

## Introdução à Arquitetura de Computadores

Aula 21

### μArquitetura MIPS: Single-cycle - I

#### Introdução

μArquitetura: Definição e Tipos

#### A μArquitetura dum CPU

Máquina Síncrona; Arq. Harvard

Datapath e Controlo

Fases de projeto

#### Datapath

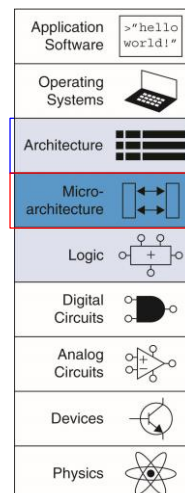
Subconjunto de Instruções (ISA)

Execução de Instruções:

Acesso a Dados (*lw* e *sw*)

Tipo-R (*add*, *sub*)

Branch (*beq*)



A. Nunes da Cruz / DETI - UA

Jun / 2021

## μArquitetura (1) - Introdução

### μArquitetura

- Como implementar o *hardware* da Arquitetura dum Processador (Arquitetura = Visão do programador do conjunto instruções, registos e memória).

### Processador

- **Datapath:** Blocos Funcionais

Memórias, registos, ALUs e multiplexers que operam sobre dados (*words* de 32-bits)

- **Unidade de Controlo:**

Determina como a instrução deve ser executada no *Datapath*, gerando sinais de seleção de multiplexers, de *enable* de registos, de *write* de memórias, etc.

*Datapath* = Caminho de Dados.

Os Blocos Funcionais estão interligados por *Buses* controlados pela Unidade de Controlo.

© A. Nunes da Cruz

IAC - MIPS - Single-cycle - I

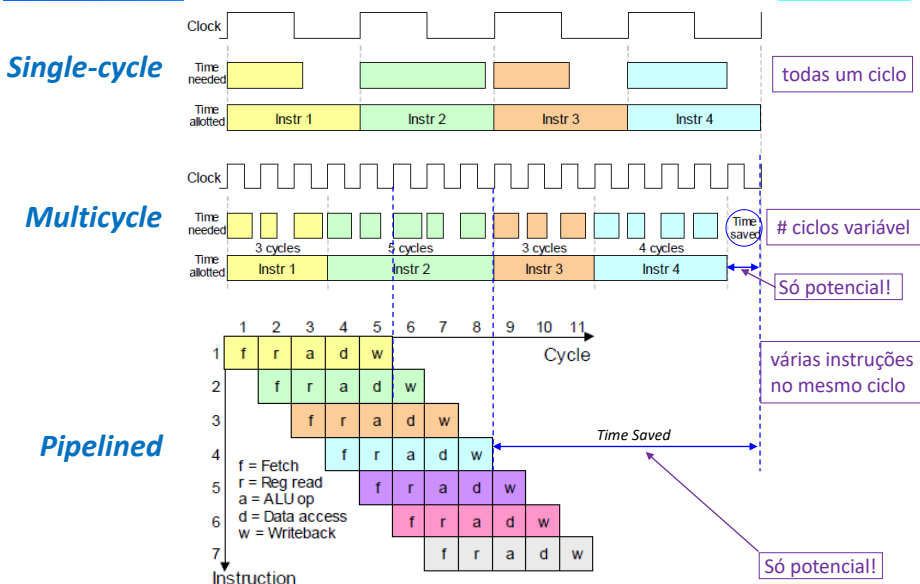
1/29

1. Introdução

## μArquitetura (2) - Três tipos de Implementações

- **Single-cycle:** A execução de cada instrução é efectuada num **único ciclo** de relógio (**clock**). Todas as instruções ocupam o **mesmo** intervalo de tempo. *↳ cada instrução tem de ser executada num único ciclo de relógio*
- **Multicycle:** A execução de cada instrução é dividida numa série de passos mais simples; cada um deles ocupa um ciclo de relógio (de maior frequência). As instruções possuem tempos de execução **diferentes** (e.g., **lw** = 5 ciclos e **beq** = 3 ciclos).
- **Pipelined:** A execução de cada instrução é dividida numa série de passos mais simples; o processador executa **múltiplas** instruções **em simultâneo** (em paralelo), aumentando, deste modo, a **performance**.

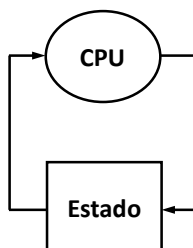
## μArquitetura (3) - Single-cycle vs Multicycle vs Pipelined



## CPU MIPS (1) - Máquina Síncrona

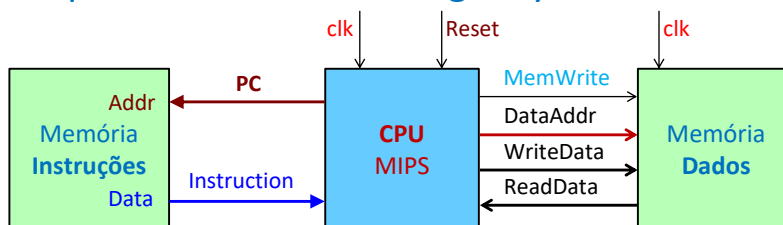
O CPU é uma máquina de estados síncrona.

- As Memórias, os Registos e outros autómatos definem o **estado**.
- A execução das instruções dum programa **altera o seu estado**.



## CPU MIPS (2) - $\mu$ Arquitetura Harvard\*

A  $\mu$ Arquitetura dum CPU Single-cycle



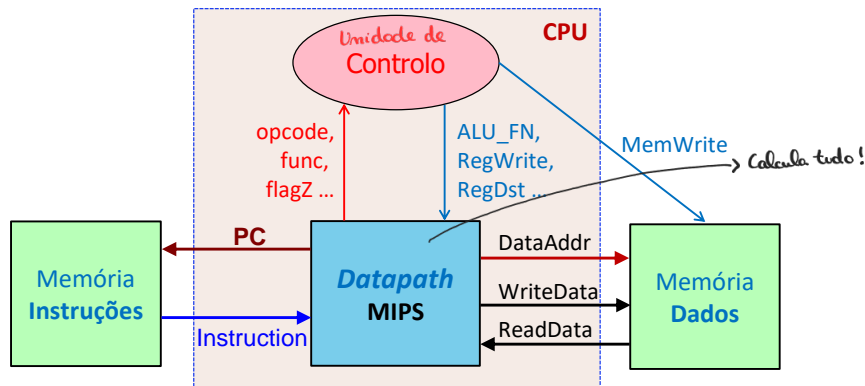
- O CPU MIPS interage com **duas** memórias.
- Após o **Reset**, o registo **PC** (Program Counter) é carregado com o **endereço** da **Memória de Instruções**, para ler a primeira **Instrução** a ser executada pelo CPU.
- O CPU gera os **sinais de Controlo** necessários à execução da **Instrução**, a qual pode ou não envolver a **Memória de Dados**.

\*Harvard = A implementação *Single-cycle* (Ciclo-único) usa memórias separadas (externas ao CPU) para permitir o acesso a ambas, em simultâneo (i.e., durante o mesmo ciclo) pelo CPU.

Não se pode fazer na arquitetura de Von Neuman: cada instrução implica um acesso à memória, como era preciso ir à memória 2 vezes no mesmo ciclo não era possível

### CPU MIPS (3) - Datapath e Unidade de Controlo

- **Datapath**: Componentes que armazenam ou processam dados
  - Registos, ALU, multiplexers, extensão-sinal, etc.



- **Controlo**: Componentes que 'dizem' ao *Datapath* o que fazer.
  - Lógica combinatória e/ou sequencial (FSMs).

### CPU MIPS (4) - Fases de Projeto

#### 1. Análise do Conjunto de Instruções (ISA)

- A execução de cada instrução requiere **transferências** entre **registos** e/ou **memórias**;
- O **Datapath** deve incluir o *hardware* necessário para suportar essas transferências.

#### 2. Seleção dos Componentes para o Datapath

- Memórias, Registos, ALU, Multiplexers, etc

#### 3. Implementação da Lógica de Controlo

- Lógica combinatória ou FSMs

**ISA** = *I*nstruction *S*et *A*rchitecture; **FSM** = *F*inite *S*tate *M*achine.

## SC Datapath (1) - Instruções (ISA) a Implementar

### ISA: Começamos com um número limitado

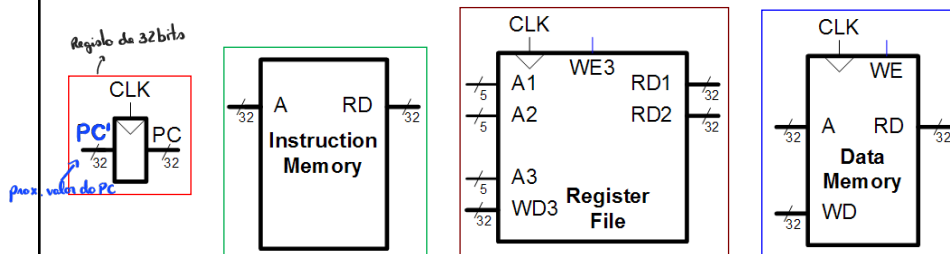
- Acesso à Memória  
**lw** e **sw**
- Tipo-R  
**and**, **or**, **add**, **sub** e **slt**
- Branch  
**beq**
- Instruções adicionais (próx. aula)  
**addi**  
**j**

3.1 Datapath - Instruções

## SC Datapath (2) - Elementos de Estado (1)

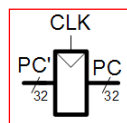
O *Datapath*\* dum CPU é baseado nos seguintes elementos de estado:

- Program Counter (PC)
- Memória de Instruções
- Banco de 32 Registos
- Memória de Dados



\*O projeto dum sistema complexo começa com o *hardware* dos elementos de estado.

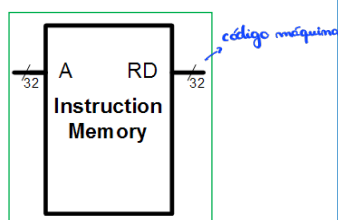
### SC Datapath (3) - Elementos de Estado (2): PC e Memórias



- **PC:** É um registro de 32-bits normal;
  - A saída (PC) é o endereço da instrução **corrente**;
  - A entrada (PC') é o endereço da instrução **seguinte**.

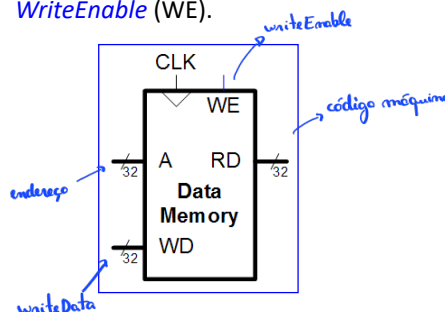
#### Memória de Instruções:

Tem um único porto de leitura (A/RD).



#### Memória de Dados:

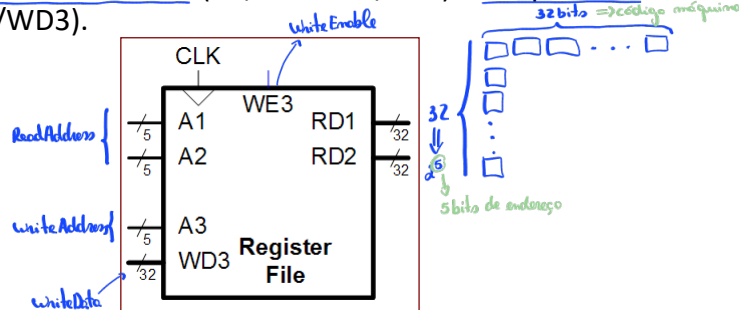
Tem um único porto de leitura (A/RD) ou de escrita (A/WD) e um sinal de WriteEnable (WE).



### SC Datapath (3) - Elementos de Estado (3) - Banco de Reg.

#### Banco de 32 Registos:

Tem **dois** portos de leitura (A1/RD1 e A2/RD2) e **um** porto de escrita (A3/WD3).



Os três *buses* **A1**, **A2** e **A3** possuem 5-bits, visto que  $2^5 = 32$  registos.

**Ex:** As instruções do tipo-R lêem os operandos dos portos (**RD1** e **RD2**) e escrevem o resultado no porto (**WD3**), e.g., **add \$t2, \$t1, \$t0**.

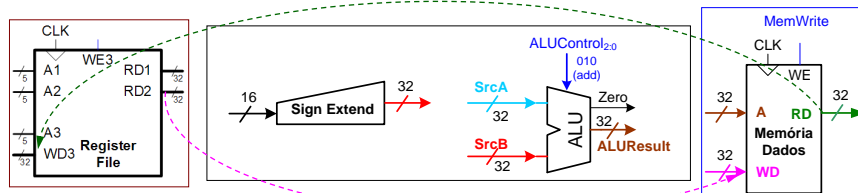
Exceptuando a Memória de Instruções, todos os elementos de estado têm entrada de CLK.

## SC Datapath (4) - Instruções Load/Store (1)

### Operações

$RT = Mem[RS + imm]$  ← **lw** **rt**, **imm**(**rs**)      **sw** **rt**, **imm**(**rs**)

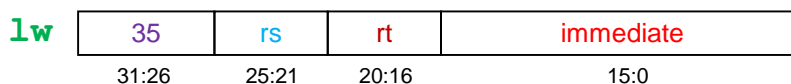
- Usam os registros **rs** e **rt** como operandos
- Calculam (na ALU) o endereço de memória efetivo usando (**rs**) e o **imm**.



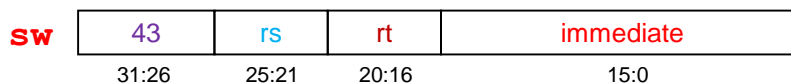
- **Load**: Lê da Memória (para o RF) o novo valor do registro **rt**.
- **Store**: Escreve o valor do registro **rt** (do RF) na Memória.

## SC Datapath (5) - Instruções Load/Store (2)

### Formato da Instrução - tipo-I



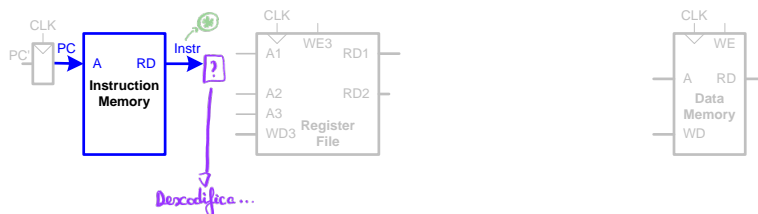
**lw** **rt**, **imm**(**rs**)



**sw** **rt**, **imm**(**rs**)

## SC Datapath (6) - *lw* Fetch

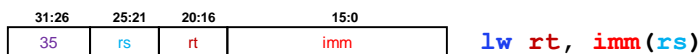
### Passo 1: Leitura da Instrução (Fetch)



1. O PC gera o endereço (A) para a Memória de Instruções.

A Instrução lida (RD) vai, em seguida, ser decodificada (e.g., o campo de bits  $\text{Instr}_{31:26}$  vai ser interpretado).

código máquina  
→ \*



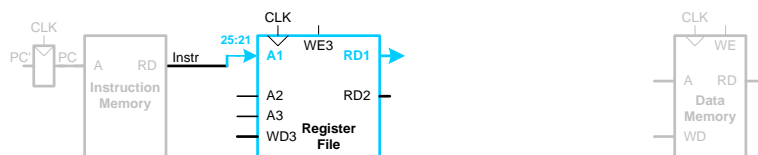
© A. Nunes da Cruz

IAC - MIPS - Single-cycle - I

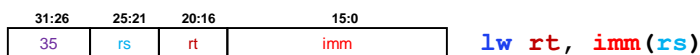
14/29

## SC Datapath (7) - *lw* Leitura do Operando

### Passo 2: Leitura do operando (rs) do Reg File (RF)



2. Ligando os bits  $\text{Instr}_{25:21}$ , rs, ao porto A1 do RF, obtemos na saída RD1 (ReadData1) o conteúdo desse registro, (rs).



© A. Nunes da Cruz

IAC - MIPS - Single-cycle - I

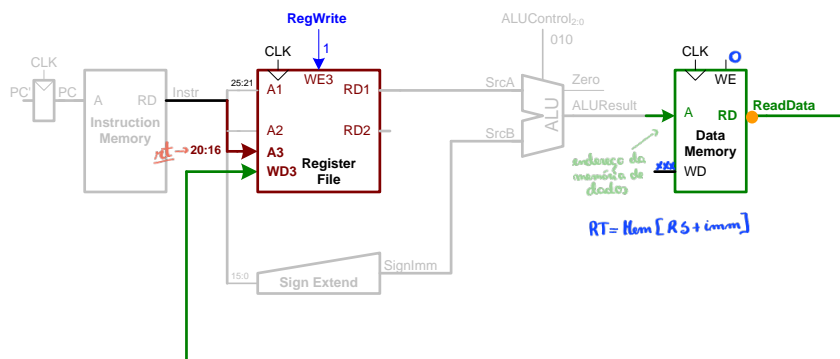
15/29



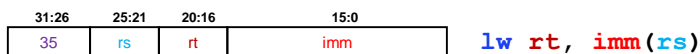


## SC Datapath (10) - *lw* Leitura da Memória de Dados

### Passo 5: Leitura do valor da Memória e escrita no RF



5. O valor lido da memória (ReadData) é escrito (RegWrite=1) no registo *rt* (*Instr*<sub>20:16</sub>) usando o porto *A3*/*WD3* do Register File.



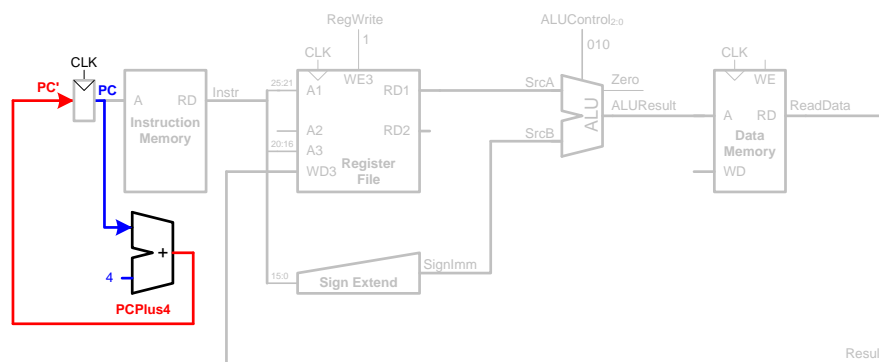
© A. Nunes da Cruz

IAC - MIPS - Single-cycle - I

18/29

## SC Datapath (11) - *lw* Incremento do PC

### Passo 6: Calcular PC'



6. Ao valor atual do Program Counter, PC, soma-se 4 para obter PC', i.e., o endereço da instrução seguinte a executar.

Acabou a execução da instrução *lw*!

© A. Nunes da Cruz

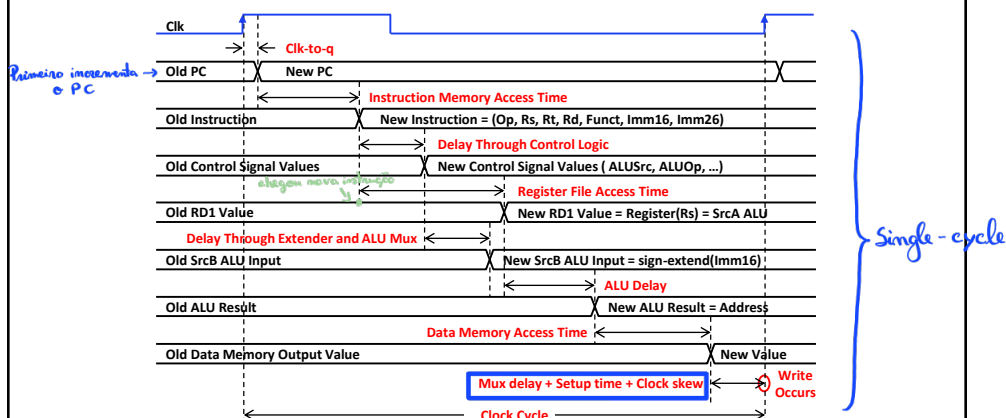
IAC - MIPS - Single-cycle - I

19/29

## SC Datapath (12) - Passo a Passo?

### Embora os slides digam 'Passo'

... todas estas operações são executadas no **mesmo** ciclo de relógio!



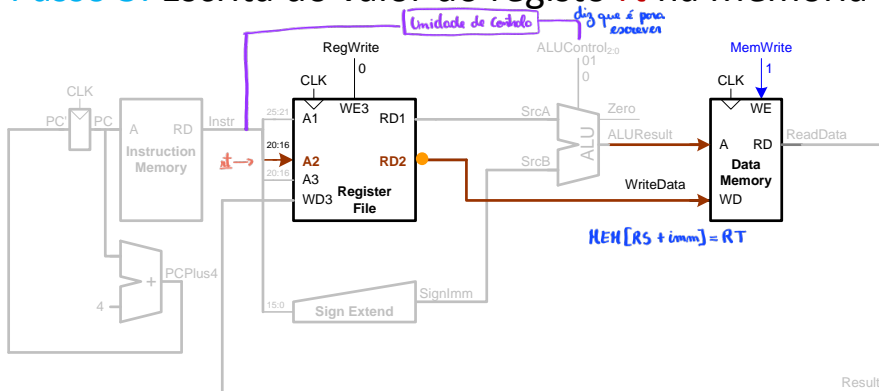
© A. Nunes da Cruz

IAC - MIPS - Single-cycle - I

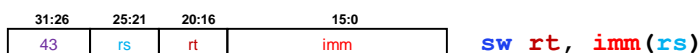
20/29

## SC Datapath (13) - **sw** Escrever na Memória de Dados

### Passo 5: Escrita do valor do registro **rt** na Memória



5. O valor do registro **rt**, RD2 (ReadData2), é escrito (MemWrite = 1) na memória (WriteData). Ao contrário de **lw**, nada é escrito no RF.



© A. Nunes da Cruz

IAC - MIPS - Single-cycle - I

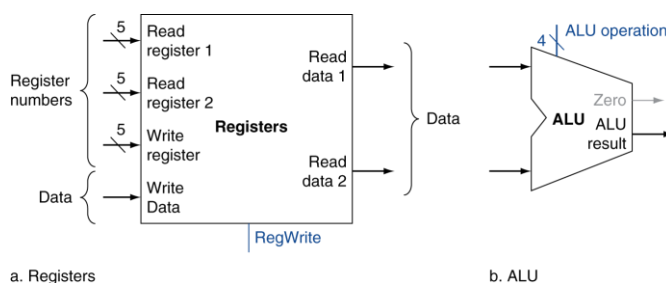
21/29

## SC Datapath (14) - Tipo-R (1) - add

### Operações

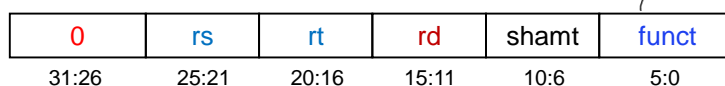
Ex: add **rd**, **rs**, **rt**

- Usa os registros **rs** e **rt** como operandos
- Executa a operação aritmética/lógica na ALU
- Escreve o resultado no registro **rd** (RF)



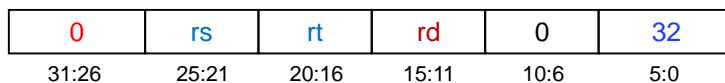
## SC Datapath (15) - Tipo-R (2) - add

### Formato da Instrução - tipo-R



add **rd**, **rs**, **rt**

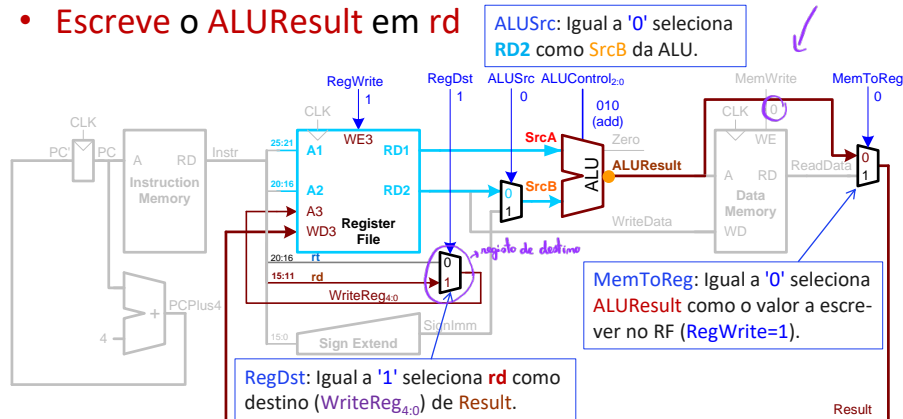
add



## SC Datapath (16) - Tipo-R (3) - add

- Lê os valores de **rs** e **rt** (add soma-os na ALU)

- Escreve o **ALUResult** em **rd**



A Memória de Dados não intervém na execução!

3 Multiplexers!

31:26	25:21	20:16	15:11	10:6	5:0
0	rs	rt	rd	0	32

add rd, rs, rt

© A. Nunes da Cruz

IAC - MIPS - Single-cycle - I

24/29

Tipo I

## SC Datapath (17) - Branch (1) - beq

Operações Ex: beq rs, rt, imm

- Calcula o endereço-alvo do branch (**BTA**) =  $(PC + 4) + imm_{16} \times 4$

- Obtem o **SignImm<sub>32</sub>** a partir do **Imm<sub>16</sub>**
- Multiplica **SignImm<sub>32</sub>** por 4 (para obter o endereço de byte)
- Soma esse valor a 'PC + 4'

$$BTA = (SignImm_{32} \ll 2) + (PC + 4)$$

- Em paralelo, compara os operandos (**rs**) e (**rt**)

- Subtrai-os na ALU e gera a saída **Zero**;
- Caso **Zero=1** então **PC' = BTA**;
- caso contrário **PC' = PC + 4**.

Se a saída Zero for 1 significa que a conta da ALU deu 0

Nome de um sinal lógico.

Se sinal "Zero" der 1 significa que a conta deu 0!

**BTA = Branch Target Address.**

(rs) = conteúdo do registro rs.

© A. Nunes da Cruz

IAC - MIPS - Single-cycle - I

25/29

Temos que fazer esta conta e decidir se é para saltar ou não.

→  $BTA = (PC + 4) + imm_{16} \times 4$

beq rs, rt, imm<sub>16</sub>

if (rs - rt) == 0

PC = BTA = (PC + 4) + imm<sub>16</sub> × 4

else PC = PC + 4

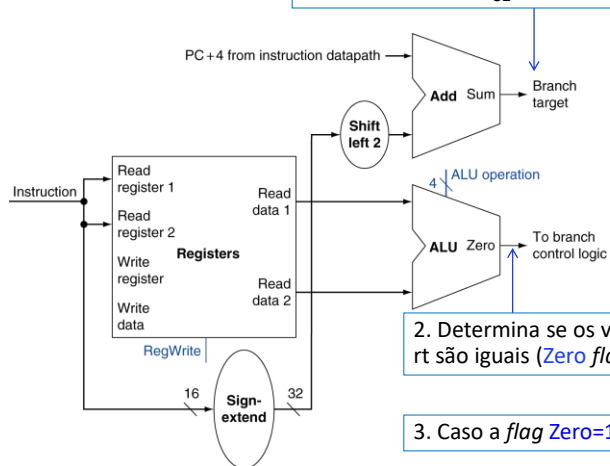
subtração e verificação se deu 0 foi feito na ALU. Logo, rs e rt já estão na entrada da ALU. Faltava um circuito que calcula isto.

## SC Datapath (18) - Branch (2) - *beq*

### Operações (cont.)

1. Calcula o *Branch Target Address* (BTA):

$$\text{BTA} = (\text{SignImm}_{32} \ll 2) + (\text{PC} + 4)$$



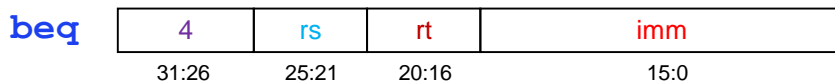
2. Determina se os valores de *rs* e *rt* são iguais (*Zero flag*)

3. Caso a *flag Zero=1*,  $\text{PC}' = \text{BTA}$

Os símbolos do *sign-extend* e *shift-left2* são diferentes mas também se usam noutros livros.

## SC Datapath (19) - Branch (3) - *beq*

### Formato da Instrução - tipo-I



*beq* *rs*, *rt*, *imm*

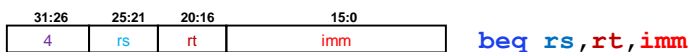
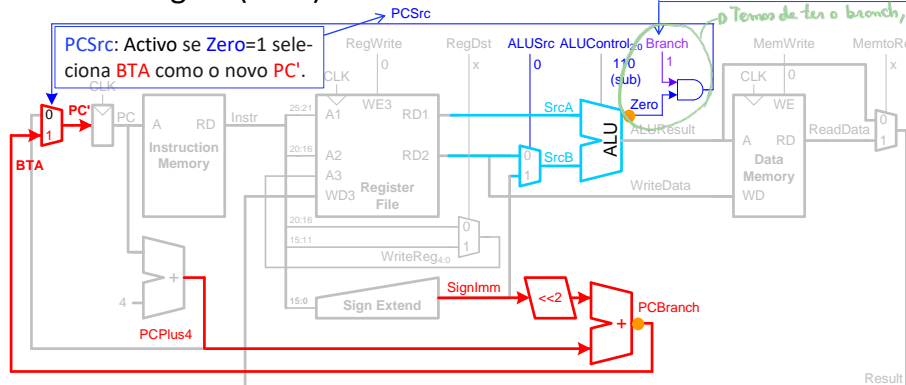
## SC Datapath (20) - Branch (4) - beq

- Calcula (PCBranch)  $BTA = (PC+4) + (\text{SignImm}_{31:0} \ll 2)$
- Determina (na ALU) se o valor dos registos  $rs$  e  $rt$  é igual (Zero)

Branch: Sinal de controlo que deteta a instrução beq.

PCSrc: Activo se Zero=1 selecciona BTA como o novo PC.

Temos de ter o branch, pois se uma Tipo R der resultado 0 não ir para o BTA



© A. Nunes da Cruz

IAC - MIPS - Single-cycle - I

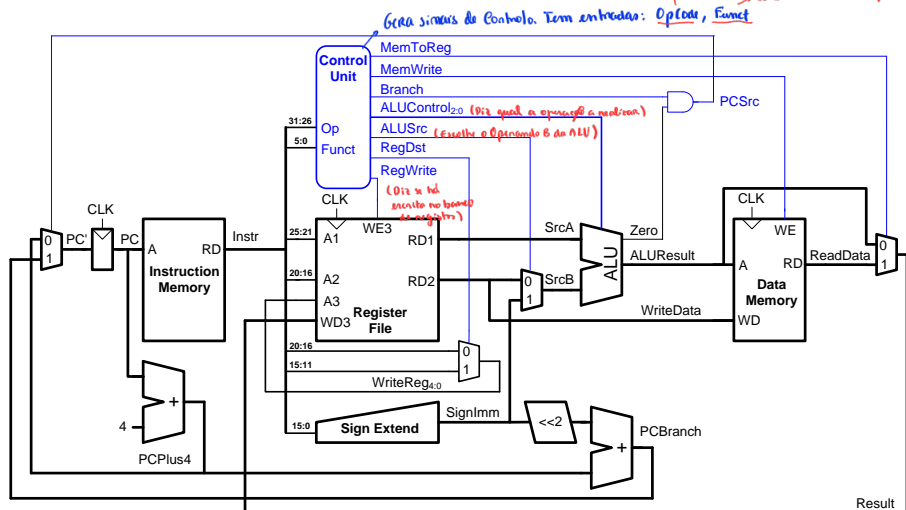
28/29

## SC Datapath (21) - Completo\* (sem Controlo)

operação a realizar

OpCode, Funct

se é necessário se OpCode for 0!



\*Mas, com um instruction set limitado a: lw/sw, tipo-R e beq

© A. Nunes da Cruz

IAC - MIPS - Single-cycle - I

29/29

- lw e sw
- add (tipo R)
- beq
- [addi, j]

jump!

O circuito da cima supõe a instrução de baixo? **Sim!**  
addi RT, RS, imm

addi → imm é sign-extended

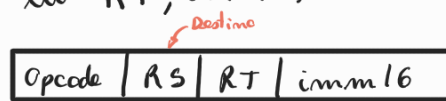
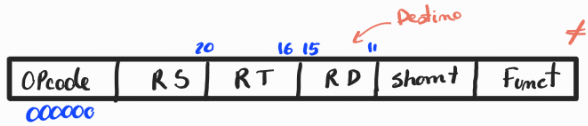
O circuito da cima supõe a instrução de baixo? **Não!**  
ori RT, RS, imm  
ori imediata!

ori → imm é zero-extended  
Não tem lá o caminho mas podemos pô-lo.

add RD, RS, RT

lw RT, im(RS)

subtrai-se e vemos se dá zero  
 beq RS, RT, imm16  
 $BTA = (PC+4) + \text{imm16} \times 4$   
 imm16 << 2



00 = ADD  
 01 = SUB  
 10 = R-Type  
 ↑

Instância	Reg Write	Reg Dest	ALU Src	Branch	Mem Write	Mem to Reg	ALUOP
R-Type	1	1 <sub>15:11</sub>	0	0	0	0	10
lw	1	0 <sub>20:16</sub>	1	0	0	1	00
sw	0	X	1	0	1	X	00
beq	0	X	0	1	0	X	01*
addi	1	0	1	0	0	0	00

addi RT, RS, imm16