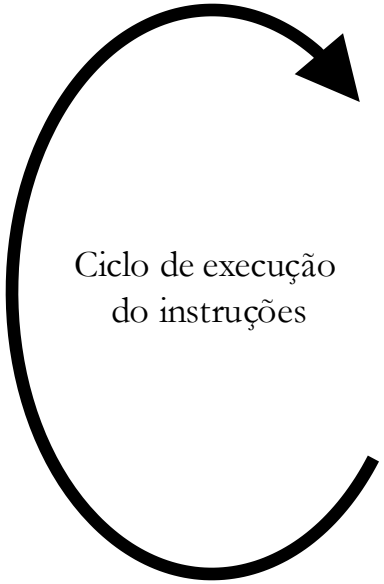
Relembrando o Conceito de Programa Armazenado

Código Assembly	Código de Máquina
`lw $t2, 32($0)`	`0x8C0A0020`
`add $s0, $s1, $s2`	`0x02328020`
`addi $t0, $s3, -12`	`0x2268FFF4`
`sub $t0, $t3, $t5`	`0x016D4022`

Programa Carregado em Memória



Memória Principa



- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT

- **O PC (Program Counter)** determina a próxima instrução a ser executada.
  - É inicializado pelo SO com o endereço 0x00400000.
  - É incrementado pelo processador a cada ciclo de instrução, PC=PC+1, exceto no caso de instrução de desvio.

Construções de Programação

- As construções básicas de programação comumente disponíveis em Linguagens de Alto Nível são as seguintes:
  - Seleção com **if**, **if-else** e **switch**.
  - Loops **for** e **while**.
  - Criação e acesso à **arrays**.
  - Criação e chamada de **funções**.

- Queremos entender como essas construções de alto nível são materializadas em Assembly MIPS?

MIPS
Construções de Programação

Geração de Constantes

Geração de Constantes de 16 bits

- A geração de uma constante de 16 bits pode ser feita usando a instrução Tipo-I `addi`:

Código C

```
//int (32 bits comp. de 2)
int a = 0x4f3c;
```

Assembly MIPS

```
# $s0 = a
addi $s0, $0, 0x4f3c
```

- Intervalo de representação para a constante de 16 bits [-32768, +32767].
- Lembre-se que `addi`, antes de realizar a soma, estende a constante para 32 bits com sinal (singn-extendend).

Geração de Constantes de 32 bits

- Constantes de 32 bits podem ser geradas usando as instruções Tipo-I `lui` (*load upper immediate*) combinada com `ori`.
- A instrução `lui` tem o seguinte formato: **`lui rt, imm`**
  - **`lui`**: mnemônico que indica o carregamento do imediato de 16 bits (`imm`) na metade superior do registrador `rt` e zero na metade inferior.
  - **`rt`**: registrador de destino.
  - **`rs`**: deve ficar zerado.
  - **`imm`**: constante de 16 bits a ser carregada no registrador `rt`.
- **Exemplo:**

Código C

```
int a = 0xFEDC8765;
```

Código Assembly MIPS

```
# $s0 = a
lui $s0, 0xFEDC
ori $s0, $s0, 0x8765
```

MIPS

Construções de Programação

`if`, `if-else` e `switch-case`

Construção If

```
// Código C
1. if (i == j){
2.    f = g + h;
3. }
4. f = f - i;
```

```
# Assembly MIPS
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
```

Construção `If`

```
// Código C
1. if (i == j){
2.    f = g + h;
3. }
4. f = f - i;
```

```
# Assembly MIPS
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
  bne $s3, $s4, L1
  add $s0, $s1, $s2
L1:
  sub $s0, $s0, $s3
```

O assembly testa o caso oposto (`i!=j`) do código C (`i==j`)

Construção `If-Else`

```
// Código C

1. if (i == j){
2.    f = g + h;
3. }else{
4.    f = f - i;
5. }
```

```
# Assembly MIPS
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
```

Construção If-Else

```
// Código C

1. if (i == j){
2.    f = g + h;
3. }else{
4.    f = f - i;
5. }
```

```
# Assembly MIPS
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
   bne $s3, $s4, ELSE
   add $s0, $s1, $s2
   j   done
ELSE:
   sub $s0, $s0, $s3
done:
```

Comando switch-Case

- O comando **switch-case** é equivalente a uma sequência de comandos **if-else** aninhados.

```
// Código C
switch (amount)
{
  case 20:  fee = 2; break;
  case 50:  fee = 3; break;
  case 100: fee = 5; break;
  default: fee = 0;
}
```

```
// Código C equivalente
if (amount == 20){
  fee = 2;
}else{
   if (amount == 50){
       fee = 3;
   }else{
      if (amount == 100)
         fee = 5;
      else fee = 0;
   }
}
```

Comando switch-Case

```
// Código C
switch (amount)
{
   case 20:  fee = 2; break;
   case 50:  fee = 3; break;
   case 100: fee = 5; break;
   default: fee = 0;
}
```

```
# Assembly MIPS
# $s0 = amount, $s1 = fee
case20:
   addi $t0, $0, 20
   bne $s0, $t0, case50
   addi $s1, $0, 2
   j done
case50:
   addi $t0, $0, 50
   bne $s0, $t0, case100
   addi $s1, $0, 3
   j done
case100:
   addi $t0, $0, 100
   bne $s0, $t0, default
   addi $s1, $0, 5
   j done
default:
   add $s1,$0,$0
done:
```

MIPS

Contruções de Programação

Loops `while` e `for`

Loop While

- O loop `while` abaixo é usado para determinar o valor de `x` tal que $2^x=128$.

```
// Código C
int pow = 1;
int x   =  0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

```
# Assembly MIPS
# $s0 = pow, $s1 = x
```

Loop While

- O loop `while` abaixo é usado para determinar o valor de x tal que $2^x = 128$.

```
// Código C
int pow = 1;
int x   =  0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

```
# Assembly MIPS
# $s0 = pow, $s1 = x
        addi $s0, $0, 1
        add  $s1, $0, $0
        addi $t0, $0, 128
while: beq  $s0, $t0, done
        sll  $s0, $s0, 1
        addi $s1, $s1, 1
        j    while
done:
```

Loop `for`

```
for (inicialização; condição; operação do loop)
{
    comandos;
}
```

- **Inicialização:** executa uma vez antes do loop começar.
- **Condição:** testada no começo de cada iteração.
- **Operação do loop:** executa no final de cada iteração.
- **Comandos:** executados a cada iteração se condição for satisfeita.

Loop `for`

- O loop for abaixo soma os número de 0 até 9.

```
// Código C
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
  sum = sum + i;
}
```

```
# Código assembly MIPS
# $s0 = i, $s1 = sum
```

Loop `for`

- O loop for abaixo soma os número de 0 até 9.

```
// Código C
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
  sum = sum + i;
}
```

```
# Código assembly MIPS
# $s0 = i, $s1 = sum
        addi $s1, $0, 0
        add  $s0, $0, $0
        addi $t0, $0, 10
for:    beq  $s0, $t0, done
        add  $s1, $s1, $s0
        addi $s0, $s0, 1
        j    for
done:
```

Comparação de Magnitude

- O MIPS fornece a instrução Tipo-R `slt` (set less than) para comparação de magnitude.

- **Formato em assembly**: **`slt rd`**, **`rs`**, **`rt`**.
  - **`slt`**: `Mnemônico` que indica comparação rs < rt.
  - **`rs, rt`**: registradores fonte com valores a serem comparados.
  - **`rd`**: registrador de destino.
  - **Resultado**: `se (rs < rt) rd = 1 senão rd = 0`

Comparação de Magnitude

- Há também a instrução Tipo-I `slti` para comparação de magnitude.

- **Formato em assembly**: **`slti rt`**, **`rs`**, **`imm`**.
  - **`slti`**: `Mnemônico` que indica comparação rs < imm.
  - **`rs`**: registrador fonte com valor a ser comparado.
  - **`rt`**: registrador de destino.
  - **`imm`**: constante de 16 a ser comparado (comp. 2).
  - **Resultado**: `se (rs < imm) rt = 1 senão rt = 0`

Exemplo com `slt`

- O código de alto nível a seguir acumula as potências de 2 menores que 101.

```
// Código C
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
  sum = sum + i;
}
```

```
# Assembly MIPS
# $s0 = i, $s1 = sum
        addi $s1, $0, 0
        addi $s0, $0, 1
        addi $t0, $0, 0
for:    slti $t1, $s0, 101
        beq  $t1, $t0, done
        add  $s1, $s1, $s0
        sll  $s0, $s0, 1
        j    for
done:
```

Estrutura Básica de Arquivos Assembly MIPS

Labels

- Os labels são "marcadores" que representam endereços de instruções no programa Assembly.
- Eles são definidos por um nome terminado com o sufixo ":" e podem ser inseridos em um programa assembly para "marcar" uma posição no programa, de modo que ele possa ser referido por instruções ou outros comandos assembly, como diretivas.
- No final das contas, os labels são a convertidos pelo assembler para um valor numérico que representa sua posição no programa. O processador usa esse valor numérico para computar um endereço de memória.

Labels

```
    addi $t0, $0, 5

loop:                           # label
    addi $t0, $t0, -1
    bne $t0, $zero, loop        # instrução referenciando o label
```

Quantas iterações vão ocorrer??

Ponto de Entrada do Programa

- Todo programa possui um ponto de entrada a partir do qual a CPU deve iniciar a execução. O ponto de entrada é definido por um endereço, que é o endereço da primeira instrução que deve ser executada.
- O arquivo executável possui um cabeçalho que contém muitas informações sobre o programa e um dos campos de cabeçalho armazena o endereço do ponto de entrada.
- Quando o Sistema Operacional carrega o programa executável na memória principal, ele inicializa o PC com o endereço do ponto de entrada para que o programa comece a ser executado.
- O linker define esse ponto de entrada, geralmente associado-o ao label `main` definido no assembler.

Definição do Ponto de Entrada em Assembly MIPS

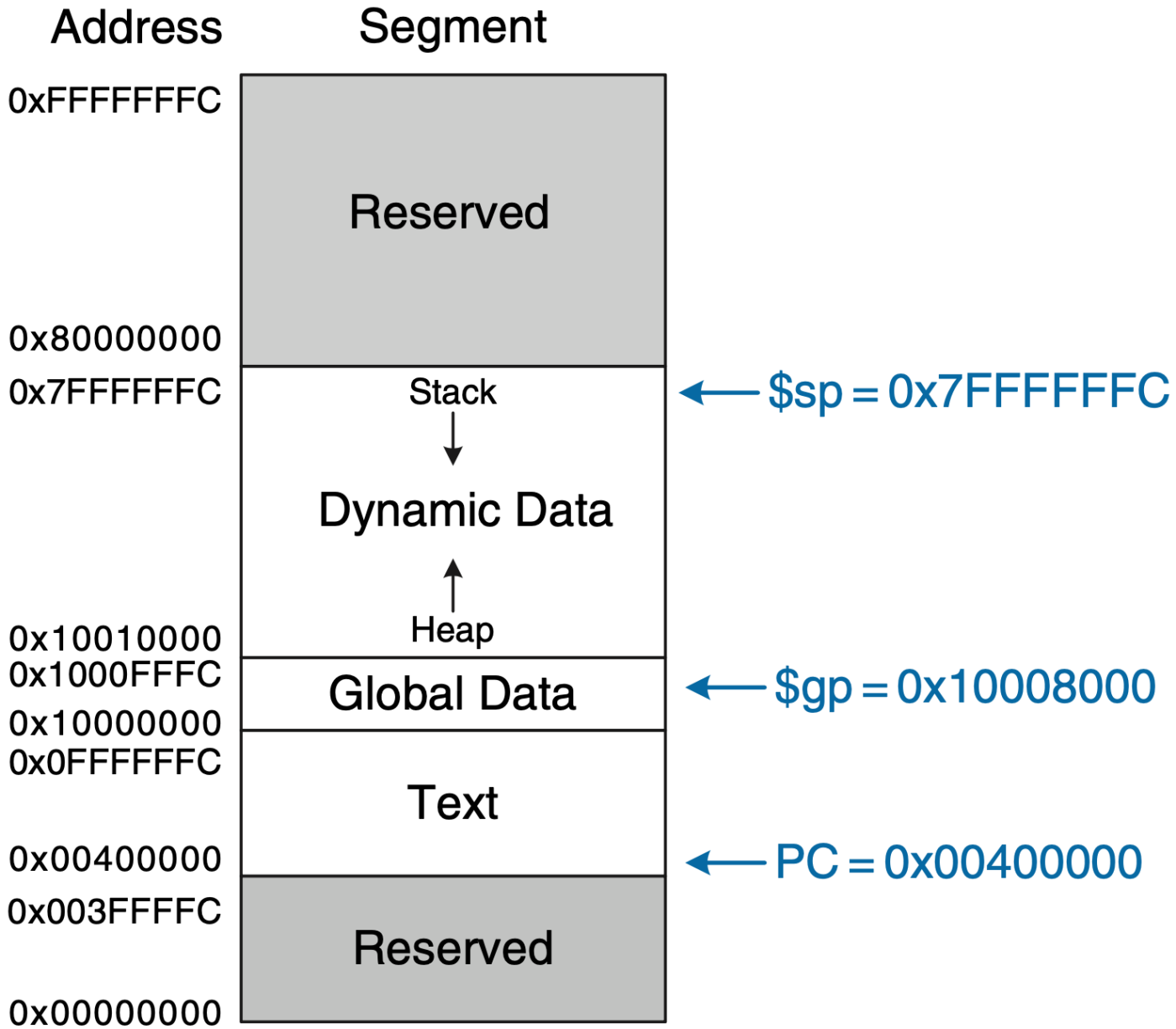
```
.text
    .globl main             # declara 'main' como global

main:                       # main é o ponto de entrada
    addi $t0, 10
    addi $t1, 5
    add $t2, $t0, $t1
```

Seções do Programa

- Arquivos executáveis , bem como programas assembly são geralmente organizados em seções.
- Uma seção pode conter dados ou instruções, e o conteúdo de cada seção é mapeado para um conjunto consecutivo de endereços da memória principal.
- A seguir discutiremos as principais seções frequentemente presentes em arquivos assembly MIPS.

Exemplo de Seções de um Programa em Memória



Seções do Programa em Assembly

- **`.text:`** seção dedicada a armazenar as instruções do programa. Mapeada para a seção de texto em memória.
- **`.data:`** seção dedicada a armazenar variáveis globais inicializadas, ou seja, as variáveis que precisam que seu valor seja inicializado antes do programa começar a ser executado.
- **`.bss:`** seção dedicada a armazenar variáveis globais não inicializadas. Não está presente no simulador MARS
- **`.rodata:`** seção dedicada a armazenar constantes, ou seja, valores que são lidos pelo programa, mas não modificados durante a execução. Não está presente no simulador MARS.

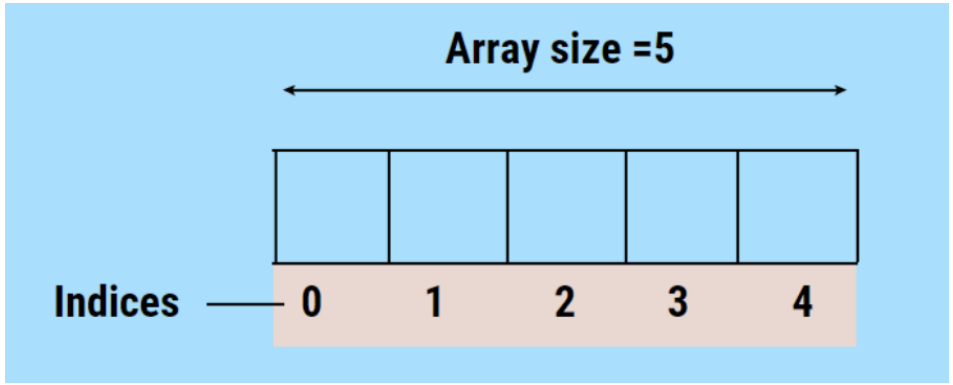
Outras Diretivas Importantes

- **Diretivas para alocação de dados:** usadas para alocar memória e inicializar valores.
  - .word w1, w2, ..., wn : aloca *n* valores de 4 bytes em posições consecutivas da memória.
  - .byte w1, w2, ..., wn : aloca *n* valores de 1 byte em posições consecutivas da memória.
  - .half w1, w2, ..., wn : aloca *n* valores de 2 byte em posições consecutivas da memória.

MIPS
Construções de Programação

Arrays

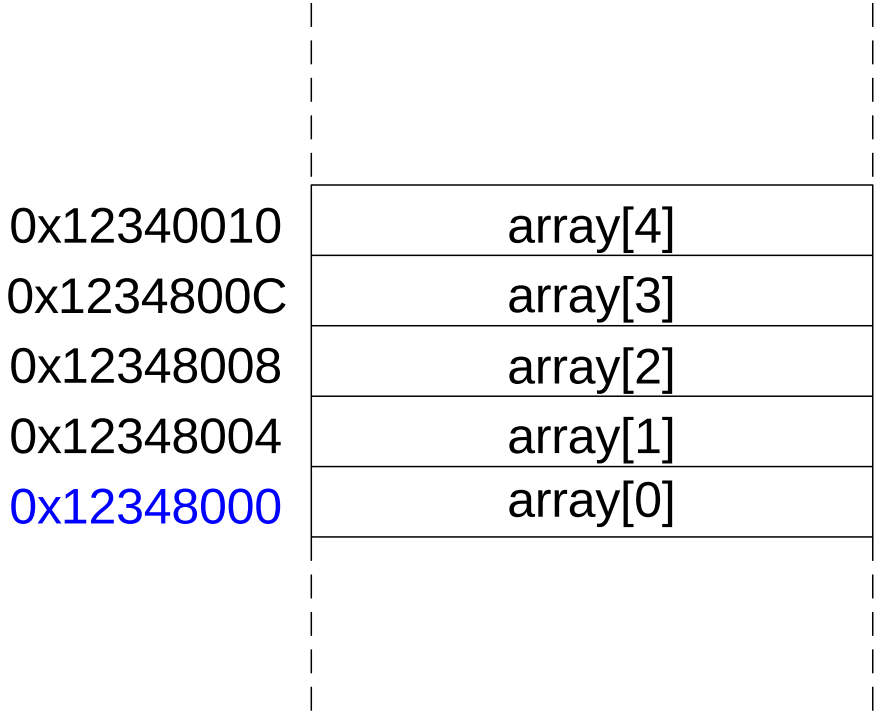
Arrays



- Arrays são estruturas de dados úteis para armazenar **em memória** dados de um mesmo tipo.
- Os elementos de um array são armazenados sequencialmete na memória, começando em um endereço chamado de **endereço base**.
- Cada elemento do array é identificado por um inteiro chamado **índice**, que dá a posição do elemento em relação ao endereço base.
- O número de elementos do array é chamado de **tamanho** do array.

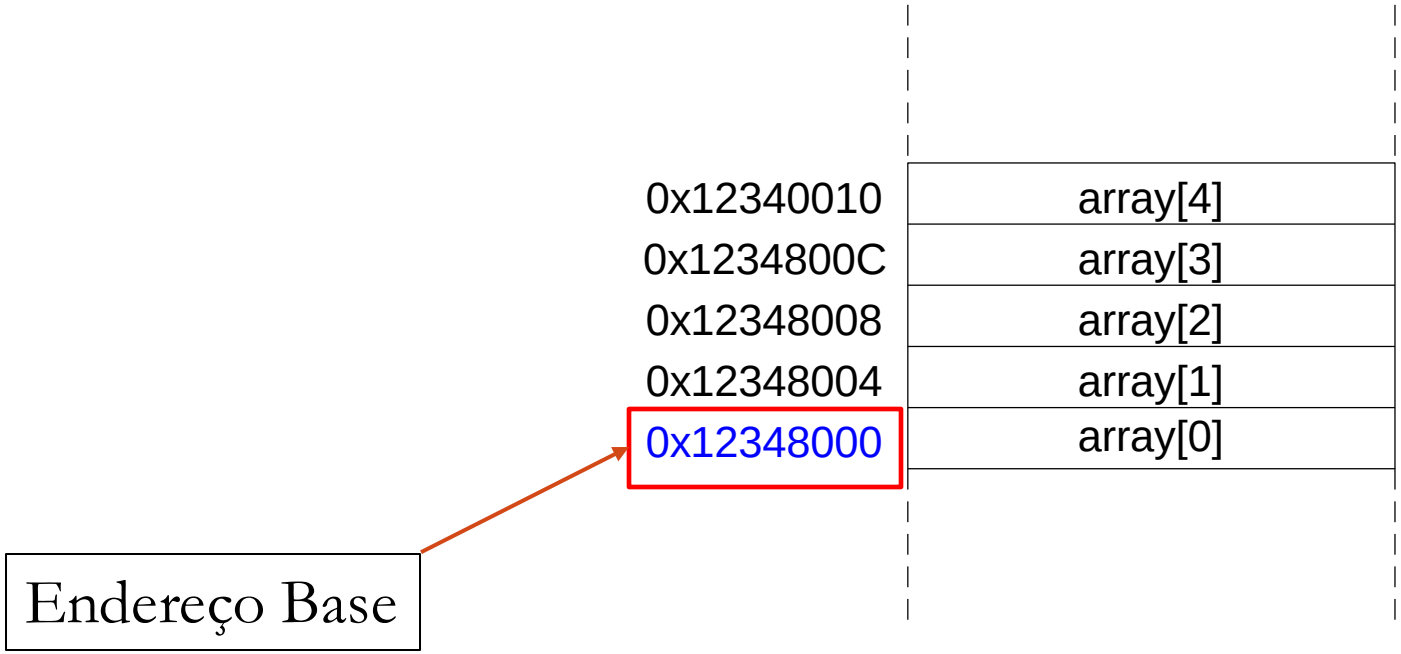
Arrays

- Abaixo uma ilustração de um array com 5 elementos inteiros de 32 bits.
- **Endereço Base** = 0x12348000 (&array[0] em C)



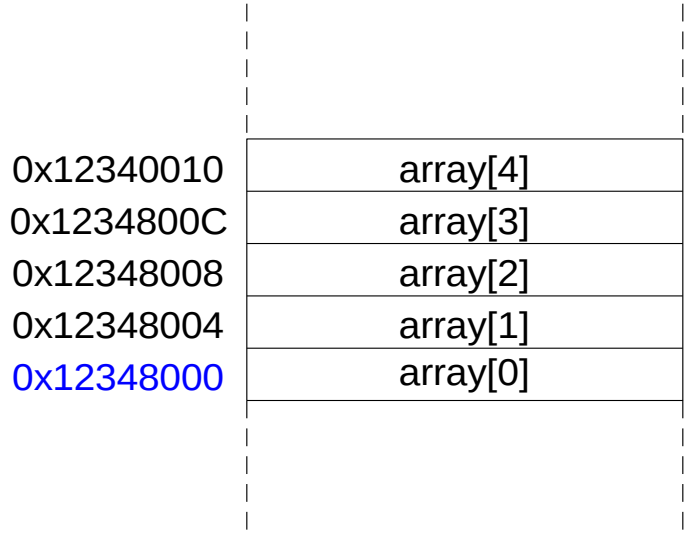
Arrays

- O primeiro passo para acessar um array é carregar (load) o seu **endereço base** para um registrador do Register File.



Manipulando Arrays

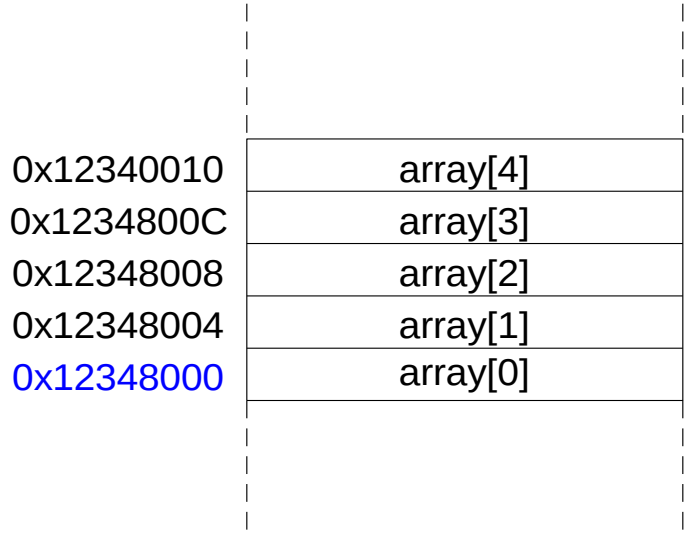
```c
// Código C
int array[5];
array[0] = array[0] * 2;
array[1] = array[1] * 2;
```




```assembly
# Assembly MIPS
# $s0 = &array[0]
```

Manipulando Arrays

```c
// Código C
int array[5];
array[0] = array[0] * 2;
array[1] = array[1] * 2;
```

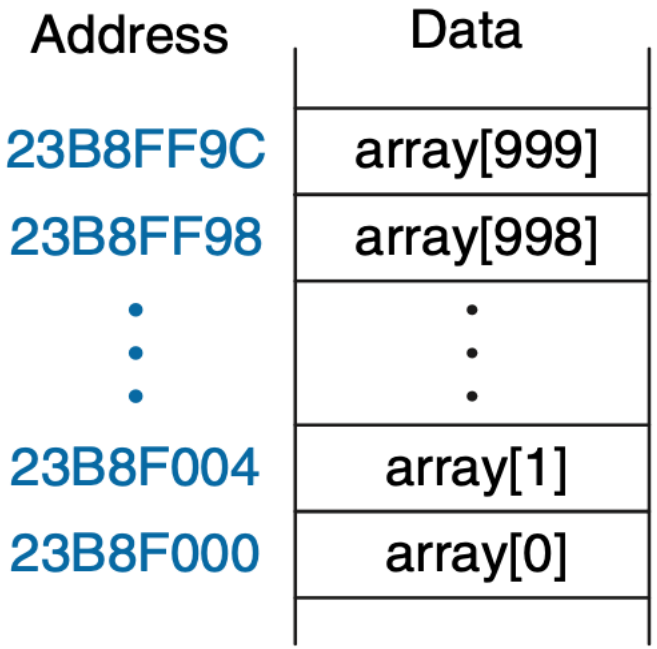



```assembly
# Assembly MIPS
# $s0 = &array[0]
  lui   $s0, 0x1234          # 0x1234 $s0[31:16]
  ori   $s0, $s0, 0x8000     # 0x8000 $s0[15:0]
  lw    $t1, 0($s0)          # $t1 = array[0]
  sll   $t1, $t1, 1          # $t1 = $t1 * 2
  sw    $t1, 0($s0)          # array[0] = $t1
  lw    $t1, 4($s0)          # $t1 = array[1]
  sll   $t1, $t1, 1          # $t1 = $t1 * 2
  sw    $t1, 4($s0)          # array[1] = $t1
```

Percorrendo Arrays

```
// Código C
   int array[1000];
   int i;

   for (i=0; i < 1000; i = i + 1)
       array[i] = array[i] * 8;
```

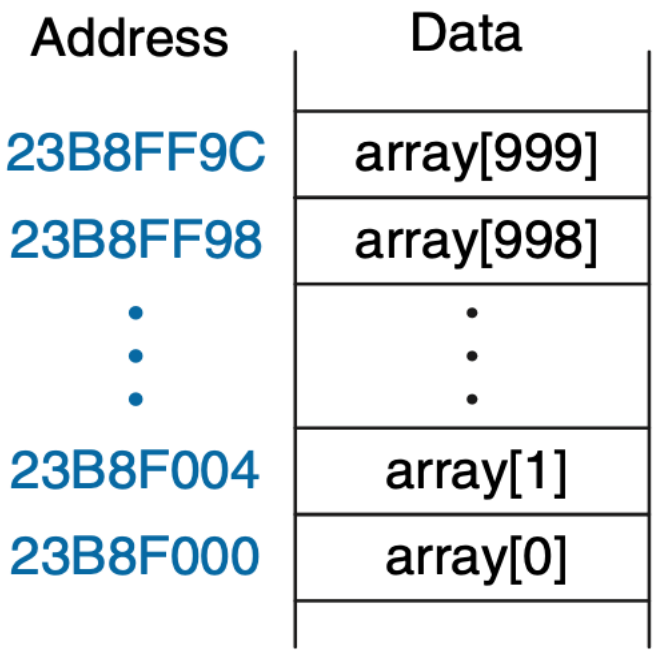



```
# Código assembly MIPS
# $s0 = endereço base do array, $s1 = i
```

Percorrendo Arrays

```
// Código C
   int array[1000];
   int i;

   for (i=0; i < 1000; i = i + 1)
       array[i] = array[i] * 8;
```




```
# Código assembly MIPS
# $s0 = endereço base do array, $s1 = i
   lui  $s0, 0x23B8          # $s0 = 0x23B80000
   ori  $s0, $s0, 0xF000     # $s0 = 0x23B8F000
   addi $s1, $0, 0           # i = 0
   addi $t2, $0, 1000        # $t2 = 1000
loop:
   slt  $t0, $s1, $t2        # i < 1000?
   beq  $t0, $0, done        # if not then done
   sll  $t0, $s1, 2          # $t0 = i * 4
   add  $t0, $t0, $s0        # address of array[i]
   lw   $t1, 0($t0)          # $t1 = array[i]
   sll  $t1, $t1, 3          # $t1 = array[i] * 8
   sw   $t1, 0($t0)          # array[i]= array[i]*8
   addi $s1, $s1, 1          # i = i + 1
   j    loop                 # repeat
done:
```

Arrays

- Nos exemplos anteriores consideramos que o array já estava carregado em memória.
- Para especificar um array de inteiros (32 bits) no assembly mips, usamos a seção `.data`, se o array for inicializado, ou `.bss` caso o array não seja inicializado.

```
.data
array:   .word 1, 2, 3, 4, 5
```

Array inicializado

```
.bss
array:   .word 1, 2, 3, 4, 5
```

Array não inicializado

Arrays

```
.data
array:   .word 1, 2, 3, 4, 5
```

Array inicializado.

```
.bss
array:   .word 1, 2, 3, 4, 5
```

Array não inicializado.

- Para recuperar o endereço base do array usamos pseudo-instrução `la`. Ela pode ser usada para carregar o endereço de qualquer label em um registrador.

```
.text
main:
    la   $t0, array
```

Exercícios

Para simulação com MARS

Exercícios

1. Escreva um programa que compare dois números inteiros (32 bits) armazenados na memória e armazene o maior deles no registrador `t0`.
2. Escreva um programa que leia um número inteiro (32 bits) armazenado na memória e determine se ele é par ou ímpar, armazenando 1 em `t0` caso seja par e 0 caso seja ímpar. (dica: resolva usando `div`).
3. Escreva um programa que leia um número inteiro (32 bits) armazenado na memória e determine se ele é par ou ímpar, armazenando 1 em `t0` caso seja par e 0 caso seja ímpar. (dica: resolva usando `andi`).
4. Escreva um programa que calcule a soma dos números de 1 até N usando a estrutura de um loop for. O valor de N deve estar armazenado em s0, e a soma final deve ser armazenada em t0.
5. Escreva um programa que receba um número inteiro em hexadecimal armazenado em s0 e calcula a soma de seus dígitos. Para implementar a extração dos dígitos, use deslocamentos e operações lógicas.

Exercícios

5. Escreva um programa que calcule a soma dos números de 1 até N usando a estrutura de um loop for. O valor de N deve estar armazenado em s0, e a soma final deve ser armazenada na memória, numa palavra referenciada pelo label `resultado`.
6. Escreva um programa que encontre o maior e o menor elemento presente em um array inicializado em memória. O menor valor deve ser salvo no registrador $s0 e o maior no registrador $s1.
7. Escreva um programa que calcule o n-ésimo número da série de Fibonacci (32 bits). O valor de n deve estar disponível no registrador `s0` e o resultado deve ser disponibilizado em `t0`.
8. Escreva um programa que calcula o fatorial de um número n armazenado em s0. O cálculo deve ser feito usando um loop e a instrução `slt`. O resultado deve ser colocado na memória no endereço referenciado pelo label `resultado`.