Aula27

# µArquitetura Multicycle: III - Exercícios

## *Performance Multicycle*

Caminho Crítico de Execução:

Período Mínimo de Clock

Tempo de Execução

## Instruções Multicycle

ori - *or immediate*

jr e jal - *jump register* e *jump and link*

bne - *branch if not equal*

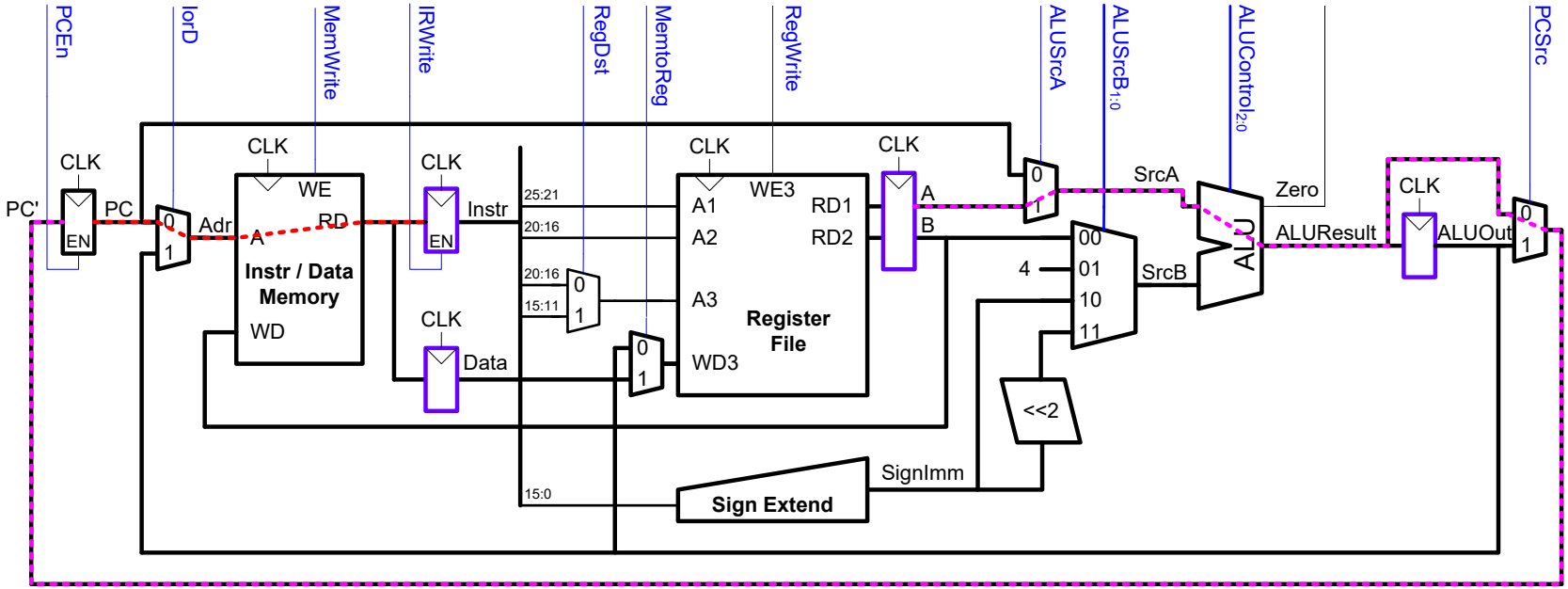
## CPI dum programa

Exemplos de cálculo

# Performance MC (1) - Caminho Crítico: $T_{C,Min}$ (1)

## Caminho Crítico *Multicycle* e $T_{C,Min}$

$$T_{C,Min} = t_{pcq\_PC} + t_{mux} + \max(t_{mem}, t_{ALU} + t_{mux}) + t_{setup}$$




- O CPU *multicycle* foi concebido por forma a que cada ciclo só envolvesse uma operação na ALU ou um acesso à memória ou um acesso ao Banco de Registos.
- Admitamos que o Banco de Registos é mais rápido do que a Memória e que a escrita na memória é mais rápida do que a leitura.
- Analisando o *datapath* podemos identificar dois caminhos críticos que limitam o período mínimo de *clock* ($T_{C,Min}$): a leitura da Memória ou o cálculo na ALU (tipo-R, BTA, etc).

# Performance MC (2) - Caminho Crítico: $T_{C,Min}$ (2)

| Elemento | Parâmetro | Atraso (ps) |
|---|---|---|
| Register clock-to-Q | $t_{pcq\_PC}$ | 30 |
| Register setup | $t_{setup}$ | 20 |
| Multiplexer | $t_{mux}$ | 25 |
| ALU | $t_{ALU}$ | 200 |
| Memory read | $t_{mem}$ | 250 |
| Register file read | $t_{RFread}$ | 150 |
| Register file setup | $t_{RFsetup}$ | 20 |

$$T_{C,Min} = t_{pcq\_PC} + t_{mux} + \max(t_{mem}, t_{ALU} + t_{mux}) + t_{setup}$$
$$= t_{pcq\_PC} + t_{mux} + t_{mem} + t_{setup}$$
$$= [30 + 25 + 250 + 20] \text{ ps}$$
$$= 325 \text{ ps } (F_{C,Max} = 3.08 \text{ GHz})$$

A mesma tecnologia que foi usada no Single-cycle.

# Performance MC (3) - Tempo de Execução (1): CPI

- As instruções requerem um número variável de ciclos:
  - 3 ciclos: beq, j
  - 4 ciclos: R-type, sw, addi
  - 5 ciclos: lw
- O CPI (#Ciclos por Instrução) é uma média pesada.
- Podemos usar este *Benchmark** SPECINT2000, para calcular CPI:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type
  - CPI Médio = (0.11 + 0.02)*3 + (0.52 + 0.10)*4 + 0.25*5 = 4.12

** Benchmark* = Programa de teste.*

# Performance MC (4) - Tempo de Execução (2)

Qual o tempo de execução dum programa* com 100 mil milhões de instruções, num CPU MC, onde cada instrução demora em média 4.12 ciclos (CPI = 4.12), com $T_C$ = 325 ps?

$$\text{Tempo de Execução} = (\#\ \text{instructions}) \times \text{CPI} \times T_{C,Min}$$
$$= (100 \times 10^9)(4.12)(325 \times 10^{-12})$$
$$= 133.9\ \text{seconds}$$

* O mesmo programa que foi usado no Single-cycle.

# Perform MC (5) - Tempo de Execução (3): MC vs SC

Tempo de Execução = 133.9 secs

Mais lento que o CPU Single-cycle (92.5 secs)! Porquê?

- O *overhead* de sequenciação (dum registo) em cada etapa:
  ($t_{setup} + t_{pcq}$= 50 ps) não pode ser ignorado!
- Embora a instrução mais lenta (lw) tenha sido decomposta em ciclos mais curtos, o ciclo do CPU *multicycle* ficou longe de ter sido reduzido de um factor de 5.

  Isto deve-se sobretudo ao *overhead* de sequenciação:
  - no *multicycle* ocorre em todos os ciclos;
  - no *single-cycle* só ocorre no início do ciclo (único).

$T_{c\_SC} / T_{c\_MC}$ = 925/325 = 2.846 < 3x (Só!?)

Embora o exemplo seja excessivamente ruim, chama a atenção para o problema 😐!

# Exercícios MC (1) - Enunciado

Consulte o Apêndice B para as instruções. Copie a Figura 7.27 (*Datapath*) para esboçar as modificações necessárias. Assinale os novos sinais de controlo.
Copie a Tabela 7.39 (*Controlador FSM*) e a Tabela 7.2 (ALU Decoder) para anotar as modificações. Descreva quaisquer outras alterações relevantes.

## Exercício 7.13

Modifique o CPU Multicycle para implementar uma das seguintes instruções:

(a) srlv

(b) ori

(c) xori

(d) jr

(e) jal - extra

## Exercício 7.14

Repita o Exercício 7.13 para as seguintes instruções:

(a) bne

(b) lb

(c) lbu

(d) andi

## Exercícios 7.23 e 7.24

Cálculo do CPI de programas ASM.

# Exercícios MC (2) - ApdxB - OperationCodes - tipo-I

| Opcode | Name | Description |
|---|---|---|
| 000000 (0) | R-type | all R-type instructions |
| 000001 (1) (rt = 0/1) | bltz rs, label / bgez rs, label | branch less than zero/branch greater than or equal to zero |
| 000010 (2) | j label | jump |
| 000011 (3) | jal label | jump and link |
| 000100 (4) | beq rs, rt, label | branch if equal |
| 000101 (5) | bne rs, rt, label | branch if not equal |
| 000110 (6) | blez rs, label | branch if less than or equal to zero |
| 000111 (7) | bgtz rs, label | branch if greater than zero |
| 001000 (8) | addi rt, rs, imm | add immediate |
| 001001 (9) | addiu rt, rs, imm | add immediate unsigned |
| 001010 (10) | slti rt, rs, imm | set less than immediate |
| 001011 (11) | sltiu rt, rs, imm | set less than immediate unsigned |
| 001100 (12) | andi rt, rs, imm | and immediate |
| 001101 (13) | ori rt, rs, imm | or immediate |
| 001110 (14) | xori rt, rs, imm | xor immediate |
| 001111 (15) | lui rt, imm | load upper immediate |
| 010000 (16) (rs = 0/4) | mfc0 rt, rd / mtc0 rt, rd | move from/to coprocessor 0 |
| 010001 (17) | F-type | fop = 16/17: F-type instructions |
| 010001 (17) (rt = 0/1) | bc1f label / bc1t label | fop = 8: branch if fpcond is FALSE/TRUE |

| Opcode | Name | Description |
|---|---|---|
| 011100 (28) (func = 2) | mul rd, rs, rt | multiply (32-bit result) |
| 100000 (32) | lb rt, imm(rs) | load byte |
| 100001 (33) | lh rt, imm(rs) | load halfword |
| 100011 (35) | lw rt, imm(rs) | load word |
| 100100 (36) | lbu rt, imm(rs) | load byte unsigned |
| 100101 (37) | lhu rt, imm(rs) | load halfword unsigned |
| 101000 (40) | sb rt, imm(rs) | store byte |
| 101001 (41) | sh rt, imm(rs) | store halfword |
| 101011 (43) | sw rt, imm(rs) | store word |
| 110001 (49) | lwc1 ft, imm(rs) | load word to FP coprocessor 1 |
| 111001 (56) | swc1 ft, imm(rs) | store word to FP coprocessor 1 |

**Single-Cycle:** lui, slti, jal

**Multicycle:** bne, ori, jal

# Exercícios MC (2) - ApdxB - FunctionCodes - tipo-R

Table B.2 R-type instructions, sorted by funct f

| Funct | Name | Description |
|---|---|---|
| 000000 (0) | sll rd, rt, shamt | shift left logical |
| 000010 (2) | srl rd, rt, shamt | shift right logical |
| 000011 (3) | sra rd, rt, shamt | shift right arithmetic |
| 000100 (4) | sllv rd, rt, rs | shift left logical variable |
| 000110 (6) | srlv rd, rt, rs | shift right logical variable |
| 000111 (7) | srav rd, rt, rs | shift right arithmetic variable |
| 001000 (8) | jr rs | jump register |
| 001001 (9) | jalr rs | jump and link register |
| 001100 (12) | syscall | system call |
| 001101 (13) | break | break |
| 010000 (16) | mfhi rd | move from hi |
| 010001 (17) | mthi rs | move to hi |
| 010010 (18) | mflo rd | move from lo |
| 010011 (19) | mtlo rs | move to lo |
| 011000 (24) | mult rs, rt | multiply |
| 011001 (25) | multu rs, rt | multiply unsigned |
| 011010 (26) | div rs, rt | divide |
| 011011 (27) | divu rs, rt | divide unsigned |

Table B.2 R-type instructions, sorted by funct field-

| Funct | Name | Description |
|---|---|---|
| 100000 (32) | add rd, rs, rt | add |
| 100001 (33) | addu rd, rs, rt | add unsigned |
| 100010 (34) | sub rd, rs, rt | subtract |
| 100011 (35) | subu rd, rs, rt | subtract unsigned |
| 100100 (36) | and rd, rs, rt | and |
| 100101 (37) | or rd, rs, rt | or |
| 100110 (38) | xor rd, rs, rt | xor |
| 100111 (39) | nor rd, rs, rt | nor |
| 101010 (42) | slt rd, rs, rt | set less than |
| 101011 (43) | sltu rd, rs, rt | set less than unsigned |

**Single-Cycle:** jr, sll

**Multicycle:** jr

# Exercícios MC (3) - Figura 7.27 - Datapath

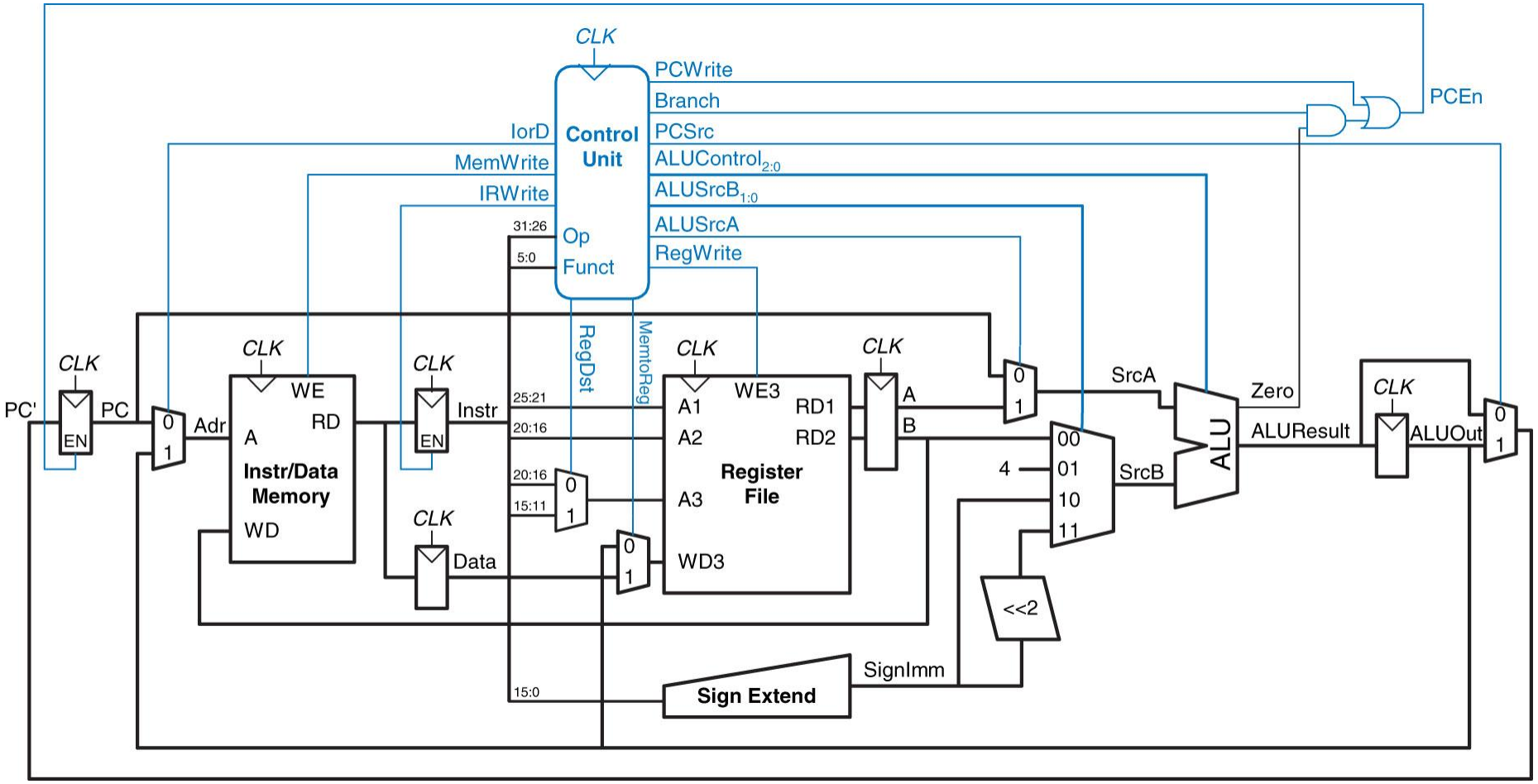



Figura 7.27 Processador MIPS *multicycle* completo*

*Mas, sem suporte para 'jump'.

# Exercícios MC (4) - Controlador FSM

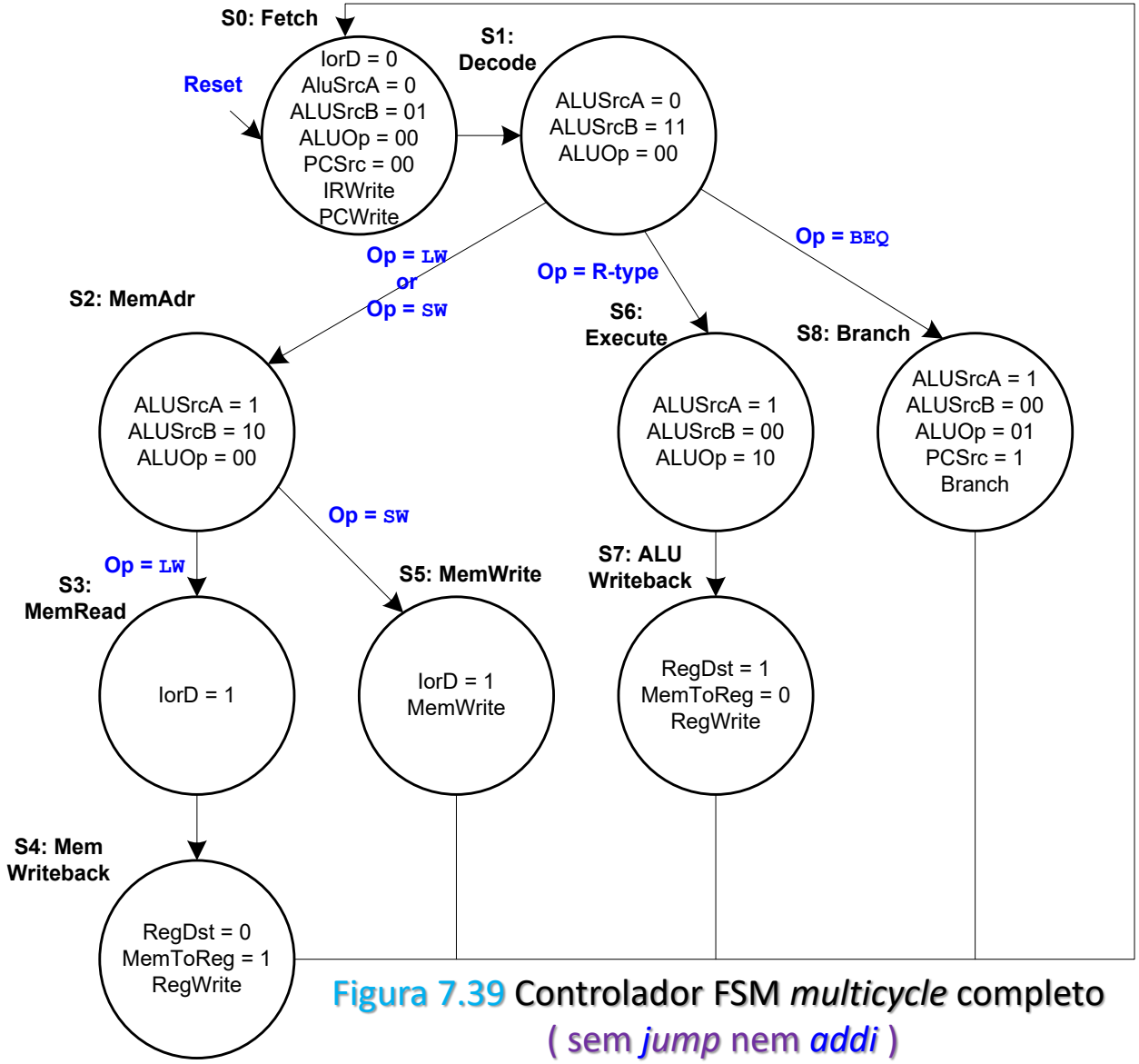


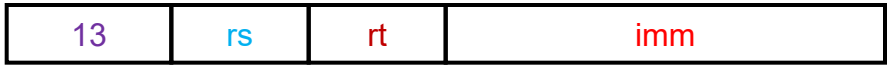

Figura 7.39 Controlador FSM *multicycle* completo
( sem *jump* nem *addi* )

# Exercícios MC (5) - ALU Decoder

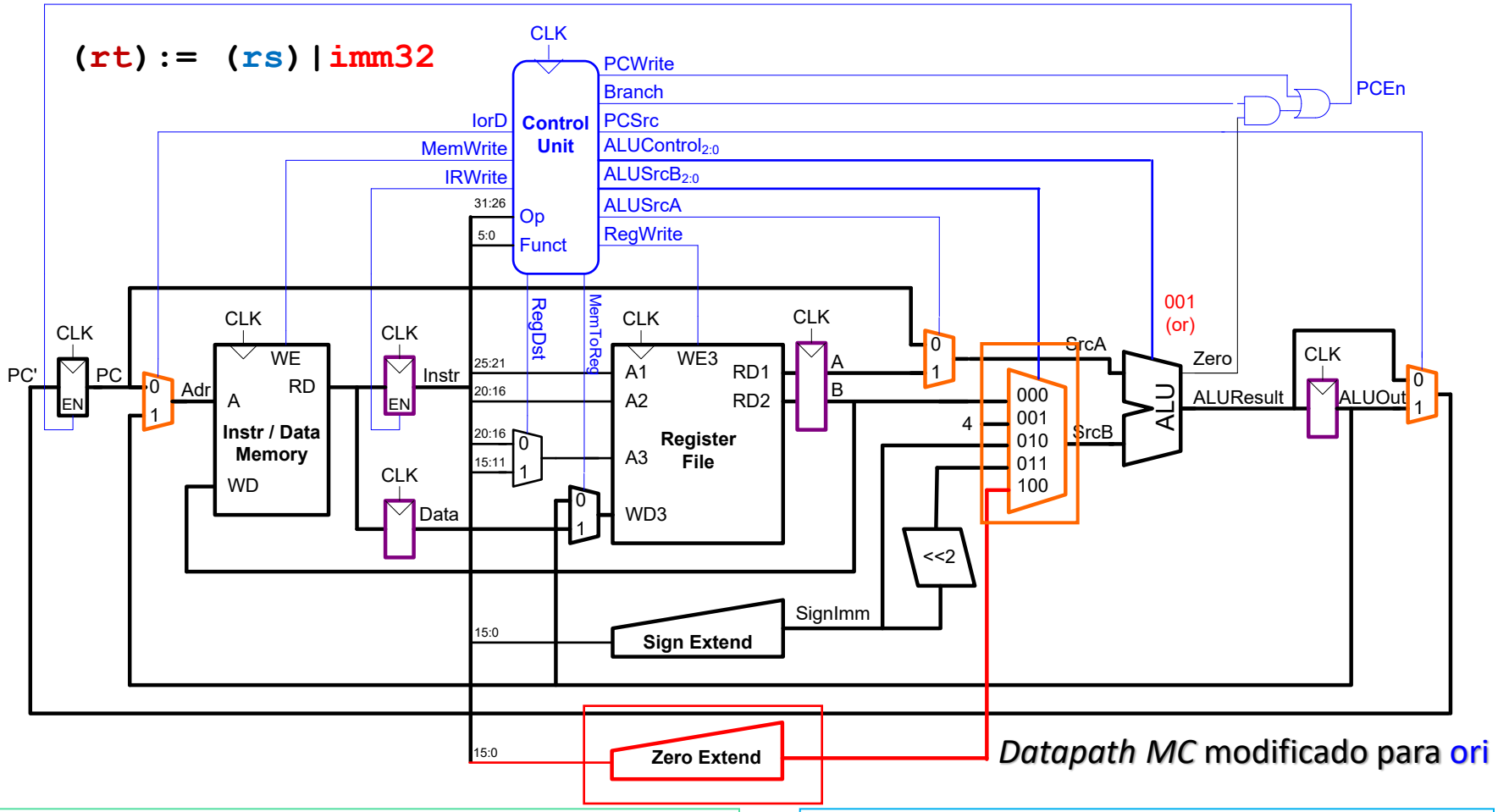
| $ALUOp_{1:0}$ | $Funct_{5:0}$ | $ALUControl_{2:0}$ |
|---|---|---|
| 00 | X | 010 (Add) |
| X1 | X | 110 (Subtract) |
| 1X | 100000 (add) | 010 (Add) |
| 1X | 100010 (sub) | 110 (Subtract) |
| 1X | 100100 (and) | 000 (And) |
| 1X | 100101 (or) | 001 (Or) |
| 1X | 100110 (xor) | 100 (Xor) |
| 1X | 100111 (nor) | 101 (Nor) |
| 1X | 101010 (slt) | 111 (Slt) |

Tabela 7.2 Tabela de verdade do *ALU decoder* (com *xor* e *nor*)

# P7.13b (1) - ori : Datapath




`ori rt,rs,imm` **tipo-I**




*Datapath MC* modificado para ori

**1.** Adiciona-se uma unidade de zero extension
**2.** O mux *ALUSrcB* passa de 4 para 5 entradas

O *ALU decoder* e o controlador FSM também precisam de ser modificados ->...

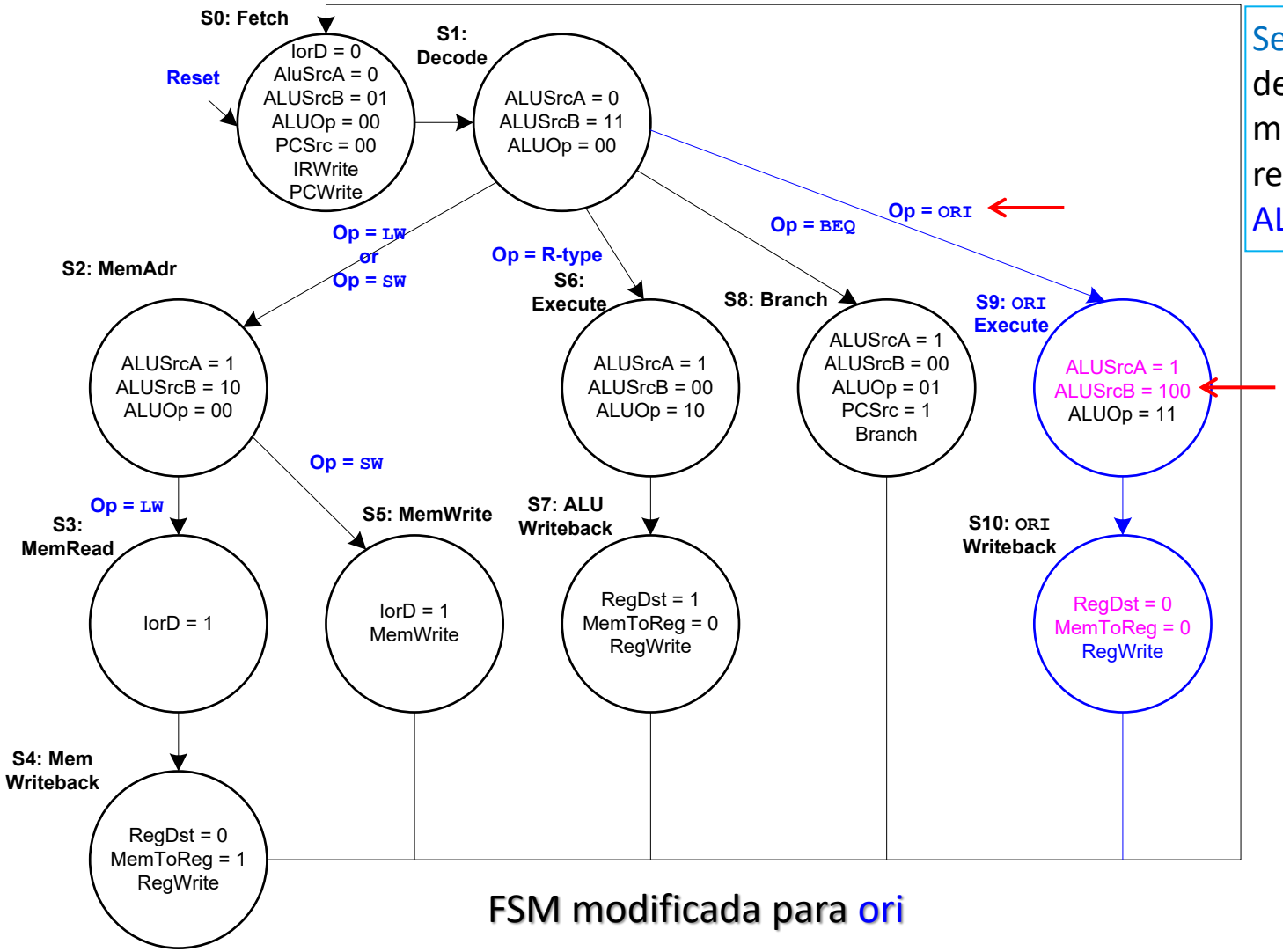
# P7.13b (2) - ori : ALU Decoder


| $ALUOp_{1:0}$ | $Funct_{5:0}$ | $ALUControl_{2:0}$ |
|---|---|---|
| 00 | X | 010 (Add) |
| 01 | X | 110 (Subtract) |
| 10 | 100000 (add) | 010 (Add) |
| 10 | 100010 (sub) | 110 (Subtract) |
| 10 | 100100 (and) | 000 (And) |
| 10 | 100101 (or) | 001 (Or) |
| 10 | 100110 (xor) | 100 (Xor) |
| 10 | 100111 (nor) | 101 (Nor) |
| 10 | 101010 (slt) | 111 (Slt) |
| 11 | X | 001 (Or) ← |

Tabela de verdade do *ALU decoder* para ori

ori é uma instrução do tipo-I, por isso precisamos de criar um novo código ALUOp = 11.
(Recorde-se que para addi não foi necessário porque o código ALUOp = 00 já existia).
No caso das instruções do tipo-R não é necessário dado que para todas ALUOp = 10.

# P7.13b (3) - ori : Controlador FSM

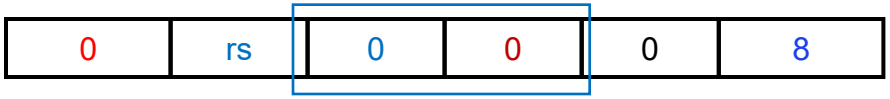



Semelhante ao caso de addi (*vd* aula MC1), mas com valores diferentes para ALUSrcB e ALUOp

FSM modificada para ori

# P7.13d (1) - jr : ALU Decoder

jr é uma instrução do tipo-R, com a particularidade de ambos rt e rd serem zero!



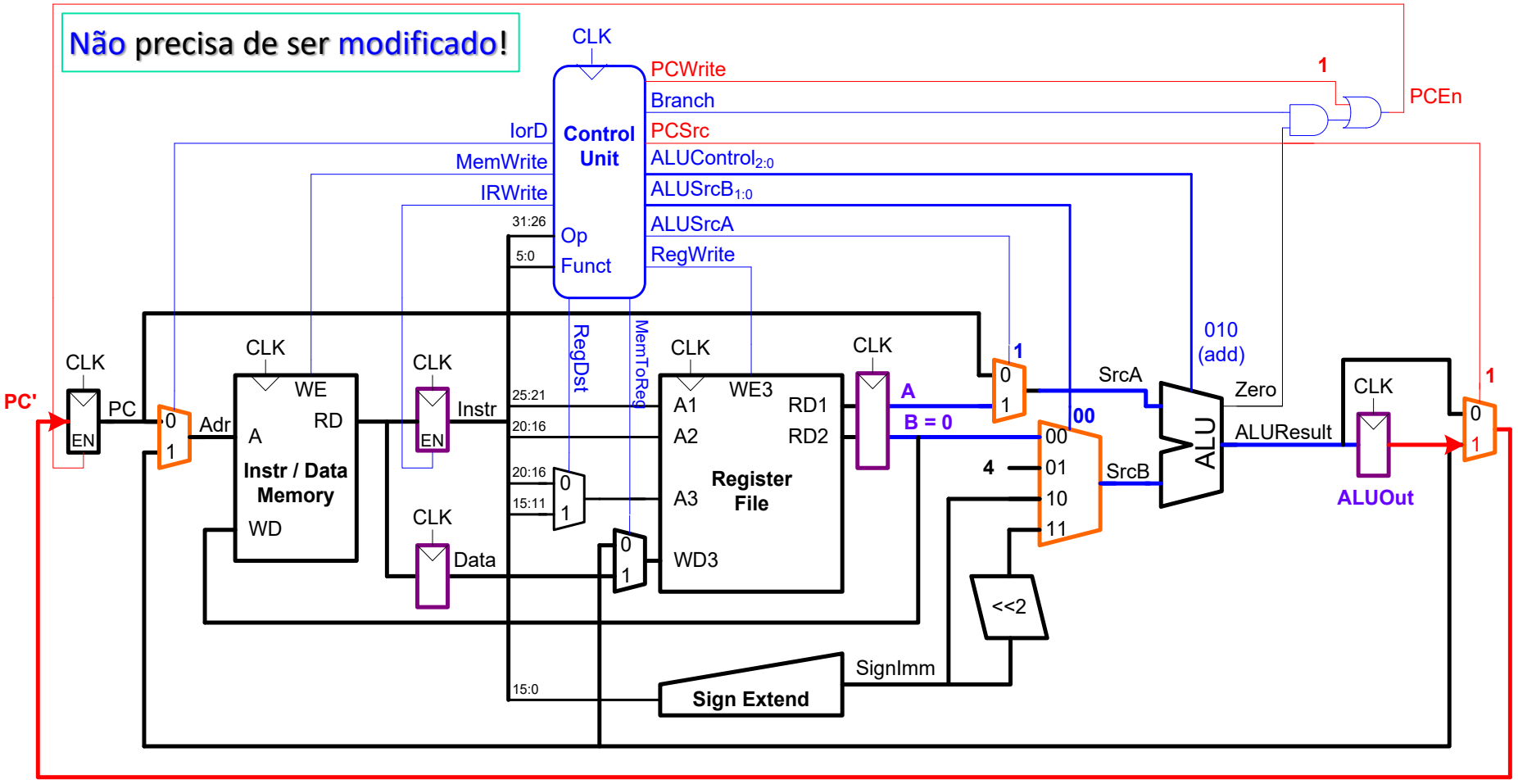

Precisamos só de adicionar uma linha para o respectivo código $Funct_{5:0}$, ao qual fazemos corresponder a saída $ALUControl_{2:0}$ igual a Add.

ALUOut = **A** + **B** (=0)

| $ALUOp_{1:0}$ | $Funct_{5:0}$ | $ALUControl_{2:0}$ |
|---|---|---|
| 00 | X | 010 (Add) |
| 01 | X | 110 (Subtract) |
| 10 | 100000 (add) | 010 (Add) |
| 10 | 100010 (sub) | 110 (Subtract) |
| 10 | 100100 (and) | 000 (And) |
| 10 | 100101 (or) | 001 (Or) |
| 10 | 100110 (xor) | 100 (Xor) |
| 10 | 100111 (nor) | 101 (Nor) |
| 10 | 101010 (slt) | 111 (Slt) |
| 10 | 001000 (jr) | 010 (Add) |

Tabela de verdade do *ALU decoder* para jr

# P7.13d (2) - jr : Datapath




S6 (Exec):

ALUResult = A + B(=0)

S6: Execute
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

S9 (JR):

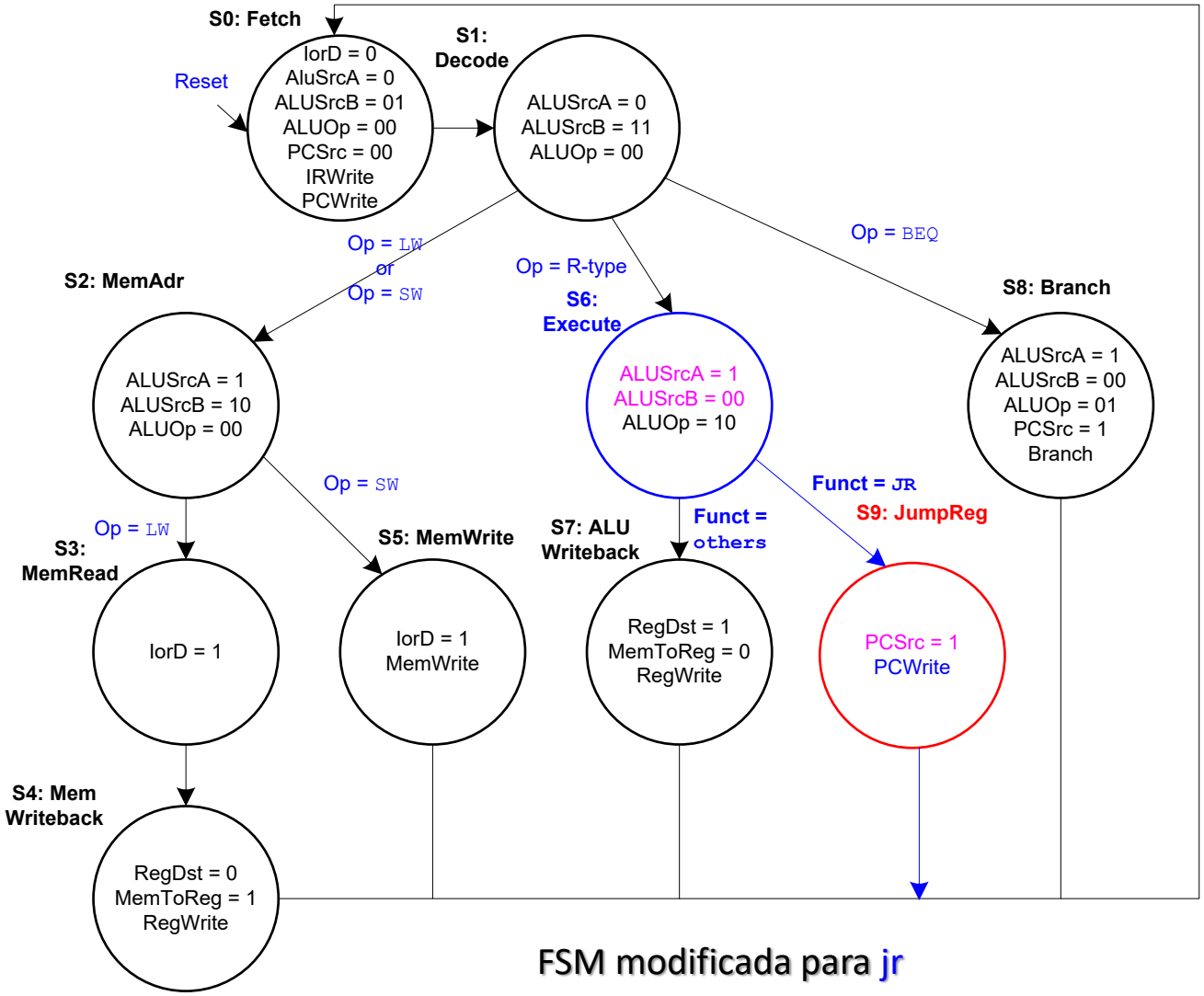
PC' := ALUOut (=A)

S9: JumpReg
PCSrc = 1
PCWrite

| 0 | rs | 0 | 0 | 0 | 8 | jr rs |
|---|---|---|---|---|---|---|

Recorde-se que rt é zero!

# P7.13d (3) - jr : Controlador FSM




FSM modificada para jr

**S6:** ALUResult = **A** + $0

**S9:** PC' := ALUOut = A

**S6:** (Instruções do tipo-R) Deteta-se o código de Função de jr e transita-se para o estado S9.

**S9:** Realiza-se a operação PC': = ALUOut = A, a qual precisa dos sinais PCSrc = 1 e PCWrite.

# PExtra (1) - jal : Datapath

P: Modifique o CPU, cujos *datapath* e diagrama de estados do controlador FSM estão indicados nas Figura 7.41 e Figura 7.42, respectivamente, para implementar a instrução jal.

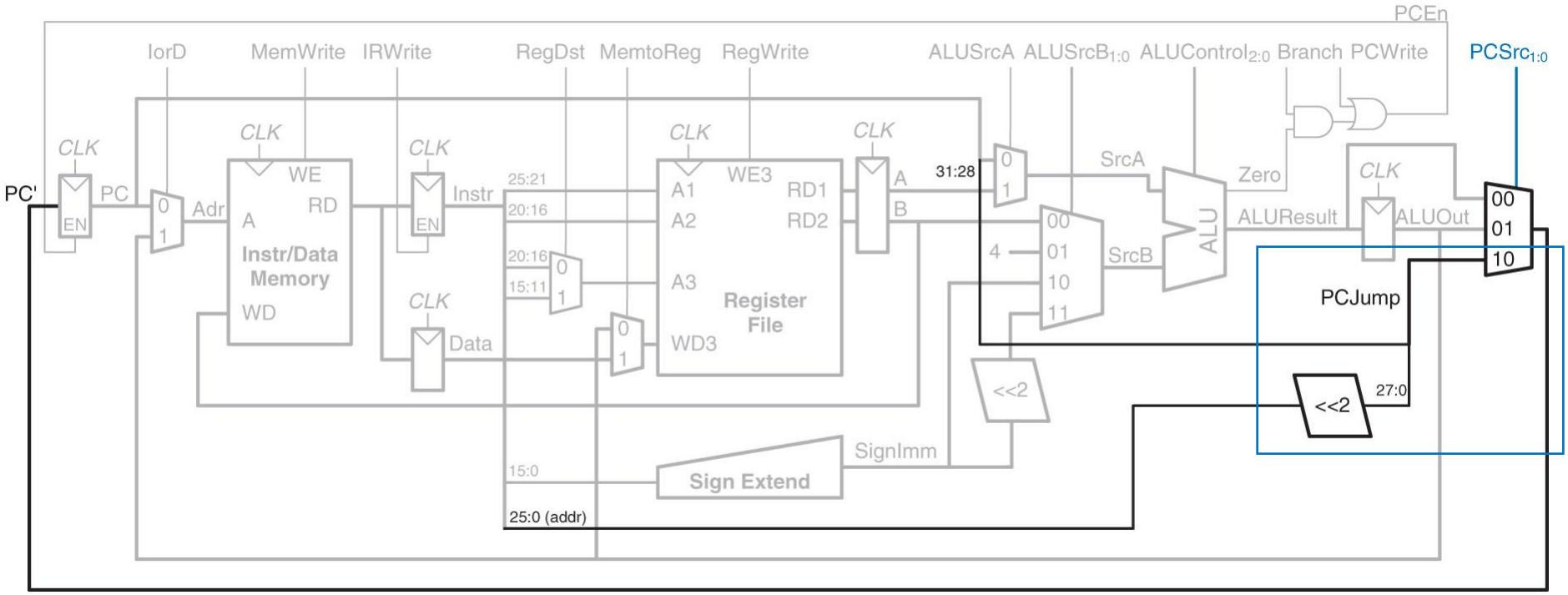



Figura 7.41 Processador MIPS *multicycle* com a extensão j

# PExtra (2) - jal : Datapath

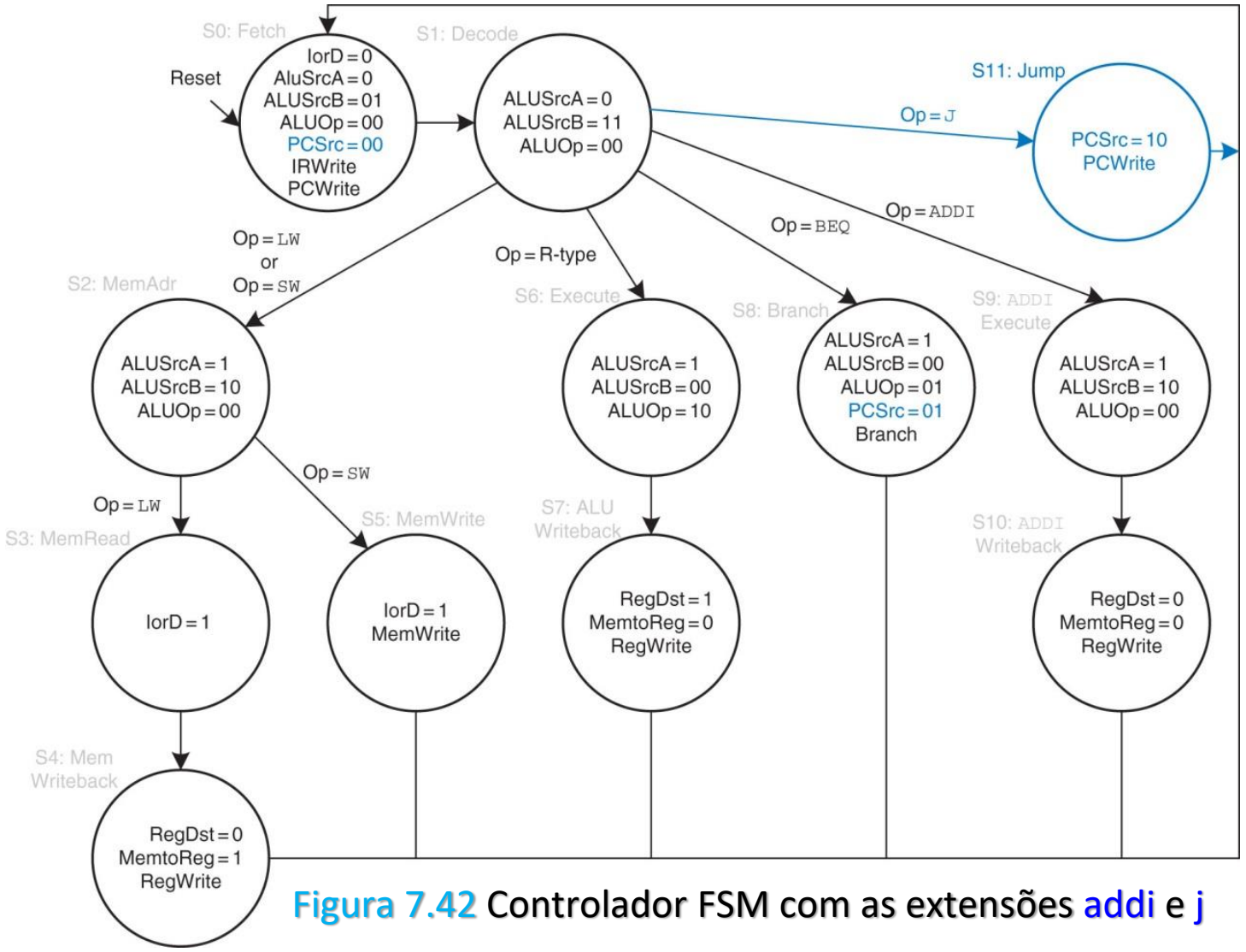



Figura 7.42 Controlador FSM com as extensões addi e j

# PExtra (3) - jal : Datapath

P: Modifique o CPU, cujos *datapath* e diagrama de estados do controlador FSM estão indicados nas Figura 7.41 e Figura 7.42, respectivamente, para implementar a instrução jal.

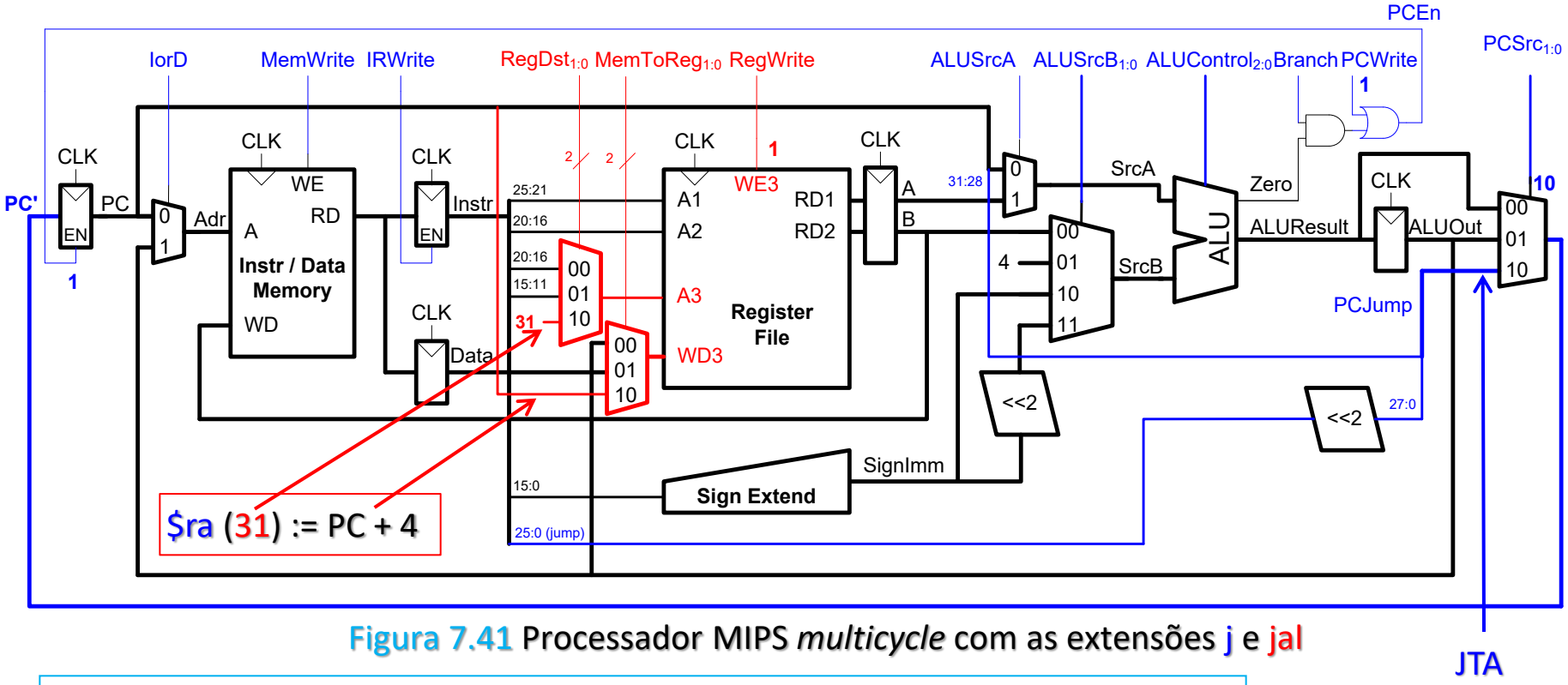
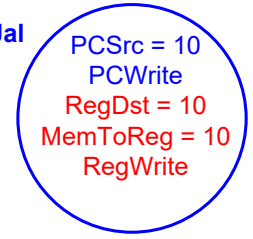



Figura 7.41 Processador MIPS *multicycle* com as extensões j e jal

1. Adiciona-se uma terceira entrada ao mux $RegDst_{1:0}$, para 31.
2. Adiciona-se uma terceira entrada ao mux $MemToReg_{1:0}$, para 'PC + 4'
3. Adiciona-se um estado extra (S12) à FSM para jal.
   São usados *datapath* e FSM já com *jump* incluído.



# PExtra (4) - jal : Controlador FSM

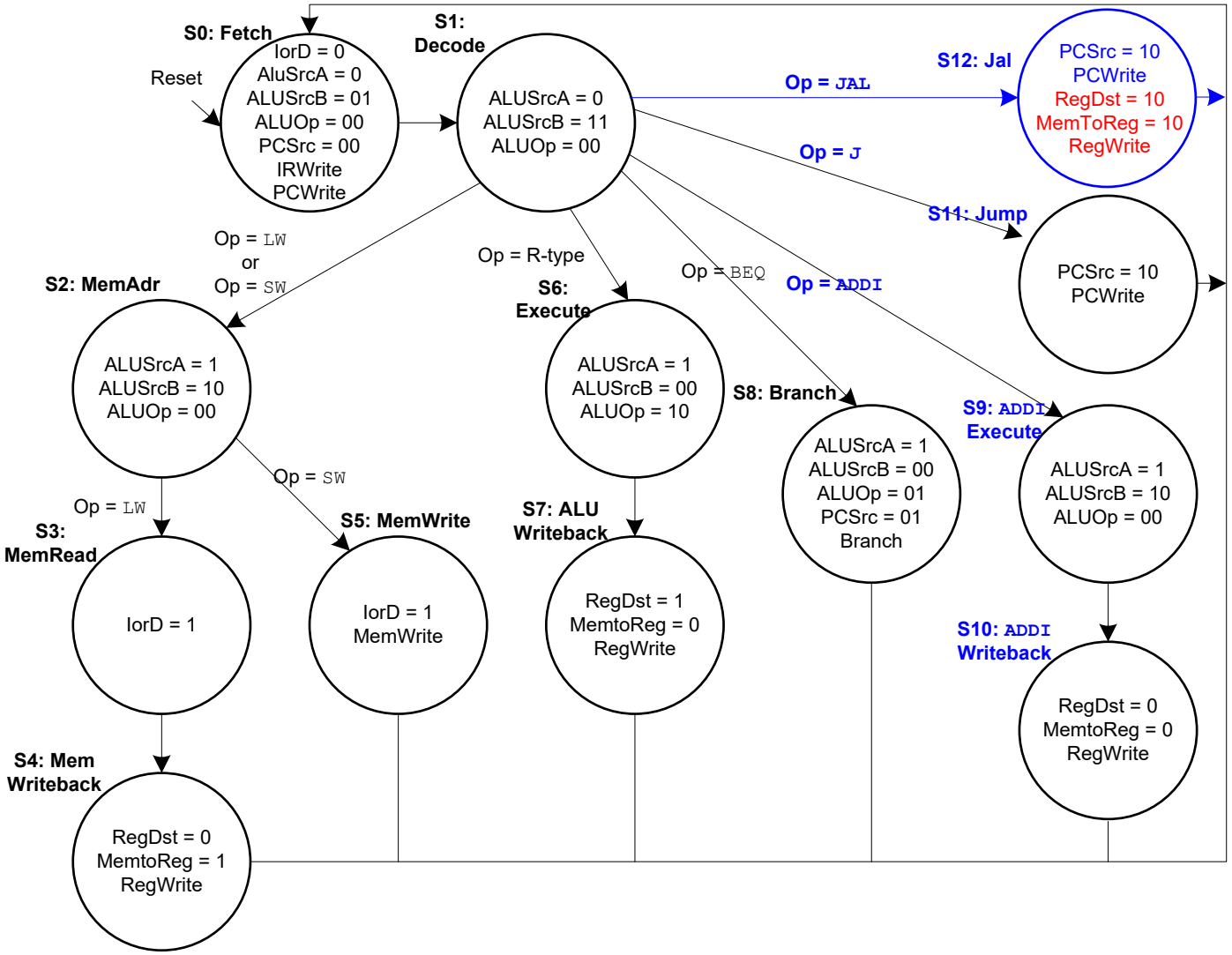
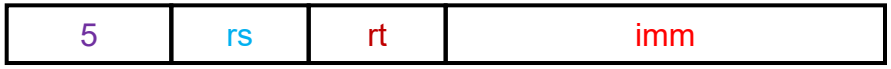
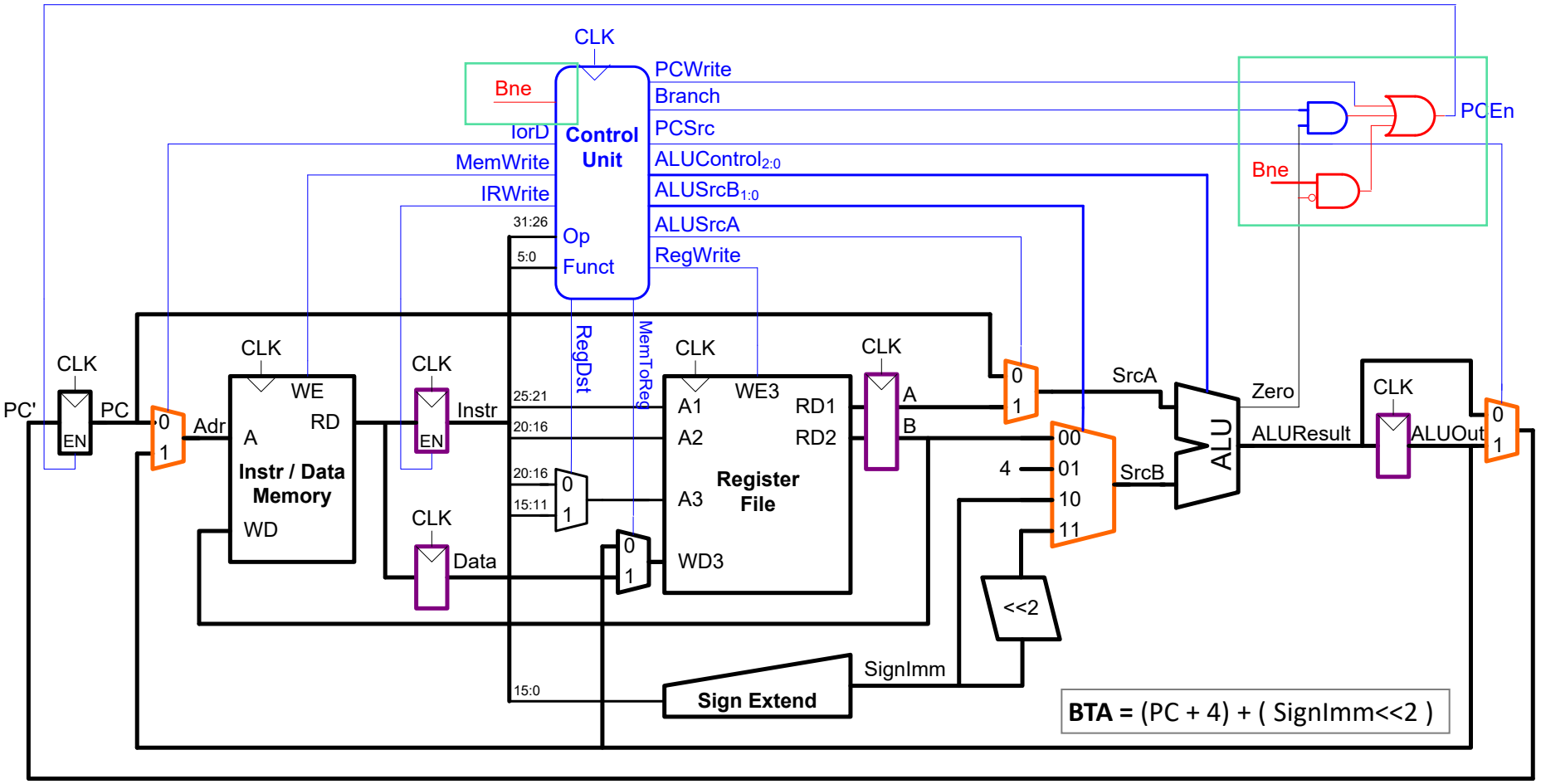



Figura 7.42 Controlador FSM com as extensões addi, j e jal

# P7.14a (1) - bne : Datapath




`bne rt,rs,imm` **tipo-I**

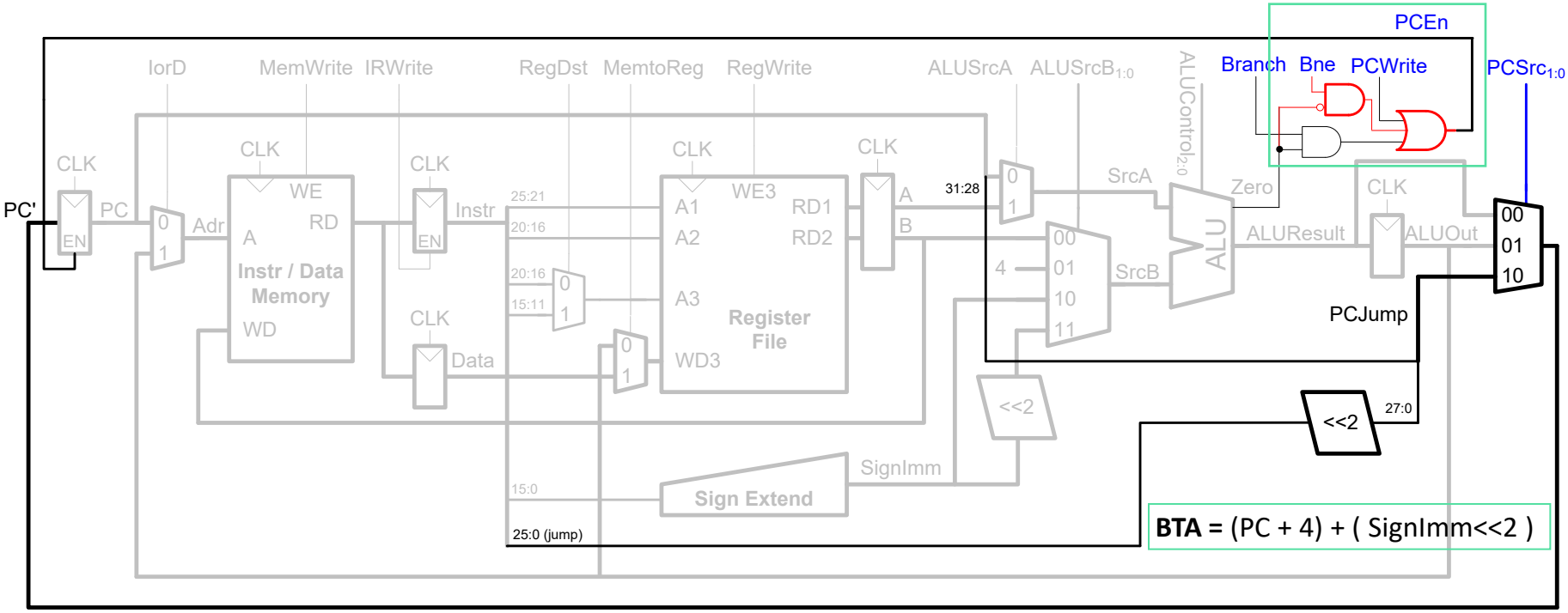



1. Adiciona-se uma *gate* and extra com entradas Bne e not-Zero .
2. Modifica-se a *gate* or (PCEn) para ter 3 entradas.

3. Adiciona-se um estado extra (S9) à FSM para bne.

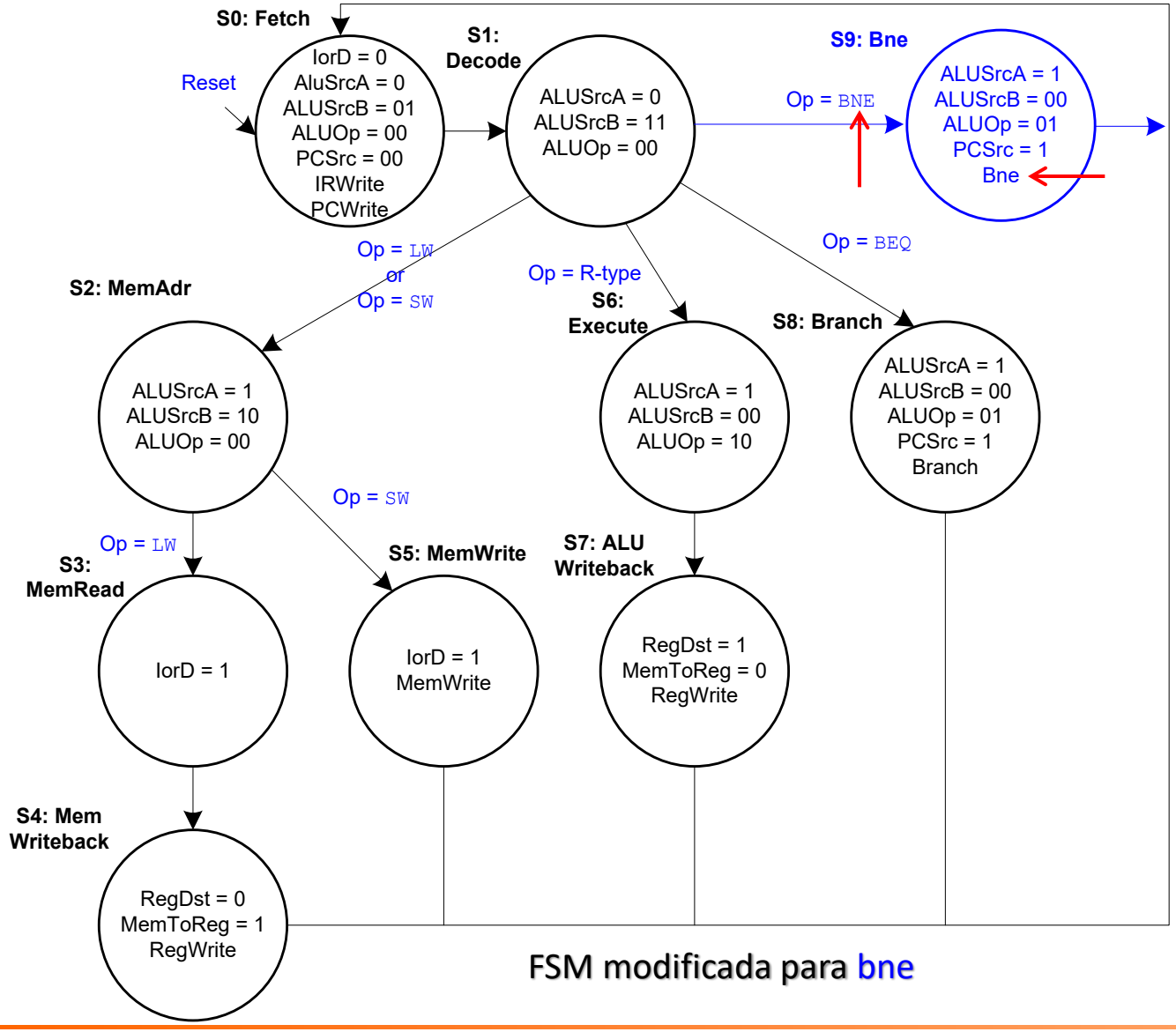
# P7.14a (1) - bne : Datapath, v2


`bne rt,rs,imm` **tipo-I**




1. Adiciona-se uma *gate* and extra com entradas Bne e not-Zero.
2. Modifica-se a *gate* or (PCEn) para ter 3 entradas.
3. Adiciona-se um estado extra (S9) à FSM para bne.

# P7.14a (2) - bne : Controlador FSM




FSM modificada para bne

Semelhante à beq (Branch), mas agora descodifica o Opcode = BNE e gera o sinal Bne.

```
bne rs,rt,label
```

# P7.23 (1) - CPI : P1

**a**) Quanto ciclos são necessários para executar o seguinte programa num processador MIPS multicycle?

**b**) Qual é o valor do CPI deste programa (CPI = #Ciclos/#Intruções)?

```
            .text
            addi   $s0, $0, 5       # result = 5
while:      beq    $s0, $0, done    # while (result > 0) {
            addi   $s0, $s0, -1     # result = result - 1;
            j      while            # }
done:
```

# P7.23 (2) - CPI : P1

```
# P 7.23
# a) How many clock cycles are required to run the
# following program on a multicycle MIPS processor?
#  A: 4 + 5 x (3 + 4 + 3) + 3 = 57 cycles
# b) What is the CPI of this program?
#  A: The #instructions executed is 1 + 5 x 3 + 1 = 17
#  Thus, CPI = 57 cycles / 17 instructions = 3.35

                .text
                addi   $s0, $0, 5       # addi = 4 cycles
                # beq = 3, addi = 4, j = 3;
                # num_cycles: 5 x( 3 + 4 + 3) + 3 = 53
                # 5 times through the loop + last 'beq'
        while:  beq    $s0, $0, done    # while (result > 0){
                addi   $s0, $s0, -1     # result = result-1;
                j      while            # }
        done:
```

# P7.24 (1) - CPI : P2

**a**) Quanto ciclos são necessários para executar o seguinte programa num processador MIPS multicycle?

**b**) Qual é o valor do CPI deste programa?

```
        .text
        add    $s0, $0, $0   # i = 0
        add    $s1, $0, $0   # sum = 0
        addi   $t0, $0, 10   # $t0 = 10
loop:   slt    $t1, $s0,$t0  # $t1 = (i < 10)? 1:0
        beq    $t1, $0, done # if (i >= 10) goto done
        add    $s1, $s1,$s0  # sum = sum + i
        addi   $s0, $s0, 1   # i++
        j      loop
done:
```

# P7.24 (2) - CPI : P2

```
# P 7.24 Repeat Exercise 7.23 for the following program.
# a) How many cycles are required to run the following
# program on a multicycle MIPS processor?
# A: 3x4 + 10x(4 + 3 + 4 + 4 + 3)+ 4 + 3 = 12 + 180 + 7 = 199
# b) What is the CPI of this program?
# A: The #instructions executed is: 3 + 10 x 5 + 2 = 55
# Thus, CPI = 199 cycles / 55 instructions = 3.62

            .text
            add    $s0, $0, $0    # i = 0       (4)
            add    $s1, $0, $0    # sum = 0     (4)
            addi   $t0, $0, 10    # $t0 = 10    (4)
            # 10 times through the loop + last slt + beq
            # 10x(4 + 3 + 4 + 4 + 3) + 4 + 3 = 187
loop:       slt    $t1, $s0, $t0 # $t1 = (i < 10)? 1:0
            beq    $t1, $0, done # if (i >= 10) goto done
            add    $s1, $s1, $s0 # sum = sum + i
            addi   $s0, $s0, 1    # i++
            j      loop
done:
```

# Datapath do µP MIPS: Multicycle

| Aula | Data | Descrição |
|---|---|---|
| 25 | 22/Mai (6ª) | *Multicycle*: Limitações das arquiteturas *single-cycle*; Versão de referência duma arquitetura *multicycle*;<br>Exemplos do processamento das instruções numa arquitetura *multicycle*. |
| 26 | 27/Mai | Unidade de Controlo *para datapath multicycle*: diagrama de estados. Sinais de controlo e valores do *datapath multicycle*. Exemplos da execução sequencial de algumas instruções *no datapath multicycle*. |
| 27 | 29/Mai (6ª) | Resolução de problemas sobre µArquiteturas *Multicycle*. |
| 28 | 03/Jun | Revisões sobre µArquiteturas *Single-cycle* e *Multicycle*. |
| 29 | 05/Jun (6ª) | Mais Revisões. |