


Design de Computadores

Aula 5

Insper

Relação entre ISA e fluxo de dados.

- 
- Tópicos:
 - Pseudocódigo;
 - Linguagem de montador;
 - Linguagem de máquina;
 - Descrição do RTL com RTN.
 - Objetivos de aprendizado:
 - Analisar um programa em alto nível e inferir o hardware necessário para executá-lo.

Transformando Pseudocódigo de Alto Nível em Hardware

- A ideia básica é:
 - Atravessar os níveis de abstração, de cima para baixo;
 - Até chegar ao nível desejado:
 - Neste caso, o RTL do hardware.
 - Tomando as decisões que definirão o tipo de processador obtido.

Metodologia

1. Transformar o pseudocódigo da aplicação em um nível de abstração mais baixo:


- Como a linguagem de montagem (*assembly*):
- Definindo as instruções necessárias.

→ Para tanto, devemos fazer algumas opções:

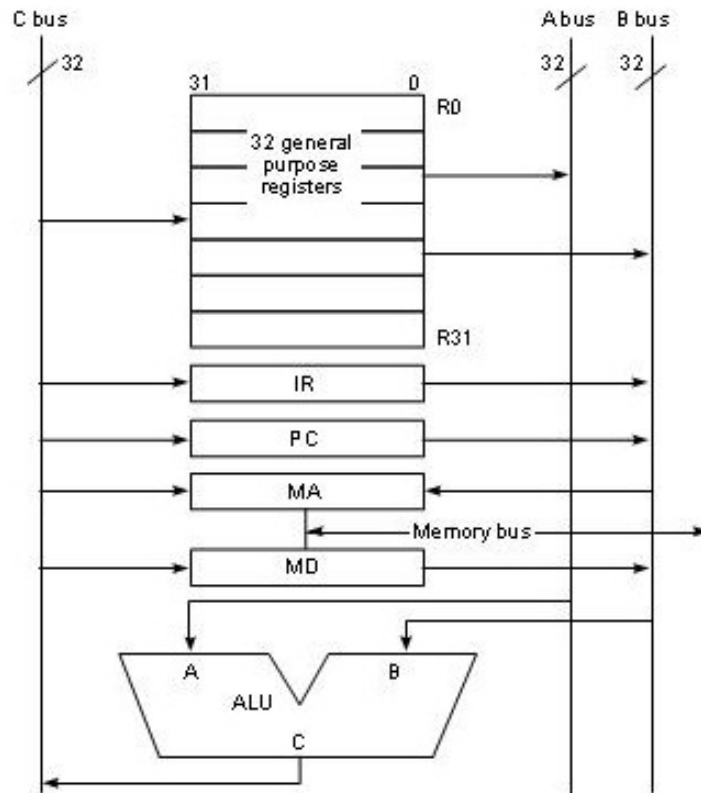
- A quantidade de operandos das instruções:
 - A quantidade de barramentos no processador;
 - O total de bits na palavra de instrução e sua função.

X #bits	Y #bits	Y #bits	Y #bits	Z #bits
opcode	reg_A	reg_B	reg_C	reservado

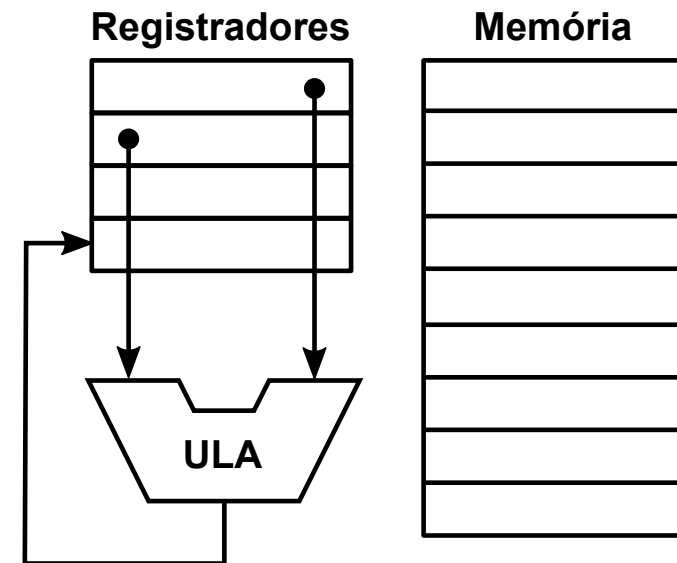
Palavra de Instrução
Largura (bits) = $(X+3Y+Z)$

- 
- Continuação das opções:
 - O tipo das instruções de desvio:
 - Desvio incondicional (jump);
 - Desvio Condicional (branch).
 - O tamanho da palavra de dados:
 - Não necessita ser o mesmo das instruções.
 - A quantidade de registradores;
 - O tratamento das constantes:
 - Valor imediato:
 - Um campo deve ser adicionado à palavra de instruções;
 - Ou criar um tipo de instrução para carga de imediatos.
 - Fixas no hardware, como o registrador zero do MIPS.

- Para simplificar, utilizaremos uma arquitetura com:
 - Três barramentos;
 - Baseada em registradores de uso geral (load/store).



Arquitetura: Registrador-Registrador



Source: Heuring – Jordan: Computer Systems Architecture and Design

- Por exemplo, o código de um laço pode ser convertido de pseudocódigo para linguagem de montador:

```
int sum = 0;
```

```
for (i = 0; i != 10; i = i + 1) {  
    sum = sum + i;  
}
```

```
# $R0 = 0; $R1 = i ; $R2 = soma;
```

```
# Carga das variáveis:
```

```
add $R2, $R0, $R0    # $R2 = soma = 0
```

```
add $R1, $R0, $R0    # $R1 = i = 0
```

```
addi $R4, $R0, 10    # $R4 = 10
```

```
inicioFOR:
```

```
# Se i == 10, vai para terminoFOR
```

```
    beq $R1, $R4, terminoFOR
```

```
    add $R2, $R2, $R1    # soma = soma + i
```

```
    addi $R1, $R1, 1    # incrementa i
```

```
    jmp inicioFOR
```

```
terminoFOR:
```

```
...
```

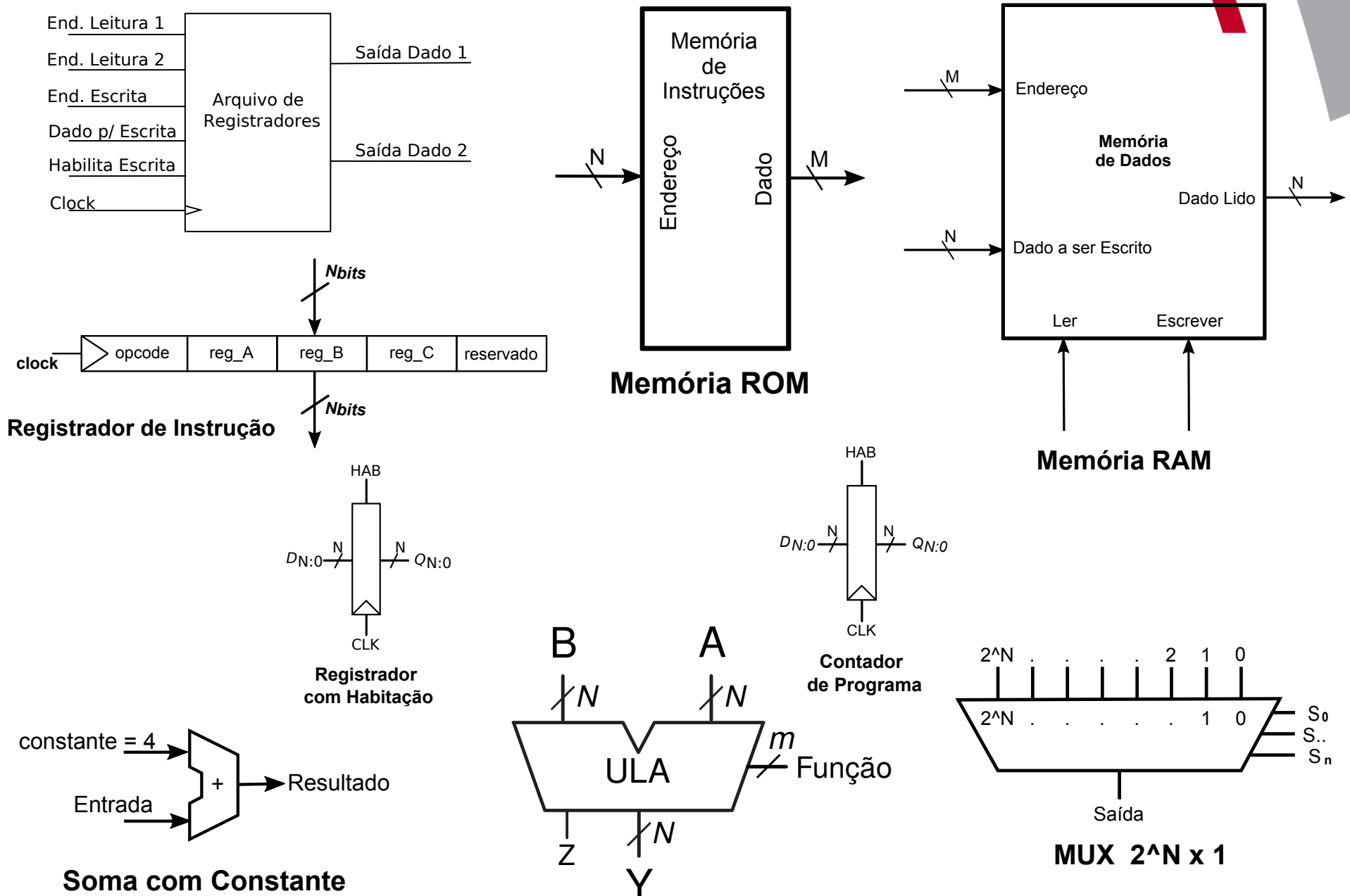
- As instruções obtidas pertencem as grupos:

Instrução	Grupo ou Tipo
add	Aritmética
addi	Aritmética com Imediato
beq	Desvio
jump	Desvio

2. Analisar esse conjunto de instruções em *assembly*:

- Dividir cada instrução em seus passos elementares, considerando:
 - A sequência de execução:
 - Busca, Decodifica, Lê Operandos, Executa, Devolve Resultado.
 - A lista de unidades funcionais disponíveis.
- Criando uma sequência de transferências entre essas unidades (RTL).
- Determine os recursos do fluxo de dados necessários:
 - Unidades funcionais, inclusive as funções da ULA;
 - Registradores e o caminho de transferência entre eles.

• Unidades funcionais mais comuns:



- Por exemplo, para as instruções do mesmo tipo da instrução *add*:

add \$R2, \$R0, \$R0 # \$R2 = soma = 0

- Podemos escrevê-las como:

operação \$Reg_{Destino}, \$Reg_{OperandoA}, \$Reg_{OperandoB}

- Podemos concluir que:
 - Utilizaremos a ULA, para fazer a operação (soma);
 - Existe um banco de registradores:
 - Com o registrador zero (\$R0) com seu valor fixo em zero.
 - E outro registrador para resultado da soma: o \$R2.
 - Esse banco alimenta as entradas da ULA.

- Considerando a sequência de execução, percebemos algumas invariantes:
 - Busca:
 - Ao carregar uma instrução:
 - Instrução = $\text{MEM}[\text{PC}]$
 - Deve ocorrer o incremento do contador de programa, para o acesso à próxima instrução:
 - $\text{PC} = \text{PC} + 1;$
 - Decodificação:
 - A instrução define a operação e endereça os registradores que serão utilizados:
 - $\text{REG}[\text{Ra}] = \text{REG}[\text{Rb}] \text{ operação } \text{REG}[\text{Rc}]$- Juntando com o que vimos da instrução *add*, podemos fazer um primeiro esboço.

- Primeiro esboço da descrição das transferências entre componentes do FD:

- $RI \leftarrow MEM[PC];$
- $UC \leftarrow RI[Bits_{opcode}];$
- $BancoREG_{ADDR-1} \leftarrow RI[Bits_{RegB}];$
- $BancoREG_{ADDR-2} \leftarrow RI[Bits_{RegC}];$
- $BancoREG_{ADDR-3} \leftarrow RI[Bits_{RegA}];$
- $REG[RegA] \leftarrow REG[RegB] \text{ operação } REG[RegC]$
- $PC \leftarrow PC + 1;$





4. Analisar cada instrução implementada:

- Determinando os pontos de controle necessários;
- E verificando os caminhos entre os componentes.

5. Para cada instrução:

- Montar as palavras controle;
- Simular manualmente a execução da instrução e fazer as alterações necessárias.



6. Para cada ponto de controle da palavra de controle:

- Definir quais instruções ativam esse determinado ponto.

7. Tente agrupar as instruções em conjuntos de características similares.

8. Para cada conjunto:

- Determine a sua utilização dos bits da instrução;
- Para cada instrução, determine a sua codificação.

9. Com essas informações, monte a unidade de controle.

- 
- Atividade:
 - Aplique essa metodologia ao projeto do processador do relógio.

Insper

www.insper.edu.br

