Introdução à Arquitetura de Computadores

Aula23

µArquitetura MIPS Single-cycle: II

Unidade de Controlo

Descodificador da ALU

Exemplo de ALU

Descodificador Principal

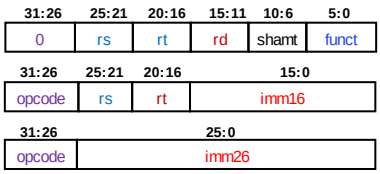
Exercício

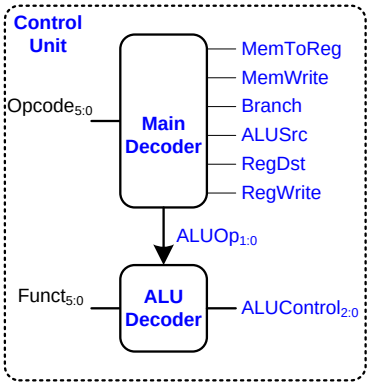
Execução da Instrução or

Suporte para mais Instruções

Instruções adicionais: addi e jump

Unidade de Controlo (1) - Entradas/Saídas





A Unidade de Controlo (UC) gera os sinais de controlo do *datapath* usando como entradas os bits do opcode e funct da instrução, i.e., $Instr_{31:26}$ e $Instr_{5:0}$, respectiva/.

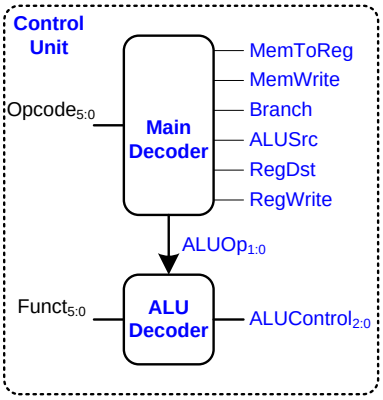
A maior parte da informação de controlo (saídas) é derivada do opcode, embora as instruções de tipo-R precisem de usar o campo funct para determinar a operação da ALU (opcode do tipo-R=0).

*Retomamos a matéria dada na 1ª aula sobre Assembly.

4. Unidade de Controlo

UC (2) - Decoders: Main + ALU

A Unidade de Controlo (UC) é dividida em dois blocos de lógica combinatória*:



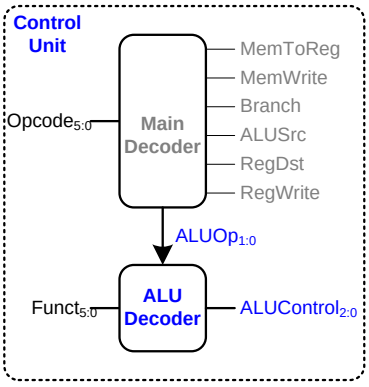
O *Main Decoder* usa o $Opcode_{5:0}$ da instrução, para gerar todos os sinais de saída e ainda o código de operação da ALU ($ALUOp_{1:0}$).

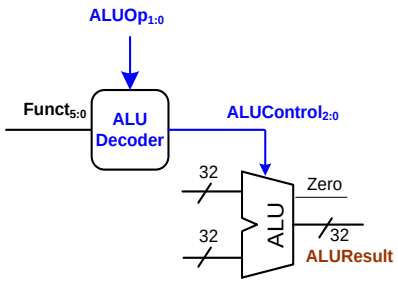
O *ALU Decoder* usa o sinal $ALUOp_{1:0}$ juntamente com o campo $Funct_{5:0}$ para gerar o sinal $ALUControl_{2:0}$, o qual controla a operação da ALU.

*Para simplificar o projeto.

UC (3) - ALU Decoder (1)

É a parte da UC responsável por descodificar o campo da instrução $Funct_{5:0}$. Para esse efeito conta com o sinal $ALUOp_{1:0}$ gerado pelo *Main Decoder*.

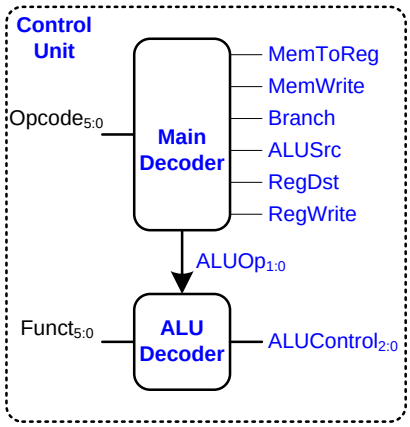




O *ALU Decoder* gera *o* sinal $ALUControl_{2:0}$ que controla a operação da ALU.

UC (4) - ALU Decoder (1) - Entrada ALUOp

O sinal $ALUOp_{1:0}$, gerado pelo *Main Decoder*, e usado pelo *ALU Decoder*, tem o significado apresentado na tabela.

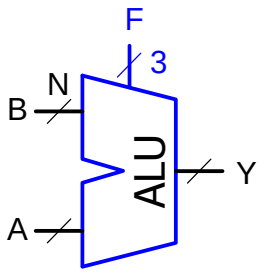


$ALUOp_{1:0}$	Descrição
00	Add
01	Subtract
10	Look at Funct
11	Not Used (yet)

Tipo-R

UC (5) - ALU Decoder (2) - Exemplo de ALU

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



$F_{2:0}$	Função
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Hoje em dia podem ser facilmente implementadas em linguagens de descrição de hardware (HDL) do tipo Verilog ou VHDL.

UC (6) - ALU Decoder (3) - Tabela Verdade

Quando ALUOp é 00 ou 01, a ALU deve somar ou subtrair, respectiva/.

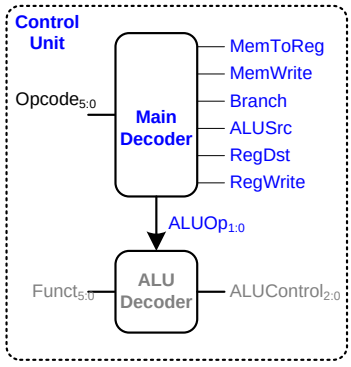
Quando ALUOp é 10, o *decoder* examina o campo funct para determinar o valor de ALUControl.

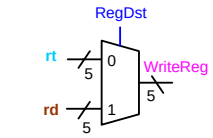
$ALUOp_{1:0}$	$Funct_{5:0}$	$ALUControl_{2:0}$
00	X	010 (Add)
01	X	110 (Subtract)
10	100000 (add)	010 (Add)
10	100010 (sub)	110 (Subtract)
10	100100 (and)	000 (And)
10	100101 (or)	001 (Or)
10	101010 (slt)	111 (SLT)

Para as instruções do tipo-R, os dois primeiros bits do campo funct são sempre 10, ou seja, podem ser ignorados aquando da implementação.

UC (7) - Main Decoder (1) - Sinais de Saída

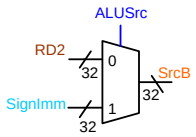
Os sinais de controlo necessários a cada instrução já foram vistos durante a construção do datapath.



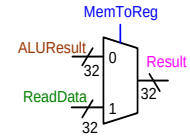


Seleção dos Multiplexers

RegDst: Activo seleciona **rd** (não **rt**) como $WriteReg_{4:0}$ (de Result).



ALUSrc: Activo seleciona **SignImm** (não **RD2**) como SrcB da ALU.



MemToReg: Activo seleciona ReadData (não ALUResult) como Result a escrever no RF.

MemWrite - *Enable* de Escrita da Memória de Dados

RegWrite - *Enable* de Escrita do Banco de Registos (RF)

Branch - Activo para a instrução Beq (ou Bne)

UC (8) - Main Decoder (2) - Tabela de Verdade

Sinais de controlo 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

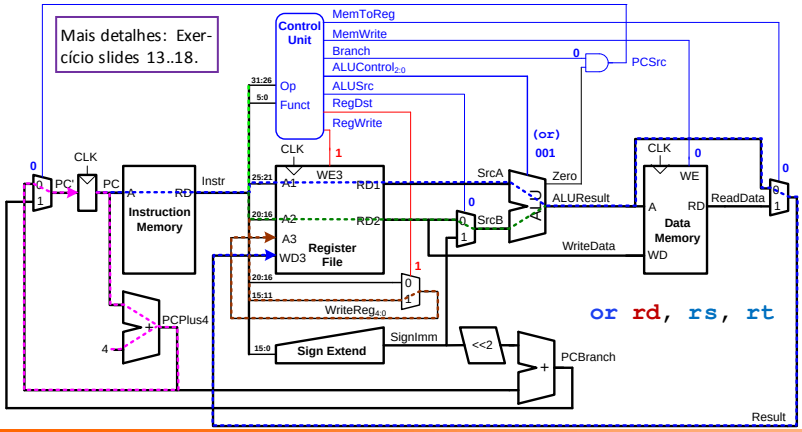
1. Todas as instruções do tipo-R usam os mesmos valores dos sinais, só diferem no código (ALUControl) gerado pelo *ALU Decoder*.

2. Para as instruções que não escrevem no Banco de Registos (eg, sw and beq), os sinais RegDst e MemToReg são *don't cares*.
Os dados (Result) e o endereço do registo (WriteReg) são ignorados porque o sinal RegWrite é zero.

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

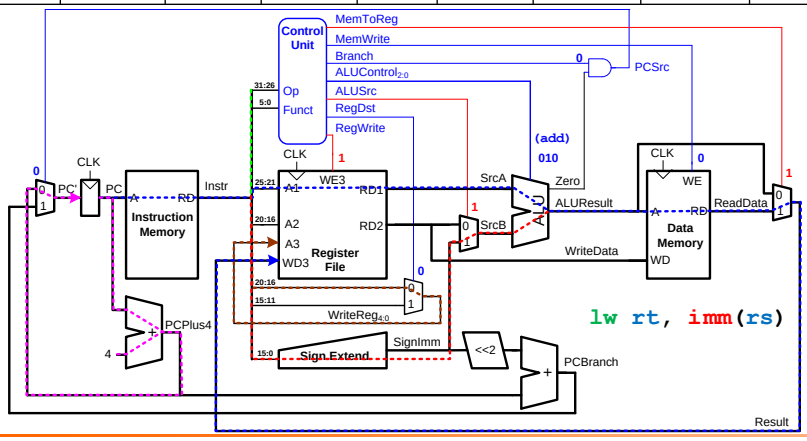
UC (9) - Main Decoder (3) - Tipo-R - or

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							



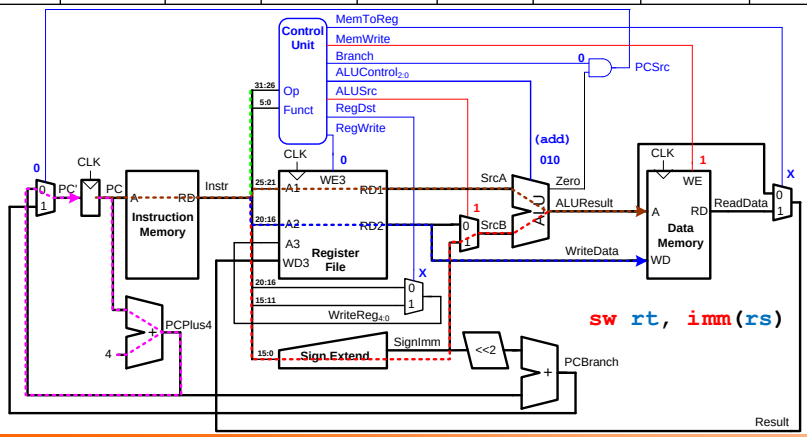
UC (10) - Main Decoder (4) - lw

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							



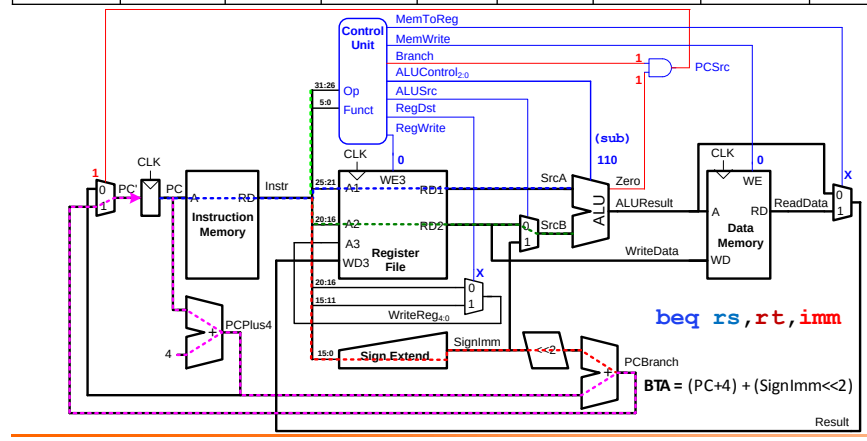
UC (11) - Main Decoder (5) - sw

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							



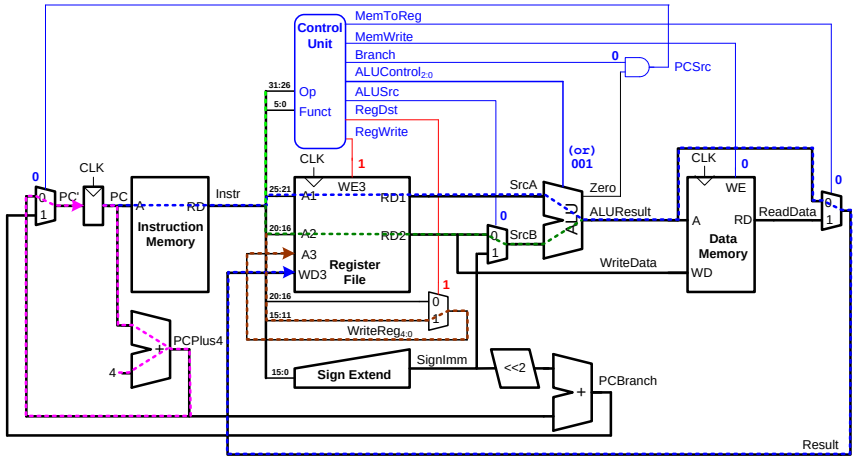
UC (12) - Main Decoder (6) - beq

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



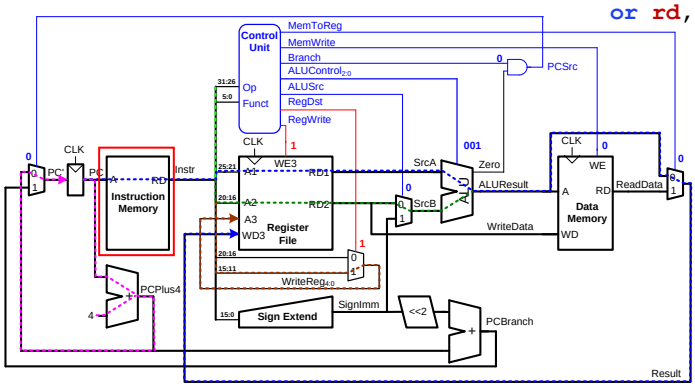
Exercício - OR (1) - Tipo-R

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



Exercício - OR (2) - Fetch

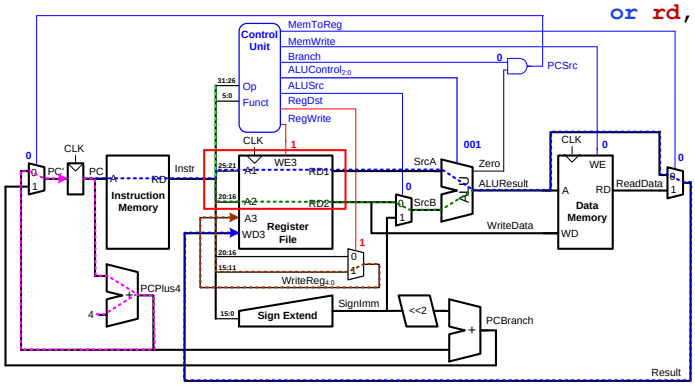
or rd, rs, rt



1. O PC aponta para a posição de memória que contém a instrução; esta fica disponível na saída da Memória de Instruções.

Exercício - OR (3) - Leitura dos Operandos

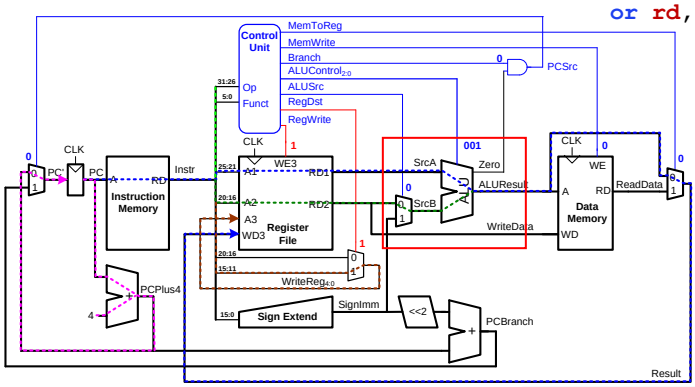
or rd, rs, rt



2. Os campos da instrução Instr25:21 e Instr20:16 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) - Execução na ALU

or rd, rs, rt



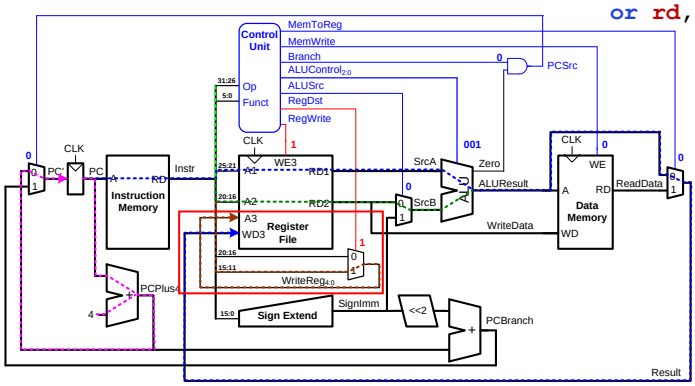
3. ALU: SrcA = RD1 e SrcB = RD2 (ALUSrc = 0).

OR é do Tipo-R, logo ALUOp=10, e 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 do Resultado no RF

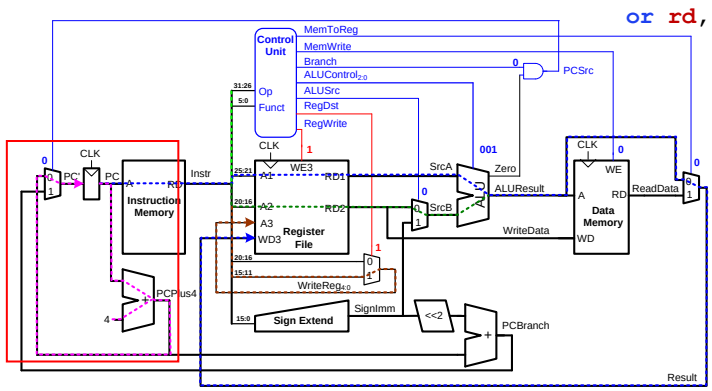
or rd, rs, rt



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

4. O Result é escrito no Banco de Registos. O registo destino rd, o campo Instr15:11, 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

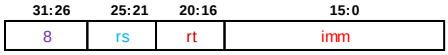
Até aqui considerámos um subconjunto limitado de instruções do MIPS.

Para ilustrar como adicionar novas instruções ao CPU, vamos acrescentar suporte para duas instruções addi e j.

Veremos que para um tipo de instruções basta adicionar algumas linhas ao *Main Decoder*, ao passo que para outras o *datapath* também precisa de ser alterado.

Mais Instruções (2) - addi (1)

A instrução '*add immediate*', addi, adiciona ao valor dum registo (rs) o valor imediato e escreve o resultado noutro registo (rt).



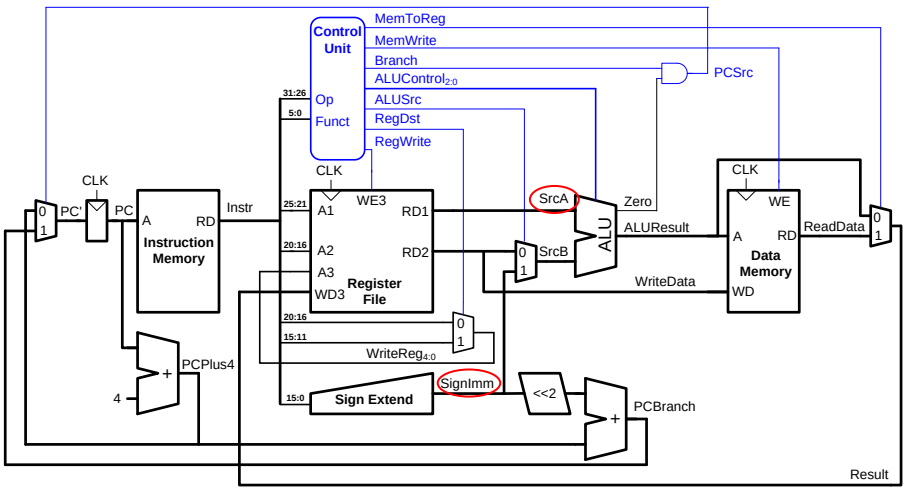

```
addi rt,rs,imm
```

O *datapath* atual já é capaz de desempenhar esta tarefa.

P: Que alterações são necessárias introduzir na Unidade de Controlo para suportar addi?

R: Precisamos somente de acrescentar 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

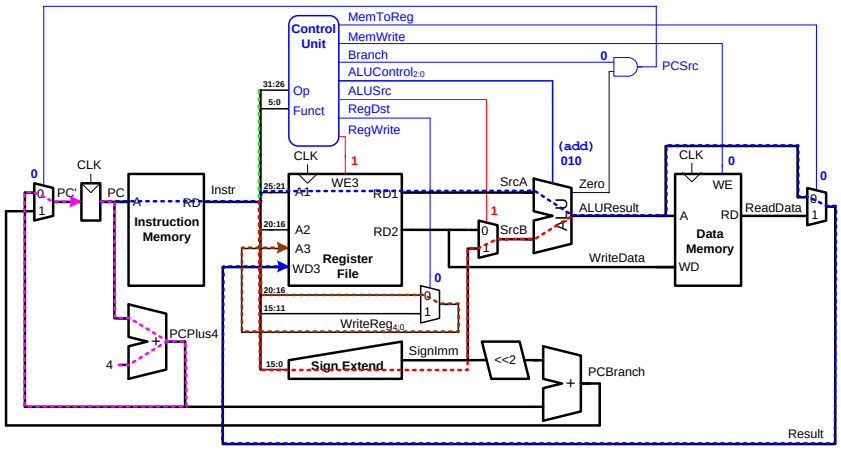
Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemToReg	$ALUOp_{1:0}$
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
addi	001000	1	0	1	0	0	0	00

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 instrução não é um *branch*, nem escreve na memória, Branch = MemWrite = 0.
5. O resultado vem da ALU, não da memória, MemToReg = 0.
6. A ALU deve somar, ALUOp = 00, e o valor de ALUControl = 010 (ver *ALU Decoder*)

```
addi rt,rs,imm
```


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

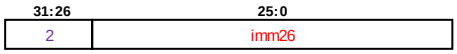
```
addi rt,rs,imm
```



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

Mais Instruções (6) - j (1)

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



`j imm26`

PC'

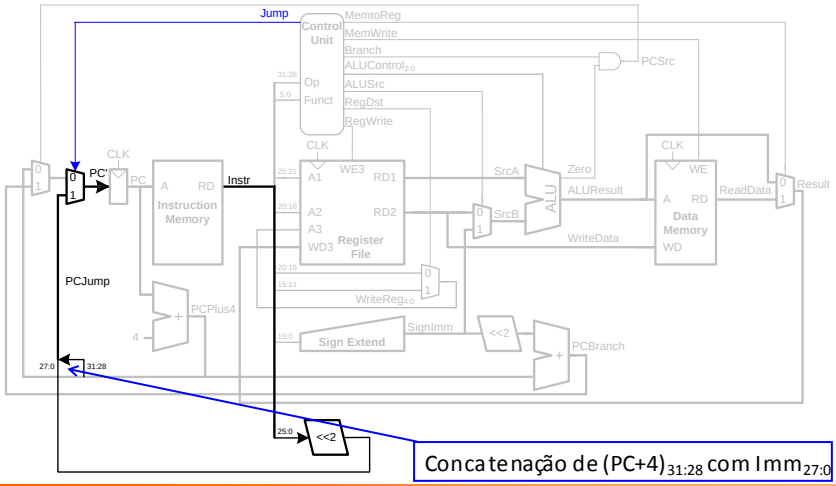
- Os dois bits menos significativos são sempre 0, porque o valor do PC é *word-aligned*, i.e., é sempre um múltiplo de 4 bytes.
- Os 26 bits seguintes são retirados do valor imediato da instrução Instr25:0.
- Os 4 bits mais significativos são iguais aos 4 bit mais significativos do PC anterior + 4.
-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

$$PC' = (PC + 4)_{31..28} : (Imm26 << 2)$$

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-P1: Primeiro, adicionamos *hardware* ao *datapath* 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*.



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

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

R-P2: 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 (*don't care*) o cálculo feito no datapath, donde RegDst = ALUSrc = Branch = MemToReg =ALUOp = X.