QXD-005 - Arquitetura de Computadores

# Conjunto de Instruções do Processador

Prof. Pedro Botelho

# Nas Aulas Passadas…

- Visão de Alto Nível do Computador
  - Estrutura do Computador
  - Instruções Básicas
- Sistemas de Memórias
  - Cache
  - Interna
  - Externa
- Entrada/Saída
- Arquitetura do Conjunto de Instruções
- Questão: Como funciona o componente mais importante, o **processador**?
  - Vamos usar um processador RISC de exemplo!

# Nesta Aula…

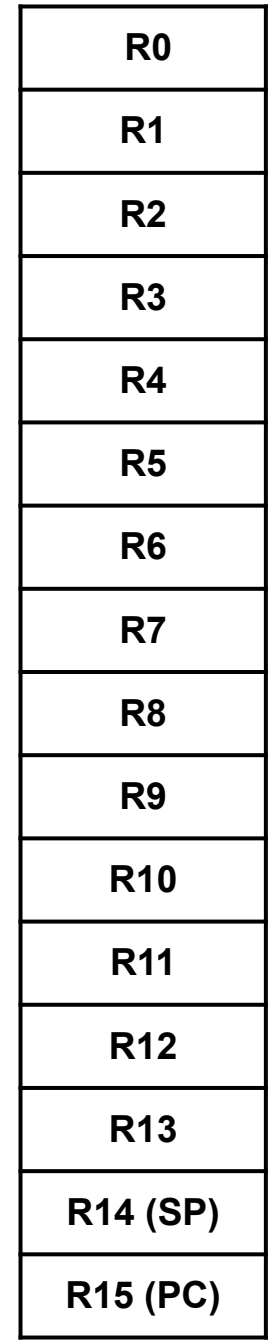
- Estrutura Básica do Processador RISC-6
- Instruções Básicas do RISC-6
- Representando Instruções Simples
- Acesso à Memória
- Controle de Fluxo
- Procedimentos

# Conjunto de Instruções do Processador

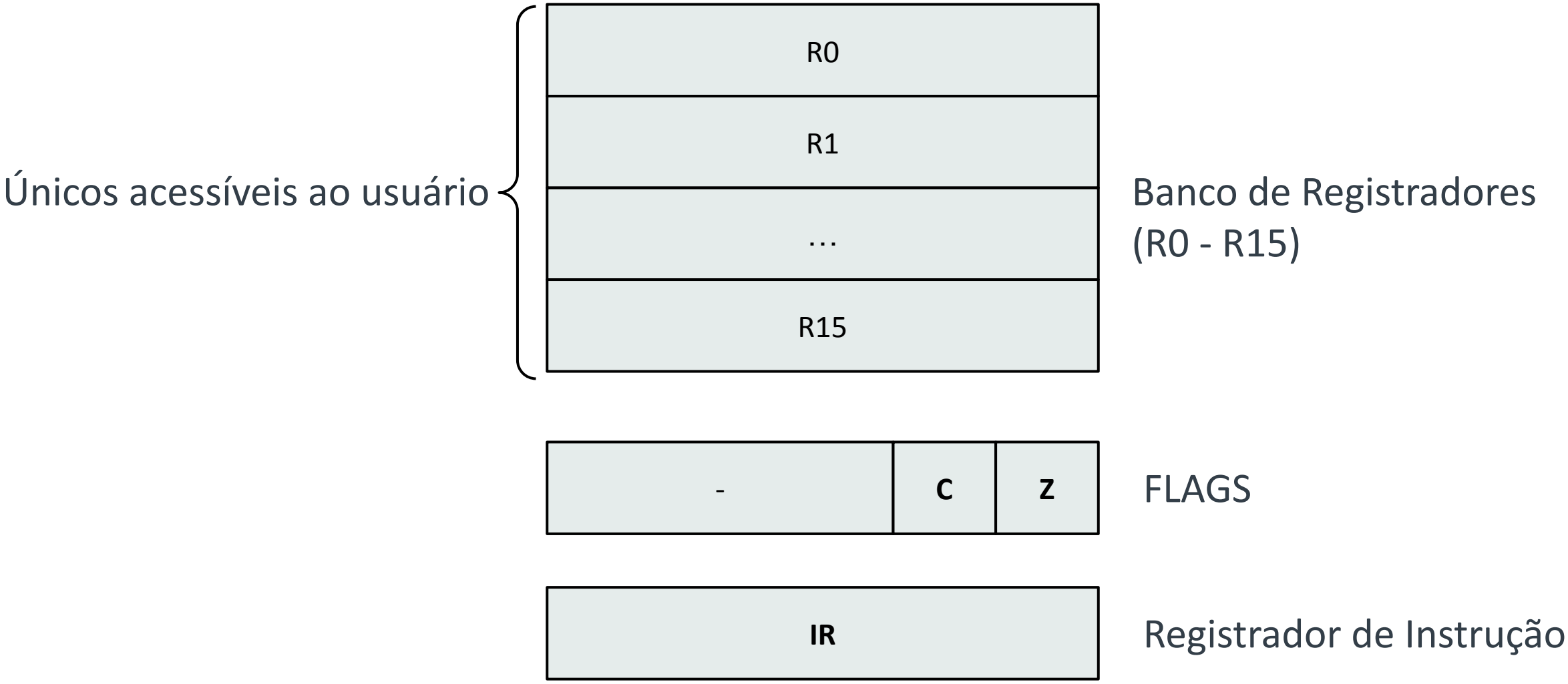
## Estrutura Básica do Processador RISC-6

# Nosso Objeto de Estudo

- Voltando ao assunto… Vamos estudar um ISA especial → Chamaremos de **RISC-6** :)
- Vamos definir a **Arquitetura do Conjunto de Instruções** (Instruções, Registradores, Memória):
  - Define a interface de programação do processador → O que está acessível ao usuário
  - Tem 16 registradores de trabalho (R0 ao R15) → Banco de Registradores
    - **R14** → Ponteiro de Pilha (SP)
    - **R15** → Contador de Programa (PC)
  - Tem dois registradores adicionais → **FLAGS** e Registrador de Instruções (**IR**)
  - Tem 20 instruções, divididas em 10 formatos diferentes
- Vamos considerar…
  - …uma **memória principal** de 16KB (cada posição tendo 2 bytes)
  - …um **módulo de E/S** para comunicação serial com quatro registradores especiais



# Registradores do RISC-6



# Formato das Instruções

- O ISA possui 10 formatos de instruções → Cada um referencia uma ou mais instruções
  - Cada instrução possui diferentes **campos** →Deve ser decodificada corretamente
  - Perceba que os endereços de **destino** e **fonte** e **opcode** estão sempre no mesmo lugar!

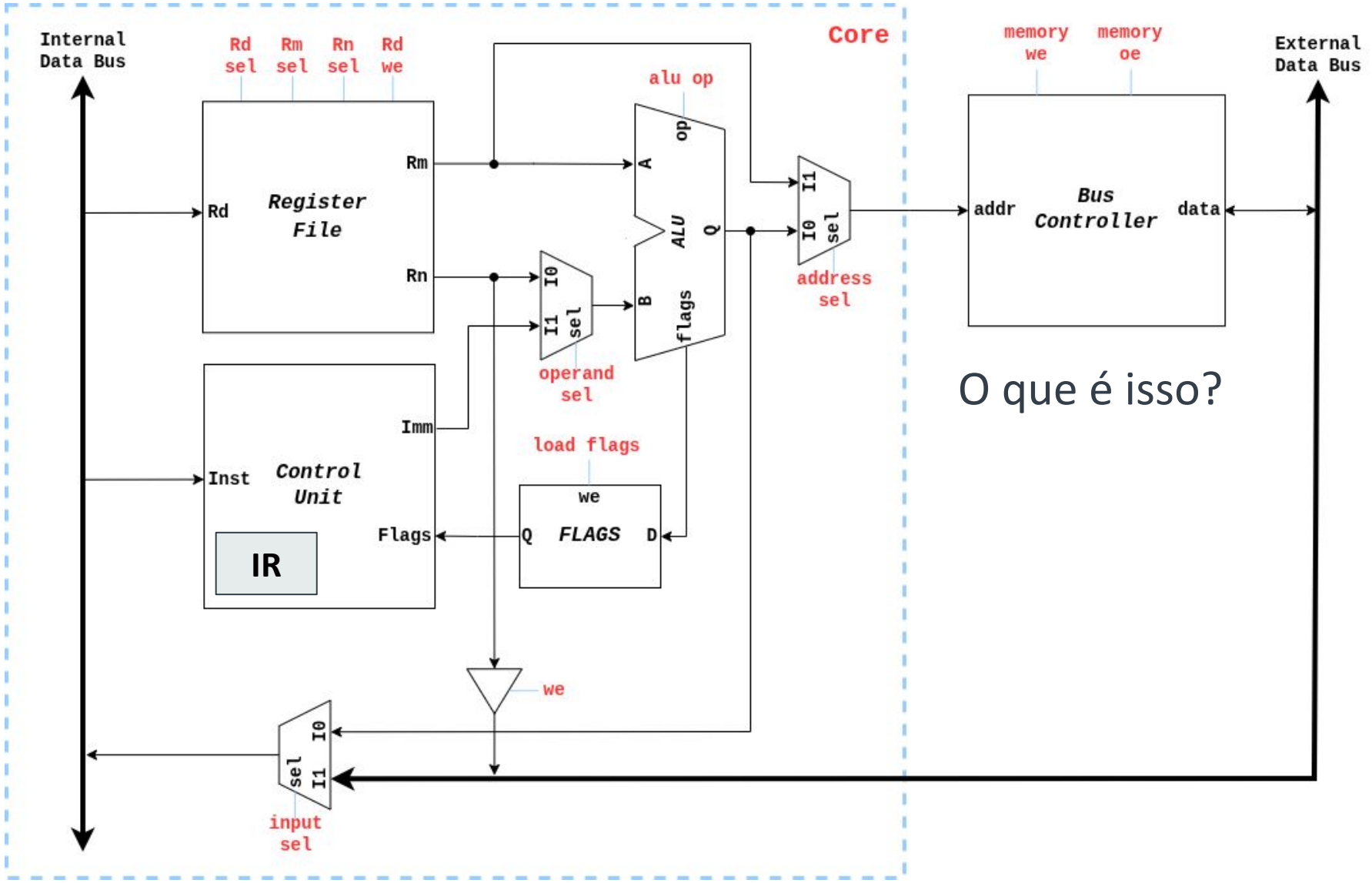
| **Instruções** | **15 14 13 12** | **11 10 9 8** | **7 6 5 4** | **3 2 1 0** |
|---|---|---|---|---|
| JMP | Imediato | | | *Opcode* |
| J<cond> | Cond. | Imediato | | *Opcode* |
| ADDI, SUBI, LDR, SHR, SHL | Rd | Rm | Imediato | *Opcode* |
| STR | Imediato | Rm | Rn | *Opcode* |
| MOV | Rd | Imediato | | *Opcode* |
| ADD, SUB, AND, OR | Rd | Rm | Rn | *Opcode* |
| CMP | - | Rm | Rn | *Opcode* |
| PUSH | - | | Rn | *Opcode* |
| POP | Rd | - | | *Opcode* |
| HALT | 0xFFFF | | | |

# Conjunto de Instruções

- O ISA possui 20 instruções → Divididas em JUMP, LOAD, STORE, MOVE, ALU, STACK e CONTROL
  - **Unidade de Controle** lê a instrução e emite sinais de controle para os outros componentes

| Instrução | Descrição | Tipo | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| JMP #Im | PC = PC + #Im | JUMP | $Im_{11}$ | $Im_{10}$ | $Im_9$ | $Im_8$ | $Im_7$ | $Im_6$ | $Im_5$ | $Im_4$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 0 | 0 | 0 | 0 |
| JEQ #Im | If ¬Z, then PC = PC + #Im | JUMP | 0 | 0 | $Im_9$ | $Im_8$ | $Im_7$ | $Im_6$ | $Im_5$ | $Im_4$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 0 | 0 | 0 | 1 |
| JNE #Im | If Z, then PC = PC + #Im | JUMP | 0 | 1 | $Im_9$ | $Im_8$ | $Im_7$ | $Im_6$ | $Im_5$ | $Im_4$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 0 | 0 | 0 | 1 |
| JLT #Im | If ¬Z and C, then PC = PC + #Im | JUMP | 1 | 0 | $Im_9$ | $Im_8$ | $Im_7$ | $Im_6$ | $Im_5$ | $Im_4$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 0 | 0 | 0 | 1 |
| JGE #Im | If Z or ¬C, then PC = PC + #Im | JUMP | 1 | 1 | $Im_9$ | $Im_8$ | $Im_7$ | $Im_6$ | $Im_5$ | $Im_4$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 0 | 0 | 0 | 1 |
| LDR Rd, [Rm, #Im] | Rd = MEM[Rm + #Im] | LOAD | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 0 | 0 | 1 | 0 |
| STR Rn, [Rm, #Im] | MEM[Rm + #Im] = Rn | STORE | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Rn_3$ | $Rn_2$ | $Rn_1$ | $Rn_0$ | 0 | 0 | 1 | 1 |
| MOV Rd, #Im | Rd = #Im | MOVE | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | $Im_7$ | $Im_6$ | $Im_5$ | $Im_4$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 0 | 1 | 0 | 0 |
| ADD Rd, Rm, Rn | Rd = Rm + Rn | ALU | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Rn_3$ | $Rn_2$ | $Rn_1$ | $Rn_0$ | 0 | 1 | 0 | 1 |
| ADDI Rd, Rm, #Im | Rd = Rm + #Im | ALU | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 0 | 1 | 1 | 0 |
| SUB Rd, Rm, Rn | Rd = Rm - Rn | ALU | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Rn_3$ | $Rn_2$ | $Rn_1$ | $Rn_0$ | 0 | 1 | 1 | 1 |
| SUBI Rd, Rm, #Im | Rd = Rm - #Im | ALU | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 1 | 0 | 0 | 0 |
| AND Rd, Rm, Rn | Rd = Rm AND Rn | ALU | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Rn_3$ | $Rn_2$ | $Rn_1$ | $Rn_0$ | 1 | 0 | 0 | 1 |
| OR Rd, Rm, Rn | Rd = Rm OR Rn | ALU | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Rn_3$ | $Rn_2$ | $Rn_1$ | $Rn_0$ | 1 | 0 | 1 | 0 |
| SHR Rd, Rm, #Im | Rd = Rm >> #Im | ALU | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 1 | 0 | 1 | 1 |
| SHL Rd, Rm, #Im | Rd = Rm << #Im | ALU | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 1 | 1 | 0 | 0 |
| CMP Rm, Rn | Z = (Rm = Rn); C = (Rm < Rn) | ALU | - | - | - | - | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Rn_3$ | $Rn_2$ | $Rn_1$ | $Rn_0$ | 1 | 1 | 0 | 1 |
| PUSH Rn | SP- -; MEM[SP] = Rn | STACK | - | - | - | - | - | - | - | - | $Rn_3$ | $Rn_2$ | $Rn_1$ | $Rn_0$ | 1 | 1 | 1 | 0 |
| POP Rd | Rd = MEM[SP]; SP++ | STACK | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | - | - | - | - | - | - | - | - | 1 | 1 | 1 | 1 |
| HALT | Para a execução | CONTROL | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

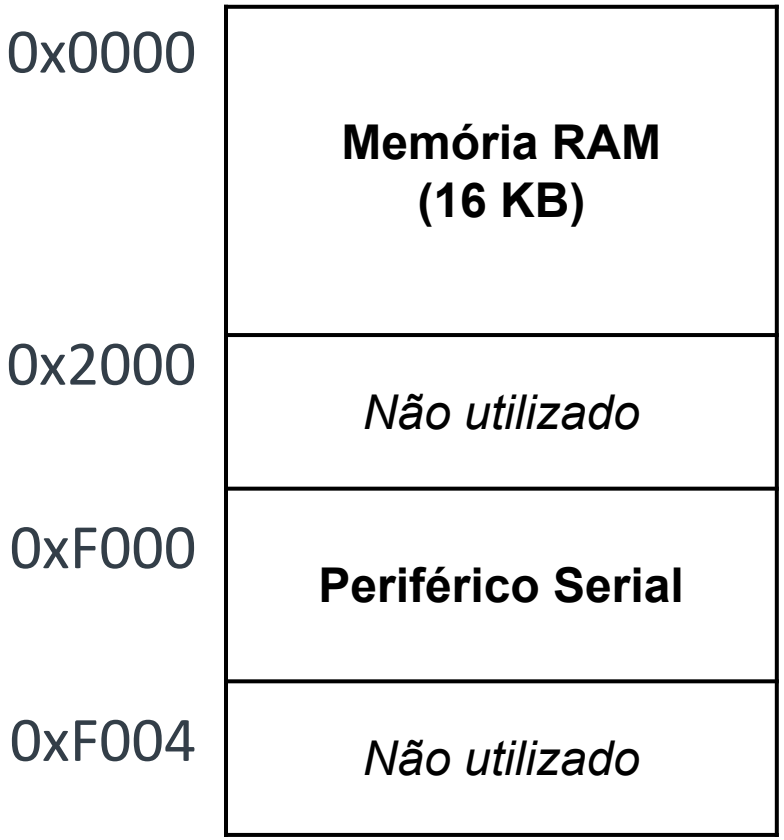
# Estrutura do Núcleo




O que é isso?

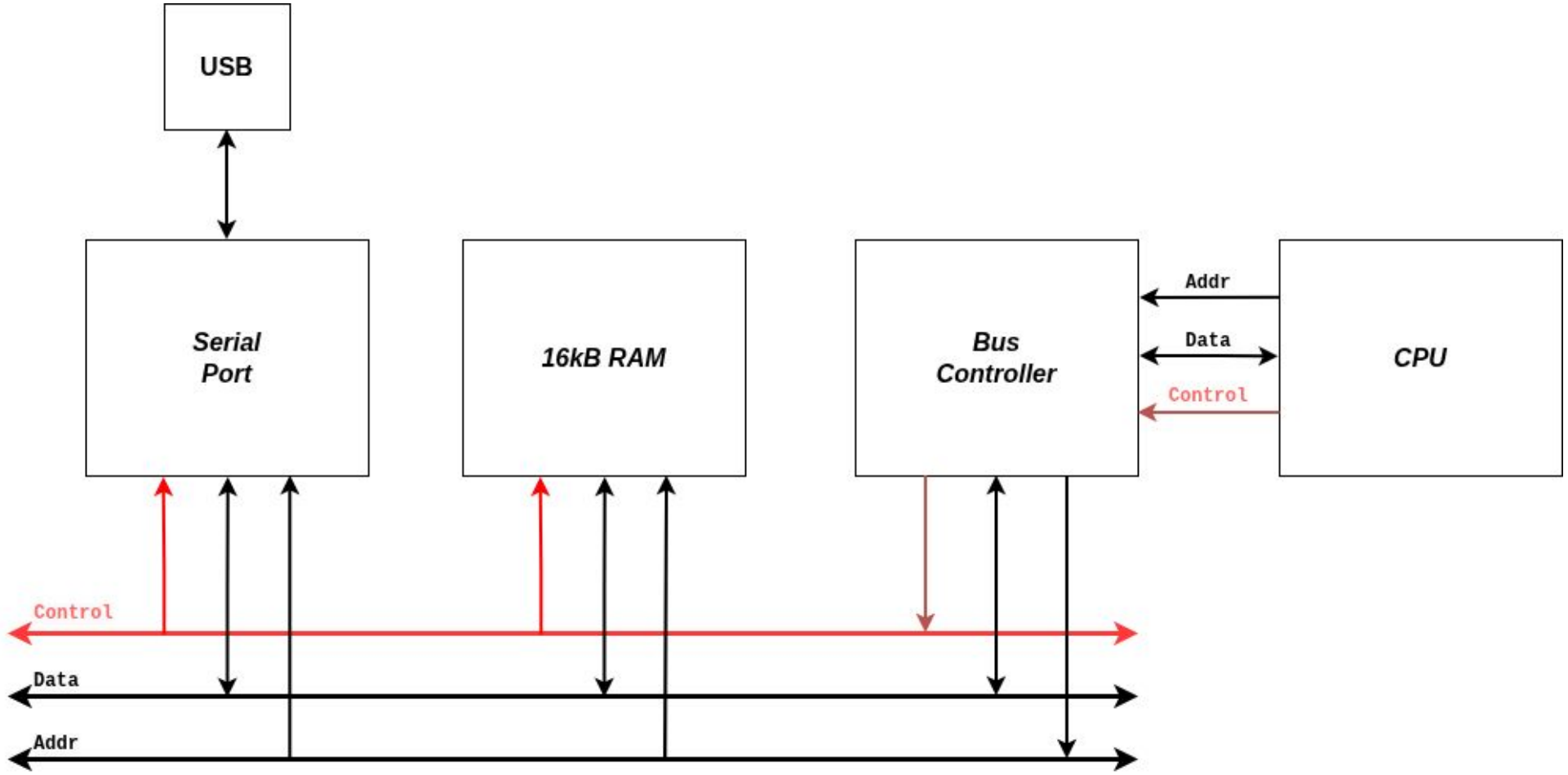
# Mapa de Memória do RISC-6

- O RISC-6 usa **MMIO** (Memory Mapped I/O) → Portas de E/S acessadas via endereços de memória
- **Controlador de Barramento** verifica o endereço e seleciona E/S ou memória:
  - Endereço **0x0200** → Mapeado para a memória RAM
  - Endereço **0xF001** → Mapeado para o periférico
- Um único periférico → Periférico Serial
  - Comunicação simplificada com o usuário
- Quatro registradores de E/S:
  - 0xF000 → **CHAR_IN**
  - 0xF001 → **CHAR_OUT**
  - 0xF002 → **INT_IN**
  - 0xF003 → **INT_OUT**
- Ler ou escrever dos registradores → Interação básica



# Estrutura do Sistema

- Processador se comunica com o restante do sistema pelo **controlador de barramento**
- Este se conecta ao barramento sistema, junto à todos os outros componentes



# Conjunto de Instruções do Processador

## Instruções Básicas do RISC-6

# Instruções Aritméticas

- Todas as instruções aritméticas (ADD, SUB) recebem três operandos: **Duas fontes** e **um destino**

```
ADD     R6, R1, R2   // R6 = g + h
ADD     R7, R3, R4   // R7 = i + j
SUB     R5, R6, R7   // R5 = (g + h) - (i + j)
```

- **1º Princípio de Projeto**: "*Simplicidade favorece a regularidade*"
  - Regularidade simplifica a implementação: **Instruções padronizadas**
  - Simplicidade permite um desempenho superior a um custo menor: **Implementação barata**

# Operandos em Registradores

- Instruções aritméticas usam **registradores** como operandos ⟶ ULA opera em registradores:

```
ADD   R6, R1, R2   // R6 = g + h
ADD   R7, R3, R4   // R7 = i + j
SUB   R5, R6, R7   // R5 = (g + h) - (i + j)
```

- Lembre-se: RISC-6 tem um banco com **16 registradores** de 16 bits ⟶ Memória 16 × 16 = 32B!
  - Usados para dados frequentemente acessados: Guardam uma **palavra** de 16 bits
  - Contador de Programa (PC) é acessível via R15 ⟶ Mantém endereço da instrução atual
- **2º Princípio de Projeto**: "*Menor é mais rápido*"
  - Por exemplo, memória principal é maior mas muito mais lento

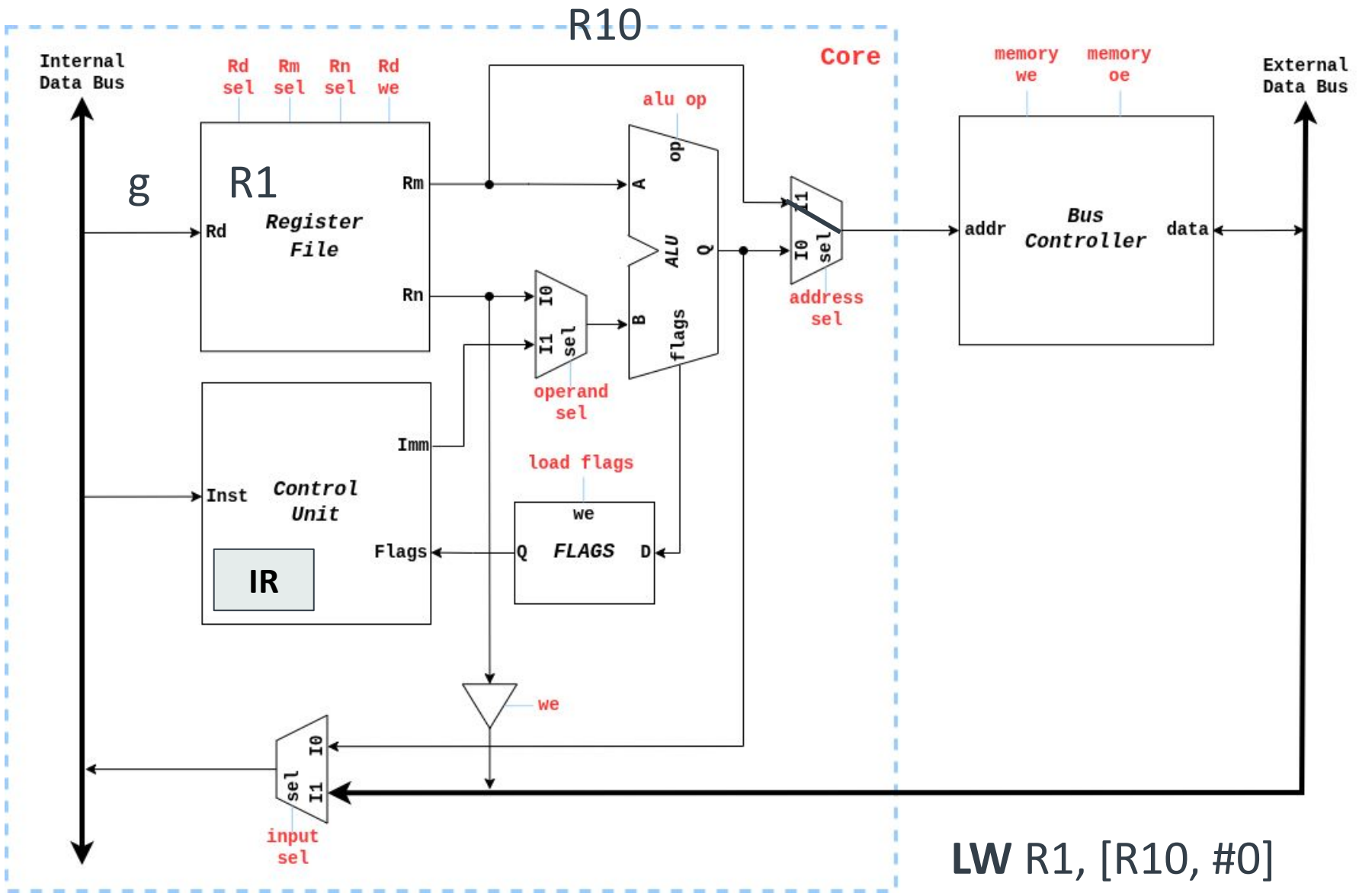
# Operandos na Memória

- RISC-6 usa instruções especiais para acessar a **memória** ⟶ Arquitetura *load*/*store*
- Para poder processar dados ⟶ Registradores necessitam de dados
  - Carregar dados nos **registradores** (*load register*, ou **LDR**)
  - Processá-los (com **ADD**)
  - Depos salvar o resultado na **memória** (*store register*, ou **STR**)
- Suponha que em **R10** esteja o endereço de **g**, **R11** o de **h** e **R12** o de **f**:
  - Usam um registrador como "ponteiro" para acessar a memória

```
LW    R1, [R10, #0]    // R1 = *(R10), R10 = &g
LW    R2, [R11, #0]    // R2 = *(R11), R11 = &h
ADD   R6, R1, R2       // R6 = g + h
SW    R6, [R12, #0]    // *(R12) = R6, R12 = &f
```

- Obs: Arquiteturas comuns compreendem a memória como se cada endereço fosse um byte!

# Operação de *Load*




**LW** R1, [R10, #0]

# Outro Exemplo de Acesso

- Suponha a seguinte **memória de dados**:
- E suponha que **R10** tem valor **0x1000**.
- RISC-6 permite aritmética de ponteiros:
  - Soma valor no registrador à um **imediato** (valor codificado na própria instrução).

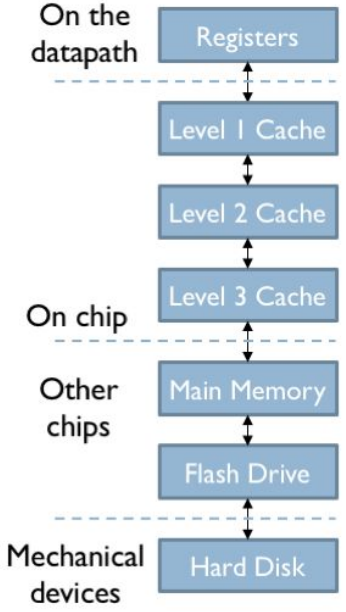
```
LDR     R1, [R10, #0]     // R1 = *(R10 + 0)
LDR     R2, [R10, #1]     // R2 = *(R10 + 1)
ADD     R6, R1, R2        // R6 = g + h
STR     R6, [R10, #2]     // *(R10 + 2) = R6
```

- Qual o valor salvo em 0x1002 ao final?

| Endereço | Conteúdo |
|---|---|
| 0x1000 | 10 |
| 0x1001 | 20 |
| 0x1002 | ? |

# Registradores v.s. Memória

- Os registradores são **mais rápidos** de acessar do que a memória
- Operar com dados na memória requer operações de *load* e *store*
  - Mais instruções precisam ser executadas
- O compilador usa **registradores para variáveis** o máximo possível
  - Apenas utilize memória para variáveis menos usadas
  - A otimização de registradores é importante!




| Access time | Capacity | Managed By |
|---|---|---|
| 1 cycle | 1 KB | Software/Compiler |
| 2-4 cycles | 32 KB | Hardware |
| 10 cycles | 256 KB | Hardware |
| 40 cycles | 10 MB | Hardware |
| 200 cycles | 10 GB | Software/OS |
| 10-100us | 100 GB | Software/OS |
| 10ms | 1 TB | Software/OS |

# Operandos Imediatos

- Dados constantes especificados diretamente na instrução ⟶ Codificados

**ADDI** R12, R12, #4 // R12 = R12 + 4

**MOV** R10, #15 // R10 = 15

- Independendo do valor em R12, este será somado a 4
- Tornar o caso comum rápido ⟶ Constantes pequenas são comuns
  - Operando imediato evita uma instrução de *load*
- RISC-6 possui várias instruções que usam valores imediatos:
  - **LDR** R2, [R5, #4] ⟶ imediato 4
  - **STR** R3, [R4, #8] ⟶ imediato 8
- Como representar imediato negativo (e.g. **MOV** R10, #-15) se o campo tem apenas 8 bits?

# Extensão de Sinal

- Representar um número usando **mais bits** preservando o **valor numérico**
  - Na instrução **MOV** R10, #-15, precisamos salvar -15 (de 8 bits) em R0 (de 16 bits)
- Replicar o bit de sinal para a esquerda
  - Para valores sem sinal: estender com **zeros**
- Exemplo: 8 bits para 16 bits
  - +15: **0**000 1111 (na instrução) ⟶ **0000 0000 0**000 1111 (no registrador)
  - –15: **1**111 0001 (na instrução) ⟶ **1111 1111 1**111 0001 (no registrador)

# Formato das Instruções

- No conjunto de instruções do RISC-6 é usado as duas formas…
  - **Extensão de zero** em imediatos de 4 bits (unsigned) → ADDI, SUBI, LDR, STR, SHL, SHR
  - **Extensão de sinal** em imediatos com mais de 4 bits (signed) → JMP, J<cond> e MOV
  - Por **padrão**, é usada a extensão de zeros → Não perder bits de representação

| **Instruções** | **15 14 13 12** | **11 10 9 8** | **7 6 5 4** | **3 2 1 0** |
|---|---|---|---|---|
| JMP | Imediato | | | *Opcode* |
| J<cond> | Cond. | Imediato | | *Opcode* |
| ADDI, SUBI, LDR, SHR, SHL | Rd | Rm | Imediato | *Opcode* |
| STR | Imediato | Rm | Rn | *Opcode* |
| MOV | Rd | Imediato | | *Opcode* |
| ADD, SUB, AND, OR | Rd | Rm | Rn | *Opcode* |
| CMP | - | Rm | Rn | *Opcode* |
| PUSH | - | | Rn | *Opcode* |
| POP | Rd | - | | *Opcode* |
| HALT | 0xFFFF | | | |

# Conjunto de Instruções do Processador

## Representando Instruções Simples

# Representando Instruções

- Lembre-se: As instruções são codificadas em **binário** → Código de máquina
- Instruções RISC-6 são codificadas como **palavras de instrução** de 16 bits
  - Pequeno número de **formatos** que codificam o opcode, endereços de registradores, …
  - Perceba a **regularidade**!

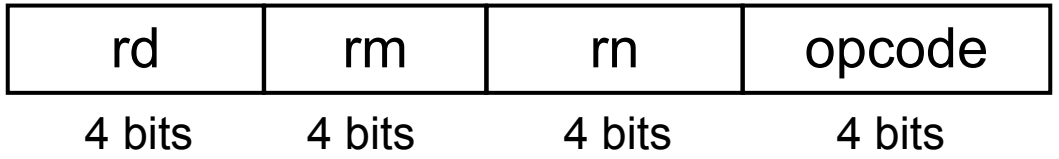
| **Instruções** | **15 14 13 12** | **11 10 9 8** | **7 6 5 4** | **3 2 1 0** |
|---|---|---|---|---|
| JMP | Imediato | | | *Opcode* |
| J<cond> | Cond. | Imediato | | *Opcode* |
| ADDI, SUBI, LDR, SHR, SHL | Rd | Rm | Imediato | *Opcode* |
| STR | Imediato | Rm | Rn | *Opcode* |
| MOV | Rd | Imediato | | *Opcode* |
| ADD, SUB, AND, OR | Rd | Rm | Rn | *Opcode* |
| CMP | - | Rm | Rn | *Opcode* |
| PUSH | - | | Rn | *Opcode* |
| POP | Rd | - | | *Opcode* |
| HALT | 0xFFFF | | | |

# Código de Operação das Instruções

- Para codificar as instruções ⟶ Devemos saber seu **opcode**!
  - Será utilizado pela unidade de controla para descobrir qual a instrução (e seu formato)

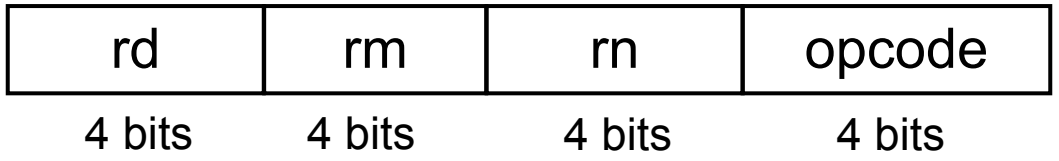
| Instrução | Descrição | Tipo | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| JMP #Im | PC = PC + #Im | JUMP | $Im_{11}$ | $Im_{10}$ | $Im_9$ | $Im_8$ | $Im_7$ | $Im_6$ | $Im_5$ | $Im_4$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 0 | 0 | 0 | 0 |
| JEQ #Im | If ¬Z, then PC = PC + #Im | JUMP | 0 | 0 | $Im_9$ | $Im_8$ | $Im_7$ | $Im_6$ | $Im_5$ | $Im_4$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 0 | 0 | 0 | 1 |
| JNE #Im | If Z, then PC = PC + #Im | JUMP | 0 | 1 | $Im_9$ | $Im_8$ | $Im_7$ | $Im_6$ | $Im_5$ | $Im_4$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 0 | 0 | 0 | 1 |
| JLT #Im | If ¬Z and C, then PC = PC + #Im | JUMP | 1 | 0 | $Im_9$ | $Im_8$ | $Im_7$ | $Im_6$ | $Im_5$ | $Im_4$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 0 | 0 | 0 | 1 |
| JGE #Im | If Z or ¬C, then PC = PC + #Im | JUMP | 1 | 1 | $Im_9$ | $Im_8$ | $Im_7$ | $Im_6$ | $Im_5$ | $Im_4$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 0 | 0 | 0 | 1 |
| LDR Rd, [Rm, #Im] | Rd = MEM[Rm + #Im] | LOAD | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 0 | 0 | 1 | 0 |
| STR Rn, [Rm, #Im] | MEM[Rm + #Im] = Rn | STORE | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Rn_3$ | $Rn_2$ | $Rn_1$ | $Rn_0$ | 0 | 0 | 1 | 1 |
| MOV Rd, #Im | Rd = #Im | MOVE | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | $Im_7$ | $Im_6$ | $Im_5$ | $Im_4$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 0 | 1 | 0 | 0 |
| ADD Rd, Rm, Rn | Rd = Rm + Rn | ALU | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Rn_3$ | $Rn_2$ | $Rn_1$ | $Rn_0$ | 0 | 1 | 0 | 1 |
| ADDI Rd, Rm, #Im | Rd = Rm + #Im | ALU | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 0 | 1 | 1 | 0 |
| SUB Rd, Rm, Rn | Rd = Rm - Rn | ALU | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Rn_3$ | $Rn_2$ | $Rn_1$ | $Rn_0$ | 0 | 1 | 1 | 1 |
| SUBI Rd, Rm, #Im | Rd = Rm - #Im | ALU | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 1 | 0 | 0 | 0 |
| AND Rd, Rm, Rn | Rd = Rm AND Rn | ALU | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Rn_3$ | $Rn_2$ | $Rn_1$ | $Rn_0$ | 1 | 0 | 0 | 1 |
| OR Rd, Rm, Rn | Rd = Rm OR Rn | ALU | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Rn_3$ | $Rn_2$ | $Rn_1$ | $Rn_0$ | 1 | 0 | 1 | 0 |
| SHR Rd, Rm, #Im | Rd = Rm >> #Im | ALU | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 1 | 0 | 1 | 1 |
| SHL Rd, Rm, #Im | Rd = Rm << #Im | ALU | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Im_3$ | $Im_2$ | $Im_1$ | $Im_0$ | 1 | 1 | 0 | 0 |
| CMP Rm, Rn | Z = (Rm = Rn); C = (Rm < Rn) | ALU | - | - | - | - | $Rm_3$ | $Rm_2$ | $Rm_1$ | $Rm_0$ | $Rn_3$ | $Rn_2$ | $Rn_1$ | $Rn_0$ | 1 | 1 | 0 | 1 |
| PUSH Rn | SP- -; MEM[SP] = Rn | STACK | - | - | - | - | - | - | - | - | $Rn_3$ | $Rn_2$ | $Rn_1$ | $Rn_0$ | 1 | 1 | 1 | 0 |
| POP Rd | Rd = MEM[SP]; SP++ | STACK | $Rd_3$ | $Rd_2$ | $Rd_1$ | $Rd_0$ | - | - | - | - | - | - | - | - | 1 | 1 | 1 | 1 |
| HALT | Para a execução | CONTROL | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

# Instruções da ULA que usam Registradores

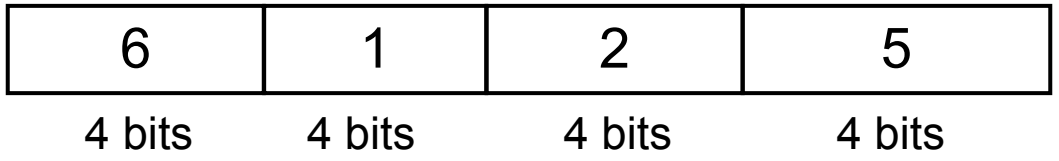



- Campos da instrução:
  - **opcode**: código de operação
  - **rd**: número do registrador de destino
  - **rm**: número do primeiro registrador de origem
  - **rn**: número do segundo registrador de origem
- Muitas instruções serão semelhantes à essa → **Regularidade**!
  - ADD, SUB, AND, OR

# Exemplo de Instrução da ULA que usa Registradores



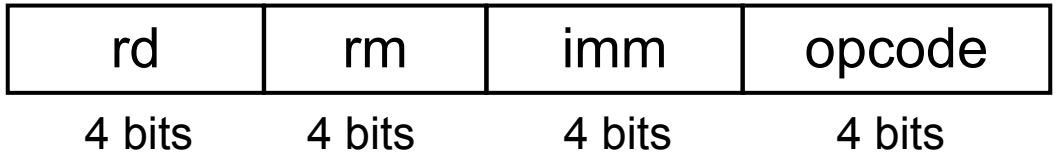

**ADD** R6, R1, R2




$$\mathbf{0110\ 0001\ 0010\ 0101_2 = 6125_{16}}$$

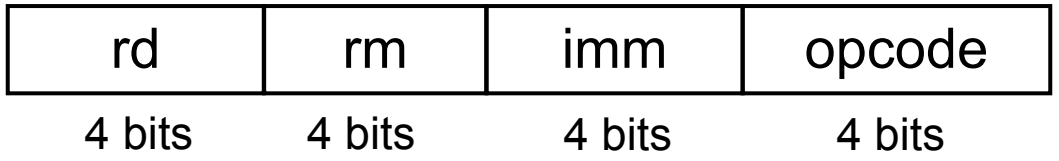
OBS: Nem sempre a tradução será direto ao ponto,
esse ISA foi feito para facilitar isso.

# Instruções da ULA que usam Imediato

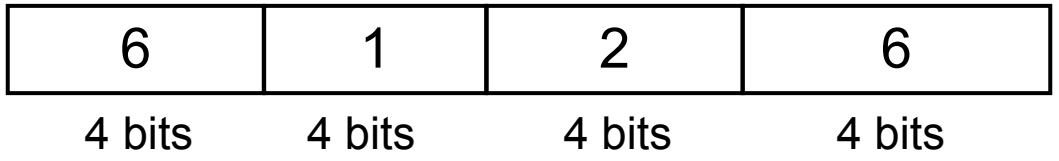



- Algumas instruções aritméticas usam imediatos:
  - ADDI, SUBI, SHL, SHR
- Campos da instrução:
  - **opcode**: código de operação
  - **rd**: número do registrador de destino
  - **rm**: número do primeiro registrador de origem
  - **imm**: operando constante (extensão de zeros)
- Muitas instruções serão semelhantes à essa → **Regularidade**!

# Exemplo de Instrução da ULA que usa Imediato




**ADDI** R6, R1, #2

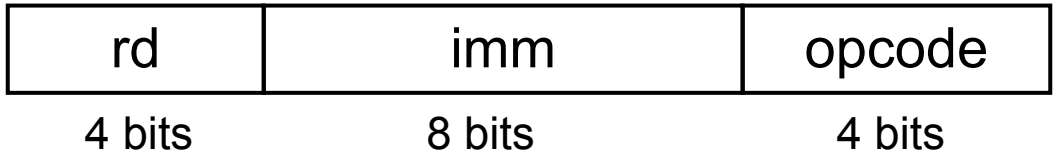



$$\mathbf{0110\ 0001\ 0010\ 0110_2 = 6126_{16}}$$

Dica: Como transferir dados de um registrador a outro?

**ADDI** R6, R5, #0

# Instrução de Movimentação (MOV)




- Instrução **MOV** usa imediatos para preencher registradores
  - **rd**: número do registrador de **destino**
  - **immediate**: valor a ser carregado (com extensão de sinal)
- Exemplo de instrução:

**MOV** R6, #2


$$\mathbf{0110\ 0000\ 0010\ 0011_2 = 6024_{16}}$$

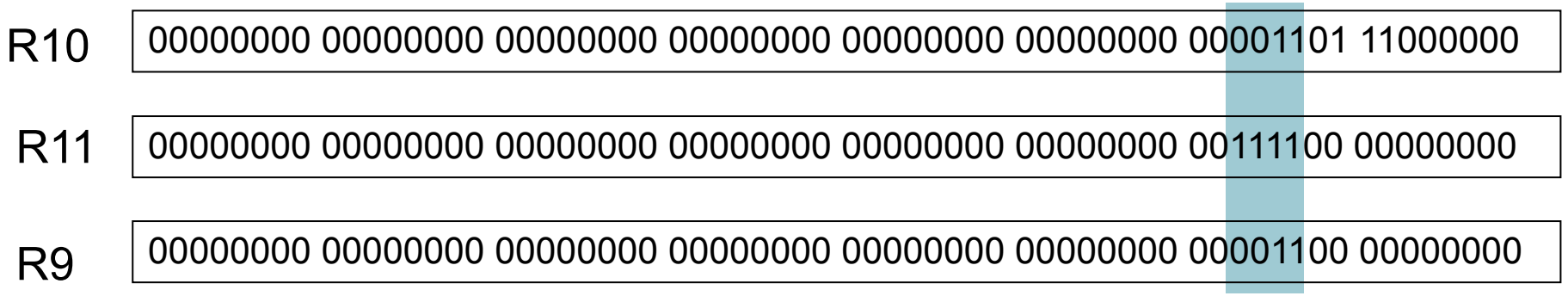
# Operações Lógicas

- Instruções para manipulação bit-a-bit (ou *bitwise*)
- Útil para **extrair** e **inserir** grupos de bits em uma palavra
  - Exemplo: Como extrair APENAS os **bits 3 a 15** da palavra e modificar APENAS os **bits 18 e 19**?

| Operação | C | Java | RISC-6 |
|---|---|---|---|
| *Shift left* | << | << | SHL |
| *Shift right* | >> | >>> | SHR |
| AND bit-a-bit | & | & | AND |
| OR bit-a-bit | \| | \| | OR |

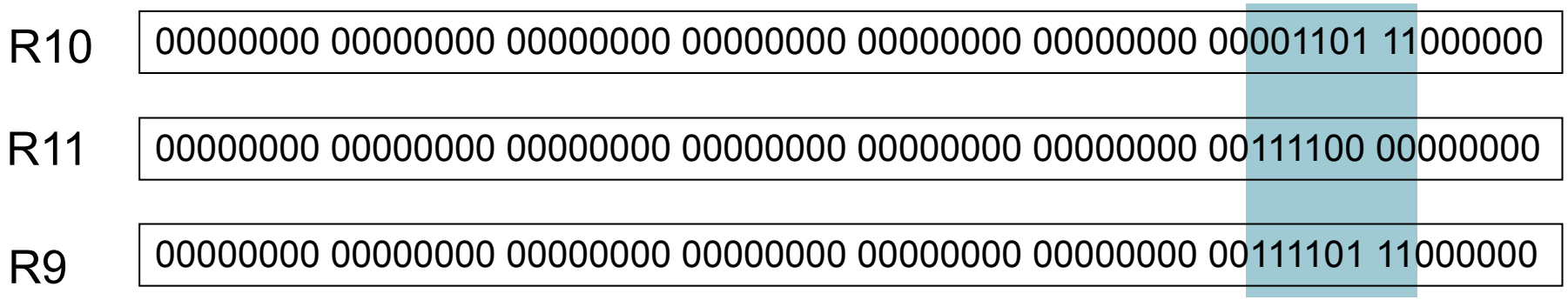
# Operação Lógica AND

- Útil para **mascarar bits** em uma palavra ⟶ Selecione alguns bits, limpe outros para 0
- Exemplo: AND R9, R10, X11



# Operação Lógica OR

- Útil para **incluir bits** em uma palavra ⟶ Atribua o valor 1 a alguns bits e não altere os outros
- Exemplo: OR R9, R10, R11



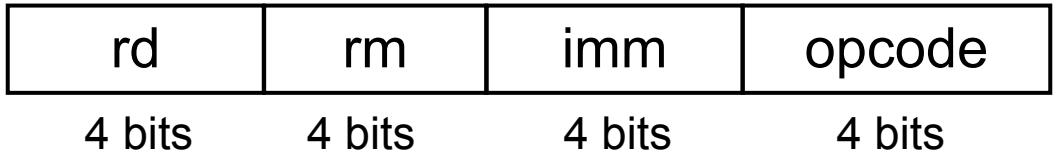
# Exemplo: Programa Simples

```
0000 0054 MOV R0, #5        // R0 = 0x0005
0001 1056 ADDI R1, R0, #5   // R1 = 0x000A
0002 2158 SUBI R2, R1, #5   // R2 = 0x0005
0003 3125 ADD R3, R1, R2    // R3 = 0x000F
0004 4327 SUB R4, R3, R2    // R4 = 0x000A
0005 540A OR R5, R4, R0     // R5 = 0x000F
0006 6509 AND R6, R5, R0    // R6 = 0x0005
0007 752C SHL R7, R5, #2    // R7 = 0x003C
0008 872B SHR R8, R7, #2    // R8 = 0x000F
0009 FFFF HALT              // Stops CPU
```

# Conjunto de Instruções do Processador

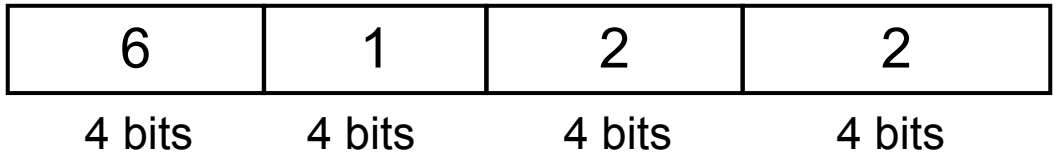
## Acesso à Memória

# Instrução de Load (LDR)



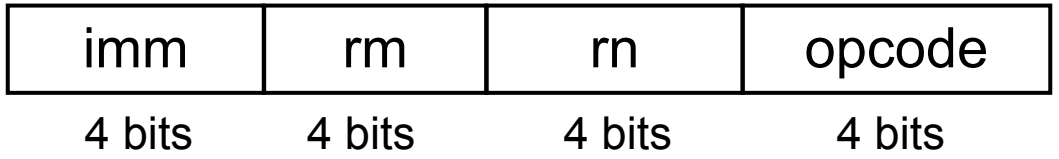

- Instrução **LDR** também usa imediatos: Mesmo formato
  - **rd**: número do registrador de **destino**
  - **rm**: número do registrador de **endereço base**
  - **imm**: deslocamento adicionado ao endereço base
- Exemplo de instrução de Load:

**LDR** R6, [R1, #2]



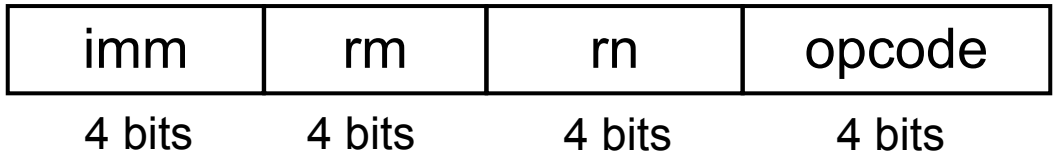

$$\mathbf{0110\ 0001\ 0010\ 0011_2 = 6122_{16}}$$

# Instrução de Store (STR)




- Instrução **STR** é semelhante à instrução **LDR** ⟶ Por que não usa o mesmo formato?
  - **LDR** R6, [R1, #2] ⟶ R6 = *(R1 + 2)
  - **STR** R6, [R1, #2] ⟶ *(R1 + 2) = R6
  - **STR** não possui registrador de destino (**rd**)!
- Precisamos de formato imediato diferente para instruções de store
  - **rm**: número do registrador de endereço base
  - **rn**: número do registrador do operando de origem
  - **imm**: deslocamento adicionado ao endereço base
- 3º Princípio de Projeto: "*Um bom projeto exige bons comprometimentos*"
  - Formatos diferentes complicam a decodificação, mas permitem instruções uniformes
  - Mantenha os formatos o mais semelhantes possível

# Exemplo de Instrução de Store




**STR** R6, [R1, #2]

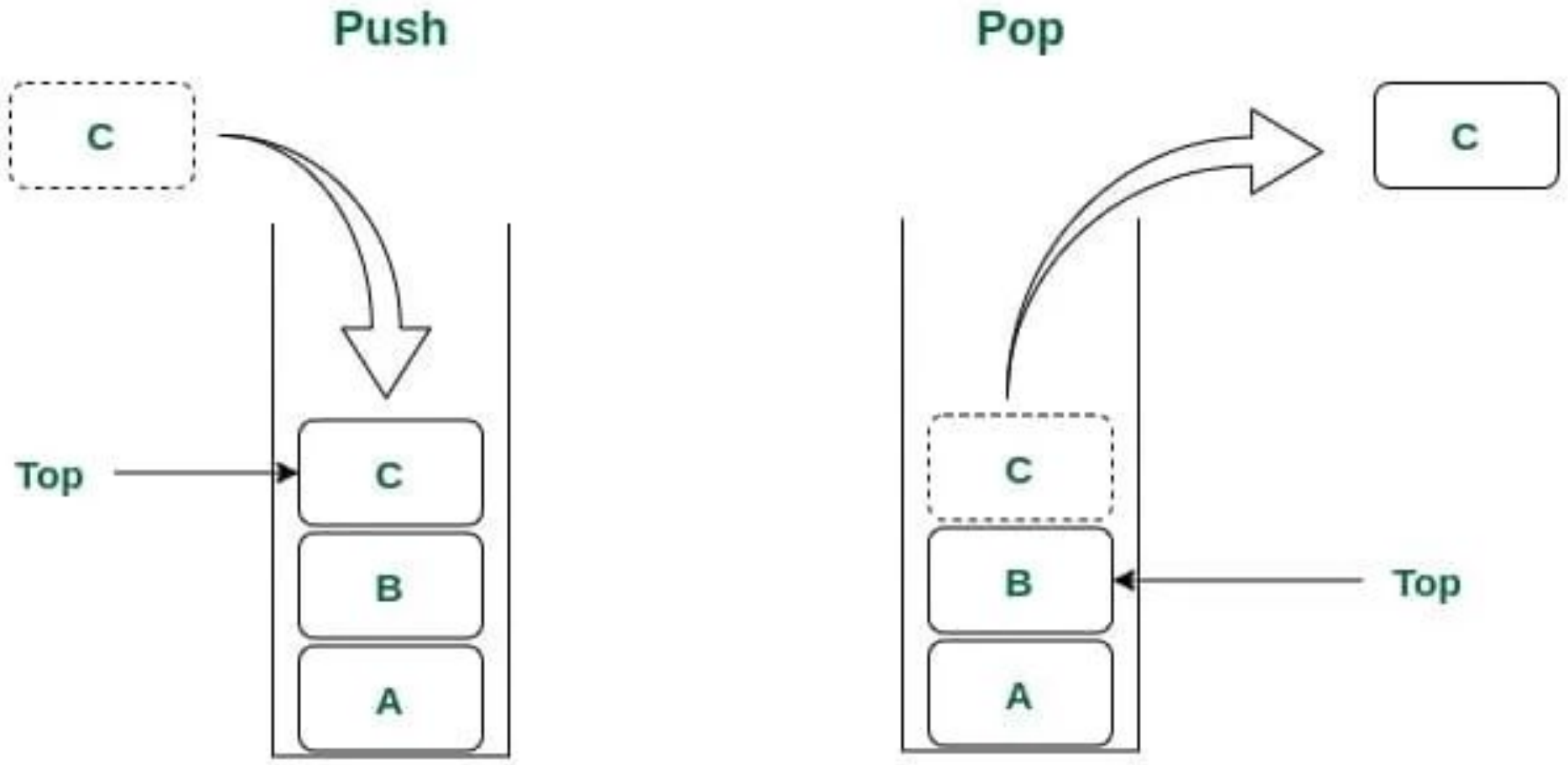

$$\mathbf{0010\ 0001\ 0110\ 0011_2 = 2163_{16}}$$

# Exemplo: Acesso à Memória

```
0000 0084 MOV R0, #var1       // R0 = 0x0008
0001 1002 LDR R1, [R0, #0]    // R1 = [0x08] = 0x0A
0002 2012 LDR R2, [R0, #1]    // R2 = [0x09] = 0x14
0003 3022 LDR R3, [R0, #2]    // R3 = [0x0A] = 0x1E
0004 4125 ADD R4, R1, R2     // R4 = 0x1E
0005 4435 ADD R4, R4, R3     // R4 = 0x3C
0006 3043 STR R4, [R0, #3]   // [0x0B] = 0x3C
0007 FFFF HALT               // Stops CPU!
0008 000A var1: 0x0A         // 10
0009 0014 var2: 0x14         // 20
000A 001E var3: 0x1E         // 30
000B 0000 var4: 0
```

# Pilha de Execução

- Todo processador mantém uma pilha de execução na memória
  - Região de memória ⟶ Próxima posição válida apontada por SP (Ponteiro de Pilha)
- Duas operações básicas ⟶ **PUSH** (empilhar) e **POP** (desempilhar)



# Funcionamento da Pilha

- Pilha de Hardware é **invertida**
  - Da posição **mais alta** para a **mais baixa**
- Geralmente SP aponta para o **final da memória**
  - Se a memória tem 16KB…
  - …SP aponta para além da última posição (0x2000)
  - SP aponta para o **topo** → Último dado empilhado
- **Empilhar** decrementa SP **antes** de colocar o dado
- **Desempilhar** incrementa SP **depois** de remover
- Empilhar R4 → **PUSH** R4, é equivalente à:
  - **SUBI** R14, R14, #1
  - **STR** R4, [R14, #0]
- Desempilhar em R4 → **POP** R4, é equivalente à:
  - **LDR** R4, [R14, #0]
  - **ADDI** R14, R14, #1

**SP →**

| Endereço | Conteúdo |
|---|---|
| 0x1FFF | 10 |
| 0x1FFE | ? |
| 0x1FFD | ? |

# Exemplo de Manipulação da Pilha

```
PUSH R0      // R0 = 10
PUSH R1      // R1 = 20
PUSH R2      // R2 = 30
POP R3       // R3 = ?
POP R4       // R4 = ?
POP R5       // R5 = ?
```

SP →

| Endereço | Conteúdo |
|---|---|
| 0x1FFF | ? |
| 0x1FFE | ? |
| 0x1FFD | ? |

# Exemplo de Manipulação da Pilha

```
PUSH R0      // R0 = 10
PUSH R1      // R1 = 20
PUSH R2      // R2 = 30
POP R3       // R3 = ?
POP R4       // R4 = ?
POP R5       // R5 = ?
```

SP →

| Endereço | Conteúdo |
|---|---|
| 0x1FFF | 10 |
| 0x1FFE | ? |
| 0x1FFD | ? |

# Exemplo de Manipulação da Pilha

```
PUSH R0      // R0 = 10
PUSH R1      // R1 = 20
PUSH R2      // R2 = 30
POP R3       // R3 = ?
POP R4       // R4 = ?
POP R5       // R5 = ?
```

SP →

| Endereço | Conteúdo |
|---|---|
| 0x1FFF | 10 |
| 0x1FFE | 20 |
| 0x1FFD | ? |

# Exemplo de Manipulação da Pilha

```
PUSH R0      // R0 = 10
PUSH R1      // R1 = 20
PUSH R2      // R2 = 30
POP R3       // R3 = ?
POP R4       // R4 = ?
POP R5       // R5 = ?
```

| | Endereço | Conteúdo |
|---|---|---|
| | 0x1FFF | 10 |
| | 0x1FFE | 20 |
| **SP →** | 0x1FFD | 30 |

# Exemplo de Manipulação da Pilha

```
PUSH R0      // R0 = 10
PUSH R1      // R1 = 20
PUSH R2      // R2 = 30
POP R3       // R3 = ?
POP R4       // R4 = ?
POP R5       // R5 = ?
```

| | Endereço | Conteúdo |
|---|---|---|
| | 0x1FFF | 10 |
| SP → | 0x1FFE | 20 |
| | 0x1FFD | 30 |

# Exemplo de Manipulação da Pilha

```
PUSH R0      // R0 = 10
PUSH R1      // R1 = 20
PUSH R2      // R2 = 30
POP R3       // R3 = ?
POP R4       // R4 = ?
POP R5       // R5 = ?
```

SP →

| Endereço | Conteúdo |
|---|---|
| 0x1FFF | 10 |
| 0x1FFE | 20 |
| 0x1FFD | 30 |

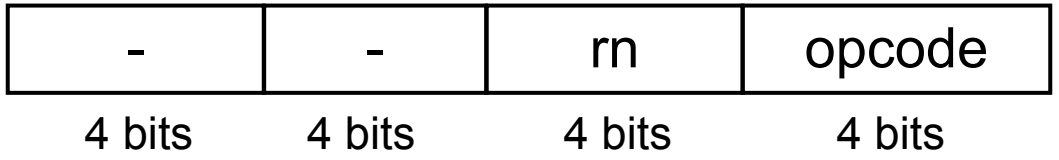
# Exemplo de Manipulação da Pilha

```
PUSH R0      // R0 = 10
PUSH R1      // R1 = 20
PUSH R2      // R2 = 30
POP R3       // R3 = ?
POP R4       // R4 = ?
POP R5       // R5 = ?
```

SP →

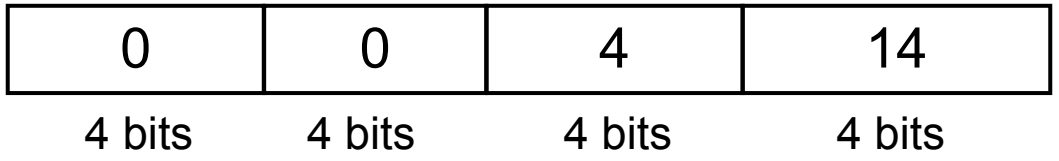
| Endereço | Conteúdo |
|---|---|
| 0x1FFF | 10 |
| 0x1FFE | 20 |
| 0x1FFD | 30 |

# Instrução PUSH



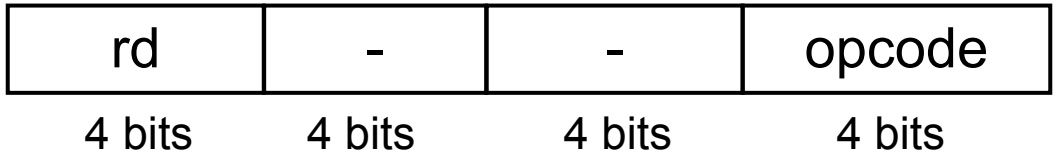

- "Empilha" ⟶ Coloca dado na pilha (após decrementar SP)
- Endereço está implicito em SP (R14), e não há registrador de destino (Rd), portanto:
  - **rn**: número do registrador de **origem** com o dado a ser empilhado
- Exemplo de instrução PUSH:

**PUSH** R4




$$\mathbf{0000\ 0000\ 0100\ 1110_2 = 004E_{16}}$$

# Instrução POP




- "Desempilha" → Retira dado da pilha (depois incrementa SP)
- Endereço está implicito em SP (R14), e não há registrador de origem (Rn), portanto:
  - **rd**: número do registrador de **destino**
- Exemplo de instrução POP:

**POP** R4


**$0100\ 0000\ 0000\ 1110_2 = 400F_{16}$**

# Exemplo: Acesso à Memória e Pilha

```
0000 00C4 MOV R0, #var1        // R0 = 0x000C
0001 1002 LDR R1, [R0, #0]     // R1 = [0x0C] = 0x0A
0002 2012 LDR R2, [R0, #1]     // R2 = [0x0D] = 0x14
0003 3022 LDR R3, [R0, #2]     // R3 = [0x0F] = 0x1E
0004 001E PUSH R1              // [SP - 1] = 0x0A, SP = 0x1FFF
0005 002E PUSH R2              // [SP - 2] = 0x14, SP = 0x1FFE
0006 003E PUSH R3              // [SP - 3] = 0x1E, SP = 0x1FFD
0007 4125 ADD R4, R1, R2       // R4 = 0x1E
0008 4435 ADD R4, R4, R3       // R4 = 0x3C
0009 3043 STR R4, [R0, #3]     // [0x0F] = 0x3C
000A 500F POP R5               // R5 = 0x1E
000B FFFF HALT                 // Stops CPU!
000C 000A var1: 0x0A           // 10
000D 0014 var2: 0x14           // 20
000E 001E var3: 0x1E           // 30
000F 0000 var4: 0
```


# Conjunto de Instruções do Processador

## Controle de Fluxo

# Fluxo do Programa

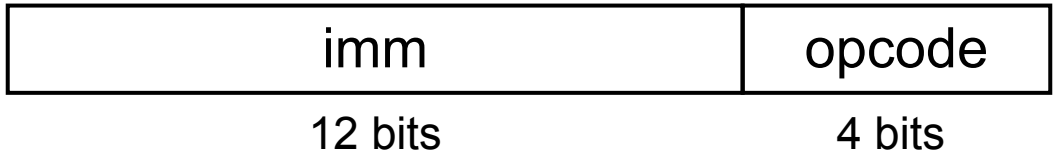
- Para um ISA ser **completo** ele deve ser capaz de realizar todas as operações necessárias!
  - Por exemplo: Conseguir compilar QUALQUER código em C
- Como construir estruturas do C em Assembly?
  - if-else
  - while
  - do-while
  - for
  - switch
- Estruturas que modificam o **fluxo do programa**
  - Precisamos de instruções para isso ⟶ Instruções de **salto**

```
if(a == b) {
    c = 1;
}
else {
    c = 2;
}
d = 3;
```

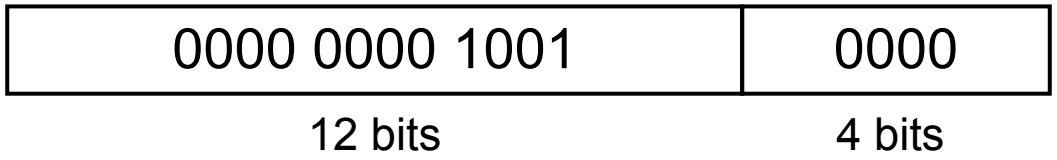
# Realizando Saltos

- Para modificar o fluxo do programa ⟶ Modificar PC (R15)
- Formas mais simples:
  - **MOV** R15, #20 ⟶ Salta para instrução cujo endereço é 20!
  - E se quiser saltar para uma instrução em um endereço além de 8 bits?
- Devemos realizar um salto relativo ao valor de PC:
  - **JMP** #9 ⟶ PC = PC + 9 (salta para frente)
  - **JMP** #-11 ⟶ PC = PC - 11 (salta para trás)
  - Salto irá depender do **deslocamento codificado** na instrução
  - Mas lembre-se: PC aponta para a próxima instrução
- ISA deve fornecer forma de manipular PC
  - Diretamente (e.g. RISC-6) ou via instruções (e.g. x86)
  - Veremos mais adiante

# Representando Saltos Incondicionais

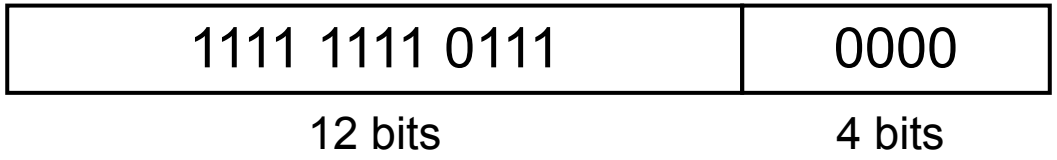



- Instrução **JMP** usa imediato para codificar o **deslocamento**
- Exemplo de instrução de **JMP** #9:




$$0000\ 0000\ 1001\ 0000_2 = 0090_{16}$$

- Exemplo de instrução de **JMP** #-11:



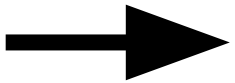

$$1111\ 1111\ 0111\ 0000_2 = FF70_{16}$$

# Exemplo: Saltos Incondicionais

- Em Assembly → Saltar para "instruções etiquetadas"
  - Assembler ajusta o deslocamento
- Exemplo: **goto** e **while(1)** do C em Assembly:

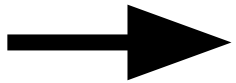
```
int main(void) {
    while(1) {
        a += b;
        b -= c;
        c |= d;
        goto skip;
        d &= e;
skip:   e += 1;
    }
}
```

C



```
main:  ADD R0, R0, R1
       SUB R1, R2, R3
       OR R2, R3, R4
       JMP skip
       AND R3, R4, R4
skip:  ADDI R4, R4, #1
       JMP main
       HALT
```

Assembly



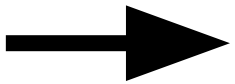
```
0000 0015 ADD R0, R0,
0001 1237 SUB R1, R2,
0002 234A OR R2, R3, F
0003 0010 JMP #1
0004 3449 AND R3, R4,
0005 4416 ADDI R4, R4,
0006 FF90 JMP #-7
0007 FFFF HALT
```

Assembled

# Exemplo: Comando break do C

```
int main(void) {
    while(1) {
        a += b;
        b -= c;
        break;
        c |= d;
    }
}
```

C



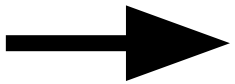
```
main:  ADD R0, R0, R1
       SUB R1, R2, R3
       JMP stop
       OR R2, R3, R4
       JMP main
stop:  HALT
```

Assembly

# Exemplo: Comando continue do C

```c
int main(void) {
    while(1) {
        a += b;
        b -= c;
        break;
        c |= d;
    }
}
```

C



```
main:  ADD R0, R0, R1
       SUB R1, R2, R3
       JMP main
       OR R2, R3, R4
       JMP main
       HALT
```

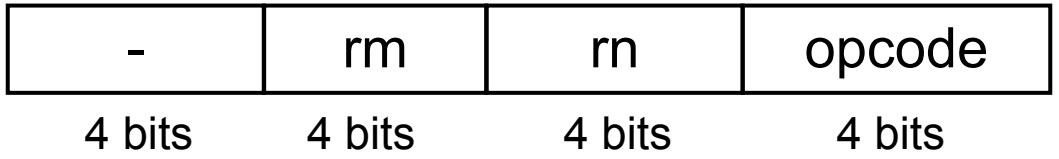
Assembly

# Comparações e Flags

- Como realizar saltos condicionais? Precisamos de instruções especiais!
- Mas antes… Como representar o **estado** em que as condições irão se basear?
- Registrador de **FLAGS** ⟶ Mantém o estado atual do processador baseado no último resultado
  - Flag **Z** ⟶ Última operação resultou em **zero** (ou seja, rd == 0)
  - Flag **C** ⟶ Último operação resultou em **carry-out** (para simplificar, rd < 0)
- Instruções da ALU ⟶ Atualizam as flags
- Exemplo: **SUB** R0, R1, R2
  - Se R0 = 4 ⟶ Z = 0 & C = 0 ⟶ Logo, R1 > R2
  - Se R0 = 0 ⟶ Z = 1 & C = 0 ⟶ Logo, R1 == R2
  - Se R0 = -4 ⟶ Z = 0 & C = 1 ⟶ Logo, R1 < R2
- Podemos realizar várias verificações!

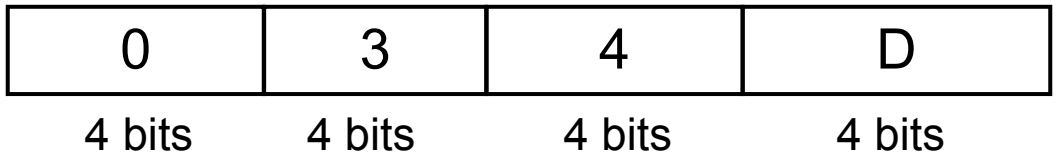

FLAGS

# Instrução de Comparação (CMP)




- Instrução **apenas** para comparação → Não modifica registradores
  - Compara os registradores **rm** e **rn**
  - Salva resultado nas *flags* **Z** (rm == rn) e **C** (rm < rn)
- Exemplo de instrução de comparação:

**CMP** R3, R4




$$\mathbf{0000\ 0011\ 0100\ 1101_2 = 034D_{16}}$$

# Saltos Condicionais

- Como saltar baseado nessas flags (condicionalmente)? Precisamos de instruções especiais!
- Saltar para “instruções etiquetadas” se uma condição foi verdadeira
  - Ou continua sequenciamente
- **JEQ** label → “*Jump if equal*”
  - Se a **flag Z**ero estiver ativa, então salta para a instrução com a etiqueta *label*
- **JNE** label → “*Jump if not equal*”
  - Se a **flag Z**ero não estiver ativa, então salta para a instrução com a etiqueta *label*
- **JGE** label → “*Jump if greater or equal*”
  - Se a **flag Z**ero estiver ativa e a **flag C**arry não, então salta para a instrução com a etiqueta *label*
- **JLT** label → “*Jump if less than*”
  - Se a **flag Z**ero não estiver ativa e a **flag C**arry sim, então salta para a instrução com a etiqueta *label*

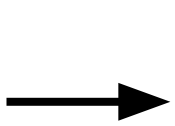
# Representando Saltos Condicionais


- Instrução **J<cond>** usa imediato para codificar o **deslocamento**
  - Tem range menor que JMP (12 bits) → 10 bits permite deslocamento de ±512 instruções
- Utiliza dois bits adicionais para codificar a condição
  - **EQ** → $00_2$
  - **NE** → $01_2$
  - **LT** → $10_2$
  - **GE** → $11_2$
- Exemplo de instrução de **JLT** #-11: $1011\ 1111\ 0111\ 0001_2 = BF71_{16}$
- Exemplo de instrução de **JNE** #11: $0100\ 0000\ 1011\ 0001_2 = 40B1_{16}$

# Bloco while do C

```
int main(void) {
    int size = 5;
    int arr[5] = {1, 2, 3, 4, 5};
    int res = 0;

    int i = 0;
    do {
        res += arr[i++]
    } while(size-- != 0);
}
```



```
main:   MOV R0, #size
        MOV R3, #0
        LDR R1, [R0, #0]
loop:   ADDI R0, R0, #1
        LDR R2, [R0, #0]
        ADD R3, R3, R2
        SUBI R1, R1, #1
        JNE loop
        STR R3, [R0, #1]
        HALT
```

```
size: 5
arr0: 1
arr1: 2
arr2: 3
arr3: 4
arr4: 5
res: 0
```

# Conjunto de Instruções do Processador

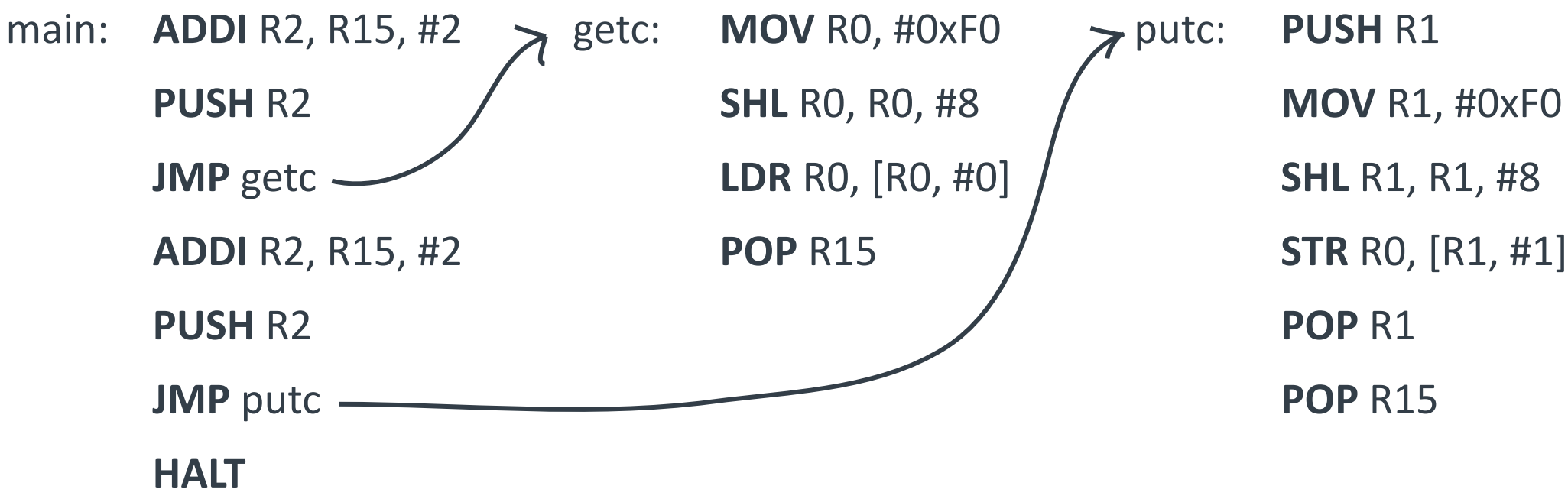
## Procedimentos

# Chamada de Procedimentos

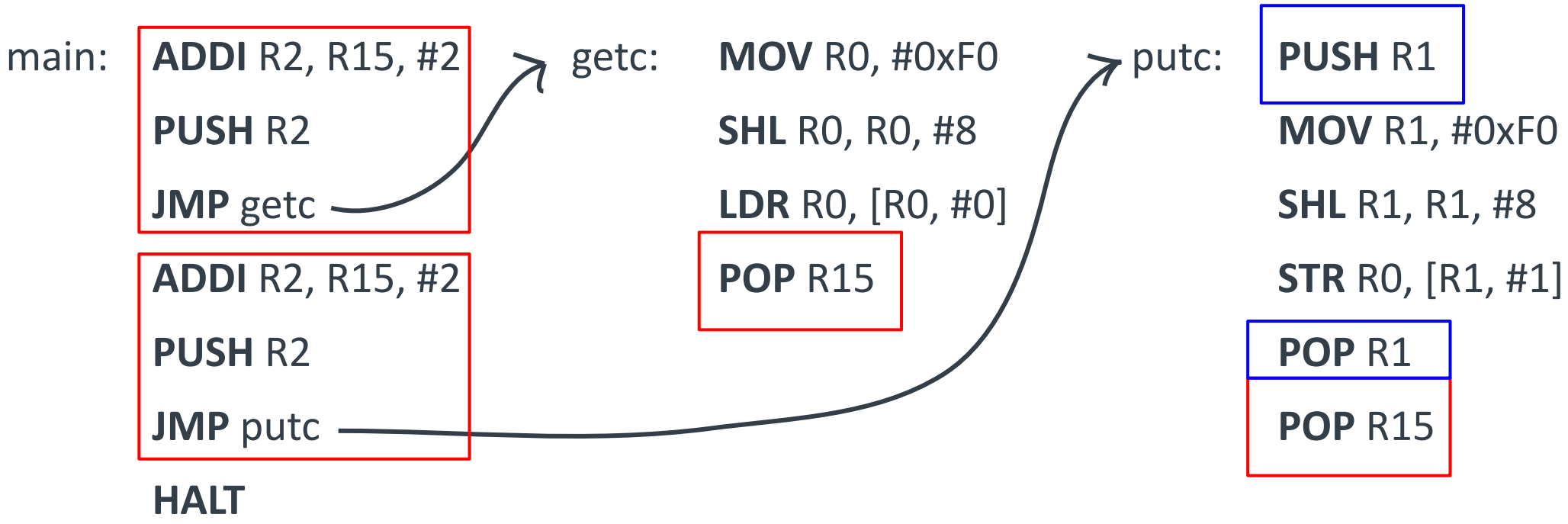
- Procedimentos ⇄ Funções
  - **Trechos de código** invocados por outros trechos de código
  - Realizam processamento
  - Retornam para quem invocou (ou **chamou**)
- Permitem mais organização e reuso de código
  - Pilha pode ser usada para salvar registradores modificados
- Para **chamar** ⟶ Saltar para o início do procedimento
  - Salvar endereço de retorno (e.g. na pilha)
- Para **retornar** ⟶ Carregar o endereço de retorno em PC
  - Se na pilha, apenas desempilhar em PC

```
main:   MOV R0, #1
        MOV R1, #2
        ADDI R2, R15, #2
        PUSH R2
        JMP func
        HALT
func:   ADD R0, R0, R1
        POP R15
```

# Exemplo: Chamada e Retorno



# Exemplo: Chamada e Retorno



# Mais sobre Procedimentos

- Convenção de Chamada → Quais registradores utilizar na chamada
  - **Parâmetros** passados em R0, R1, R2 e R3, e o resto na **pilha**
  - **Retorno** em R0
  - Funções devem salvar registradores a serem modificados → Utilizados pelas funções que invocaram
- Procedimentos podem reservar espaço na pilha → Guardar **variavéis locais**
  - Decrementar SP em 8 → Reservar 8 byte na pilha!
- Procedimentos Aninhados → Procedimentos podem invocar outros procedimentos

# Exemplo: Procedimentos Aninhados

```
main:   MOV R0, #char1
        ADDI R1, R15, #2
        PUSH R1
        JMP puts
        HALT
char1:  'H'
char2:  'e'
char3:  'l'
char4:  'l'
char5:  'o'
char6:  '\0'
```

```
puts:   PUSH R1
start:  PUSH R0
        LDR R0, [R0, #0]
        OR R0, R0, R0
        JEQ end
        ADDI R1, R15, #2
        PUSH R1
        JMP putc
        POP R0
        ADDI R0, R0, #1
        JMP start
end:    POP R0
        POP R1
        POP R15
```

```
putc:   PUSH R1
        MOV R1, #0xF0
        SHL R1, R1, #8
        STR R0, [R1, #1]
        POP R1
        POP R15
```

# Conjunto de Instruções do Processador

## Conclusão

# Resumo da Aula

- O **Conjunto de Instruções** define a operação de um processador
- Instruções podem ser representadas em **binário** (executável) ou **assembly** (mais legível)
  - Instruções tem vários **campos** que indicam a operação e seu os elementos
  - Ainda, podem ter vários **formatos** de instrução e vários **registradores**
- Processadores realizam diversas **operações**: aritméticas, transferência de dados, lógicas, etc…
  - Operam em vários tipos de **dados** (números, endereços, caracteres e dados lógicos)
  - Arquiteturas precisam permitir que procedimentos sejam chamados e aninhados

# Conclusão

- Nessa Aula:
  - Conjunto de Instruções do Processador
- Bibliografia Principal:
  - Arquitetura e Organização de Computadores; Stallings, W.; 10ª Edição (Capítulo 12)
- Próxima Aula:
  - Projeto e Implementação de Processadores