O Processador - Parte 1

Arquitetura de Computadores

Emilio Francesquini
e.francesquini@ufabc.edu.br

2020.Q1

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de **Arquitetura de Computadores** na UFABC.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- O conteúdo destes slides foi baseado no conteúdo do livro *Computer Organization And Design: The Hardware/Software Interface*, 5th Edition.

Lógica combinacional

Basicão de design lógico

UFABC

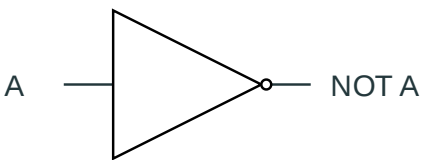
- Informação é codificada em binário
  - Tensão baixa = 0, tensão alta = 1
  - Um fio por bit
  - Dados multi-bit são transmitidos em barramentos multi-fios
- Elemento combinacional
  - Opera nos dados
  - Saída depende da entrada
- Elementos de estado (sequenciais)
  - Armazenam informação

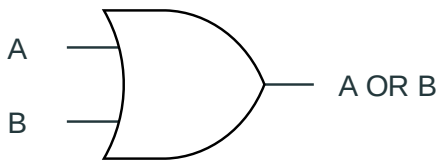
A lógica combinacional

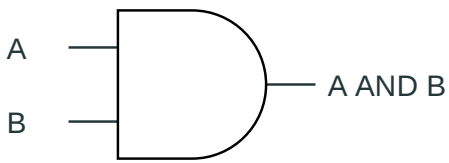
UFABC

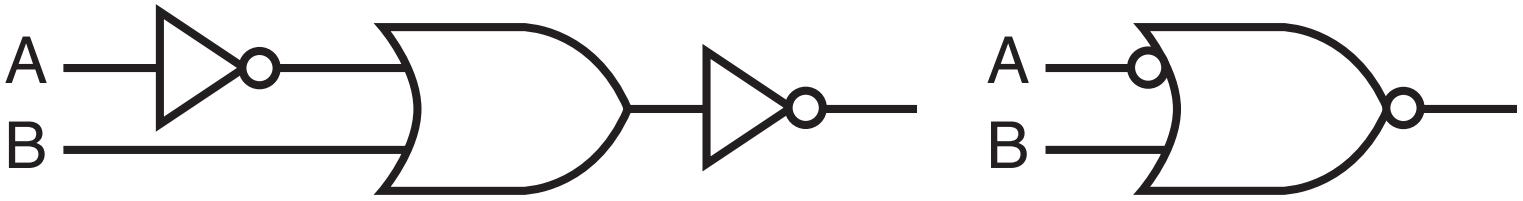
- Nesta aula revisamos brevemente alguns conceitos fundamentais que utilizaremos no restante das aulas.
- Esta disciplina não trata destes assuntos diretamente (outras disciplinas como Circuitos Digitais e Sistemas Digitais servem este propósito) mas utilizaremos alguns dos conceitos que elas apresentam.
- Informações mais detalhadas podem ser vistas em [PH]: Appendix B.

Portas lógicas







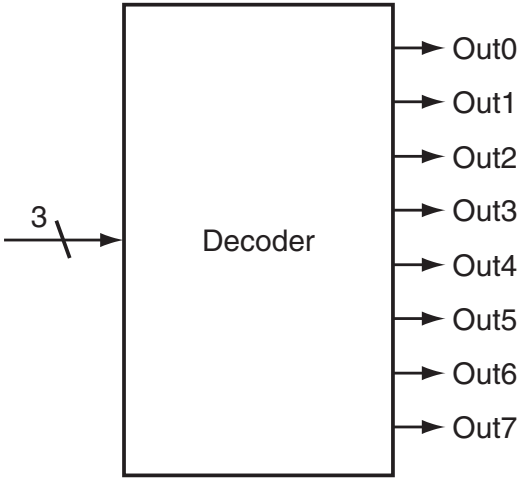


Pergunta

Que expressão lógica os circuitos acima representam?

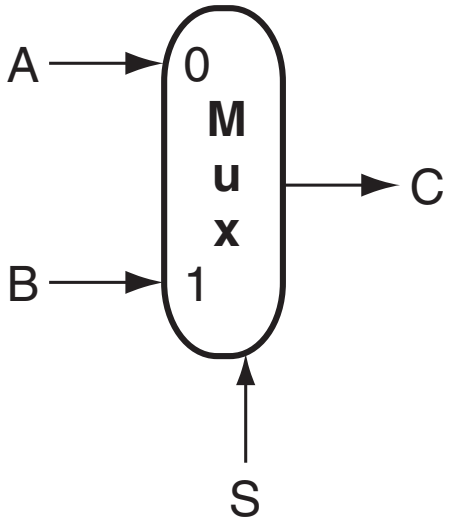
UFABC

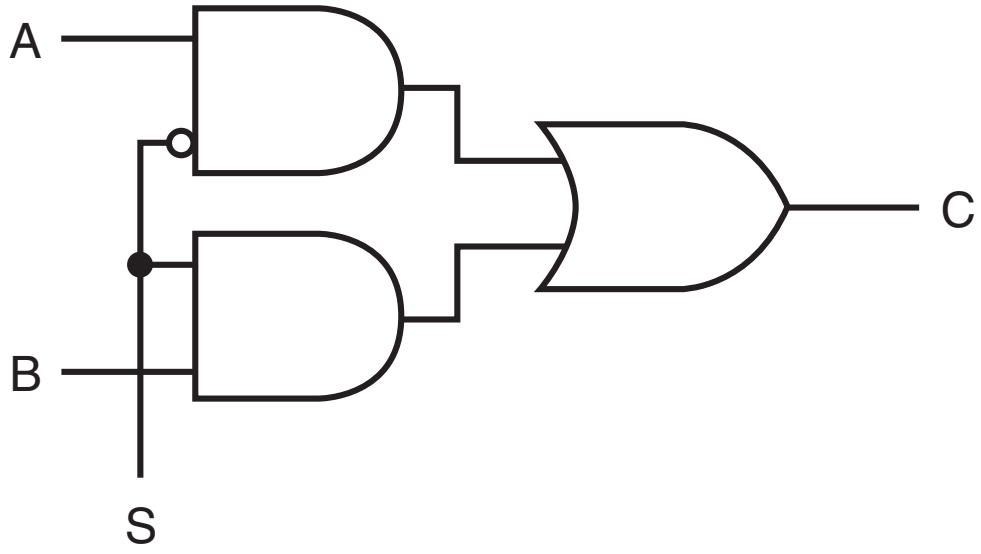
- Decoders escolhem uma dentre 2^n saídas baseando-se em uma entrada de n bits.
- Exitem também os encoders que fazem o caminho inverso.



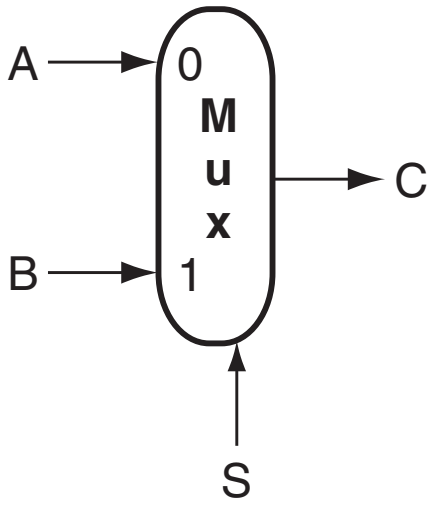
Inputs			Outputs							
12	11	10	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

- Permitem escolher uma entrada baseando-se num seletor





Multiplexadores de 1 bit

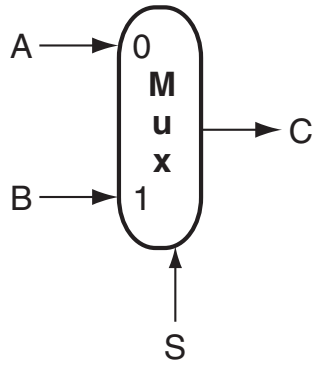


- Como montar um?
- Usamos uma tabela verdade!

A	B	S	C
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

- Como transformar em um circuito?

Multiplexadores de 1 bit

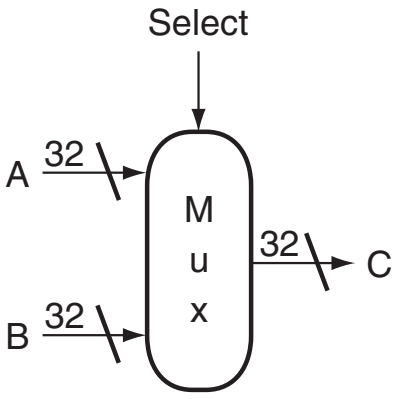


A	B	S	C
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

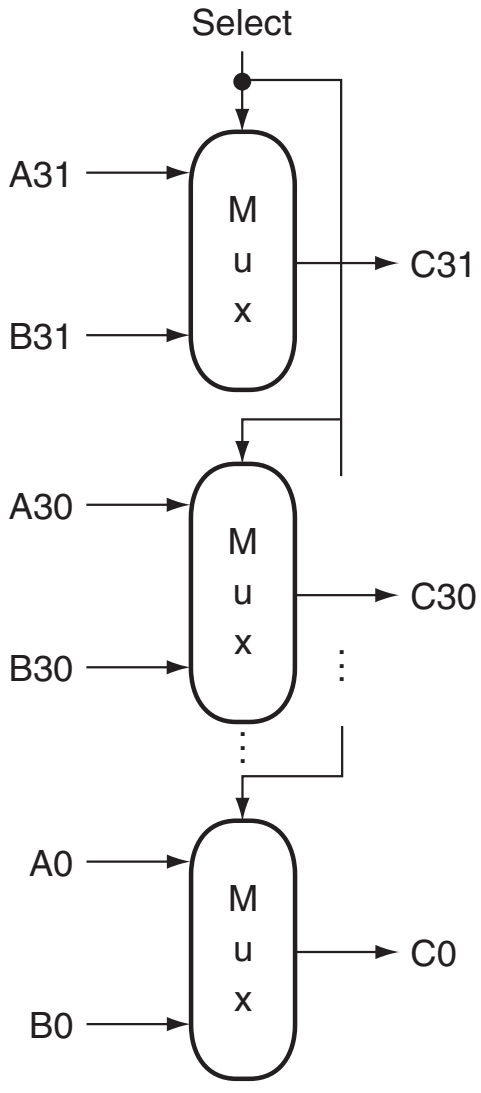
A tabela ao lado é equivalente a:

$$C = \overline{A}.B.S + A.\overline{B}.\overline{S} + A.B.\overline{S} + A.B.S =$$
$$\overline{A}.B.S + A.(\overline{B}.\overline{S} + B.\overline{S} + B.S) =$$
$$\overline{A}.B.S + A.(\overline{S}(\overline{B} + B) + B.S) =$$
$$\overline{A}.B.S + A.(\overline{S} + B.S) =$$
$$\overline{A}.B.S + A.\overline{S} + A.B.S =$$
$$A.\overline{S} + B.(\overline{A}.S + A.S) =$$
$$A.\overline{S} + B.(S.(\overline{A} + A)) =$$
$$A.\overline{S} + B.S$$

Multiplexadores de 32 bits



a. A 32-bit wide 2-to-1 multiplexor



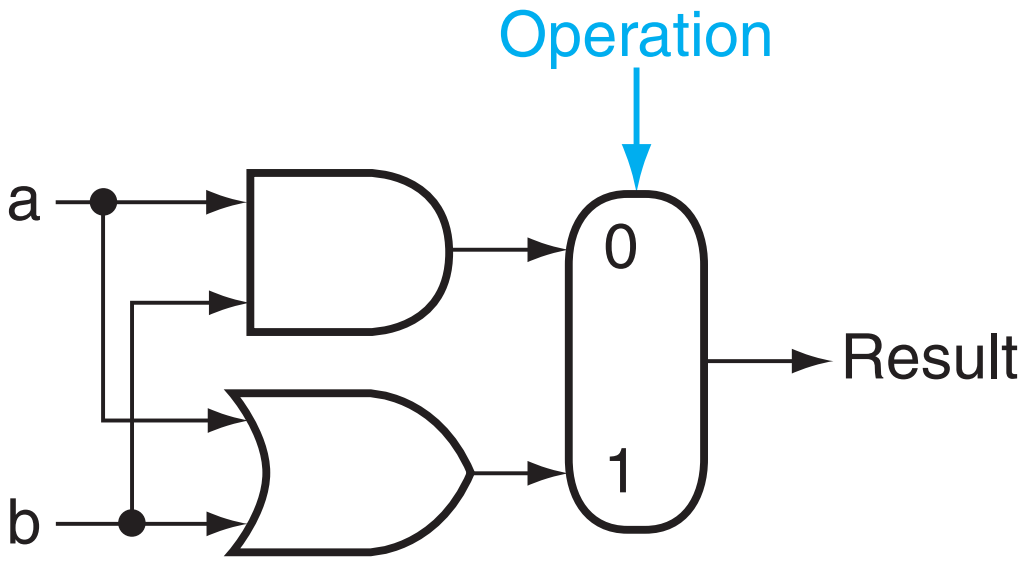
b. The 32-bit wide multiplexor is actually an array of 32 1-bit multiplexors

Construindo uma ULA básica de 1 bit

UFABC

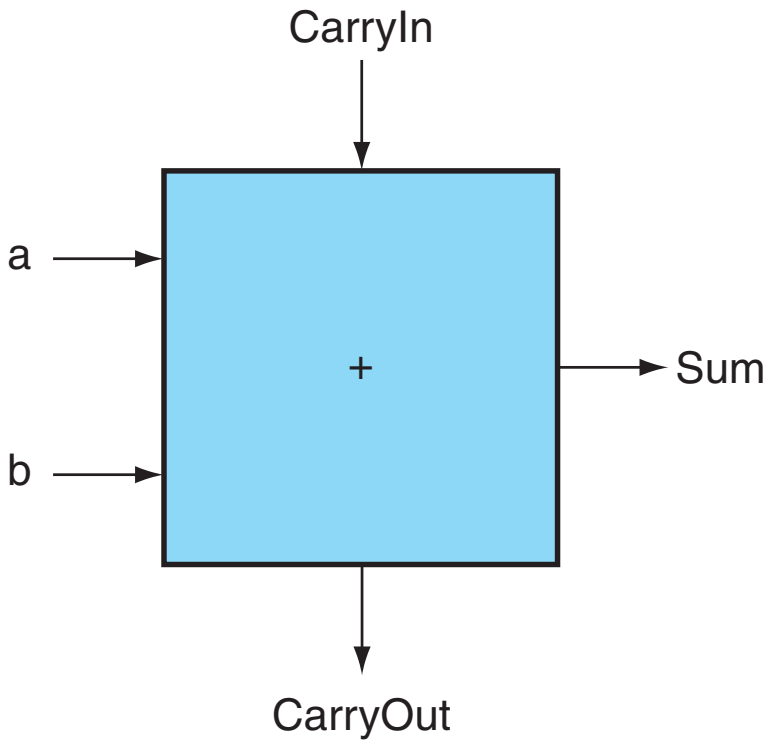
- Se queremos chegar até um MIPS, temos várias instruções que precisam ser implementadas
- Vamos começar com as mais básicas
  - AND, OR, ADD, SUB, SLT, BEQ, ...
- Para as duas primeiras já temos as portas lógicas prontas! Vamos começar por elas.

ULA de 1 bit para **AND** e **OR**



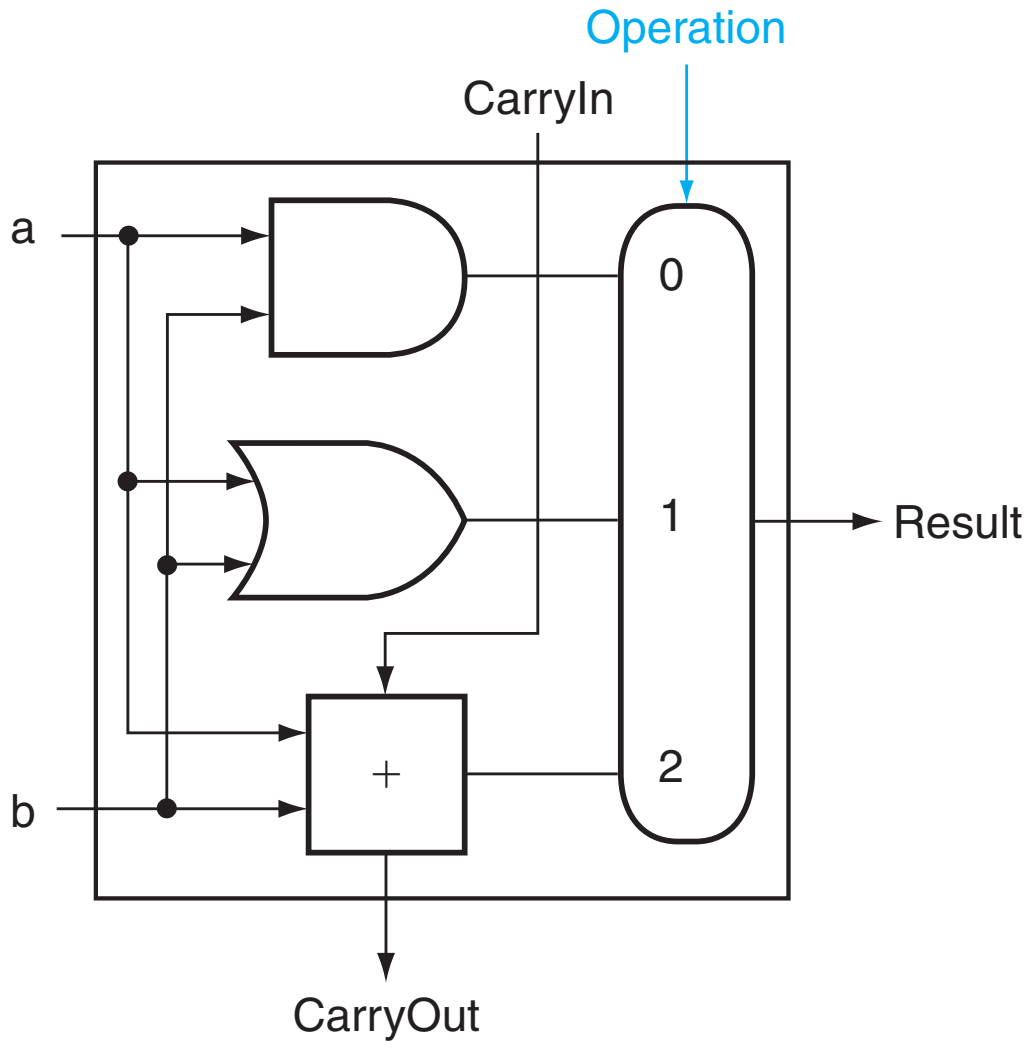
- Vamos adicionar **ADD**...

Um somador de 1 bit



- Um somador de 1 bit pode ser facilmente implementado usando o método que usamos para fazer um multiplexador.
- Vamos integra-lo na nossa ULA.

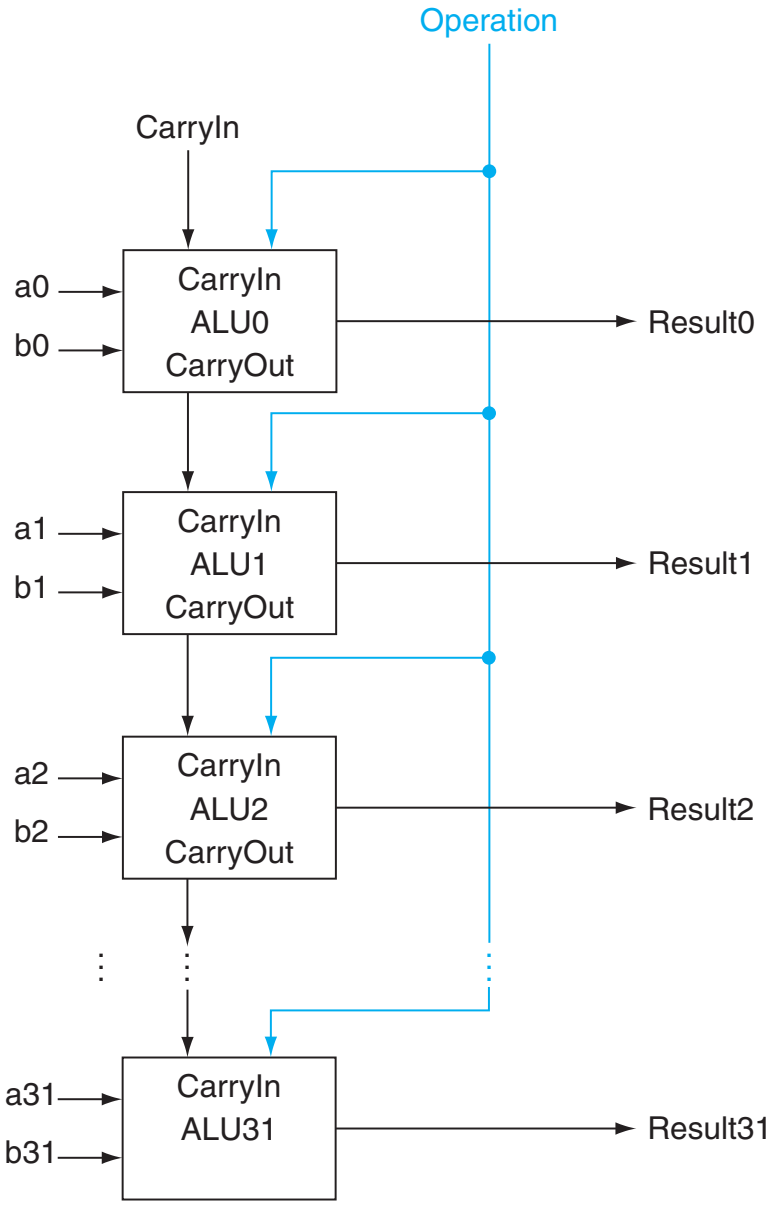
ULA de 1 bit com **AND**, **OR** e **ADD**



Pergunta

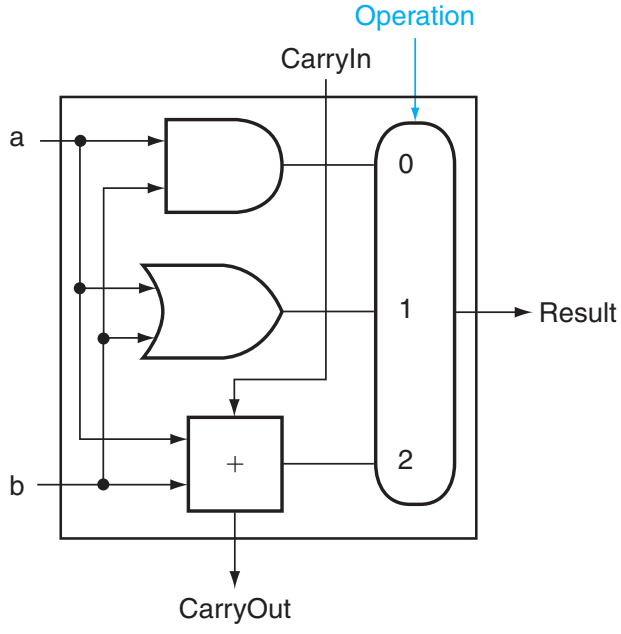
Como fazemos para criar uma ULA de 32 bits?

ULA de 32 bits com **AND**, **OR** e **ADD**



Incluindo a operação **SUB**

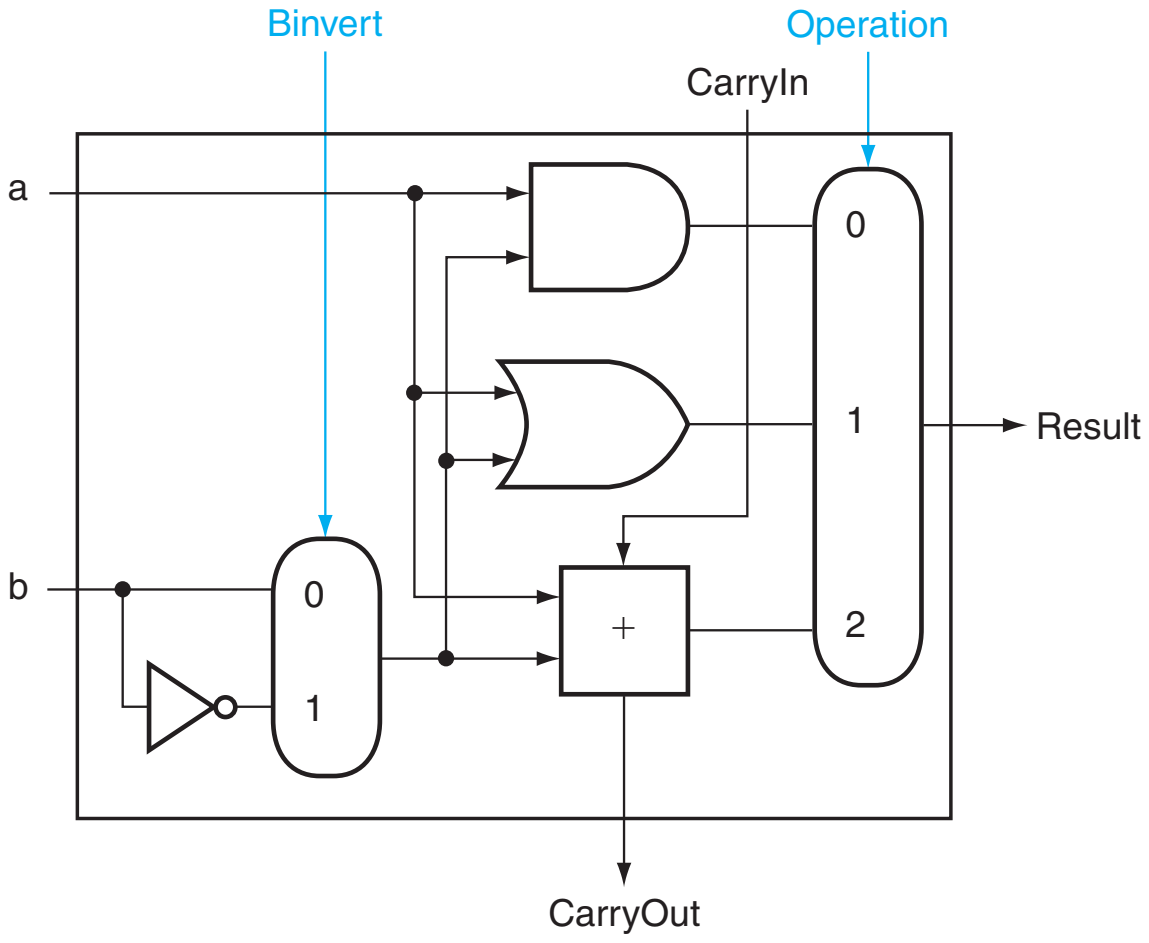
- Nas aulas anteriores vimos que para números em complemento de 2, a subtração é muito parecida com a adição.
- Para calcular $A - B$, invertemos todos os bits de B, somamos 1 ao resultado e em seguida somamos com A.



Note

Conseguimos adaptar o que já temos pronto para fazer isso?

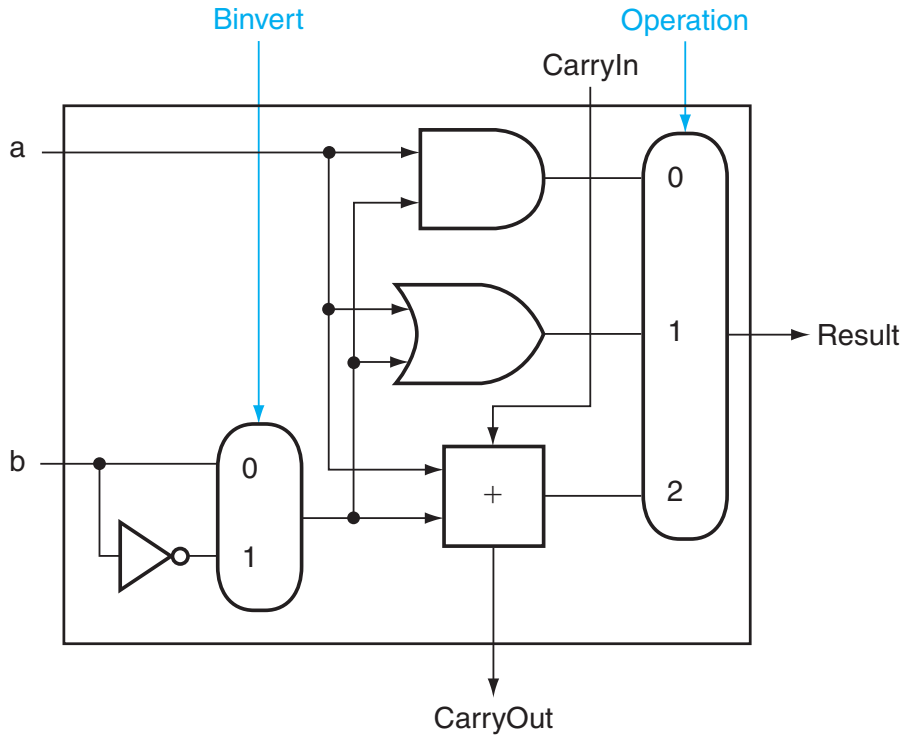
Incluindo a operação **SUB**



- Basta então fornecer os sinais de controle corretamente.

Incluindo a operação NOR

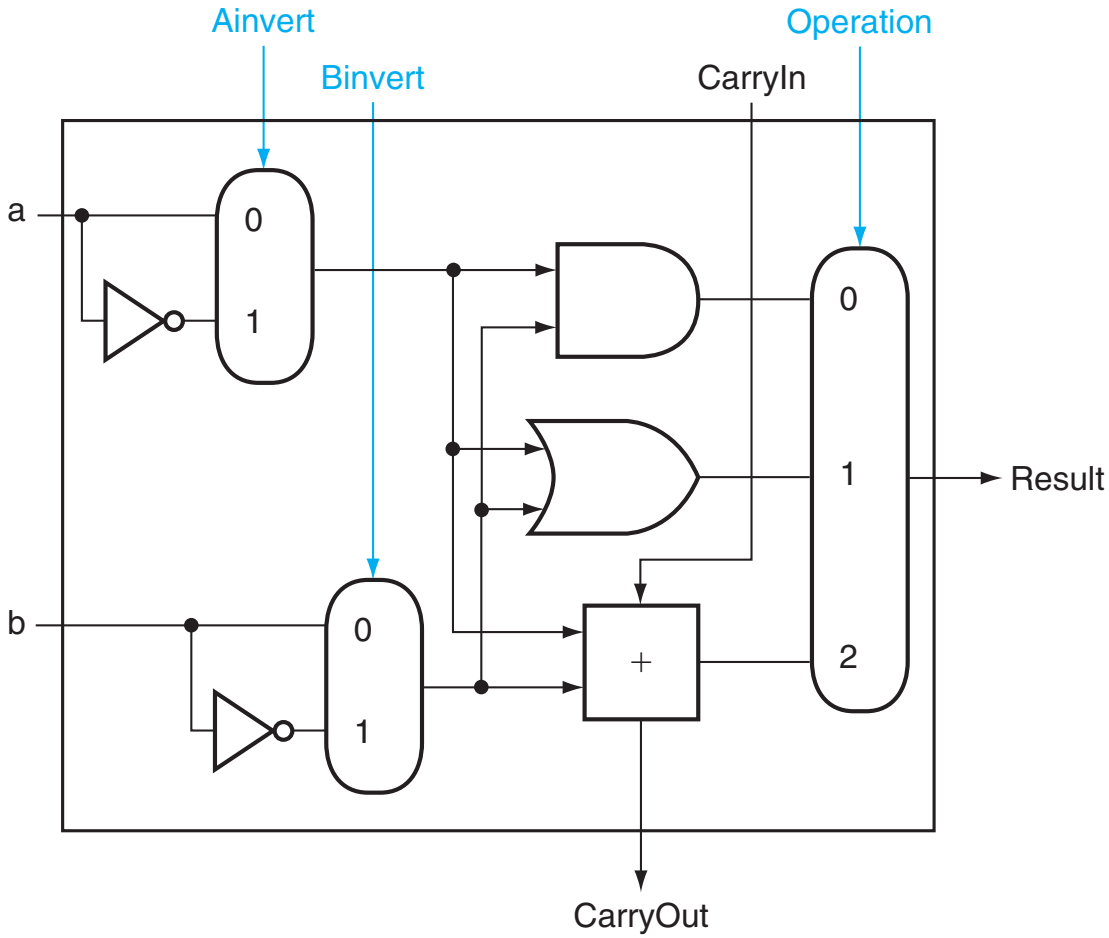
- De maneira semelhante, com pouquíssimas modificações já conseguimos adicionar a operação NOR
  - `NOR A B = NOT(A OR B) = (NOT A) AND (NOT B)`



Pergunta

Como alteramos o circuito acima para fazer isso?

Incluindo a operação **NOR**



Pergunta

Quais são os sinais de controle para cada uma das operações que já implementamos (`AND OR ADD SUB NOR`)?

Sinais de controle

UFABC

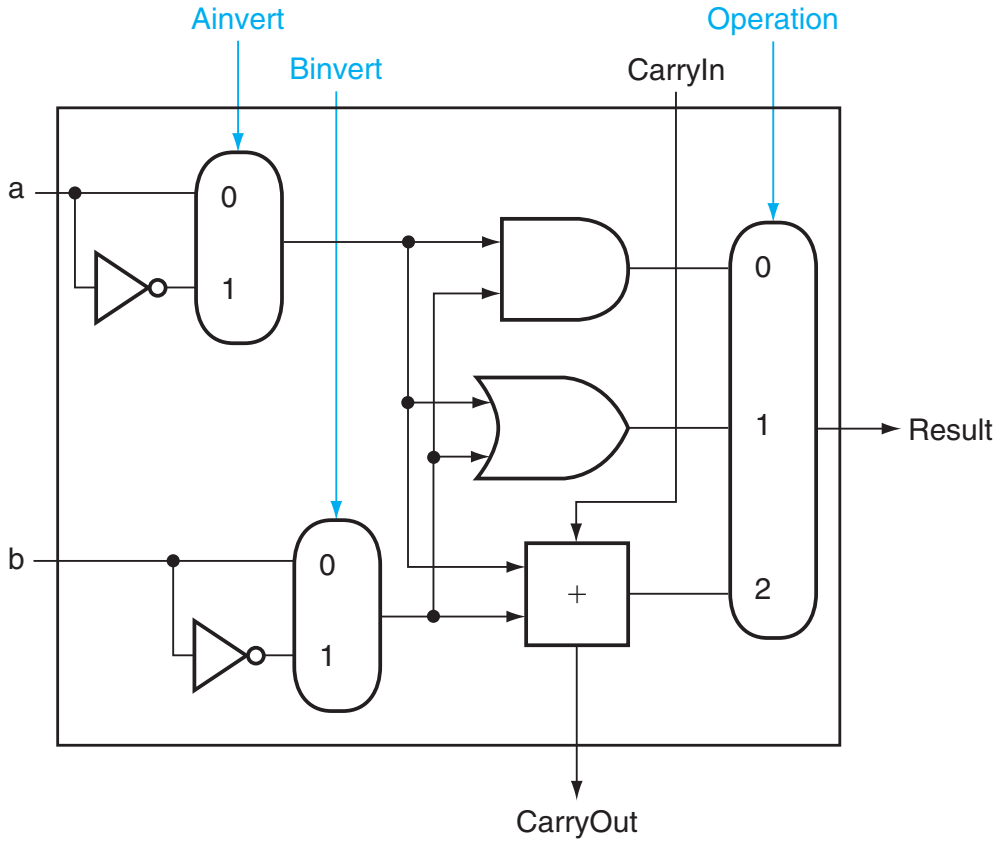
	Operation	Carry in	Ainvert	Binvert
AND	00	_	0	0
OR	01	_	0	0
NOR	01	_	1	1
ADD	10	0	0	0
SUB	10	1	0	1

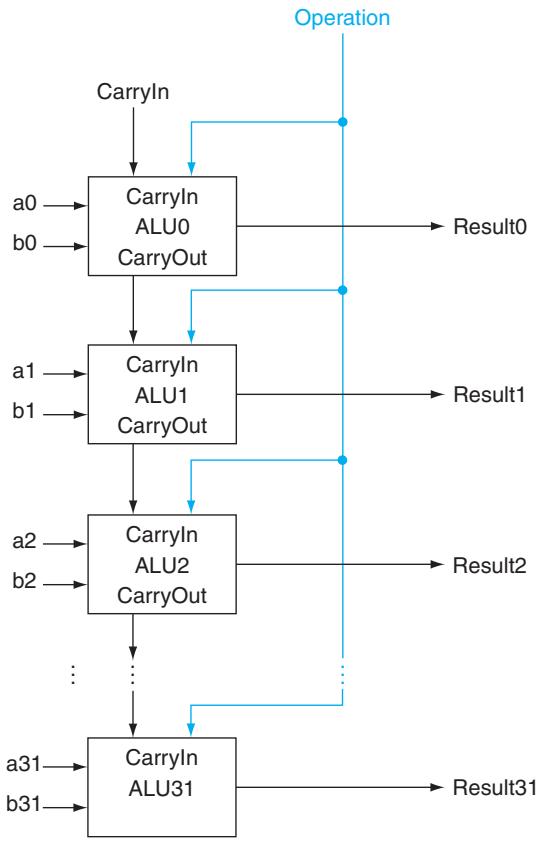
Tip

Conseguimos implementar diretamente mais uma operação só com o hardware que temos? Se sim, qual?

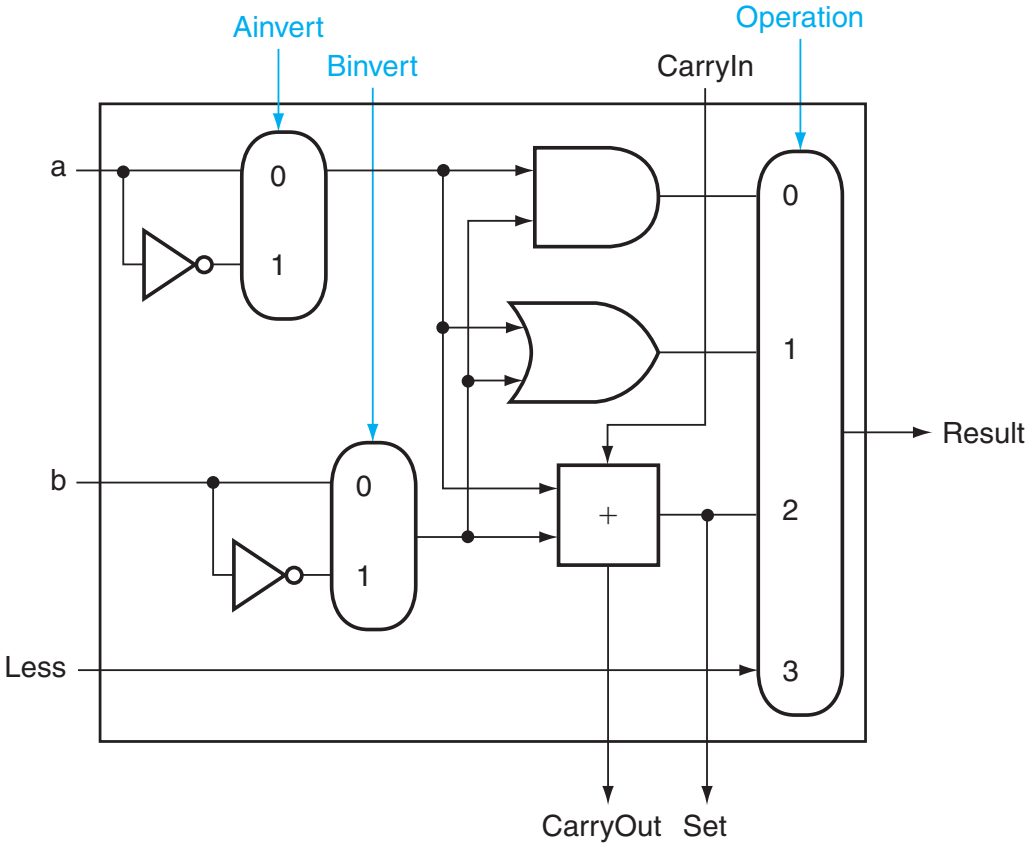
Mas ainda está faltando algo...

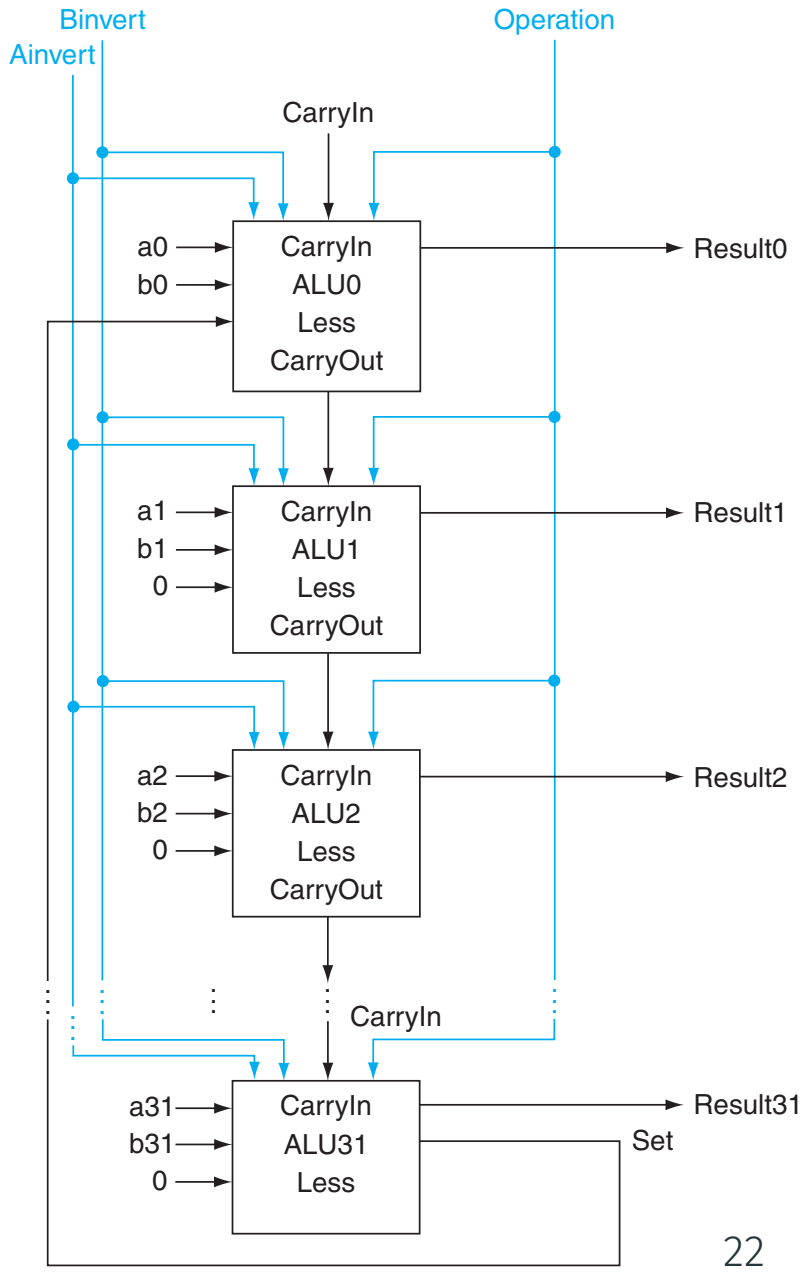
- Para criarmos o nosso processador MIPS, ainda temos 2 operações que são essenciais: `slt` e `beq`
- A ALU tem que ser capaz de verificar se `A < B` e se `A = B`
- O que dá para fazer com que já temos?





Implementando o `slt`





UFABC

- Para saber se $A = B$ podemos fazer $A - B$. Se for 0, então $A = B$. Basta negar o resultado!
- Vamos consertar o circuito! Mas antes vamos olhar novamente a tabela de sinais de controle

	Operation	Carry in	Ainvert	Binvert
AND	00	_	0	0
OR	01	_	0	0
NOR	01	_	1	1
ADD	10	0	0	0
SUB	10	1	0	1
SLT	11	1	0	1

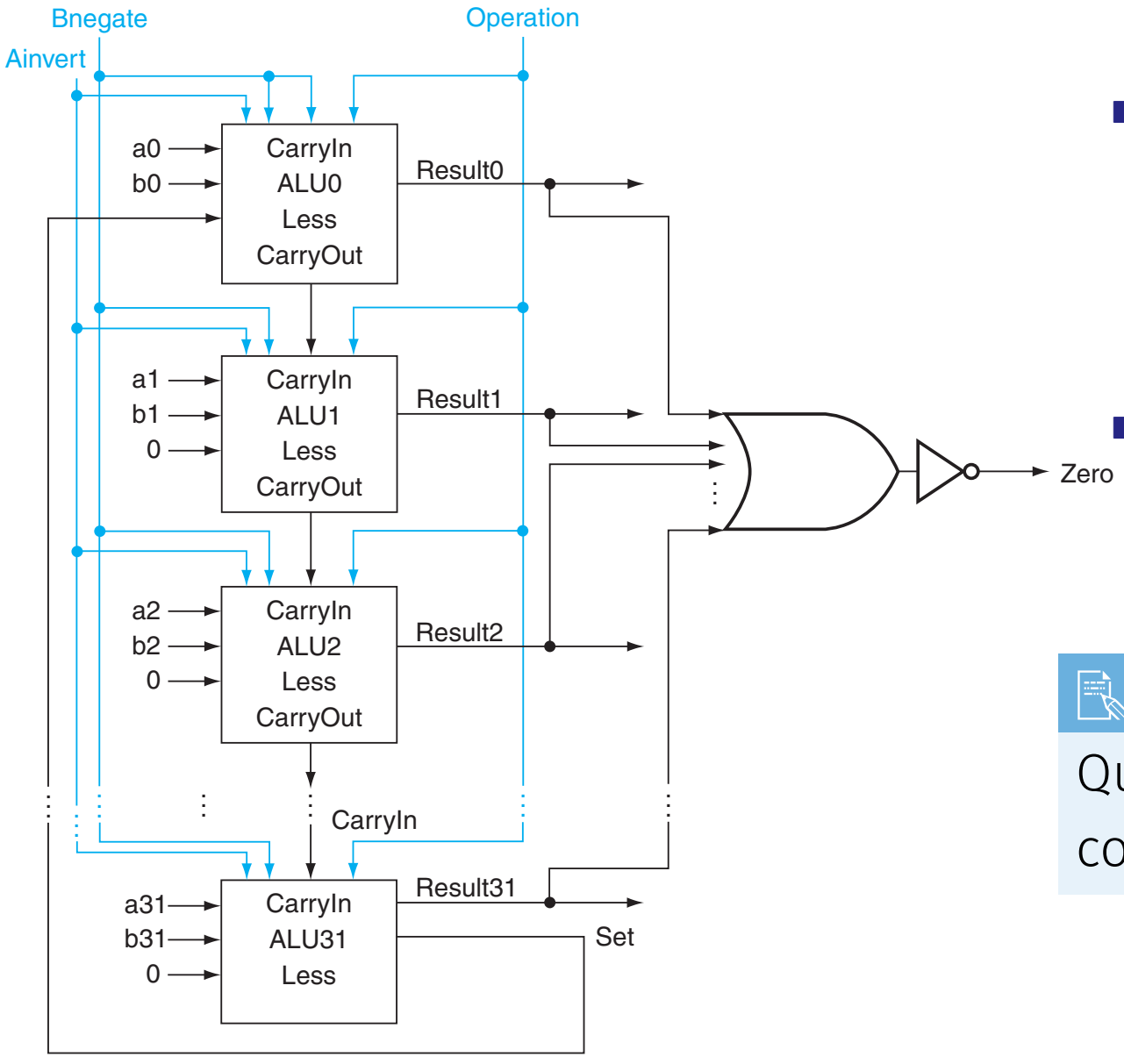
UFABC

- Podemos juntar o *Carry in* e *Binvert*. Vamos chamar de Bnegate

	Operation	Bnegate	Ainvert
AND	00	0	0
OR	01	0	0
NOR	01	1	1
ADD	10	0	0
SUB	10	1	0
SLT	11	1	0

- Voltando ao **beq**...

beq

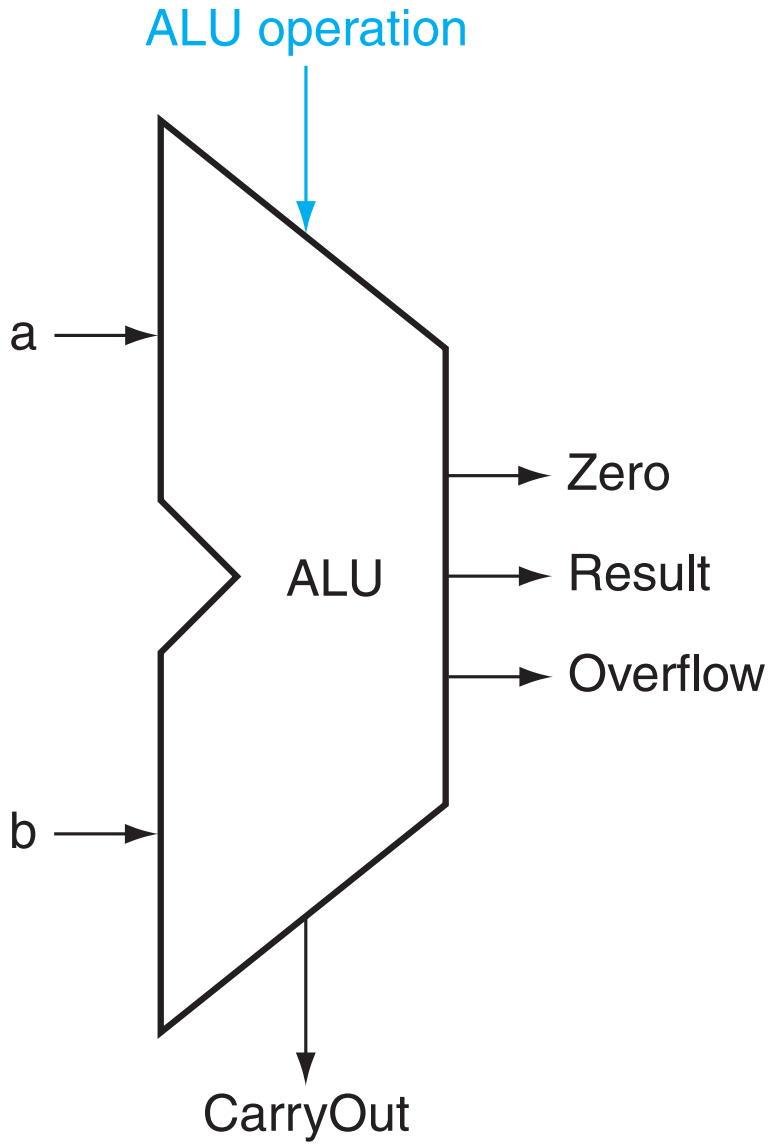


- Criamos uma nova saída da ALU, chamada **`Zero`**
- Quando **`Zero`** for 1, *A* e *B* são iguais.

Note

Quais são os bits de controle para **`beq`**?

Finalmente temos uma ALU



O Processador

UFABC

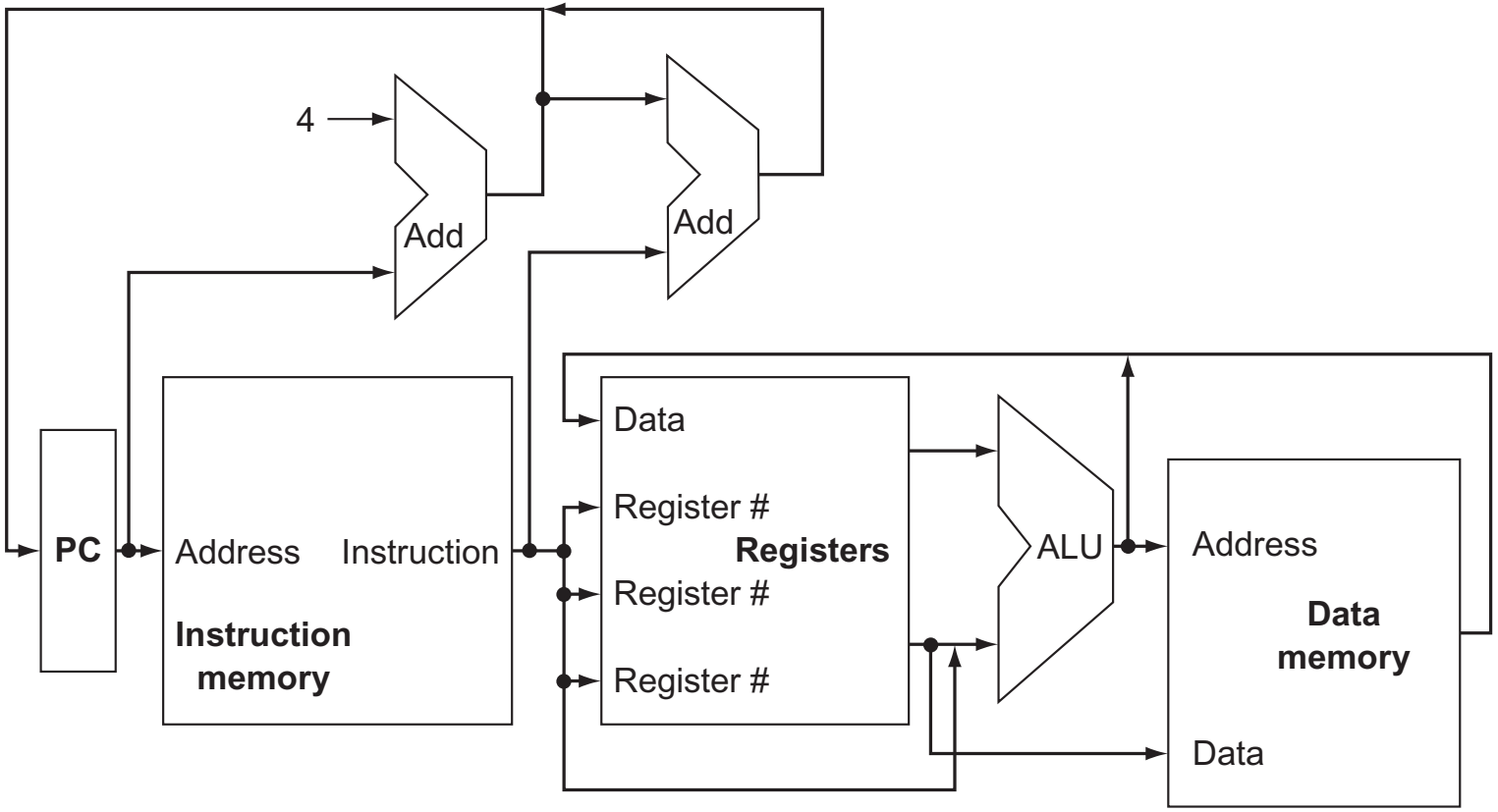
- O desempenho da CPU é influenciado por
  - Número de instruções executadas
    - Determinado pela ISA e pelo compilador
  - CPI e Frequência
    - Determinada pelo hardware
- Vamos estudar duas implementações de MIPS
  - Uma versão simplificada monociclo
  - Uma versão mais realista, com pipelines
- Vamos nos concentrar em um subconjunto das instruções que, apesar de simples, demonstram a maior parte dos detalhes
  - Acesso à memória: `lw`, `sw`
  - Aritimética e lógica: `add`, `sub`, `and`, `or`, `slt`
  - Branches e saltos: `beq`, `j`

Execução de instruções

UFABC

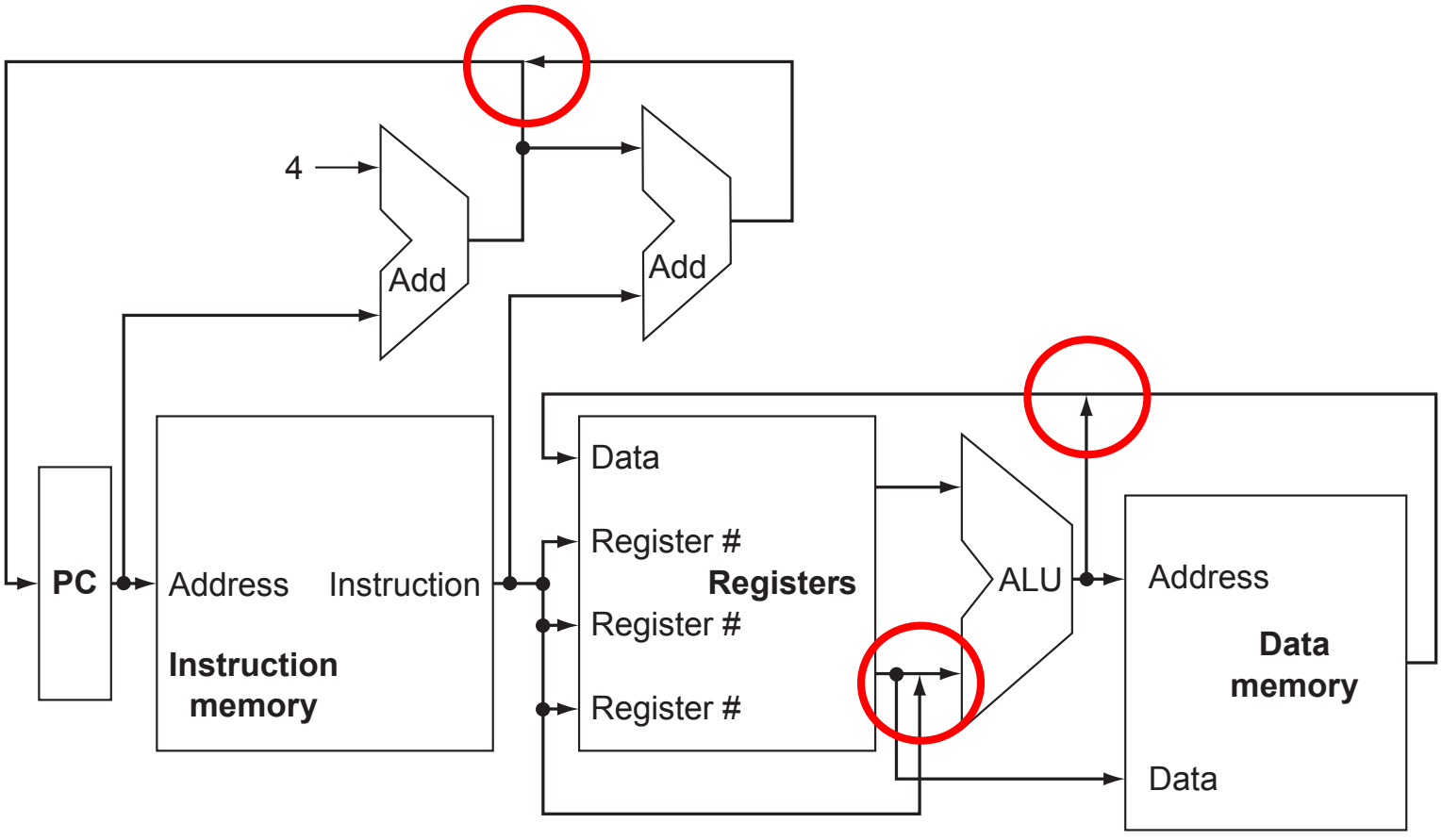
- PC $\rightarrow$ memória de instruções, carrega instrução
- Números de registradores $\rightarrow$ banco de registradores, leitura de registradores
- Dependendo to tipo de instrução
  - Usa a ALU para calcular
    - Resultado aritmético
    - Endereço de memória para load/store
    - Endereço para branch
  - Acessa a memória de dados para load/store
  - PC $\leftarrow$ endereço de destino ou PC + 4

Uma visão geral do processador



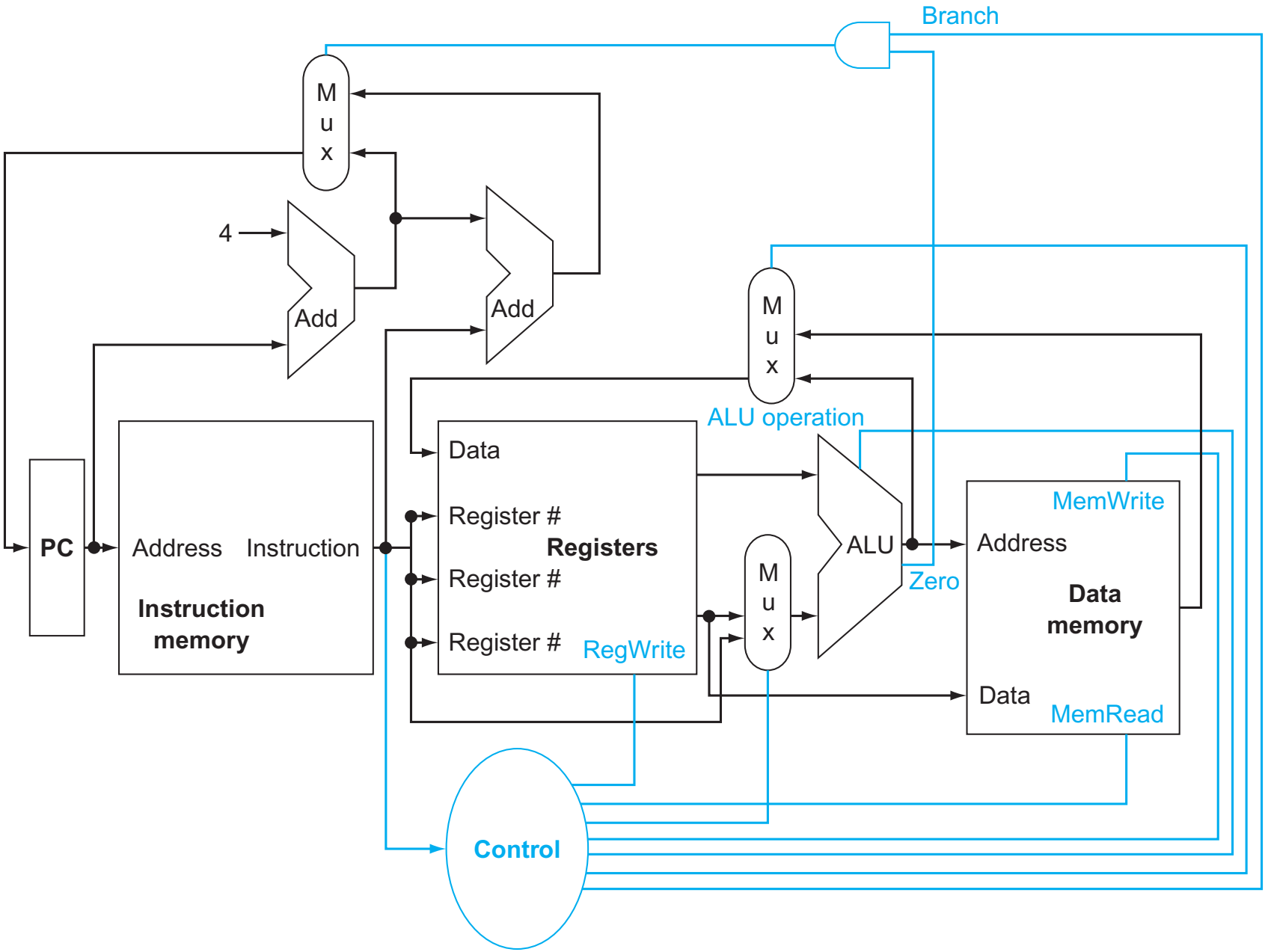
- Elementos combinacionais que operam nos dados
- Elementos sequenciais que armazenam o estado

Multiplexadores



- Não podemos sair juntando fios. Precisamos de multiplexadores para selecionar qual sinal é de interesse.

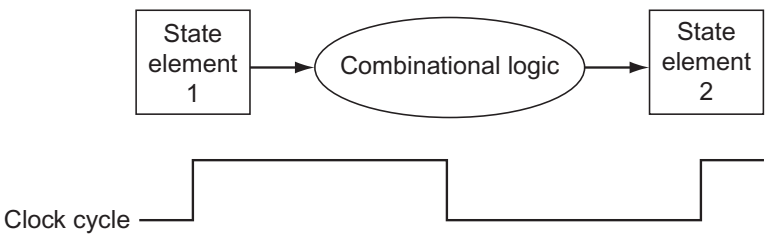
Controle

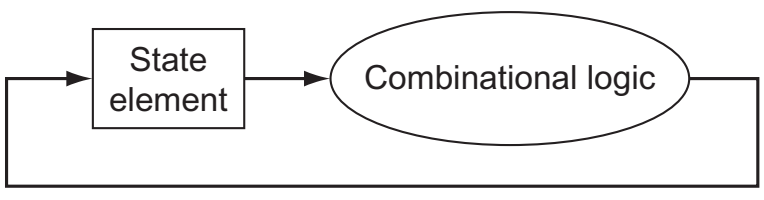

Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

Clocking methodology

UFABC

- Lógica combinacional transforma os dados durante os ciclos de clock
  - Entre os bordas do clock
  - Entrada de elementos de estado, saída para elemento de estado
  - O atraso mais longo determina o período do clock



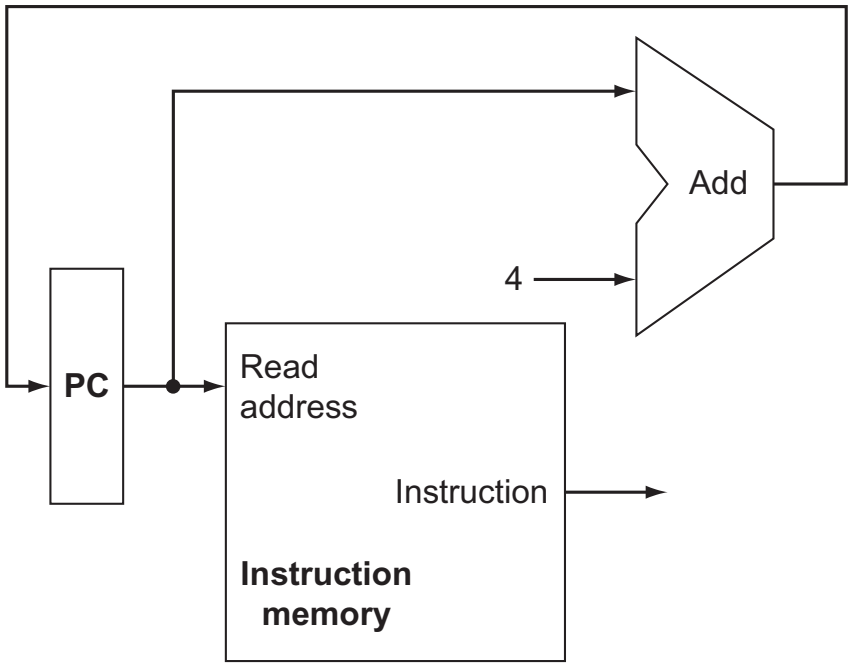


Datapath Monociclo

UFABC

- Datapath
  - Elementos que processam os dados e endereços na CPU.
    - Registradores, ALUs, mux's, memórias, ...
- Vamos construir um datapath de MIPS de maneira incremental.
  - Refinaremos pouco a pouco o design geral que vimos.

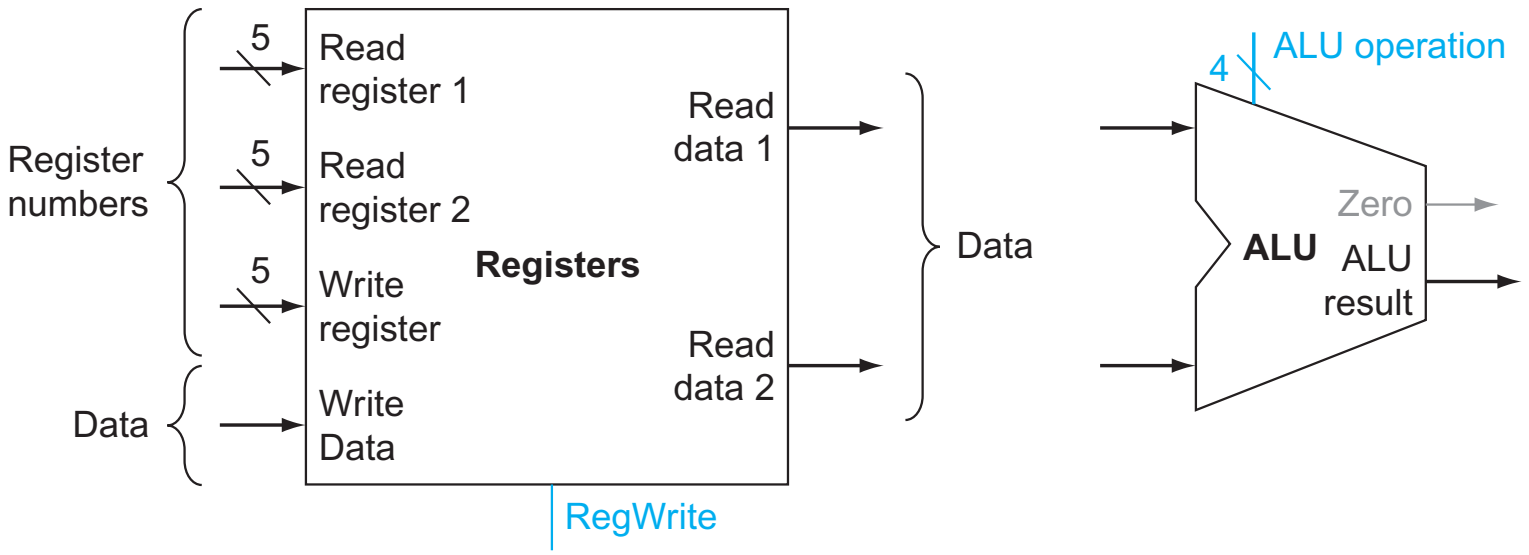
Intruction Fetch



- PC é um registrador de 32 bits.
- ALU soma 4 no PC para pular para a próxima instrução.

Instruções R-Format

- Leem 2 operandos de registradores
- Fazem uma operação lógica/aritmética
- Escrevem o resultado em um registrador

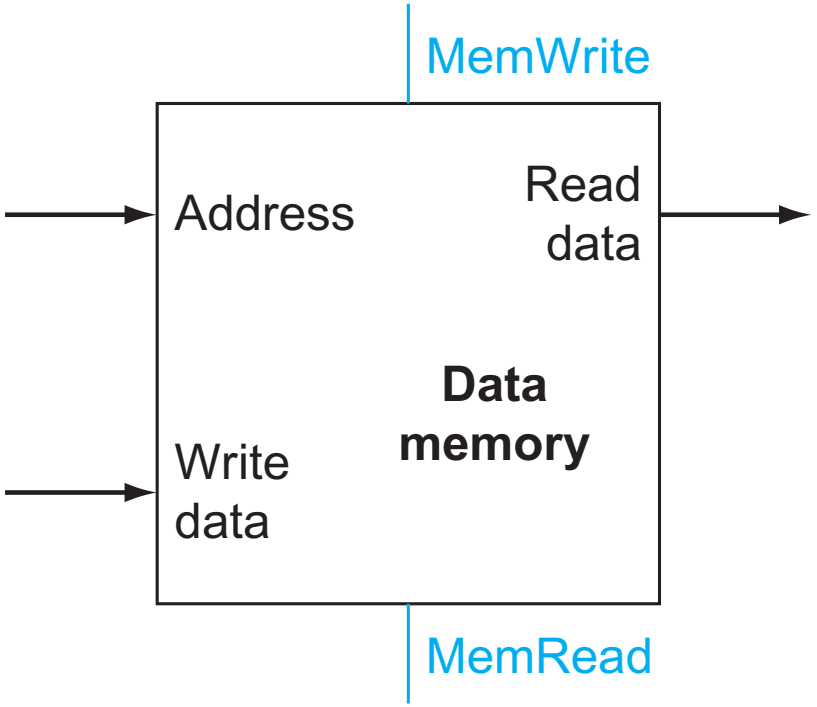


a. Registers

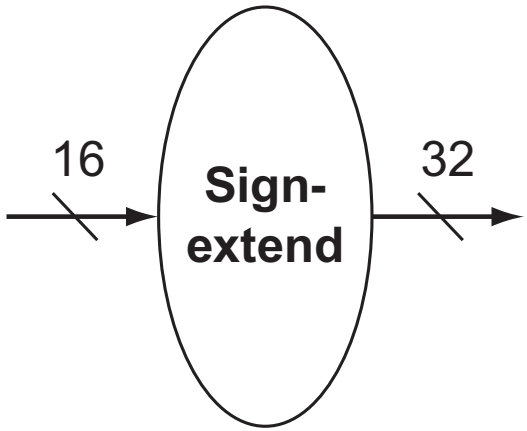
b. ALU

Instruções de Load/Store

- Lê os operandos dos registradores
- Calcula o endereço usando um offset de 16 bits
  - Usa a ALU, mas faz a extensão de sinal do offset
- Load: lê a memória e atualiza o registrador apropriado
- Store: escreve o valor do registrador na memória



a. Data memory unit



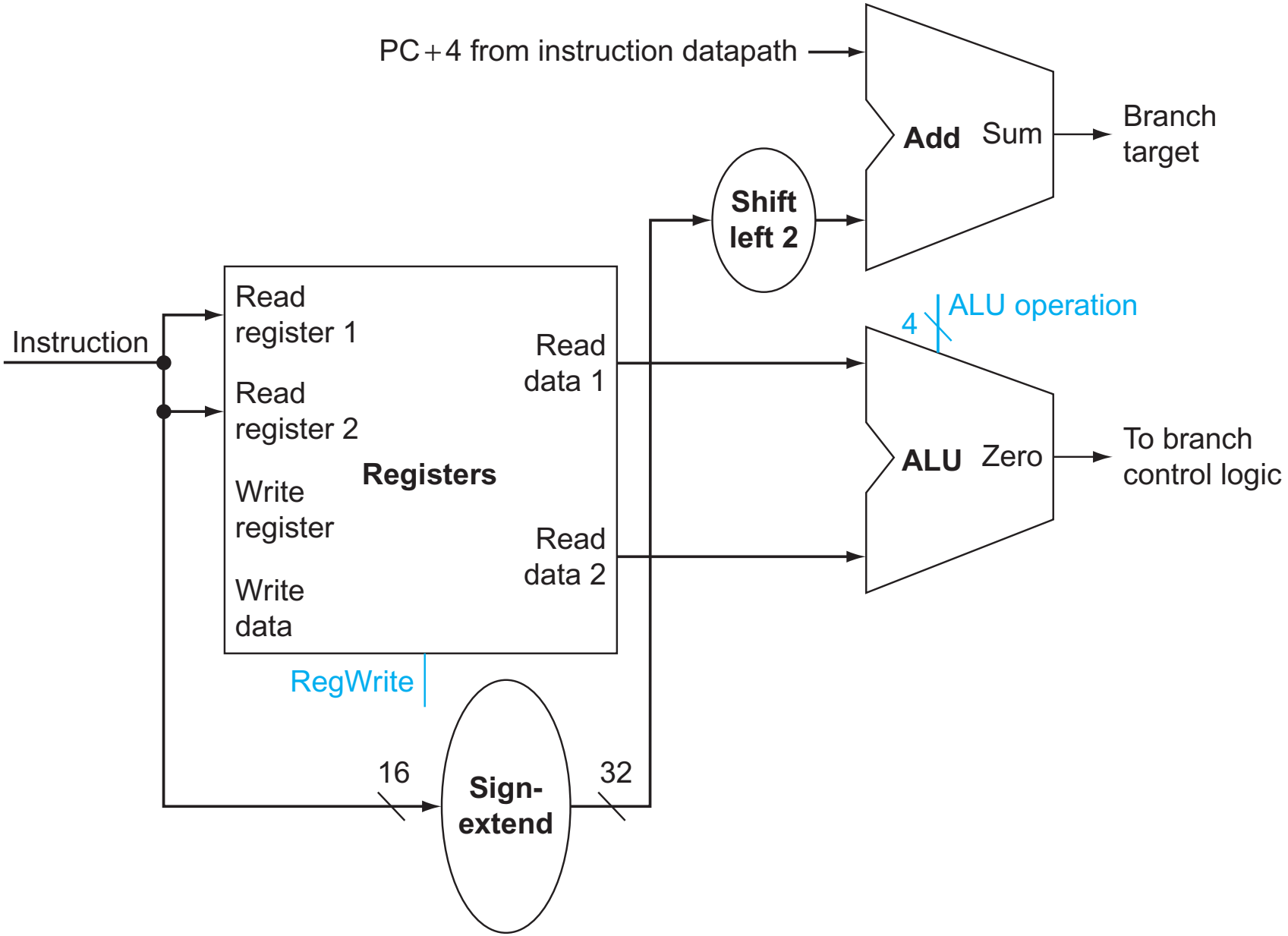
b. Sign extension unit

Instruções de Branch

UFABC

- Lê os operandos dos registradores
- Compara os operandos
  - Usa a ALU, subtrai os operando e usa a saída `Zero`
- Calcula o endereço alvo
  - Estende o deslocamento (com sinal)
  - Faz um shift de 2 (uma palavra)
  - Adiciona 4 ao PC
    - Já feito pelo passo de fetch de instruções

Instruções de Branch

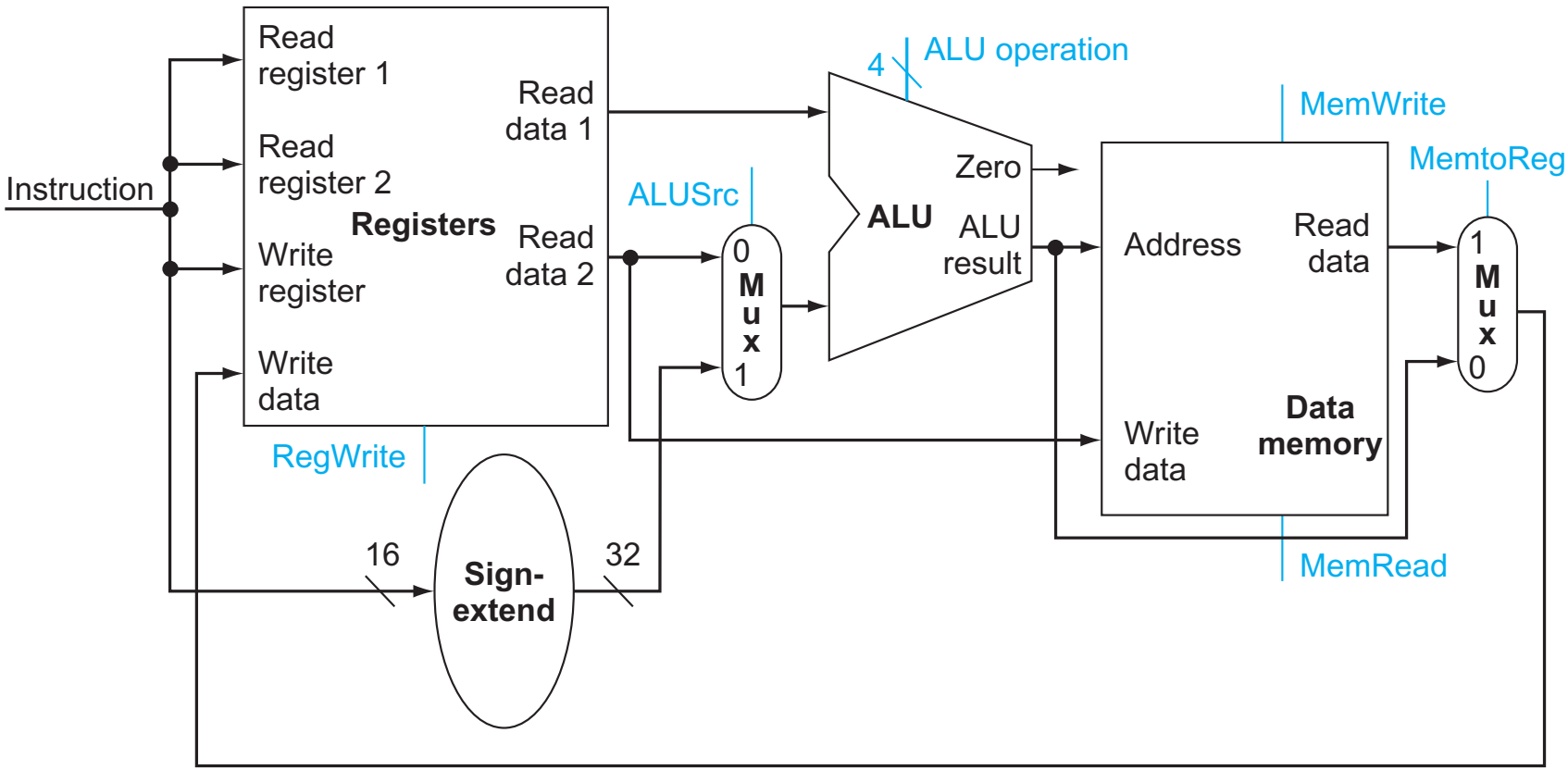


Juntando tudo

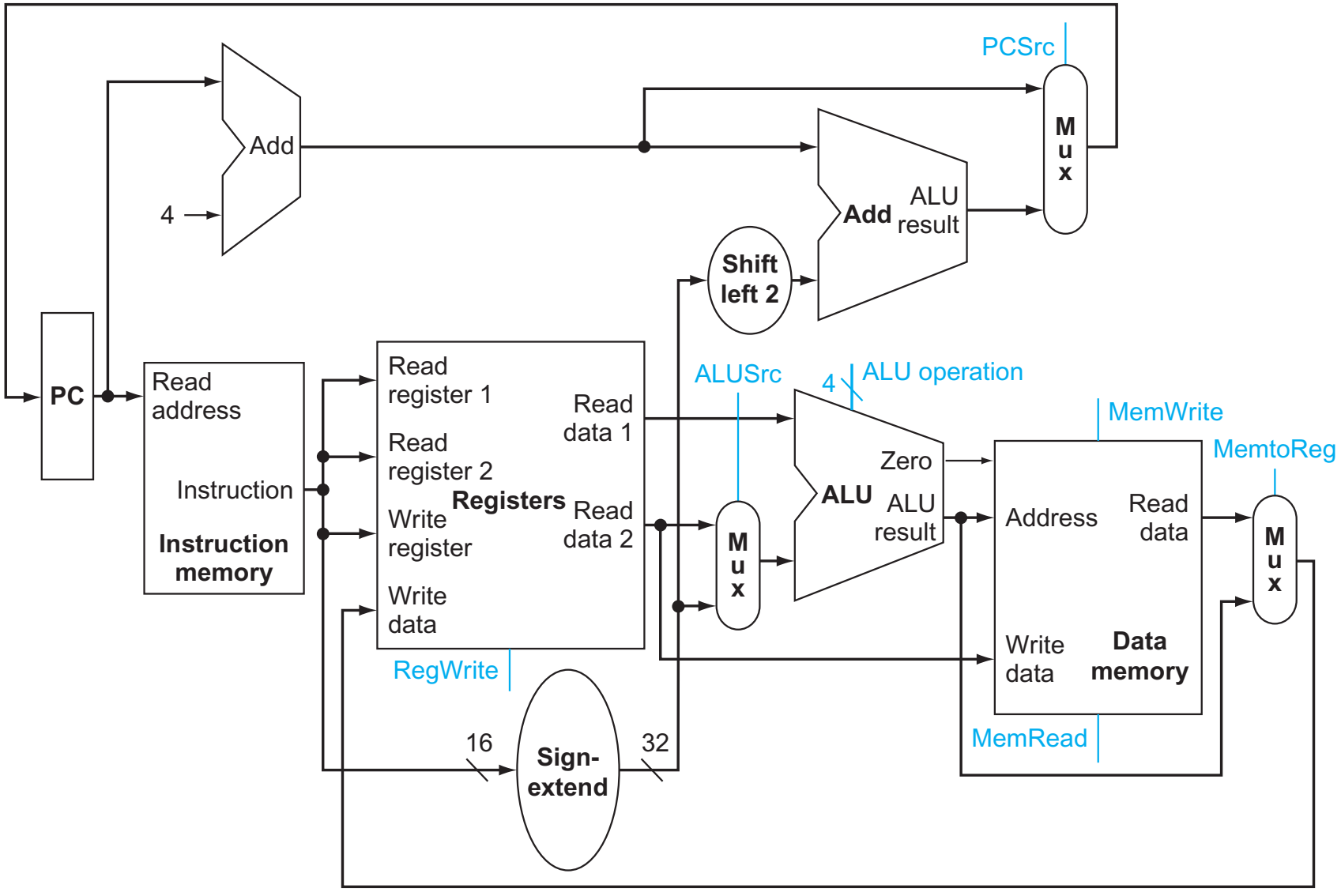
UFABC

- O Datapath executa uma instrução por ciclo
  - Cada elemento do datapath só pode fazer uma função por vez
  - Logo precisamos separar as memórias de instruções e de dados
- Emprega-se multiplexadores nos locais onde fontes de dados alternativas são usadas por diferentes instruções.

R-type/Load/Store Datapath



O Datapath completo



Controle da ALU

- ALU usada para
  - Load/Store: Função = **ADD**
  - Branch: Funcão = **SUB**
  - R-Type: Função depende do campo `funct`

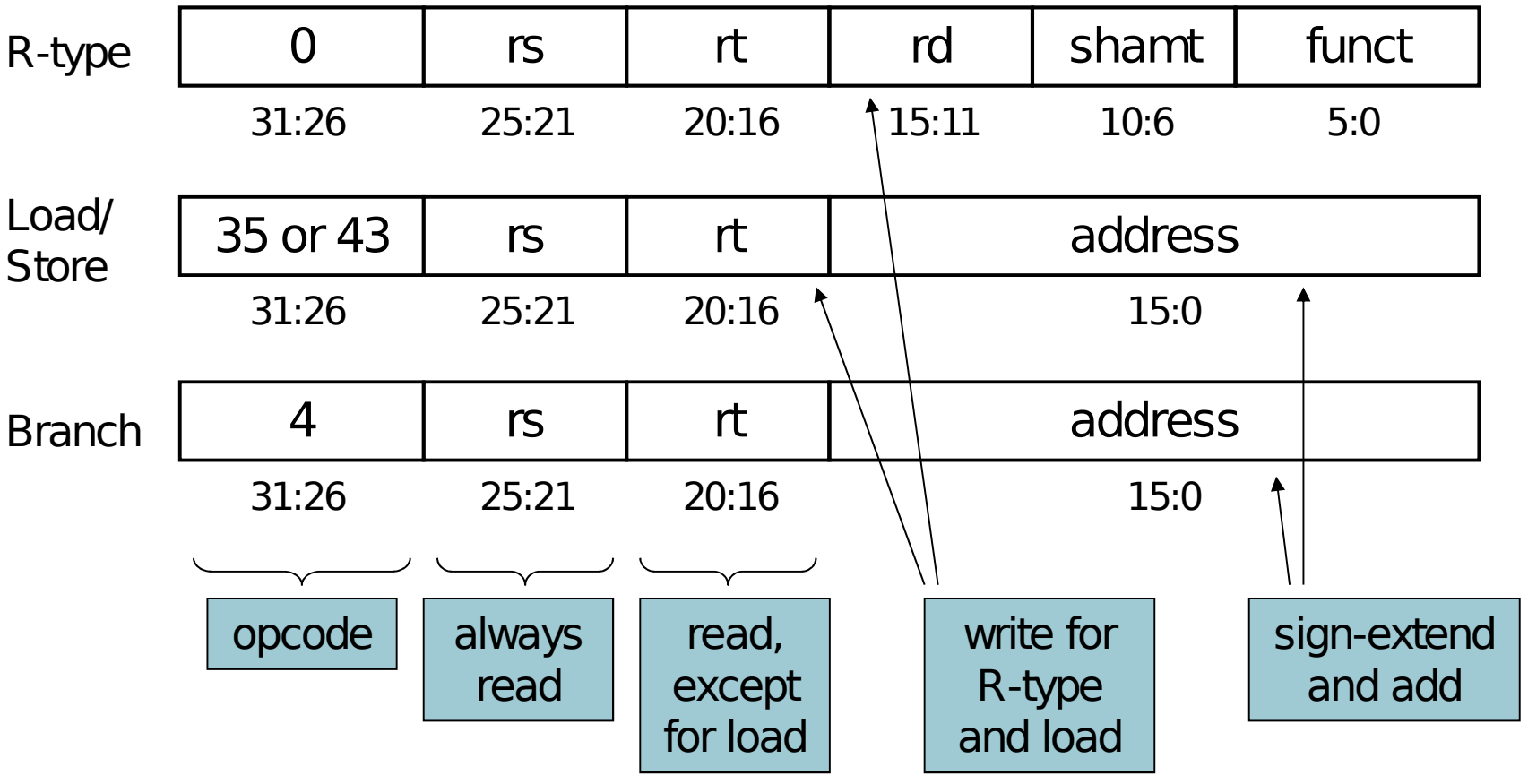
ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

UFABC

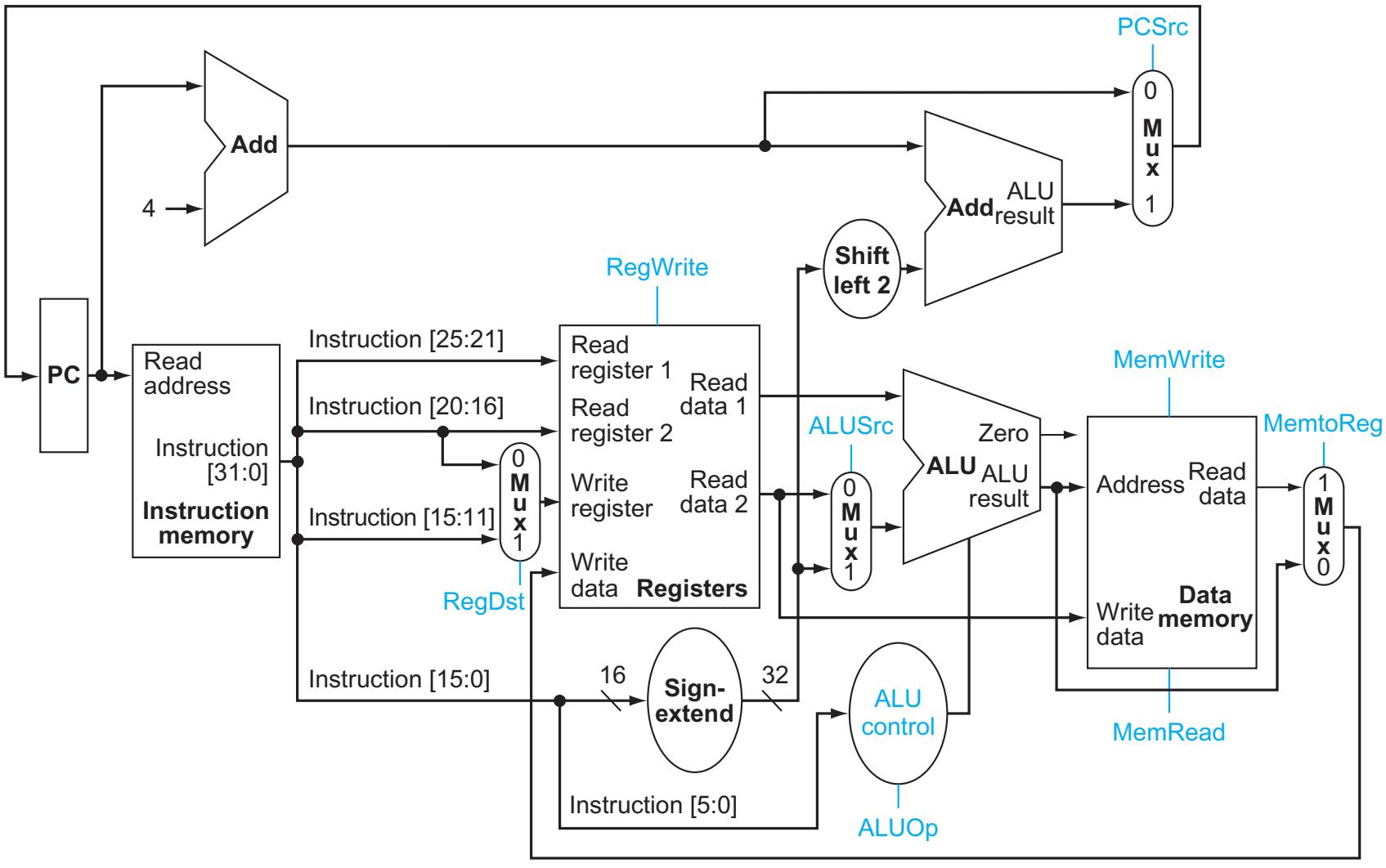
- Assuma que **ALUOp** de 2 bits é derivada do opcode
  - Lógica combinacional é usada par derivar o controle da ALU

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

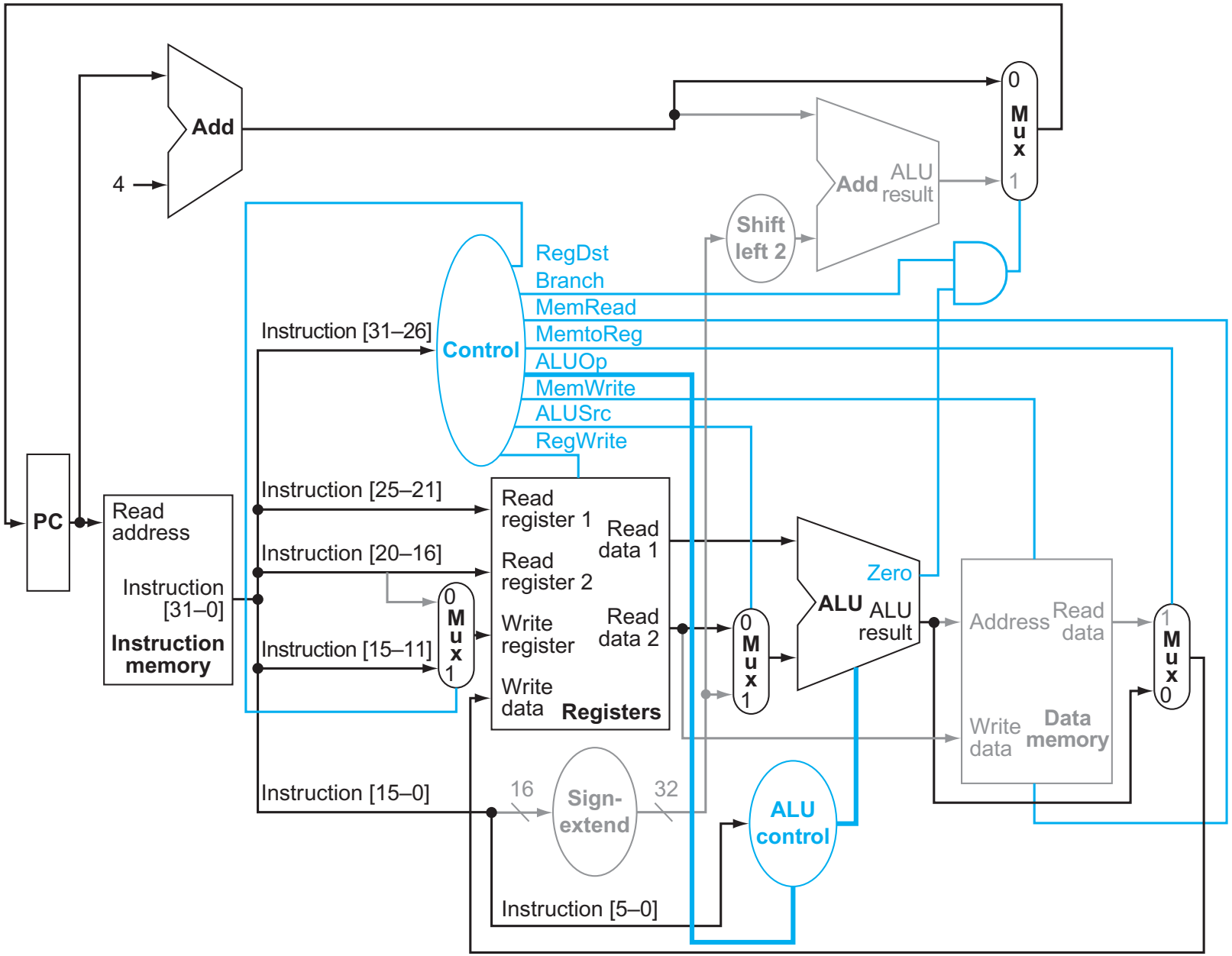
A unidade de controle principal



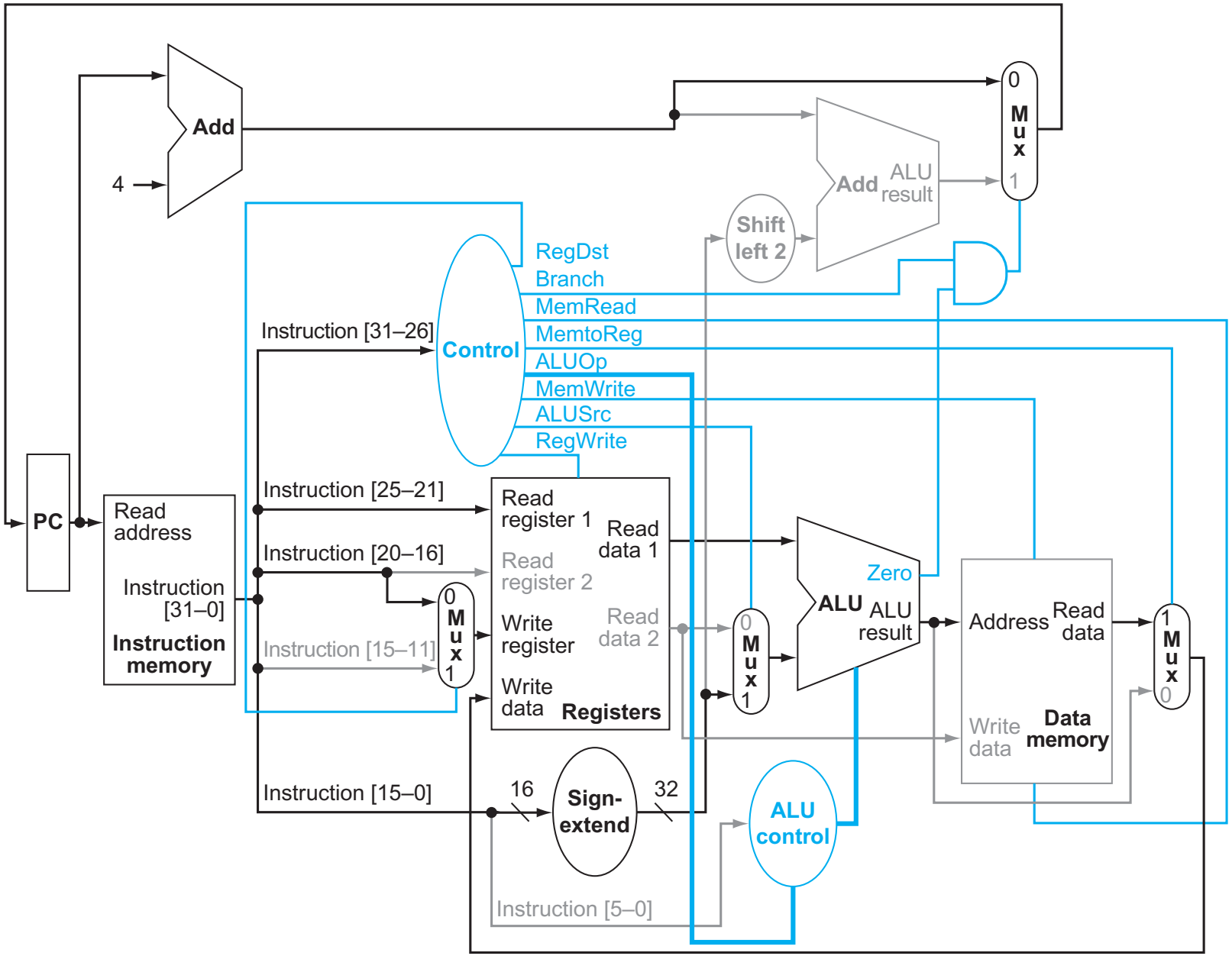
O Datapath com a unidade de Controle



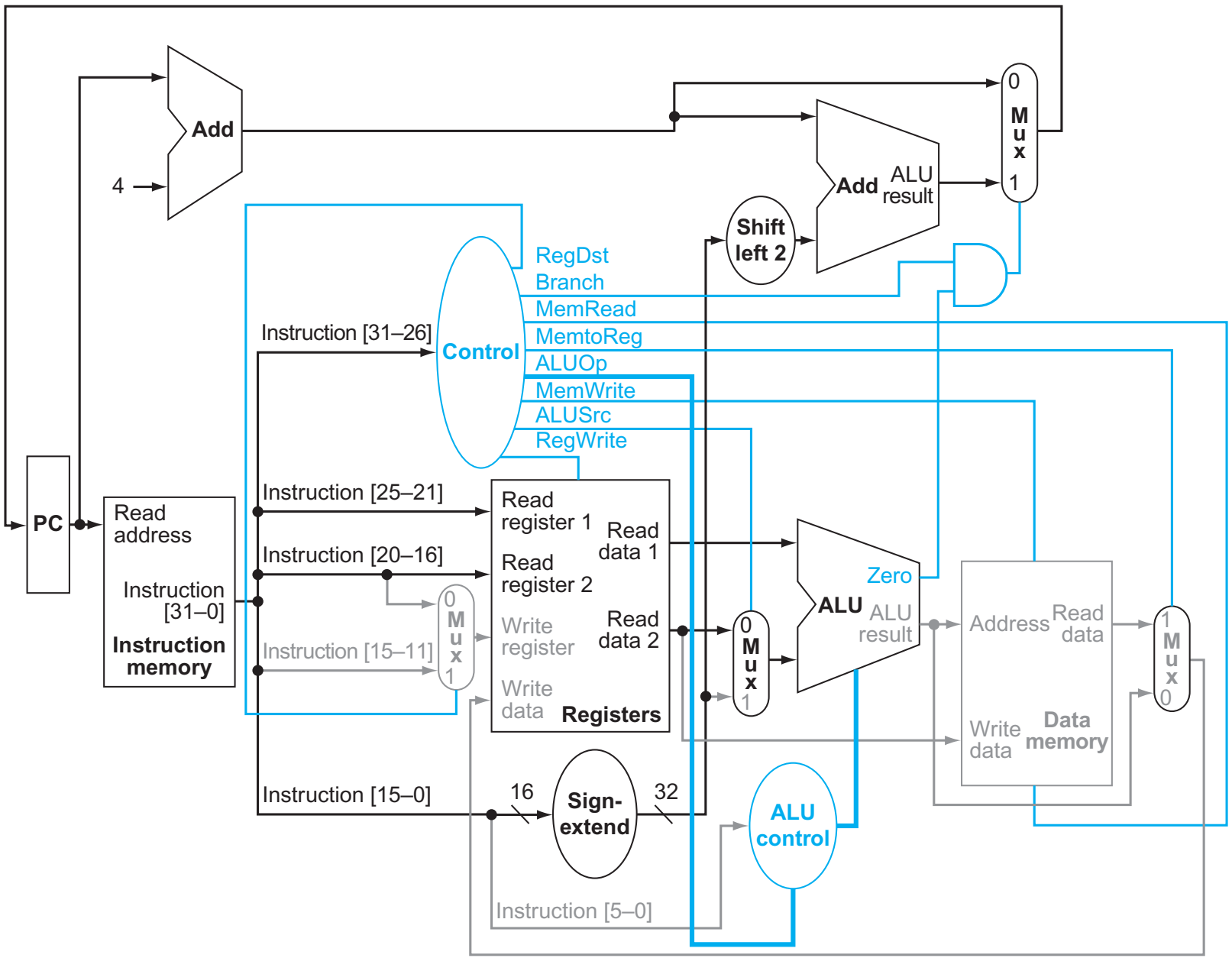
Instruções R-Type



Instruções Load



Instrução Branch-on-Equal



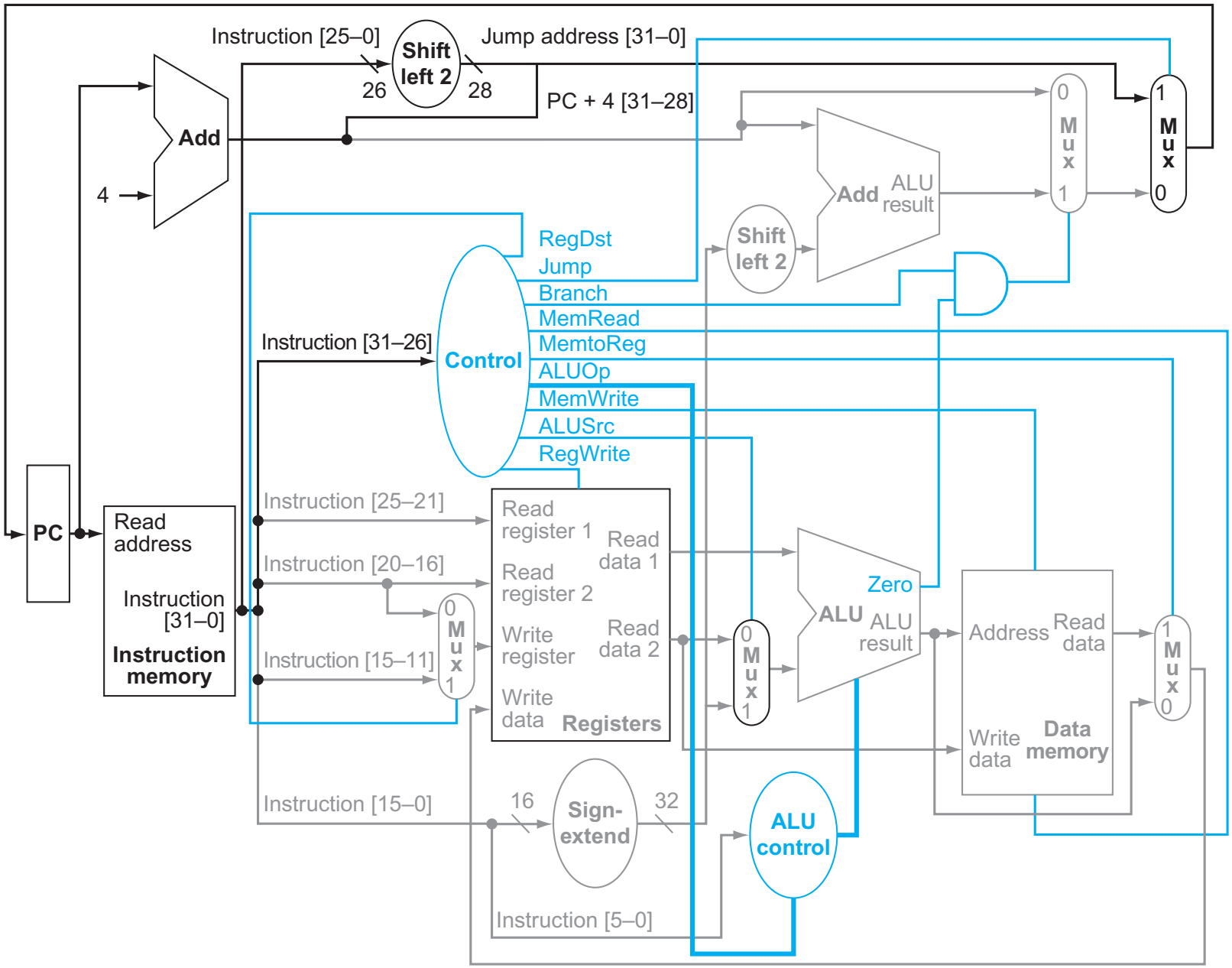
Implementando Jumps

UFABC

Field	000010	address
Bit positions	31:26	25:0

- Jump usa o endereço de uma palavra
- Atualiza o PC com a concatenação dos
  - 4 bits mais altos do PC
  - Endereço de 26 bits da instrução de jump
  - 00
- Precisa de um sinal de controle extra (que pode ser gerado pelo opcode)

Datapath com Jumps



Porque este tipo de Datapath não é usado hoje em dia?

UFABC

- O delay mais longo determina o período do clock
  - Caminho crítico - Load Instruction
  - Memória de instrução $\rightarrow$ banco de registradores $\rightarrow$ ALU $\rightarrow$ memória de dados $\rightarrow$ banco de registradores
- Não é razoável alterar o período a cada instrução
- Viola o princípio de tornar rápido o caso mais comum
- Veremos como melhorar o desempenho através de pipelining