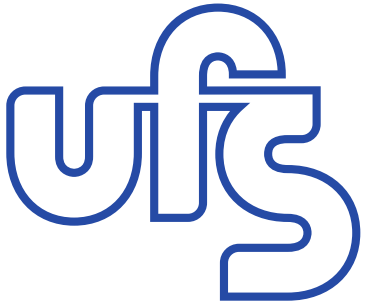
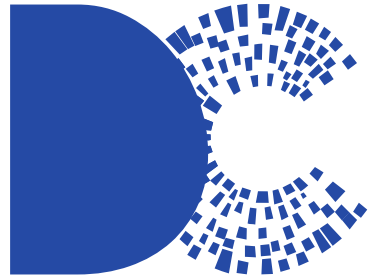




Memória e registradores

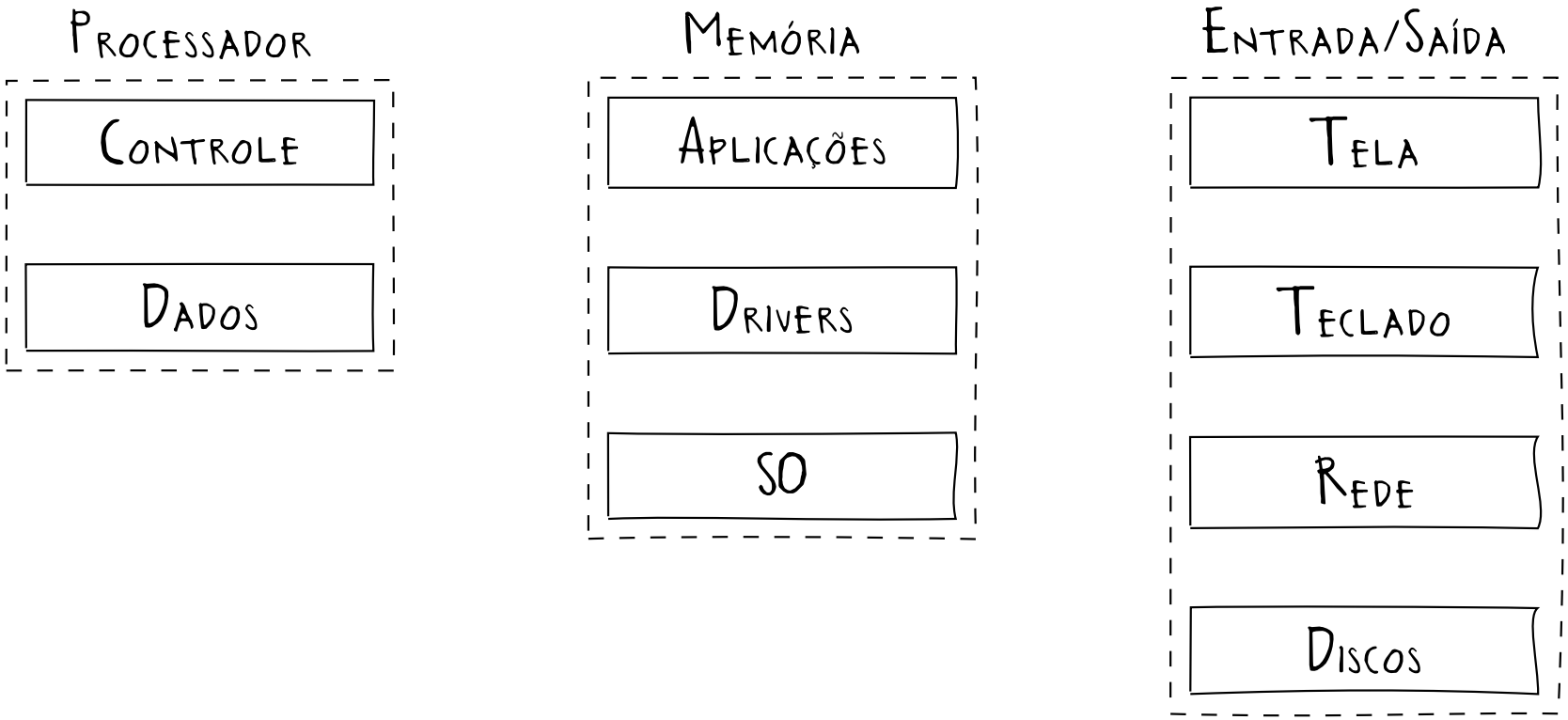
Arquitetura de Computadores

Bruno Prado

Departamento de Computação / UFS

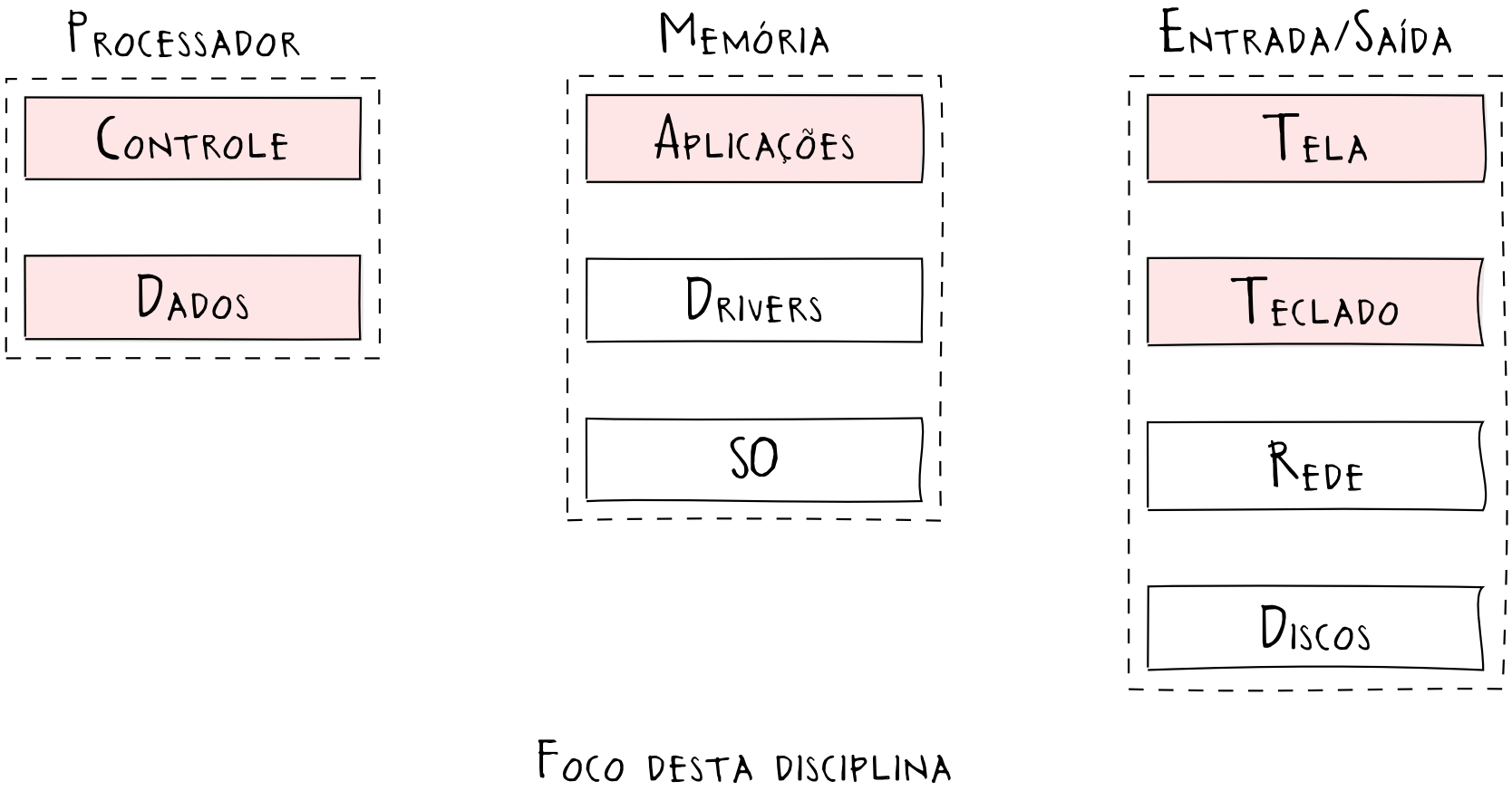
Introdução

- Componentes do computador



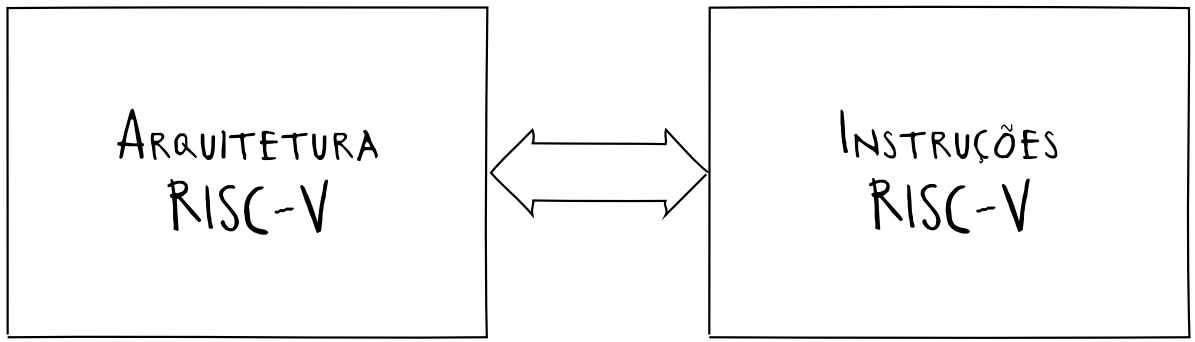
Introdução

- Componentes do computador



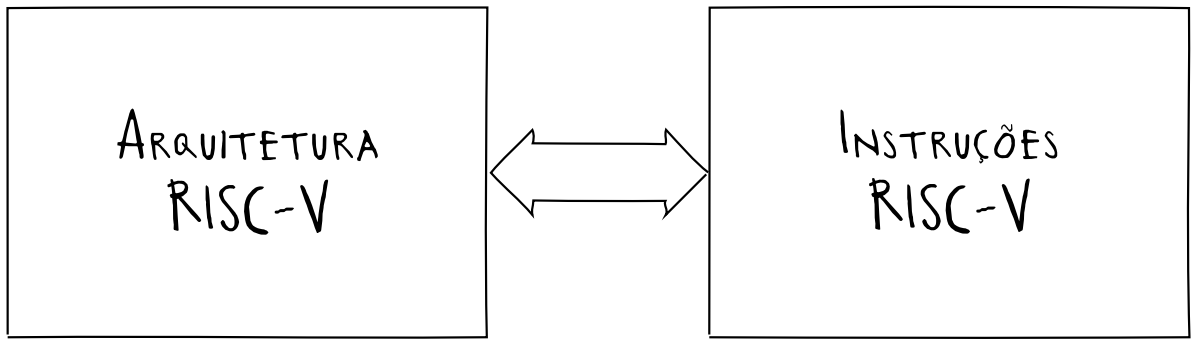
Introdução

- O que é o conjunto de instruções da arquitetura?
  - Um conjunto de instruções da arquitetura é o idioma que um computador é capaz de interpretar o comportamento



Introdução

- O que é o conjunto de instruções da arquitetura?
  - Um conjunto de instruções da arquitetura é o idioma que um computador é capaz de interpretar o comportamento



  - As instruções são equivalentes às palavras de um texto e cada arquitetura possui pelo menos uma linguagem que o processador é capaz de entender

Instruction Set Architecture (ISA)

Introdução

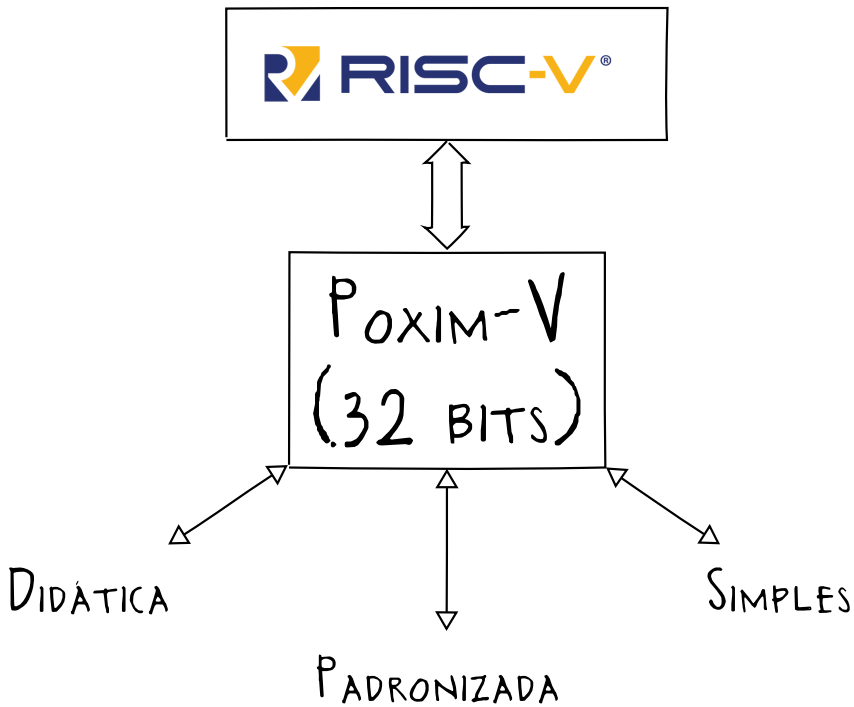
- Por que usar a arquitetura RISC-V?

Introdução

- Por que usar a arquitetura RISC-V?
  - É um padrão aberto de arquitetura de computadores com 32, 64 e 128 bits, suportando instruções personalizadas e com várias extensões disponíveis

Introdução

- Por que usar a arquitetura RISC-V?
  - É um padrão aberto de arquitetura de computadores com 32, 64 e 128 bits, suportando instruções personalizadas e com várias extensões disponíveis
  - Utilizado em aplicações comerciais, sem pagamento de *royalties*, com suporte do GCC e do Linux



Introdução

- Linguagem C

```c
// E/S padrão
#include <stdio.h>
// Função principal
int main() {
    // Imprimindo mensagem no terminal
    printf("Hello World!\n");
    // Retornando 0 (sucesso)
    return 0;
}
```

Introdução

- Linguagem de montagem (*assembly*)

```
1  .section .text
2  .global  main
3  main:
4      addi sp,sp,-16
5      sw   ra,12(sp)
6      sw   s0,8(sp)
7      addi s0,sp,16
8      lui  a5,%hi(message)
9      addi a0,a5,%lo(message)
10     call puts
11     li   a5,0
12     mv   a0,a5
13     lw   ra,12(sp)
14     lw   s0,8(sp)
15     addi sp,sp,16
16     jr   ra
17 .section .rodata
18 message:
19     .string "Hello␣World!"
```

Introdução

- Linguagem de máquina

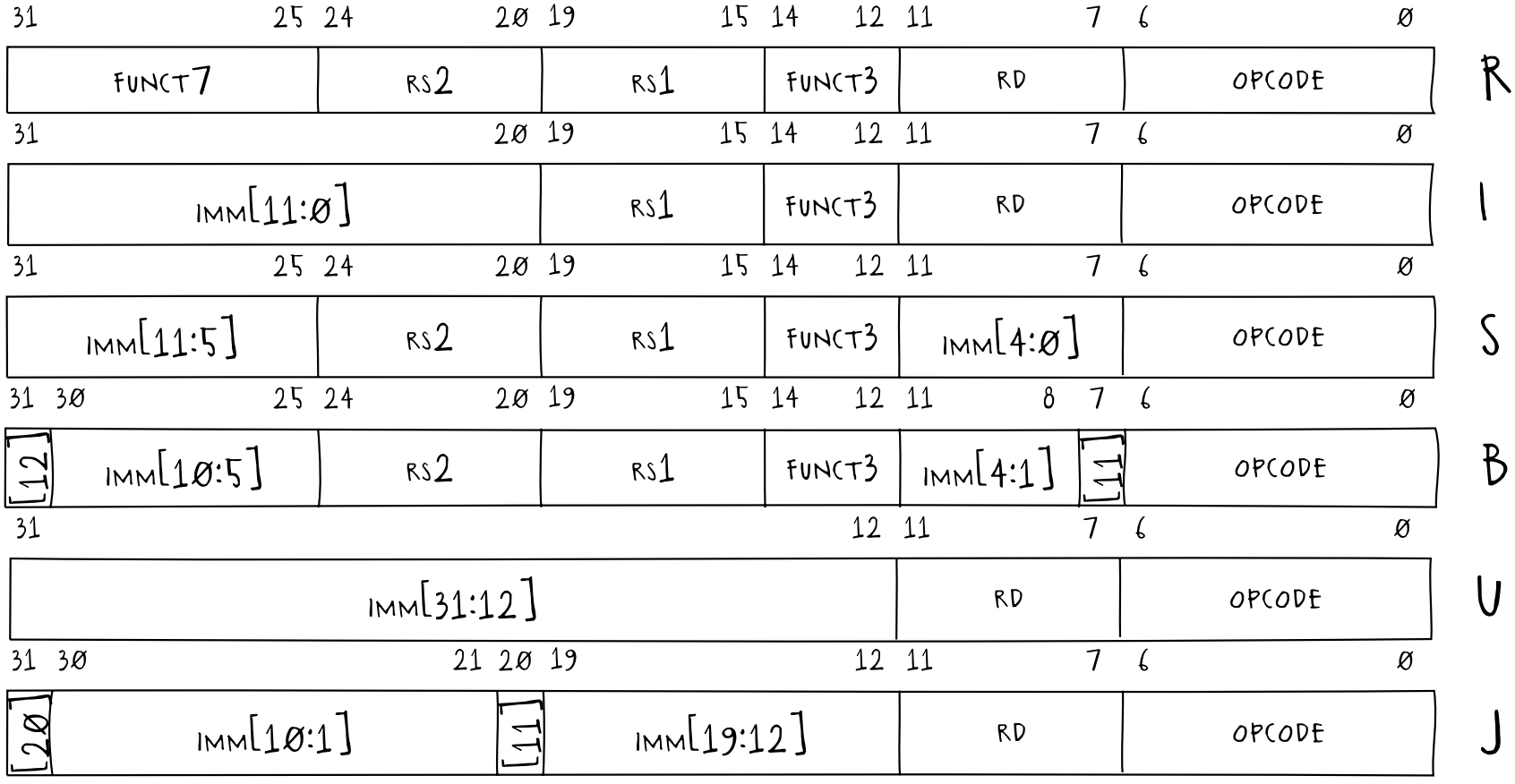
```
1  13 01 01 FF
2  23 26 11 00
3  23 24 81 00
4  13 04 01 01
5  B7 07 00 00
6  13 85 07 00
7  97 00 00 00
8  E7 80 00 00
9  93 07 00 00
10 13 85 07 00
11 83 20 C1 00
12 03 24 81 00
13 13 01 01 01
14 67 80 00 00
15 48 65 6C 6C
16 6F 20 57 6F
17 72 6C 64 21
18 00 00 00 00
```

Introdução

- Como dizer ao computador o que deve ser feito?

Introdução

- Como dizer ao computador o que deve ser feito?
  - Especificação do comportamento de cada instrução, além da formatação dos parâmetros de entrada e de saída

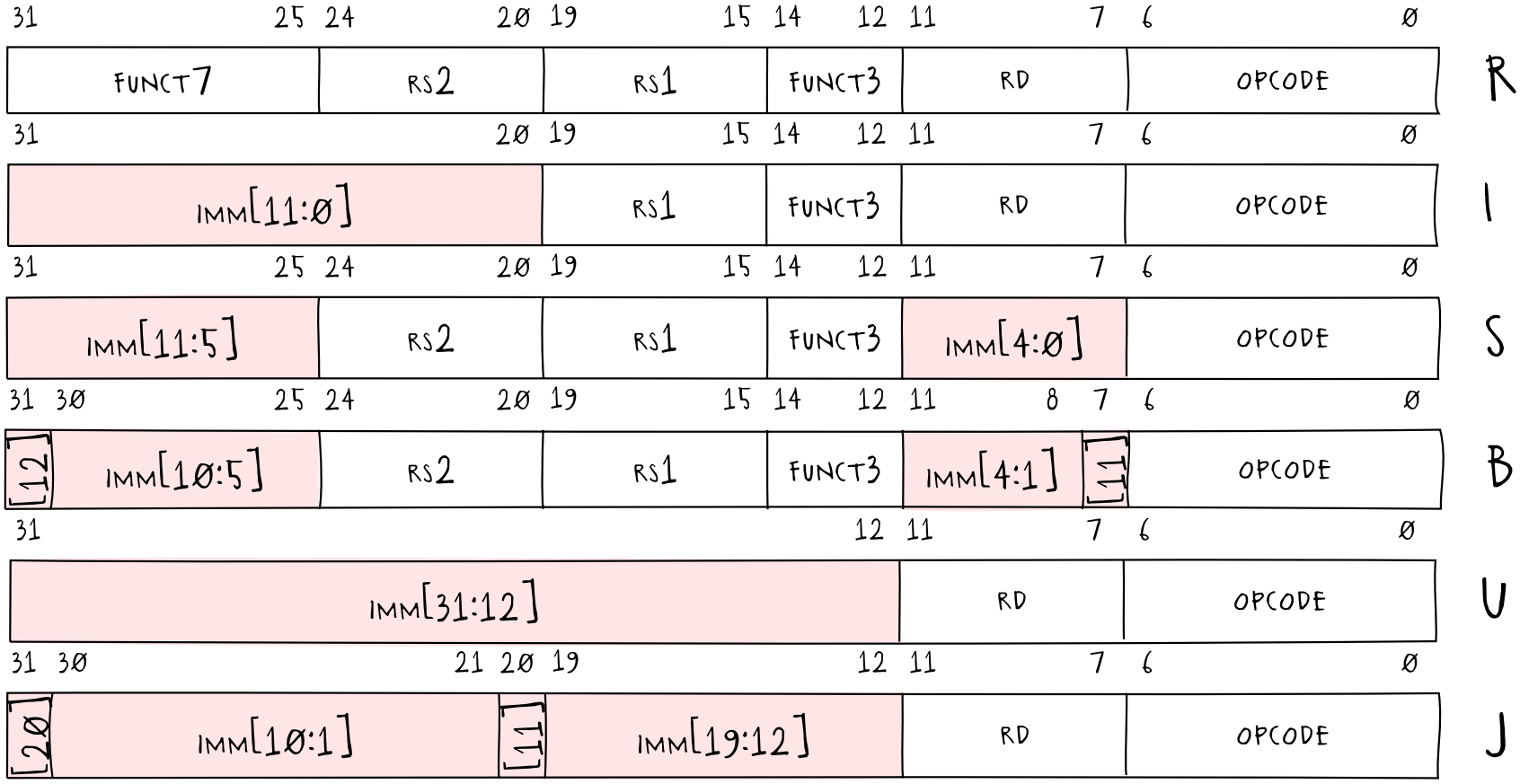


A FORMATAÇÃO REGULAR E SIMPLES DAS INSTRUÇÕES

PERMITE A DECODIFICAÇÃO E EXECUÇÃO MAIS EFICIENTE DAS OPERAÇÕES

Introdução

- Como dizer ao computador o que deve ser feito?
  - Especificação do comportamento de cada instrução, além da formatação dos parâmetros de entrada e de saída



Os campos imediatos armazenam o operando na própria instrução

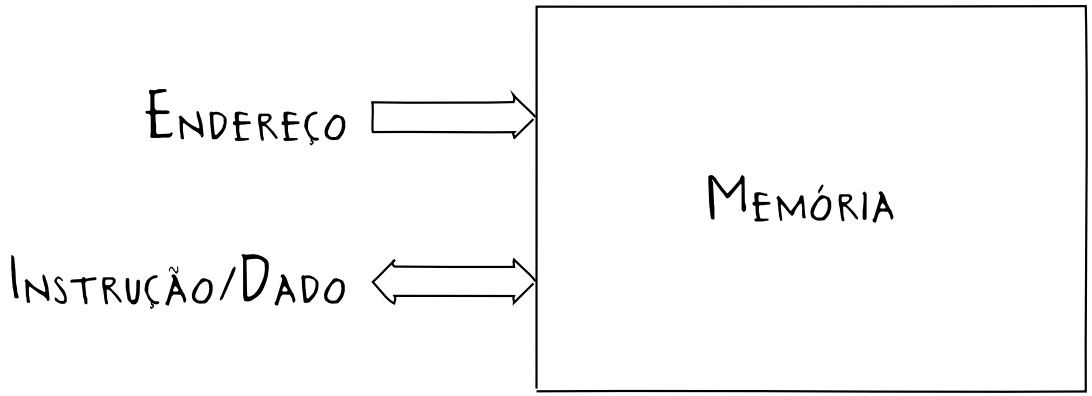
Introdução

- Como as informações são representadas?
  - *Nibble* (4 bits) $\equiv$ Dígito hexadecimal (base 16)

Decimal	Binário	Hexadecimal
0_{10}	0000_2	0_{16}
1_{10}	0001_2	1_{16}
2_{10}	0010_2	2_{16}
$\vdots$	$\vdots$	$\vdots$
13_{10}	1101_2	D_{16}
14_{10}	1110_2	E_{16}
15_{10}	1111_2	F_{16}

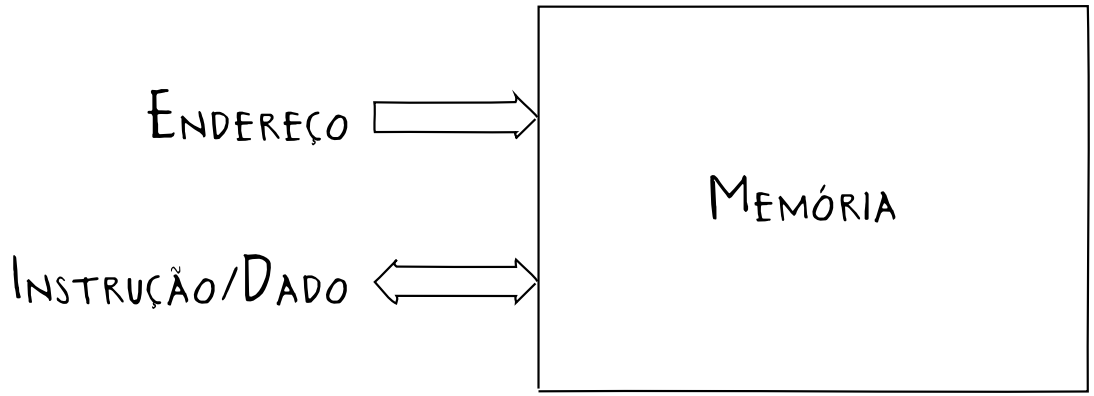
Memória

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



Memória

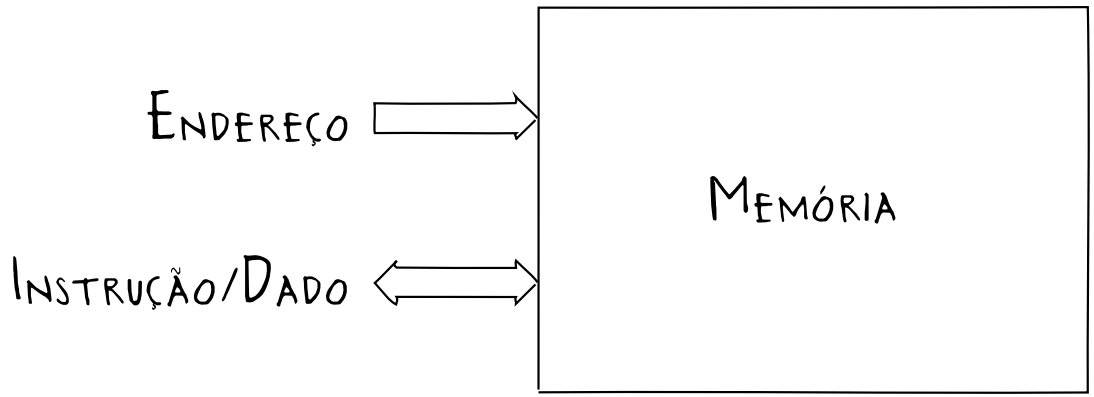
- O que é uma memória?
  - É um dispositivo semicondutor para armazenamento em estado sólido de informações



  - Permite o endereçamento de posições para realização de operações de escrita e de leitura

Memória

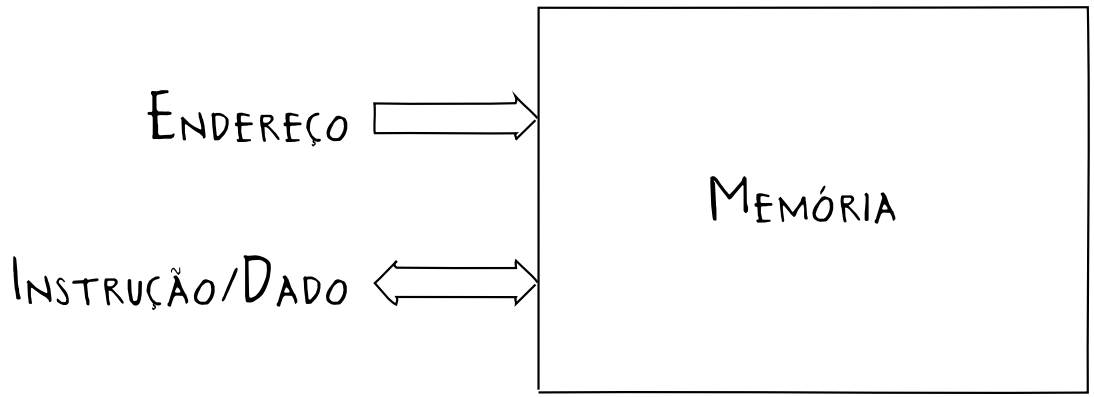
- O que é uma memória?
  - É um dispositivo semicondutor para armazenamento em estado sólido de informações



  - Permite o endereçamento de posições para realização de operações de escrita e de leitura
  - Todos os dados são codificados em formato binário

Memória

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

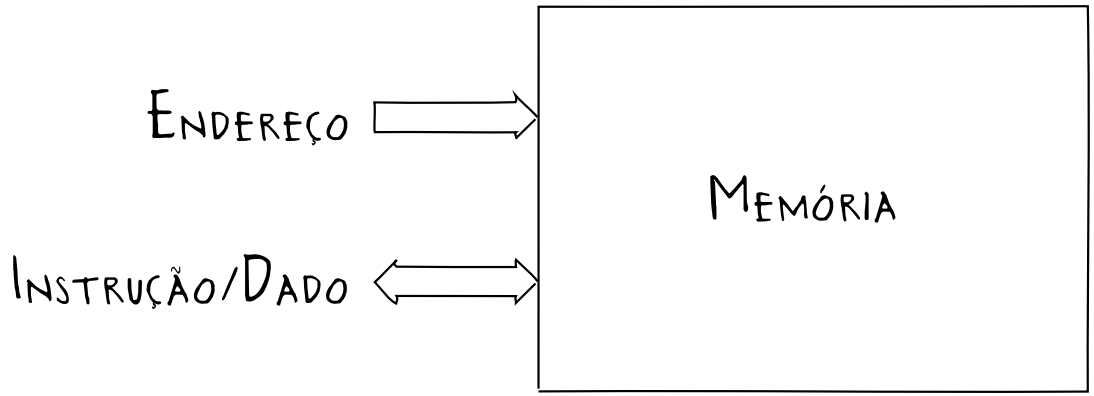


  - Permite o endereçamento de posições para realização de operações de escrita e de leitura
  - Todos os dados são codificados em formato binário

A memória principal usa tecnologia DRAM (volátil)

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

O disco de estado sólido (SSD) utiliza FLASH (não volátil)

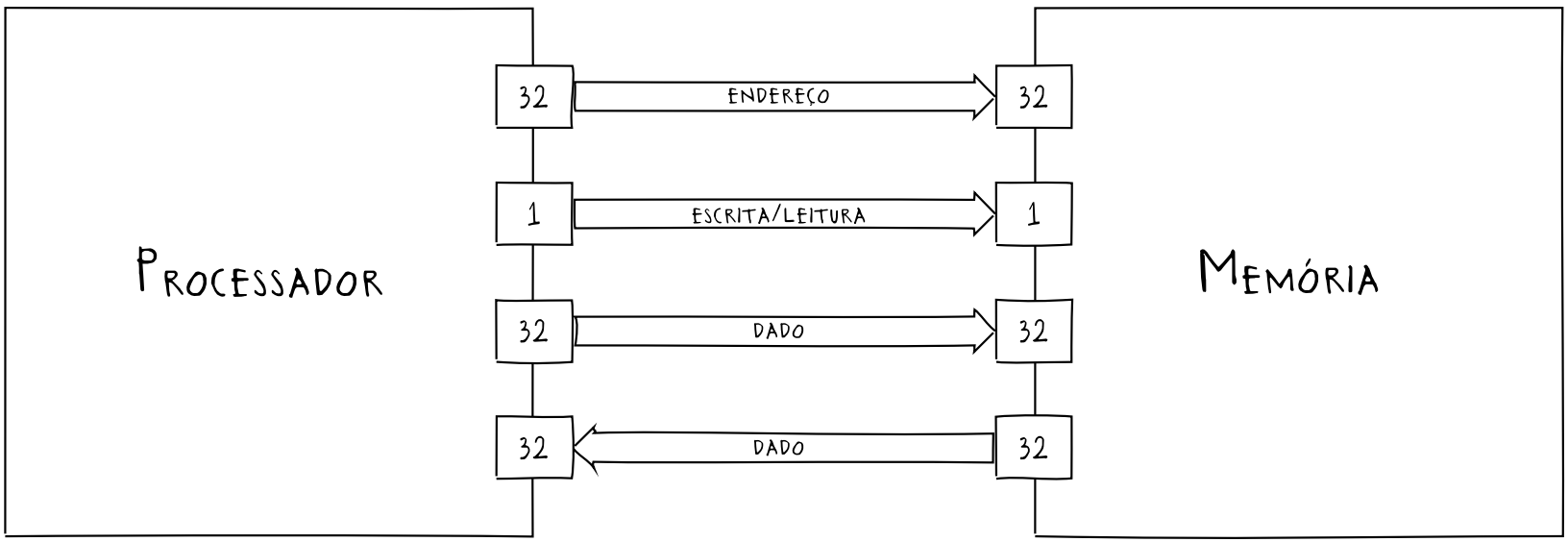
Memória

- Análise comparativa das memórias

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$ 1 / GiB	~10 ns
FLASH	0,1 <-> 32 TB	~US$ 0,03 / GB	~1 ms

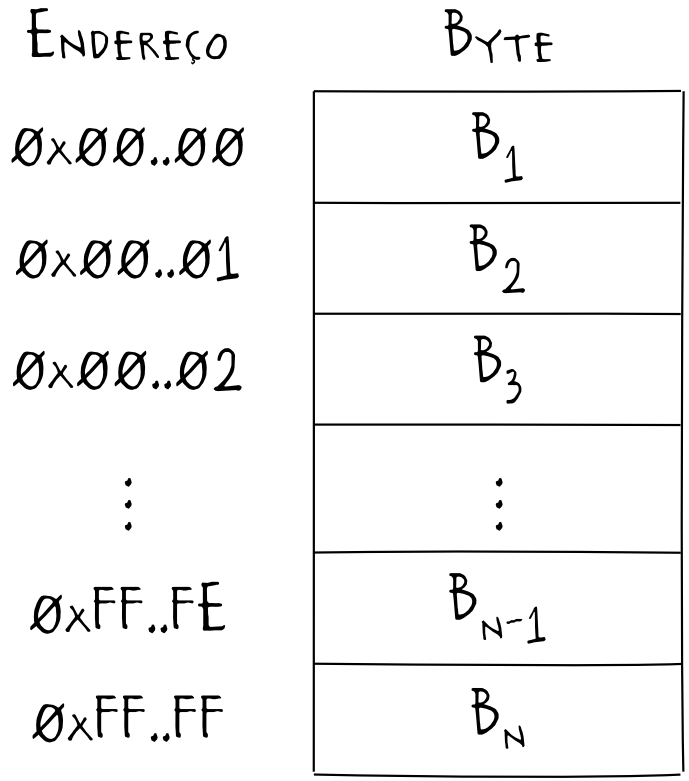
Memória

- Endereçamento de 32 bits (4 GiB)



Memória

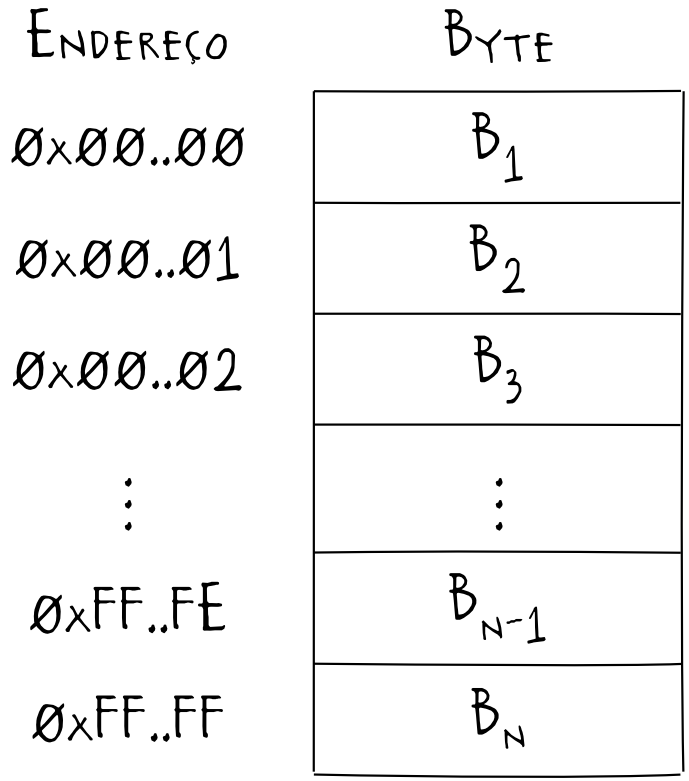
- Como é organizada a memória?



Os dados são divididos em bytes

Memória

- Como é organizada a memória?

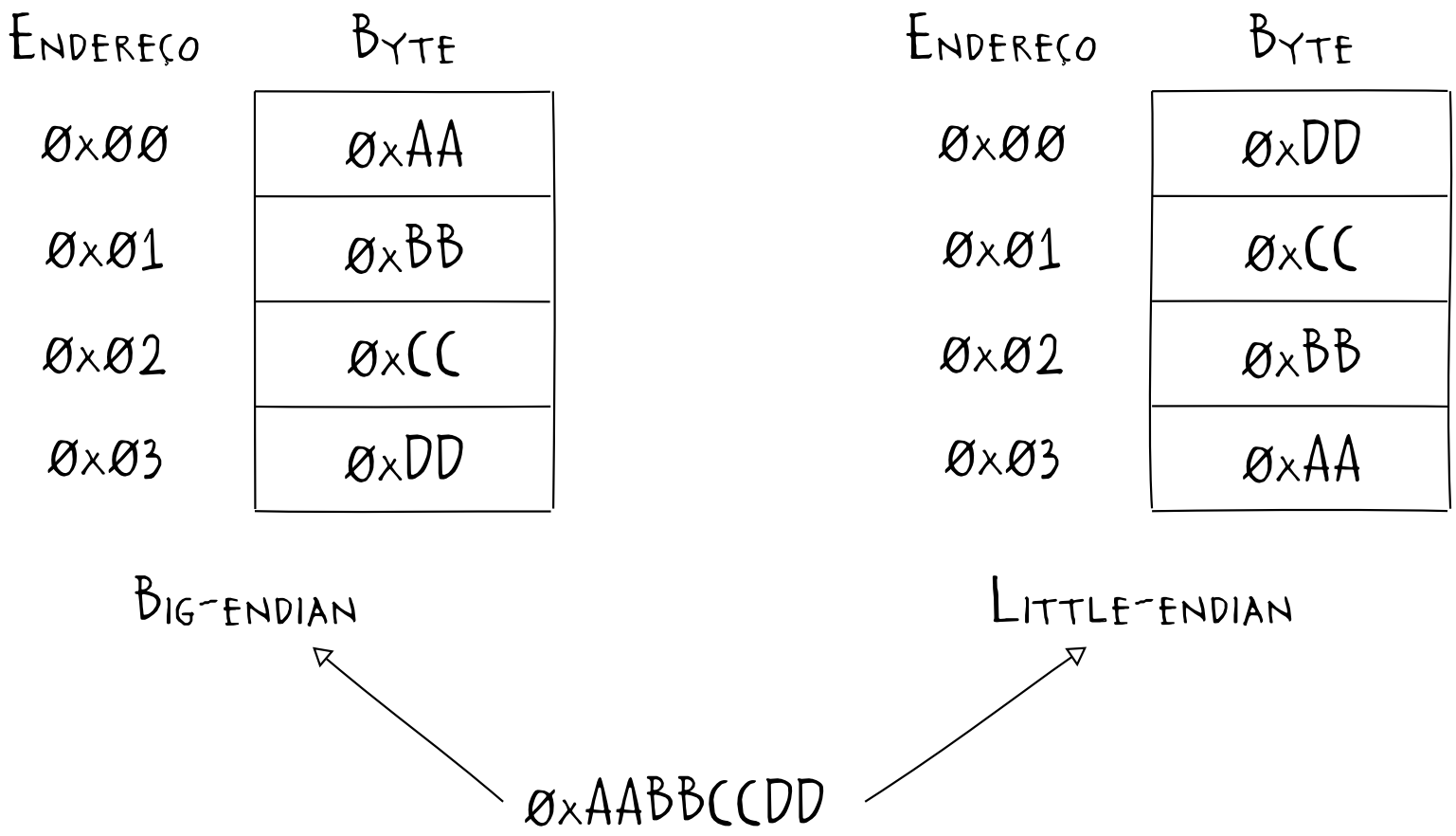


Os dados são divididos em bytes

4 GiB -> $N = 2^{32} = 4.294.967.296$ bytes

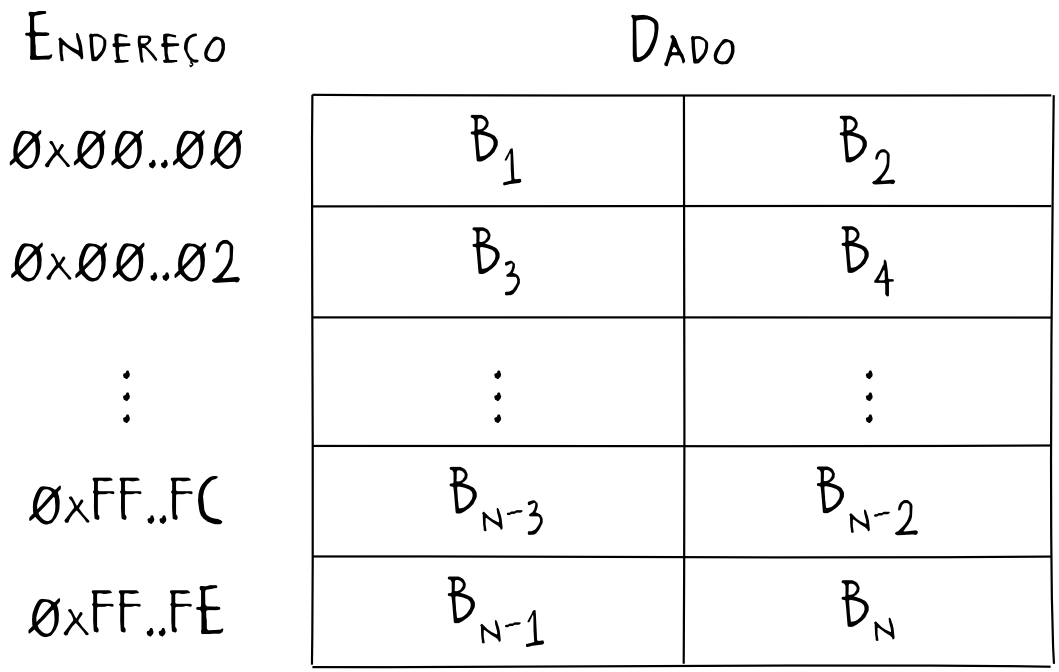
Memória

- Como armazenar dados com mais de 1 byte?
  - Mais significativo primeiro (*big-endian*)
  - Menos significativo primeiro (*little-endian*)



Memória

- Como é feito o endereçamento na memória?



Definido pelo alinhamento: 16 bits (2 bytes)

Memória

- Como é feito o endereçamento na memória?

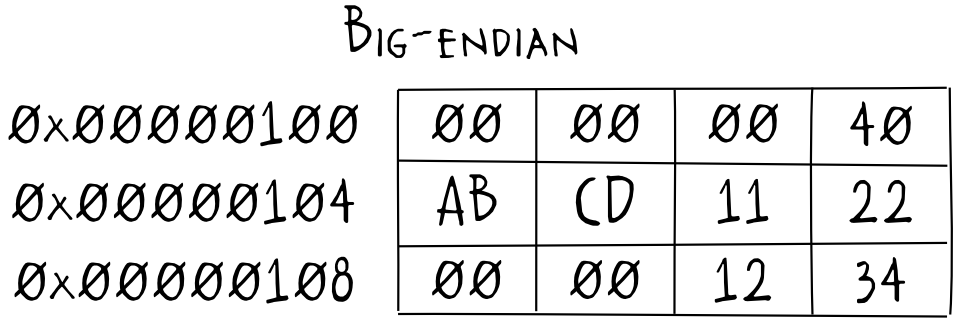
Endereço	Dado			
0x00..00	B_1	B_2	B_3	B_4
0x00..04	B_5	B_6	B_7	B_8
⋮	⋮	⋮	⋮	⋮
0xFF..F8	B_{N-7}	B_{N-6}	B_{N-5}	B_{N-4}
0xFF..FC	B_{N-3}	B_{N-2}	B_{N-1}	B_N

Definido pelo alinhamento: 32 bits (4 bytes)

Memória

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

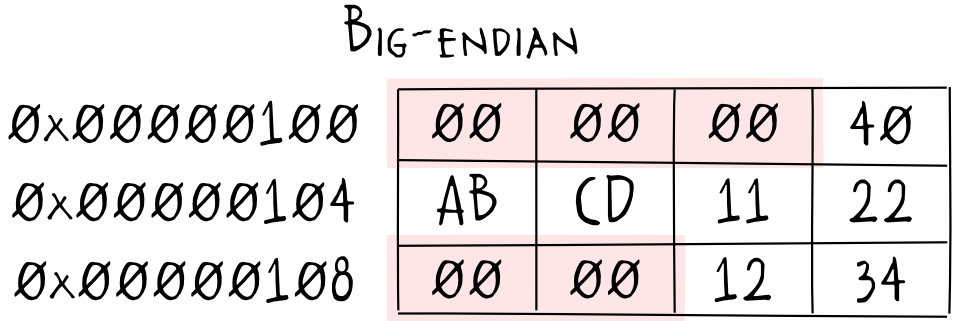
```
1 char a = '@';
2 uint32_t b = 0xABCD1122;
3 int16_t c = 0x1234;
```



Memória

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

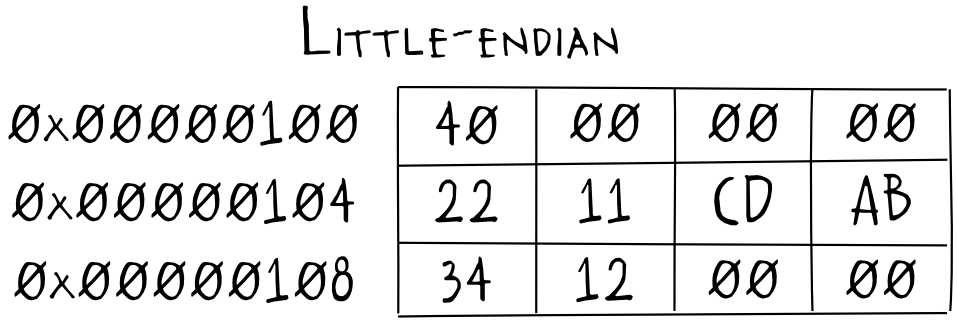
```
1 char a = '@';
2 uint32_t b = 0xABCD1122;
3 int16_t c = 0x1234;
```



Memória

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

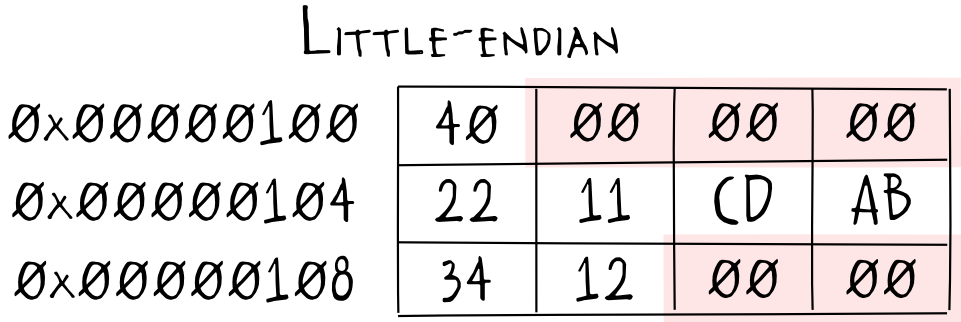
```
1 char a = '@';
2 uint32_t b = 0xABCD1122;
3 int16_t c = 0x1234;
```



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;
```



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;
```

✓ Desempenho no acesso

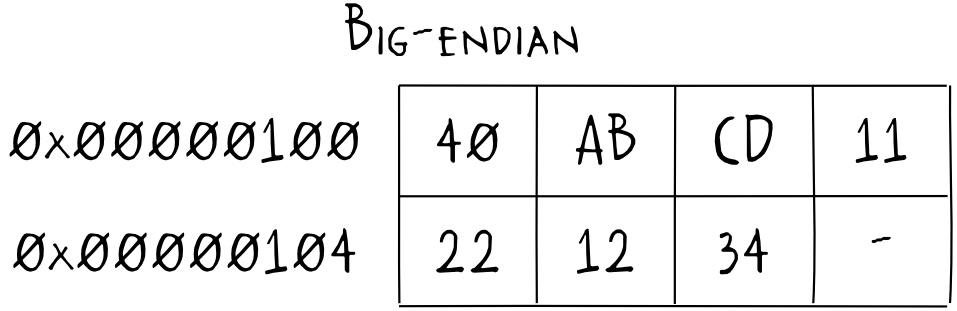
✓ Simplicidade de uso

✗ Desperdício de espaço

Memória

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

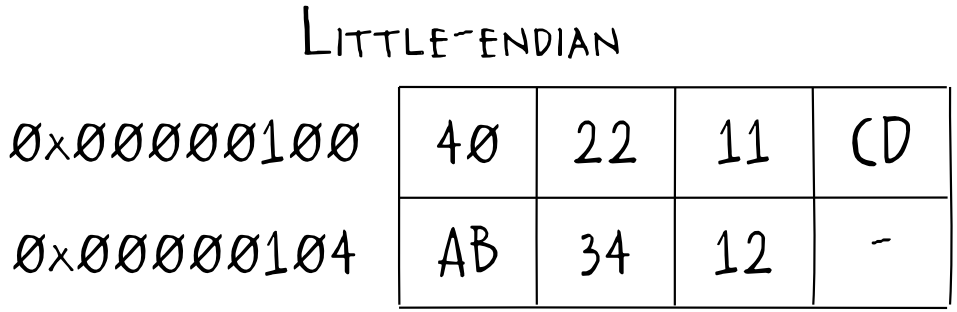
```
1 char a = '@';
2 uint32_t b = 0xABCD1122;
3 int16_t c = 0x1234;
```



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;
```



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;
```

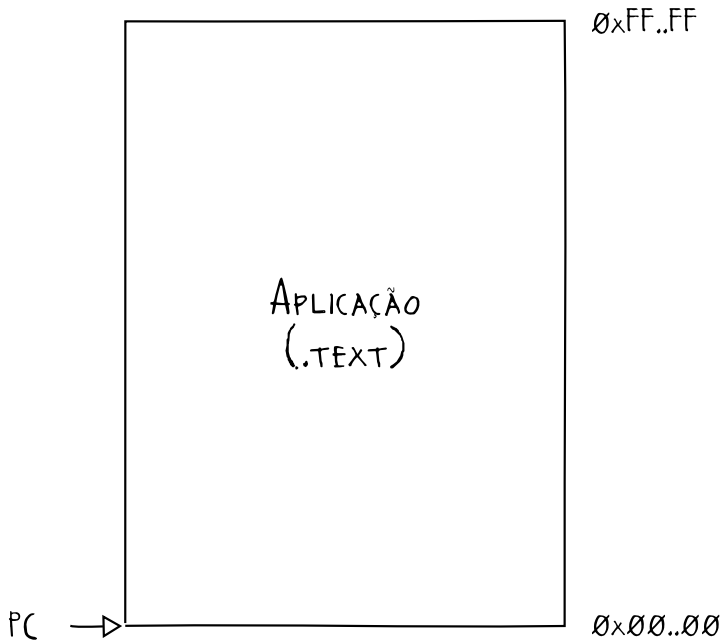
✓ Economia de espaço

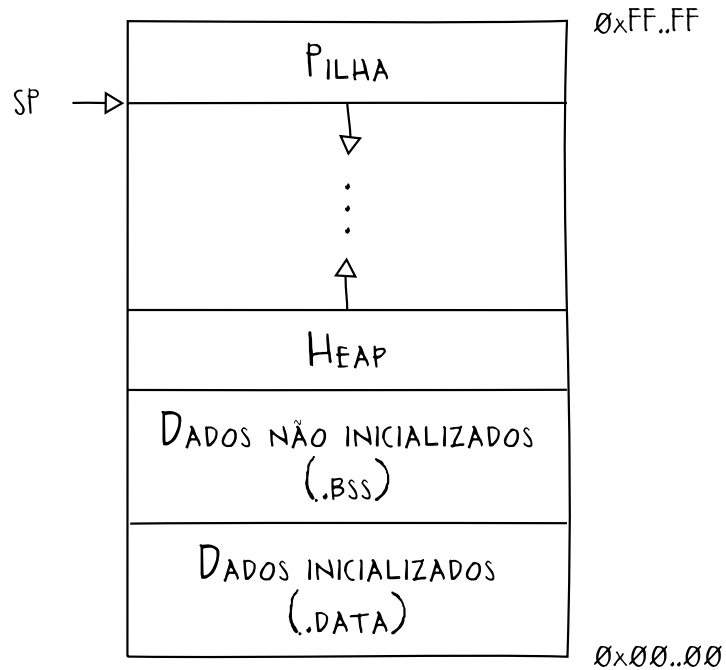
✗ Complexidade de acesso

✗ Suporte da arquitetura

Memória

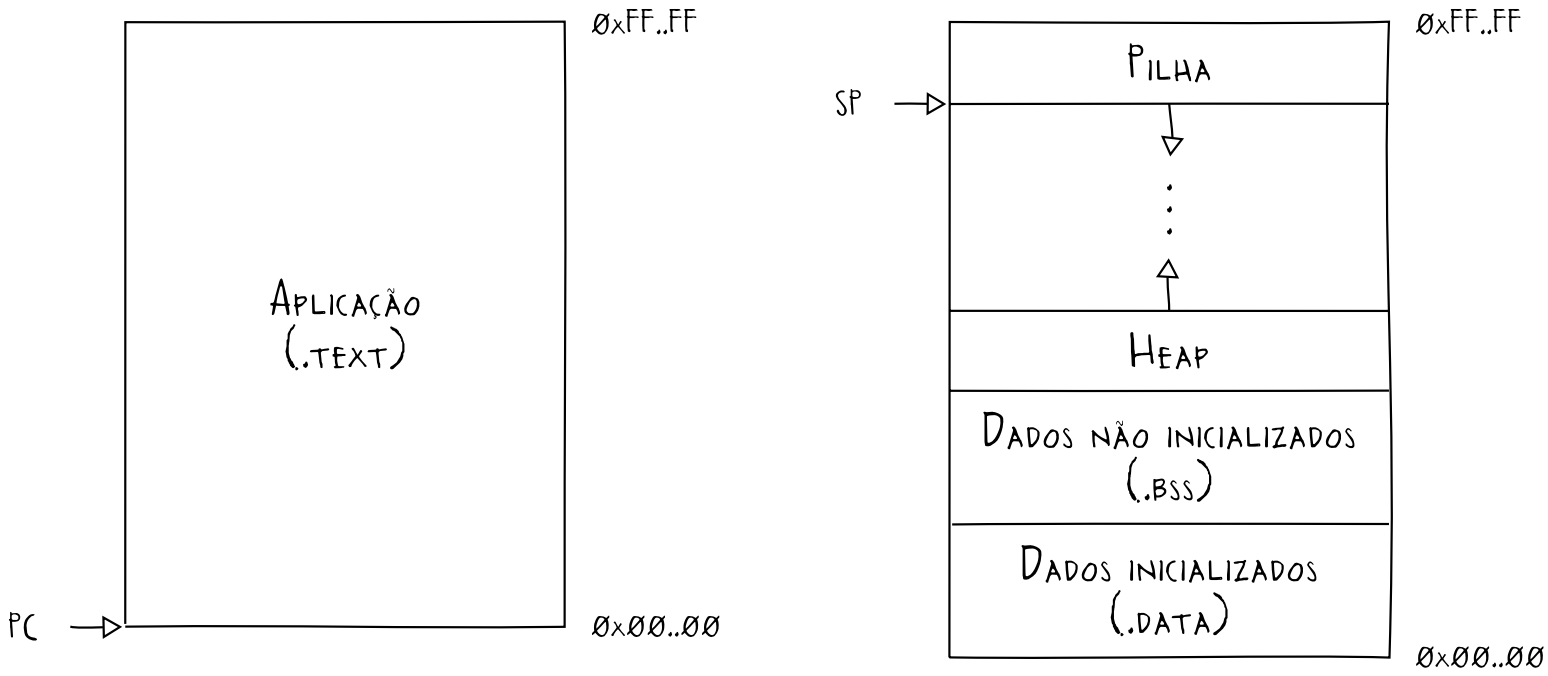
- Arquitetura Harvard
  - São utilizadas duas memórias fisicamente separadas para armazenar as instruções e os dados
  - A memória de programa só permite leitura, enquanto que a memória de dados permite escrita e leitura





Memória

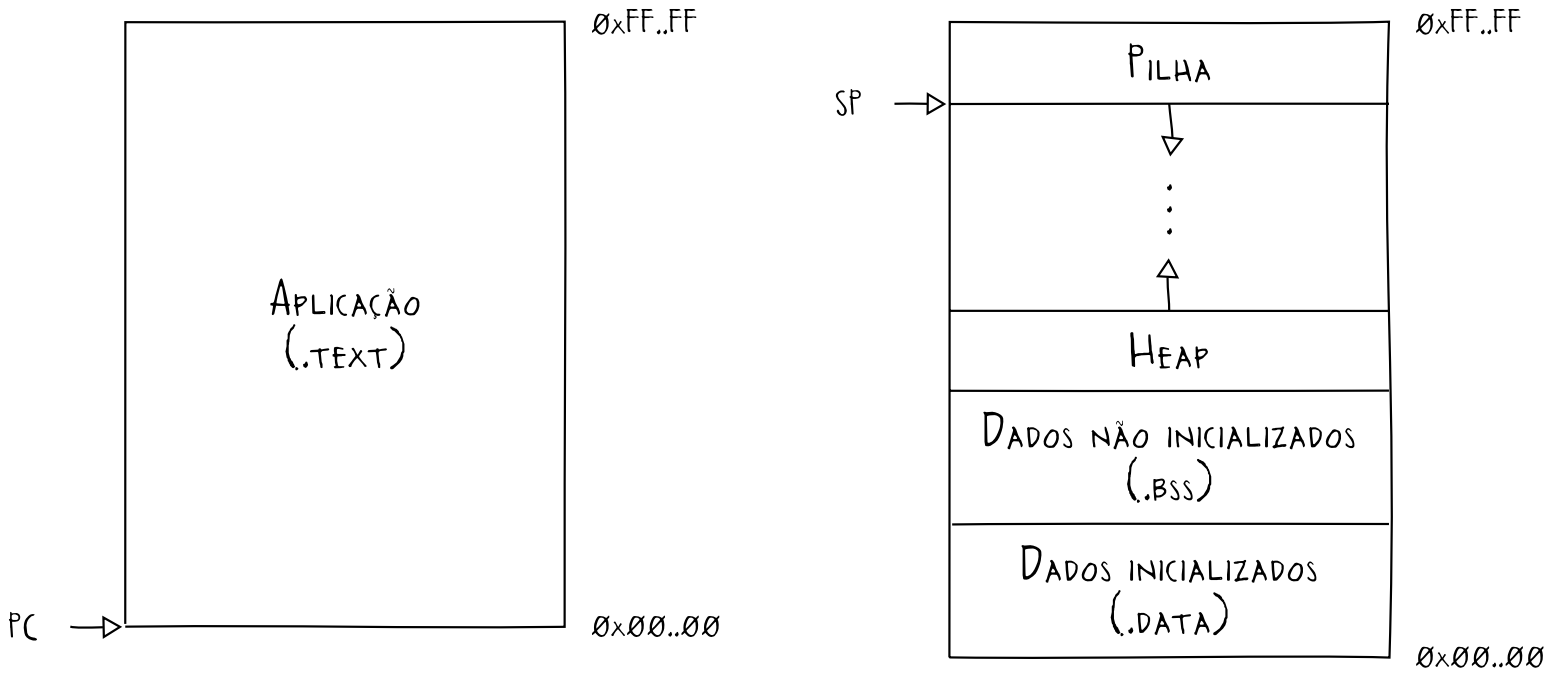
- Arquitetura Harvard
  - São utilizadas duas memórias fisicamente separadas para armazenar as instruções e os dados
  - A memória de programa só permite leitura, enquanto que a memória de dados permite escrita e leitura



✓ Acesso paralelo das instruções e dos dados

Memória

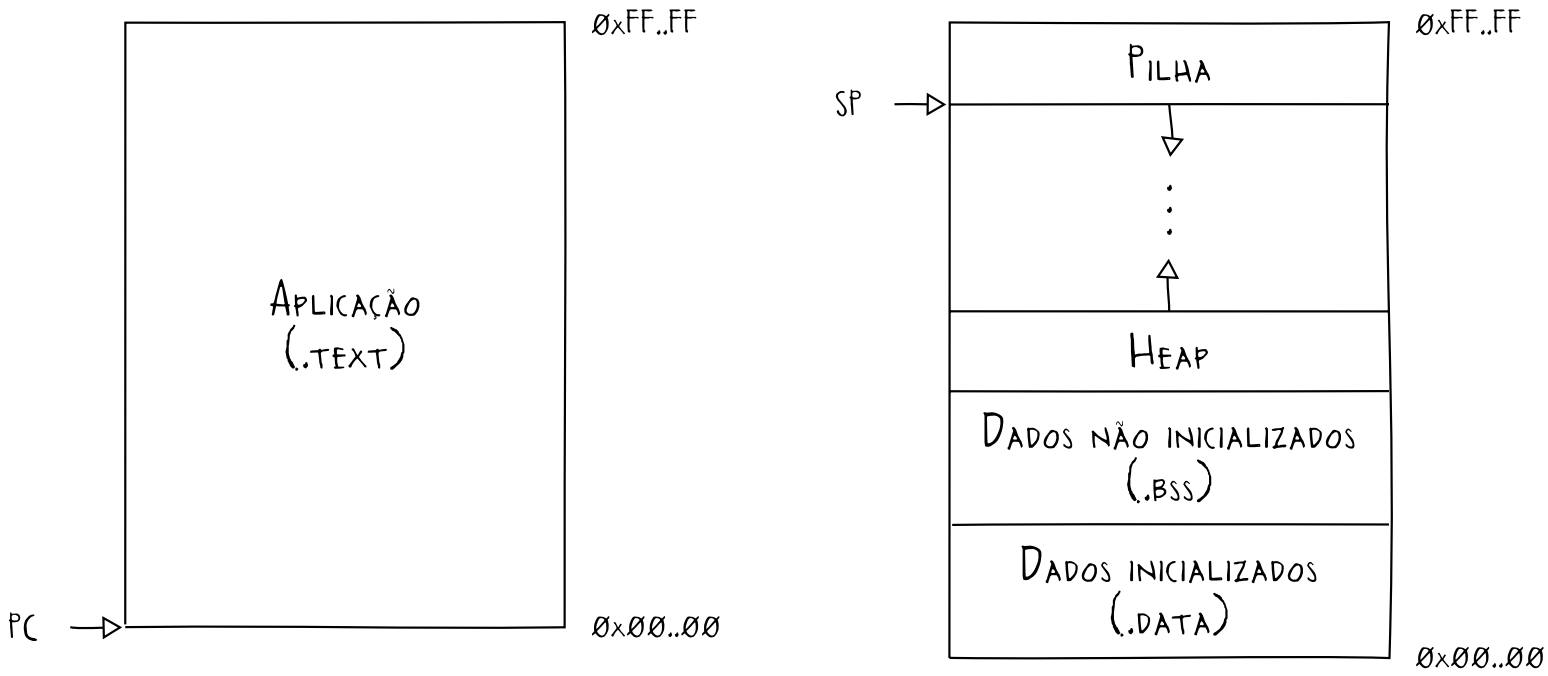
- Arquitetura Harvard
  - São utilizadas duas memórias fisicamente separadas para armazenar as instruções e os dados
  - A memória de programa só permite leitura, enquanto que a memória de dados permite escrita e leitura



✓ Proteção contra modificação da aplicação

Memória

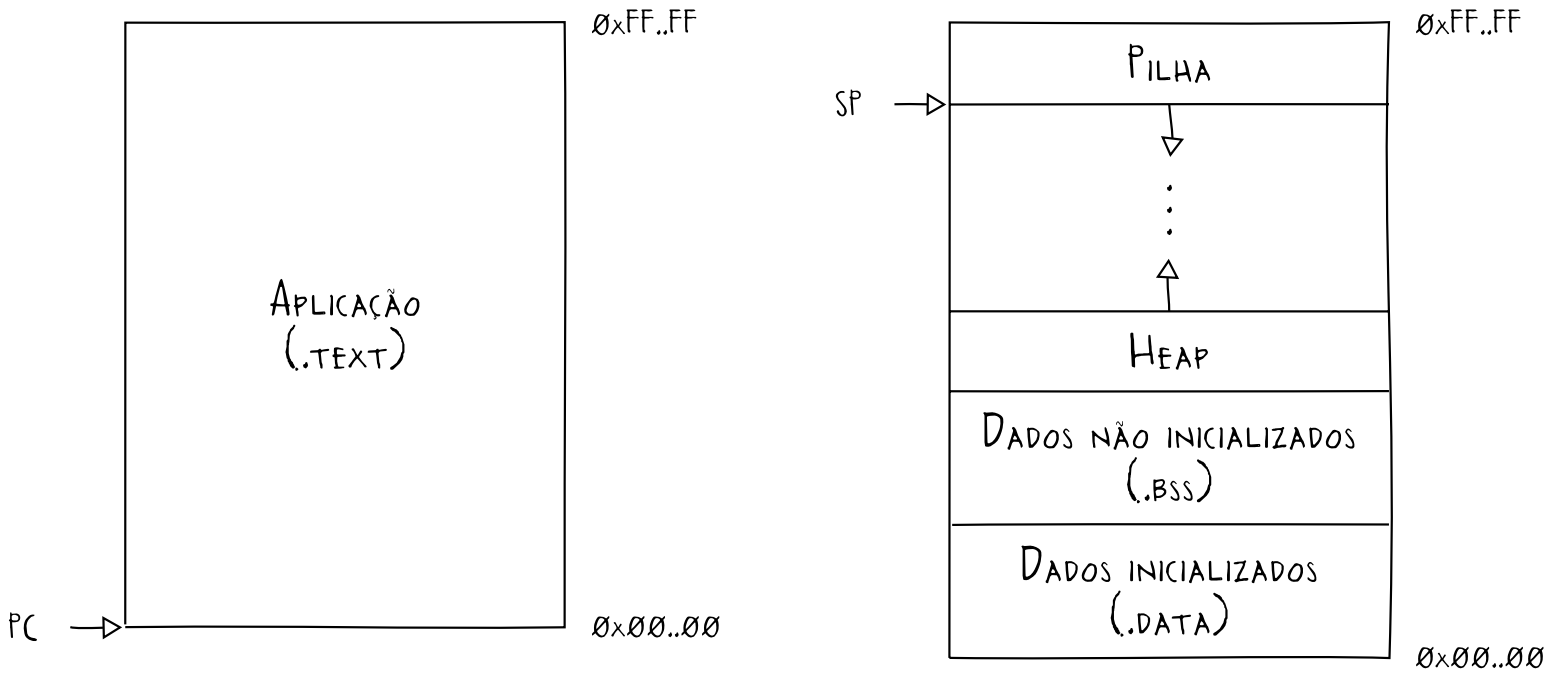
- Arquitetura Harvard
  - São utilizadas duas memórias fisicamente separadas para armazenar as instruções e os dados
  - A memória de programa só permite leitura, enquanto que a memória de dados permite escrita e leitura



❌ Componente adicional de memória

Memória

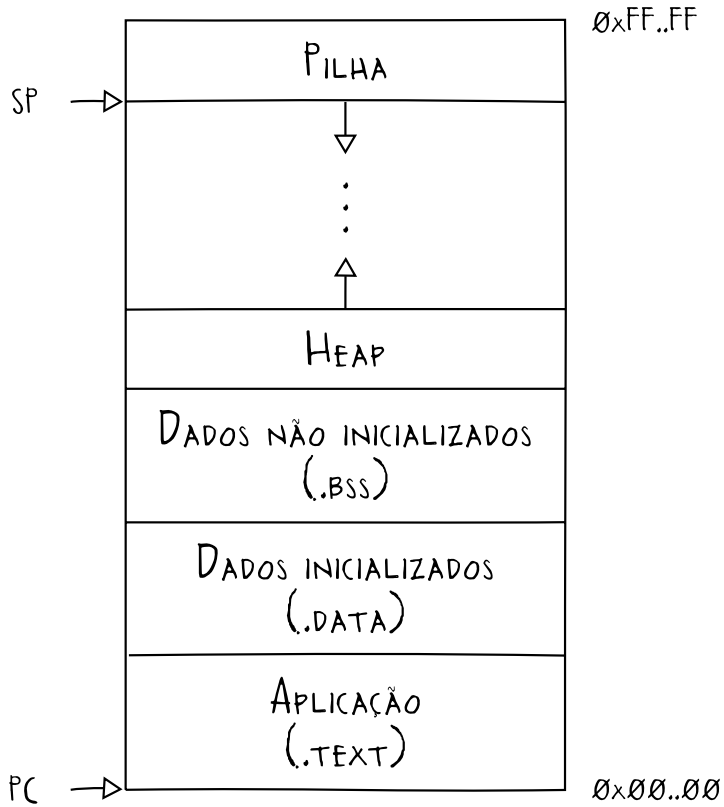
- Arquitetura Harvard
  - São utilizadas duas memórias fisicamente separadas para armazenar as instruções e os dados
  - A memória de programa só permite leitura, enquanto que a memória de dados permite escrita e leitura



✗ Pior aproveitamento da capacidade

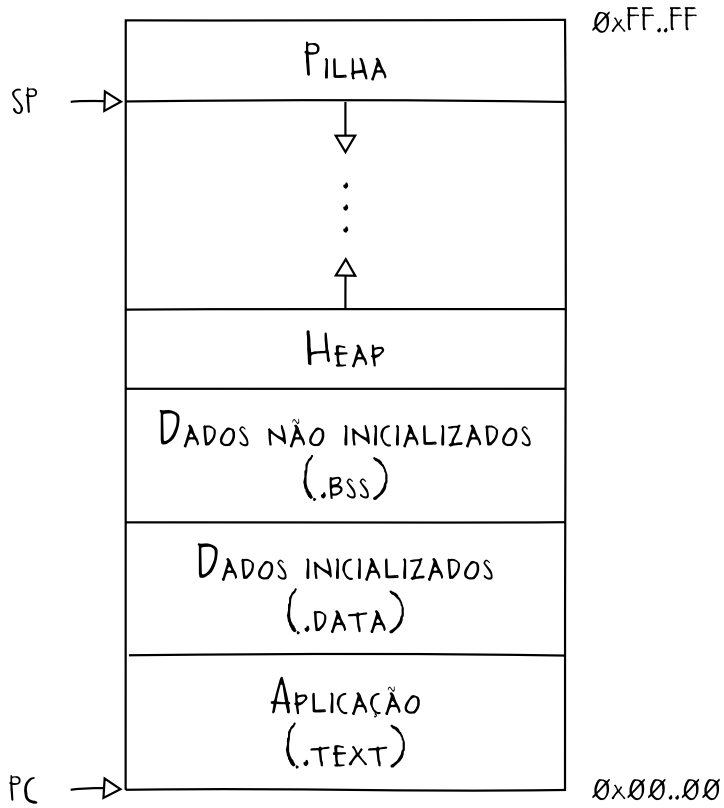
Memória

- 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

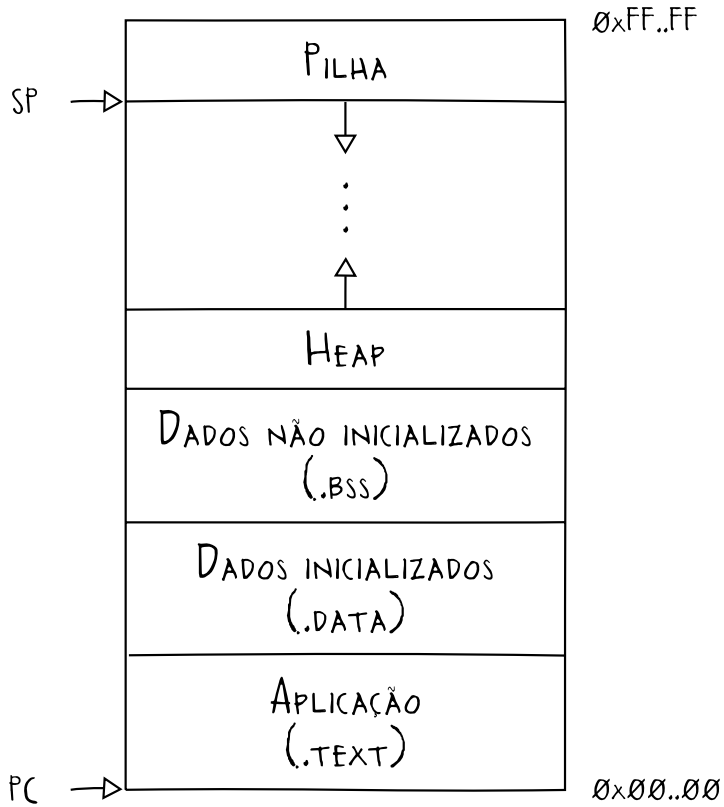
- Arquitetura Von Neumann (Princeton)
  - É utilizada somente uma memória compartilhada para armazenar as instruções e os dados
  - Tanto o código de programa como os dados podem ser acessados para escrita e leitura



✓ Capacidade de automodificação da aplicação

Memória

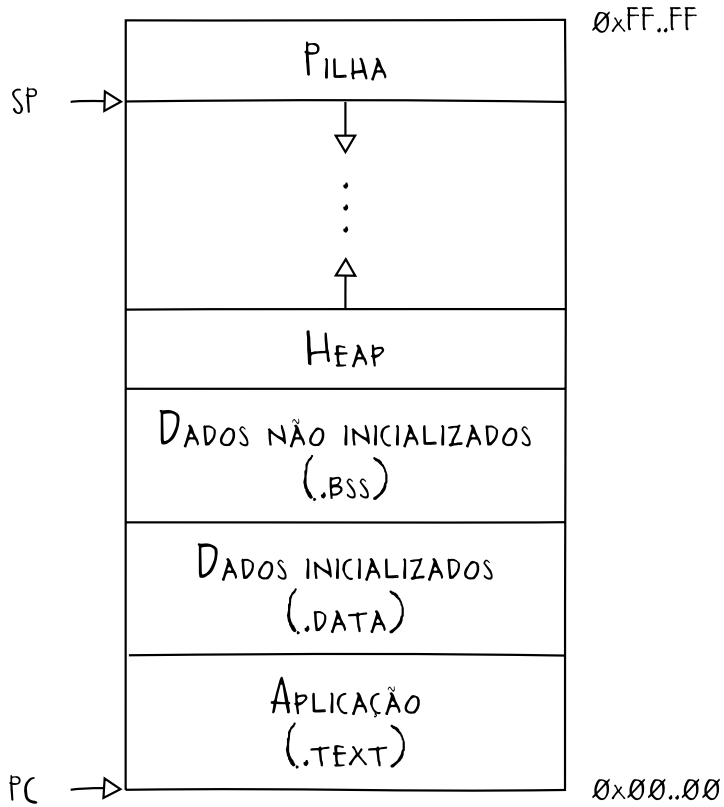
- Arquitetura Von Neumann (Princeton)
  - É utilizada somente uma memória compartilhada para armazenar as instruções e os dados
  - Tanto o código de programa como os dados podem ser acessados para escrita e leitura



✓ Uso eficiente da memória

Memória

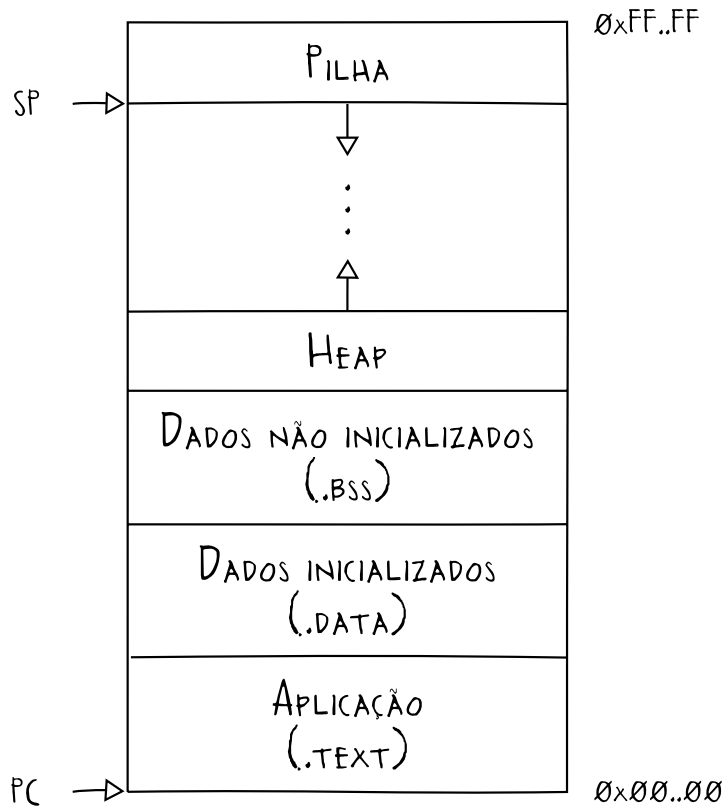
- Arquitetura Von Neumann (Princeton)
  - É utilizada somente uma memória compartilhada para armazenar as instruções e os dados
  - Tanto o código de programa como os dados podem ser acessados para escrita e leitura



✗ Acesso sequencial de instruções e dados

Memória

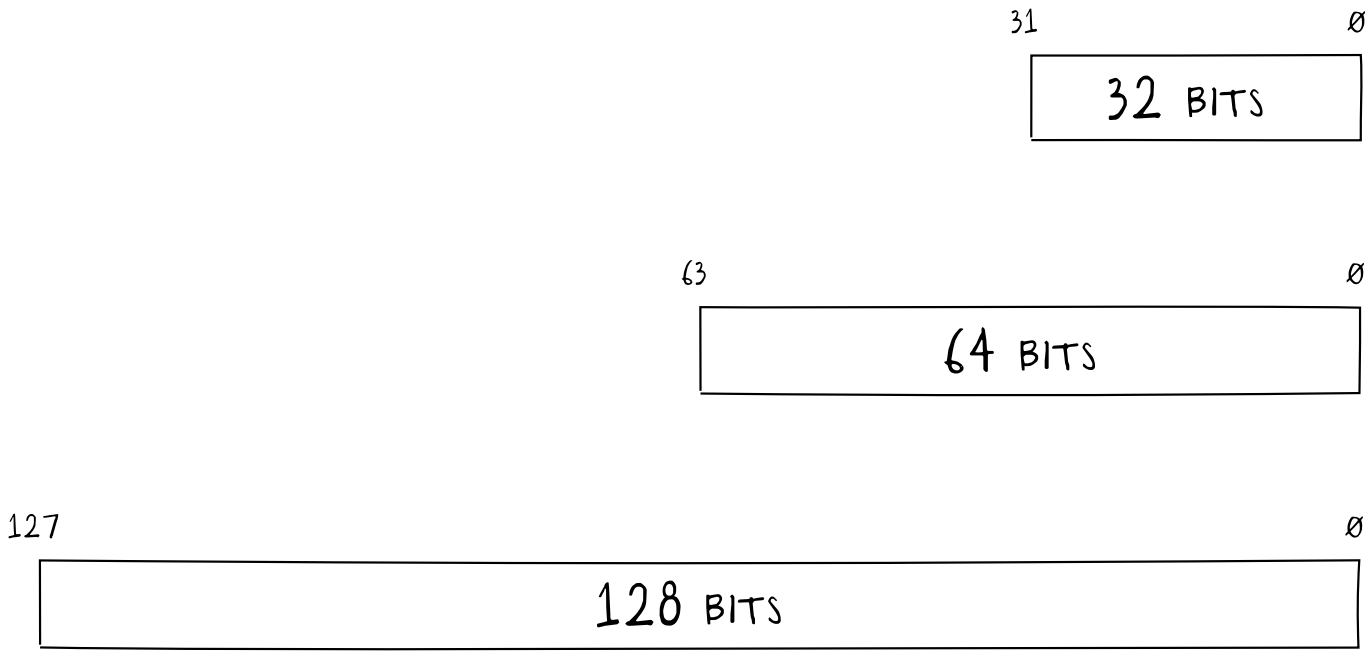
- Arquitetura Von Neumann (Princeton)
  - É utilizada somente uma memória compartilhada para armazenar as instruções e os dados
  - Tanto o código de programa como os dados podem ser acessados para escrita e leitura



✗ Mais vulnerável a ataques e falhas

Registrador

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



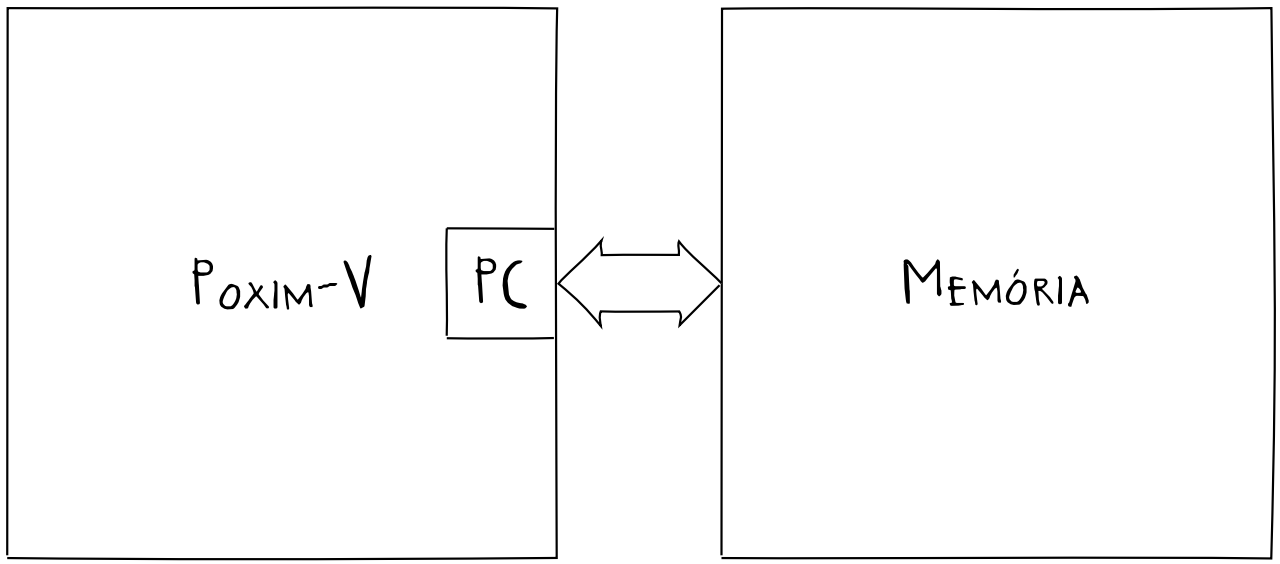
Registrador

- Propósito geral

Registrador	Rótulo	Descrição
x0	zero	Valor constante zero
x1	ra	Endereço de retorno
x2	sp	Ponteiro da pilha
x3	gp	Ponteiro global
x4	tp	Ponteiro de thread
x5	t0	Temporário/Link alternativo
x6-x7	t1-t2	Temporários
x8	s0/fp	Valor salvo/Ponteiro do quadro
x9	s1	Valor salvo
x10-x11	a0-a1	Argumentos de função/retorno
x12-x17	a2-a7	Argumentos da função
x18-x27	s2-s11	Valores salvos
x28-x31	t3-t6	Temporários

Registrador

- Processo de busca-decodificação-execução
  - A programação armazenada em memória é indexada pelo contador de programa (PC), que é um registrador específico para controlar o fluxo de execução



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

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

Registrador

- O que fazer quando não existirem registradores disponíveis para armazenar todos os dados?
  - Acessando múltiplos elementos de um vetor
  - Alocação dinâmica e estática de dados
  - Chamadas de funções aninhadas ou recursivas
  - ...

Registrador

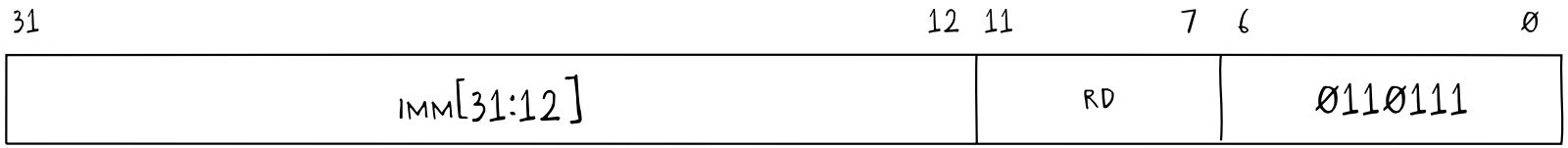
- O que fazer quando não existirem registradores disponíveis para armazenar todos os dados?
  - Acessando múltiplos elementos de um vetor
  - Alocação dinâmica e estática de dados
  - Chamadas de funções aninhadas ou recursivas
  - ...

    ✓ Realizar mais acessos à memória

    ✓ Utilizar a estrutura de pilha

Instruções de carregamento imediato

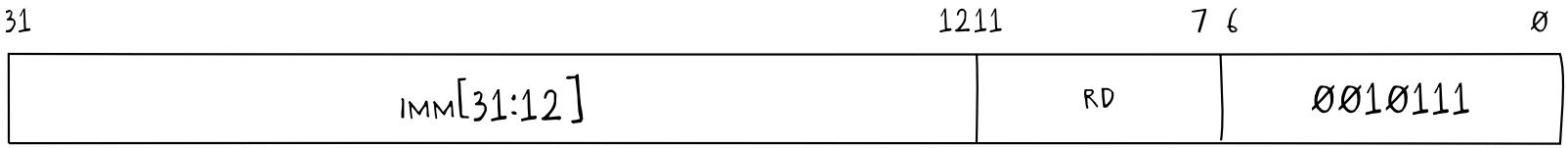
- Carregamento dos 20 bits mais significativos (*load upper immediate*)




```
lui rd, imm:
    rd = { imm, 000000000000 }
```

Instruções de carregamento imediato

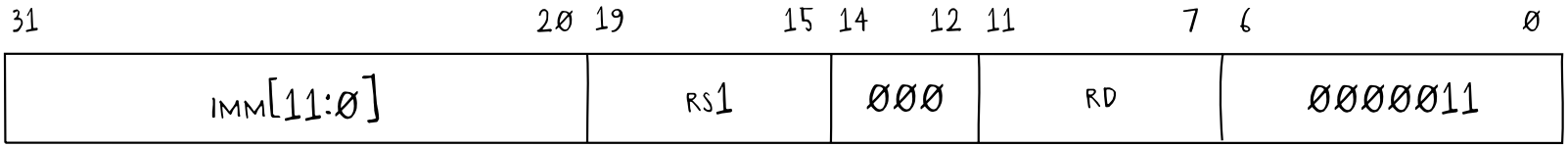
- Carregamento de endereço relativo ao PC com deslocamento de 20 bits (*add upper immediate to pc*)




```
auipc rd, imm:
    rd = pc + { imm, 000000000000 }
```

Instruções de acesso à memória

- Leitura de 8 bits com sinal (*load byte*)

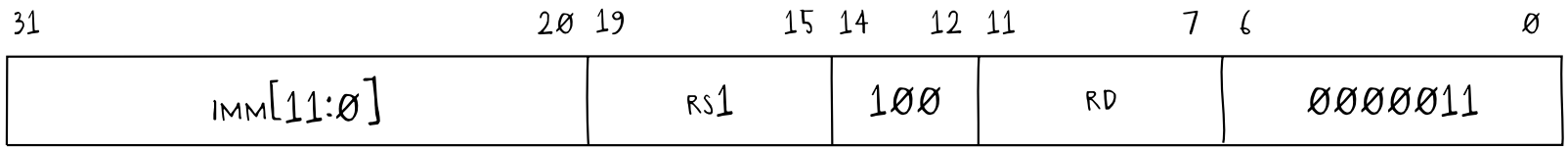



```
lb rd, imm(rs1):
    address = rs1 + sign_extension(imm)
    byte = read_8(memory, address)
    rd = sign_extension(byte)

lb rd, imm32:
    auipc rd, get_31_12(imm32)
    lb rd, get_11_0(imm32)(rd)
```

Instruções de acesso à memória

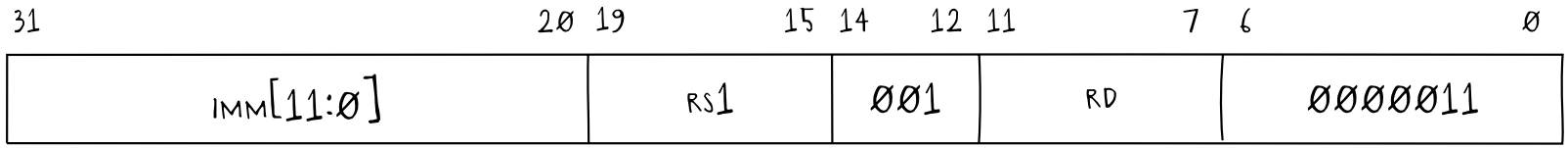
- Leitura de 8 bits sem sinal (*load byte unsigned*)




```
lbu rd, imm(rs1):
    address = rs1 + sign_extension(imm)
    byte = read_8(memory, address)
    rd = zero_extension(byte)
```

Instruções de acesso à memória

- Leitura de 16 bits com sinal (*load half*)

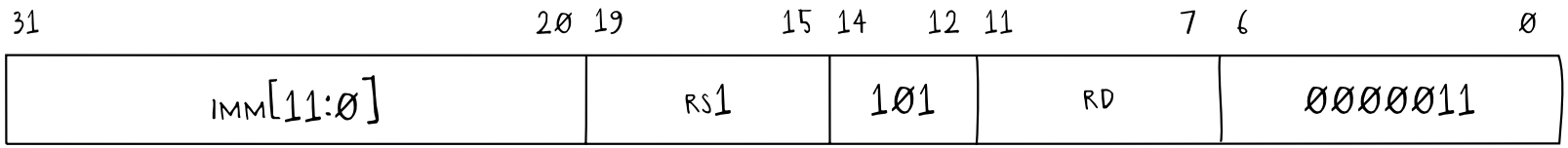



```
lh rd, imm(rs1):
    address = rs1 + sign_extension(imm)
    half = read_16(memory, address)
    rd = sign_extension(half)

lh rd, imm32:
    auipc rd, get_31_12(imm32)
    lh rd, get_11_0(imm32)(rd)
```

Instruções de acesso à memória

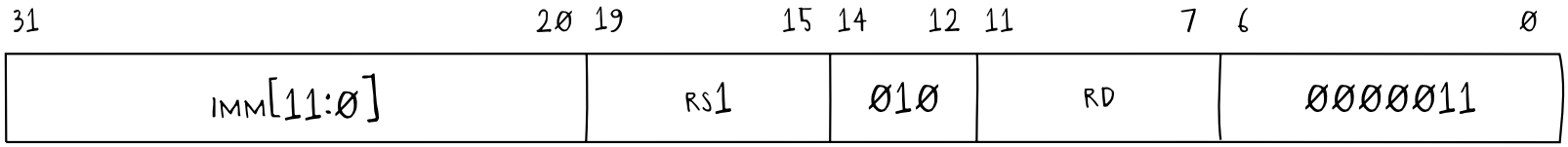
- Leitura de 16 bits sem sinal (*load half unsigned*)




```
lhu rd, imm(rs1):
    address = rs1 + sign_extension(imm)
    half = read_16(memory, address)
    rd = zero_extension(half)
```

Instruções de acesso à memória

- Leitura de 32 bits (*load word*)

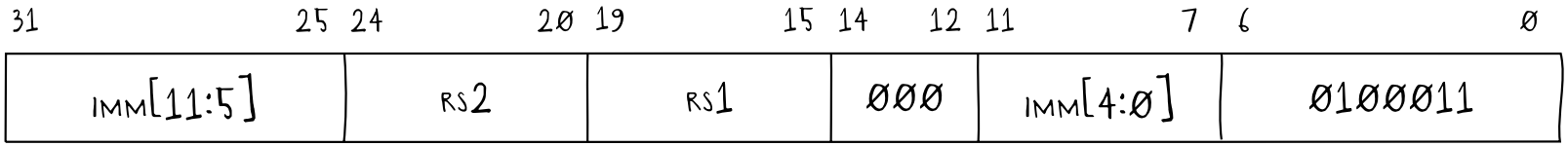



```
lw rd, imm(rs1):
    address = rs1 + sign_extension(imm)
    word = read_32(memory, address)
    rd = word

lw rd, imm32:
    auipc rd, get_31_12(imm32)
    lw rd, get_11_0(imm32)(rd)
```

Instruções de acesso à memória

- Escrita de 8 bits (*store byte*)

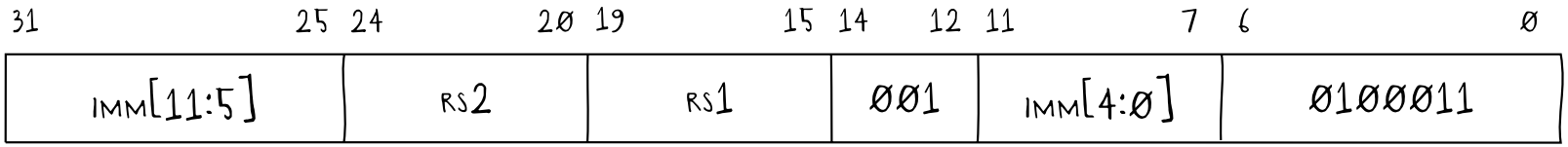



```
sb rs2, imm(rs1):
    address = rs1 + sign_extension(imm)
    data = get_7_0(rs2)
    write_8(memory, address, data)

sb rs2, imm32, rs1:
    auipc rs1, get_31_12(imm32)
    sb rs2, get_11_0(imm32)(rs1)
```

Instruções de acesso à memória

- Escrita de 16 bits (*store half*)

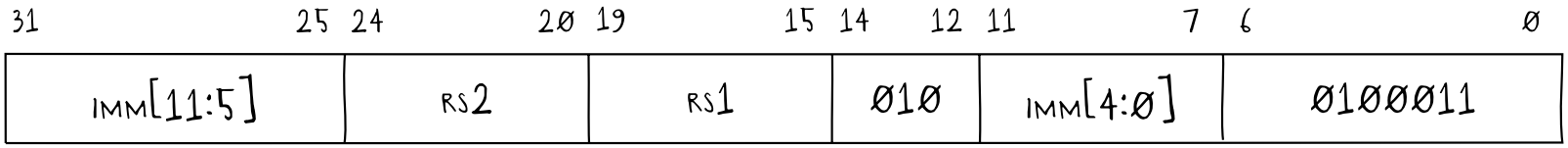



```
sh rs2, imm(rs1):
    address = rs1 + sign_extension(imm)
    data = get_15_0(rs2)
    write_16(memory, address, data)

sh rs2, imm32, rs1:
    auipc rs1, get_31_12(imm32)
    sh rs2, get_11_0(imm32)(rs1)
```

Instruções de acesso à memória

- Escrita de 32 bits (*store word*)




```
sw rs2, imm(rs1):
    address = rs1 + sign_extension(imm)
    data = rs2
    write_32(memory, address, data)

sw rs2, imm32, rs1:
    auipc rs1, get_31_12(imm32)
    sw rs2, get_11_0(imm32)(rs1)
```

Exercício

- Inicie a preparação do ambiente de desenvolvimento
  1. Escolha uma distribuição Linux
     - Subsistema do Windows para Linux (WSL)
     - Instalação nativa
     - Máquina virtual
  2. Defina qual linguagem de programação será adotada para implementação das atividades práticas, verificando a disponibilidade e os exemplos fornecidos
  3. Entenda o formato do arquivo hex, utilizando os exemplos fornecidos, que será usado para carregamento dos dados e das instruções na memória do processador