Assembly RISC-V

Como Programar Nisso?

Introdução

“Quem você acha que é”

Thales Menezes

- Bacharelando em Ciência da Computação (17.1)
- Estudante ávido de Python 3 e Assembly’s (16.2/17.1 - ...)
- Tutor de Introdução à Ciência da Computação (18.2 - ...)
- Monitor de Introdução à Sistemas Computacionais (19.1 - ...)

“Pra que Assembly?”

Mitos

- Assembly é difícil de ler/escrever/aprender/depurar

- Com compiladores tão bons, aprender Assembly é desnecessário

- A era do assembly já passou, hoje em dia recursos* não faltam

- Assembly não é portátil

* Memória para processamento e espaço em disco.

“Pra que Assembly?”

- Assembly é rápido
- Assembly é compacto
- Assembly permite que tirar melhor proveito do hardware*
- Assembly ajuda a entender melhor os programas e abstrações

“Porque RISC-V?”

“O RISC-V é uma ISA recente, iniciada do zero, minimalista e aberta, informada por erros de ISA anteriores”

- ISA recente e aberta (*Free 2 Use*)

- Atual, academicamente falando (MIPS 1985 | RISC-V 2010/11)

- Legibilidade maior

- Arquitetura Modular (RV32I)
  Vs Incremental (Compatibilidade Retroativa)

- Veloz a baixo custo

RISC vs CISC

RISC:
Reduced Instruction Set Computer

- Processador com poucas instruções
- Apenas instruções simples
- “Rápidas” e “Compactas”
- Ex.: ARM, MIPS, RISC-V

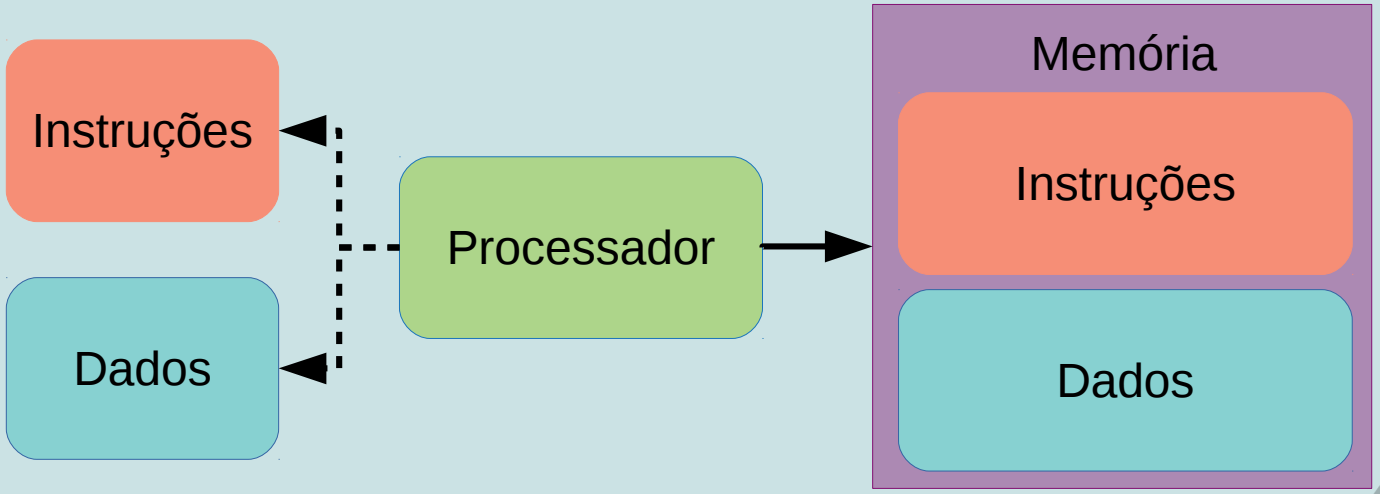
CISC:
Complex Instruction Set Computer

- Processador com muitas instruções
- Contém instruções simples e complexas
- “Lentas” e “Grandes”
- Ex.: x86, x64

Harvard vs Programa Armazenado

Harvard
(Como a faculdade)

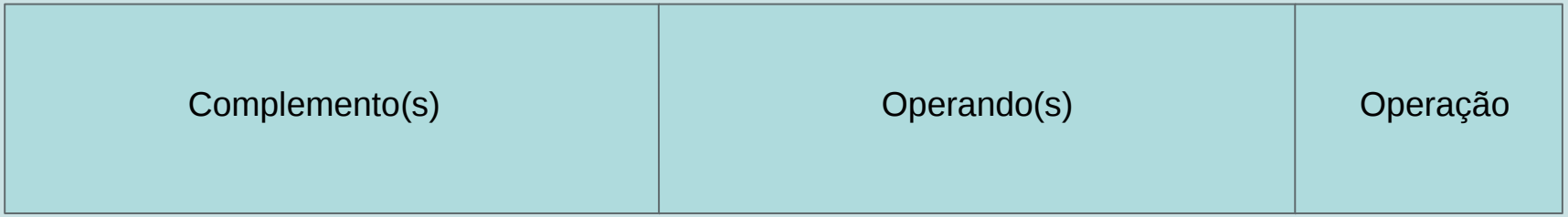
Programa Armazenado
(Equipe Prof. Von Neumann)



Comandos

Um programa pode ser visto como uma série de comandos armazenados na memória, sendo indistinguíveis de um dado.

Como representar um comando?



Glossário

- Labels:
  - Nomes para endereços da memória (Ponteiros)

- Diretivas:
  - Palavras reservadas usadas na tradução e compilação do código; não existem para o processador.

Módulo 1:
Sintaxe Primitiva

Principais Blocos

- Defines
- Dados
- Instruções

"Uma linguagem não é considerada grandiosa somente pela beleza da sua implementação, mas, principalmente, pela riqueza da sua documentação"

Rodrigo Bonifácio

Dentro do ambiente RARS,
pressione F1 para abrir a ajuda

Armazenar Dados

- Registradores:
  - Acesso rápido; porém quantidade limitada.
- Memória de Dados (RAM):
  - Maior, porém mais lenta.
- Imbutido na Instrução (Valor Imediato):
  - Acesso rápido, melhor otimização possível

Banco de Registradores

- zero: valor constante 0 (*ground*)
- t0~t6: registradores temporários
- s0~s11: registradores salvos
- a0~a7: argumentos/resultados de funções (a0 e a1)
- ra: endereço de retorno (*return address*)
- sp: ponteiro da pilha (*stack pointer*)

Os principais

"Se todo mundo respeitar a convenção o projeto sai sem complicação"

```
addi t0, t0, 4
# t0 = t0 + 4
```

Dados

Declarar variáveis e outras informações

- **.data <address>**

  Área “reservada” para dados

  Simboliza que os valores à partir dele estarão inseridos no primeiro endereço disponível na área de dados (default) ou <address> (argumento opcional).

Dados

Declarar variáveis e outras informações

```
int num;
int t0 = 10;
```


Dados

Declarar variáveis e outras informações

```
.data
texto:  .string      “Hello World”
hum:    .word        0xFFFFFFFF  # 32 bits
meio:   .halfword    0xFFFF      # 16 bits = ½ .word
char:   .byte        0xFF        #   8 bits = 1 byte
# Podemos colocar mais de um valor do mesmo tipo, separando
# com vírgulas ou espaços
numbers:   .word  4, 8 12        # numbers[] = { 4, 8, 12 }
# Podemos deixar um “espaço vazio”
void:         .space 20  # 20 bytes reservados
fpoint:       .float   4.0
big_fpoint: .double 8.25
```

Dados

Declarar variáveis e outras informações

- **Diretivas:**

  .string e .asciz → sequência de characteres + null

  .word → número inteiro de 32 bits

  .halfword → número inteiro de 16 bits (½ .word)

  .byte → número inteiro de 8 bits (1 byte)

  .float → número de ponto flutuante de precisão simples

  .double → número de ponto flutuante de precisão dupla

  .space <bytes> → reserva um número de bytes

Instruções

Compõem a lógica do programa

- **.text <address>**

  Área “reservada” para instruções do processador

  Simboliza que os valores à partir dele estarão inseridos no primeiro endereço disponível na área de instruções (default) ou <address> (argumento opcional).

Instruções

Compõem a lógica do programa

```
num = 4
t0 = num
t1 = 0 + 10
t0 += t1
a0 = 0 + t0
print(a0)
```


Instruções

Compõem a lógica do programa

```
int num = 4;

int main(){

        int t0 = &num;
        t0 = * (int *) t0;
        int t1 = 0 + 10;
        t0 += t1;
        int a0 = 0 + t0;
        printf(“%d”, a0);

        return 0;
}
```

Instruções

Compõem a lógica do programa

```
.data
num:  .word 4

.text
      la t0, num              # t0 = &num
      lw t0, (t0)             # t0 = num = 4
      addi  t1, zero, 10      # t1 = 0 + 10 = 10
      # li t1, 10
      add  t0, t0, t1         # t0 = 4 + t1
      add  a0, zero, t0       # a0 = 0 + t0 = t0
      # mv a0, t0
      li  a7, 1               # a7 = 1 NOTE: pseudoinstrução
      ecall                   # interrupção de sistema
```


Instruções

Compõem a lógica do programa

- **Soma e Consulta à Memória:**

add t0,t1,t2 → t0 = t1 + t2 (Add operation)

addi t0,t1, Imm → t0 = t1 + Imm (Add with Immediate)

la t0, label → t0 = &label; (Load Address)

Carrega o endereço apontado por label para o registrador

lw t0, Imm (t1) → t0 = mem[t1 + Imm] (Load Word)

Carrega, em t0, o valor contido no endereço de t1 + Immediate

Immediate (Imm) = Número inteiro de 12 bits (complemento 2)

Instruções

Compõem a lógica do programa

- **Pseudoinstruções :**

  Comandos que não são compreendidos pelo processador, são traduzidas para instruções válidas durante a compilação.

  li t0, Imm → t0 = Imm (Load Immediate) → addi t0, zero, Imm

  mv t0, t1 → t0 = t1 (Move) → add t0, t1, zero

Instruções

Compõem a lógica do programa

- **Interrupções de sistema:**

  Temos à disposição alguns serviços de sistema implementados:

  Para executar esses serviços precisamos:

  - Carregar o número do serviço desejado no registrador a7
  - Carregar os argumentos necessários (a0,a1,...)
  - Instrução ecall → chamada/interrupção do sistema
  - Recuperar resultados, caso existam

Instruções

Compõem a lógica do programa

- **Interrupções de sistema:**
  - Ler e imprimir valores através do RUN I/O (“Terminal”)
  - Encerrar o programa
  - Pausar a execução do programa
  - Tocar notas musicais
  - Manipular arquivos
  - Gerar números aleatórios

```
.text
        li a0, 4    # a0 = 4
        li a7, 1    # código imprime int
        ecall       # system call
```


FIM Módulo 1: Exercícios

Exercícios de Fixação

1) Faça um programa que calcule a média entre dois números

Input:
2 inteiros em linhas separadas
Output:
1 único número inteiro

Exemplo:

INPUT	OUTPUT
2 3	2

Exercícios de Fixação

2) Faça um programa que recebe um texto e o imprime

Input:
1 única linha de texto
Output:
1 única linha de texto

Exemplo:

INPUT	OUTPUT
oi	oi
RISC-V	RISC-V

Módulo 2:
Saltos de Execução

Condicionais

- **Qual o comportamento/essência de uma estrutura condicional?**

  Quando o programa se depara com uma estrutura condicional, avalia a condição e executa, ou não, as instruções contidas nela.

  Em outras palavras, caso a condição seja falsa o programa “pula” o conjunto de instruções, portanto, existem 2 “ramos” ou possibilidades de execução: executar ou “pular”.

  Branch → Ramo

Instruções

Compõem a lógica do programa

- **Condicionais e saltos:**
  - Branch if EQual → beq t0, t1, label

    → Se t0 == t1, “pula” para endereço <label>
  - Jump → j label *

    → “Pula” para endereço <label>, incondicionalmente

  * Pseudoinstrução

Condicionais

```
int main(){

        int a = 0;
        scanf(“%d”, &a);
        if ( a == 0 )
                printf(“Zero!!!!”);
        else
                printf(“Não Zero!!!”);

        return 0;
}
```


Condicionais

E se trocássemos:

j ~~fora~~ if

```
.data
eh_zero: .asciz "É zero!!!"
not_zero:.asciz "Não é zero!!!"
.text
        li a7, 5                    # código serviço: Ler inteiro
        ecall                       # chamada de sistema
        li a7, 4                    # código serviço: Imprimir string
if:     bne a0, zero, else          # if ( a0 != 0 ) “pula” para else, senão...
              la a0, eh_zero        #     { a0 = &“É zero!!!” }
        j fora                      # fim do if → salto incondicional
else:la a0, not_zero                # else{ a0 = &“Não é zero!!!” }
fora:ecall                          # chamada de sistema
        li a7, 10                   # código serviço: Encerra programa
        ecall                       # chamada de sistema
```

While

- **Qual o comportamento/essência de uma estrutura de repetição?**

  Podemos imaginar como uma estrutura condicional e um comando que “pula para trás”, voltando para a condição.

```
if ( a > 0 ) {
        printf(“%d...”, a--);
        repete if;
}
```

While

```
int main(){

	int a = 0;
	scanf(“%d”, &a);

	while ( a > 0 )
		printf(“%d...”, a--);

	printf(“Kabum!!!\n”);
	return 0;
}
```

While

if + “repete” → while

```
[...]
        li a7, 4                    # código serviço: Imprimir string
        mv s0, a0                   # s0 = a0
while:  bltz s0, fora               # if ( s0 < 0 ) “pula” para fora, senão...
                mv a0, s0           #     a0 = s0
                li a7, 1            #     código serviço: Imprimir int
                ecall               #     chamada de sistema
                addi s0, s0, -1     #     s0--
                la a0, dotdotdot    #     a0 = &“...”
                li a7, 4            #     código serviço: Imprimir string
                ecall               #     chamada de sistema
        j while                     # repete → volta ao inicio do loop
fora:   [...]
```

Instruções

Compõem a lógica do programa

- **Saltos:**
  - Jump And Link → jal t0, label
    → Salva o endereço de retorno* em t0 e “pula” para <label>
  - Jump And Link Register → jalr t0, Imm(t1)
    → Semelhante ao anterior, mas “pula” para <t1+Imm>
  - Jump → j label → jal zero, label

“pula”, mas sabe voltar

Funções

Porque ponteiros?

```
void zerar ( int * a ){
        if ( *a == 0 )
                *a = 0;
}

int main(){

        int a = 0;
        zerar(&a);
        return 0;
}
```

Funções

```
.text
      li a0, 0          # 1ª a0 = 0
      jal ra, zerar     # 2ª salva o endereço da próxima instrução e
                        #     "pula" para o endereço zerar
      li a7, 10         # 5ª código encerra programa
      ecall             # 6ª system call

## Função zerar
zerar:  beq a0, zero, back  # 3ª if (a0 == 0) "pula" pra back
      li a0, 0          # if (a0 != 0) {a0 = 0;}
back:
      ret               # 4ª jalr zero, 0 (ra) → retorna da função
```

FIM Módulo 2: Exercícios

Exercícios de Fixação

1) Crie uma função que calcule a média entre dois números

Input:
2 inteiros
Output:
1 único número inteiro

Exemplo:

INPUT	OUTPUT
2 3	2

Exercícios de Fixação

2) Crie uma função que calcule o fatorial de um número

Input:
1 inteiro
Output:
1 único número inteiro

Exemplo:

INPUT	OUTPUT
2	2
3	

Módulo 3: Defines

Defines

Aqueles que não pertence aos outros grupos

```
.include "cool.asm"          # Caminho até o arquivo

.macro SUM(%a, %b)           # SUM recebe 2 registradores
      add %a, %a, %b         # retorna o resultado da soma
.end_macro                   # no primeiro

.eqv GRAVIDADE 10            # GRAVIDADE = 10

.text
      addi t0, zero, GRAVIDADE # t0 = 0 + GRAVIDADE = 0 + 10
      SUM(t1,t0)               # t1 = t1 + t0
```

Defines

Aqueles que não pertence aos outros grupos

- **.include “cool.asm”**

  Incluir arquivos: em linguagem C, seria `#include "cool.asm"`

  Representam a inclusão de um código contido em outro arquivo.

  Na prática, podemos imaginar como um grande *Copy and Paste* de código: no lugar desse comando, será colocado o conteúdo do arquivo.

Mais configurações:
.global .extern

Defines

Aqueles que não pertence aos outros grupos

- **.macro && .end_macro**

  Definem macros: em linguagem C, seria `#define SUM(a,b) (a + b)`

  Representa um “molde” de código, ou seja, um trecho de código que pode ser modelado de acordo com o uso.

  Na prática, podemos imaginar como *Copy and Paste* de código: no lugar desse comando, será colocado o “molde” com os argumentos.

Macros vs Funções

Macros

Prós	Contras
- Executadas mais rápidas (*inline*) - Idealmente, são simples e objetivas	- Códigos mais extensos

Funções

Prós	Contras
- Interrupções - Permitem recursão - “Reusáveis”	- Mais lentas, se comparadas à macros

Defines

Aqueles que não pertence aos outros grupos

- **.eqv GRAVIDADE 10**

  Equivalência: em linguagem C, seria `#define GRAVIDADE 10`

  Representam um valor literal que será substituído durante a compilação.

  Pode ser visto como uma macro sem argumentos.

Memória Principal

Endereço	Conteúdo	Label
0x00400000	add a0,t0,t1	.text
0x00400004	ecall	
…	…	
0x0FFFFFFC	ecall	END .text
…	…	
0x10010000	“Hey!”	.data
…	…	
0x1003FFFC	“Bye!”	END .data

Referências Bibliográficas

<RARS: RISC-V Assembler Runtime Simulator>*

* Buscar por *Latest Release*, clicar em *Assets* e <rars>.jar

<Guia Prático RISC-V: Atlas de uma arquitetura aberta>

<The Art of Assembly Language>

The End

Thales Menezes

GitHub @thaleslim