Arquitetura de Computadores

Microarquitetura

(MIPS)

UFMT

Fevereiro de 2025

Agenda

- **Introdução.**
- Processador MIPS de Ciclo Único (Single-Cycle Processor).
- Análise de Performance.

Introdução

Introdução

- O nível de **Microarquitetura** é voltado para a implementação em hardware de uma arquitetura.
- Consiste em um arranjo específico de registradores, ULAs, multiplexadores, memórias e outros blocos de construção necessários para implementar uma arquitetura.
- Cada microarquitetura é projetada de modo a atender requisitos específicos de **custo**, **desempenho** e **consumo energético**.

Componentes Gerais do Processador

- O processador é composto por duas partes principais:

1. **Caminho de Dados** (**Datapath**): É formado por blocos funcionais interligados, incluindo memórias, registradores, ULAs e multiplexadores, e tem o propósito de executar instruções. O MIPS é uma arquitetura de 32 bits, por isso o datapath processa dados de 32 bits.

2. **Unidade de Controle**: Interpreta a instrução atual e gera os sinais de controle para coordenar a execução no Caminho de Dados. Por exemplo, os sinais de controle incluem valores para seletores de multiplexador, habilitação de gravação em registradores e memória, entre outros.

Estado arquitetural

- O **Estado Arquitetural** caracteriza o ponto de execução de um programa. É formado pelos valores armazenados nos registradores e na memória, consistindo na informação necessária para descrever a operação do processador em um determinado momento.

- No caso do MIPS, o estado arquitetural de um programa é definido pelos valores do PC, 32 registradores do Register File e pelo conteúdo da memória. Qualquer microarquitetura MIPS deve ter recursos para salvar e restaurar o estado arquitetural de um programa.

- Com base no estado arquitetural atual, o processador executa uma instrução específica com um conjunto específico de dados. O resultado da execução da instrução gera um novo estado arquitetural, isto é, pode modificar o valor dos registradores e da memória.

Microarquiteturas MIPS que Estudaremos

- Uma mesma arquitetura pode ser implementada de diferentes formas, cada uma com diferentes requisitos de desempenho, custo e complexidade. Todos elas executam os mesmos programas, mas seus designs internos variam bastante.

- Vamos estudar três microarquiteturas MIPS:
  - **Microarquitetura de Ciclo Único**: A execução de cada instrução é executada em um único ciclo de Clock.
  - **Microarquitetura Multiciclo**: A execução de cada instrução é quebrada em uma sequência passos mais curtos, cada qual ocupando um ciclo de clock.
  - **Microarquitetura com Pipeline**: A execução de cada instrução é quebrada em uma sequência de passos mais curtos e múltiplas instruções são executadas simultaneamente.

Processador MIPS de Ciclo Único

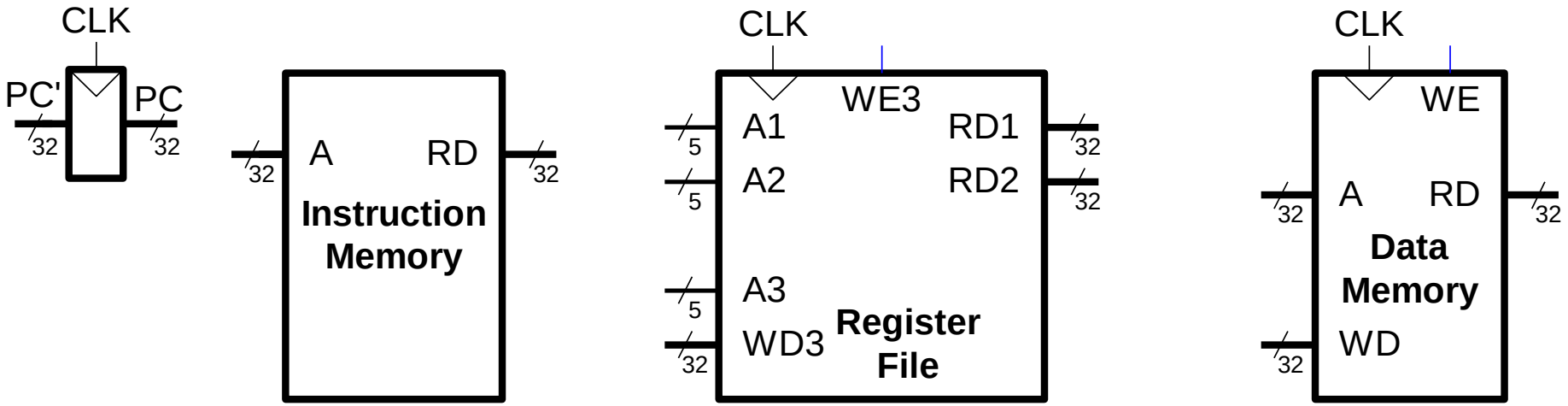
(Single-Cycle Processor)

Instruções Implementadas

- Inicialmente, estudaremos um Processador MIPS de Ciclo Único, projetado para executar um subconjunto das instruções da ISA:
  - Instruções lógicas e aritméticas: `and`, `or`, `add`, `addi`, `sub`, `slt`.
  - Instruções de acesso à Memória: `lw`, `sw`.
  - Instruções de desvio: `beq` e `j`.
- Estas instruções foram escolhidas porque são suficientes para escrever muitos programas interessantes.
- Depois de entender como implementar essas instruções, você mesmo poderá expandir o hardware para lidar com outras instruções.

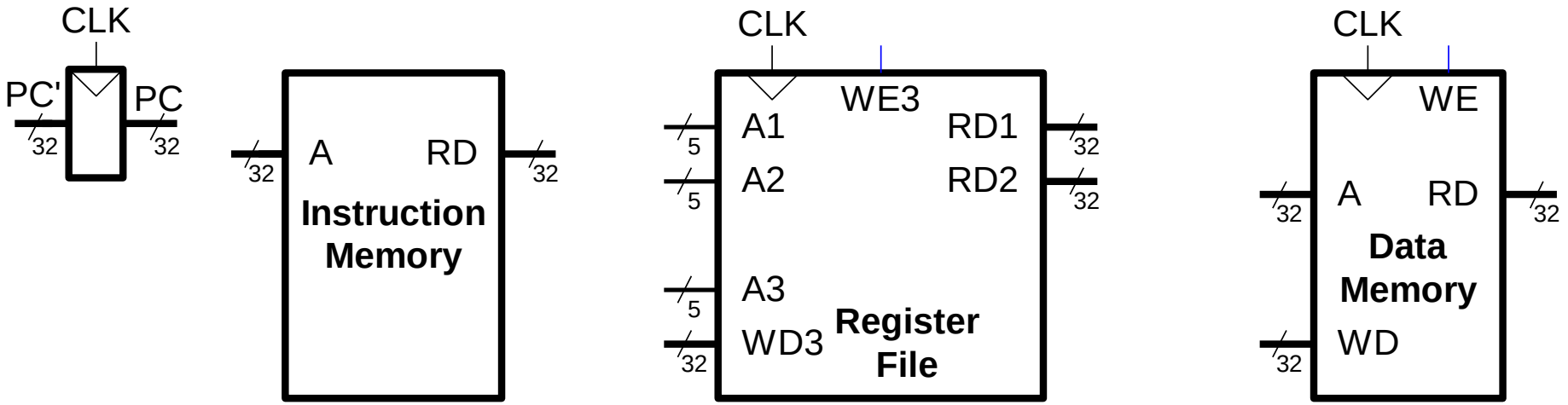
Elementos de Estado do Processador

- Nesta primeira microarquitetura MIPS, a memória é dividida em duas partes: **Memória de instruções** (read only) e **Memória de dados**.
- O **Caminho de Dados** consiste na interligação desses elementos de armazenamento de modo a possibilitar a execução das instruções.



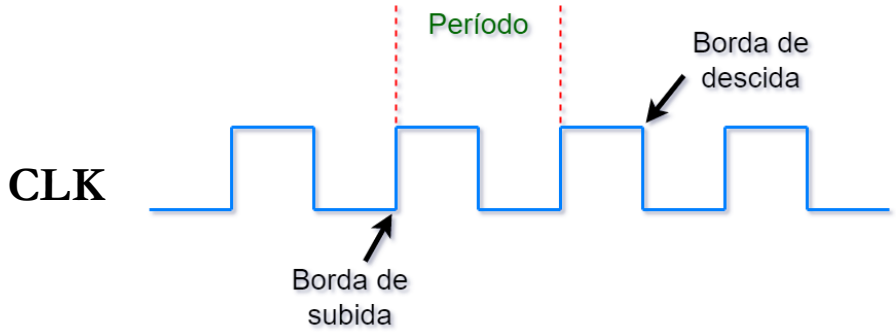
Elementos de Estado do Processador

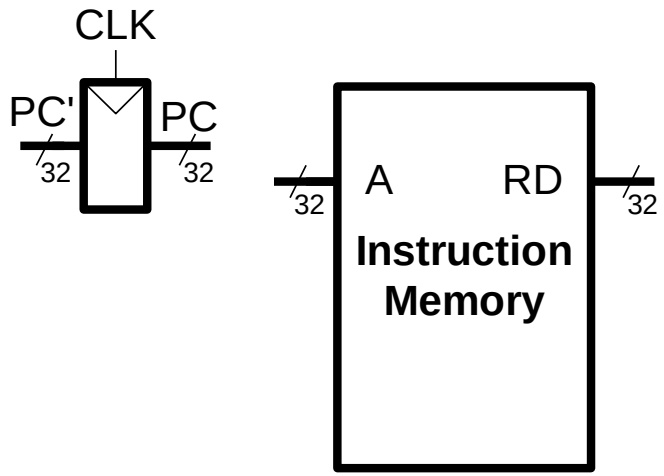
- A **leitura** da Memória de Instrução, do Register File e da Memória de Dados são **combinacionais**.
- Significa que se o endereço mudar, os novos dados aparecerão nas saídas RD, após algum atraso de propagação, sem que o sinal de clock esteja envolvido.

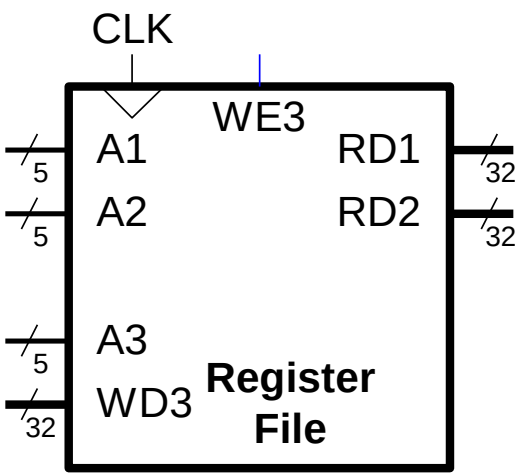


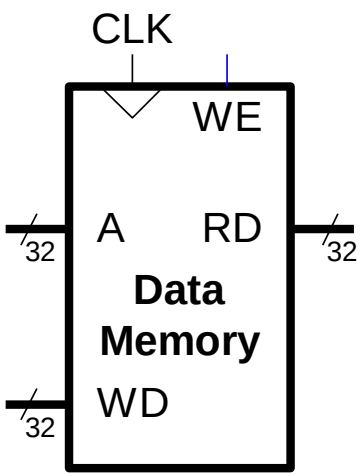
Elementos de Estado do Processador

- As **gravações** são sincronizadas pelo sinal de clock, acontecendo na borda de subida, no final de cada ciclo.



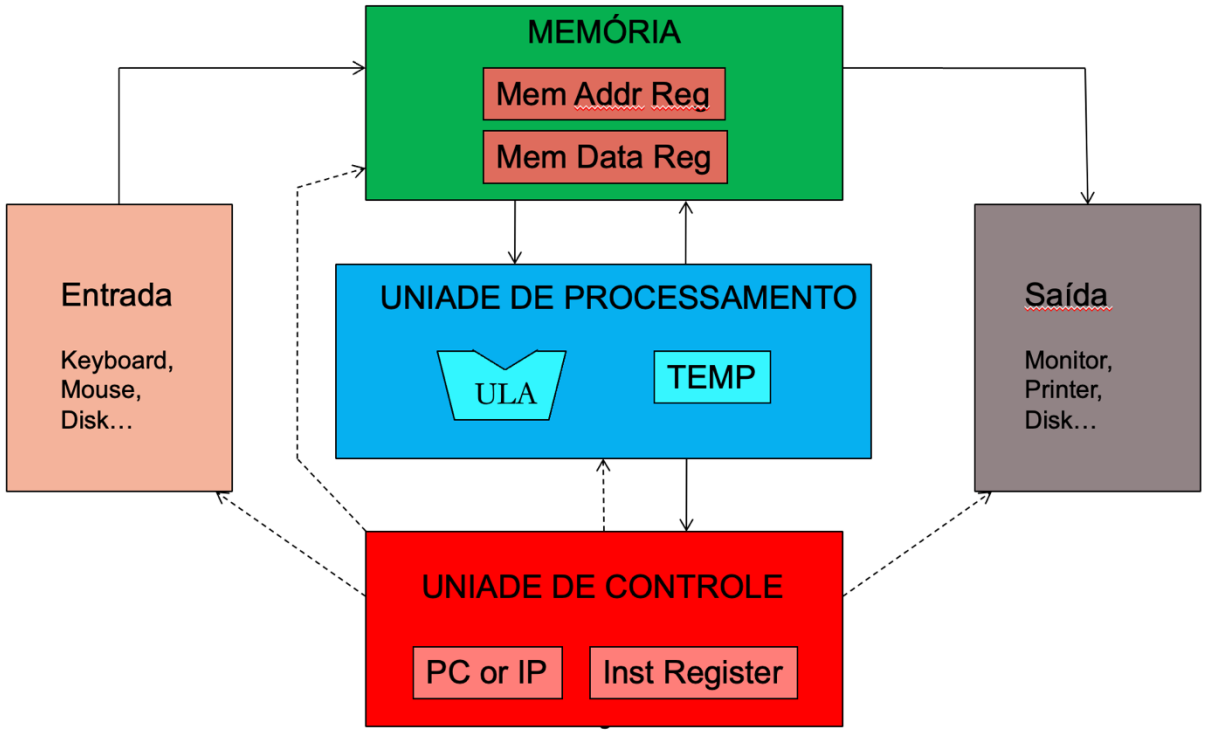






Relembrando: Programa Armazenado & Fases do Ciclo de Execução de Instruções

Ciclo de Instrução



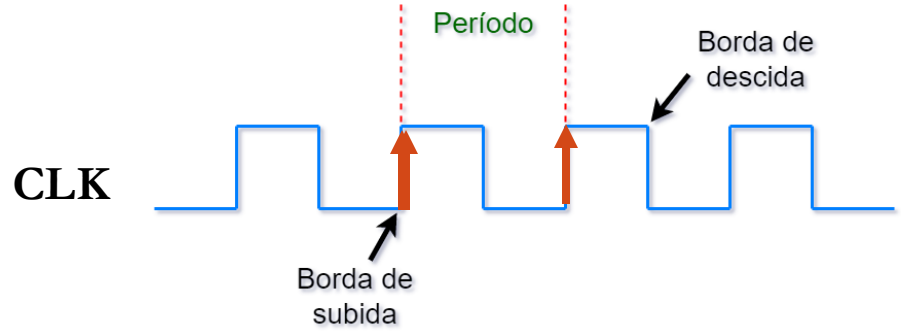
Datapath de Ciclo Único

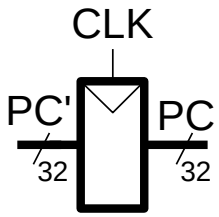
Fetch

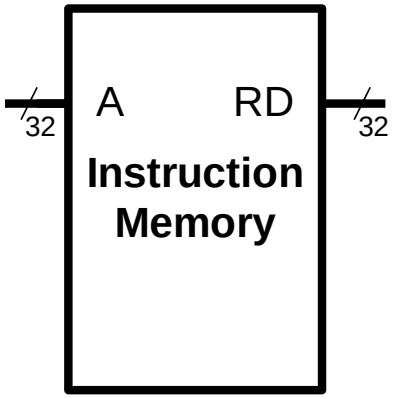
(Busca de Instrução)

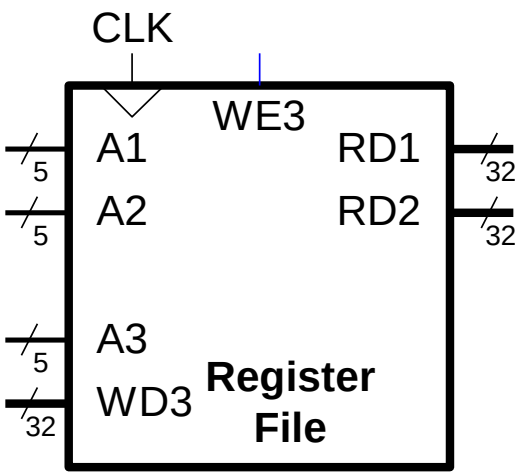
Fetch (busca de Instrução)

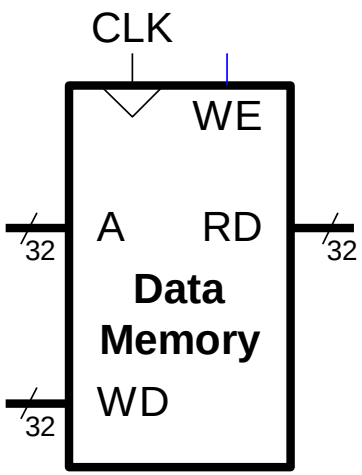
- No começo de um ciclo de clock, o PC armazena o endereço da próxima instrução a ser executada. Como usar esse endereço para ler a instrução da memória?





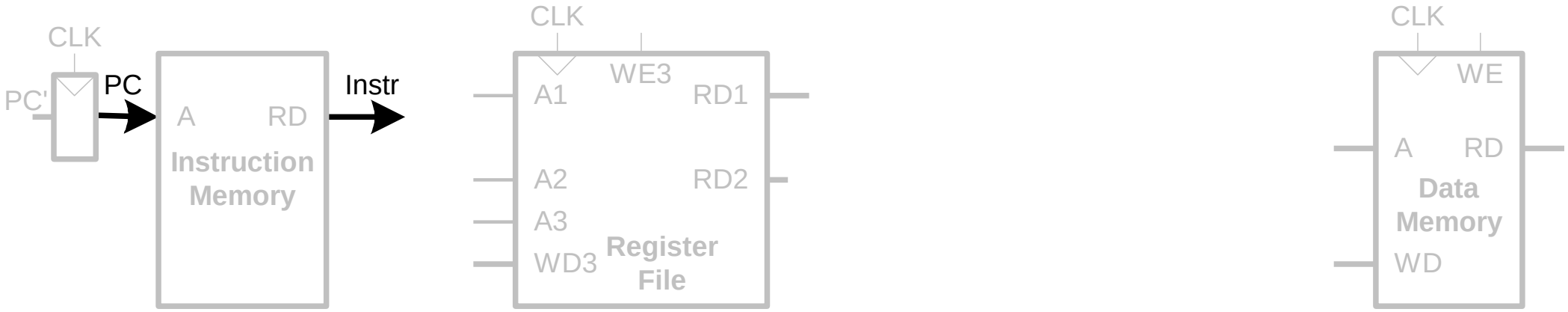






Busca de Instrução - Fetch

- Portanto, o valor no PC deve ser usado para endereçar a Memória de Instrução para que a próxima instrução seja lida.
- Após um atraso de propagação relacionado à leitura da memória, os 32 bits que compõe a instrução são disponibilizados em RD.



Datapath de Ciclo Único

`lw`

(Instrução Load Word)

Instrução `lw`

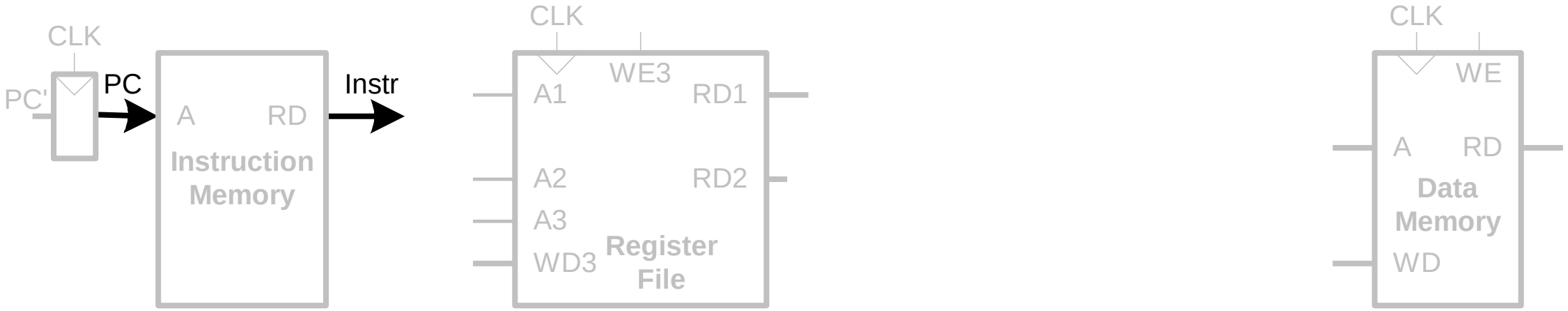
- `lw` é uma instrução Tipo-I usada para carregar um registrador do Register File (`rt`) com o valor armazenado em uma palavra de memória.
- A palavra de memória a ser carregada é endereçada de acordo com o modo de endereçamento *registrador base* + *offset*.
- O endereço é calculado somando o valor contido no registrador `rs` com o imediato (sign-extended(`imm`)) obtido da própria instrução.

I-Type


```
lw rt, imm(rs)
```

Datapath para `lw`

- **Passo 1:** primeiramente, o registrador `rs` precisa ser lido do Register File. Para isso, os bits 25:21 da instrução devem alimentar a porta A1 do Register File. Como resultado, o conteúdo do registrador é disponibilizado em RD1.



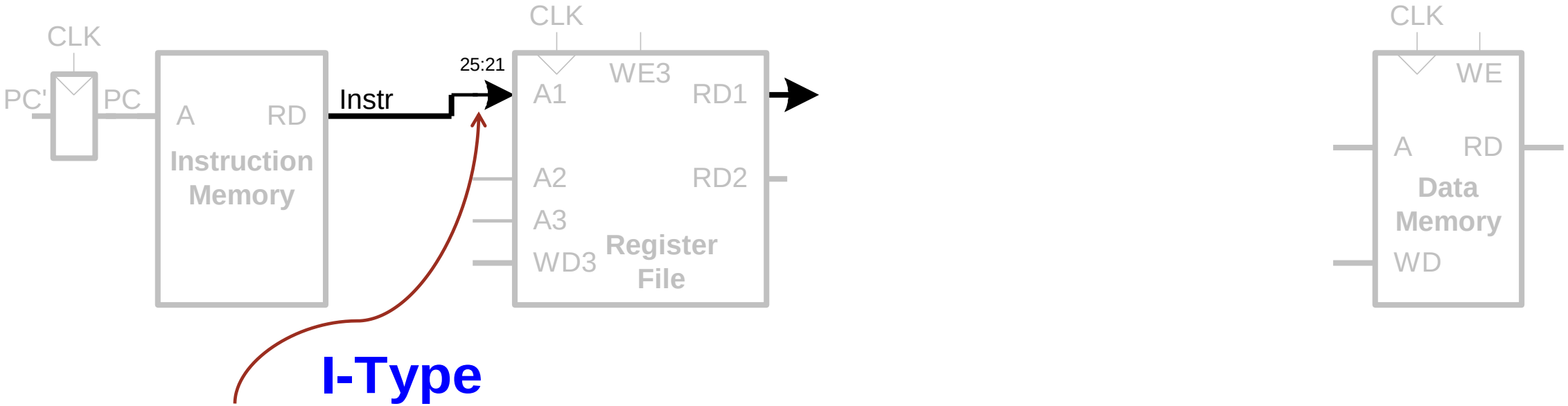
I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

```
lw rt, imm(rs)
```

Datapath para `lw`

- **Passo 1:** primeiramente, o registrador `rs` precisa ser lido do Register File. Para isso, os bits 25:21 da instrução devem alimentar a porta A1 do Register File. Como resultado, o conteúdo do registrador é disponibilizado em RD1.



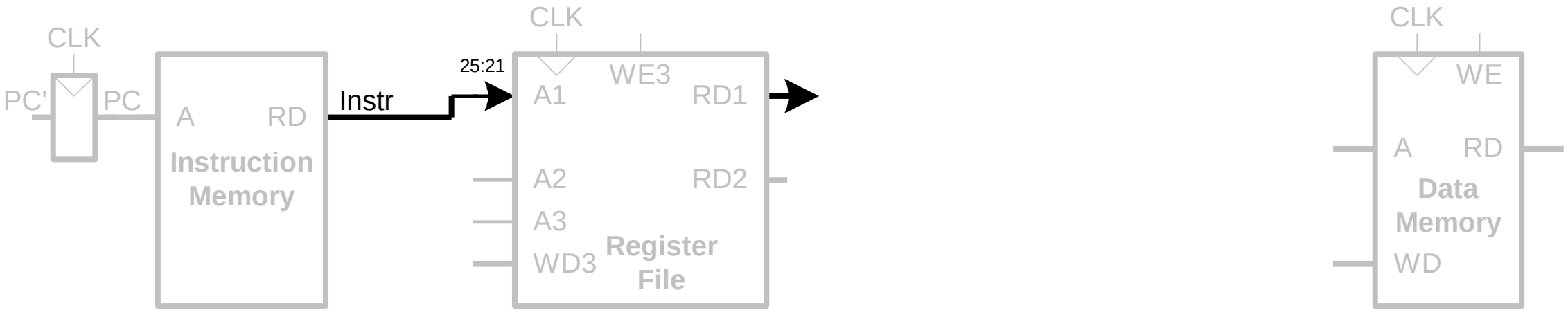
I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

```
lw rt, imm(rs)
```

Datapath para `lw`

Passo 2: O imediato da instrução (bits 15:0) é estendido (com sinal) para 32 bits.



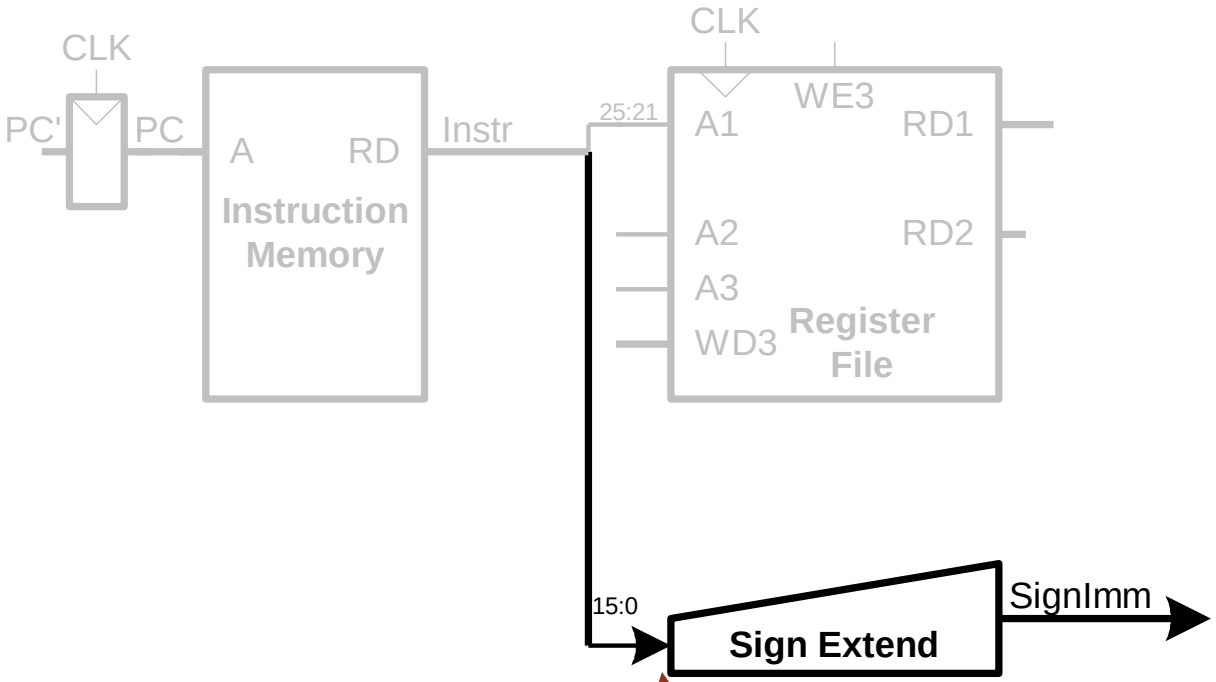
I-Type

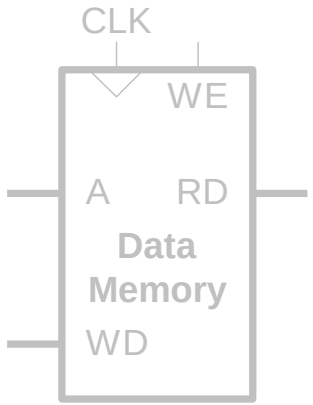
op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

```
lw rt, imm(rs)
```

Datapath para `lw`

Passo 2: O imediato da instrução (bits 15:0) é estendido (com sinal) para 32 bits.



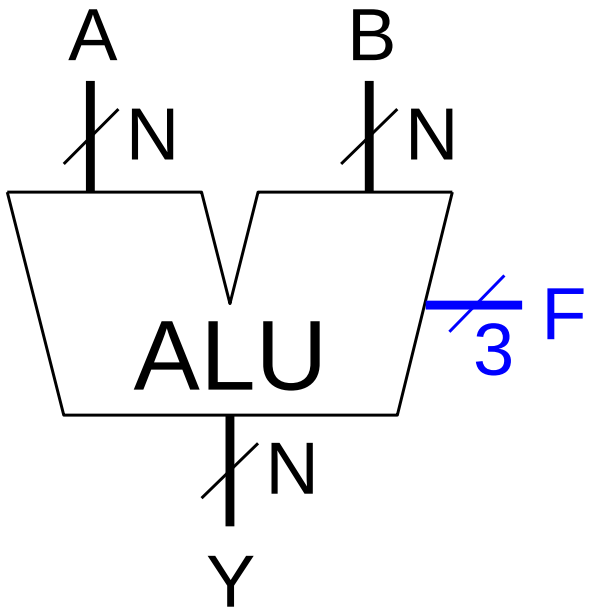


I-Type


```
lw rt, imm(rs)
```

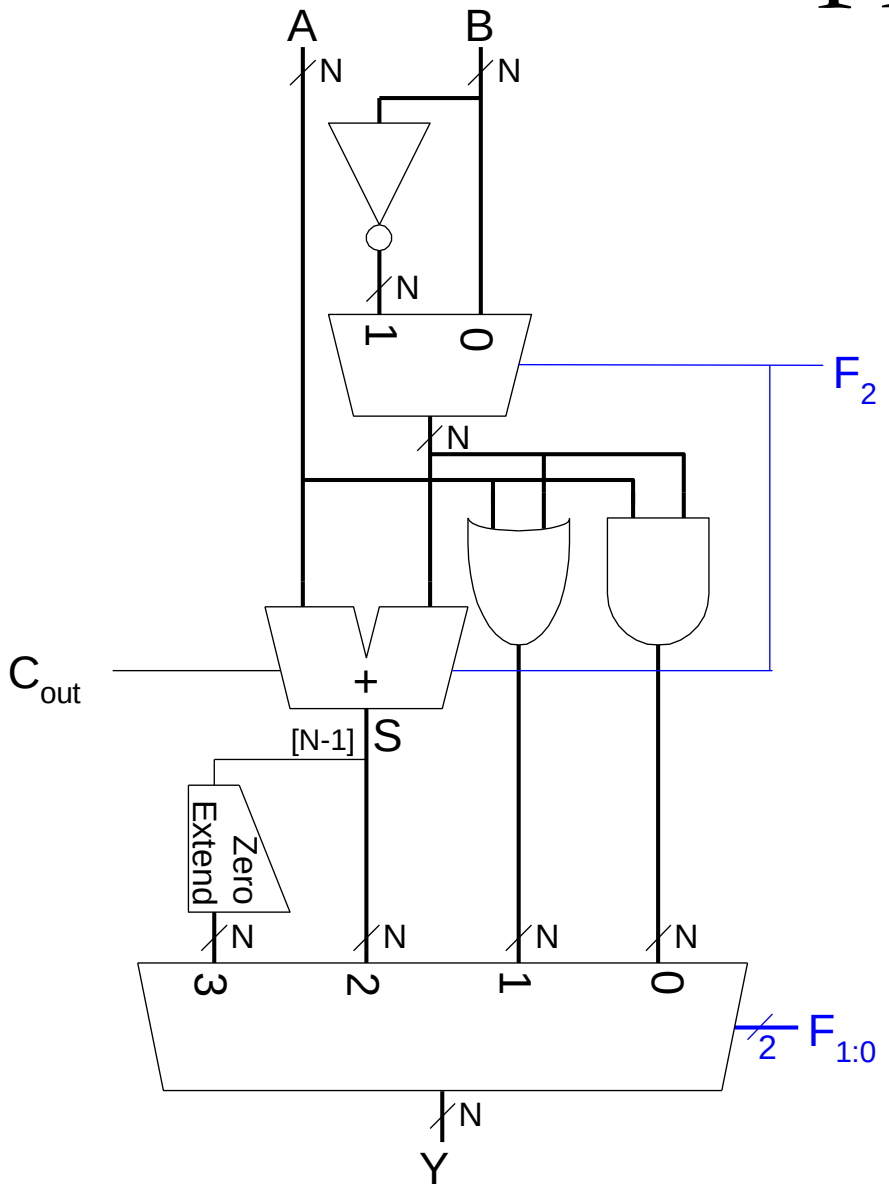
Datapath para `lw`

- **Passo 3**: precisamos de uma ULA que realize a soma de `rs` com o imediato estendido. Nesse projeto, a ULA realiza as operações descritas na tabela abaixo.



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

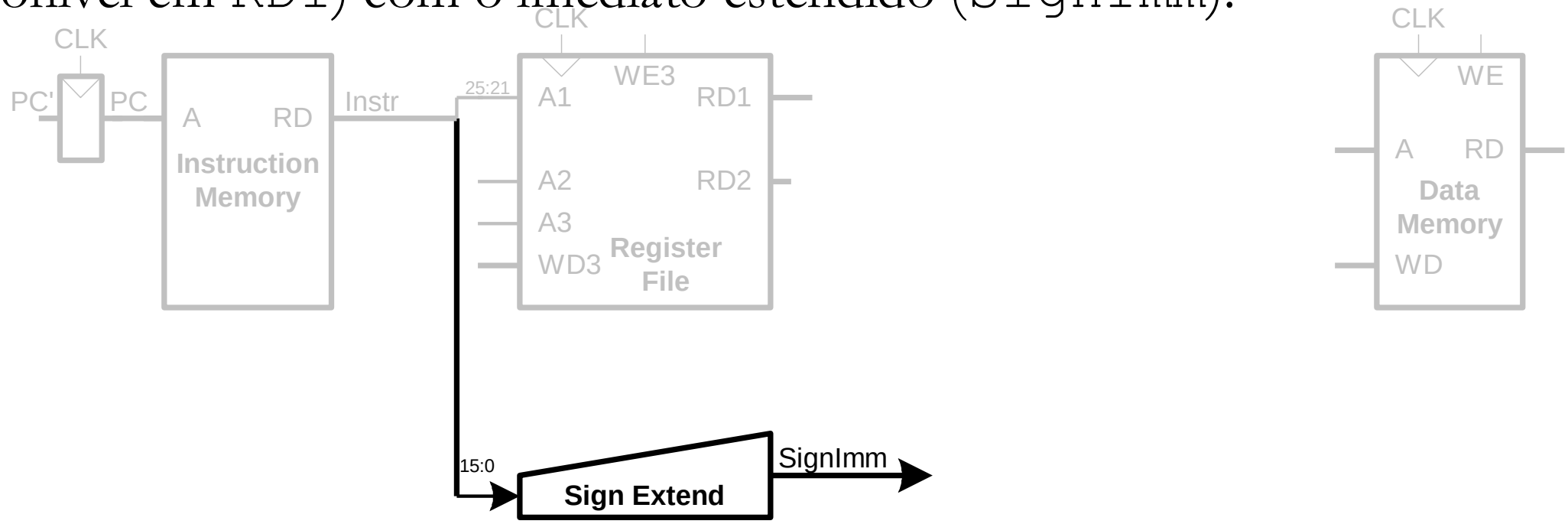
Projeto da ULA



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

Datapath para `lw`

Passo 3: A ULA calcula o endereço de memória somando o conteúdo de `rs` (disponível em `RD1`) com o imediato estendido (`SignImm`).



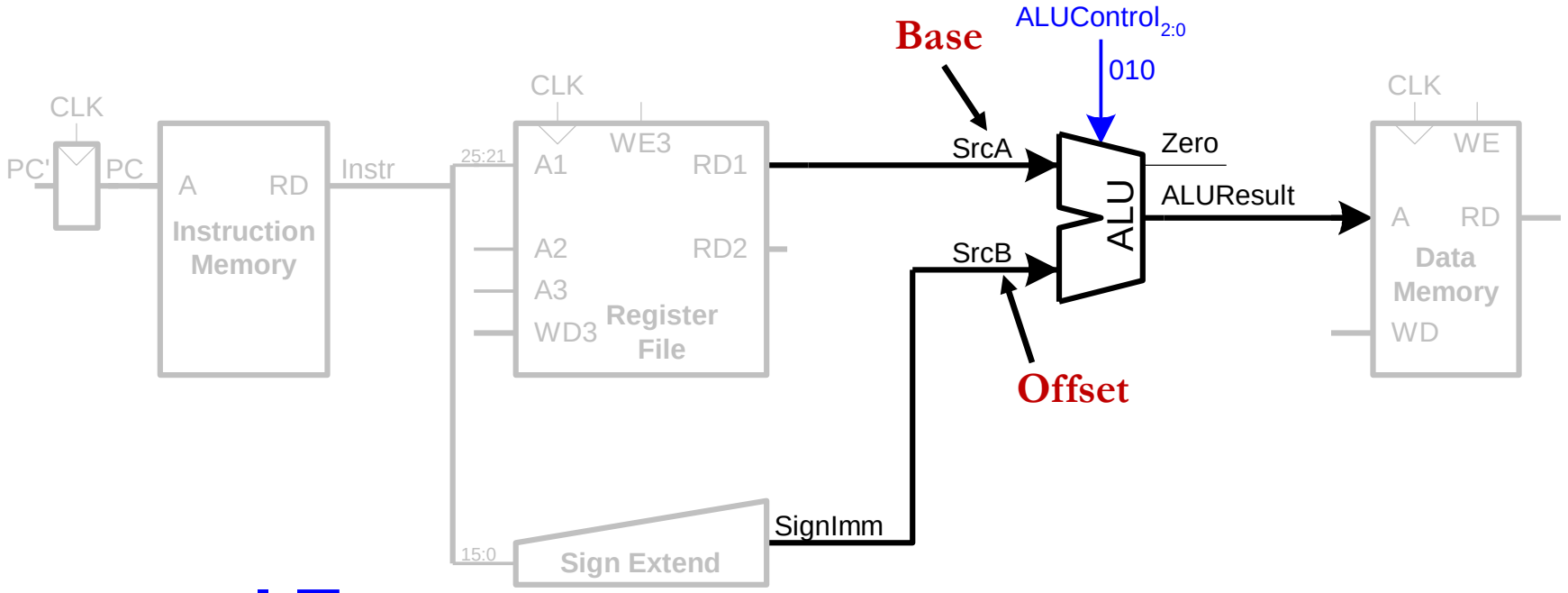
I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

```
lw rt, imm(rs)
```

Datapath para `lw`

Passo 3: A ULA calcula o endereço de memória somando o conteúdo de `rs` (disponível em `RD1`) com o imediato estendido (`SignImm`).



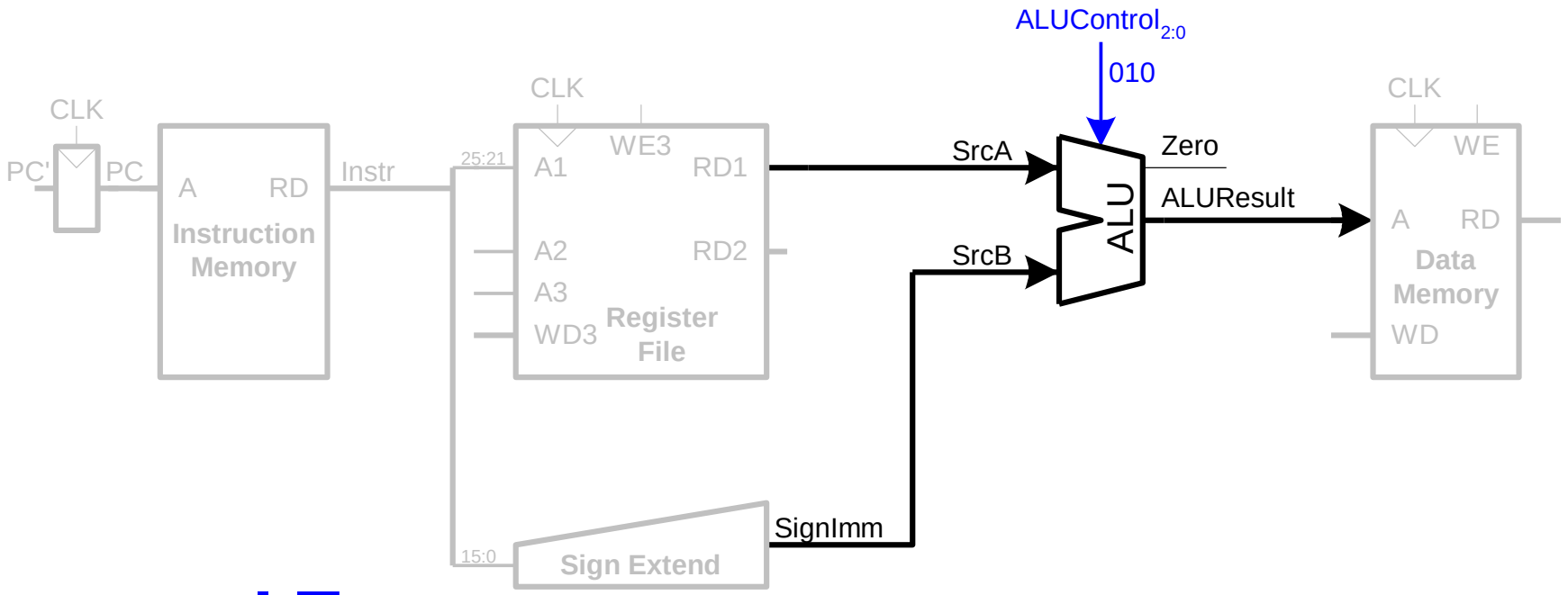
I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

```
lw rt, imm(rs)
```

Datapath para `lw`

- **Passo 4:** gravação no Register File, no registrador `rt` (bits 20:16), na borda de subida, no final do ciclo de clock, a palavra lida da memória de dados.



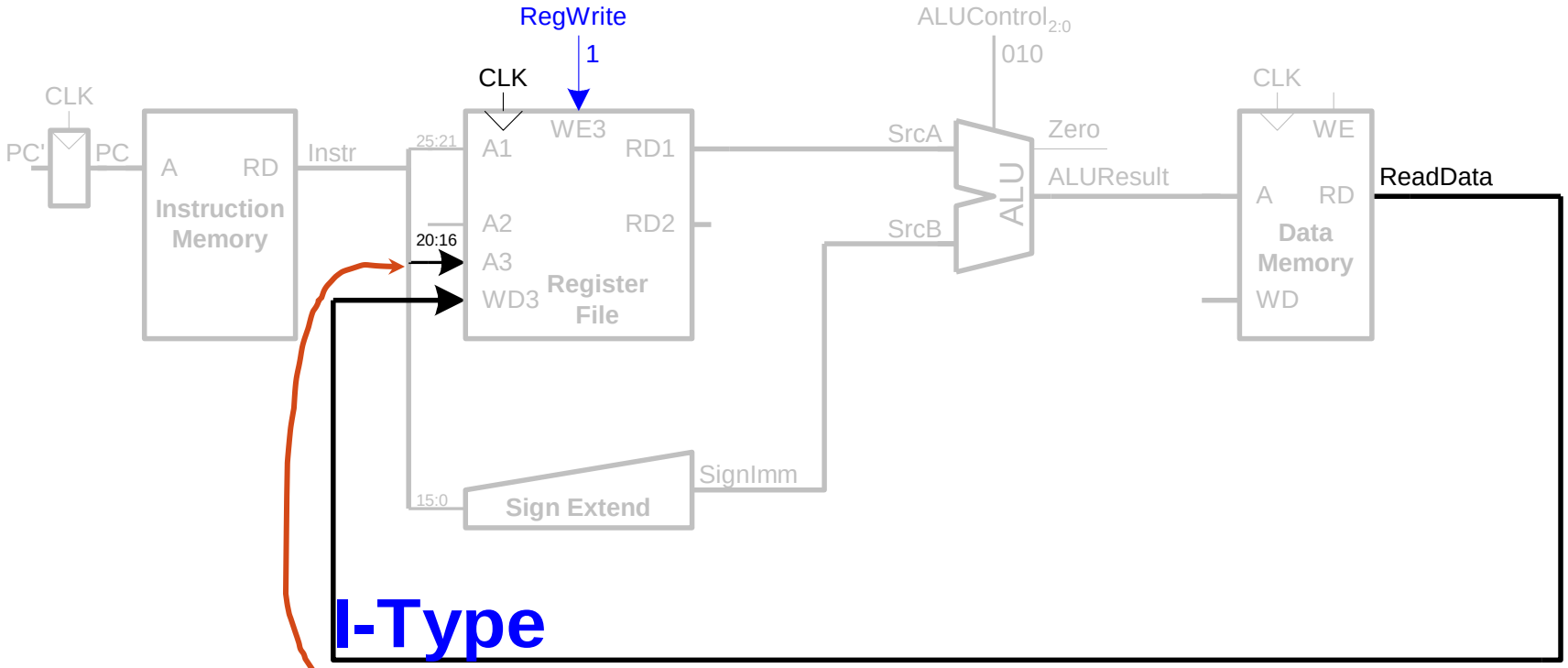
I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

```
lw rt, imm(rs)
```

Datapath para `lw`

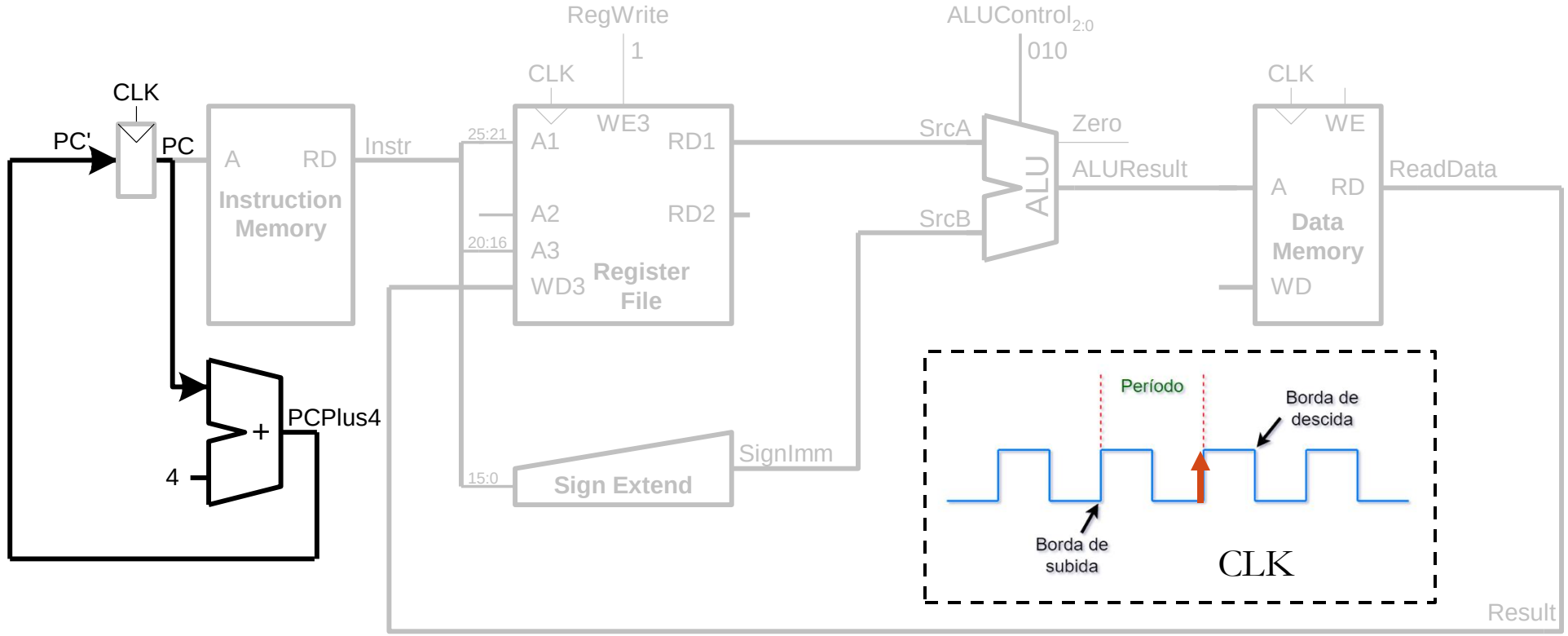
- **Passo 4:** grava no Register File, no registrador `rt` (bits 20:16), na borda de subida, no final do ciclo de clock, a palavra lida da memória de dados.




```
lw rt, imm(rs)
```

Incremento do PC

- **Incremento do PC:** Em paralelo à leitura da memória de instrução, no início do ciclo de clock, o valor armazenado em PC é incrementado. No final do ciclo de clock, na borda de subida, o PC é atualizado.



Datapath de Ciclo Único

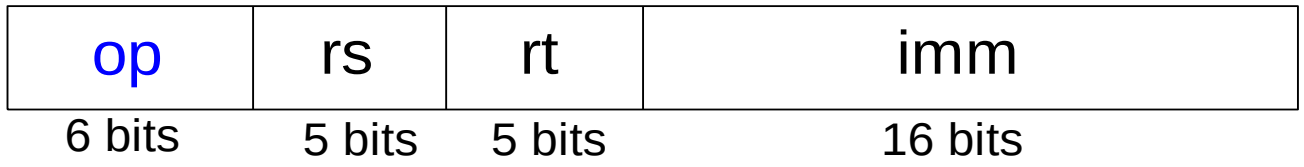
`sw`

(Instrução Store Word)

Instrução `sw`

- `sw` é uma instrução Tipo-I que grava em uma palavra da memória o conteúdo de um registrador do Register File (`rt`).
- A palavra da memória é endereçada de acordo com o modo de endereçamento *registrador base* + *offset*. Ou seja, o endereço é calculado somando o valor contido no registrador `rs` com uma constante (sign-extended(`imm`)) obtida da própria instrução.

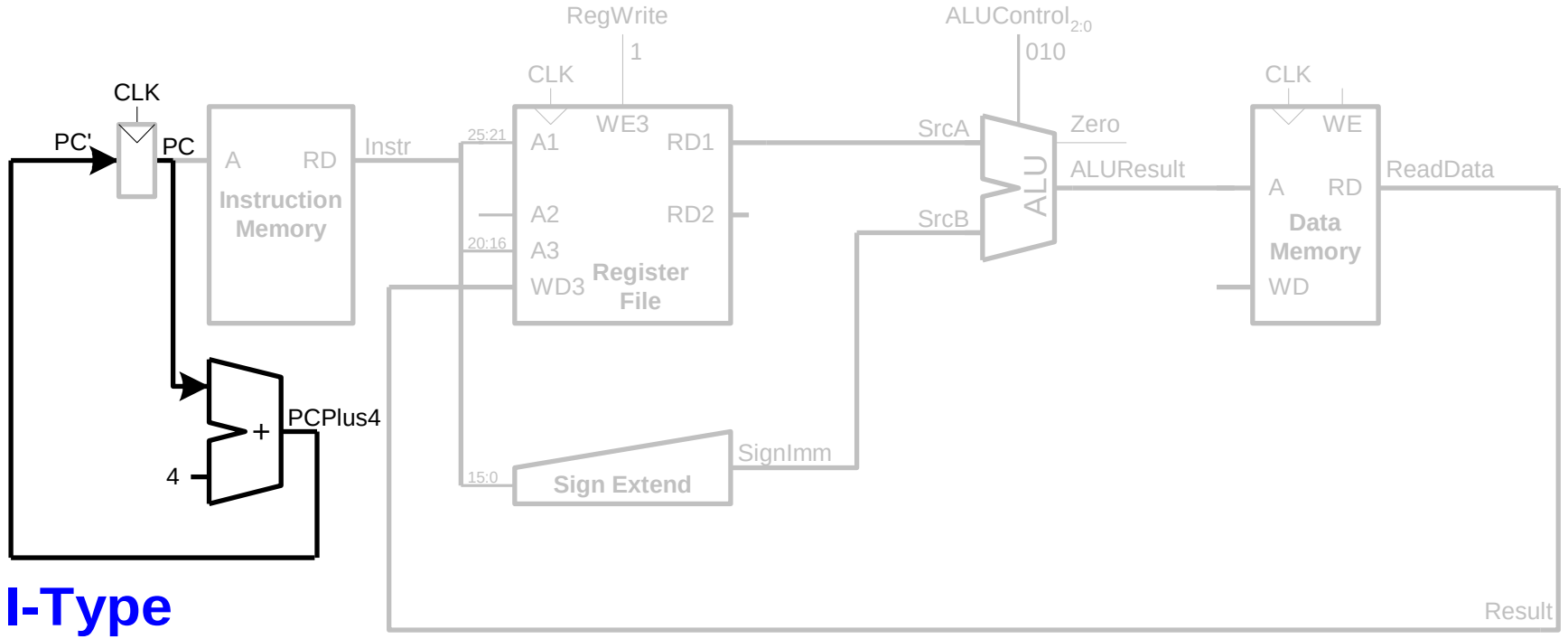
I-Type



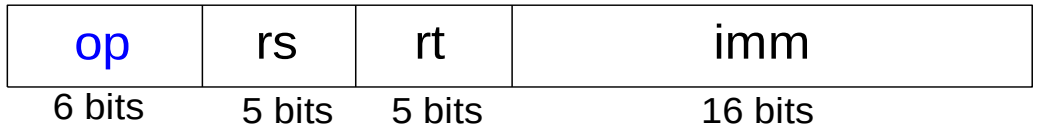

```
sw rt, imm(rs)
```

Instrução `sw`

- **Passo 1 e 2:** iguais aos da instrução `lw`. Resumindo, a ULA calcula um endereço de memória somando o conteúdo de `rs` com o imediato estendido. A seguir, na borda de subida, no final do ciclo de clock, o conteúdo de `rt` é gravado na memória de dados.



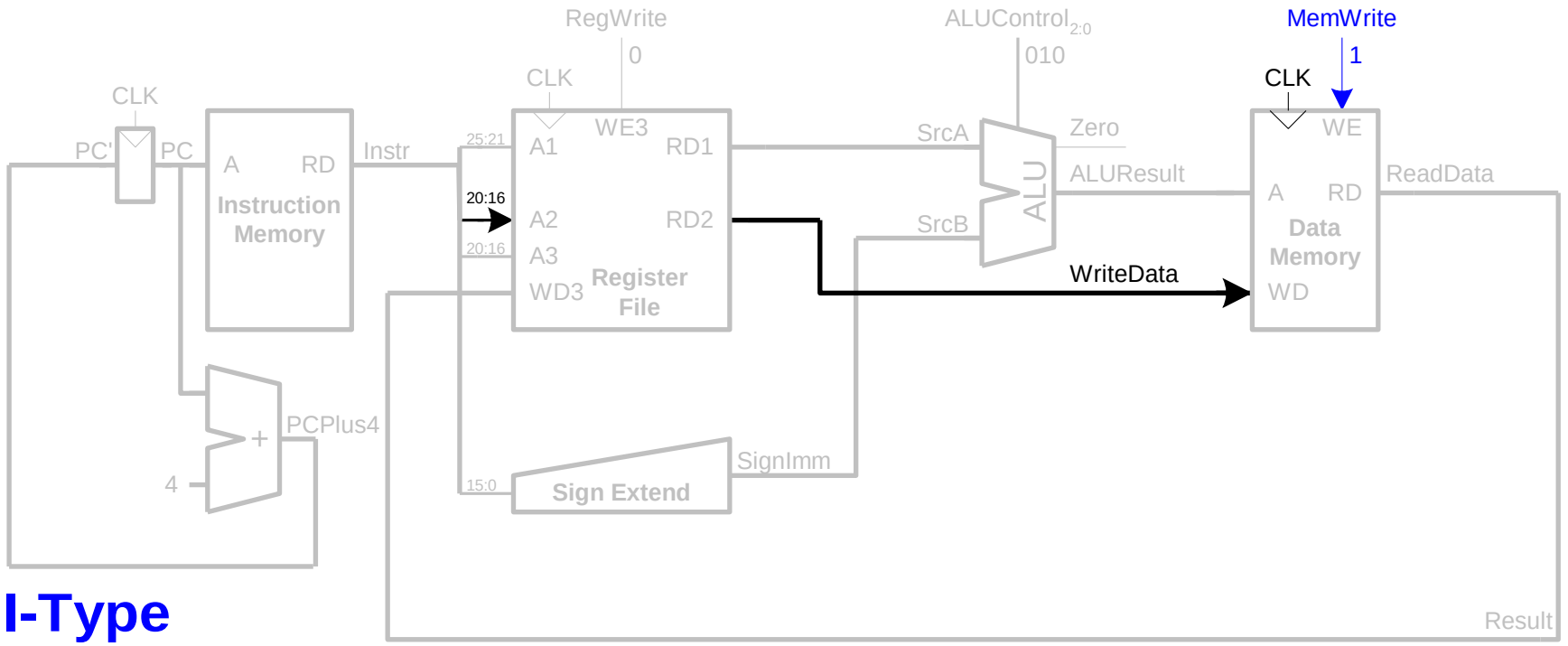
I-Type




```
sw rt, imm(rs)
```

Instrução `sw`

- **Passo 3:** na borda de subida, no final do ciclo de clock, o conteúdo de `rt` é gravado na memória de dados.



I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

```
sw rt, imm(rs)
```

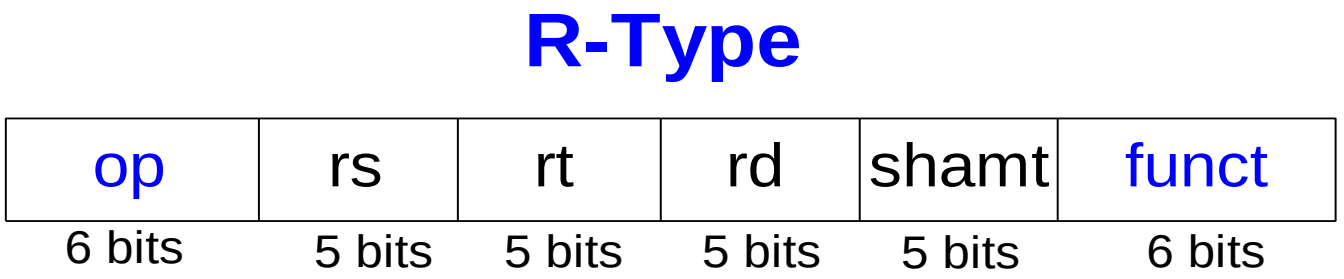
Datapath de Ciclo Único

Instruções Tipo-R

(`and`, `or`, `add`, `sub`, `slt`)

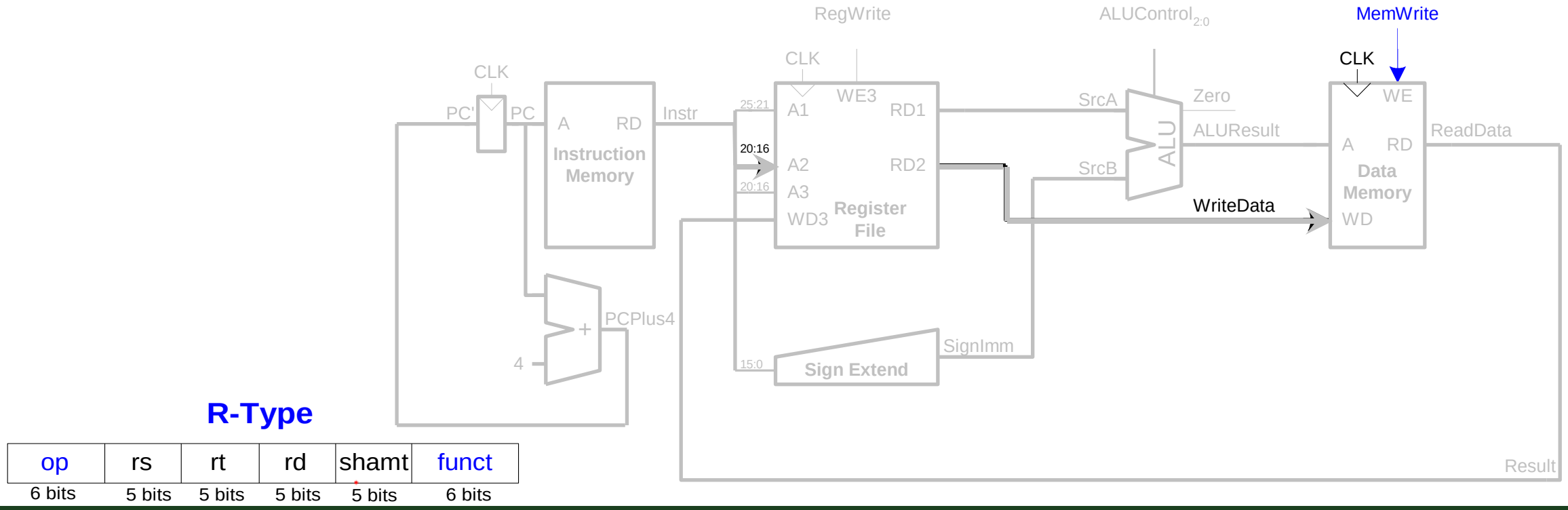
Instruções `and`, `or`, `add`, `sub`, `slt`

- Essas instruções Tipo-R operam da mesma forma:
  1. Leitura dos registradores `rs` (bits 25:21 da instrução) e `rt` (bits 20:16 da instrução).
  2. A ULA realiza a operação indicada pelo campo `funct` (bits 5:0 da instrução).
  3. O resultado é gravado no registrador `rd` (bits 15:11) no final do ciclo de clock.



Instruções `and`, `or`, `add`, `sub`, `slt`

- Essas instruções Tipo-R operam da mesma forma:
  1. Leitura dos registradores `rs` (bits 25:21 da instrução) e `rt` (bits 20:16 da instrução).
  2. A ULA realiza a operação indicada pelo campo `funct` (bits 5:0 da instrução).
  3. O resultado é gravado no registrador `rd` (bits 15:11) no final do ciclo de clock.

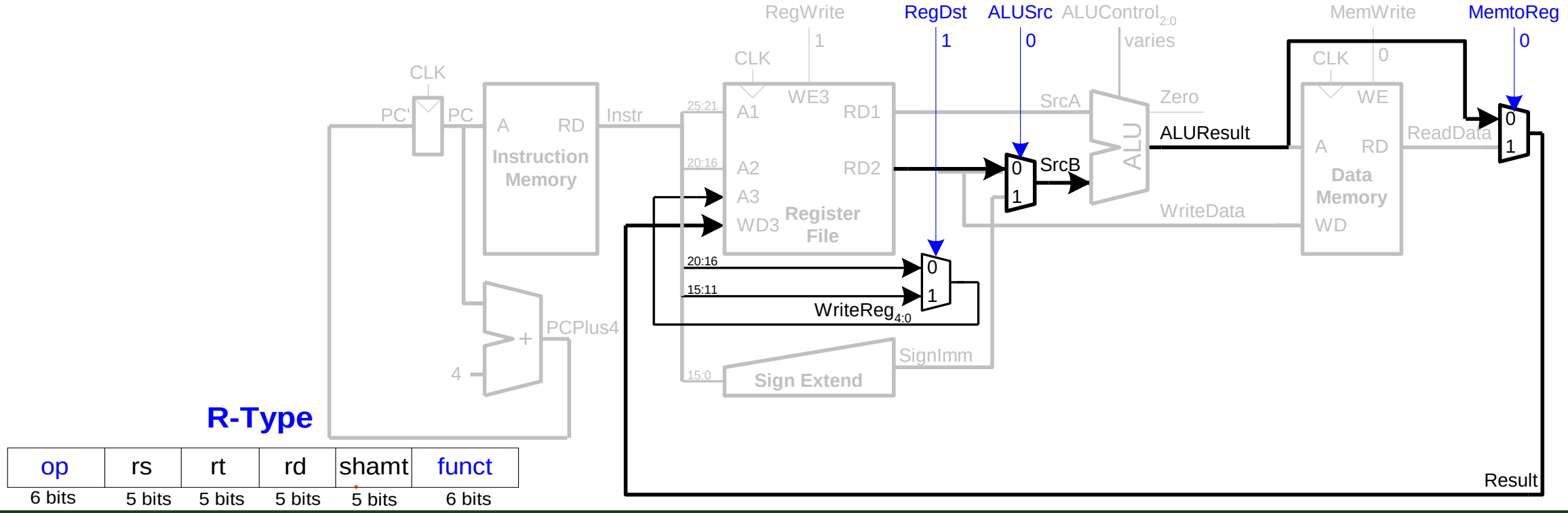


R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Instruções `and`, `or`, `add`, `sub`, `slt`

- Essas instruções Tipo-R operam da mesma forma:
  1. Leitura dos registradores `rs` (bits 25:21 da instrução) e `rt` (bits 20:16 da instrução).
  2. A ULA realiza a operação indicada pelo campo `funct` (bits 5:0 da instrução).
  3. O resultado é gravado no registrador `rd` (bits 15:11) no final do ciclo de clock.



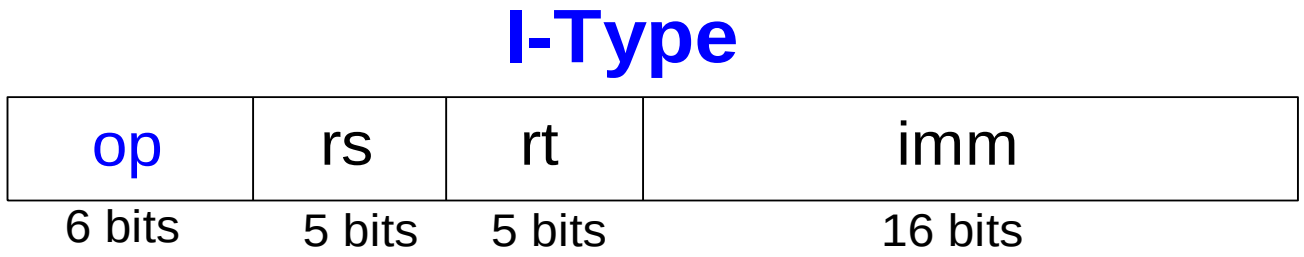
Datapath de Ciclo Único

beq

(Instrução de Desvio Condicional)

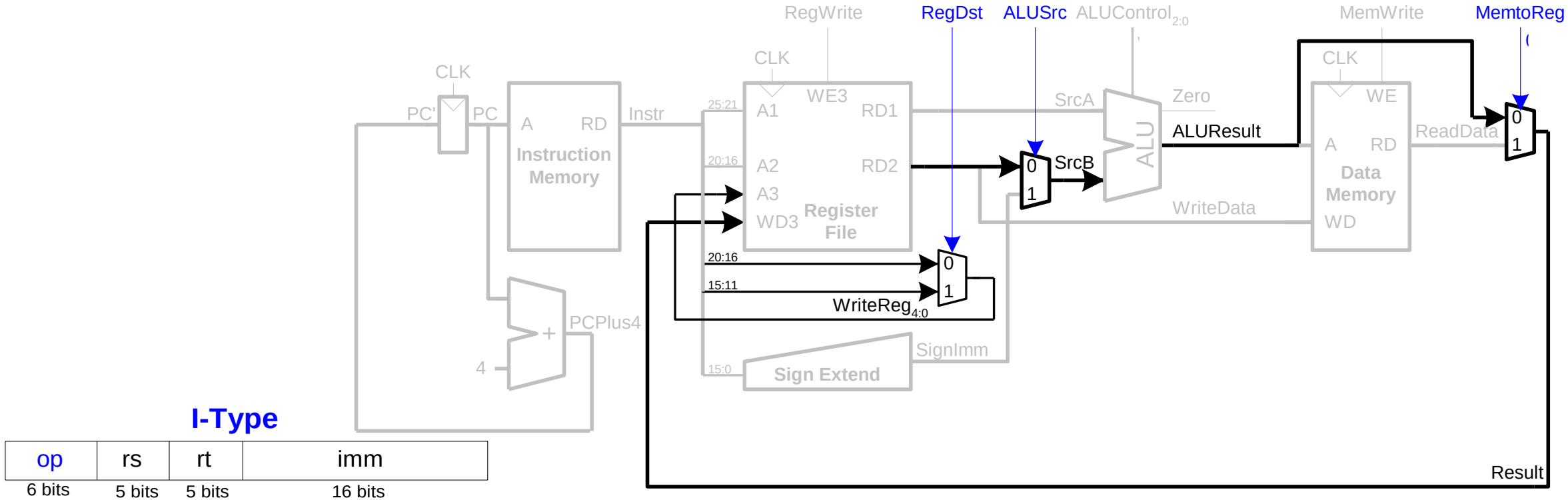
Instrução beq

- O processamento dessa instrução consiste em verificar se rs = rt. Caso isso seja verdadeiro, PC é atualizado com o endereço de desvio, chamado BTA.
- BTA (Branch Target Address) = (PC + 4) + sign-extended(`imm`) << 2.



Instrução beq

- O processamento dessa instrução consiste em verificar se rs = rt. Caso isso seja verdadeiro, PC é atualizado com o endereço de desvio, chamado BTA.
- BTA (Branch Target Address) = (PC + 4) + sign-extended(imm) << 2.

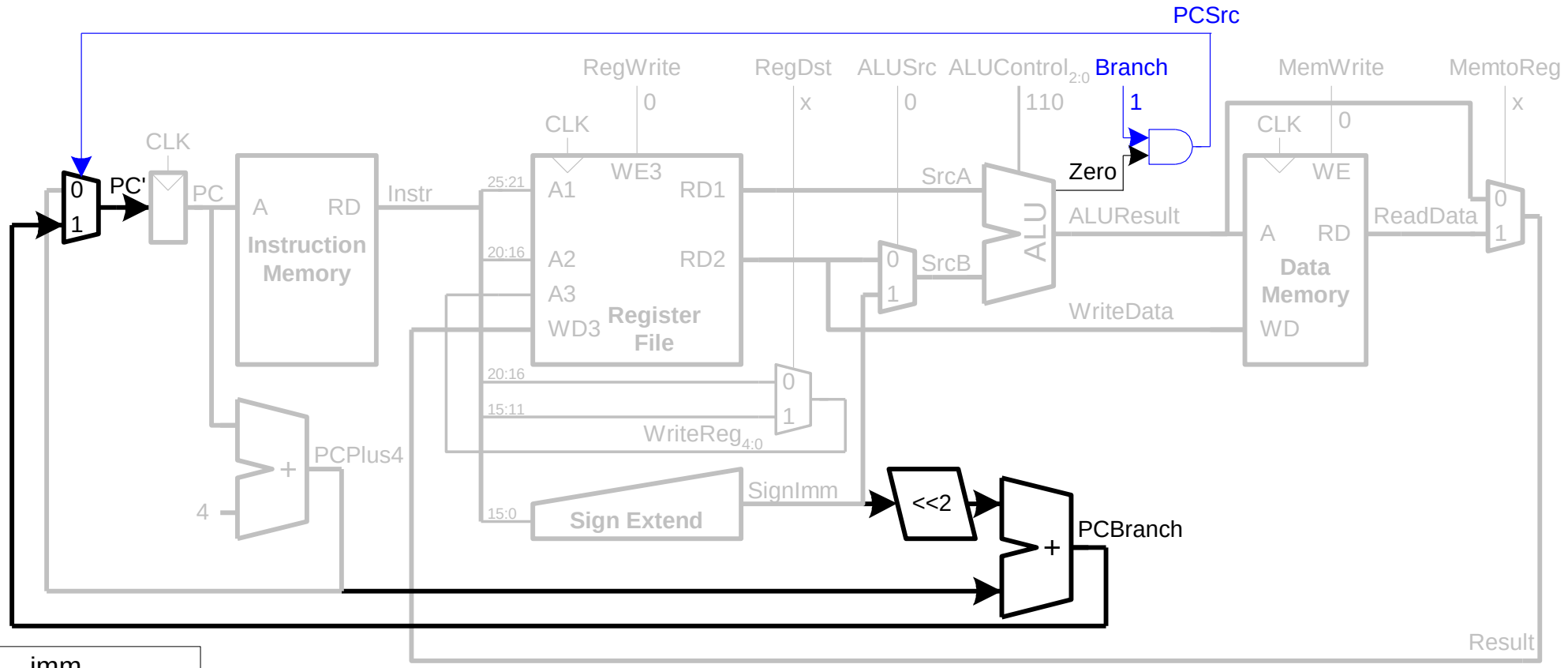


I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Instrução beq

- Para determinar se `rs` = `rt`, é feita uma **subtração**. Caso o resultado seja 0, o bit zero recebe 1.
- BTA (Branch Target Address) = (PC + 4) + sign-extended(`imm`) << 2.



I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

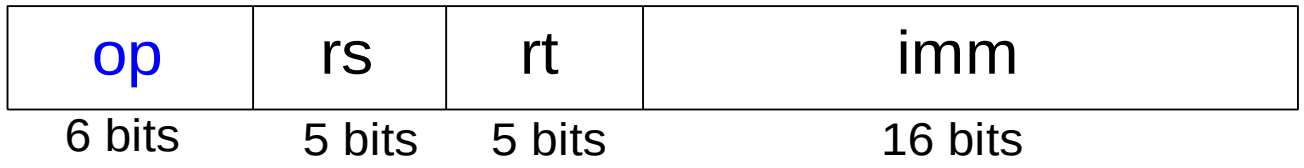
Datapath de Ciclo Único

addi

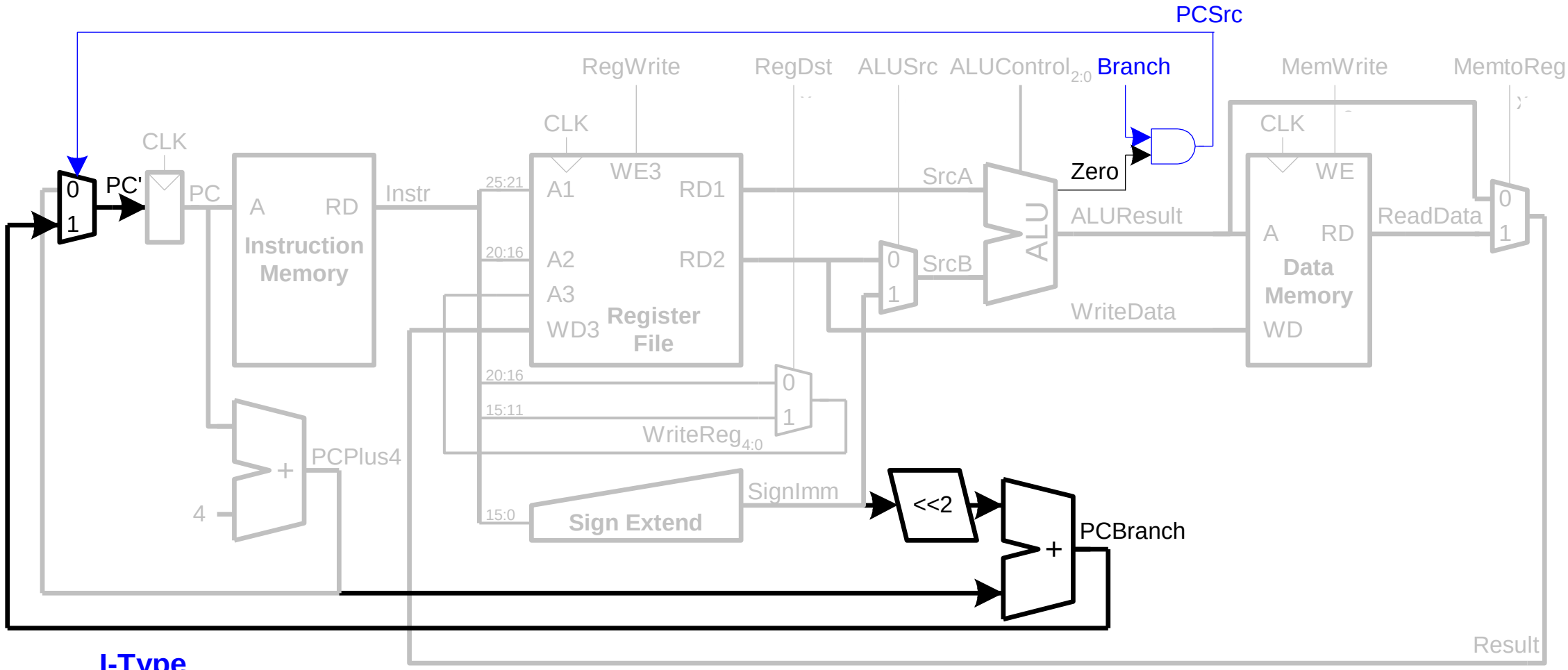
Instrução `addi`

- A instrução add imediato, `addi`, adiciona o conteúdo do registrador (`rs`) ao imediato estendido (sign-extended(`imm`)) e grava o resultado em outro registrador (`rt`).
- No slide a seguir, vamos analisar o datapath atual e verificar que ele já é capaz de realizar esta tarefa.

I-Type



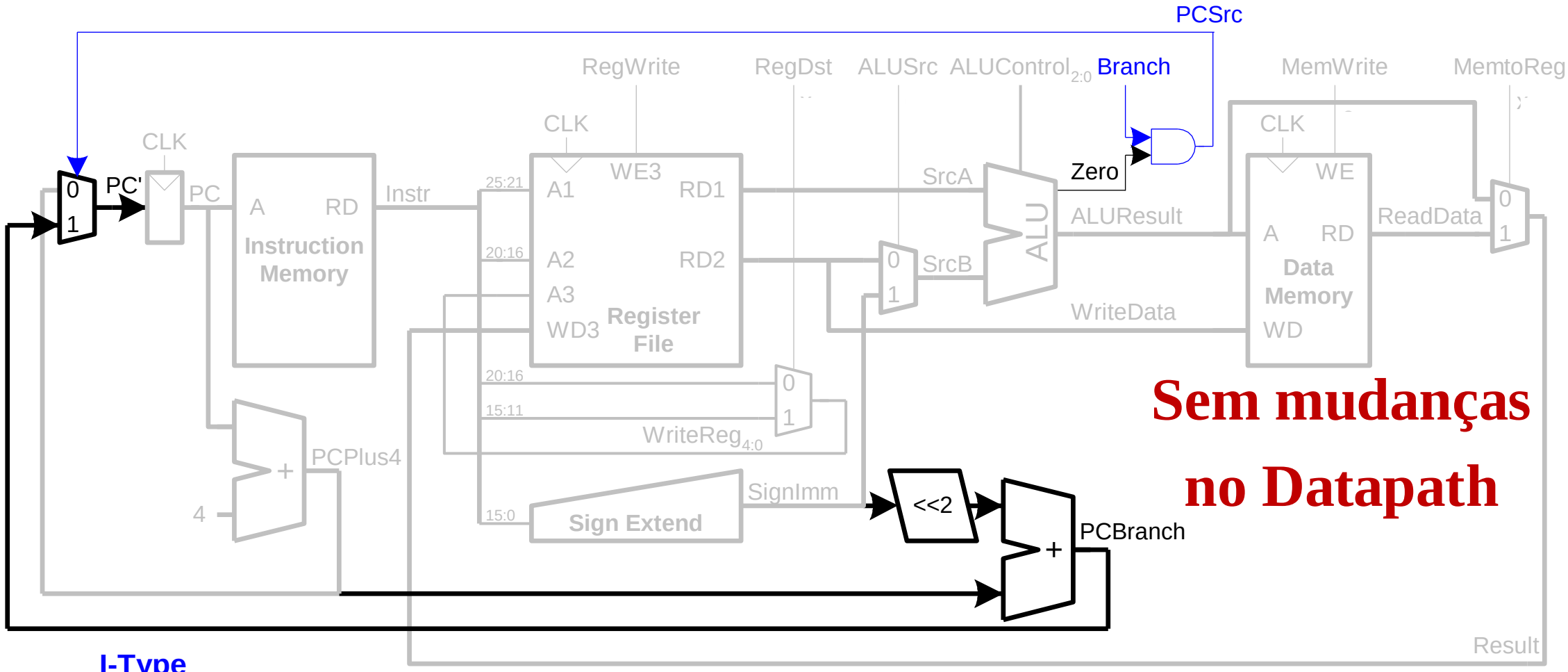
Instrução addi



I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Instrução addi



I-Type

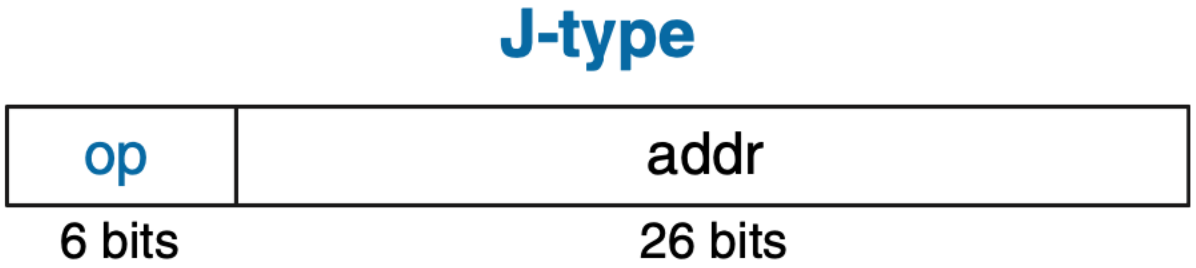
op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Datapath de Ciclo Único

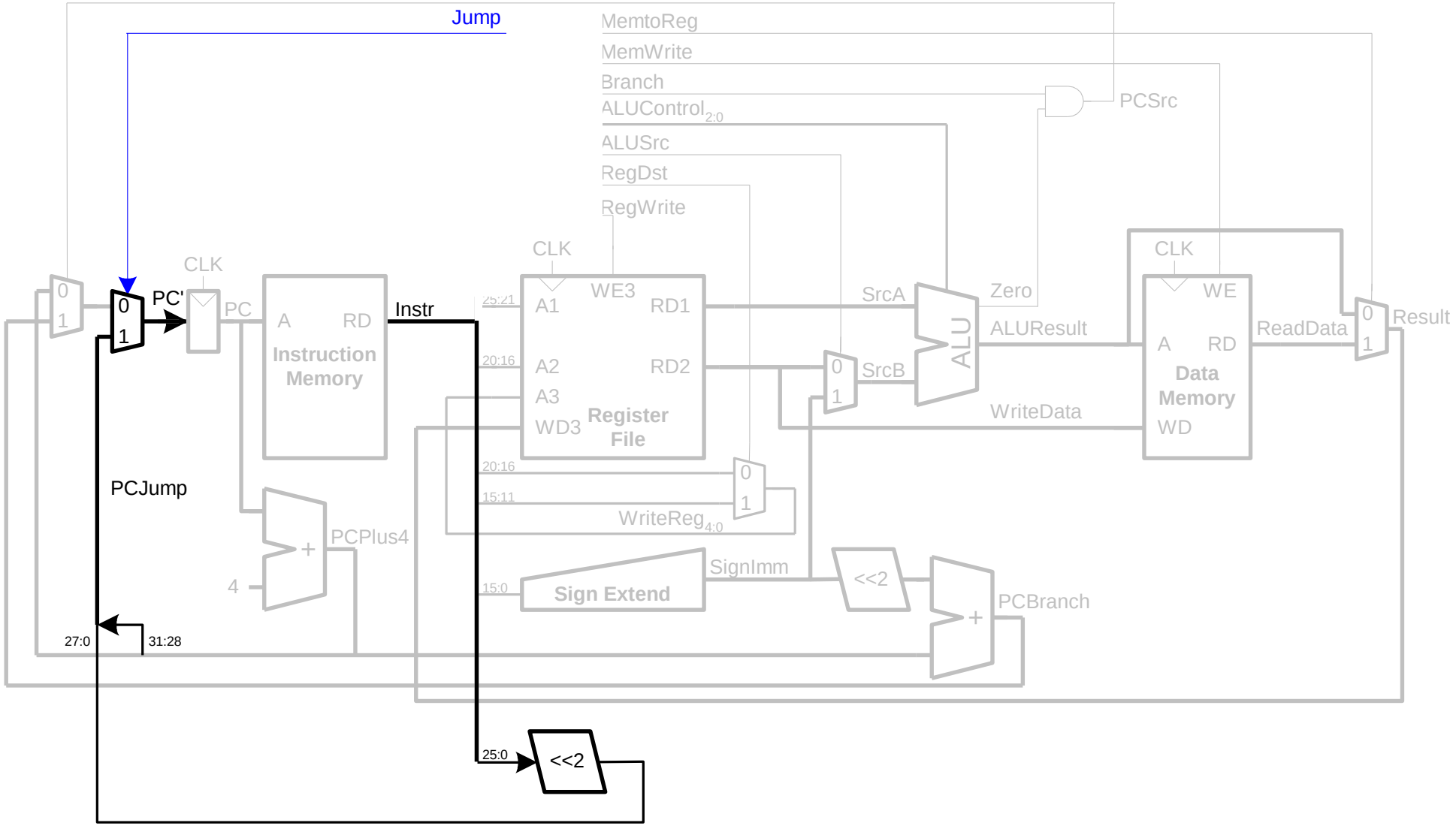
`j`

Instrução J

- A instrução de desvio incondicional, `j`, faz com que PC receba JTA (jump target address).
- JTA = $\{(PC+4)_{31:28}, addr<<2\}$

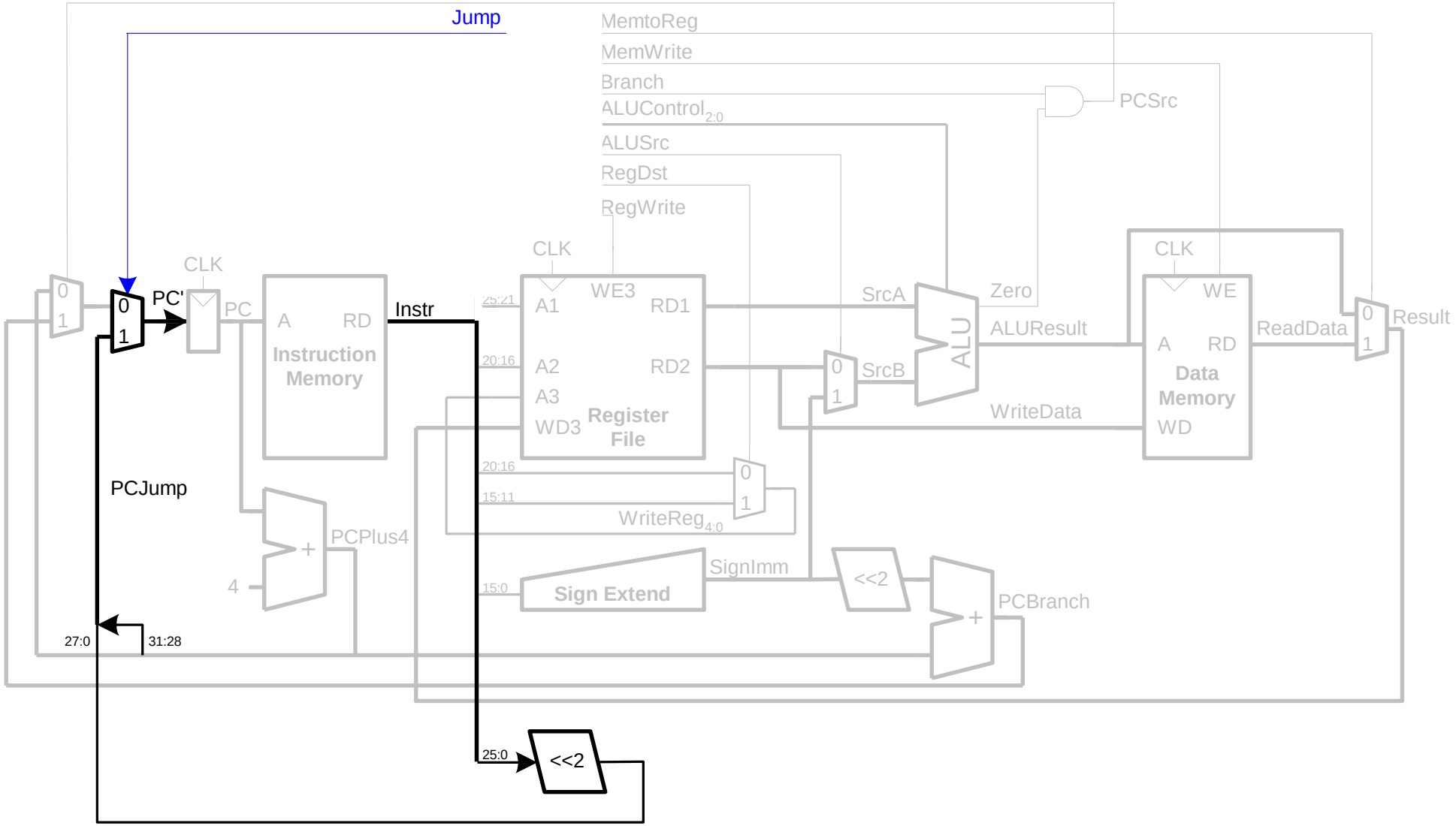


Instrução J


Jump
MemtoReg
MemWrite
Branch
ALUControl2:0
ALUSrc
RegDst
RegWrite
PCSrc
CLK
PC'
PC
Instr
A
RD
Instruction Memory
25:21
20:16
A1
WE3
RD1
A2
RD2
A3
WD3
Register File
SrcA
SrcB
ALU
Zero
ALUResult
WriteData
CLK
WE
A
RD
Data Memory
WD
ReadData
Result
PCJump
PCPlus4
4
+
20:16
15:11
WriteReg4:0
15:0
Sign Extend
SignImm
<<2
PCBranch
27:0
31:28
25:0
<<2

Datapath de Ciclo Único Completo

Datapath Completo



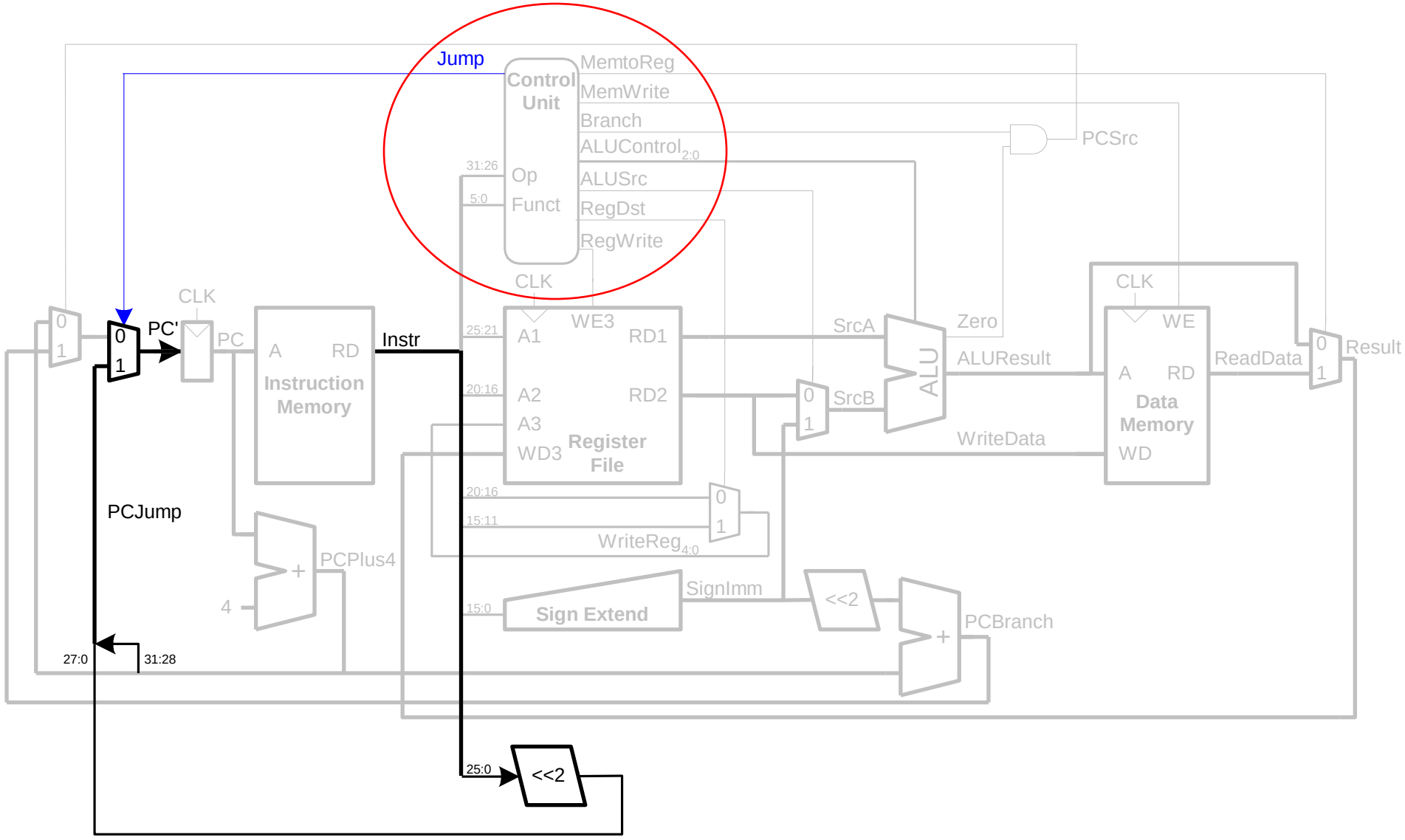
Trabalho 1

- Dois alunos por grupo.
- Implementar no logisim-evolution o datapth completo do processador de ciclo único.
- Observe que a maioria dos componentes já estão prontos no logisim-evolution. Portanto, você não precisará implementar todos.
- Prazo: duas semanas.

Unidade de Controle

Gera os sinais que o controlam o Datapath

Sinais Gerados pela Unidade de Controle

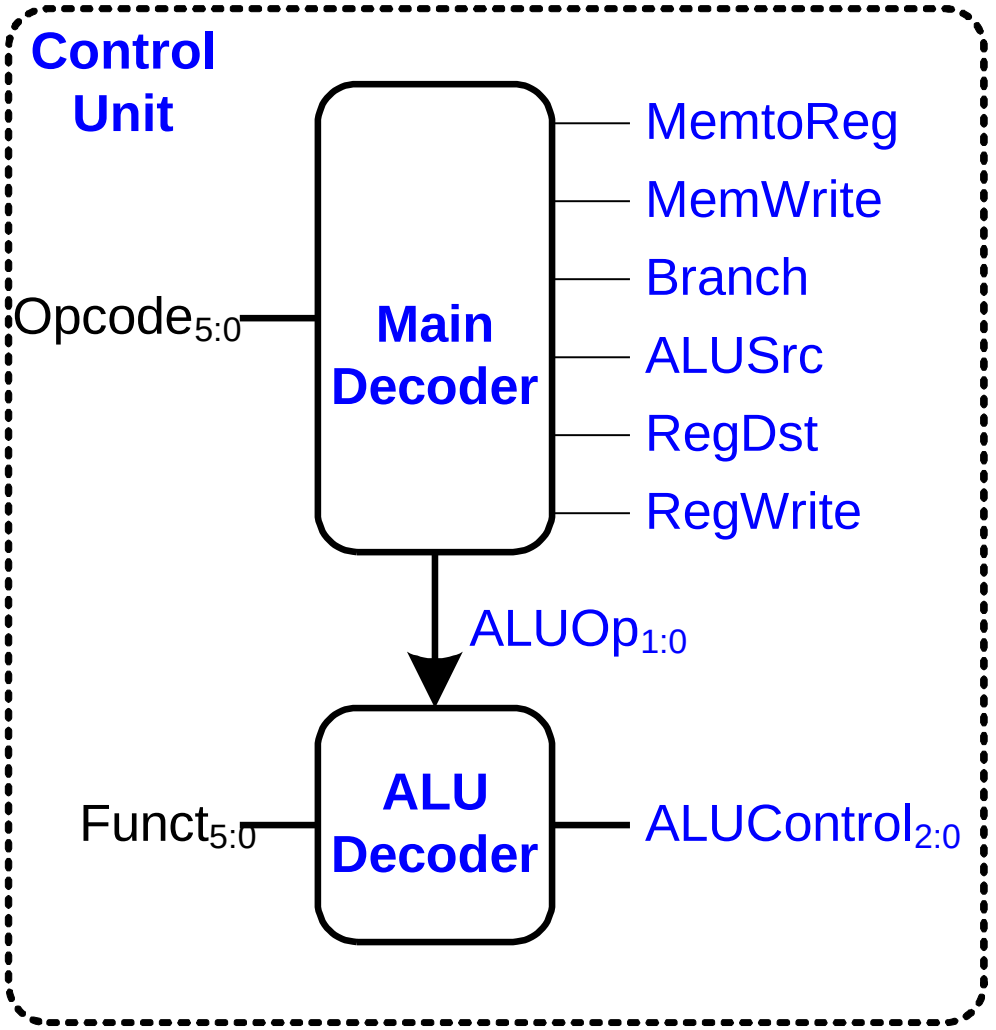


Projeto da Unidade de Controle

- Para facilitar o projeto, a unidade de controle foi dividida em duas partes: **Main Decoder** e **ALU Decoder**.

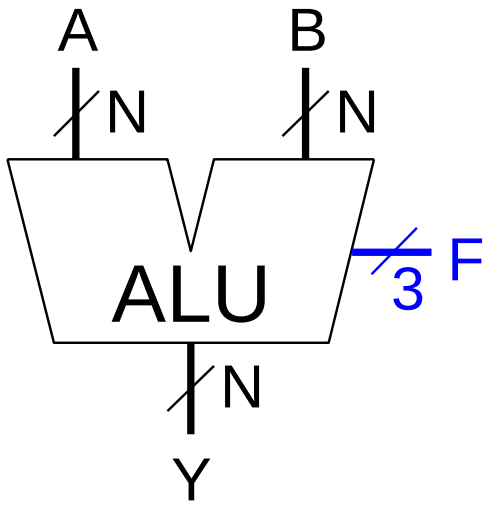
- **Main Decoder:** gera os sinais indicados com base no opcode das instruções.

- **ALU Decoder:** gera os sinais que controlam a ULA com base em ALUOp e Funct.

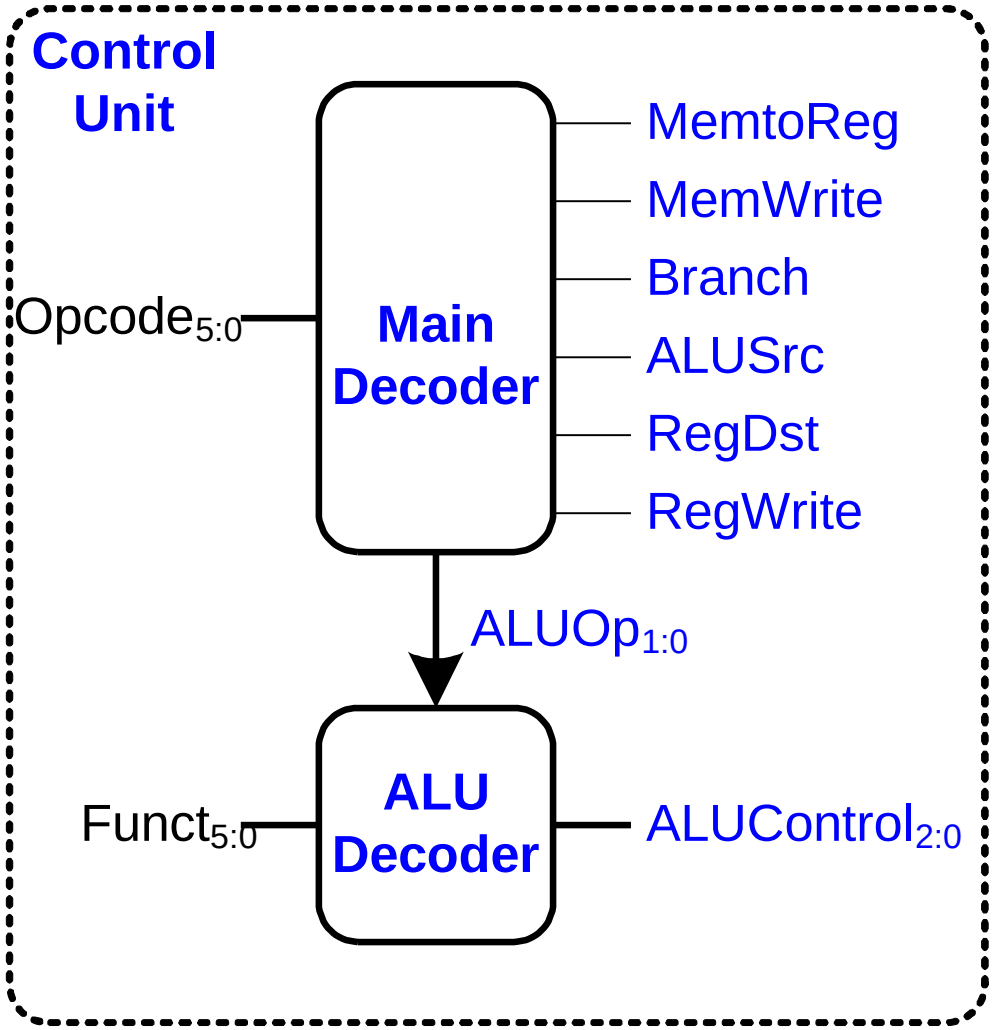


Codificação dos Sinais $ALUOp_{1:0}$

$ALUOp_{1:0}$	Significado
00	Adição (`lw, sw`)
01	Subtração (`beq`)
10	Olhar o campo Funct (Tipo-R)
11	Não Usado

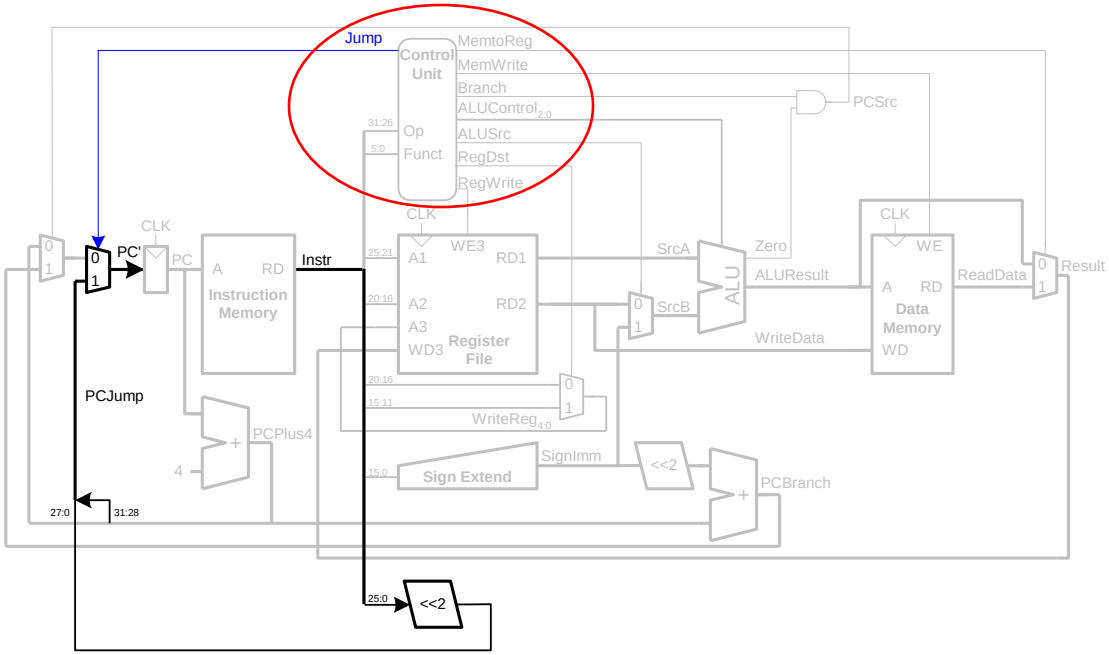


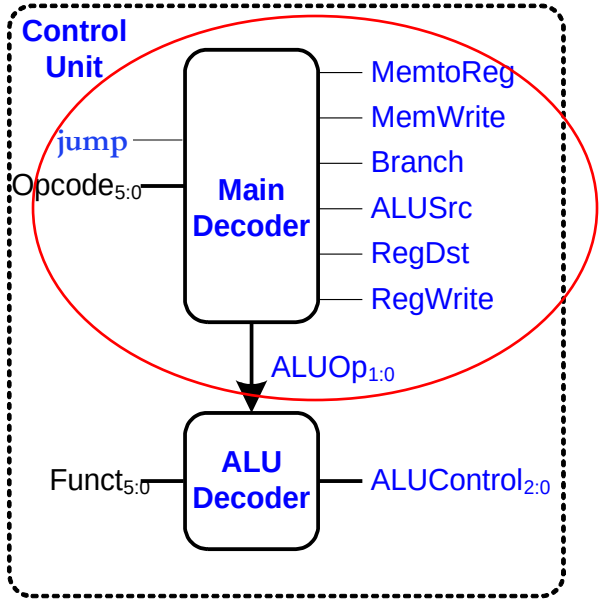
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT



Main Decoder

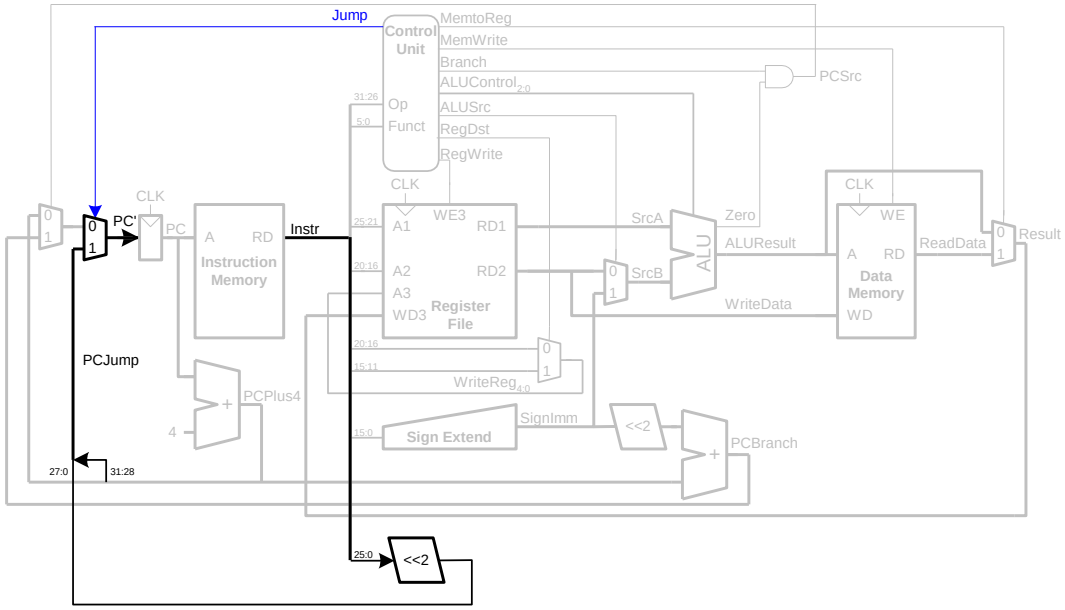
Instrução	$Opcode_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	jump	$ALUOp_{1:0}$
R-type	000000								
`lw`	100011								
`sw`	101011								
`beq`	000100								

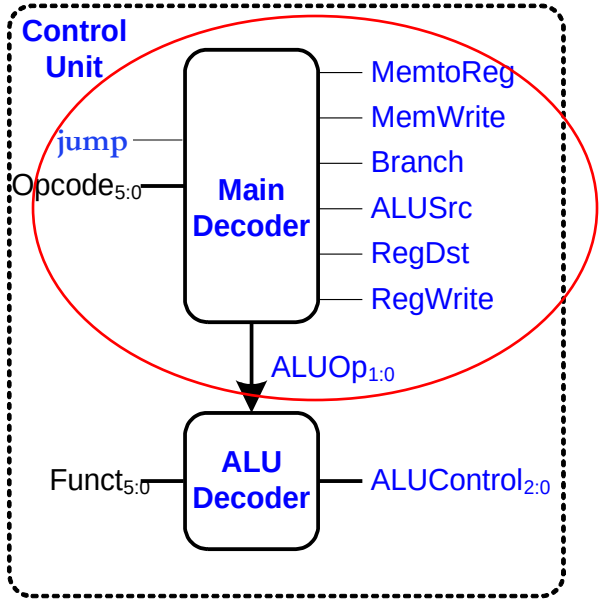




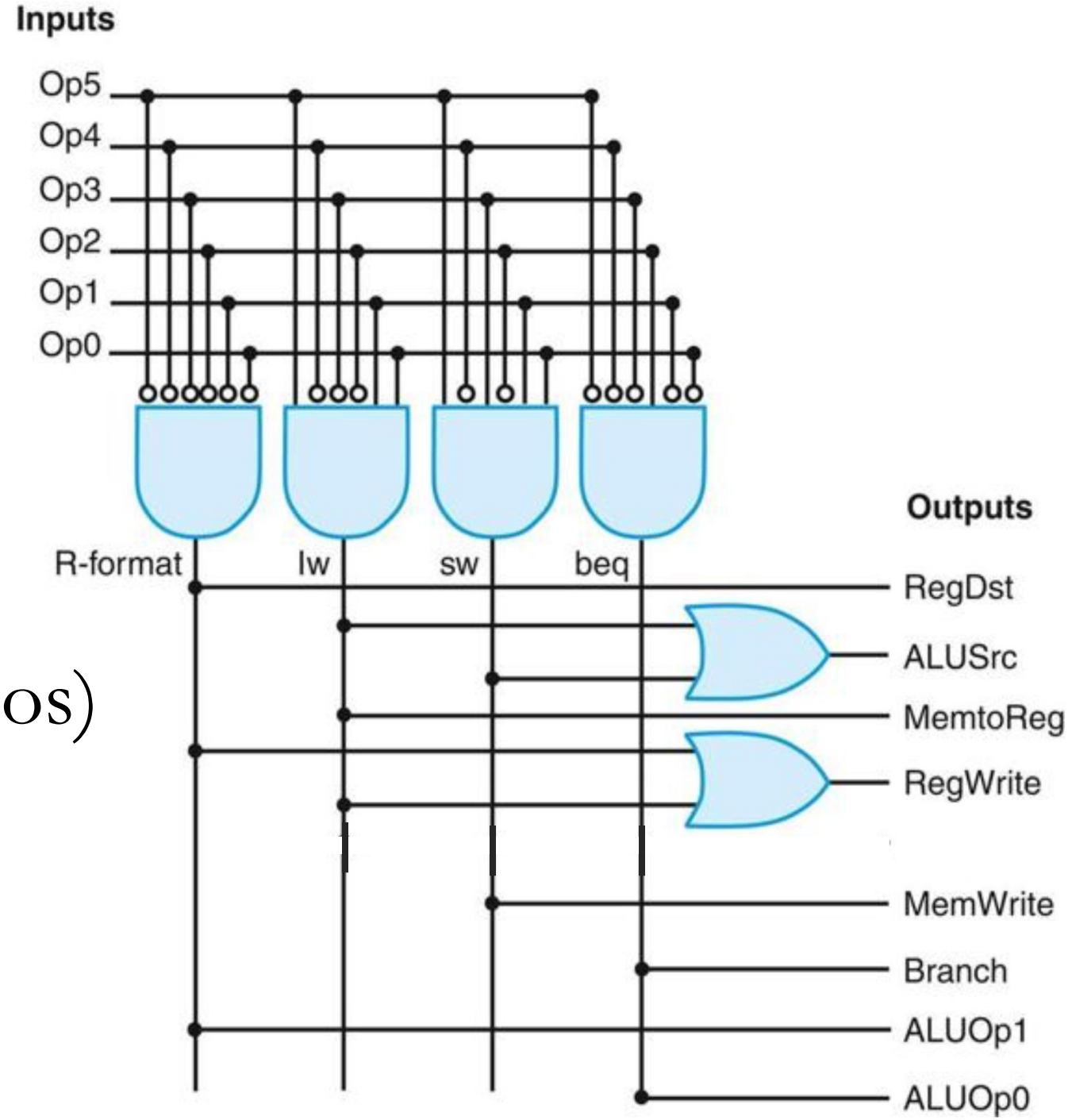
Main Decoder

Instrução	$Opcode_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	jump	$ALUOp_{1:0}$
R-type	000000	1	1	0	0	0	0	0	10
`lw`	100011	1	0	1	0	0	1	0	00
`sw`	101011	0	X	1	0	1	X	0	00
`beq`	000100	0	X	0	1	0	X	0	01
`addi`	001000	1	0	1	0	0	0	0	00
`j`	000010	0	X	X	X	0	X	1	XX



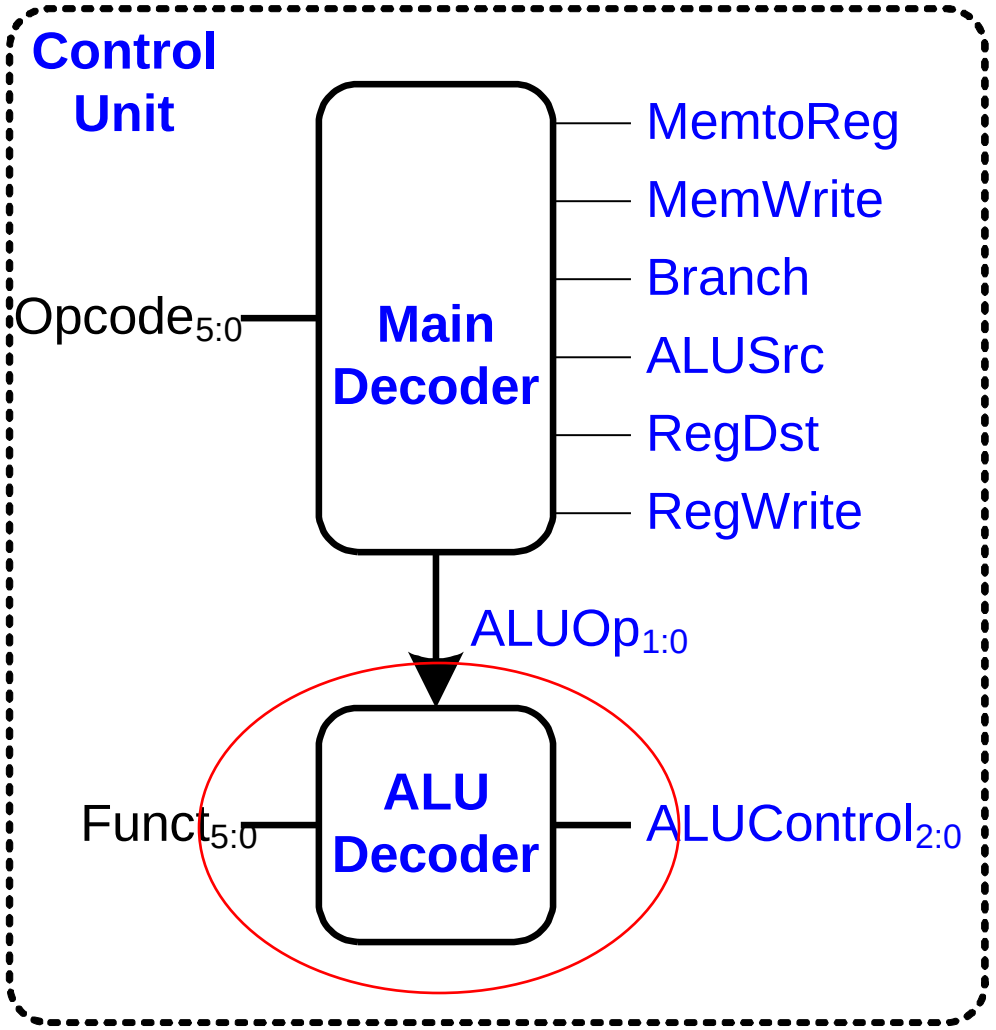


Main decoder
(`j` e `addi` não incluídos)



ALU Decoder

ALU Decoder		
$ALUOp_{1:0}$	$Funct_{5:0}$	$ALUControl_{2:0}$
00 (`lw`, `sw`, `addi`)	XXXXXX	
01 (`beq`)	XXXXXX	
10	100000 (`add`)	
10	100010 (`sub`)	
10	100100 (`and`)	
10	100101 (`or`)	
10	101010 (`slt`)	



ALU Decoder

ALU Decoder		
$ALUOp_{1:0}$	$Funct_{5:0}$	$ALUControl_{2:0}$
00 (`lw`, `sw`, `addi`)	XXXXXX	010
01 (`beq`)	XXXXXX	110
10	100000 (`add`)	010
10	100010 (`sub`)	110
10	100100 (`and`)	000
10	100101 (`or`)	001
10	101010 (`slt`)	111

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

ALU Decoder: $ALUControl_0$

$ALUOp_{1:0}$	Funct	$ALUControl_{2:0}$
00	XXXXXX	010 (Add)
X1	XXXXXX	110 (Subtract)
1X	XX0000 (`add`)	010 (Add)
1X	XX0010 (`sub`)	110 (Subtract)
1X	XX0100 (`and`)	000 (And)
1X	XX0101 (`or`)	001 (Or)
1X	XX1010 (`slt`)	111 (SLT)

$$ALUControl_0 = ALUOp1(F3 + F0)$$

ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
1	X	X	X	X	X	X	1
1	X	X	X	1	X	X	X

ALU Decoder: $ALUControl_1$

$ALUOp_{1:0}$	Funct	$ALUControl_{2:0}$
00	XXXXXX	010 (Add)
01	XXXXXX	110 (Subtract)
1X	XX0000 (`add`)	010 (Add)
1X	XX0010 (`sub`)	110 (Subtract)
1X	XX0100 (`and`)	000 (And)
1X	XX0101 (`or`)	001 (Or)
1X	XX1010 (`slt`)	111 (SLT)

$$ALUControl_1 = \overline{ALUOp1} + \overline{F2}$$

ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
0	X	X	X	X	X	X	X
X	X	X	X	X	0	X	X

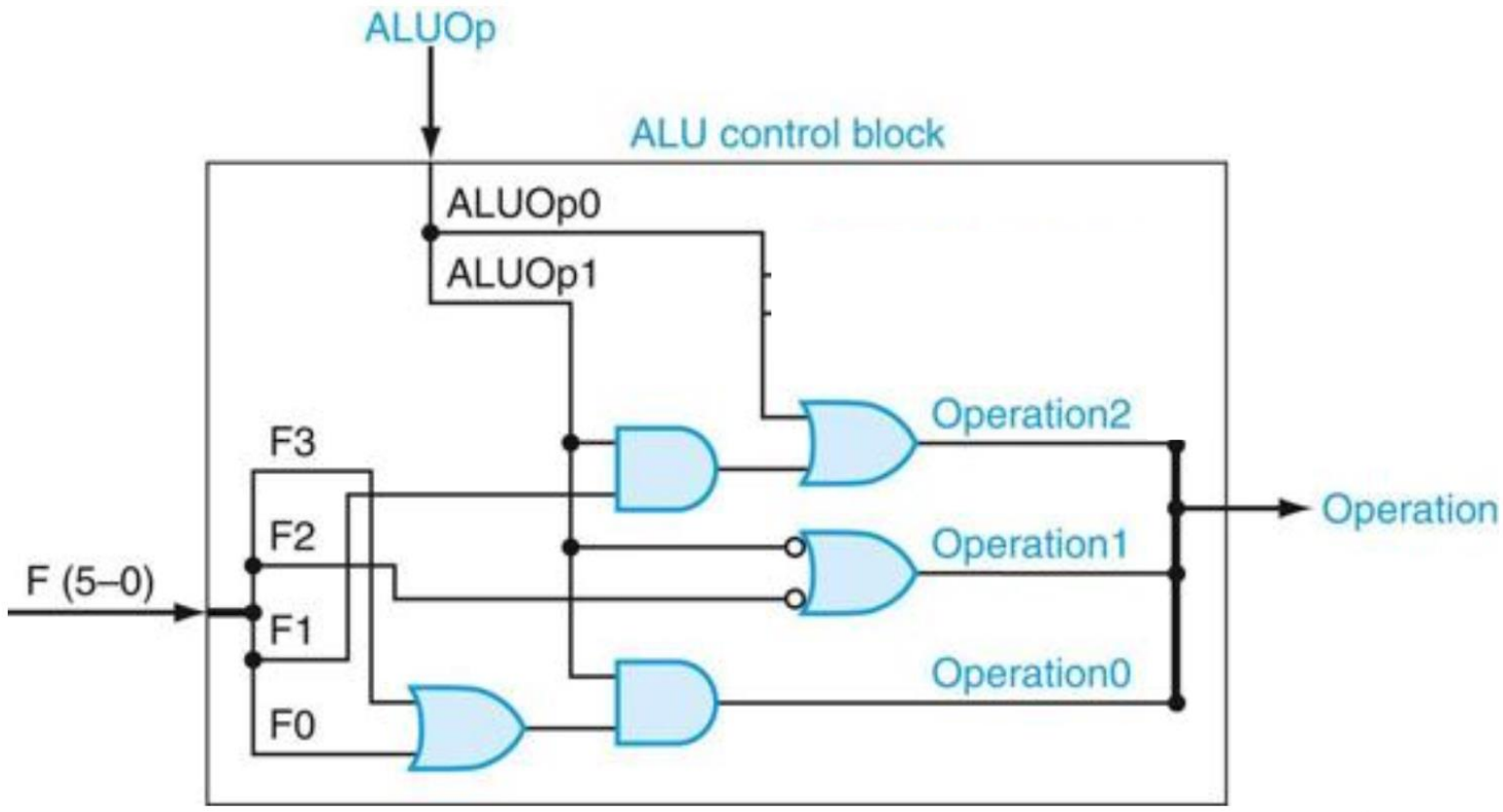
ALU Decoder: $ALUControl_2$

$ALUOp_{1:0}$	Funct	$ALUControl_{2:0}$
00	XXXXXX	010 (Add)
01	XXXXXX	110 (Subtract)
1X	XX0000 (`add`)	010 (Add)
1X	XX0010 (`sub`)	110 (Subtract)
1X	XX0100 (`and`)	000 (And)
1X	XX0101 (`or`)	001 (Or)
1X	XX1010 (`slt`)	111 (SLT)

$$ALUControl_2 = ALUOp_0 + ALUOp_1 F_1$$

ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
0	1	X	X	X	X	X	X
1	X	X	X	X	X	1	X

ALU decoder


ALUOp
ALU control block
ALUOp0
ALUOp1
F3
F2
F1
F0
F (5–0)
Operation2
Operation1
Operation0
Operation

Trabalho 2

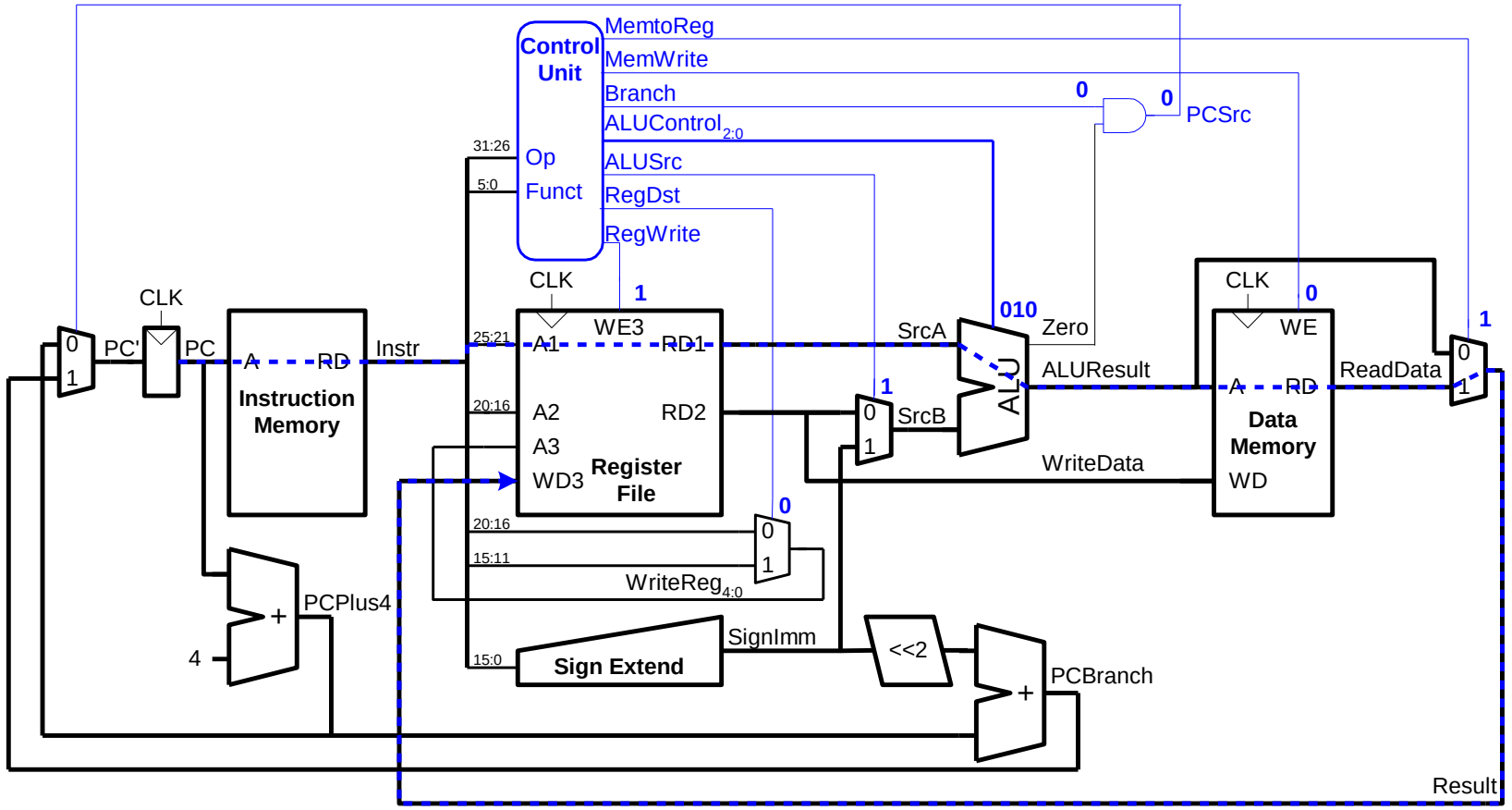
- Dois alunos por grupo.
- Implementar no logisim-evolution a unidade de controle do processador de ciclo único e interligar com o datapath.
- Executar um programa de teste simples.
- Prazo: três semanas.

Análise de Performance

Como medir a performance do processador?

- Definições:
  - CPI (Cyclos per Instruction): Ciclos/instrução.
  - Período de Clock: Tempo (em segundos)/ciclo.
  - IPC (Instruction per Cyclo): instruções/ciclo.
- Tempo de execução de programa = (#instructions) x CPI x TC
- O desafio é projetar um processador que satisfaça as restrições do projeto:
  - De Custo.
  - De Consumo de energia.
  - De Performance.

Performance do Processador de Ciclo Único



T_C limitado pelo caminho crítico (`lw`)

Performance do Processador de Ciclo Único

- Cada instrução toma um ciclo de clock para ser executada, então o CPI = 1.

- Caminho crítico de ciclo único:

$$T_c = t_{pcq\_PC} + t_{\text{mem}} + \max(t_{RF\text{read}}, t_{sext} + t_{\text{mux}}) + t_{\text{ALU}} + t_{\text{mem}} + t_{\text{mux}} + t_{RF\text{setup}}$$

- Tipicamente, as etapas limitantes são memória, ULA, Register File. Portanto, em geral, podemos simplificar o cálculo da seguinte forma:

$$T_c = t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{mux}} + t_{\text{ALU}} + t_{RF\text{setup}}$$

Exemplo

Elemento	Parâmetro	Atrado (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

$T_c = ?$

Exemplo

Elemento	Parâmetro	Atrado (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

$$T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$$
$$= [30 + 2(250) + 150 + 25 + 200 + 20] \text{ ps}$$
$$= 925 \text{ ps}$$

Exemplo

Com essas especificações, considere um program com 100 bilhões de instruções:

Tempo de execução = # instruções x CPI x T_C

= $(100 \times 10^9)(1)(925 \times 10^{-12}\ \text{s})$

= **92.5 seconds**