



UNIVERSIDADE
FEDERAL DE
SERGIPE



DEPARTAMENTO
DE COMPUTAÇÃO

Registradores e memória

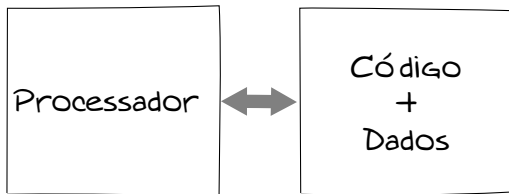
Arquitetura de Computadores

Bruno Prado

Departamento de Computação / UFS

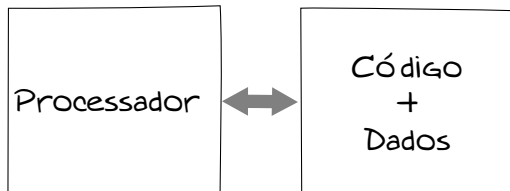
Introdução

- ▶ Já foi visto que os computadores funcionam executando uma programação (sequência de operações e de dados) armazenada em memória



Introdução

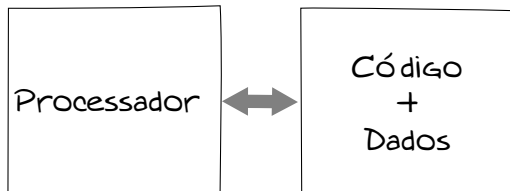
- ▶ Já foi visto que os computadores funcionam executando uma programação (sequência de operações e de dados) armazenada em memória



- ▶ Como os operandos são armazenados?
 - ▶ Imediata
 - ▶ Registrador
 - ▶ Memória

Introdução

- ▶ Já foi visto que os computadores funcionam executando uma programação (sequência de operações e de dados) armazenada em memória



- ▶ Como os operandos são armazenados?
 - ▶ Imediata
 - ▶ Registrador
 - ▶ Memória

Por que utilizar diferentes formas de armazenamento para os operandos?

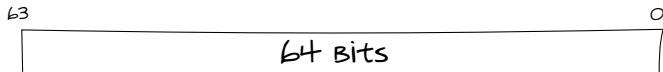
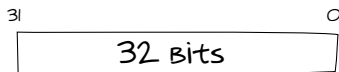
Introdução

► Capacidade × Custo × Latência

Tipo	Capacidade	Custo	Latência
Imediato	1 <-> 3 Bytes	-	-
SRAM	2 KiB <-> 32 MBit	~US\$ 5k / GiB	0,20 <-> 2 ns
DRAM	1 <-> 16 GiB	~US\$ 3 / GiB	~10 ns

Registrador

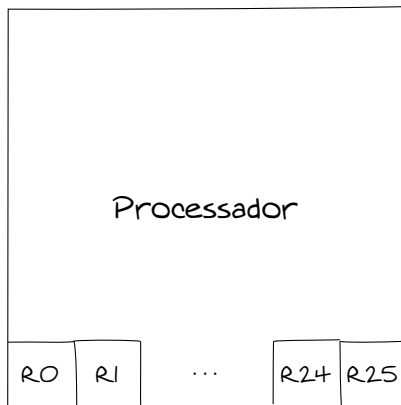
- ▶ O que é um registrador?
 - ▶ É uma memória interna do processador
 - ▶ Geralmente do tamanho da arquitetura



- ▶ Tipos
 - ▶ Propósito geral
 - ▶ Controle e status

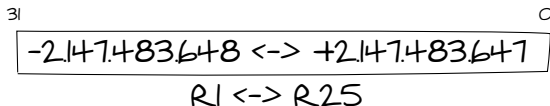
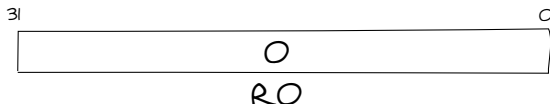
Registrador

- ▶ Propósito geral (R0-R25)
 - ▶ Definido pelo programador
 - ▶ Armazenamento ou endereçamento



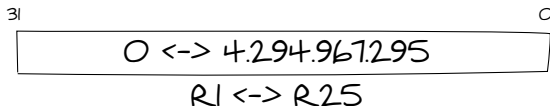
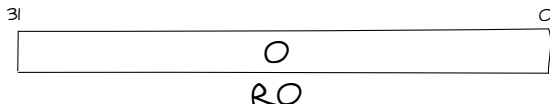
Registrador

- ▶ Propósito geral (R0-R25)
 - ▶ Indexáveis de 0 até 25
 - ▶ R0 possui valor constante 0 e os demais registradores podem armazenar valores de 32 bits com ou sem sinal



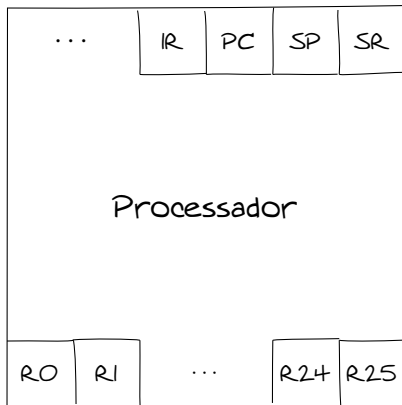
Registrador

- ▶ Propósito geral (R0-R25)
 - ▶ Indexáveis de 0 até 25
 - ▶ R0 possui valor constante 0 e os demais registradores podem armazenar valores de 32 bits com ou sem sinal



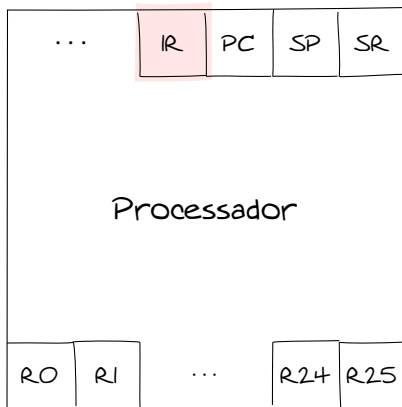
Registrador

- ▶ Controle e status (IR, PC, SP, SR)
 - ▶ Indexáveis de 28 até 31
 - ▶ São utilizados para o funcionamento do processador



Registrador

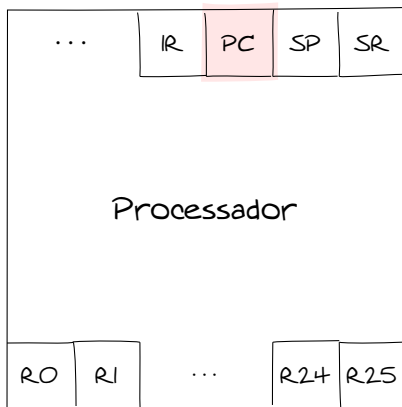
- ▶ Controle e status (IR, PC, SP, SR)
 - ▶ Indexáveis de 28 até 31
 - ▶ São utilizados para o funcionamento do processador



Registrador de instrução (IR): armazena a instrução carregada da memória e em execução

Registrador

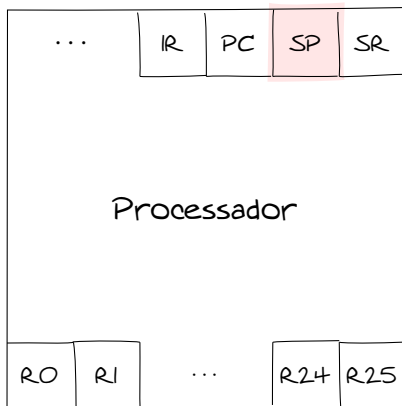
- ▶ Controle e status (IR, PC, SP, SR)
 - ▶ Indexáveis de 28 até 31
 - ▶ São utilizados para o funcionamento do processador



Contador de programa (PC): controla o fluxo de execução da aplicação apontando para as instruções

Registrador

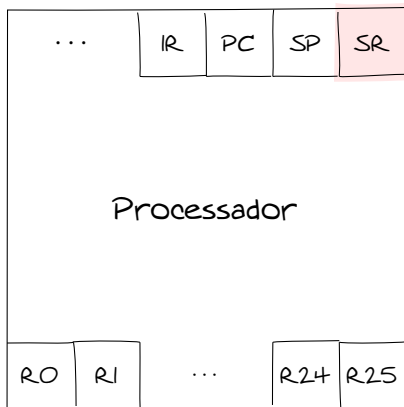
- ▶ Controle e status (IR, PC, SP, SR)
 - ▶ Indexáveis de 28 até 31
 - ▶ São utilizados para o funcionamento do processador



Ponteiro de pilha (SP): referencia o topo da pilha na memória (alocação estática e subrotinas)

Registrador

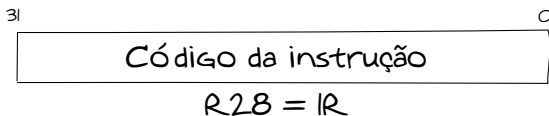
- ▶ Controle e status (IR, PC, SP, SR)
 - ▶ Indexáveis de 28 até 31
 - ▶ São utilizados para o funcionamento do processador



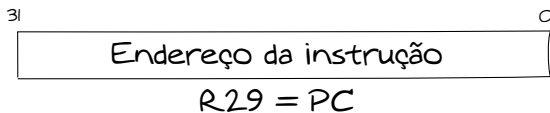
Registrador de status (SR): controle de configurações e status das operações do processador

Registrador

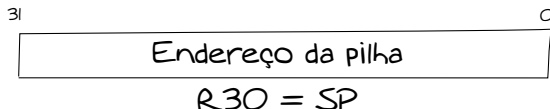
- ▶ Registrador de instrução (IR)
 - ▶ Índice 28



- ▶ Contador de programa (PC)
 - ▶ Índice 29

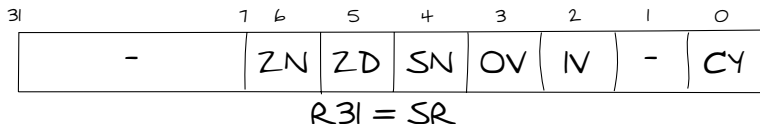


- ▶ Ponteiro de pilha (SP)
 - ▶ Índice 30



Registrador

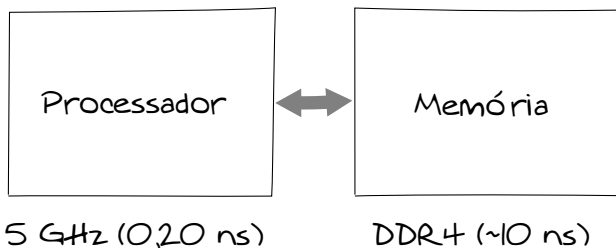
- ▶ Registrador de status (SR)
 - ▶ Índice 31



- ▶ Resultado de operações entre A e B
 - ▶ ZN (zero): igual a 0
 - ▶ ZD (divisão por zero): divisor $B = 0$
 - ▶ SN (sinal): sinal negativo
 - ▶ OV (overflow): extrapolação de capacidade
 - ▶ IV (instrução inválida): código de operação inválido
 - ▶ CY (carry): vai a um aritmético

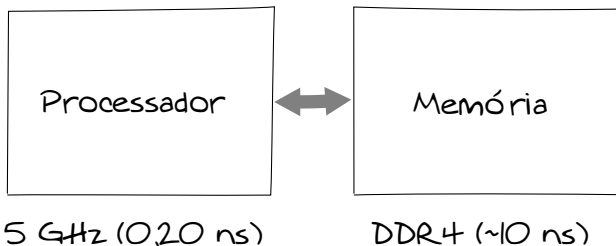
Registrador

- Quais os cenários de operação com os dados?
 1. Acessar e operar diretamente em memória
 2. Ler os dados da memória, realizar as operações em registradores e escrever os resultados na memória



Registrador

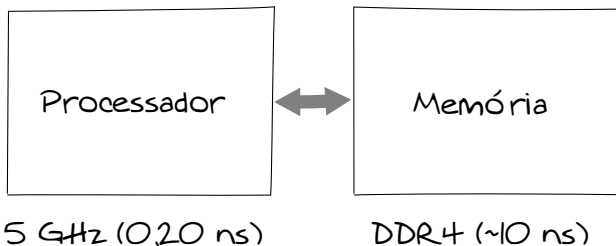
- ▶ Quais os cenários de operação com os dados?
 1. Acessar e operar diretamente em memória
 2. Ler os dados da memória, realizar as operações em registradores e escrever os resultados na memória



- ▶ Cenário com 100 instruções que acessam a memória para ler uma variável em memória
 - ▶ Cenário 1: $100 \times (0,2 \text{ ns} + 10 \text{ ns}) = 1.020 \text{ ns}$
 - ▶ Cenário 2: $10,2 \text{ ns} + 100 \times 0,2 \text{ ns} + 10,2 \text{ ns} = 40,4 \text{ ns}$

Registrador

- ▶ Quais os cenários de operação com os dados?
 1. Acessar e operar diretamente em memória
 2. Ler os dados da memória, realizar as operações em registradores e escrever os resultados na memória



- ▶ Cenário com 100 instruções que acessam a memória para ler uma variável em memória
 - ▶ Cenário 1: $100 \times (0,2 \text{ ns} + 10 \text{ ns}) = 1.020 \text{ ns}$
 - ▶ Cenário 2: $10,2 \text{ ns} + 100 \times 0,2 \text{ ns} + 10,2 \text{ ns} = 40,4 \text{ ns}$

Operações com registradores
são 25 vezes mais rápidas!

- ▶ Arquitetura de carregamento-armazenamento
 - ▶ Operações apenas com registradores (*load-store*)
 - ▶ Sempre que um dado é necessário, é feito seu carregamento prévio da memória
 - ▶ Quando todas as operações já foram concluídas, o dado é armazenado de volta para a memória

Registrador

- ▶ Arquitetura de carregamento-armazenamento
 - ▶ Operações apenas com registradores (*load-store*)
 - ▶ Sempre que um dado é necessário, é feito seu carregamento prévio da memória
 - ▶ Quando todas as operações já foram concluídas, o dado é armazenado de volta para a memória
- ✓ Acesso muito rápido com registradores
- ✓ Regularidade e simplicidade no endereçamento

Registrador

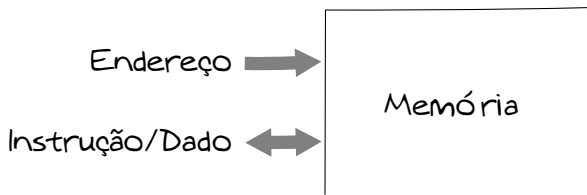
- ▶ Arquitetura de carregamento-armazenamento
 - ▶ Operações apenas com registradores (*load-store*)
 - ▶ Sempre que um dado é necessário, é feito seu carregamento prévio da memória
 - ▶ Quando todas as operações já foram concluídas, o dado é armazenado de volta para a memória
- ✗ Baixa capacidade de armazenamento
- ✗ Número limitado de registradores

- ▶ E quando forem necessários mais registradores do que estão disponíveis?
 - ▶ Acessando múltiplos elementos de um vetor
 - ▶ Alocação dinâmica e estática de dados
 - ▶ Chamadas de funções aninhadas ou recursivas
 - ▶ ...

- ▶ E quando forem necessários mais registradores do que estão disponíveis?
 - ▶ Acessando múltiplos elementos de um vetor
 - ▶ Alocação dinâmica e estática de dados
 - ▶ Chamadas de funções aninhadas ou recursivas
 - ▶ ...
 - ✓ Realizar mais acessos a memória
 - ✓ Utilizar a estrutura de pilha

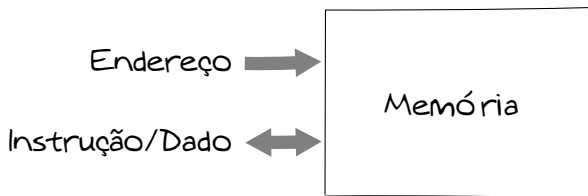
Memória

- ▶ O que é uma memória?
 - ▶ É um dispositivo semicondutor para armazenamento em estado sólido de informações



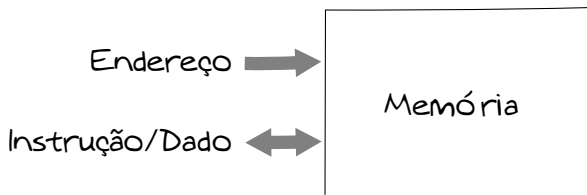
Memória

- ▶ O que é uma memória?
 - ▶ É um dispositivo semicondutor para armazenamento em estado sólido de informações
 - ▶ Permite o endereçamento de posições para realização de operações de escrita e de leitura



Memória

- ▶ O que é uma memória?
 - ▶ É um dispositivo semicondutor para armazenamento em estado sólido de informações
 - ▶ Permite o endereçamento de posições para realização de operações de escrita e de leitura
 - ▶ Todos os dados são codificados em formato binário

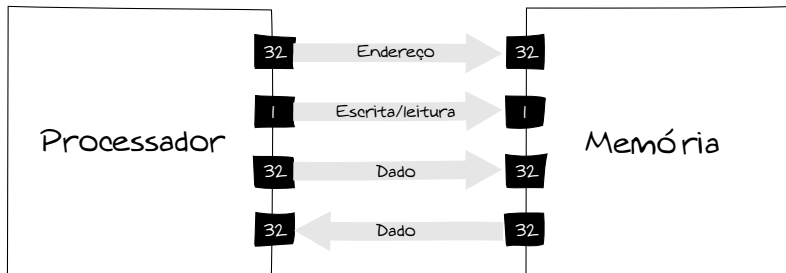


Memória

- ▶ Tecnologias de memória
 - ▶ Volátil
 - ▶ Não retém as informações com a interrupção de fornecimento de eletricidade
 - ▶ Ex: DRAM, SRAM
 - ▶ Não volátil
 - ▶ Dados são escritos e lidos eletricamente, com persistência da informação
 - ▶ Ex: ROM, EEPROM, Flash

Memória

- ▶ Endereçamento de 32 bits (4 GiB)
 - ▶ Diagrama de blocos



Memória

- ▶ Tipos de endereçamento
 - ▶ Valor imediato para acessar endereços constantes

```
1 // R1 = Memória[0x00000100]
2 l8 r1, [0x100]
3 // R2 = Memória[0x00000100]
4 l16 r2, [0x80]
5 // R3 = Memória[0x00000100]
6 l32 r3, [0x40]
```

Memória

- ▶ Tipos de endereçamento
 - ▶ Baseado em registrador para acesso indireto

```
1 // R1 = 0x100
2 mov r1, 0x100
3 // R2 = Memória[0x00000100]
4 l8 r2, [r1]
5 // R1 = R1 >> 1;
6 sra r1, 1
7 // R3 = Memória[0x00000100]
8 l16 r3, [r1]
9 // R1 = R1 >> 1;
10 sra r1, 1
11 // R4 = Memória[0x00000100]
12 l32 r4, [r1]
```

Memória

- ▶ Tipos de endereçamento
 - ▶ Indireto e imediato para acesso linear

```
1 // R1 = 0x40
2 mov r1, 0x40
3 // R2 = Memória[0x00000100]
4 l32 r2, [r1 + 0]
5 // R3 = Memória[0x00000104]
6 l32 r3, [r1 + 1]
7 // R4 = Memória[0x00000108]
8 l32 r4, [r1 + 2]
```


Memória

- ▶ Tamanho típico dos tipos de dados
 - ▶ Tipos inteiros

Tipo	Bits	Com sinal	Sem sinal
char	8	$-2^7 \leftrightarrow +2^7 - 1$	$0 \leftrightarrow 2^8 - 1$
short	16	$-2^{15} \leftrightarrow +2^{15} - 1$	$0 \leftrightarrow +2^{16} - 1$
int ¹	$16 \leftrightarrow 64$	$-2^{63} \leftrightarrow +2^{63} - 1$	$0 \leftrightarrow +2^{64} - 1$
long ¹	$32 \leftrightarrow 64$	$-2^{63} \leftrightarrow +2^{63} - 1$	$0 \leftrightarrow +2^{64} - 1$
long long ¹	64	$-2^{63} \leftrightarrow +2^{63} - 1$	$0 \leftrightarrow +2^{64} - 1$

¹ Valores dependentes da plataforma

Memória

- ▶ Tamanho típico dos tipos de dados
 - ▶ Tipos inteiros

Tipo	Bits	Com sinal	Sem sinal
char	8	$-2^7 \leftrightarrow +2^7 - 1$	$0 \leftrightarrow 2^8 - 1$
short	16	$-2^{15} \leftrightarrow +2^{15} - 1$	$0 \leftrightarrow +2^{16} - 1$
int ¹	$16 \leftrightarrow 64$	$-2^{63} \leftrightarrow +2^{63} - 1$	$0 \leftrightarrow +2^{64} - 1$
long ¹	$32 \leftrightarrow 64$	$-2^{63} \leftrightarrow +2^{63} - 1$	$0 \leftrightarrow +2^{64} - 1$
long long ¹	64	$-2^{63} \leftrightarrow +2^{63} - 1$	$0 \leftrightarrow +2^{64} - 1$

¹ Valores dependentes da plataforma

Padronização por tamanho e sinal em *stdint.h*

- ▶ Tamanho típico dos tipos de dados
 - ▶ Tipos reais

Tipo	Bits	Alcance
float	32	$1,2E^{-38} \leftrightarrow 3,4E^{+38}$
double	64	$2,3E^{-308} \leftrightarrow 1,7E^{+308}$
long double ¹	80 \leftrightarrow 128	$3,4E^{-4932} \leftrightarrow 1,1E^{+4932}$

¹ Valores dependentes da plataforma

Memória

- ▶ Alinhamento dos dados na memória
 - ▶ Dados alinhados
 - ▶ Preenchimento com zeros (*padding*)

```
1 char a = '@';  
2 uint32_t b = 0xABCD1122;  
3 int16_t c = 0x1234;
```

Big-endian

0x00000000	00	00	00	40
0x00000004	AB	CD	11	22
0x00000008	00	00	12	34

Memória

- ▶ Alinhamento dos dados na memória
 - ▶ Dados alinhados
 - ▶ Preenchimento com zeros (*padding*)

```
1 char a = '@';  
2 uint32_t b = 0xABCD1122;  
3 int16_t c = 0x1234;
```

Big-endian

0x00000000	00	00	00	40
0x00000004	AB	CD	11	22
0x00000008	00	00	12	34

Memória

- ▶ Alinhamento dos dados na memória
 - ▶ Dados alinhados
 - ▶ Preenchimento com zeros (*padding*)

```
1 char a = '@';  
2 uint32_t b = 0xABCD1122;  
3 int16_t c = 0x1234;
```

Little-endian

0x00000000	00	40	00	00	00
0x00000004	04	22	11	CD	AB
0x00000008	08	34	12	00	00

Memória

- ▶ Alinhamento dos dados na memória
 - ▶ Dados alinhados
 - ▶ Preenchimento com zeros (*padding*)

```
1 char a = '@';  
2 uint32_t b = 0xABCD1122;  
3 int16_t c = 0x1234;
```

Little-endian

0x00000000	00	40	00	00	00
0x00000004	04	22	11	CD	AB
0x00000008	08	34	12	00	00

Memória

- ▶ Alinhamento dos dados na memória
 - ▶ Dados não alinhados
 - ▶ Acesso complexo e específico

```
1 char a = '@';  
2 uint32_t b = 0xABCD1122;  
3 int16_t c = 0x1234;
```

Big-endian

0x000000100	40	AB	CD	
0x000000104	22	12	34	-

Memória

- ▶ Alinhamento dos dados na memória
 - ▶ Dados não alinhados
 - ▶ Acesso complexo e específico

```
1 char a = '@';  
2 uint32_t b = 0xABCD1122;  
3 int16_t c = 0x1234;
```

Little-endian

0x00000000	40	22	11	CD
0x00000004	AB	34	12	-

Memória

- ▶ Alinhamento da memória
 - ▶ Qual é a melhor abordagem?

Alinhados

- ✓ Desempenho no acesso
- ✓ Simplicidade de uso
- ✗ Desperdício de espaço

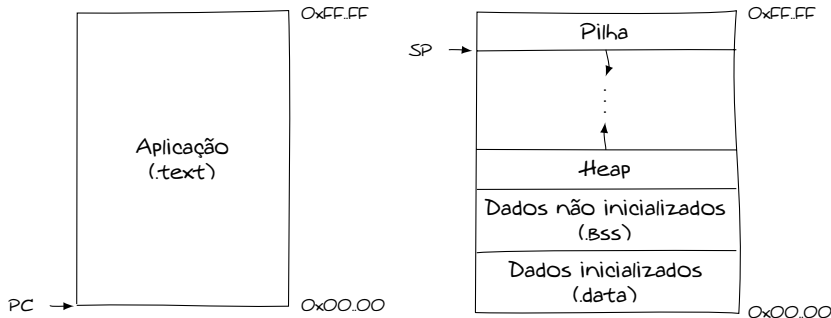
Não alinhados

- ✓ Economia de espaço
- ✗ Complexidade de acesso
- ✗ Suporte da arquitetura

Memória

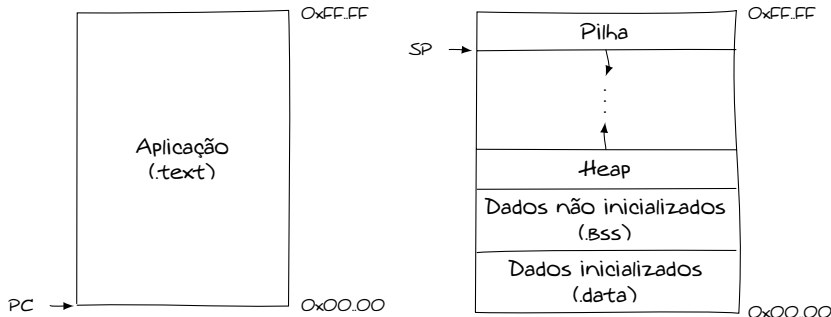
► Arquitetura Harvard

- São utilizadas duas memórias fisicamente separadas para armazenar as instruções e os dados
- A memória de programa só permite leitura, enquanto que a memória de dados permite escrita e leitura



Memória

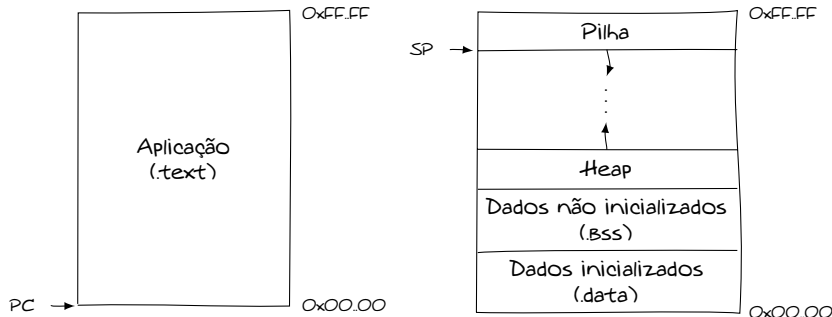
- ▶ Arquitetura Harvard
 - ▶ São utilizadas duas memórias fisicamente separadas para armazenar as instruções e os dados
 - ▶ A memória de programa só permite leitura, enquanto que a memória de dados permite escrita e leitura



✓ Acesso paralelo das instruções e dos dados

Memória

- ▶ Arquitetura Harvard
 - ▶ São utilizadas duas memórias fisicamente separadas para armazenar as instruções e os dados
 - ▶ A memória de programa só permite leitura, enquanto que a memória de dados permite escrita e leitura

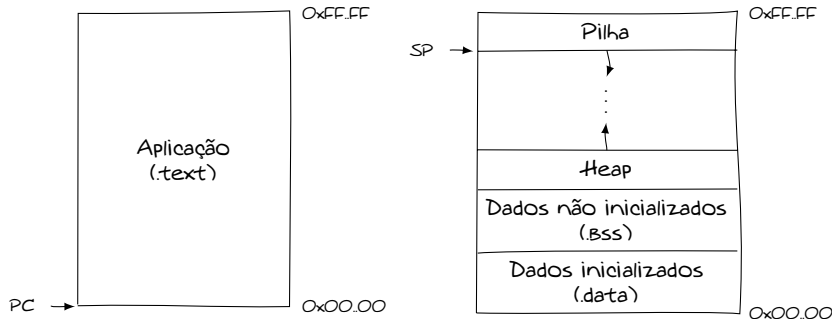


✓ Proteção contra modificação da aplicação

Memória

► Arquitetura Harvard

- São utilizadas duas memórias fisicamente separadas para armazenar as instruções e os dados
- A memória de programa só permite leitura, enquanto que a memória de dados permite escrita e leitura

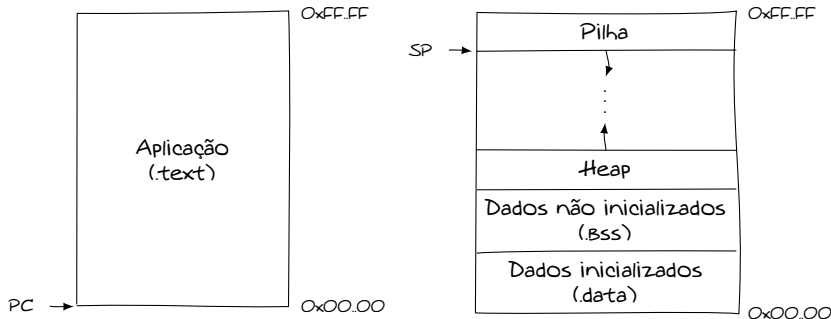


✗ Componente adicional de memória

Memória

► Arquitetura Harvard

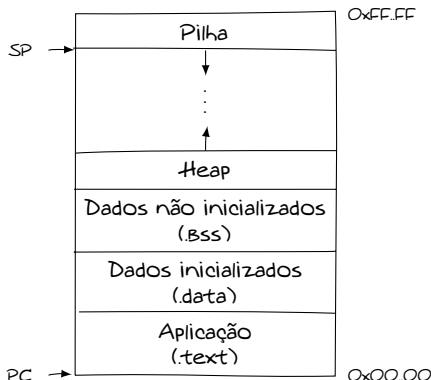
- São utilizadas duas memórias fisicamente separadas para armazenar as instruções e os dados
- A memória de programa só permite leitura, enquanto que a memória de dados permite escrita e leitura



✗ Pior aproveitamento da capacidade

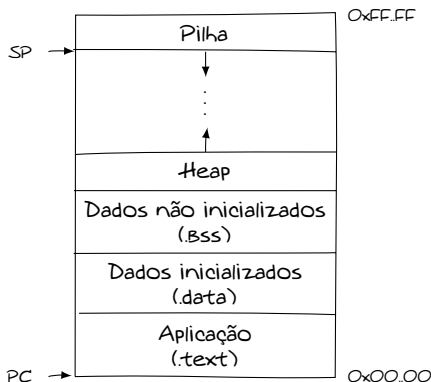
Memória

- ▶ Arquitetura Von Neumann (Princeton)
 - ▶ É utilizada somente uma memória compartilhada para armazenar as instruções e os dados
 - ▶ Tanto o código de programa como os dados podem ser acessados para escrita e leitura



Memória

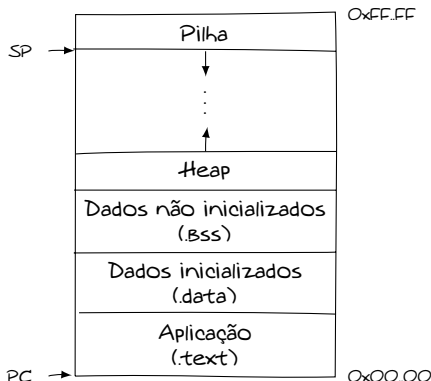
- ▶ Arquitetura Von Neumann (Princeton)
 - ▶ É utilizada somente uma memória compartilhada para armazenar as instruções e os dados
 - ▶ Tanto o código de programa como os dados podem ser acessados para escrita e leitura



✓ Capacidade de auto-modificação da aplicação

Memória

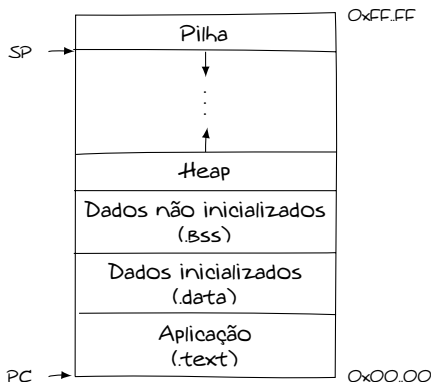
- ▶ Arquitetura Von Neumann (Princeton)
 - ▶ É utilizada somente uma memória compartilhada para armazenar as instruções e os dados
 - ▶ Tanto o código de programa como os dados podem ser acessados para escrita e leitura



✓ Uso eficiente da memória

Memória

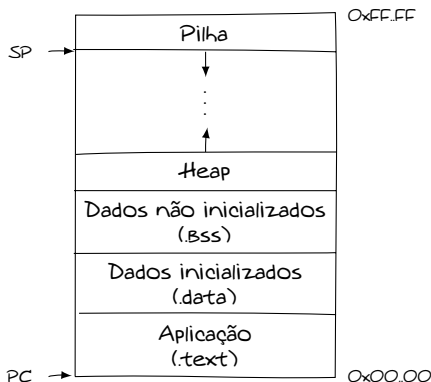
- ▶ Arquitetura Von Neumann (Princeton)
 - ▶ É utilizada somente uma memória compartilhada para armazenar as instruções e os dados
 - ▶ Tanto o código de programa como os dados podem ser acessados para escrita e leitura



✗ Acesso sequencial de instruções e dados

Memória

- ▶ Arquitetura Von Neumann (Princeton)
 - ▶ É utilizada somente uma memória compartilhada para armazenar as instruções e os dados
 - ▶ Tanto o código de programa como os dados podem ser acessados para escrita e leitura



✗ Mais vulnerável a ataques e falhas

Exemplo

- ▶ Considerando uma arquitetura *load-store* e o código fonte abaixo, calcule seu tempo de execução
 - ▶ O processador opera com um frequência de 4 GHz e a memória possui latência média de 10 ns

```
1 // Padrão de tipos por tamanho
2 #include <stdint.h>
3 // Função principal
4 int main() {
5     uint32_t a = 1, i = 1000;
6     while(i > 0) {
7         a = a + i;
8         i--;
9     }
10    return 0;
11 }
```

- ▶ Compare o tempo de execução, caso todas as operações acessem diretamente a memória