Programação Assembly RISC-V

Prof. Fabrício B. Gonçalves

Instituto Federal Fluminense
Campus Bom Jesus do Itabapoana

Agenda

1 Introdução

Agenda

1. Introdução
2. Registradores RISC-V

Agenda

1. Introdução
2. Registradores RISC-V
3. Instruções RISC-V

Agenda

1. Introdução
2. Registradores RISC-V
3. Instruções RISC-V
4. Assembly RISC-V

Introdução

Introdução

A linguagem de máquina, ou código de máquina, é a linguagem de programação mais básica que existe no nível do hardware de um computador. É composta por sequências de bits (0s e 1s) que são interpretadas diretamente pelo processador de um computador.

Cada sequência de bits representa instruções específicas que comandam o computador a realizar operações elementares, como somar, subtrair, mover dados e manipular a memória.

Para a comunicação mais direta e eficiente com o hardware, a linguagem Assembly foi criada como uma camada de abstração sobre a linguagem de máquina.

Assembly é essencialmente uma representação legível por humanos do código de máquina, onde cada instrução de Assembly corresponde diretamente a uma instrução em linguagem de máquina.

Registradores RISC-V

Registradores RISC-V

RISC-V especifica 32 registradores de propósito geral, rotulados de **x0** a **x31**. Cada registrador tem 32 bits em sistemas RV32 (RISC-V 32-bit), 64 bits em sistemas RV64 (RISC-V 64-bit), e 128 bits em sistemas RV128 (RISC-V 128-bit), refletindo a capacidade de processamento de dados do processador.

A função destes registradores inclui:

- **x0:** Sempre contém o valor 0. Qualquer tentativa de escrever neste registrador não terá efeito.
- **x1 - x31:** Usados para várias operações, incluindo mas não limitado a armazenamento temporário de dados, resultados de operações aritméticas, endereços de memória para operações de carga e armazenamento, e para passar argumentos de função e valores de retorno em convenções de chamada de função.

Registradores RISC-V

Dentre os registradores de **x1** a **x31**, alguns têm usos especiais em convenções de chamada de função ou pelo sistema operacional:

- **x1 (ra):** Registrador de retorno (return address) para armazenar o endereço de retorno de chamadas de função.

  x2 (sp): Ponteiro de pilha (stack pointer), usado como o topo da pilha em convenções de chamada.

  x3 (gp): Ponteiro global (global pointer), usado para acessar dados globais.

  x4 (tp): Ponteiro de thread (thread pointer), usado para dados específicos do thread.

  x5-x7, x10-x17, x28-x31: Registradores temporários ou para passagem de argumentos em chamadas de função.

Registradores RISC-V

Continuação:

- **x8 (fp/s0):** Frame pointer ou s0, usado em gerenciamento de frames de pilha.
- **x9 (s1):** Registrador salvo, para preservar valores entre chamadas de função.
- **x18-x27 (s2-s11):** Mais registradores salvos para uso geral.

Registradores RISC-V

Em Assembly RISC-V, os registradores são tipicamente referidos por nomes numéricos (x0 a x31) ou por seus pseudônimos, que indicam usos convencionais específicos. Aqui está uma lista dos registradores em RISC-V com seus labels e usos convencionais:

- **x0 (zero):** Valor constante 0.
- **x1 (ra):** Registrador de retorno de chamada de função.
- **x2 (sp):** Ponteiro de pilha.
- **x3 (gp):** Ponteiro global.
- **x4 (tp):** Ponteiro de thread.
- **x5-x7 (t0-t2):** Temporários, não preservados entre chamadas de função.

Registradores RISC-V

Continuação:

- **x8 (s0/fp):** Ponteiro de frame ou salvo, preservado entre chamadas de função.
- **x9 (s1):** Salvo, preservado entre chamadas de função.
- **x10-x11 (a0-a1):** Argumentos de função e valores de retorno para funções.
- **x12-x17 (a2-a7):** Argumentos de função.
- **x18-x27 (s2-s11):** Salvos, preservados entre chamadas de função.
- **x28-x31 (t3-t6):** Temporários, não preservados entre chamadas de função.

Instruções RISC-V

Instruções RISC-V

Na arquitetura RISC-V, as instruções são organizadas em vários formatos, dependendo do tipo de operação que realizam.

Esses formatos são projetados para simplificar a decodificação das instruções pelo processador, mantendo a flexibilidade para suportar uma ampla gama de operações.

As instruções em RISC-V têm um tamanho fixo de 32 bits, e a divisão desses bits varia de acordo com o formato da instrução.

Instruções RISC-V

Cada formato especifica como os bits da instrução são divididos para representar diferentes partes, como o código de operação (opcode), registradores fonte e destino, constantes imediatas e offsets de endereçamento. Os principais formatos de instrução em RISC-V são:

- Formato R (Tipo Registrador);
- Formato I (Tipo Imediato);
- Formato S (Tipo Store);
- Formato B (Tipo Branch);
- Formato U (Tipo Upper Imediate);
- Formato J (Tipo Jump).

Formato R (Tipo Registrador)

Instruções RISC-V

Formato R (Tipo Registrador)

Para instruções aritméticas e lógicas que operam em dois registradores fonte e armazenam o resultado em um registrador destino.

Campos:

- **opcode (7 bits):** Especifica o tipo de operação a ser realizada. O opcode determina que a instrução é uma operação aritmética ou lógica entre registradores.
- **rd (5 bits):** Identifica o registrador destino onde o resultado da operação será armazenado.
- **funct3 (3 bits):** Um campo funcional adicional que, junto com o opcode, ajuda a determinar a operação exata a ser realizada.

Instruções RISC-V

Formato R (Tipo Registrador)

Continuação:

- **rs1 (5 bits):** Identifica o primeiro registrador fonte da operação.
- **rs2 (5 bits):** Identifica o segundo registrador fonte da operação.
- **funct7 (7 bits):** Outro campo funcional que fornece informações adicionais para determinar a operação exata, usado em operações aritméticas e lógicas que precisam de mais especificações.

Formato I (Tipo Imediato)

Instruções RISC-V

Formato I (Tipo Imediato)

Para operações que usam um registrador fonte e um valor imediato, como carga de dados e operações aritméticas imediatas.

Campos:

- **opcode (7 bits):** Define o tipo de operação, indicando que a instrução utiliza um valor imediato em sua execução.
- **rd (5 bits):** O registrador destino onde o resultado da operação será salvo.
- **funct3 (3 bits):** Auxilia na determinação da operação específica a ser realizada, junto ao opcode.
- **rs1 (5 bits):** O registrador fonte da operação.
- **imm (12 bits):** O valor imediato utilizado na operação. Este valor pode ser usado para adições, comparações ou como um offset para operações de carga e armazenamento.

Formato S (Tipo Store)

Instruções RISC-V

Formato S (Tipo Store)

Para instruções de armazenamento de dados na memória.

Campos:

- **opcode (7 bits):** Indica que a instrução é uma operação de armazenamento de dados na memória.
- **imm (12 bits, dividido):** O valor imediato que especifica o offset de endereçamento na memória a partir do endereço base fornecido pelo rs1. Dividido entre os campos superior e inferior para codificação.
- **funct3 (3 bits):** Determina o tamanho da unidade de dados a ser armazenada (por exemplo, byte, halfword, word).
- **rs1 (5 bits):** O registrador que contém o endereço base para o armazenamento de dados.
- **rs2 (5 bits):** O registrador que contém os dados a serem armazenados na memória.

Formato B (Tipo Branch)

Instruções RISC-V

Formato B (Tipo Branch)

Para instruções de desvio condicional baseadas na comparação de dois registradores.

Campos:

- **opcode (7 bits):** Especifica que a instrução é um desvio condicional.
- **imm (12 bits, dividido):** O valor imediato que representa o offset de desvio caso a condição seja verdadeira. Este valor é dividido e codificado em vários campos.
- **funct3 (3 bits):** Define a condição de desvio (por exemplo, igualdade, menor que).
- **rs1 e rs2 (5 bits cada):** Os registradores cujos valores serão comparados para determinar se o desvio será tomado.

Formato U (Tipo Upper Imediate)

Instruções RISC-V

Formato U (Tipo Upper Imediate)

Para carregar um valor imediato de 20 bits no registrador destino, usado principalmente para operações de alto nível e configuração de endereços.

Campo:

- **opcode (7 bits):** Indica que a instrução carrega um valor imediato de 20 bits no registrador destino.
- **rd (5 bits):** O registrador destino para o valor imediato.
- **imm (20 bits):** O valor imediato de 20 bits que é carregado na parte superior do registrador destino, geralmente usado para formar endereços ou constantes grandes.

Formato R (Tipo Jump)

Instruções RISC-V

Formato J (Tipo Jump)

Para instruções de salto incondicional, com um endereço de destino calculado a partir de um valor imediato.

Campo:

- **opcode (7 bits):** Denota uma instrução de salto incondicional.
- **rd (5 bits):** O registrador onde o endereço de retorno (normalmente o endereço da instrução seguinte) será armazenado, geralmente x1 para chamadas de função.
- imm (20 bits): O valor imediato que representa o offset de salto a partir do endereço atual. Esse valor é utilizado para calcular o endereço de destino do salto.

Assembly RISC-V

Organização de um Programa em Assembly RISC-V

Assembly RISC-V

Organização de um Programa em Assembly RISC-V

A organização de um programa em Assembly RISC-V segue um padrão que facilita a leitura, a manutenção e o entendimento do código, tanto pelo programador quanto pelas ferramentas de desenvolvimento.

Embora a estrutura exata possa variar conforme as necessidades específicas do projeto e as preferências do desenvolvedor, uma organização típica inclui várias seções:

- Seção de dados;
- Seção de texto;
- Declaração do ponto de entrada;
- Intruções.

Assembly RISC-V

Organização de um Programa em Assembly RISC-V: Seção de Dados

A seção de dados é usada para declarar variáveis estáticas e inicializar constantes que seu programa irá usar.

Variáveis e constantes são alocadas aqui com nomes simbólicos, facilitando o acesso no restante do programa.

Código

```
.data

    msg: .asciz "Ola, mundo!\n" #Declara variavel do tipo string

    numero: .word 123 #Declara variavel do tipo inteiro
```

Assembly RISC-V

Organização de um Programa em Assembly RISC-V: Seção de Dados

A seção de texto contém o código executável do programa. É aqui que as instruções do programa são escritas.

Esta seção começa com a diretiva **.text** e geralmente inclui uma etiqueta (label) para o ponto de entrada do programa, frequentemente denominado main em programas simples.

Código

```
.text
    li a7 4 #Carrega codigo de chamada de sistema PrintString

    la a0 msg #Carrega a string armazenada no endereco de msg

    ecall #Realiza a chamada de sistema
```

Assembly RISC-V

Organização de um Programa em Assembly RISC-V: Declaração do Ponto de Entrada

O ponto de entrada do programa é onde a execução começa. Em muitos programas Assembly RISC-V, isso é explicitamente marcado com um label **main:**. Esse ponto de entrada é frequentemente declarado como global para que o *linker* possa identificá-lo.

Código

```
.text

.global main

main:
    li a7 4 #Carrega codigo de chamada de sistema PrintString

    la a0 msg #Carrega a string armazenada no endereco de msg

    ecall #Realiza a chamada de sistema
```

Assembly RISC-V

Organização de um Programa em Assembly RISC-V: Instruções

Aqui você coloca as instruções que seu programa executará, começando pelo ponto de entrada.

Isso inclui operações lógicas e aritméticas, controle de fluxo, chamadas de função e manipulação de dados.

Código

```
main:
    li a7 4 #Carrega codigo de chamada de sistema PrintString

    la a0 msg #Carrega a string armazenada no endereco de msg

    ecall #Realiza a chamada de sistema
```

Tipos de Dados Básicos

Assembly RISC-V

Tipos de Dados Básicos

Como toda linguagem de programação, o Assembly RISC-V também possui tipos básicos de dados. Estes tipos são os seguintes:

- Inteiros;
- Números de ponto flutuante;
- Caracteres e Strings;
- Vetores (Arrays).

Assembly RISC-V

Tipos de Dados Básicos: Inteiros

Para inteiros de 32 bits (em um sistema RV32) ou 64 bits (em um sistema RV64), deve-se usar as diretivas **.word** ou **.dword** respectivamente para reservar espaço e inicializar as variáveis.

Código

```
.data
    varInt32: .word 123
    varInt64: .dword 123456789
```

Assembly RISC-V

Tipos de Dados Básicos: Números de Ponto Flutuante

Para números de ponto flutuante, use **.float** para precisão simples e **.double** para precisão dupla.

Código

```
.data
    varFloat: .float 1.23
    varDouble: .double 1.23456789
```

Assembly RISC-V

Tipos de Dados Básicos: Caracteres e Strings

Para caracteres e strings, você pode usar **.byte** para um único caractere e **.asciz** ou **.string** para strings (sequências de caracteres terminadas em null).

Código

```
.data
    varChar: .byte "A"
    varString: .asciiz "Variavel String"
```

Assembly RISC-V

Tipos de Dados Básicos: Vetores (Arrays)

Arrays podem ser declarados especificando o tipo de dado e a quantidade. Para um array de inteiros, você repetiria a diretiva de tipo o número necessário de vezes ou usaria uma diretiva para reservar um bloco de espaço diretamente.

Código

```
.data
    arrayInt: .word  1, 2, 3, 4
```

Para um array de tamanho fixo, mas sem inicializar os valores, você pode usar **.space** para reservar uma quantidade específica de espaço em bytes.

Código

```
.data
    array: .space
```

Carregamento e Armazenamento de Dados

Assembly RISC-V

Carregamento e Armazenamento de Dados

Em RISC-V, não há instruções que operam diretamente em valores na memória.

Os valores precisam ser carregados primeiramente nos registradores para que, em seguida, as instruções possam ser executadas.

Assembly RISC-V

Carregamento e Armazenamento de Dados

Aqui estão as instruções básicas de carregamento de dados em Assembly RISC-V, juntamente com uma breve descrição do que cada uma faz:

- **Inteiros:**
  - **lb:** Carrega um byte da memória para o registrador, com extensão de sinal.
  - **lh:** Carrega uma halfword (2 bytes) da memória para o registrador, com extensão de sinal.
  - **lw:** Carrega uma word (4 bytes) da memória para o registrador.
  - **lbu:** Carrega um byte da memória para o registrador, sem extensão de sinal (zero-extend).
  - **lhu:** Carrega uma halfword (2 bytes) da memória para o registrador, sem extensão de sinal (zero-extend).
  - **ld:** (somente RV64I): Carrega uma doubleword (8 bytes) da memória para o registrador.

Assembly RISC-V

Carregamento e Armazenamento de Dados

Continuação:

- **Números de Pontos Flutuantes:**
  - **flw:** Carrega um valor de ponto flutuante de precisão simples (32 bits) da memória para um registrador de ponto flutuante.
  - **fld:** Carrega um valor de ponto flutuante de precisão dupla (64 bits) da memória para um registrador de ponto flutuante.
- **Endereços de Memória:**
  - **la:** Carrega o valor de um endereço em um registrador.

Assembly RISC-V

Carregamento e Armazenamento de Dados

Aqui estão as instruções básicas de armazenamento de dados em Assembly RISC-V, juntamente com uma breve descrição do que cada uma faz:

- **Inteiros:**
  - **sb:** Store Byte - Armazena o byte menos significativo de um registrador na memória.
  - **sh:** Store Halfword - Armazena os dois bytes menos significativos de um registrador na memória.
  - **sw:** Store Word - Armazena os quatro bytes de um registrador na memória.

    sd: Store Doubleword - Armazena os oito bytes de um registrador na memória (apenas em RV64I).
- **Endereços de Memória:**

Assembly RISC-V

Carregamento e Armazenamento de Dados

Continuação:

- **Números de Ponto Flutuante:**
  - **fsw:** Store Word Floating-Point - Armazena um valor de ponto flutuante de precisão simples de um registrador de ponto flutuante na memória.
  - **fsd:** Store Doubleword Floating-Point - Armazena um valor de ponto flutuante de precisão dupla de um registrador de ponto flutuante na memória.

Chamadas de Sistema

Assembly RISC-V

Chamadas de Sistema

Invocar uma chamada de sistema (syscall) em RISC-V envolve o uso de uma instrução especial chamada ecall.

Chamadas de sistema são usadas para solicitar serviços do kernel do sistema operacional, como operações de entrada/saída, criação e gerenciamento de processos, e comunicação entre processos.

O procedimento exato para realizar uma chamada de sistema varia dependendo do sistema operacional (por exemplo, Linux), mas o conceito básico é similar entre diferentes ambientes.

Assembly RISC-V

Chamadas de Sistema

Os passos para invocar uma chamada de sistemas são os seguintes:

1. **Definir o Número da Chamada de Sistema:** Cada chamada de sistema tem um número único associado. Esse número deve ser colocado em um registrador específico. Em sistemas Linux sobre RISC-V, o número da chamada de sistema é colocado no registrador **a7**.

2. **Configurar Argumentos:** Se a chamada de sistema requer argumentos, eles devem ser colocados nos registradores **a0** a **a6**, conforme a convenção de chamadas do sistema operacional. Por exemplo, **a0** pode ser usado para um descritor de arquivo, a1 para o endereço de um buffer de dados, e assim por diante.

Assembly RISC-V

Chamadas de Sistema

Os passos para invocar uma chamada de sistemas são os seguintes:

3. **Executar a Instrução *ecall*:** A instrução ecall é usada para invocar a chamada de sistema. O kernel do sistema operacional lê o número da chamada de sistema do registrador **a7**, interpreta os argumentos dos registradores **a0** a **a6**, e executa a operação solicitada.

4. **Obter o Resultado:** Após a execução da chamada de sistema, o resultado (se houver) é geralmente retornado no registrador **a0**. Se a chamada de sistema falhar, o valor retornado pode indicar o tipo de erro ocorrido.

Operações Aritméticas

Assembly RISC-V

Operações Aritméticas

Aqui estão as principais operações aritméticas em Assembly RISC-V, juntamente com uma breve descrição do que cada uma faz:

- **Operações com Inteiros:**
  - **add:** Soma dois registradores.
  - **sub:** Subtrai um registrador de outro.
  - **mul:** Multiplica dois registradores.
  - **div:** Divide um registrador por outro, resultando no quociente.
  - **rem:** Divide um registrador por outro, resultando no resto.
  - **addi:** Soma um registrador com um valor imediato.

Assembly RISC-V

Operações Aritméticas

Continuação:

- **Operações com Números de Ponto Flutuante:**
  - **Precisão Simples (Extensão "F"):**
    - **fadd.s:** Soma de ponto flutuante de precisão simples.
    - **fsub.s:** Subtração de ponto flutuante de precisão simples.
    - **fmul.s:** Multiplicação de ponto flutuante de precisão simples.
    - **fdiv.s:** Divisão de ponto flutuante de precisão simples.
    - **fsqrt.s:** Raiz quadrada de ponto flutuante de precisão simples.
    - **fmin.s:** Mínimo de dois números de ponto flutuante de precisão simples.
    - **fmax.s:** Máximo de dois números de ponto flutuante de precisão simples.

Assembly RISC-V

Operações Aritméticas

Continuação:

- **Operações com Números de Ponto Flutuante:**
  - **Precisão Simples (Extensão "D"):**
    - **fadd.d:** Soma de ponto flutuante de precisão dupla.
    - **fsub.d:** Subtração de ponto flutuante de precisão dupla.
    - **fmul.d:** Multiplicação de ponto flutuante de precisão dupla.
    - **fdiv.d:** Divisão de ponto flutuante de precisão dupla.
    - **fsqrt.d:** Raiz quadrada de ponto flutuante de precisão dupla.
    - **fmin.d:** Mínimo de dois números de ponto flutuante de precisão dupla.
    - **fmax.d:** Máximo de dois números de ponto flutuante de precisão dupla.

Exemplos 1

Assembly RISC-V

Exemplos 1

Somando dois números

```
.data
  msg_1: .asciz "Informe o primeiro numero: "
  msg_2: .asciz "Informe o segundo numero: "
  resultado: .asciz "Resultado: "
.text
.global main
main:
  li a7 4
  la a0 msg_1
  ecall #Imprime menssagem para primeiro numero
  li a7 5
  ecall #Leitura do teclado do segundo numero
  mv t0 a0 #Copia o valor de a0 e t0
  li a7 4
  la a0 msg_1
  ecall #Imprime menssagem para segundo numeroe
  li a7 5
  ecall #Leitura do teclado do segundo numero
  mv t1 a0 #Copia o valor de a0 e t1
  ...
```

Assembly RISC-V

Exemplos 1

Somando dois números

```
...
li a7 4
la a0 resultado
ecall  #Imprime mensagem de resultado
li a7 1
mv a0 t2 #Copia o valor de t2 para a0
ecall #Imprime um inteiro
```

Desvios Condicionais

Assembly RISC-V

Desvios Condicionais

Aqui estão as principais operações aritméticas em Assembly RISC-V, juntamente com uma breve descrição do que cada uma faz:

- **Operações com Inteiros:**
  - **beq:** Branch if Equal - Desvia se dois registradores são iguais.
  - **bne:** Branch if Not Equal - Desvia se dois registradores não são iguais.
  - **blt:** Branch if Less Than - Desvia se o primeiro registrador é menor que o 0segundo (sinalizado).
  - **bge:** Branch if Greater Than or Equal - Desvia se o primeiro registrador é maior ou igual ao segundo (sinalizado).
  - **bltu:** Branch if Less Than Unsigned - Desvia se o primeiro registrador é menor que o segundo (não sinalizado).
  - **bgeu:** Branch if Greater Than or Equal Unsigned - Desvia se o primeiro registrador é maior ou igual ao segundo (não sinalizado).

Assembly RISC-V

Desvios Condicionais

Aqui estão as principais operações de desvio condicional em Assembly RISC-V, juntamente com uma breve descrição do que cada uma faz:

- **Operações com Números de Ponto Flutuante:**
  - **Precisão Simples (Extensão "F"):**
    - **feq.s:** Floating-point Equal Single-precision - Compara se dois números de ponto flutuante de precisão simples são iguais.
    - **flt.s:** Floating-point Less Than Single-precision - Compara se um número de ponto flutuante de precisão simples é menor que outro.
    - **fle.s:** Floating-point Less than or Equal Single-precision - Compara se um número de ponto flutuante de precisão simples é menor ou igual a outro.

Assembly RISC-V

Desvios Condicionais

Continuação:

- **Operações com Números de Ponto Flutuante:**
  - **Precisão Dupla (Extensão "D"):**
    - **feq.d:** Floating-point Equal Single-precision - Compara se dois números de ponto flutuante de precisão dupla são iguais.
    - **flt.d:** Floating-point Less Than Single-precision - Compara se um número de ponto flutuante de precisão dupla é menor que outro.
    - **fle.d:** Floating-point Less than or Equal Single-precision - Compara se um número de ponto flutuante de precisão dupla é menor ou igual a outro.

Desvios Incondicionais

Assembly RISC-V

Desvios Incondicionais

Aqui estão as principais operacões de desvios incodicionais em Assembly RISC-V, juntamente com uma breve descrição do que cada uma faz:

- **j:** Salto incondicional para uma etiqueta.
- **jal:** Salta para um endereço e salva o endereço de retorno no registrador ra.

Exemplos 2

Assembly RISC-V

Exemplos 2

Comparando números

```
.data
  msg_1: .asciz "Informe o primeiro numero: "
  msg_2: .asciz "Informe o segundo numero: "
  resultado_1: .asciz " e maior que "
  resultado_2: .asciz " e igual a "
  resultado_3: .asciz " e menor que "
.text
.global main

main:

  li a7 4
  la a0 msg_1
  ecall

  li a7 5
  ecall
  mv t0 a0

  li a7 4
```

Assembly RISC-V

Exemplos 2

Comparando números (cont.)

```
  li a7 5
  ecall
  mv t1 a0

  blt t1 t0 t0_maior_t1 #Verifica se t0 e maior que t1
  blt t0 t1 t0_menor_t1 #Verifica se t0 e maior que t1
  j t0_igual_t1

t0_maior_t1:
  li a7 1
  mv a0 t0
  ecall

  li a7 4
  la a0 resultado_1
  ecall

  li a7 1
  mv a0 t1
  ecall
```

Assembly RISC-V

Exemplos 2

Comparando números (cont.)

```
t0_menor_t1:
  li a7 1
  mv a0 t0
  ecall

  li a7 4
  la a0 resultado_3
  ecall

  li a7 1
  mv a0 t1
  ecall

  j sair
```

Assembly RISC-V

Exemplos 2

Comparando números (cont.)

```
t0_igual_t1:
  li a7 1
  mv a0 t0
  ecall

  li a7 4
  la a0 resultado_2
  ecall

  li a7 1
  mv a0 t1
  ecall

  j sair

sair:
  li a7 10
  ecall
```

Exercícios 1

Assembly RISC-V

Exercícios 1

- Implemente um programa que leia dois números e um código de operação (1 para adição, 2 para subtração, 3 para multiplicação, 4 para divisão). Execute a operação correspondente e mostre o resultado. Implemente verificações para prevenir a divisão por zero.

- Escreva um programa que, dado um número que representa a idade de uma pessoa, identifique sua faixa etária: criança (0-12), adolescente (13-18), adulto (19-59) ou idoso (60+). Utilize desvios condicionais para direcionar para o bloco de código apropriado.

- Desenvolva um programa que determine o maior de três números. Se todos os números forem iguais, armazene um valor específico que indique essa igualdade.

Assembly RISC-V

Exercícios 1

- Crie um programa que simula um controle de direção básico com quatro direções (Norte=0, Sul=1, Leste=2, Oeste=3). Para uma entrada de direção, o programa deve indicar a ação correspondente (por exemplo, "Mover para o norte").

- Implemente um programa que verifique se um número fornecido é uma potência de dois. O programa deve armazenar 1 em um registrador se o número for uma potência de dois, e 0 se não for.

- Escreva um programa que calcule o discriminante (b^2-4ac) de uma equação quadrática e indique se as raízes são reais e distintas, reais e iguais, ou complexas.

Assembly RISC-V

Exercícios 1

- Desenvolva um programa que converta a temperatura de Celsius para Fahrenheit e vice-versa, baseado em um código de operação. Inclua verificações para entradas válidas.
- Implemente um programa que teste se um número n é divisível por outro número m, sem deixar resto. Utilize operações aritméticas para encontrar o resto da divisão.
- Crie um programa que classifique um caractere (valor ASCII) como letra maiúscula, letra minúscula, dígito ou símbolo especial. Utilize a tabela ASCII e desvios condicionais para a classificação.

Assembly RISC-V

Exercícios 1

- Escreva um programa que, dadas três notas, calcule a média e a classifique em categorias ("Excelente"para médias acima de 90, "Bom"para médias entre 70 e 89, "Satisfatório"para médias entre 50 e 69, e "Insuficiente"para médias abaixo de 50).

Operações Lógicas

Operações Lógicas

Operações lógicas são fundamentais em qualquer linguagem de programação ou conjunto de instruções de uma arquitetura de processador, e RISC-V não é exceção.

Essas operações permitem manipular dados em nível de bit, o que é crucial para muitas tarefas de programação de baixo nível.

Operações Lógicas

As operações lógicas básicas são as seguintes:

- **and:** Esta instrução realiza uma operação lógica "E"bit a bit entre dois registradores. O resultado é armazenado em um terceiro registrador. Se ambos os bits em uma posição específica dos operandos forem 1, o bit resultante nessa posição será 1; caso contrário, será 0.
- **or:** Realiza uma operação lógica "OU"bit a bit. Se pelo menos um dos bits em uma posição específica dos operandos for 1, o bit resultante nessa posição será 1.
- **xor:** Executa uma operação "OU EXCLUSIVO"bit a bit. Se os bits em uma posição específica dos operandos forem diferentes, o bit resultante nessa posição será 1. Se forem iguais, o resultado será 0.

Operações Lógicas

Exemplo

```
.data

.text

.global main

main:
   li t0 14 #1110
   andi t1 t0 13 #1101
   mv a0 t1
   li a7 1
   ecall #Imprime o inteiro 12

   ori t1 t0 1#0001
   mv a0 t1
   li a7 1
   ecall #Imprime o inteiro 15

   xori t1 t0 13
   mv a0 t1
   li a7 1
```

Operações de Deslocamento de Bits

Operações de Deslocamento de Bits

As instruções de deslocamento de bits são essenciais na arquitetura RISC-V, permitindo a manipulação eficiente de dados ao nível de bit.

Essas operações são amplamente utilizadas para otimização de cálculos matemáticos, manipulação de campos de bits, e outras operações de baixo nível que requerem precisão ao nível de bit.

Em RISC-V, existem várias instruções de deslocamento, cada uma adequada para diferentes casos de uso. As instruções podem ser categorizadas em deslocamentos lógicos e aritméticos, e podem operar com valores imediatos (fixos) ou variáveis (contidos em registradores).

Operações de Deslocamento de Bits

Aqui estão as principais operações de deslocamento lógico e artimético em RISC-V e como elas são usadas:

- **sll**: Desloca bits para a esquerda, inserindo zeros à direita. É usado para multiplicar por potências de 2.
- **srl**: Desloca bits para a direita, inserindo zeros à esquerda. Utilizado para divisão por potências de 2, sem preservar o sinal.
- **sra**: Desloca bits para a direita, inserindo cópias do bit mais significativo (bit de sinal) nos espaços à esquerda. Serve para divisão por potências de 2, preservando o sinal do número.

Operações de Deslocamento

Exemplo

```
.data

.text

.global main

main:
   li t0 4
   slli t1 t0 3 #Multiplica 4 por 8 (2 elevado a 3)
   li a7 1
   mv a0 t1
   ecall #Imprime o inteiro 32

   mv t0 t1
   srli t1 t0 2 #Divide 32 por 4 (2 elevado a 2)
   li a7 1
   mv a0 t1
   ecall #Imprime o inteiro 8
```

Comparando Valores

Comparando Valores

Aqui estão as principais operações de comparação em RISC-V e como elas são usadas:

- **slt:** Compara dois registradores e define o registrador de destino como 1 se o primeiro for menor que o segundo, considerando valores com sinal.
- **sgt:** Compara dois registradores e define o registrador de destino como 1 se o primeiro for maior que o segundo, considerando valores com sinal.

Repetições

Repetições

As estruturas de repetição, ou loops, em Assembly RISC-V, são implementadas usando instruções de desvio (branch) condicionais e incondicionais.

Ao contrário de linguagens de alto nível que possuem palavras-chave específicas para loops (como for, while, do-while), em Assembly, é necessáro criar a lógica de loop manualmente.

Isso envolve configurar e testar condições, e usar instruções de desvio para repetir um bloco de código.

Assembly RISC-V

Repetições: Exemplo de Implementação WHILE

Implementação do WHILE

```
.data

.text

.global main

main:
   li t0 10

while_loop:
   # Verifica a condicao do loop: se t0 <= 0, saia do loop.
   blez t0, end_while  # Se t0 for menor ou igual a zero, pule para o
     fim.
   # Corpo do loop: (faca algo aqui).
   # Por exemplo, vamos apenas decrementar t0 por agora.
   addi t0, t0, -1
   # Volta ao inicio do loop para verificar a condicao novamente.
   j while_loop

end_while:
```

Assembly RISC-V

Repetições: Implementação FOR

Implementar um loop for em Assembly RISC-V envolve definir explicitamente a inicialização, a condição de continuação e o incremento (ou decremento), que são os três componentes principais de um loop for em linguagens de alto nível.

Diferente dessas linguagens, em Assembly, você precisa manualmente gerenciar estas etapas e o desvio condicional para controlar o fluxo do programa.

Assembly RISC-V

Repetições: Exemplo de Implementação FOR

Implementação do FOR

```
.data

.text

.global main

main:
   li t0 0

for_loop:
    li t1, 10      # t1 e o limite do nosso loop, 10 neste caso
    bge t0, t1, end_for  # Condicao de Continuacao: Se t0 >= 10, saia do
     loop
    # Corpo do loop: (faca algo aqui)
    addi t0, t0, 1 # Incremento: t0 = t0 + 1
    j for_loop     # Salto incondicional de volta para o inicio do loop

end_for:
    # O codigo aqui sera executado apos a conclusao do loop
```

Assembly RISC-V

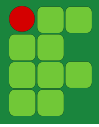
Repetições: Implementação DO..WHILE

A estrutura de loop do..while executa o corpo do loop pelo menos uma vez antes de verificar a condição para repetir o loop.

Para implementar um loop do..while em Assembly RISC-V, você seguirá um padrão similar ao de outros loops, com a diferença principal de que a condição é verificada após o corpo do loop ter sido executado.

Assembly RISC-V

Repetições: Implementação DO..WHILE



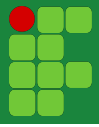
Exemplo de Implementação do DO..WHILE

```
.data

.text

.global main

main:
   li t0 10

do_loop:
    # Corpo do loop: (faca algo aqui)
    # Por exemplo, vamos apenas decrementar t0 por agora.
    addi t0, t0, -1

    # Verifica a condicao do loop: se t0 nao e igual a 0, repita o loop.
    # Como estamos contando para baixo, queremos continuar ate t0 atingir
      0.
    bnez t0, do_loop  # Se t0 nao e zero, salta de volta para o inicio do
      loop.
```

Funções

Assembly RISC-V

Funções



A implementação de chamadas de funções ou procedimentos em RISC-V segue um conjunto de convenções para passar argumentos, retornar valores e salvar/restaurar o estado dos registradores conforme necessário.

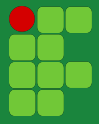
As funções permitem modularizar o código, facilitando a reutilização e a manutenção. Aqui estão os passos básicos e as convenções para implementar chamadas de funções em RISC-V:

1. **Definindo uma função:** Uma função é simplesmente um bloco de código com um ponto de entrada identificado por um rótulo. A função pode receber argumentos, executar operações e opcionalmente retornar um valor.

2. **Passando argumentos:** RISC-V utiliza registradores a0 a a7 para passar os primeiros oito argumentos inteiros ou ponteiros para funções.

Assembly RISC-V

Funções



Continuação:

3. **Chamando a função** A instrução jal (Jump And Link) é usada para chamar funções. Ela salva o endereço de retorno no registrador ra (registrador de link) e salta para o endereço da função.

4. **Salvando registradores:** Se uma função chama outras funções (chamada de função aninhada) ou altera os valores de registradores que precisam ser preservados (como registradores temporários t0-t6 ou salvos s0-s11), esses registradores devem ser salvos na pilha no início da função e restaurados antes de retornar.

5. **Retornando de uma função:** O valor de retorno é normalmente colocado no registrador **a0**. A função retorna ao endereço salvo em **ra** usando a instrução **ret** ou **jalr ra**.

Assembly RISC-V

Funções

Exemplo de Implementação de uma Função Soma

```
.data

.text

.global main

main:
    # Prepara os argumentos para a funcao soma
    li a0, 5        # Primeiro argumento: 5
    li a1, 10       # Segundo argumento: 10
    # Chama a funcao soma
    jal soma

    li a7, 93       # Codigo da syscall para terminar o programa (exit)
    ecall
# Funcao soma
# Assume que os argumentos estao em a0 e a1
# Retorna o resultado em a0
soma:
    add a0, a0, a1  # Soma os argumentos a0 e a1, resultado vai para a0
```

Assembly RISC-V

Funções: Chamadas Aninhadas de Funções

Para fazer uma chamada aninhada de funções em assembly RISC-V, é necessário entender como as chamadas de funções são realizadas nessa arquitetura e como você pode manipular a pilha e os registros para acomodar múltiplas chamadas de funções dentro de outras funções.

Para tanto, os seguintes conceitos precisam ser entendidos:

- Função Ativa;
- Gerenciamento de pilha;
- Passagem de argumentos;
- Preservação de estado;
- Chamada de função;
- Retorno de funções.

Assembly RISC-V

Funções: Chamadas Aninhadas de Funções

- **Função Ativa:**

  Uma função torna-se ativa quando é chamada e continua sendo a função ativa até que complete sua execução e retorne ao chamador.

  Durante seu período ativo, a função possui controle total sobre os registradores especificados (de acordo com a convenção de chamada) e a parte da pilha que alocou para seu uso.

  Quando uma função (**função A**) chama outra função (**função B**), **função A** se torna a função chamadora e **função B** a função chamada ou função ativa.

  Isso cria uma pilha de chamadas, onde **função B** será a função ativa até que conclua sua execução e retorne a **função A**.

Assembly RISC-V

Funções: Chamadas Aninhadas de Funções

- **Função Ativa:**

  A função ativa geralmente aloca espaço na pilha para armazenar variáveis locais, salvar os valores dos registradores que são modificados durante sua execução, e para possíveis argumentos que excedam os registradores disponíveis para passagem de parâmetros.

  Este espaço na pilha é dedicado exclusivamente à função ativa e é desalocado quando a função completa sua execução e o controle retorna à função chamadora.

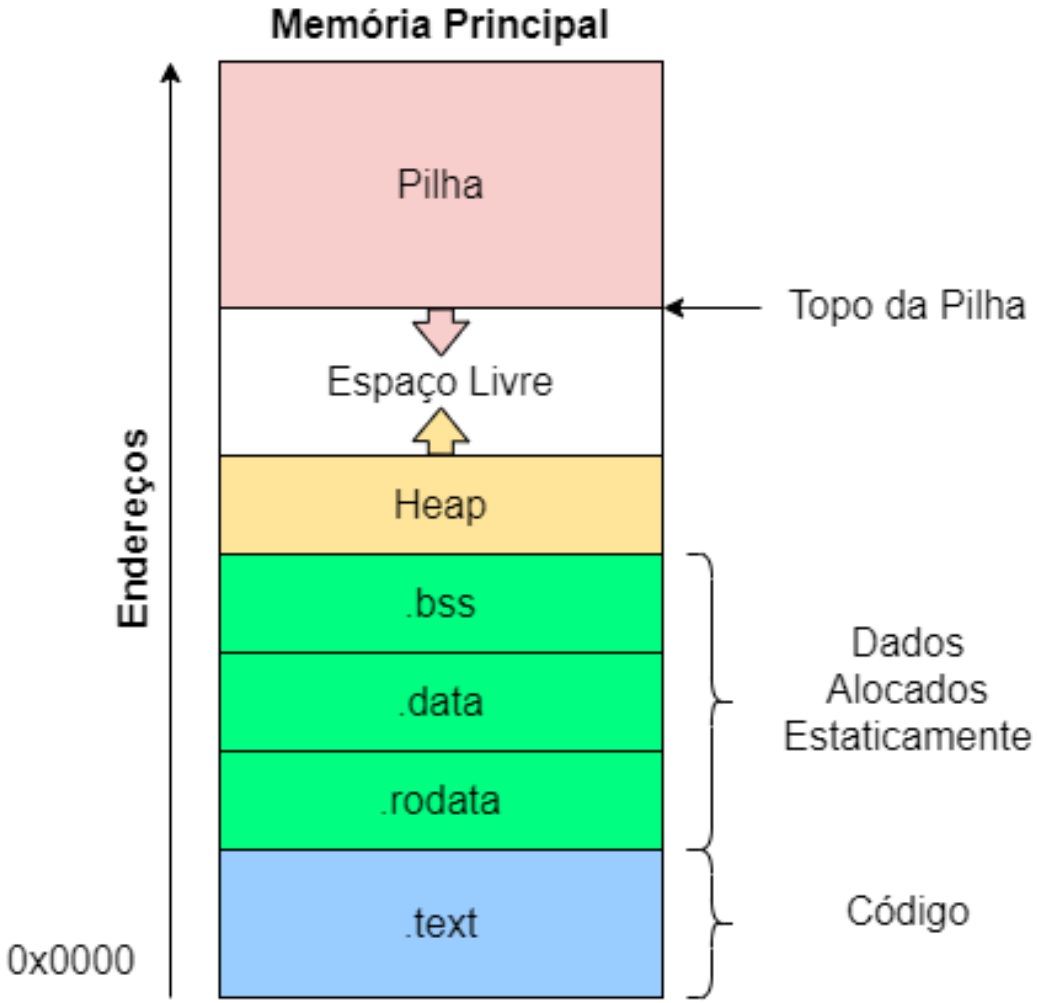
Assembly RISC-V

Funções: Chamadas Aninhadas de Funções

- **Gerenciamento de Pilha:**
  - A pilha é usada para armazenar o estado dos registradores antes das chamadas de funções e para passar argumentos que não cabem nos registradores.
  - No RISC-V, o registrador sp (stack pointer) aponta para o topo da pilha.
  - É uma convenção comum mover o **sp** para reservar espaço na pilha antes de uma chamada de função e restaurá-lo depois:
    - **Reservar espaço: addi sp, sp, -X** onde X é o número de bytes para reservar.
    - **Restaurar o espaço: addi sp, sp, X** após a conclusão da função, para desfazer a reserva.

Assembly RISC-V

Funções: Chamadas Aninhadas de Funções



Assembly RISC-V

Funções: Chamadas Aninhadas de Funções

Exemplo de Gerenciamento de Pilha

```
# Funcao A chama Funcao B
funcaoA:
  addi sp, sp, -8       # Move o sp para baixo para criar espaco na
 pilha
  sd ra, 4(sp)          # Salva o registrador de retorno ra em sp+4
  sd a0, 0(sp)          # Salva o argumento a0 em sp+0

  # chamada de Funcao B
  call funcaoB

  ld a0, 0(sp)          # Restaura o a0
  ld ra, 4(sp)          # Restaura o ra
  addi sp, sp, 8        # Move o sp de volta para cima, liberando o
 espaco
  ret                   # Retorna para o chamador
```

Assembly RISC-V

Funções: Chamadas Aninhadas de Funções

- **Passagem de Argumentos:**

  Em RISC-V, a passagem de argumentos segue uma convenção de chamada definida que especifica como os argumentos são passados das funções chamadoras para as chamadas. As principais características dessa convenção incluem:

  - **Registradores de Argumentos:**
    - RISC-V usa os registradores **a0** até **a7** (x10 até x17) para passar os primeiros oito argumentos inteiros ou ponteiros às funções.
    - Os argumentos são colocados nesses registradores pela função chamadora antes de fazer a chamada.
    - **a0** também é usado para retornar valores da função. Se uma função retorna um valor, esse valor é colocado em **a0**.

Assembly RISC-V

Funções: Chamadas Aninhadas de Funções

- **Passagem de Argumentos:**

  Continuação:

  - **Passagem de Argumentos Adicionais:**
    - Se uma função requer mais do que oito argumentos, os argumentos adicionais são passados usando a pilha.
    - Os argumentos são empilhados na ordem em que são passados, começando do nono argumento em diante.
  - **Tipos de Dados de Ponto Flutuante:**
    - Para argumentos de ponto flutuante, RISC-V usa um conjunto separado de registradores de ponto flutuante **fa0** até **fa7** para os primeiros oito argumentos de ponto flutuante.
    - Argumentos de ponto flutuante adicionais também são passados via pilha.

Assembly RISC-V

Funções: Chamadas Aninhadas de Funções

- **Passagem de Argumentos:**

  Continuação:

  - **Passagem de Parâmetros por Valor:** uma cópia do dado (valor do argumento) é passada para a função chamada. Isso significa que a função recebe uma cópia do argumento original, e quaisquer modificações feitas nesse argumento dentro da função não afetam o valor original fora da função.

  - **Passagem de Parâmetros por Referência:** em vez de passar uma cópia do valor, passa-se um endereço de memória (referência) onde os dados estão armazenados. Qualquer modificação feita aos dados através desse endereço refletirá no valor original, pois a função chamada opera diretamente na localização da memória do argumento.

Assembly RISC-V

Funções: Chamadas Aninhadas de Funções

Exemplo de Passagem de Parâmetros por Valor

```
.data

.text
.global main

main:
    ...
    li a0 5
    jal pow2
    ...

pow2:
    mul a0 a0 a0
    ret
```

Assembly RISC-V

Funções: Chamadas Aninhadas de Funções

Exemplo de Passagem de Parâmetros por Valor

```
.data
    x: .word 5

.text
.global main

main:
    ...
    la t1 x
    lw a0 (t1)
    jal pow2
    ...

pow2:
    mul a0 a0 a0
    ret
```

Assembly RISC-V

Funções: Chamadas Aninhadas de Funções

Exemplo de Passagem de Parâmetros por Referência

```
.data
  x: .word 5
  y: .word 1
.text
.global main
main:
  ...
  la t1 x
  la t2 y
  mv a0 t1
  mv a1 t2
  jal troca
  ...

troca:
  lw t0 (a0)
  lw t1 (a1)
  sw t0 (a1)
  sw t1 (a0)
  ret
```

Assembly RISC-V

Funções: Chamadas Aninhadas de Funções

- **Preservação de Estado:**

  É uma prática essencial para garantir que o programa funcione corretamente ao longo de chamadas de função.

  Preservar o estado envolve salvar e posteriormente restaurar o conteúdo de registradores e outras variáveis críticas que são modificadas durante a execução de uma função.

  Isso é vital para manter a integridade do programa, especialmente em ambientes com muitas chamadas de funções e operações recursivas.

  Quando uma função é chamada, ela pode modificar os valores de registradores e outras áreas de memória que foram usadas por outra função.

Assembly RISC-V

Funções: Chamadas Aninhadas de Funções

- **Preservação de Estado:**

  Se esses valores originais não forem restaurados após a execução da função, isso pode levar a resultados incorretos, comportamentos inesperados e bugs difíceis de rastrear.

  Portanto, preservar o estado permite que cada função opere de maneira isolada, sem interferir nos resultados de outras partes do programa.

Assembly RISC-V

Funções: Chamadas Aninhadas de Funções

- **Preservação de Estado:**
  - **Registradores de Salvamento de Chamador (Caller-Saved):**
    - Esses registradores, quando usados dentro de uma função que os altera, devem ser salvos (tipicamente na pilha) pelo chamador se ele deseja preservar seus valores originais após a chamada da função.
    - Em RISC-V, os registradores **a0**-**a7** (usados para argumentos e retornos de funções) e **t0**-**t6** (registradores temporários) são considerados caller-saved.
  - **Registradores de Salvamento de Chamado (Callee-Saved):**
    - São registradores que, se modificados por uma função, devem ser salvos e restaurados por essa função antes de retornar ao chamador.
    - Em RISC-V, os registradores **s0**-**s11** são callee-saved. Isso significa que qualquer função que utilize esses registradores para operações durante sua execução deve salvá-los no início da função e restaurá-los antes de retornar.

Assembly RISC-V

Funções: Chamadas Aninhadas de Funções

Exemplo de Preservação de Estados

```
.data

.text
.global main

main:
   li a0 5 #a
   li a1 3 #b
   li a2 4 #c
   li a3 2 #d

   addi sp sp -4 #Reserva espaco na pilha para a3
   sw a3 0(sp) #Salva o valor de a3 no topo da pilha

   jal calcular

   lw a3 0(sp) #Restaura o valor original de a3 do topo da pilha
   addi sp sp 4 #Libera espaco na pilha

   li a7 93 #Syscall para sair do programa
```

Assembly RISC-V

Funções: Chamadas Aninhadas de Funções

Exemplo de Preservação de Estados

```
calcular:
  addi sp sp -16 #Aloca 4 espacos para registradores
  sw ra 12(sp)#Salva ra no da pilha
  sw a0 8(sp) #Salva a0 na pilha
  sw a1 4(sp) #Salva a1 na pilha
  sw a2 0(sp) #Salva a2 na pilha

  jal somar #Chamada da funcao somar

  mv a1 a2 #Passagem de parametro para multiplicar
  jal multiplicar #Chamada da funcao multiplicar

  mv a1 a3 #Passagem de parametro para substituir
  jal subtrair #Chamada da funcao subtrair

  lw a2 0(sp) #Restaura a2 da pilha
  lw a1 4(sp) #Restaura a1 da pilha
  lw a0 8(sp) #Restaura a0 da pilha
  lw ra 12(sp) #Restaura endereco de retorno
  addi sp sp 16 #Libera 4 espacos na pilha
```

Assembly RISC-V

Funções: Chamadas Aninhadas de Funções

Exemplo de Preservação de Estados

```
somar:
  addi sp sp -4 #Aloca 1 espaco na pilha
  sw ra 0(sp) #Salva o ponto de retorno na pilha

  add a0 a0 a1 #Soma a0 e a1 e salva em a0

  lw ra 0(sp) #Restaura endereco de retorno
  addi sp sp 4 #Libera um espaco na pilha
  ret #Retorna para o ponto de chamada de calcular

multiplicar:
  addi sp sp -4 #Aloca 1 espaco na pilha
  sw ra 0(sp) #Salva o ponto de retorno na pilha

  mul a0 a0 a1 #Multiplica a0 e a1 salva em a0

  lw ra 0(sp) #Restaura endereco de retorno
  addi sp sp 4 #Libera 1 espaco na memoria
  ret #Retorna para o ponto de chamada de calcular
```

Assembly RISC-V

Funções: Chamadas Aninhadas de Funções

Exemplo de Preservação de Estados

```
subtrair:
  addi sp sp -4 #Aloca 1 espaco na pilha
  sw ra 0(sp) #Salva o ponto de retorno na pilha

  sub a0 a0 a1 #Subtrai a0 e a1 e salva em a0

  lw ra 0(sp) #Restaura endereco de retorno
  addi sp sp 4 #Libera 1 espaco na memoria
  ret #Retorna para o ponto de chamada de calcular
```

Assembly RISC-V

Funções: Chamadas Aninhadas de Funções

- **Chamada de Função:**

  A instrução para chamar uma função é **jal** (*Jump And Link*).

  jal salva o endereço de retorno no registrador **ra** (registrador de retorno) e depois salta para o endereço da função chamada.

  O endereço de retorno é o ponto logo após a chamada da função, onde o controle do programa deve retornar após a conclusão da função.

Assembly RISC-V

Funções: Chamadas Aninhadas de Funções

- **Retorno de Funções:**

  Em RISC-V, quando uma função é chamada usando a instrução **jal**, o endereço logo após a chamada é automaticamente armazenado no registrador **ra**.

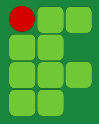
  Isso prepara o programa para saber exatamente onde retornar após a conclusão da função.

  A instrução **ret** é um pseudônimo para a instrução **jalr zero 0(ra)**, que faz com que o programa salte para o endereço armazenado em **ra**.

  Essa instrução é usada no final da função para devolver o controle ao chamador.

Assembly RISC-V

Funções: Chamadas de Funções Recursivas



Uma função recursiva é uma função que chama a si mesma durante sua execução.

Esse tipo de função é comumente usada em programação para resolver problemas que podem ser divididos em subproblemas mais simples de natureza similar.

A recursividade é uma ferramenta poderosa, especialmente útil para trabalhar com estruturas de dados que possuem uma natureza recursiva, como árvores e grafos, ou para implementar algoritmos como busca e ordenação.

Assembly RISC-V

Funções: Chamadas de Funções Recursivas

Exemplo de Função Recursiva

```
    .data
    .text
    .global main

   main:
        li a0, 5            # Carrega o numero 5 em a0, para calcular 5!
        jal ra, factorial   # Chama a funcao factorial e salva o endereco
    de retorno em ra

        li a7 93
        ecall
```

Assembly RISC-V

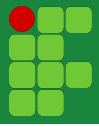
Funções: Chamadas de Funções Recursivas

Exemplo de Função Recursiva

```
factorial:
    addi sp, sp, -8     # Decrementa o ponteiro de pilha para salvar
 espaco para ra e a0
    sw ra, 4(sp)        # Salva o conteudo do registrador ra na pilha
    sw a0, 0(sp)        # Salva o conteudo do registrador a0 na pilha

    li a1, 1            # Preparar para testar se n == 1
    beq a0, a1, base_case  # Se sim, tratar o caso base (fatorial de
 1)
    li a1, 0
    beq a0, a1, base_case  # Se n == 0, tratar tambem como caso base

    addi a0, a0, -1     # Preparar para a chamada recursiva: calcular
 factorial(n-1)
    jal ra, factorial   # Chamada recursiva
    lw a1, 0(sp)        # Recuperar o valor original de n
    mul a0, a0, a1      # n * factorial(n-1)

    lw ra, 4(sp)        # Restaurar o valor original de ra
    addi sp, sp, 8      # Restaurar o ponteiro da pilha
```

Assembly RISC-V

Funções: Chamadas de Funções Recursivas



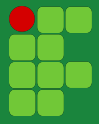
Exemplo de Função Recursiva

```
base_case:
    li a0, 1            # O fatorial de 1 ou 0 e 1
    lw ra, 4(sp)        # Restaurar o valor original de ra
    addi sp, sp, 8      # Restaurar o ponteiro da pilha
    ret                 # Retornar ao endereco salvo em ra
```

Exercícios 2

Assembly RISC-V

Exercícios 2



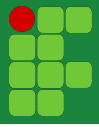
1. Escreva um programa em Assembly RISC-V que conta progressivamente de 1 até um número N fornecido. O programa deve imprimir cada número da contagem. Para este exercício, assuma que existe uma maneira de imprimir um número no seu ambiente.

2. Crie um programa em Assembly RISC-V que calcula a soma dos números de 1 até N, onde N é um valor fornecido. O programa deve armazenar o resultado da soma e, ao final, imprimir esse resultado.

3. Desenvolva um programa em Assembly RISC-V que verifica se um número N fornecido é primo. O programa deve imprimir uma mensagem indicativa se o número é primo ou não. Lembre-se, um número primo é um número maior que 1 que não tem divisores positivos além de 1 e ele mesmo.

Assembly RISC-V

Exercícios 2



4. Implemente um programa em Assembly RISC-V que calcula o fatorial de um número N fornecido (ou seja, N!). O fatorial de um número é o produto de todos os inteiros positivos menores ou iguais a ele. Por exemplo, o fatorial de 5 (5!) é 5 x 4 x 3 x 2 x 1 = 120.

5. Escreva um programa em Assembly RISC-V que calcula o Máximo Divisor Comum (MDC) de dois números, A e B, fornecidos. O MDC de dois números é o maior número que divide ambos sem deixar resto. Você pode usar o algoritmo de Euclides, que repete a operação de subtração ou divisão modular até encontrar o MDC.

Vetores

Assembly RISC-V

Vetores

Trabalhar com vetores no assembly RISC-V pode é bastante desafiador, já que você estará lidando diretamente com a manipulação de memória e registradores a um nível muito baixo.

Para trabalhar com vetores em Assembly RISC-V, é preciso:

- Alocar espaço na memória para o vetor na seção de dados, usando **.space** para definir o tamanho necessário.
- Carregar o endereço base do vetor em um registrador.
- Use as instruções **li** para definir, **lw** para carregar, **sw** para armazenar valores.
- Acessar elementos específicos usando seu índice.
- Calcular o endereço absoluto do elemento (índice x tamanho do tipo de dado).

Assembly RISC-V

Vetores

Exemplo de Implementação de um Vetor

```
.data
   numeros: .space 40 #vetor de tamanho 10
   msg_1: .asciz "Informe o numero:"
.text
.global main
main:
   la t0 numeros #carregar o endereco do vetor
   li t1 0 #inicializa o indice do vetor

leitura:
   li a7 4
   la a0 msg_1
   ecall
   li a7 5
   ecall
   sw a0 0(t0) #armazena numero na posicao
   addi t0 t0 4 #incrementa endereco
   addi t1 t1 1 #incrementa indice
   li t2 10
   blt t1 t2 leitura #continua se t1 < 10
```

Assembly RISC-V

Vetores

Exemplo de Implementação de um Vetor

```
   ...
   #saiu do loop
   la t0 numeros #carregar o
   li t1 0

impressao:
   lw t3 0(t0) #carrega numero da posicao para registrador
   li a7 1
   mv a0 t3
   ecall
   addi t0 t0 4 #incrementa endereco
   addi t1 t1 1 #incrementa indice
   li t2 10
   blt t1 t2 impressao #continua se t1 < 10
```

Exercícios 3

Assembly RISC-V

Exercícios 3

1. Dados dois vetores de 10 inteiros cada, crie um terceiro vetor que seja a soma elemento a elemento dos dois primeiros.
2. Encontre e armazene o valor do maior elemento de um vetor de 15 inteiros.
3. Inverta a ordem dos elementos de um vetor de 10 inteiros.
4. Identifique e armazene a posição do menor elemento em um vetor de 20 inteiros.
5. Preencha um vetor de 10 inteiros e, em seguida, ordene-o de forma crescente.

Assembly RISC-V

Exercícios 3

6. Preencha a diagonal principal de uma matriz 7x7 com um valor específico, deixando os outros elementos com valor 0.

7. Conte o número de elementos negativos em uma matriz 10x10.

8. Calcule a soma dos elementos das diagonais principal e secundária de uma matriz quadrada 5x5.

9. Preencha uma matriz 10 x 10 e, em seguida, zere todos os elementos abaixo da diagonal principal.

10. Encontre e retorne as coordenadas de um elemento específico dentro de uma matriz 8x8.

Programação Assembly RISC-V

Prof. Fabrício B. Gonçalves

Instituto Federal Fluminense
Campus Bom Jesus do Itabapoana