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

Modos de Endereçamento

Modos de Endereçamento do MIPS

- MIPS emprega cinco modos de endereçamento:
  1. **Register Only** (apenas registrador)
  2. **Immediate** (imediato)
  3. **Base Addressing** (base + offset)
  4. **PC-Relative** (PC + offset)
  5. **Pseudo-Direct** (Pseudodireto)

- Os três primeiros são usados para **endereçar operandos** para leitura ou escrita.
- Os dois últimos para **escrita do PC** (Contador de Programa).

Modos de Endereçamento: Registrador e Imediato

Register Only (apenas registrador)

- Todos os operandos são obtidos de registradores. Usado por todas as instruções Tipo-R. **Exemplos:**

```
add $s0, $t2, $t3
sub $t8, $s1, $0
```

Immediate (imediato)

- Um operando é obtido da própria instrução como um imediato de 16-bits (sign-extended), enquanto outros operandos vem de registradores. Usado por parte das instruções Tipo-I. **Exemplos:**

```
addi $s4, $t5, -73
lui  $t3, 0xFFFF
```

Modos de Endereçamento: base + offset

Base Addressing (base + offset)

- Usado por instruções de acesso à memória. O endereço do operando em memória é calculado somando o conteúdo de um registrador com uma constante (offset): Endereço base (registrador) + imediato de 16 bits (sign-extended).**Exemplo:**
  - `lw  $s4, 72($0)`
  - endereço = `$0 + 72`
  - `sw  $t2, -25($t1)`
  - endereço = `$t1 - 25`
  - `lb  $s4, 3($0)`
  - endereço = `$0 + 3`

Modos de Endereçamento: PC-Relative

Endereçamento PC-Relative (PC + offset)

- As instruções de desvio condicional (e.g., `beq` e `bnq`) usam endereçamento PC-Relative para calcular o novo valor do PC caso seja necessário desviar o fluxo de execução. São instruções Tipo-I, que instruem o processador a somar o valor do campo imediato (`imm`) ao endereço da instrução subsequente ao PC vigente, para obter o novo valor do PC.
- O imediato de 16 bits é calculado pelo assembler e fornece o **número de instruções** entre PC + 4 (instrução subsequente à de desvio) e o *label* alvo.
- O novo valor do PC, calculado pelo processador, é chamado de BTA (*Branch Target Address*): BTA = PC + 4 + (sign-extended-imm << 2).

Modos de Endereçamento: PC-Relative

Exemplo de endereçamento PC-Relative (Relativo ao PC)

```
0x10              beq   $t0, $0, else
0x14              addi  $v0, $0, 1
0x18              addi  $sp, $sp, i
0x1C              jr          $ra
0x20      else:   addi  $a0, $a0, -1
0x24              jal   factorial
```

$$BTA = 0x20$$

Código Assembly

```
beq $t0, $0, else
```

Valor dos Campos

op	rs	rt	imm
4	8	0	3
6 bits	5 bits	5 bits	16 bits

Código de Máquina

op	rs	rt	imm	
000100	01000	00000	0000000000000011	(0x11000003)
6 bits	5 bits	5 bits	16 bits	

Exercício: Modos de Endereçamento PC-Relative

1. Quem é o responsável por calcular o campo imediato de instruções de desvio condicional? Como isso é feito?
2. Calcule o campo imediato e mostre o código de máquina para a instrução branch not equal (`bne`) no programa a seguir. Explique como o processador utiliza o campo imediato para calcular o BTA.

```
0x40 loop: add  $t1, $a0, $s0
0x44       lb   $t1, 0($t1)
0x48       add  $t2, $a1, $s0
0x4C       sb   $t1, 0($t2)
0x50       addi $s0, $s0, 1
0x54       bne  $t1, $0, loop
0x58       lw   $s0, 0($sp)
```

Modos de Endereçamento: Pseudo-Direct

- No endereçamento direto, um endereço é diretamente especificado na instrução.
- Se houvesse espaço suficiente, as instruções de salto, `j` e `jal`, idealmente usariam endereçamento direto. Infelizmente, o formato das instruções tipo `j` não disponibiliza bits suficientes para especificar um JTA (*jump target address*) completo de 32 bits. Seis bits da instrução são usados para o opcode, apenas 26 bits são deixados para codificar o JTA.
- Felizmente, os dois bits menos significativos, $JTA_{1:0}$, devem ser sempre 0, porque as instruções são alinhadas com palavras. Os próximos 26 bits, $JTA_{27:2}$, são retirados do campo `addr` da instrução. Os quatro bits mais significativos, $JTA_{31:28}$, são obtidos dos quatro bits mais significativos de PC + 4.
- Este modo de endereçamento é chamado **pseudo-direto**.

Modos de Endereçamento: Pseudo-Direct

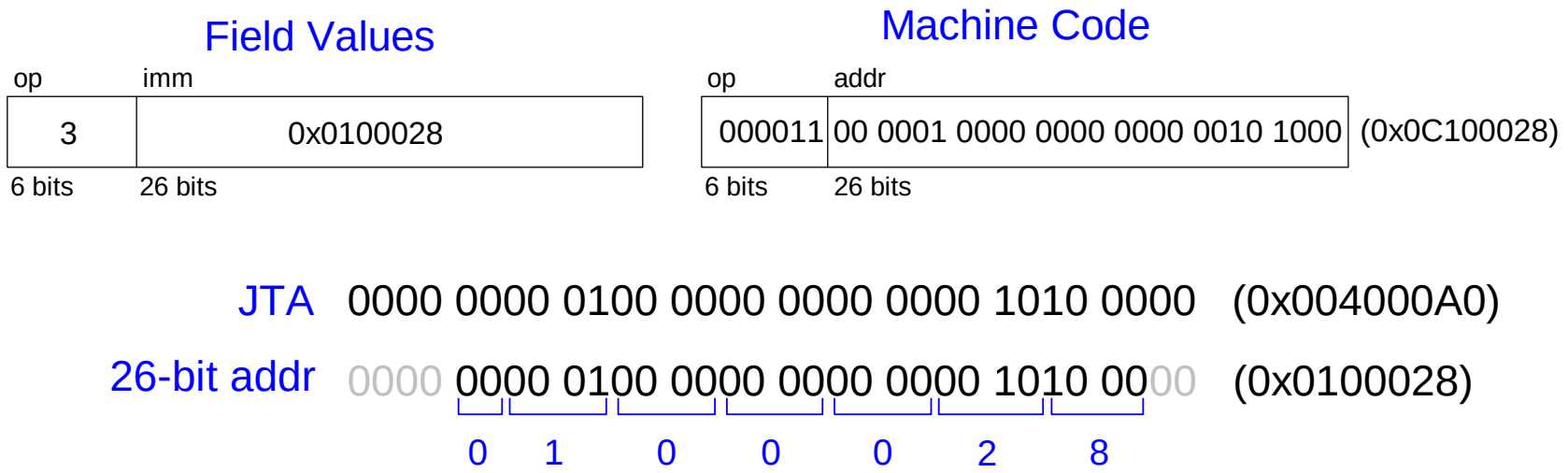
Endereçamento Pseudo-Direct (Pseudo-direto)

- Usado pelas instruções `j` e `jal` (Tipo-J).
- O processador calcula o JTA = $\{(PC + 4)_{[31:28]}, addr, 2'b0\}$
- `addr` consiste no campo imediato de 16 bits. Como os quatro bits mais significativos do JTA são retirados do PC + 4, o alcance do salto é limitado.
- Observe que a instrução `jr` não segue esse modo de endereçamento. Trata-se de uma instrução Tipo-R que salta para o endereço de 32 bits contido no registrador `rs`.

Modos de Endereçamento : Pseudo-Direct

Endereçamento Pseudo-direct (Pseudo-direto)

```
0x0040005C          jal    sum
...
0x004000A0  sum:    add    $v0, $a0, $a1
```



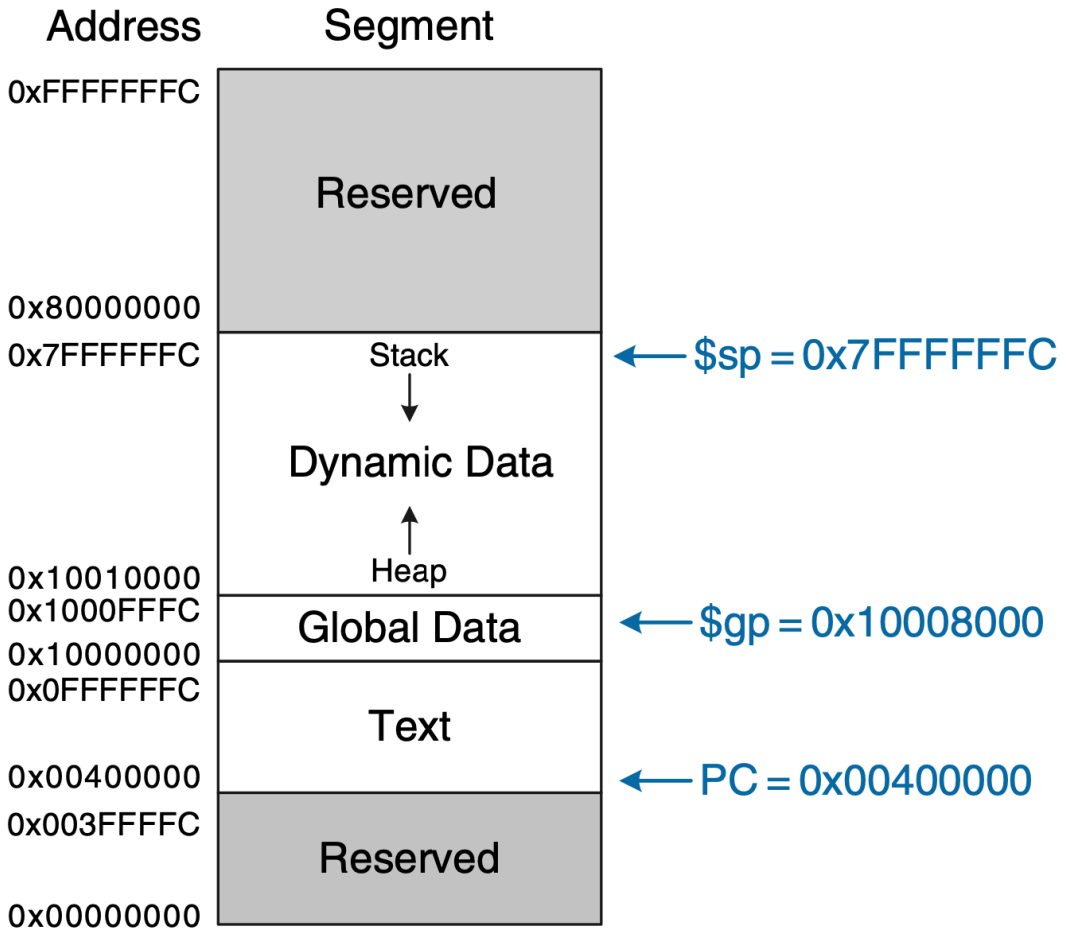
Compilação e Execução de Programas

Introdução

- Até agora, mostramos como traduzir pequenos trechos de código de alto nível em assembly e código de máquina.
- Falaremos agora sobre como compilar, montar e carregar um programa completo na memória para execução.
- Primeiramente, veremos o mapa de memória do MIPS, que define onde o código, os dados e a pilha estão localizados na memória.
- Em seguida, veremos as etapas de tradução de código para um programa de amostra.

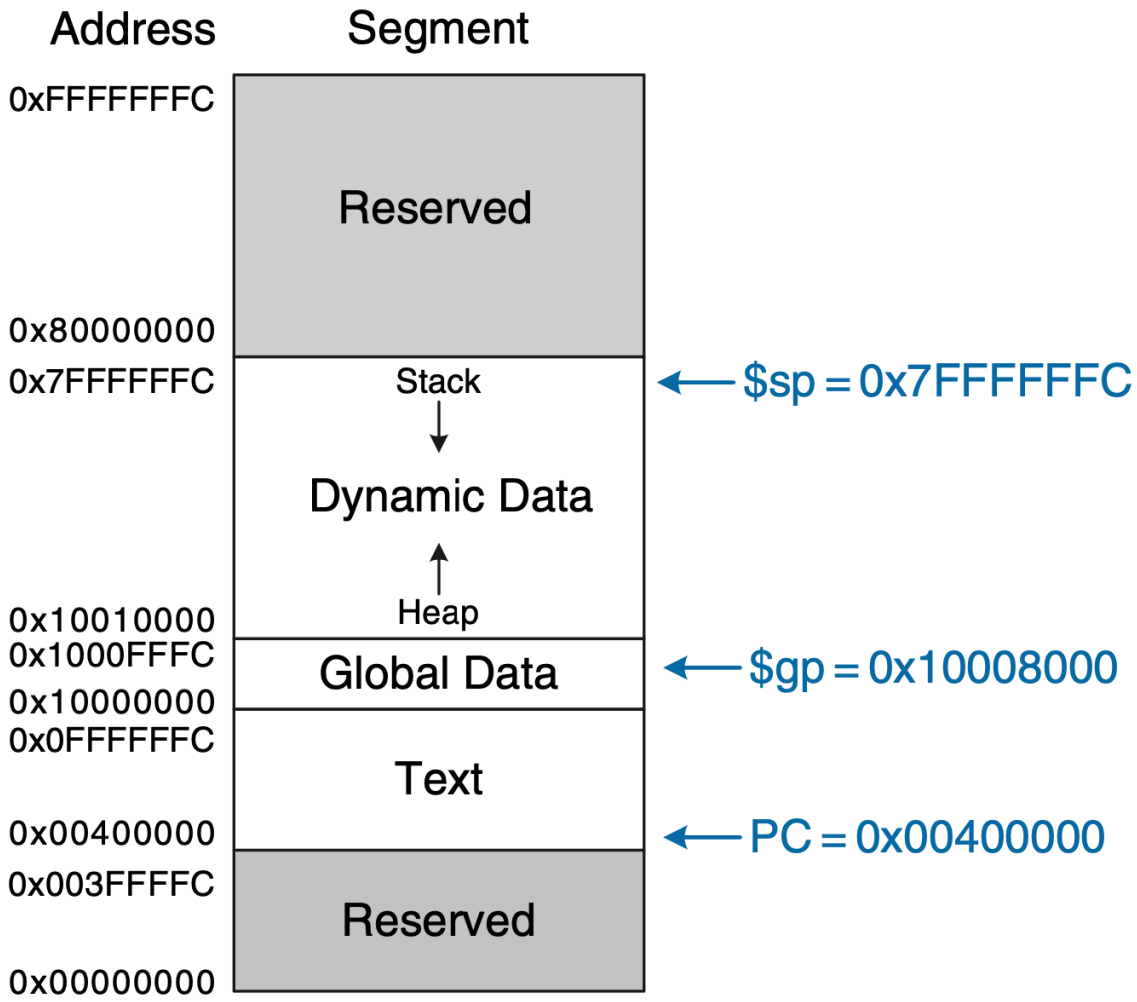
Mapa de Memória da Arquitetura do MIPS

- Memória endereçável por bytes, com endereços de 32 bits:
  - 2^{32} endereços = 4 GB.
  - Do endereço 0x00000000 até 0xFFFFFFFF.
  - Da palavra 0x00000000 à 0xFFFFFFFC
- Memória dividida quatro partes/segmentos: código (**text**), sdados globais (**Global Data**), dados alocados dinamicamente (**Heap** e **Pilha**) e segmentos reservados (**SO** e **I/O**).



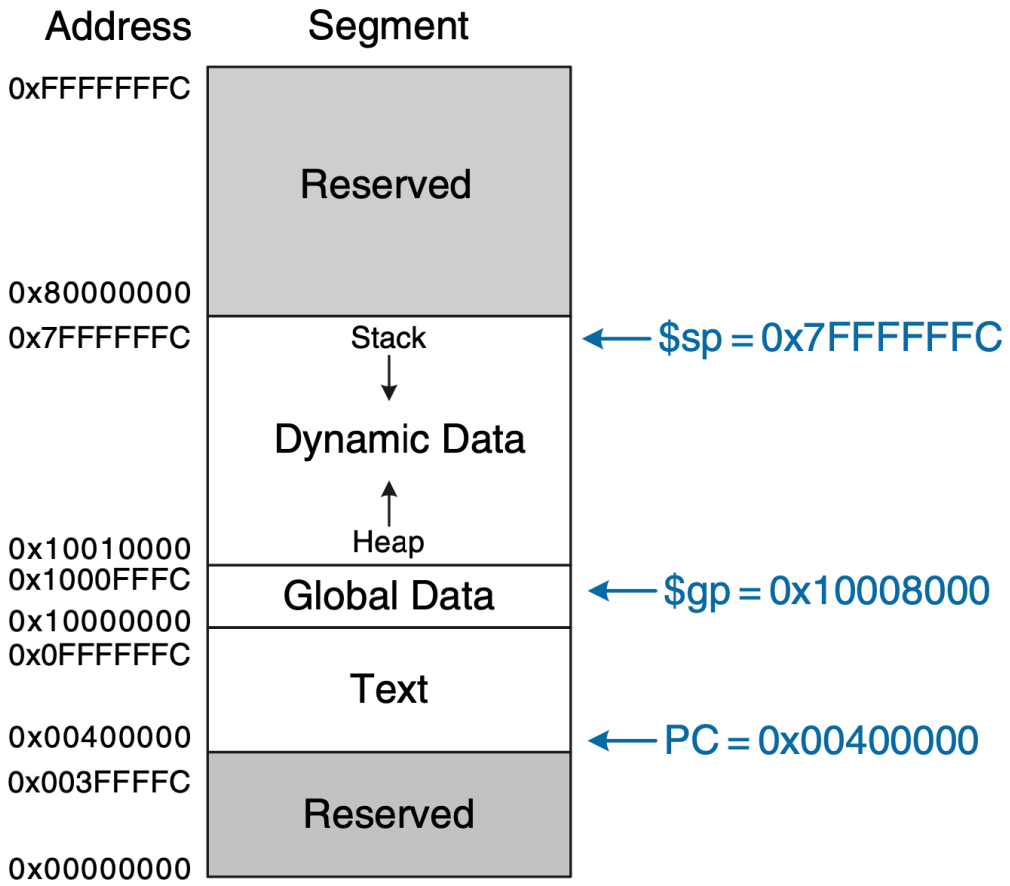
Mapa de Memória: Segmento de Texto (Text)

- Armazena o programa em linguagem de máquina (0s e 1s).
- 256 MB de espaço, do endereço 0x00400000 até 0x0FFFFFFC.
- Área endereçada pelo Contador de Programa (PC).
- Os quatro bits mais significativos são todos 0, portanto a instrução `j` e `jal` pode saltar diretamente para qualquer endereço no espaço reservado ao programa programa.



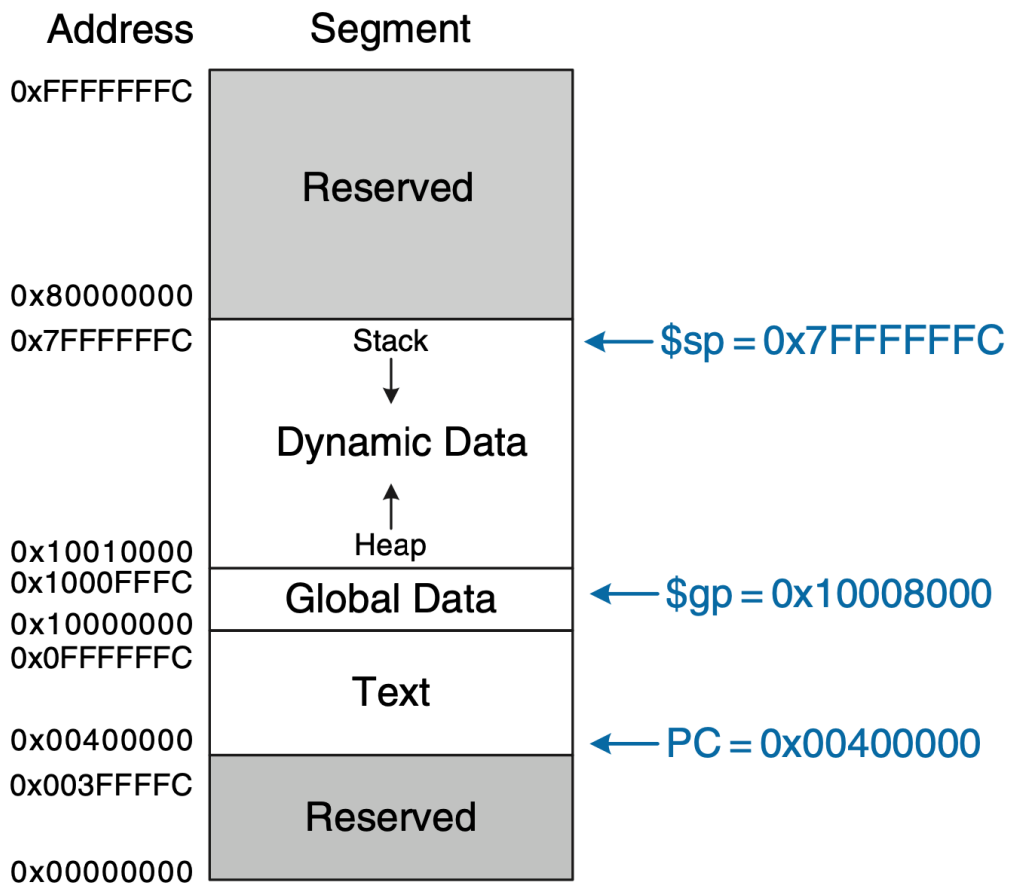
Mapa de Memória: Segmento de Dados Globais

- 64 KB usados para armazenar variáveis globais, alocadas na inicialização, antes da execução.
- Essas variáveis são acessadas tendo o registrador `$gp` como base, o qual é inicializado com 0x100080000 e não muda durante a execução do programa.
- Qualquer variável global pode ser acessada com um *offset* positivo ou negativo de 16 bits relativo a $gp. O *offset* é conhecido em tempo de montagem (assembly). Portanto, as variáveis globais podem ser acessadas de forma eficiente usando o modo de endereçamento *base* + *offset*.



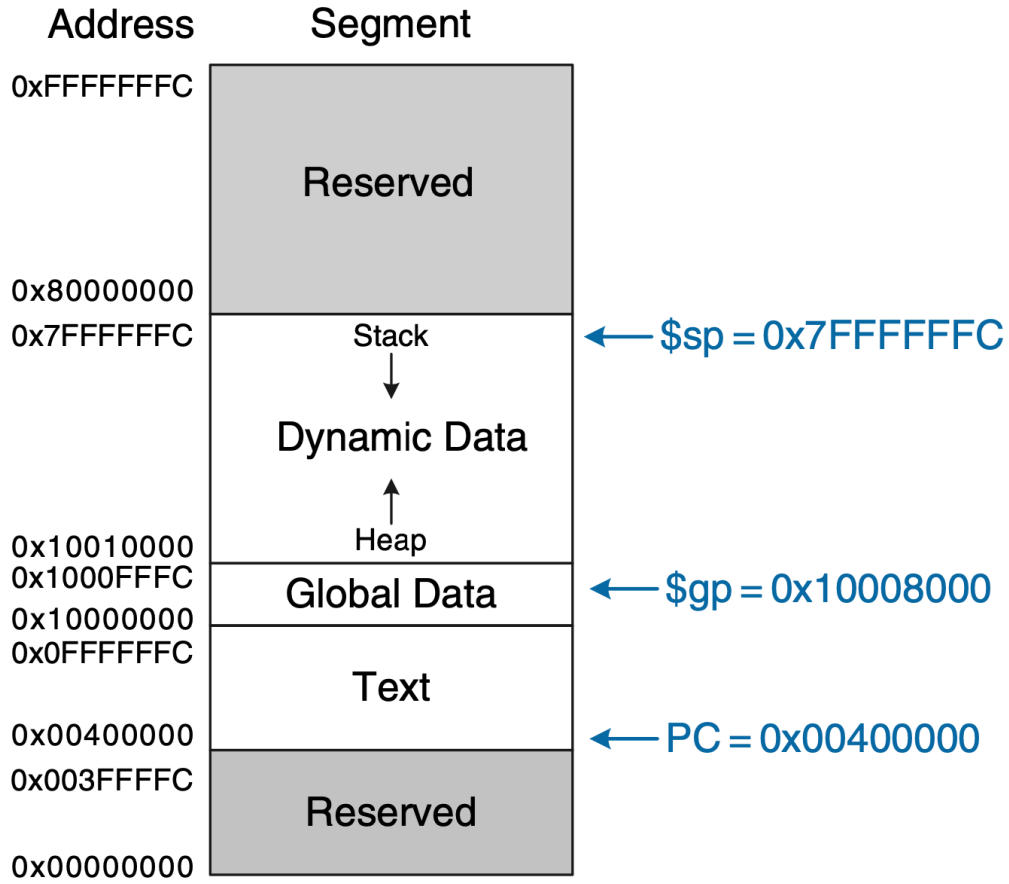
Mapa de Memória: Segmento de Dados Dinâmicos

- O segmento de dados dinâmicos abriga a **pilha** e o **heap**. Os dados deste segmento são alocados e desalocados **durante** a execução do programa.
- É o maior segmento usado por um programa, abrangendo quase 2 GB de memória.
- A pilha cresce a partir do topo do segmento de dados dinâmicos (0x7FFFFFFC) e é acessada via registrador `$sp` seguindo as convenções já discutidas.
- O heap cresce "para cima" e armazena dados alocados em tempo de execução via SO: em C com **malloc**; em C++ e Java, com **new**.



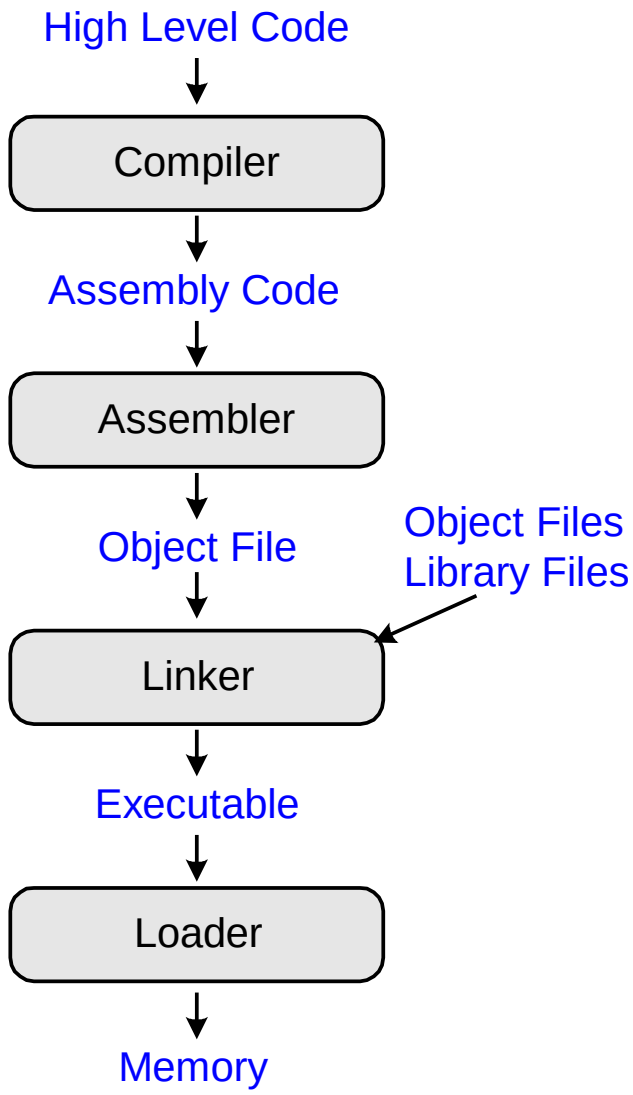
Mapa de Memória: Segmentos Reservados

- Os segmentos reservados são usados pelo Sistema Operacional e não podem ser acessados diretamente pelo programa do usuário.
- Parte da memória reservada é usada para interrupções e para I/O mapeado em memória.



Como um Programa é Compilado & Executado

- Na prática, a maioria dos compiladores executa as três primeira etapas: compilação, montagem e ligação.
- O loader normalmente é parte do Sistema Operacional, tendo como função alocar memória e configurar o ambiente de execução dos programas.



Exemplo: Código Fonte em C

```
int f, g, y;  // variáveis globais

int main(void)
{
  f = 2;
  g = 3;

  y = sum(f, g);
  return y;
}

int sum(int a, int b) {
  return (a + b);
}
```

Programa com três variáveis globais, `f`, `g` e `y`, e duas funções, `main` e `sum`.

Passo 1: Compilação.

```c
int f, g, y;  // variáveis globais

int main(void)
{
  int f = 2;
  int g = 3;

  y = sum(f, g);
  return y;
}

int sum(int a, int b) {
  return (a + b);
}
```

```
.data                    #diretiva do assembler
  f: .word 0
  g: .word 0
  y: .word 0
.text                    #diretiva do assembler
  globl main
  main:
    addi $sp, $sp, -4    # stack frame
    sw   $ra, 0($sp)     # store $ra
    addi $a0, $0, 2      # $a0 = 2
    sw   $a0, f          # f = 2
    addi $a1, $0, 3      # $a1 = 3
    sw   $a1, g          # g = 3
    jal  sum             # call sum
    sw   $v0, y          # y = sum()
    lw   $ra, 0($sp)     # restore $ra
    addi $sp, $sp, 4     # desempilha
    jr   $ra             # retorna para o SO
  sum:
    add  $v0, $a0, $a1   # $v0 = a + b
    jr   $ra             # retorna para main
```

Passo 2: Assembling

- O assembler faz duas passagens pelo programa. Na primeira, ele atribui endereços para instruções e monta a tabela de símbolos.
- Variáveis globais recebem endereços no segmento de dados globais, começando no endereço 0x10000000.
- Na segunda passagem pelo código, o assembler produz o código de máquina. Os endereços das variáveis globais são obtidos da tabela de símbolos.
- O código e a tabela de símbolos, após a primeira passagem do assembler, são mostrados ao lado.

```
0x00400000 main: addi $sp, $sp, -4
0x00400004       sw   $ra, 0($sp)
0x00400008       addi $a0, $0, 2
0x0040000C       sw   $a0, f
0x00400010       addi $a1, $0, 3
0x00400014       sw   $a1, g
0x00400018       jal  sum
0x0040001C       sw   $v0, y
0x00400020       lw   $ra, 0($sp)
0x00400024       addi $sp, $sp, 4
0x00400028       jr   $ra
0x0040002C sum:  add  $v0, $a0, $a1
0x00400030       jr   $ra
```

Symbol	Address
f	**0x10000000**
g	**0x10000004**
y	**0x10000008**
main	**0x00400000**
sum	**0x0040002C**

Passo 3: Linking

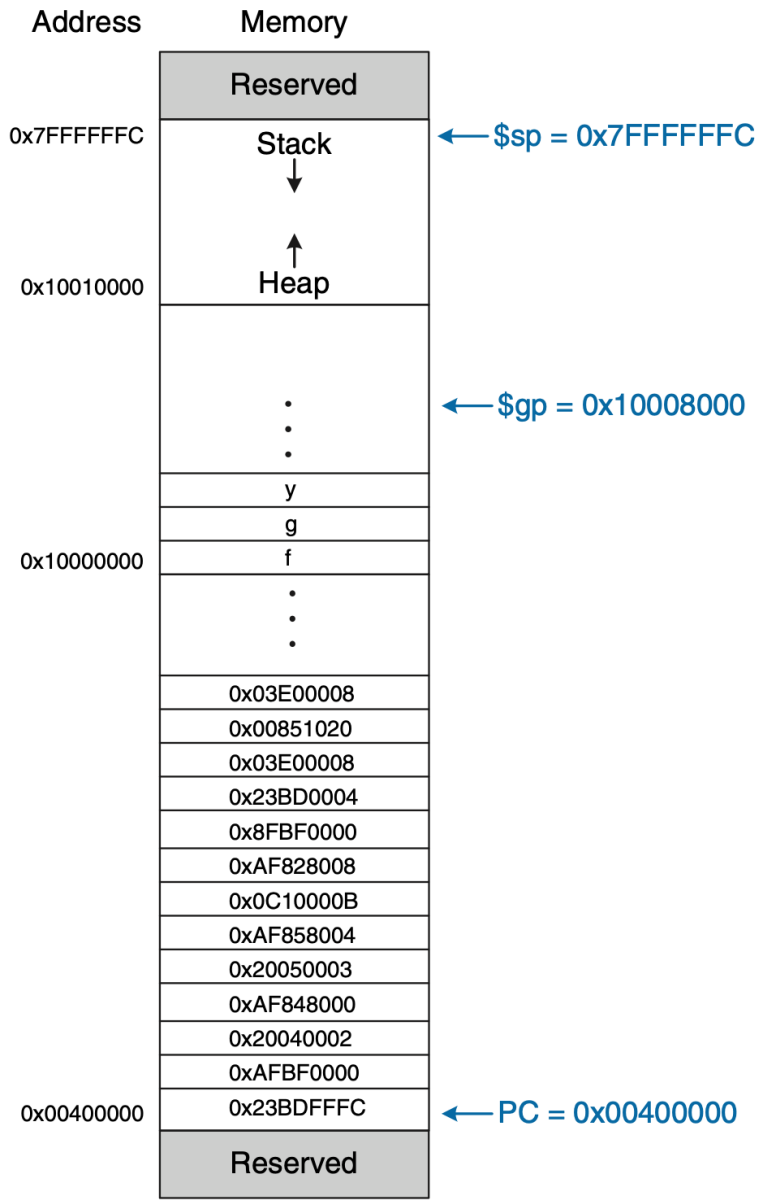
- A tarefa do linker é combinar os arquivos objeto em um arquivo *executável*. Para isso pode ser necessário fazer a **relocação de endereços**.
- Quando necessário o linker reatribui endereços a dados e instruções nos arquivos objeto para que não haja sobreposição. Para isso utiliza as informações das tabelas de símbolos.
- Em nosso exemplo, há apenas um arquivo objeto, portanto nenhuma realocação é necessária.
- Observe que o acesso a variáveis globais toma `$gp` como base.

Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)
Text segment	**Address**	**Instruction**
	0x00400000	0x23BDFFFC
	0x00400004	0xAFBF0000
	0x00400008	0x20040002
	0x0040000C	0xAF848000
	0x00400010	0x20050003
	0x00400014	0xAF858004
	0x00400018	0x0C10000B
	0x0040001C	0xAF828008
	0x00400020	0x8FBF0000
	0x00400024	0x23BD0004
	0x00400028	0x03E00008
	0x0040002C	0x00851020
	0x00400030	0x03E00008
Data segment	**Address**	**Data**
	0x10000000	f
	0x10000004	g
	0x10000008	y

```
addi $sp, $sp, -4
sw   $ra, 0 ($sp)
addi $a0, $0, 2
sw   $a0, 0x8000 ($gp)
addi $a1, $0, 3
sw   $a1, 0x8004 ($gp)
jal  0x0040002C
sw   $v0, 0x8008 ($gp)
lw   $ra, 0 ($sp)
addi $sp, $sp, -4
jr   $ra
add  $v0, $a0, $a1
jr   $ra
```

Passo 4: Loading

- O sistema operacional carrega o segmento de texto do arquivo executável para o segmento de texto da memória.
- O sistema operacional inicializa `$gp` com 0x10008000, `$sp` com 0x7FFFFFFC e, em seguida, executa a instrução `jal 0x00400000` para iniciar a execução do programa.



Pseudo-instruções

Pseudo-instruções

- Se uma instrução não estiver disponível no conjunto de instruções MIPS, provavelmente é porque a mesma operação pode ser executada usando uma ou mais instruções MIPS existentes.
- MIPS é um processador RISC, portanto a complexidade do hardware é minimizada mantendo um número pequeno de instruções simples.
- Entretanto, existem as pseudo-instruções, que não fazem parte do conjunto de instruções, mas são comumente usadas por programadores e compiladores.
- Quando convertidas em código de máquina, as pseudo-instruções são traduzidas em uma ou mais instruções MIPS.

Exemplos de Pseudo-instruções

Pseudoinstruções	Instruções MIPS
`li $s0, 0x1234AA77`	`lui $s0, 0x1234` `ori $s0, 0xAA77`
`clear $t0`	`add $t0, $0, $0`
`move $s1, $s2`	`add $s2, $s1, $0`
`nop`	`sll $0, $0, 0`
`beq $t2, imm, Loop`	`addi $at, $0, imm` `beq $t2, $at, Loop` **`#$at é reservado`**

Exceptions
(Exceções)

O Que São Exceções?

- São chamadas <u>não programadas</u> (unscheduled) ao Sistema Operacional. Mais precisamente, chamadas ao manipulador de exceções (*exception handler*).

- Podem ser causadas por Hardware ou Software:
  - Hardware, também chamadas de <u>*interrupções.*</u>
    - geradas pelo teclado, controladora de disco, etc.
  - Software, também chamadas de *traps*.
    - instruções indefinidas, divisão por zero, detecção de overflow, etc.

Tratamento de Exceções

- Quando uma exceção ocorre, o processador:
  1. Registra a causa da exceção e salva o estado do programa que estava em execução (PC e valores do Register File).
  2. Desvia para o manipulador da exceção (no MIPS sempre no endereço 0x80000180).
  3. Ao terminar, o manipulador da exceção retorna ao programa que foi interrompido.

- Nos slides a seguir essas etapas serão detalhadas, com a indicação mais precisa sobre o suporte de hardware necessário para implementação dessas etapas.

Registradores de Exceções

- Para tratar exceptions são usados registradores especiais que não são parte do Register File!
  - **Cause**: Para registrar a causa da exceção.
  - **EPC** (Exception PC): Para salvar o valor do PC no momento que a exceção ocorreu, como parte do salvamento do estado arquitetural do programa interrompido.
- Para copiar **EPC** e **Cause** o Sistema Operacional, mais precisamente o mamipulador de exceções, usa a instrução `mfc0` (Move from Coprocessor 0). Por exemplo:
  - `mfc0 $k0, EPC`
  - Move o conteúdo de EPC into $k0. Lembre-se que $k0 e $k1 são registradores do Register File reservados para o Sistema Operacional.

Causas de Exceções

Exce	Cause
Hardware Interrupt	0x00000000
System Call	0x00000020
Breakpoint / Divide by 0	0x00000024
Undefined Instruction	0x00000028
Arithmetic Overflow	0x00000030

O que acontece quando ocorre uma exceção no MIPS?

- **Passo 1: Assim que acontece uma exceção o processador.**
  1. Salva o endereço da instrução que causou a exceção no **EPC** (Registrador 14 do Coprocessador 0).
  2. Atualiza o registrador **Cause** (Registrador 13 do Coprocessador 0) para indicar o motivo da exceção.
  3. Desativa interruções e entra em modo kernel, atualizando um bit do registrador **Status** (Registrador 12 do Coprocessador 0).
  4. Desvia a execução para endereço do Manipulador de Exceções (por padrão, 0x80000180). A partir desse ponto, o Sistema Operacional está no controle.

- **Passo 2:** O kernel então salva o **Contexto do Programa**, isso inclui salvar o conteúdo do Register File na pilha do Kernel.

O que acontece quando ocorre uma exceção no MIPS?

- **Passo 3: O kernel trata a exceção.**
  1. Agora que o contexto do programa está salvo, o kernel pode analisar a exceção e decidir o que fazer. Por exemplo, se for uma divisão por zero, pode imprimir um erro e encerrar o processo.

- **Passo 4:** O kernel restaura o contexto e retorna ao programa.
  1. Se o programa puder continuar, o kernel restaura o Contexto do Programa, e volta a executá-lo de onde ele parou.

Resumo Sobre Exceções

- Processador salva a causa da exceção e o PC nos registradores Cause e EPC.
- Processador desvia para o manipulador de exceções (0x80000180).
- Manipulador de exceções:
  1. Salva registradores na pilha (do SO).
  2. Lê o registrador Cause: `mfc0 $k0, Cause`
  3. Trata a exceção
  4. Restaura registradores.
  5. Retorna ao programa:
     - `mfc0 $k0, EPC`
     - `jr $k0`

Chamada de Sistema (`syscall`)

- As instrução `syscall` causa um trap para executar uma Chamada de Sistema.

- Chamadas de Sistema são exceções programadas, que possibilitam solicitar serviços do Sistema Operacional.

- Na ocorrência de uma `syscall` o manipulador de exceção usa o EPC para localizar a instrução e determinar a qual a Chamada de Sistema solitada, examinando os campos da instrução.

Instruções Com e Sem Sinal

Adição & Subtração

- **Com Sinal:** add, addi, sub
  - Faz as mesmas operações que a versão sem sinal.
  - Mas o processador chama uma rotina de exceção na ocorrência de overflow.

- **Sem Sinal:** addu, addiu, subu
  - O processador não aciona exceção na ocorrência de overflow.

Nota: addiu sign-extends o imediato

Multiplicação & Divisão

- **Com sinal:** `mult`, `div`
- **Sem sinal:** `multu`, `divu`

- A multiplicação e a divisão comportam-se de maneira diferente para números com e sem sinal. Por exemplo, como um número sem sinal, 0xFFFFFFFF representa um número grande, mas como um número com sinal representa –1.
- Portanto, 0xFFFFFFFF × 0xFFFFFFFF seria igual a 0xFFFFFFFE00000001 se os números fossem sem sinal, mas 0x0000000000000001 se os números fossem com sinal.

Set Less Than

- **Com sinal:** `slt`, `slti`
- **Sem sinal:** `sltu`, `sltiu`

- Numa comparação com sinal, 0x80000000 é menor que qualquer outro número, porque é o número mais negativo em complemento de dois. Em uma comparação sem sinal, 0x80000000 é maior que 0x7FFFFFFF, mas menor que 0x80000001, porque todos os números são positivos.

Nota: `sltiu` sign-extends o imediato antes de compará-lo com o registrador

Loads

- **Com sinal:**
  - Sign-extends para 32-bits e carrega em um registrador.
  - Load halfword: `lh`
  - Load byte: `lb`
- **Sem sinal:**
  - Zero-extends para 32-bits e carrega em um registrador.
  - Load halfword unsigned: `lhu`
  - Load byte: `lbu`

Qual o próximo passo?

Microarquitetura

construir um processador MIPS!