

μArquitetura MIPS: Single-cycle - II

Unidade de Controlo

Entradas e Saídas

Descodificador da ALU

Exemplo de ALU

Descodificador Principal

Instruções de tipo-R: **or**

Instruções de tipo-I: **lw/sw** e **beq**

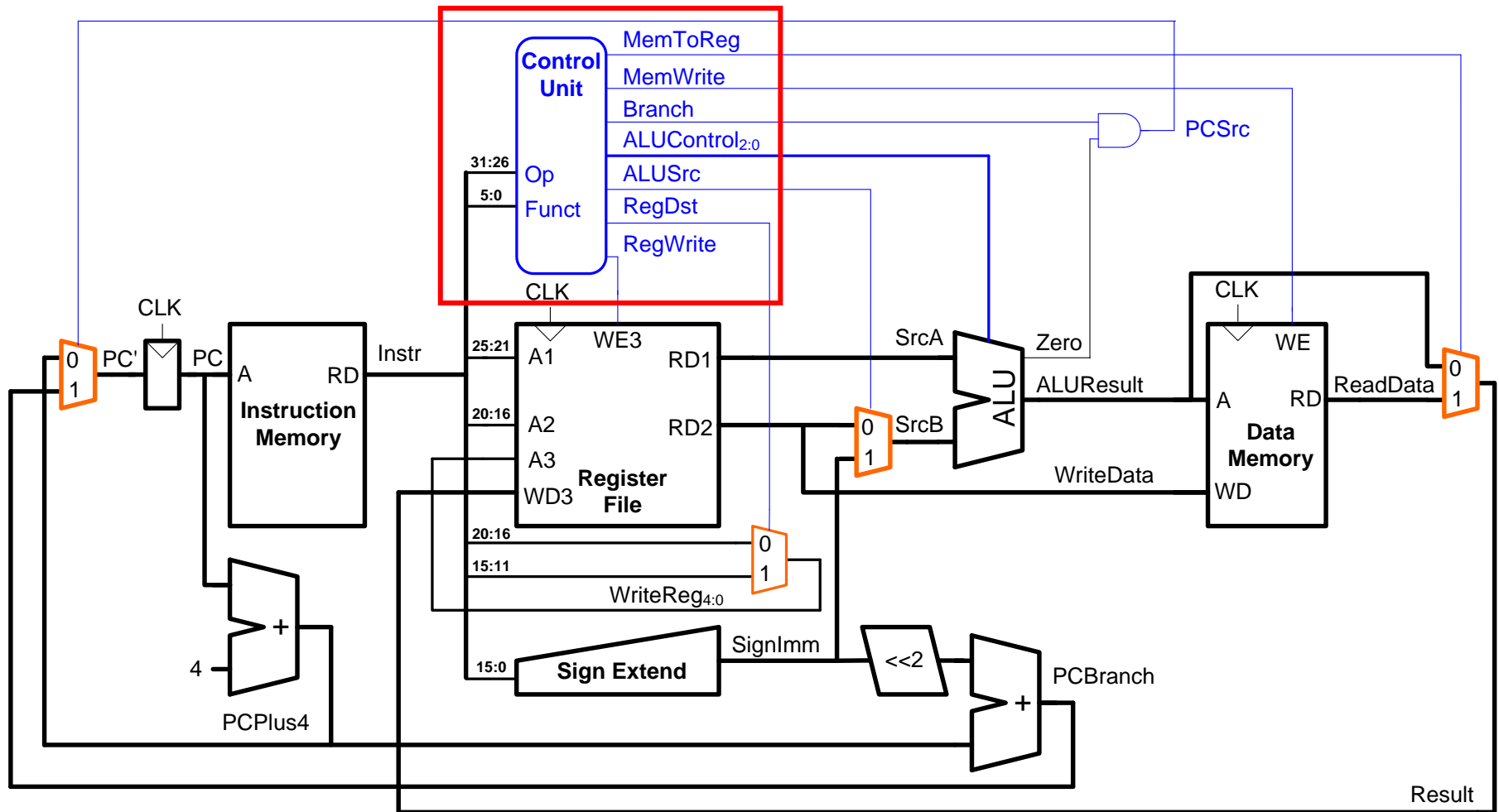
Exercício

Execução da instrução **or**

Suporte para instruções Adicionais

addi e **jump**

Datapath Single-Cycle (SC) - Unidade de Controlo

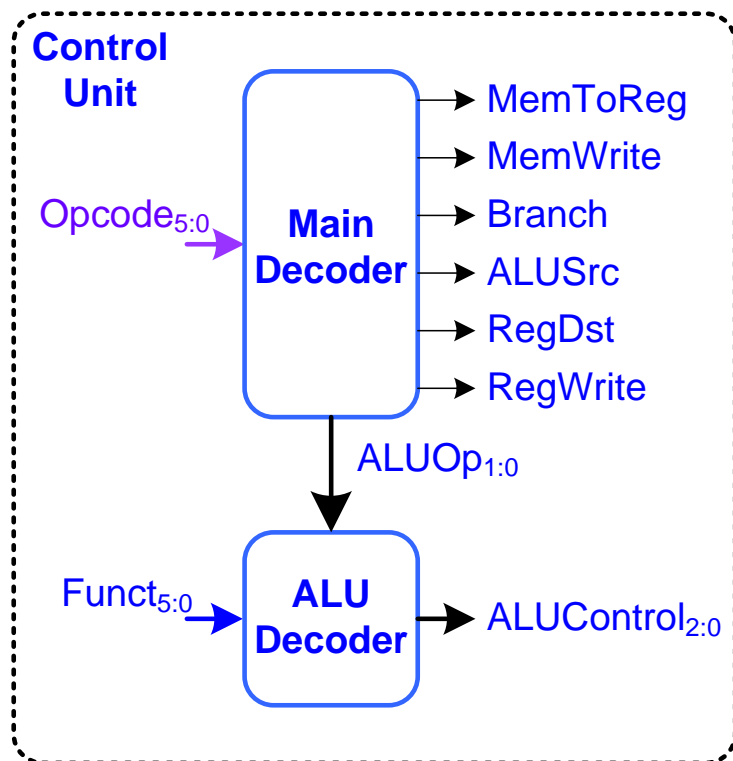


ISA limitado a: lw, sw, tipo-R e beq

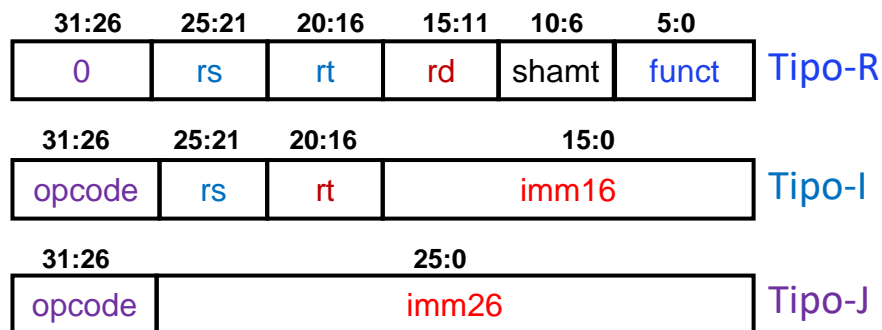
Unidade de Controlo (1) - Entradas e Saídas

A Unidade de Controlo (UC)

Gera os sinais de controlo do *datapath*, usando como **entradas** os bits de **opcode** e de **funct** da instrução.



- A maior parte das **saídas** de controlo é derivada do **opcode**; as instruções do tipo-R precisam **ainda** de usar o campo **funct** para determinar a operação da ALU.

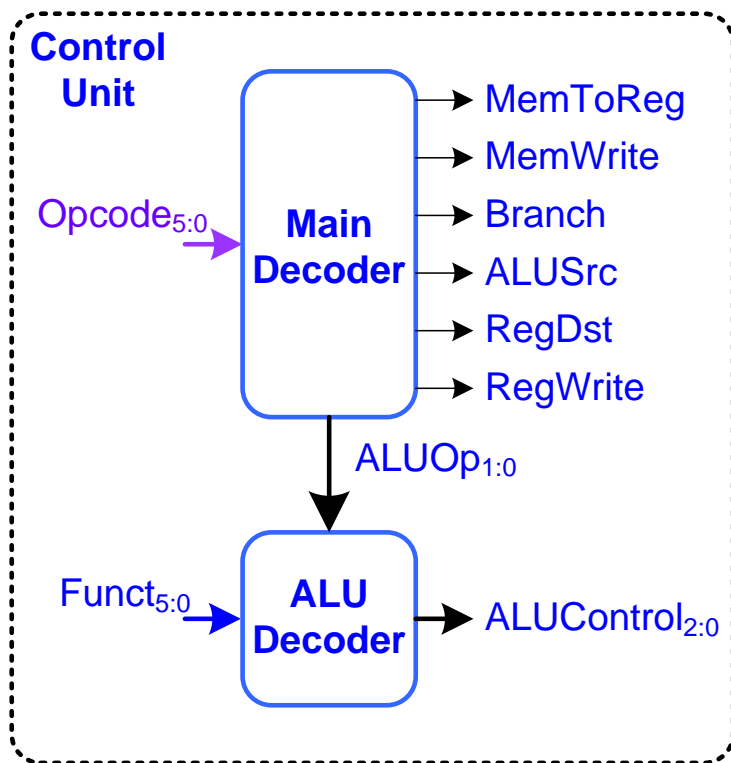


Retomamos a matéria dada na 1ª aula sobre Assembly (codificação de instruções).

Unidade de Controlo (2) - Decoders: Main + ALU

A Unidade de Controlo

Está dividida em dois* blocos de lógica combinatória:

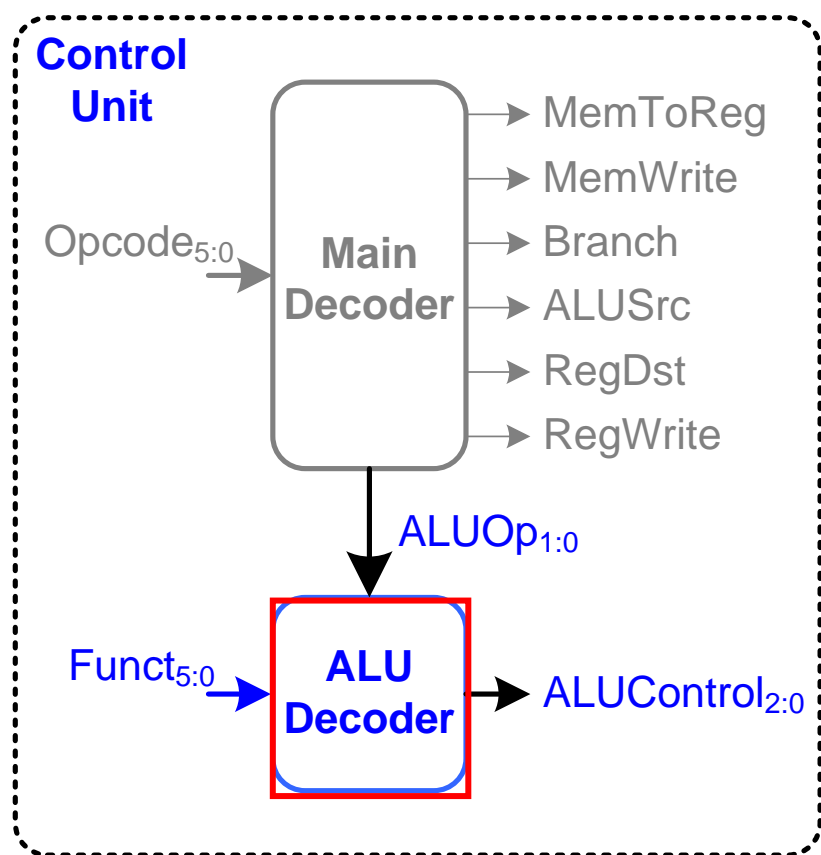


- O *Main Decoder* usa o Opcode_{5:0} da instrução, para gerar todos os sinais de saída e ainda o código de operação da ALU (ALUOp_{1:0}).
- O *ALU Decoder* usa o ALUOp_{1:0} juntamente com o campo Funct_{5:0} para gerar o sinal ALUControl_{2:0}, o qual controla a operação da ALU.

*Para simplificar o projeto.

UC (3) - ALU Decoder (1) - Entrada ALUOp

O *ALU Decoder* é a parte da UC responsável por decodificar o campo da instrução $\text{Funct}_{5:0}$. Usa o sinal ALUOp gerado pelo *Main Decoder*, como entrada auxiliar.



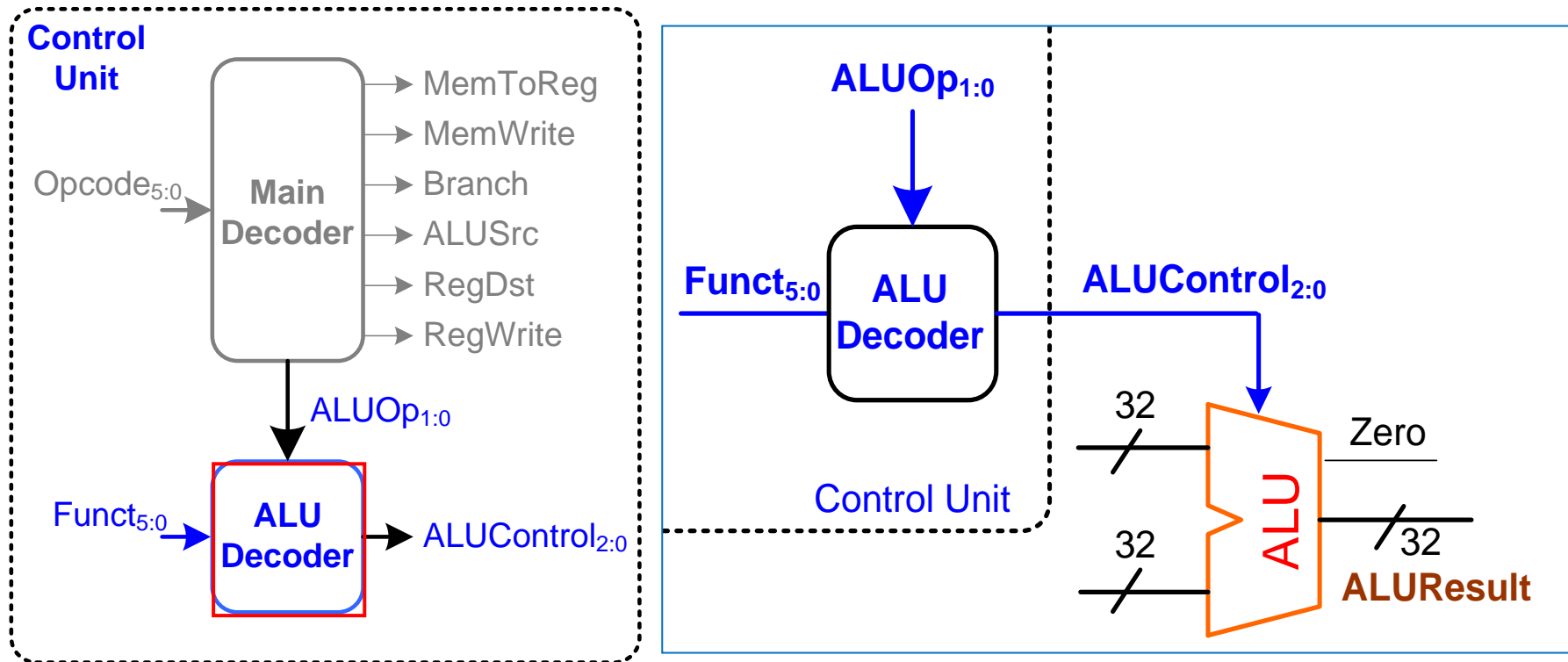
- Os valores do *bus* $\text{ALUOp}_{1:0}$ têm o seguinte significado:

$\text{ALUOp}_{1:0}$	Descrição
00	Add
01	Subtract
10	Look at $\text{Funct}_{5:0}$
11	Not Used (yet)

Tipo-R

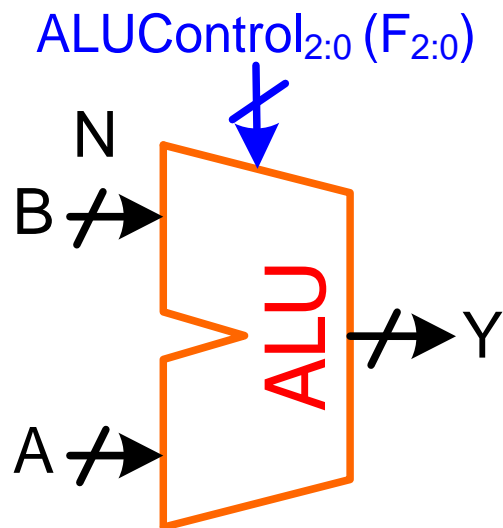
UC (4) - ALU Decoder (2) - Saída ALUControl

- O *ALU Decoder* gera o bus $ALUControl_{2:0}$ o qual controla a operação da **ALU**.

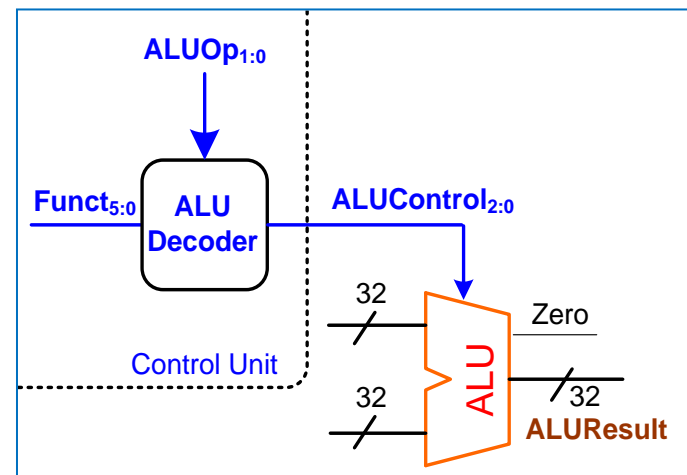


UC (5) - ALU Decoder (3) - Exemplo de ALU

A **ALU** (Arithmetic and Logic Unit) é a unidade combinatória que implementa as funções aritméticas e lógicas.



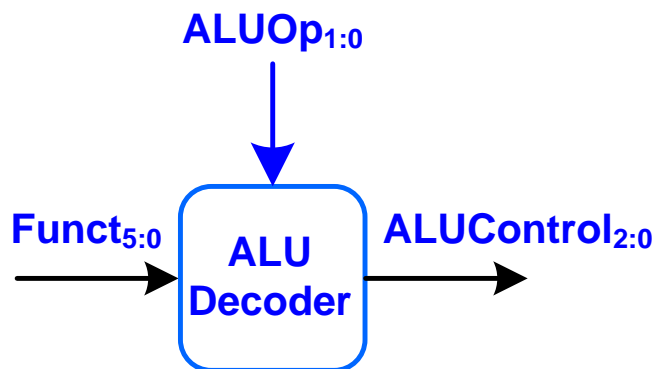
F _{2:0}	Função (Y)
000	A & B
001	A B
010	A + B
011	not used
100	A ^ B
101	~(A B)
110	A - B
111	SLT



A ALU pode facilmente ser implementada em linguagens de descrição de hardware (HDL) do tipo Verilog ou VHDL.

UC (6) - ALU Decoder (4) - Tabela Verdade

- Quando **ALUOp** é **00** ou **01**, a **ALU** soma ou subtrai, respectiva/.
- Quando **ALUOp** é **10**, o campo **Func** é examinado para determinar o valor de **ALUControl**.

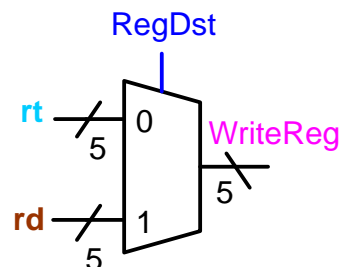
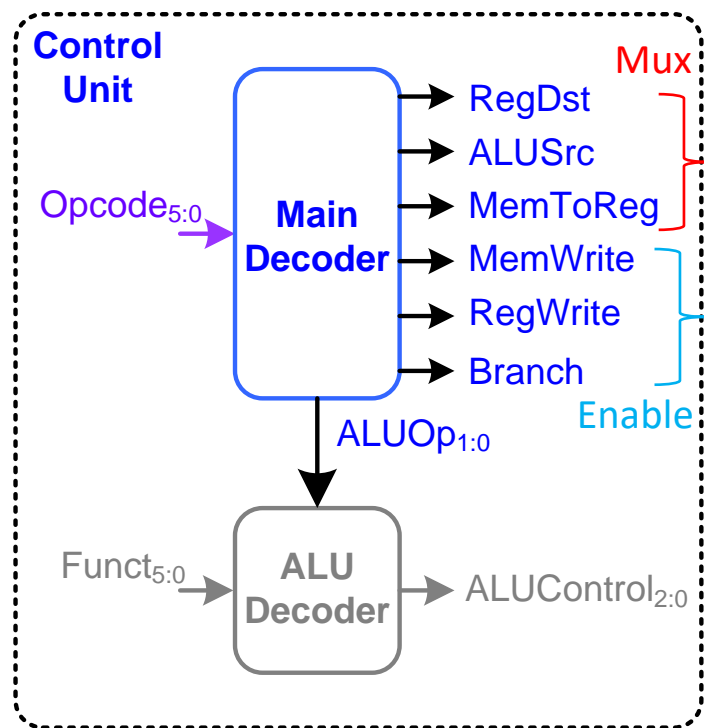


ALUOp_{1:0}	Func_{5:0}	ALUControl_{2:0}
00	X	010 (Add)
01	X	110 (Subtract)
10	100000 (add)	010 (Add)
10	100010 (sub)	110 (Subtract)
10	100100 (and)	000 (And)
10	100101 (or)	001 (Or)
10	101010 (slt)	111 (SLT)

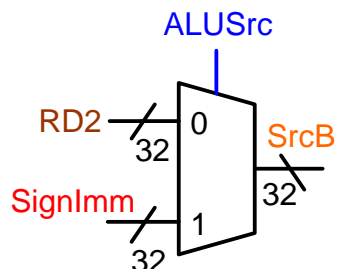
Para as instruções do tipo-R, os **dois primeiros bits** do campo **Func** são sempre **10**, podendo ser ignorados aquando da implementação.

UC (7) - Main Decoder (1) - Seleção de Multiplexers

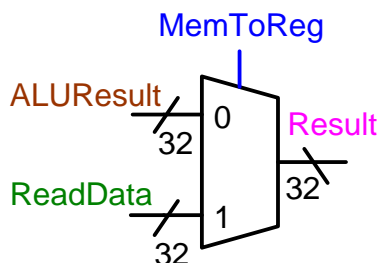
- O *Main Decoder* gera dois tipos de sinais: entradas de Seleção dos *Multiplexers* e de *Enable* da Memória, do Banco de Registos e de ainda outra lógica.



RegDst: Activo seleciona **rd** como **WriteReg** do RF.



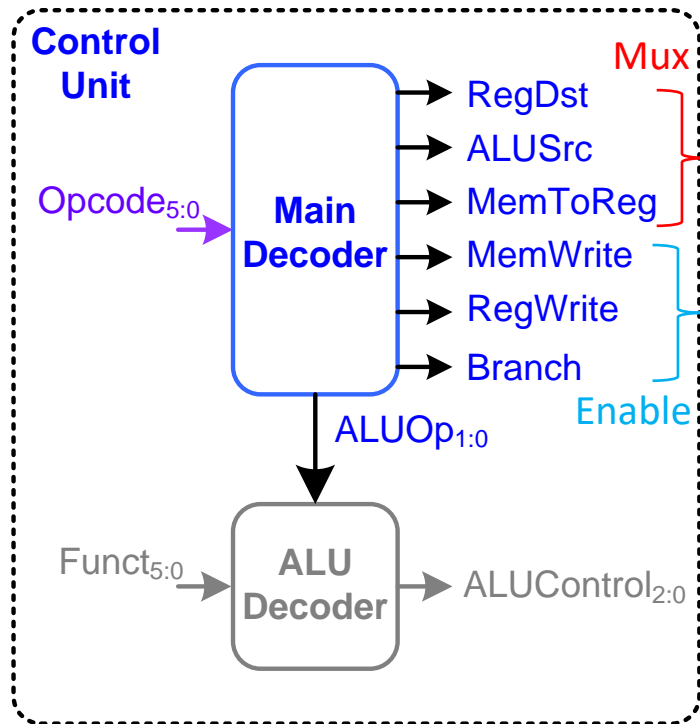
ALUSrc: Activo seleciona **SignImm** como **SrcB** da ALU.



MemToReg: Activo seleciona **ReadData** como **Result** a escrever no RF.

Os sinais de controlo necessários a cada instrução já foram vistos durante a construção do datapath.

UC (7) - Main Decoder (2) - Sinais de Enable

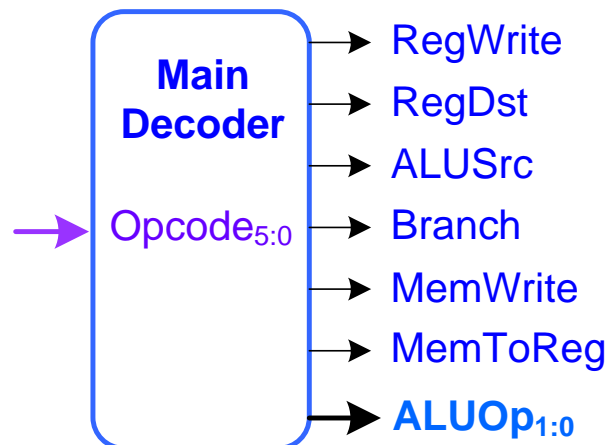


- **MemWrite** - *Enable* de escrita da Memória de Dados
- **RegWrite** - *Enable* de escrita do Banco de Registos
- **Branch** - Activo só para a instrução **beq** (ou **bne**)

UC (8) - Main Decoder (3) - Tabela de Verdade

Sinais de saída em função do *opcode* da instrução.

Instruction	Opcode _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemToReg	ALUOp _{1:0}
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01



1. tipo-R: Todas as instruções usam os mesmos valores dos sinais, só diferem no código (ALUControl) gerado pelo *ALU Decoder*.

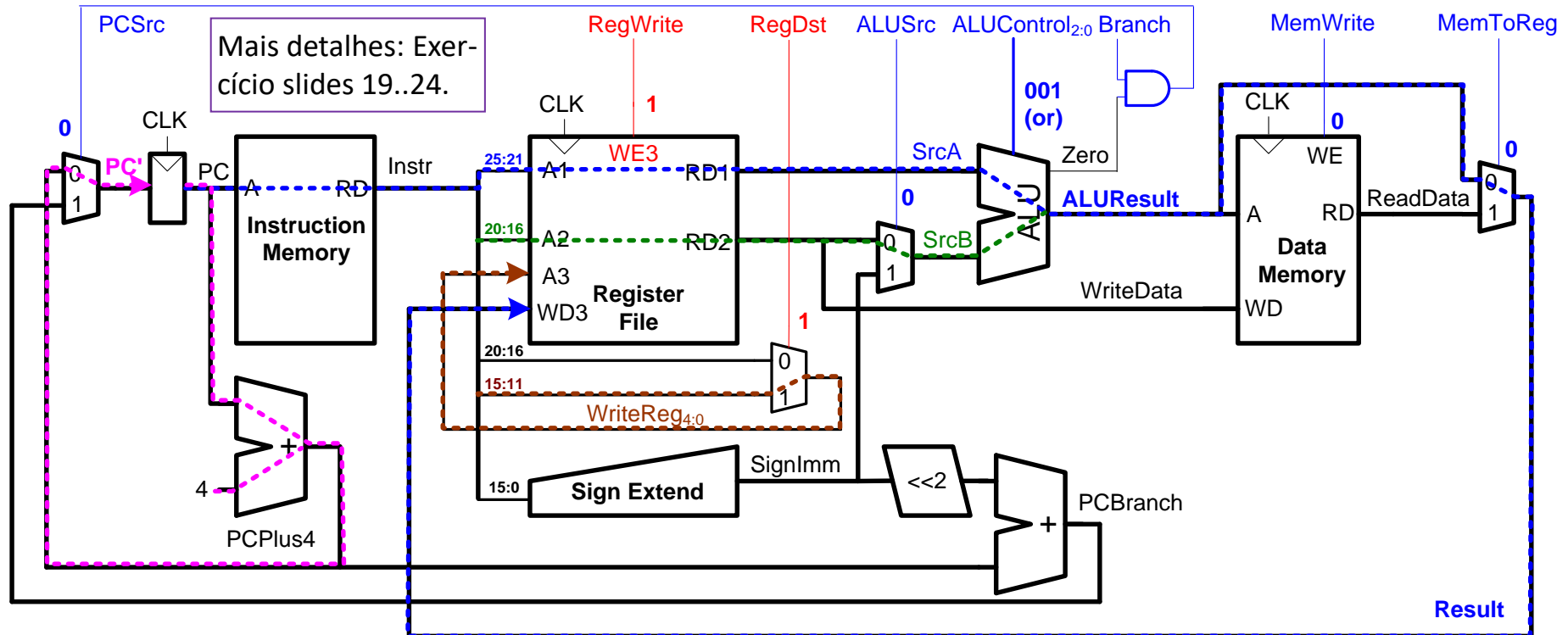
2. Se não escrevem no Banco de Registos: Os sinais **RegDst** e **MemToReg** são *don't cares* (X), dado que o sinal **RegWrite** é zero. (Exs: **sw** e **beq**).

Em seguida, analisamos em detalhe cada tipo de instrução...

UC (9) - Main Decoder (4) - *tR* - or

31:26	25:21	20:16	15:11	10:6	5:0
0	rs	rt	rd	0	37

Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemToReg	ALUOp _{1:0}
→ R-type	000000	1	1	0	0	0	0	10
lw	100011							
sw	101011							
beq	000100							

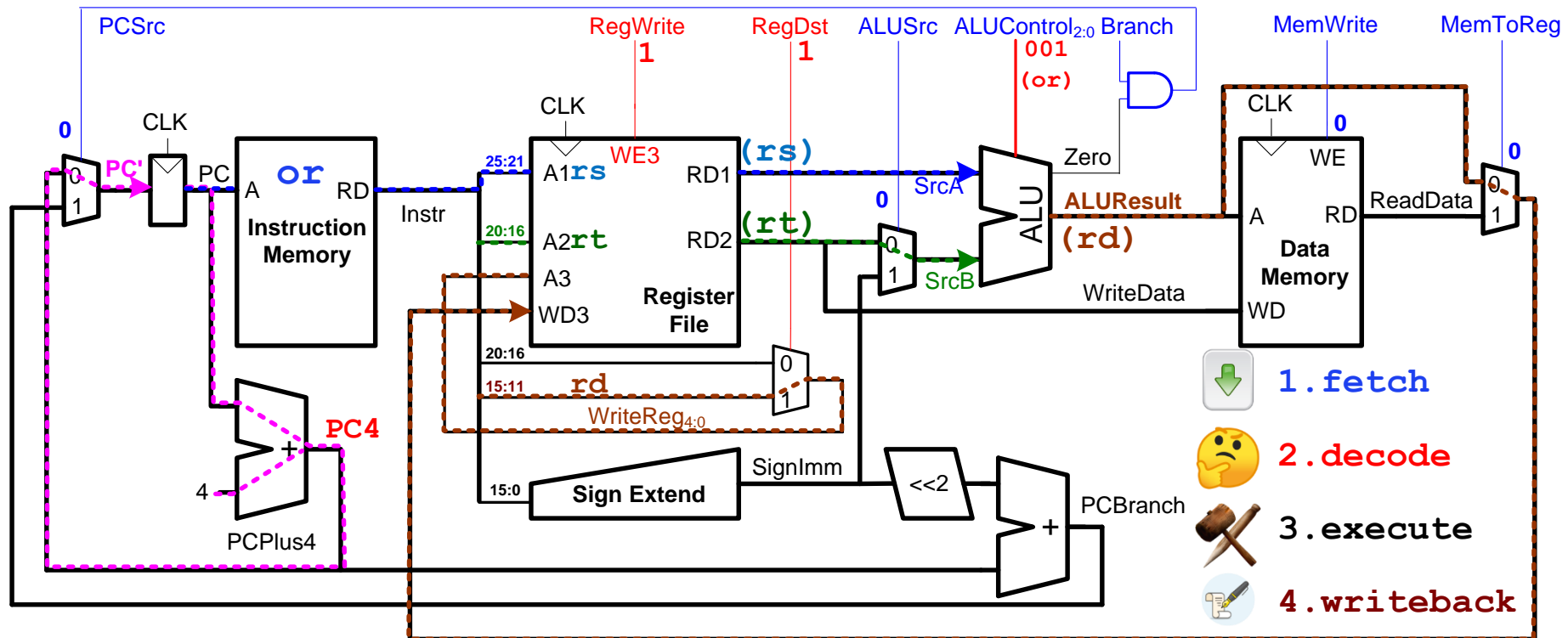


*Eliminando o símbolo da Unidade de Controlo para melhorar a visibilidade.

or rd, rs, rt

UC (9) - Main Decoder (4) - tipo-R - or - animado

Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemToReg	ALUOp _{1:0}
→ R-type	000000	1	1	0	0	0	0	10
lw	100011							
sw	101011							
beq	000100							

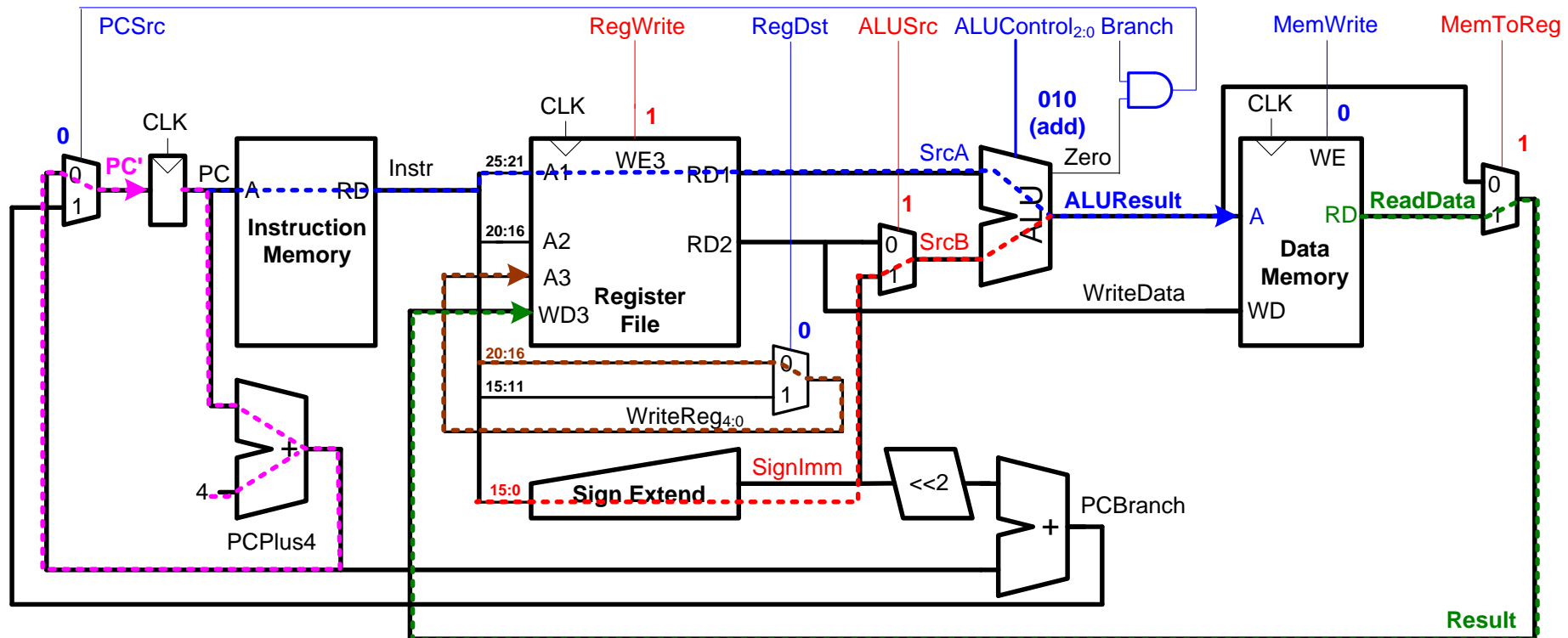


or rd, rs, rt

UC (10) - Main Decoder (5) - lw

31:26	25:21	20:16	15:0
35	rs	rt	imm16

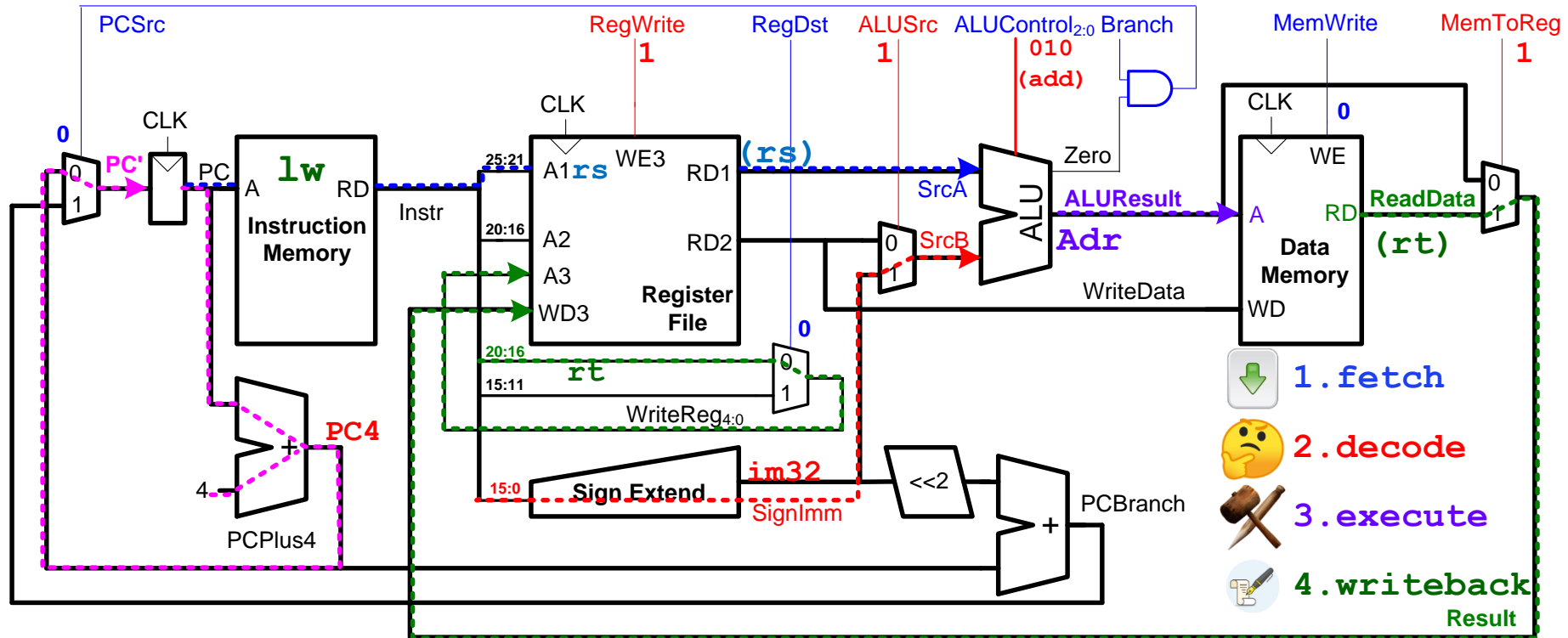
Instruction	Op _{5:0}	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemToReg	ALUOp _{1:0}
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011							
beq	000100							



lw rt, imm(rs)

UC (10) - Main Decoder (5) - *lw* - animado

Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemToReg	ALUOp _{1:0}
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011							
beq	000100							

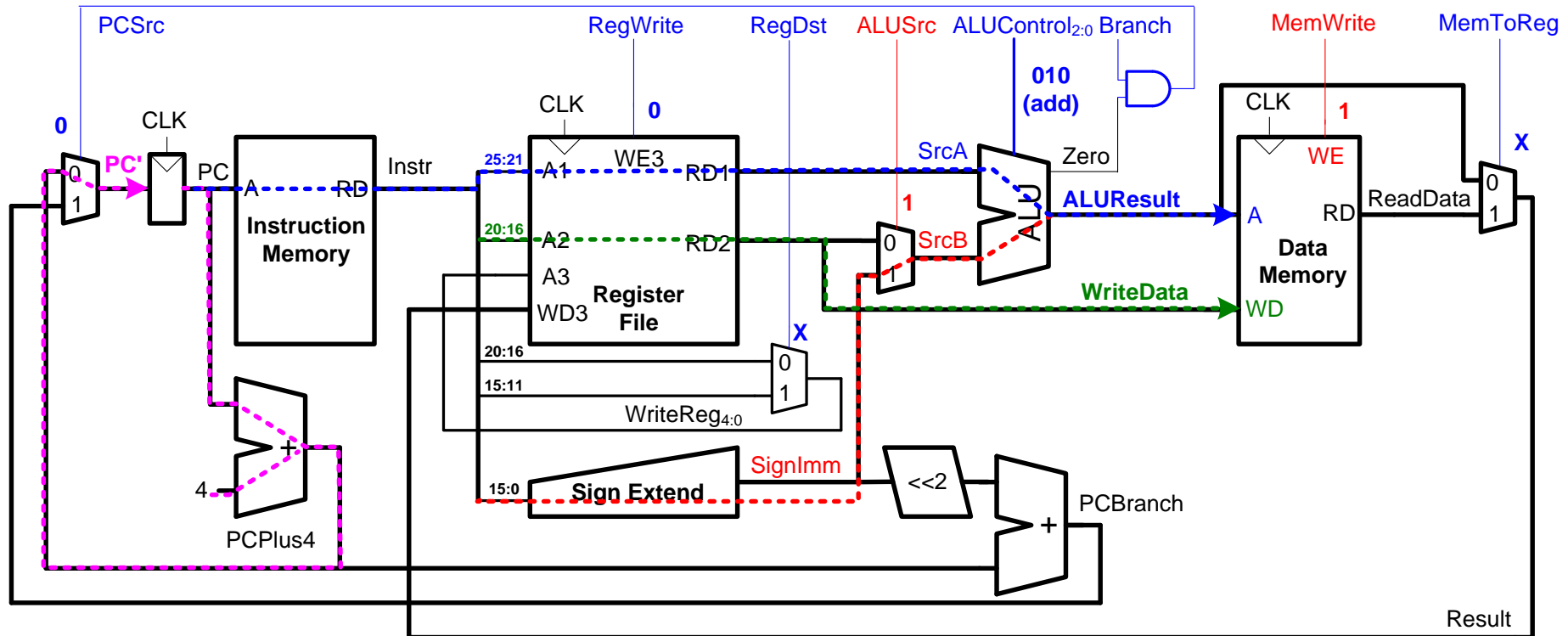


lw rt, imm(rs)

UC (11) - Main Decoder (6) - SW


31:26	25:21	20:16	15:0
43	rs	rt	imm16

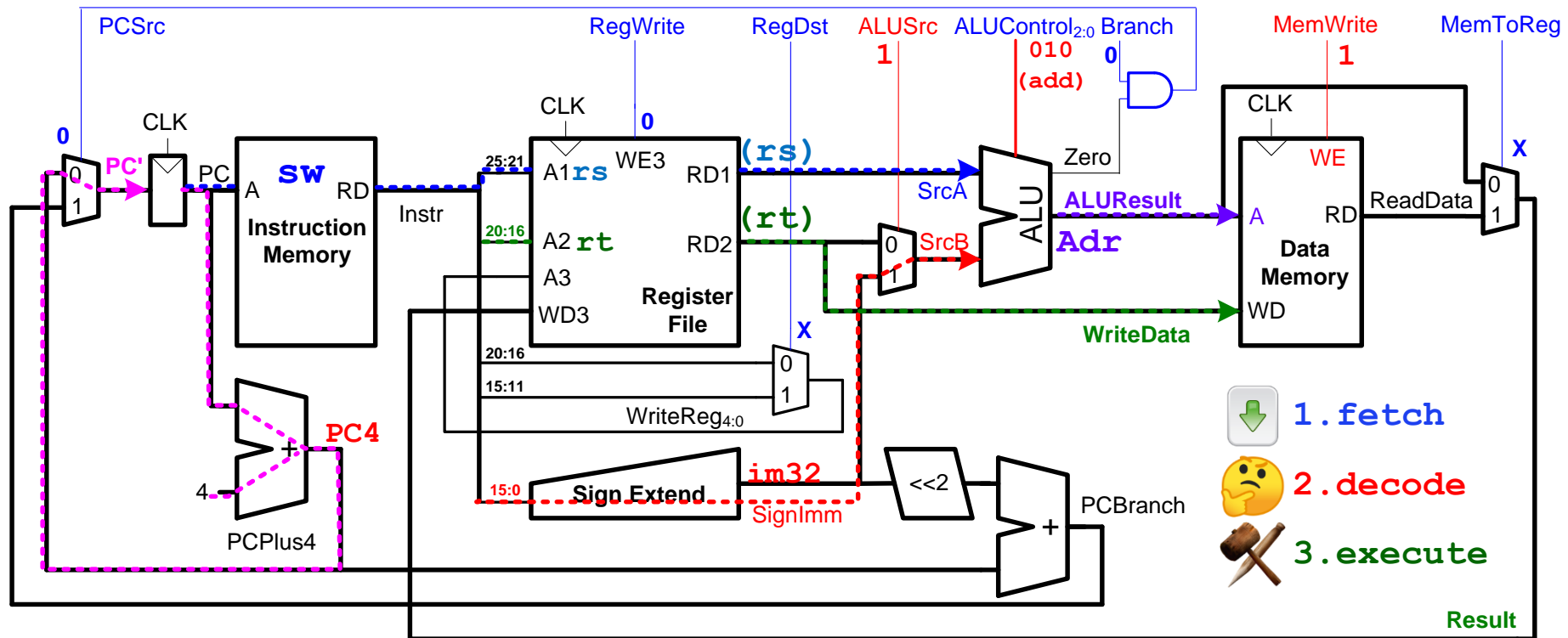
Instruction	Op _{5:0}	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemToReg	ALUOp _{1:0}
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
→ sw	101011	0	X	1	0	1	X	00
beq	000100							



sw rt, imm(rs)

UC (11) - Main Decoder (6) - *sw* - animado

Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemToReg	ALUOp _{1:0}
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
 sw	101011	0	X	1	0	1	X	00
beq	000100							

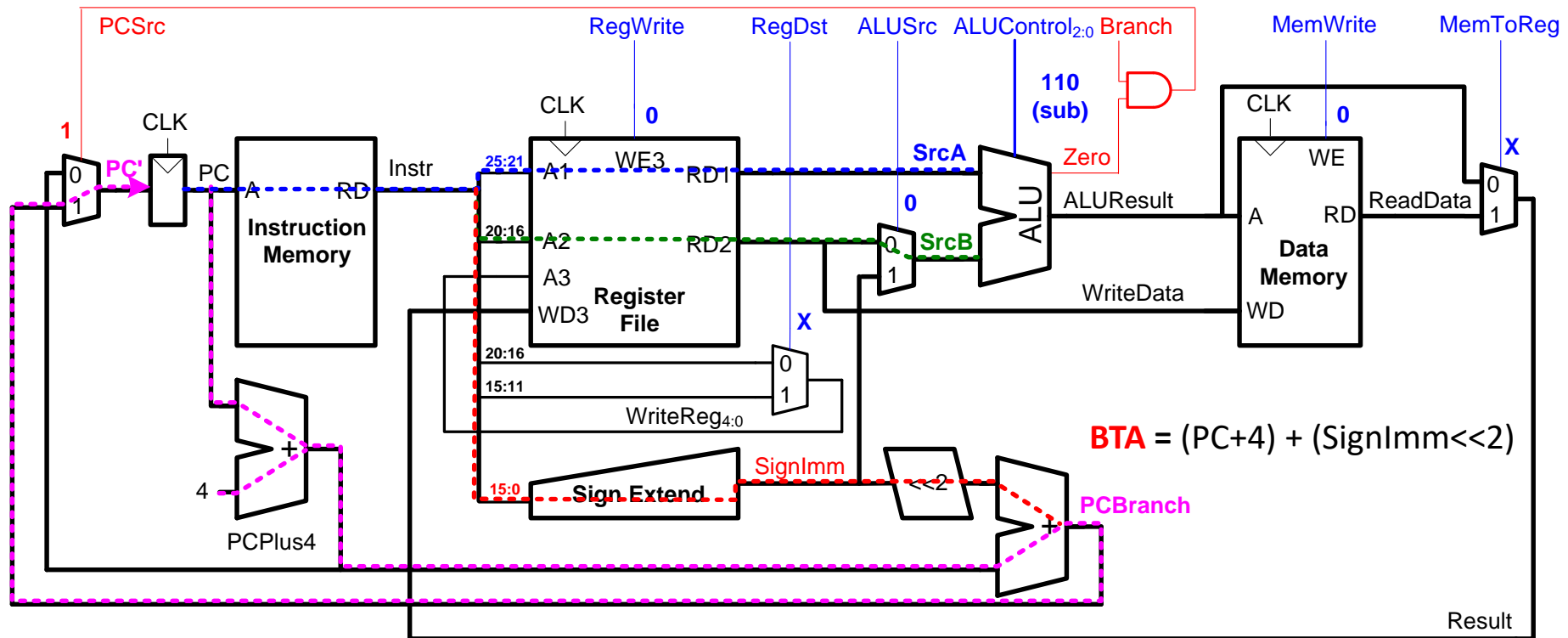


sw *rt*, *imm*(*rs*)

UC (12) - Main Decoder (7) - beq

31:26	25:21	20:16	15:0
8	rs	rt	imm16

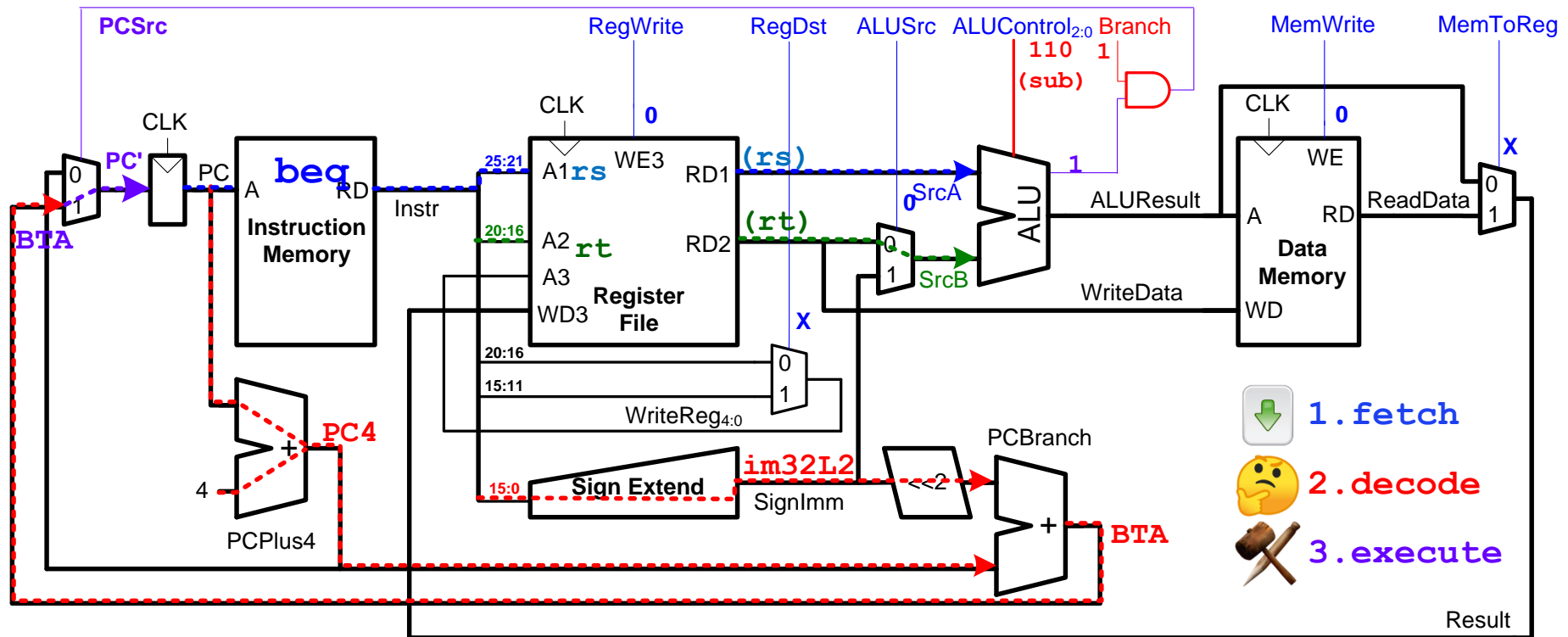
Instruction	Op _{5:0}	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemToReg	ALUOp _{1:0}
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
➔ beq	000100	0	X	0	1	0	X	01



beq rs, rt, imm

UC (12) - Main Decoder (7) - beq - animado

Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemToReg	ALUOp _{1:0}
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
➔ beq	000100	0	X	0	1	0	X	01



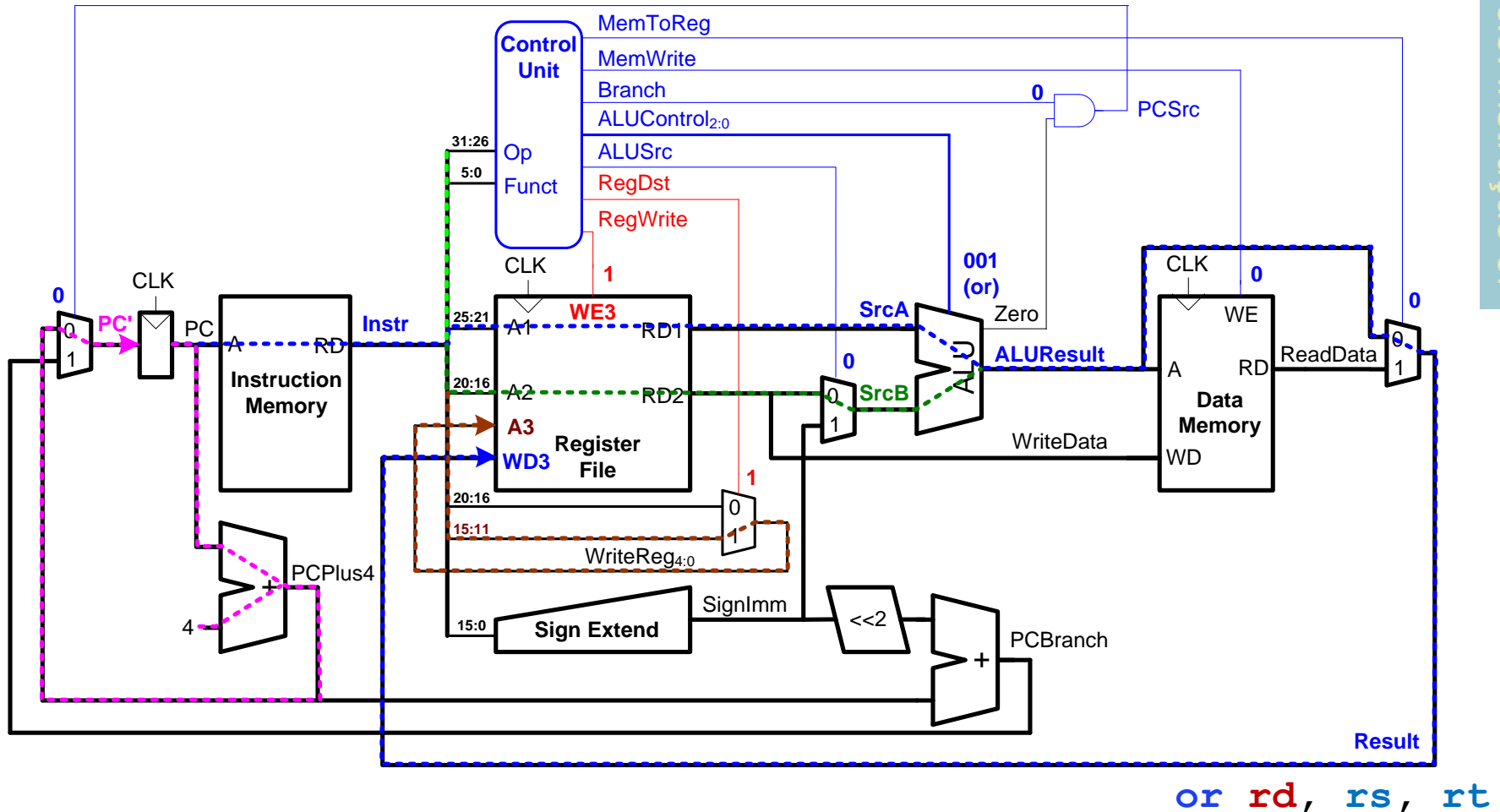
$$BTA = (PC+4) + (SignImm \ll 2)$$

beq rs, rt, imm

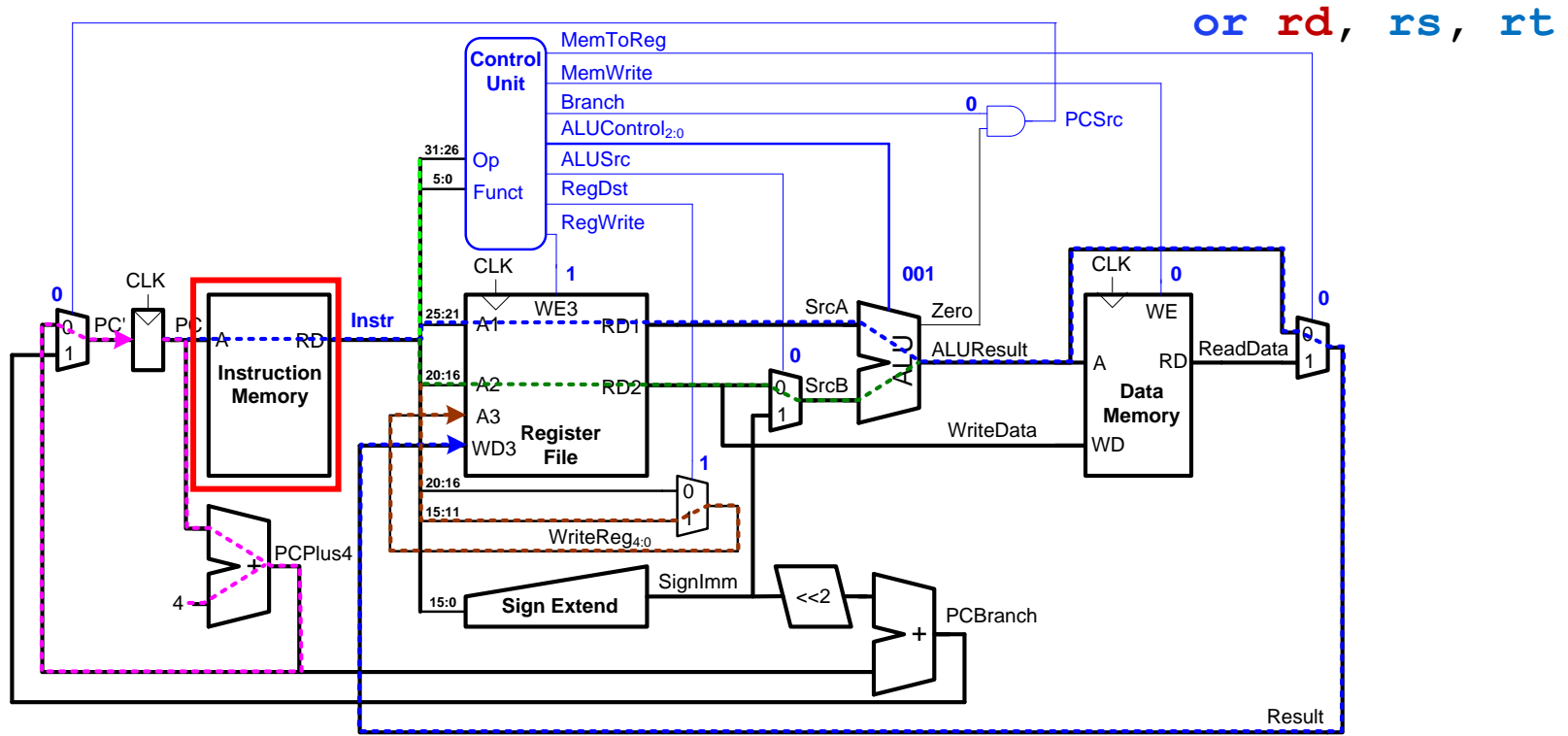
Exercício - OR (1) - tipo-R

Determinar os valores dos sinais de controlo e as zonas do *datapath* durante a execução da instrução.

5. Exercício: instrução or



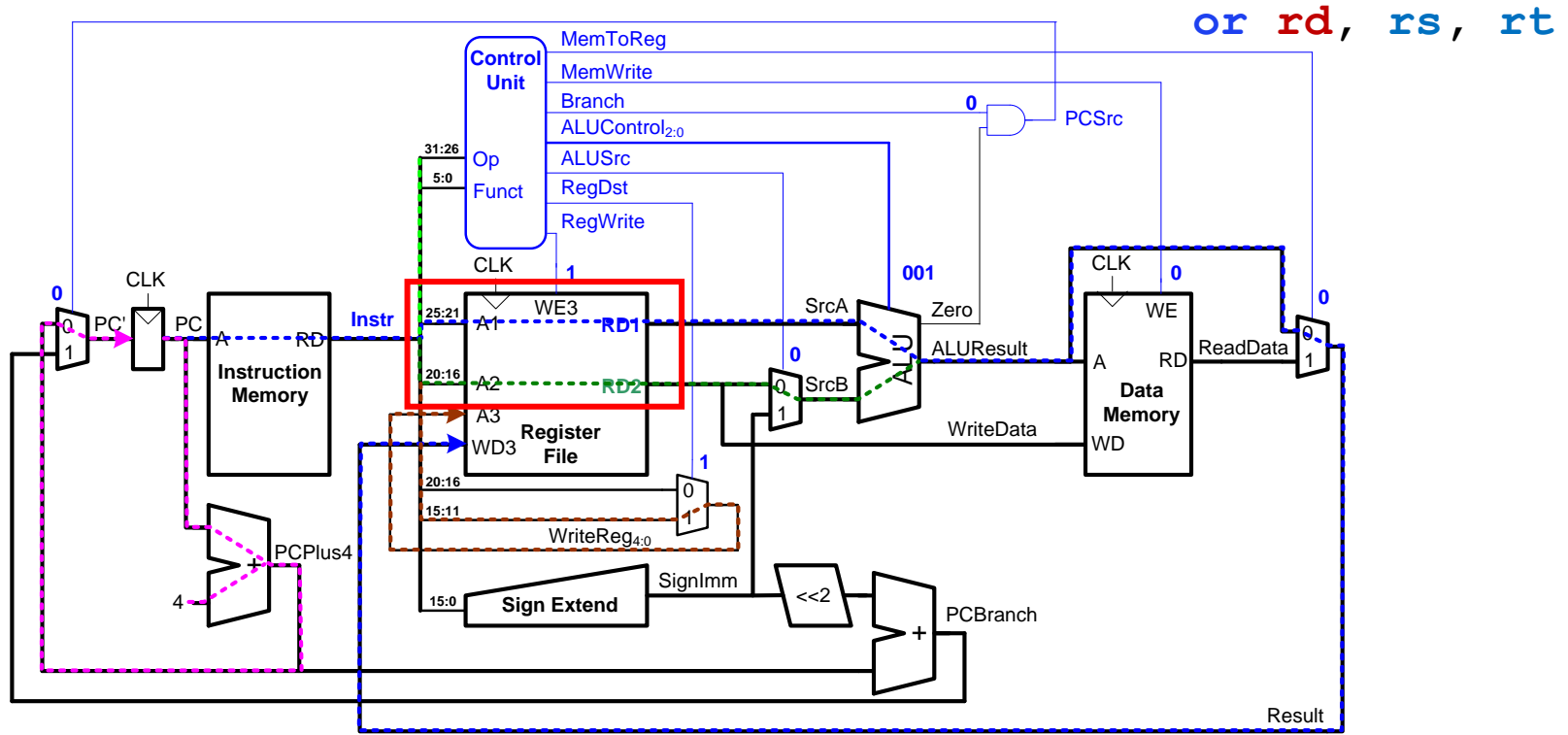
Exercício - OR (2) - Fetch da Instrução



1. O PC aponta para a posição de memória que contém a instrução; esta fica disponível na saída RD (*Instr*).

Exercício - OR (3) - Ler Operandos

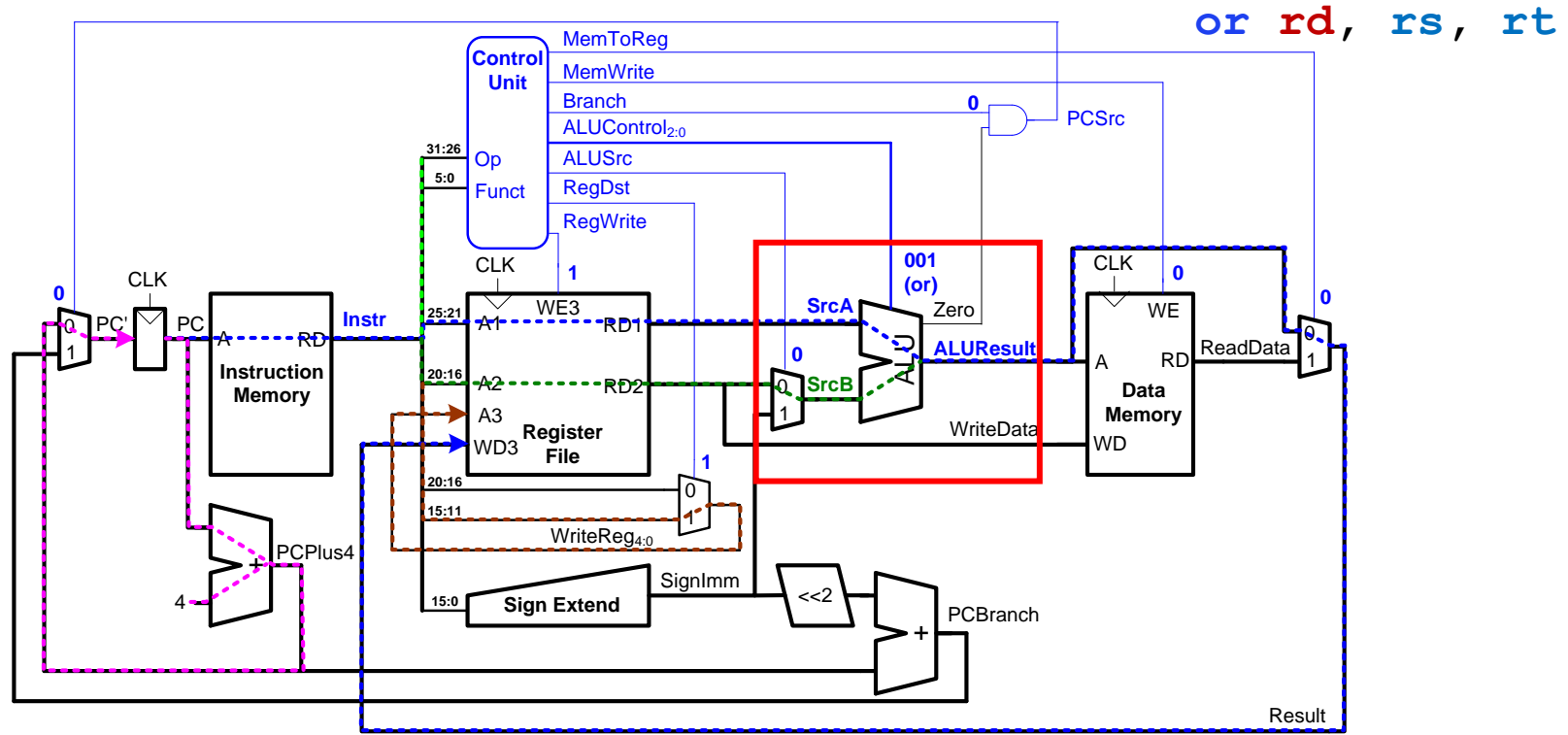
31:26	25:21	20:16	15:11	10:6	5:0
0	rs	rt	rd	0	37



2. Os campos da instrução $Instr_{25:21}$ e $Instr_{20:16}$ são os endereços dos registos *rs* e *rt* dentro do RF, respectiva/. O conteúdo destes registos aparecem nas saídas RD1 e RD2.

Exercício - OR (4) - Exec. na ALU

31:26	25:21	20:16	15:11	10:6	5:0
0	rs	rt	rd	0	37

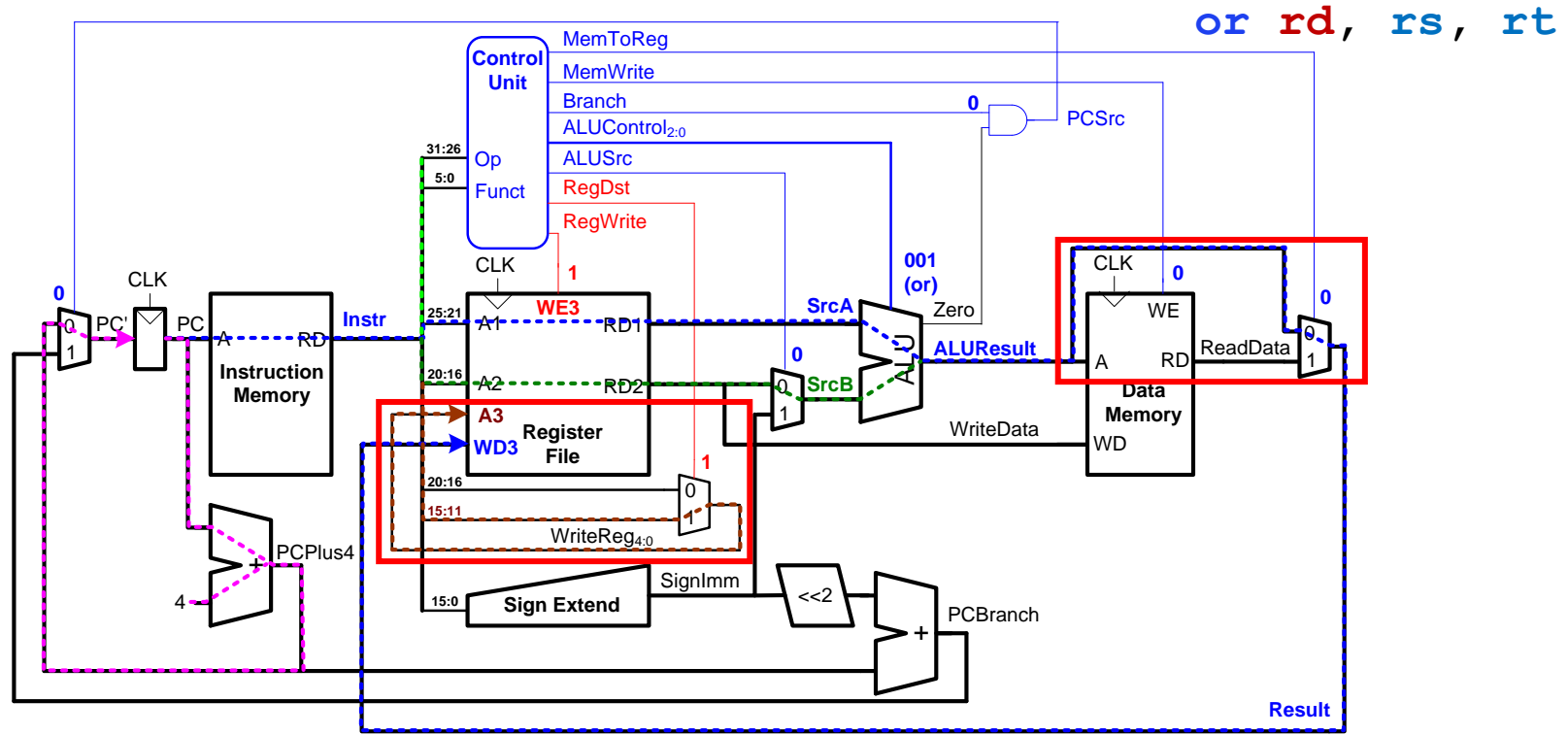


- 3.1 SrcA = RD1 e SrcB = RD2 (ALUSrc = 0);
- 3.2 **OR** é do tipo-R logo **ALUOp=10**; o valor de **ALUControl** depende de **Funct**, sendo neste caso igual a **001** (ver **ALU Decoder**).

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	101010 (slt)	111 (SLT)

Exercício - OR (5) - Escrita no RF

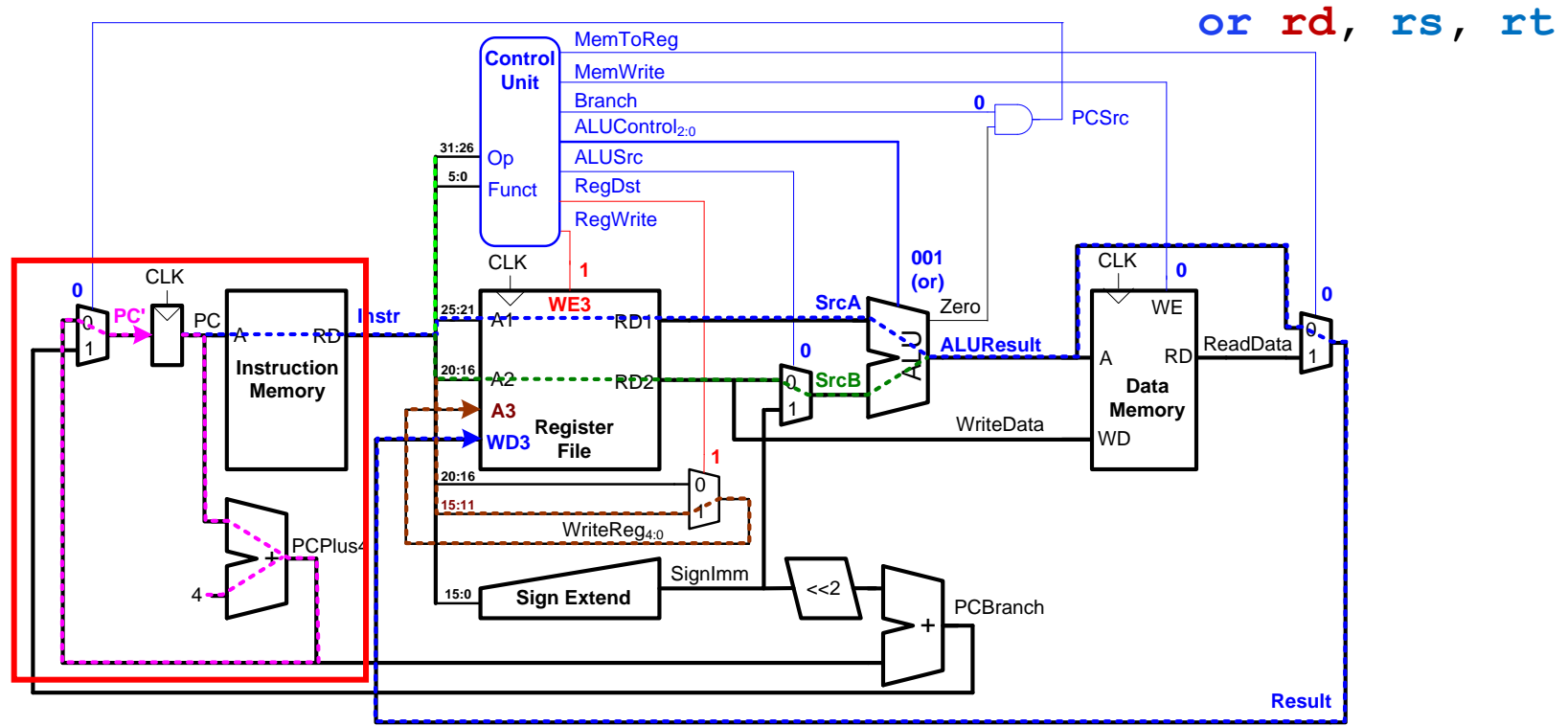
31:26	25:21	20:16	15:11	10:6	5:0
0	rs	rt	rd	0	37



4.1 A instrução não escreve na Memória de Dados, **MemWrite=0**. O resultado da operação vem direta/ da ALU, **MemToReg=0**.

4.2 O **Result** é escrito (**RegWrite=1**) no Banco de Registos. O registo destino **rd**, o campo **Instr_{15:11}**, é selecionado fazendo **RegDst=1**.

Exercício - OR (6) - Atualização do PC



5. A atualização do valor de PC, i.e., $PC' = PC + 4$, está indicada com a linha rosa tracejada

Finalmente, é de referir que também há fluxo de dados nas zonas não tracejadas, todavia os sinais de controlo **impedem** que esses dados tenham qualquer influência no resultado.

Mais Instruções (1) - *addi* e *j*

Até aqui considerámos um subconjunto limitado de instruções do MIPS.

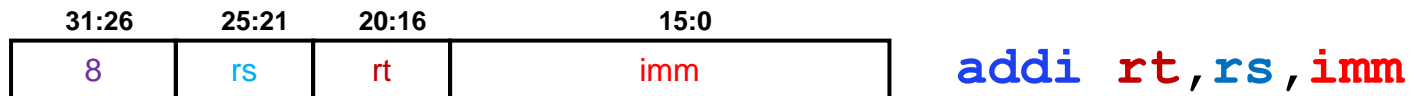
Como adicionar novas instruções ao CPU (ou ao ISA)?

- Para ilustrar, vamos acrescentar suporte para duas instruções *addi* e *j*.
- Veremos que para um tipo de instruções (*addi*) basta adicionar algo à Tabela de Verdade do *Main Decoder*, ao passo que para outras (*j*) o *datapath* também precisa de ser alterado.

Mais Instruções (2) - *addi* (1)

addi, 'add immediate':

- Adiciona o valor **imediato** ao valor do registo **rs** e escreve o resultado no registo **rt**.

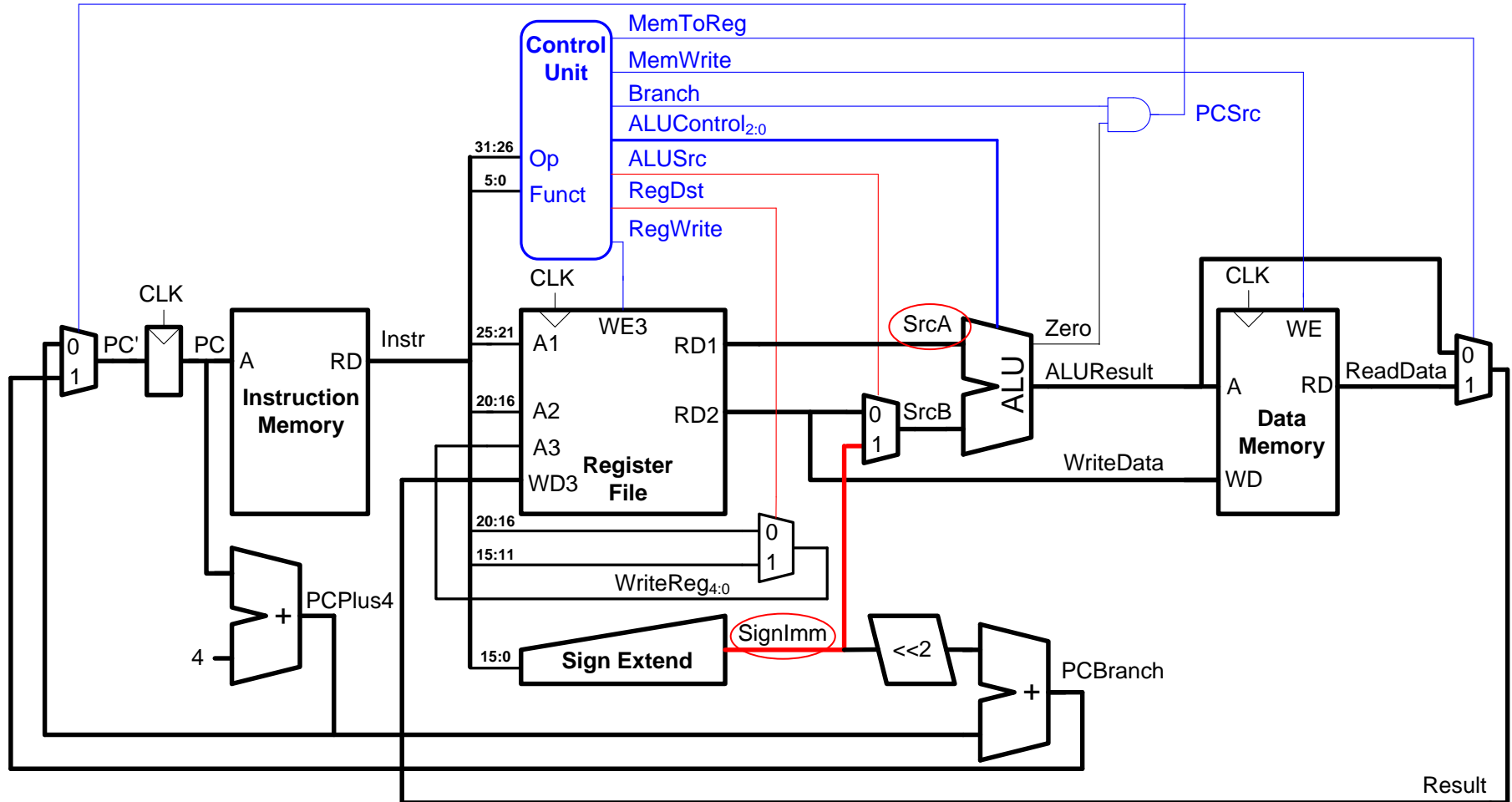


O *datapath* atual **já é capaz** de executar este tipo de tarefas (i.e., somar e escrever o resultado no RF).

P: Quais as alterações necessárias a introduzir na Unidade de Controlo para suportar **addi**?

R: Precisamos de acrescentar somente mais uma linha à Tabela de Verdade do *Main Decoder*, para gerar os sinais de controlo necessários à execução da instrução **addi**.

Mais Instruções (3) - *addi* (2) - Datapath



Não há necessidade de modificar o *datapath*!

Mais Instr. (4) - *addi* (3) - Unidade de Controlo

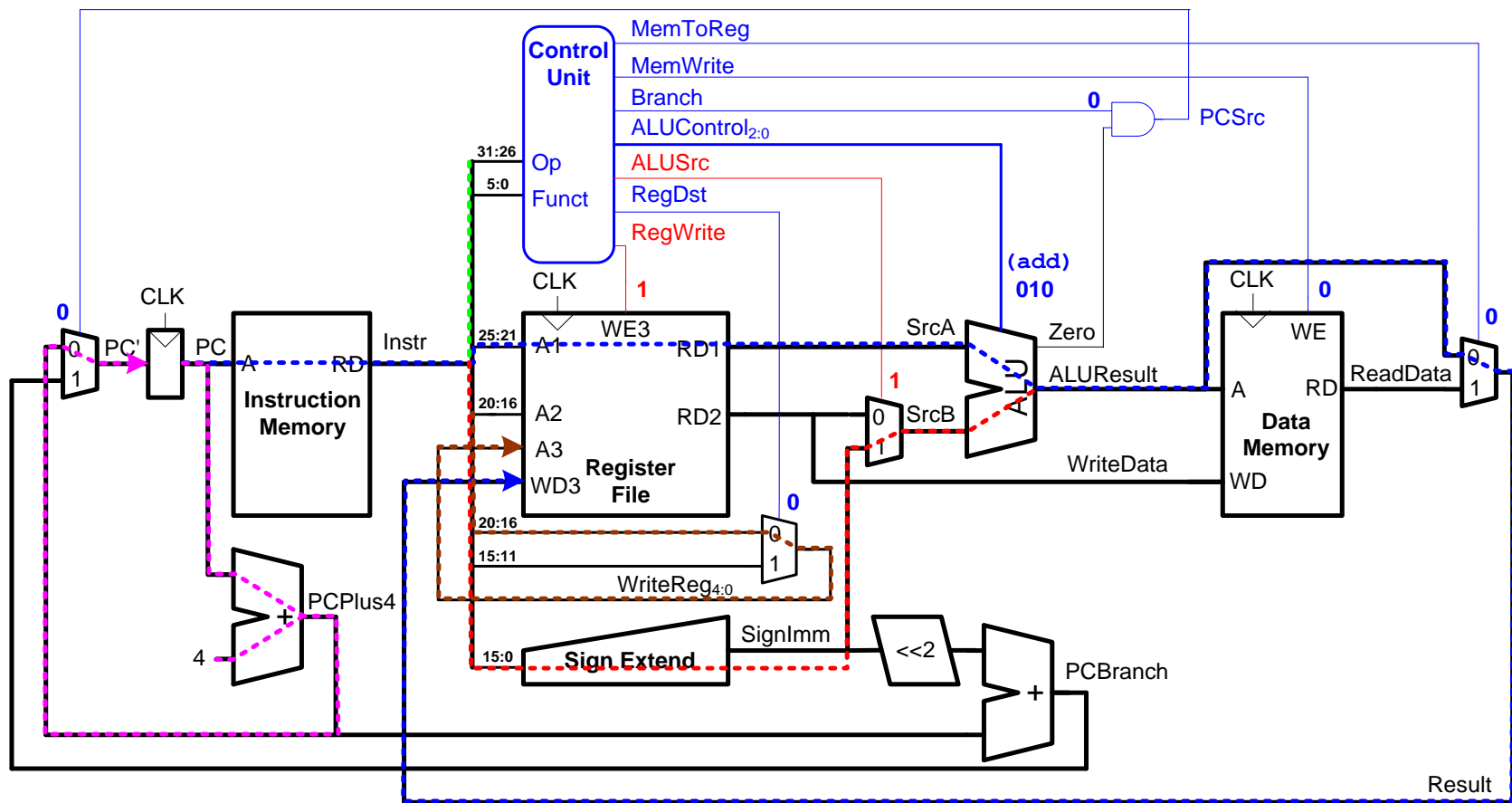
addi **rt**, **rs**, **imm**

Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemToReg	ALUOp _{1:0}
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
→ <i>addi</i>	001000	1	0	1	0	0	0	00

1. O resultado tem de ser escrito no RF, **RegWrite = 1**.
2. O registo destino é especificado no campo **rt**, **RegDst = 0**.
3. O **SrcB** da ALU deriva do *immediate*, **ALUSrc = 1**.
4. A ALU deve somar, **ALUOp = 00**, e o valor de **ALUControl = 010** (ver *ALU Decoder*)
5. A instrução não é um *branch*, nem escreve na memória, **Branch = MemWrite = 0**.
6. O resultado vem da ALU, não da memória, **MemToReg = 0**.

Mais Instr. (5) - *addi* (4) - Datapath + Control

addi **rt**, **rs**, **imm**

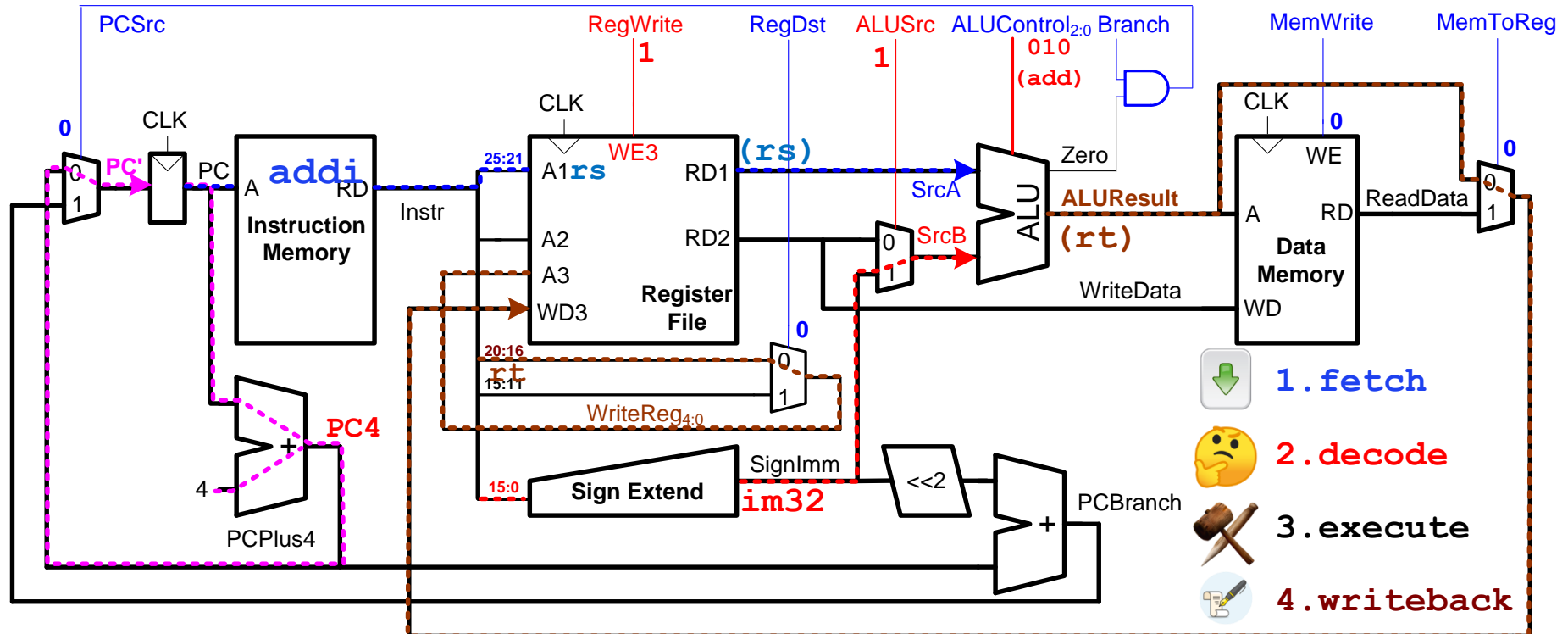


RegWrite=1, RegDst=0, ALUSrc=1 e MemToReg=0

Mais Instr. (5) - addi (5) - animado

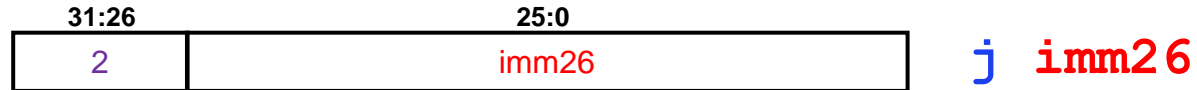
addi **rt**, **rs**, **imm**

Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemToReg	ALUOp _{1:0}
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
➔ addi	001000	1	0	1	0	0	0	00



Mais Instruções (6) - j (1)

A instrução *jump* escreve um novo valor (**PC'**) no PC .



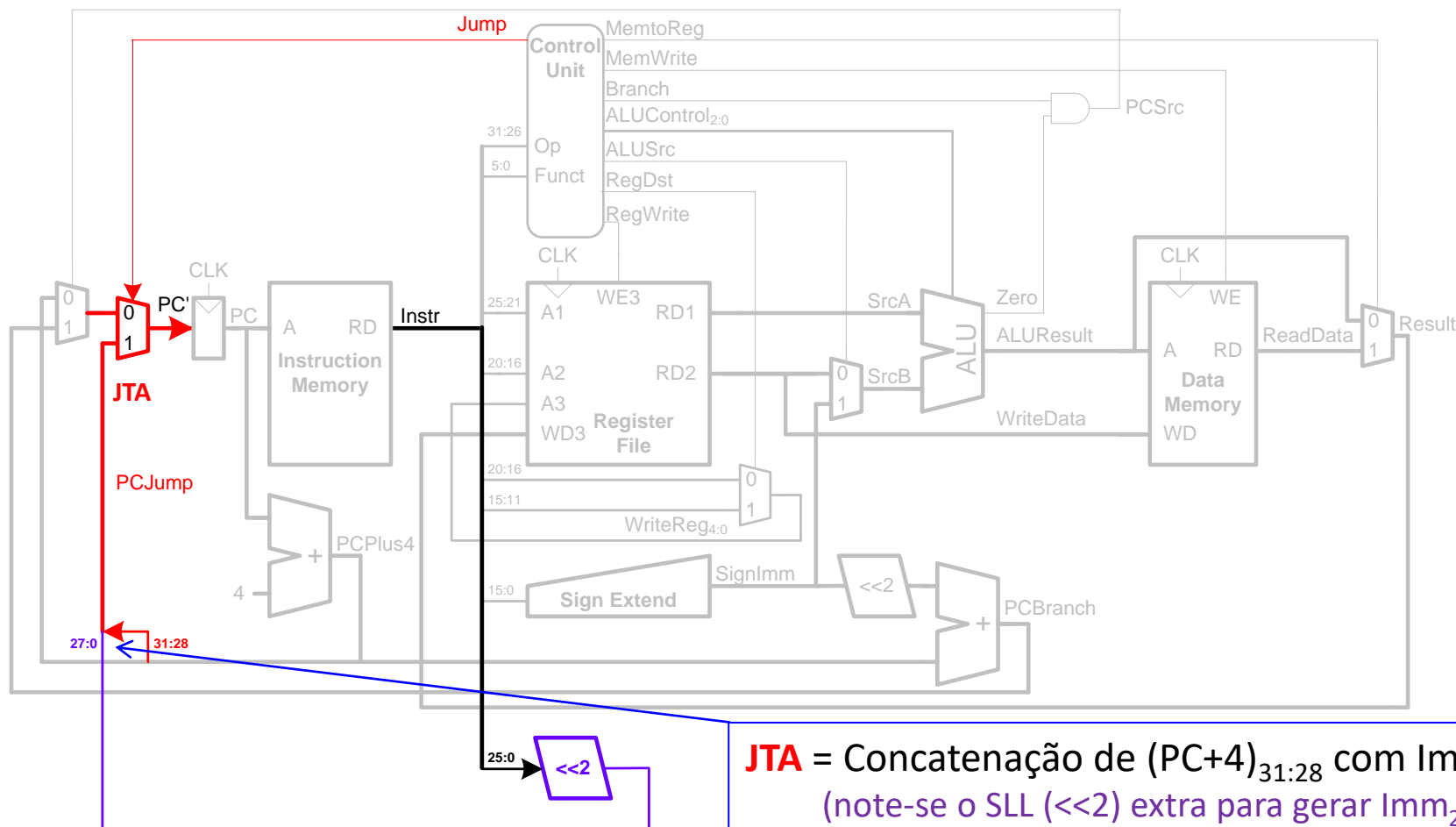
$$\text{PC}' = \text{JTA} = (\text{PC} + 4)_{31..28} : (\text{Imm26} \ll 2)$$

- Os dois bits menos significativos são sempre 0, porque o valor do PC é sempre *word-aligned*, (i.e., é múltiplo de 4 bytes);
 - Os 26 bits seguintes são retirados do valor imediato da instrução **Instr25:0**;
 - Os 4 bits mais significativos são iguais aos 4 bits mais significativos do (PC + 4).
- O *datapath* existente não é capaz de calcular este **PC'**.

P: Quais são as modificações a fazer, tanto ao *datapath* como ao *Main Decoder*, para suportar a instrução **j**?

Mais Instruções (7) - j (2) - Datapath

R-1: Alteração ao Datapath: adicionamos *hardware* para calcular o valor do **PC'**. Isto pode ser conseguido através de mais um *multiplexer*, controlado por um novo sinal **Jump** (proveniente do *Main Decoder*), a cuja entrada_1 ligamos PCJump (**JTA**).



Mais Instruções (8) - j (3) - Main Decoder

R-2: Alteração à Tabela de Verdade

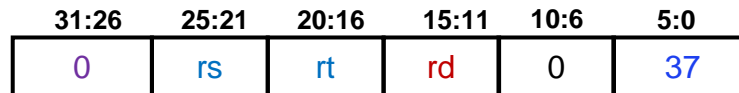
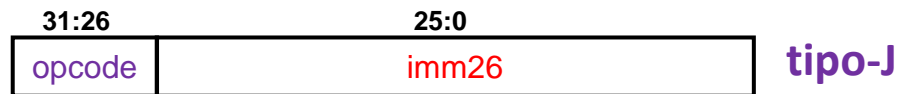
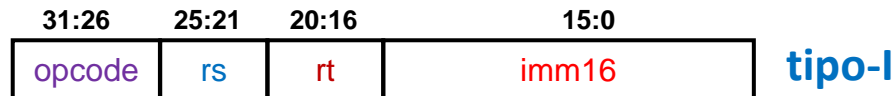
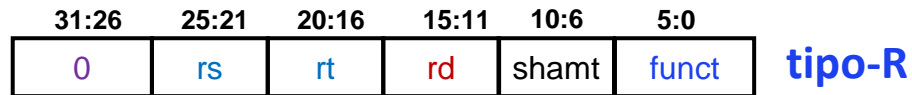
Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemToReg	ALUOp _{1:0}	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
addi	001000	1	0	1	0	0	0	00	0
→ j	000010	0	X	X	X	0	X	XX	1

Acrescentamos **mais uma linha** à Tabela de Verdade do *Main Decoder*, com os sinais de controlo para a instrução **j**, e uma nova coluna para o sinal **Jump**:

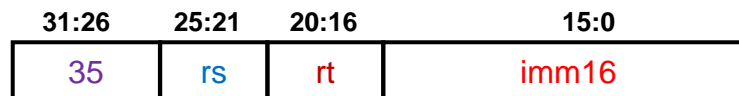
- O sinal **Jump** é '1' para a instrução **j** e '0' para as demais.
- A instrução **j** não escreve no Banco de Registos nem na Memória, logo **RegWrite = MemWrite = 0**.
- De fato, podemos ignorar o cálculo feito no *datapath*, donde **RegDst = ALUSrc = Branch = MemToReg = ALUOp = X**.

A seguir: Resolução de (mais) Exercícios SC

Formato das Instruções- Resumo



or rd, rs, rt



lw rt, imm(rs)



sw rt, imm(rs)



beq rs, rt, imm

Datapath do μ P MIPS: Single-cycle

Aula	Data	Descrição
22	13/Mai (4ª)	VI - Organização interna do processador: Unidades operativas e de controlo. <i>Datapath</i> : Pressupostos para a construção de um <i>datapath</i> genérico para uma arquitectura tipo MIPS. Análise dos blocos constituintes necessários à execução de cada tipo de instruções básicas: tipo R; <i>load</i> e <i>store</i> ; salto condicional.
23	15/Mai	Unidade de Controlo Descodificador da ALU; Exemplo de ALU Principal Descodificador Principal Exercício: Execução da instrução <i>or</i> Instruções adicionais: <i>addi</i> e <i>j(ump)</i>
24	20/Mai (4ª)	Resolução de problemas sobre uArquiteturas Single-cycle.

Datapath do μ P MIPS: *Multicycle*

Aula	Data	Descrição
25	22/Mai	<i>Multicycle</i> : Limitações das arquiteturas <i>single-cycle</i> ; Versão de referência duma arquitetura <i>multicycle</i> ; Exemplos do processamento das instruções numa arquitetura <i>multicycle</i> .
26	27/Mai (4ª)	<i>Unidade de Controlo para datapath multicycle</i> : diagrama de estados. Sinais de controlo e valores do <i>datapath multicycle</i> . Exemplos da execução sequencial de algumas instruções <i>no datapath multicycle</i> .
27	29/Mai	Resolução de <i>problemas</i> sobre uArquiteturas Multicycle.

Comunicação do μ P com o exterior (I/O)

Aula	Data	Descrição
28	3/Jun	VII - Comunicação com o exterior: Entrada e saída de dados (I/O) Acesso a dispositivos de I/O: I/O Memory-Mapped Hardware e Software (Asm). Dispositivos de I/O Embedded uControlador PIC32 (MIPS): I/O Digital: Switches e LEDs I/O Série : SPI e UART.
29	5/Jun	I/O sob interrupção: Timers e Interrupções I/O analógico: ADC e DAC; Outros periféricos: LCD e Bluetooth.