5. O Processador – Datapath e Unidade de Controle

- **Datapath ➔ Componente do processador que realiza operações aritméticas**

- **Controle ➔ Componente do processador que comanda o datapath, memória e dispositivos de E/S de acodo com as instruções de um programa**

- **Independente da classe da instrução, as duas primeiras etapas para sua execução são as mesmas:**

    - **Enviar o PC para a memória e buscar ao instrução**

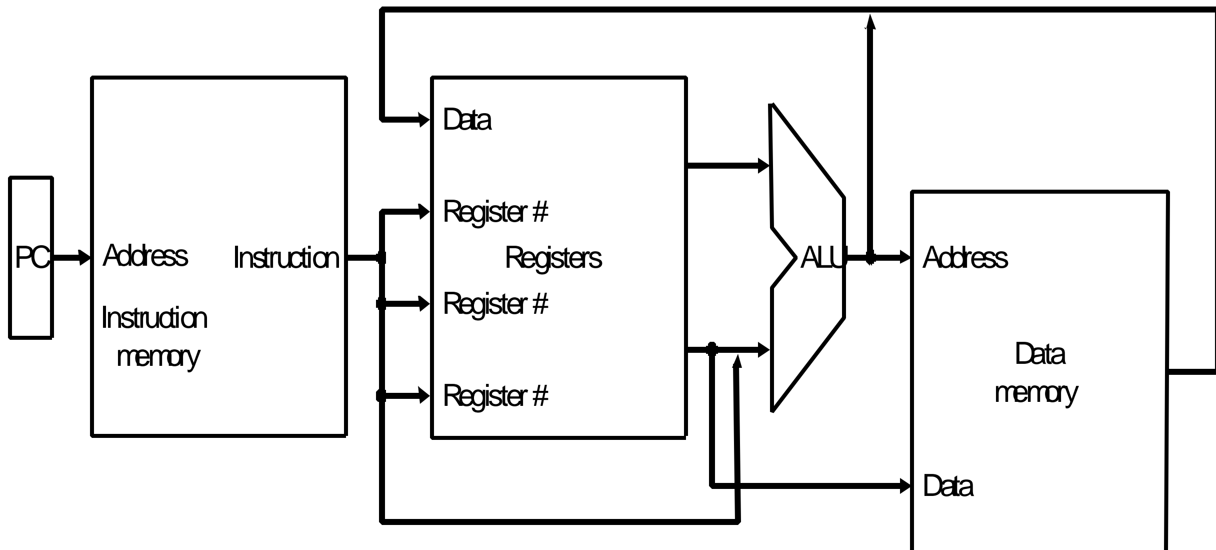
    - **Ler um ou dois registradores (usando o campo da instrução, para selecionar os registradores a serem lidos)**

- **Os passos seguintes dependem da classe da instrução (referência à memória, lógica-aritmética e desvios) ➔ estes passos são bastantes semelhantes e independem do opcode**

    - **Por exemplo, todas as instruções, independente da classe utilizam a ULA após a leitura de um registrador. Para uma instrução de referência à memória, utiliza para cálculo do endereço, lógica-aritmética para execução e desvios para comparação**

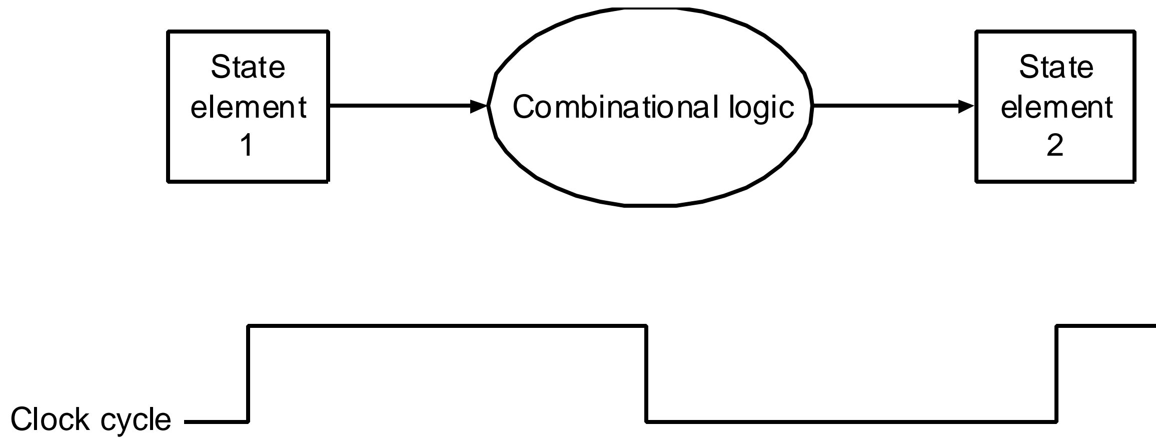
- **Após a utilização da ULA, os passos são diferentes para as diferentes classes.**

- **Figura 5.1 - Implementação do MIPS – visão em alto nível**

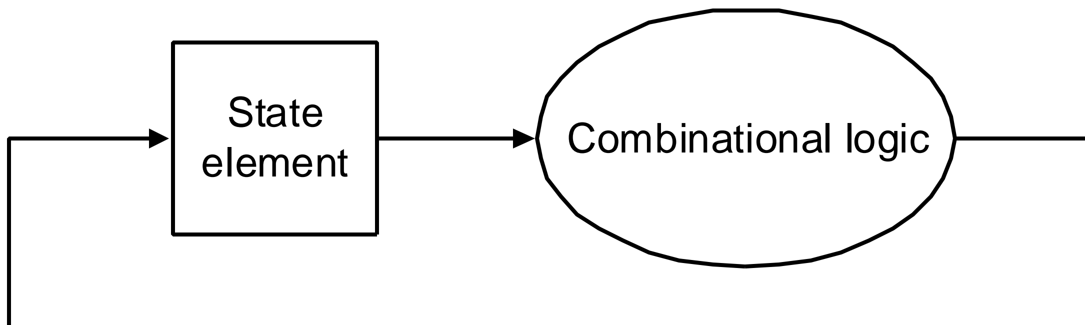


- **Revisão/Convenções adotadas**
    - **Sinal lógico alto –** ***asserted***
    - **Sinal que pode ser logicamente alto -** ***assert***
    - **Elementos combinacionais ➔ Exemplo: ULA**
    - **Elementos de estado ➔ Exemplo: Registradores e Memória**
    - **Sinal de Clock ➔ usado para determinar quando se pode escrever em um elemento de estado. A leitura pode ser a qualquer momento**
- **Metodologia de sincronização ➔ sincroniza o elemento de estado para a permissão de leitura e de escrita ➔ Porque é necessário ?**

- **Figura 5.2 – Lógica combinacional, elemento de estados e clock(sensível à subida).**



- **Figura 5.3 – A metodologia edge-triggered permite a um elemento de estado ler e escrever no mesmo período de clock.**



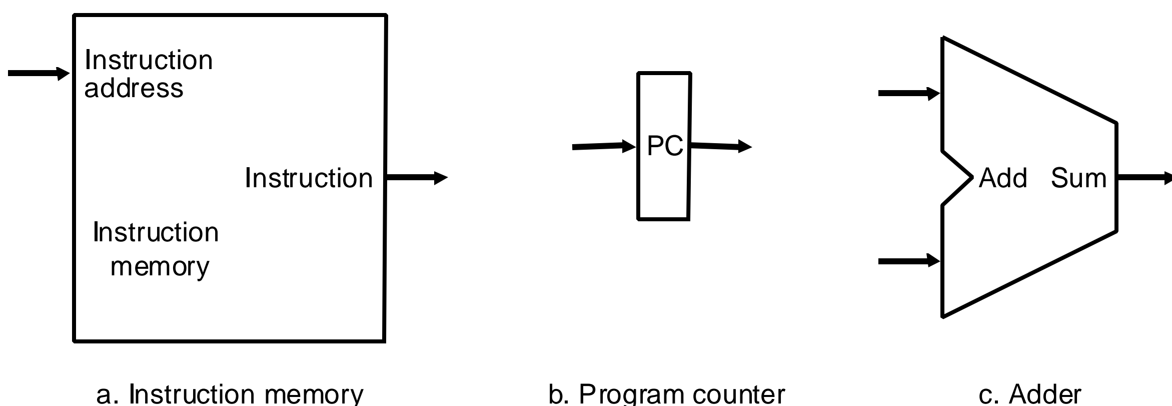
- **Datapath**

  - **OBS.: Primeiro implementaremos um Datapath utilizando apenas um clock com ciclo grande. Cada instrução começa a ser executada em uma transição e acaba na próxima transição do clock ➔ na prática isto não é factível, pois temos instruções de diferentes classes e portanto de diferentes números de ciclos de clock**

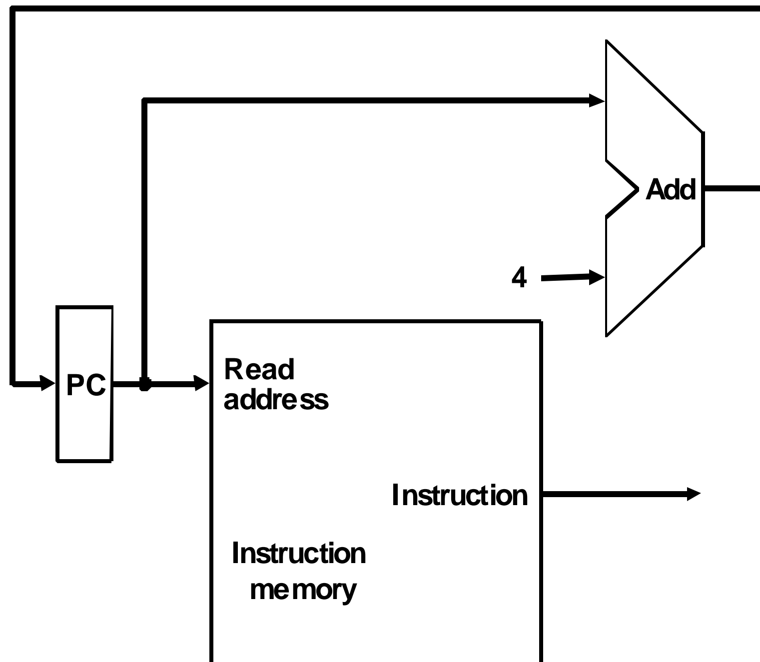
  - **Para construir um Datapath:**

    - **Um lugar para armazenar as instruções do programa ➔ Memória de instruções**
    - **Um lugar para armazenar o endereço da instrução a ser lida na memóia ➔ Program Counter - PC**
    - **Somador para incrementar o PC para que ele aponte para a próxima instrução**

- **Figura 5.4 – Elementos necessários a armazenar e acessar informações mais um somador para calcular o endereço do próximo estado.**

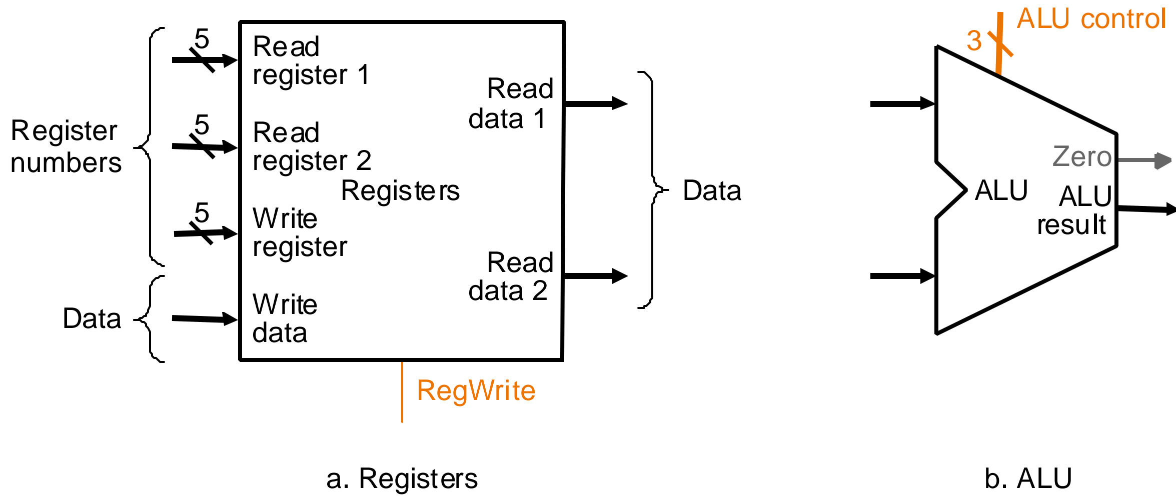


- **Figura 5.5 – Parte do datapath para fetch e incremento de PC**

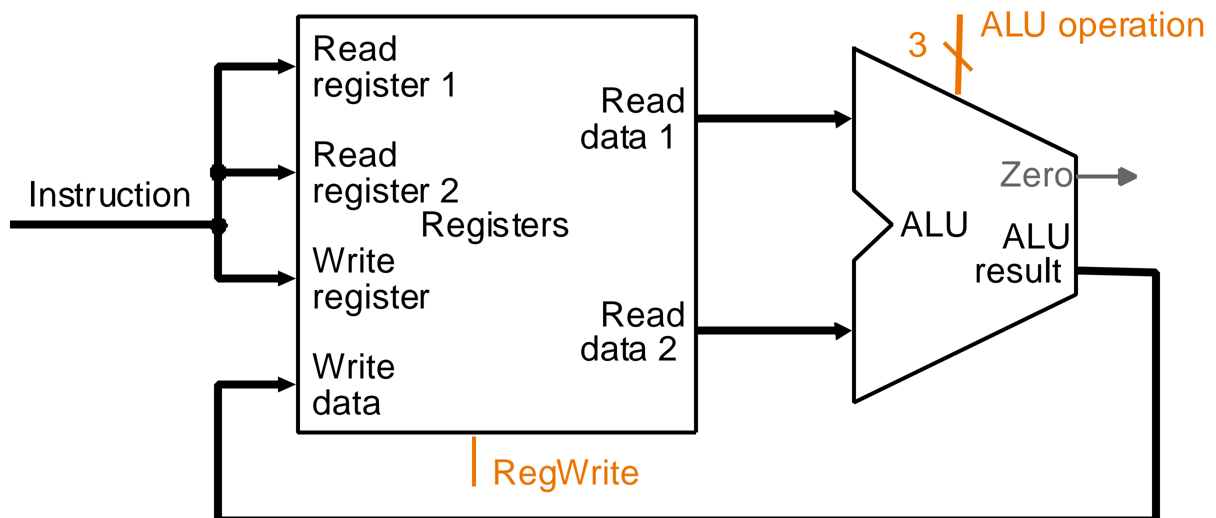


- **Instruções R-type**
  - **Instruções aritméticas e lógicas ➔ add, sub, slt**
  - **Estrutura de registradores (32) chamada de register file ➔que é um conjunto de registradores que podem ser acessados (lidos ou escritos) especificando seu número. Nele se encontra o registrador de estado da máquina**
  - **Instruções de formato R tem 3 operandos registradores (add $t1,$t2,$t3) ➔necessidade de ler 2 dados do register file e escrever um dado nele, para cada instrução**
  - **Para ler um dado do register file é preciso de uma entrada (número do do registrador) e uma saída (o dado lido)**
  - **Para escrever um dado no register file, são necessárias duas entradas: o número do registrador e o dado a ser escrito**
  - **Para escrever ➔ sinal de controle (RegWrite)**
  - **A ULA é controlada por um sinal de controle (ALU control)**

- **Figura 5.6 – Elementos necessários para a implementação de operações da ULA, de instruções do tipo-R.**



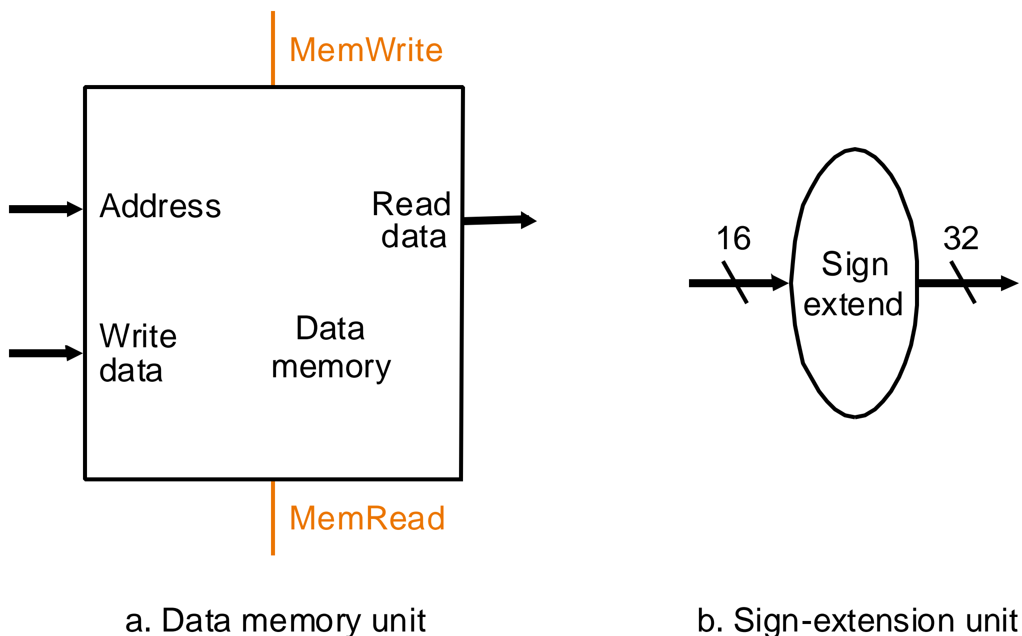
- **Figura 5.7 – O datapath para instruções do tipo R**



- **Instruções LOAD e STORE**

  - **lw $t1,offset_value($t2) e sw $t1,offset_value,($t2)**
  - **Endereço de memória = value_offset + $t2**
  - **Value_offset → offset sinalizado de 16 bits**
  - **É preciso de um register file e uma ULA**
  - **Unidade que transforme valor de 16 bits sinalizado em um valor de 32 bits**
  - **Unidade de memória de dados com controle de leitura (MemRead) e escrita (MemWrite)**

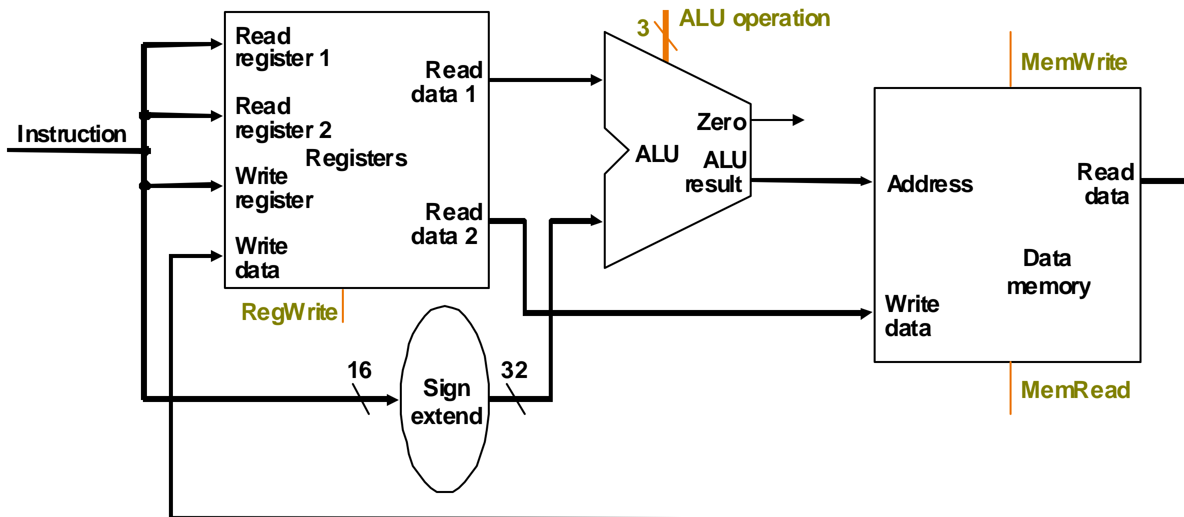
- **Figura 5.8 – Unidades para implementação de lw e sw**



a. Data memory unit

b. Sign-extension unit

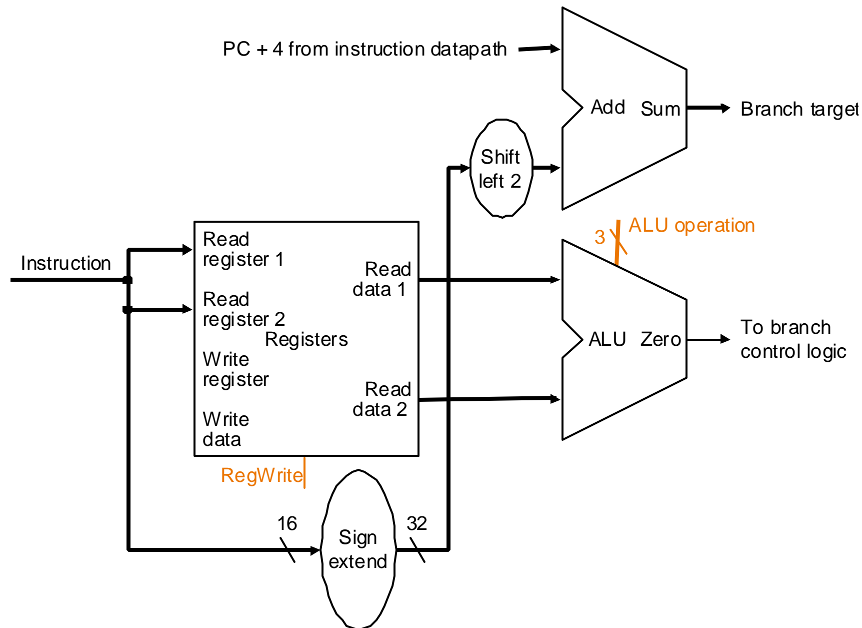
- **Figura 5.9 – Datapath para instruções lw e sw.**



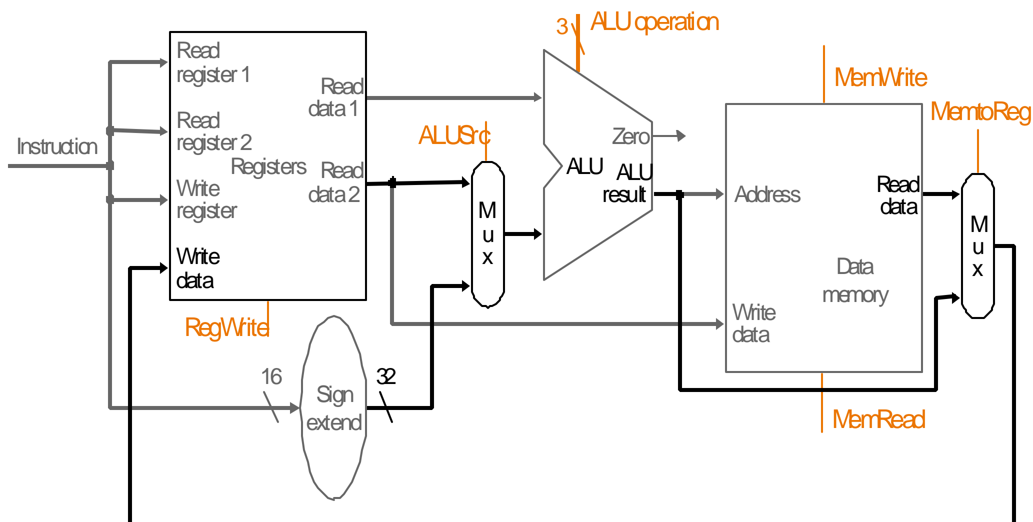
- **Instrução beq**

  - **beq $t1,$t2,offset ➔2 registradores que são comparados e um offset de 16 bits usado para calcular o endereço relativo, alvo do branch**
  - **A base para o cálculo do endereço alvo de branch é o endereço da próxima instrução em relação à instrução branch**
  - **O campo offset é deslocado de 2 bits para aumentar o alcance do desvio (multiplicado por 4)**
  - **Além do cálculo do endereço do branch, deve-se determinar qual endereço será escolhido, o do branch (taken) ou o armazenado em PC (not taken) dependendo do resultado da comparação**
  - **OBS.: Instrução jump ➔os 28 bits menos significativos de PC são substituídos pelos 26 bits do imediato, deslocado de 2 bits (X 4).**

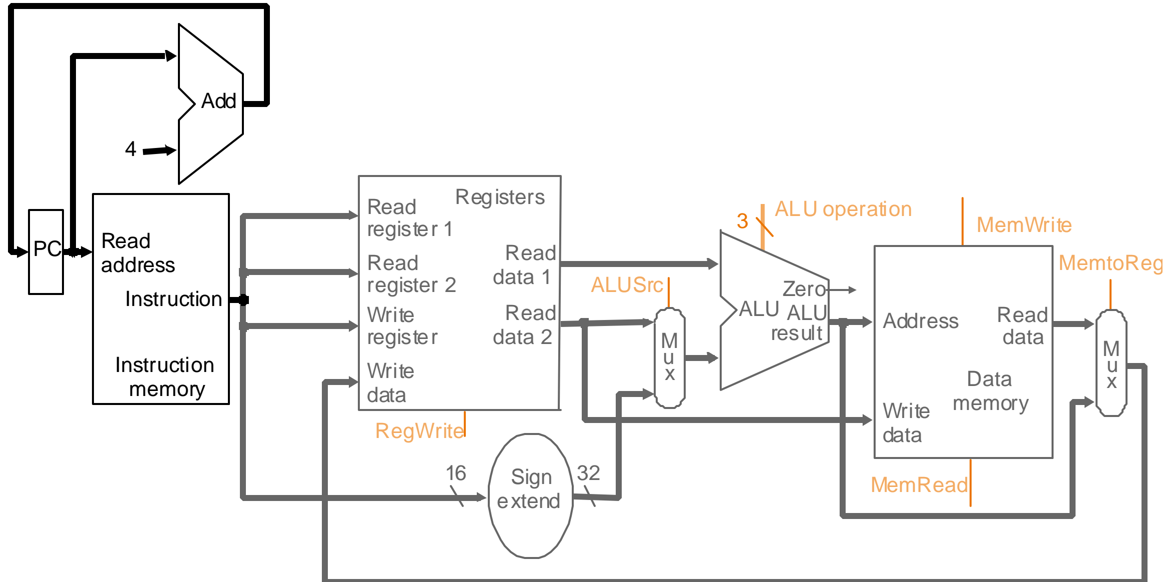
- **Figura 5.10 – Datapath para branchs**



- **Datapath geral - Instruções de memória + instruções R-type (instruções da ULA)**



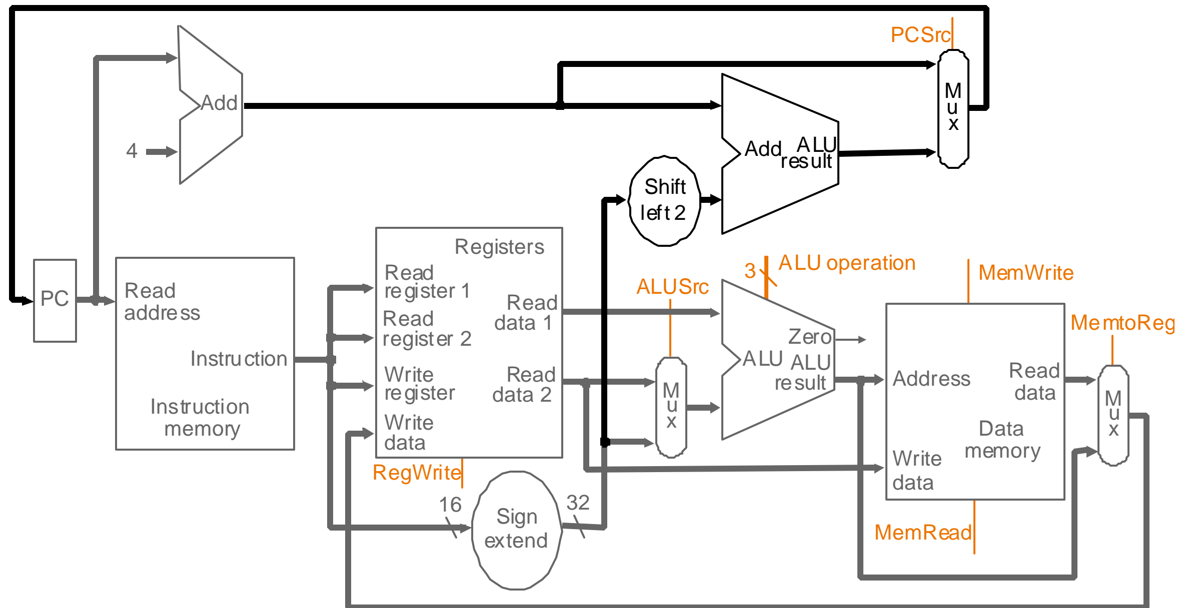
- **Figura 5.12 - Fetch da instrução + execução de instruções de memória e da ULA**



- **Controle da ULA**

Sinal de Controle da ULA	Função
000	AND
001	OR
010	ADD
110	SUB
111	SLT

- **Datapath para as classes de instruções do MIPS**



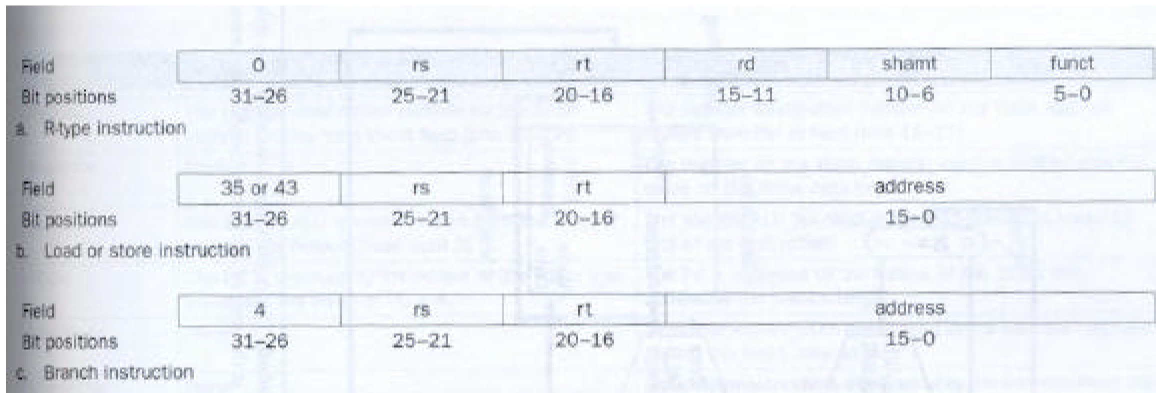
Tabelas – figuras 5.14 e 5.15

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	010
SW	00	store word	XXXXXX	add	010
Branch equal	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less than	101010	set on less than	111

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

- **Projeto da Unidade de Controle Principal**

Figura 5.16 – Classes de instruções - tipo-R, load&store e branch)



Field	0	rs	rt	rd	shamt	funct
Bit positions	31-26	25-21	20-16	15-11	10-6	5-0

a. R-type instruction

Field	35 or 43	rs	rt	address
Bit positions	31-26	25-21	20-16	15-0

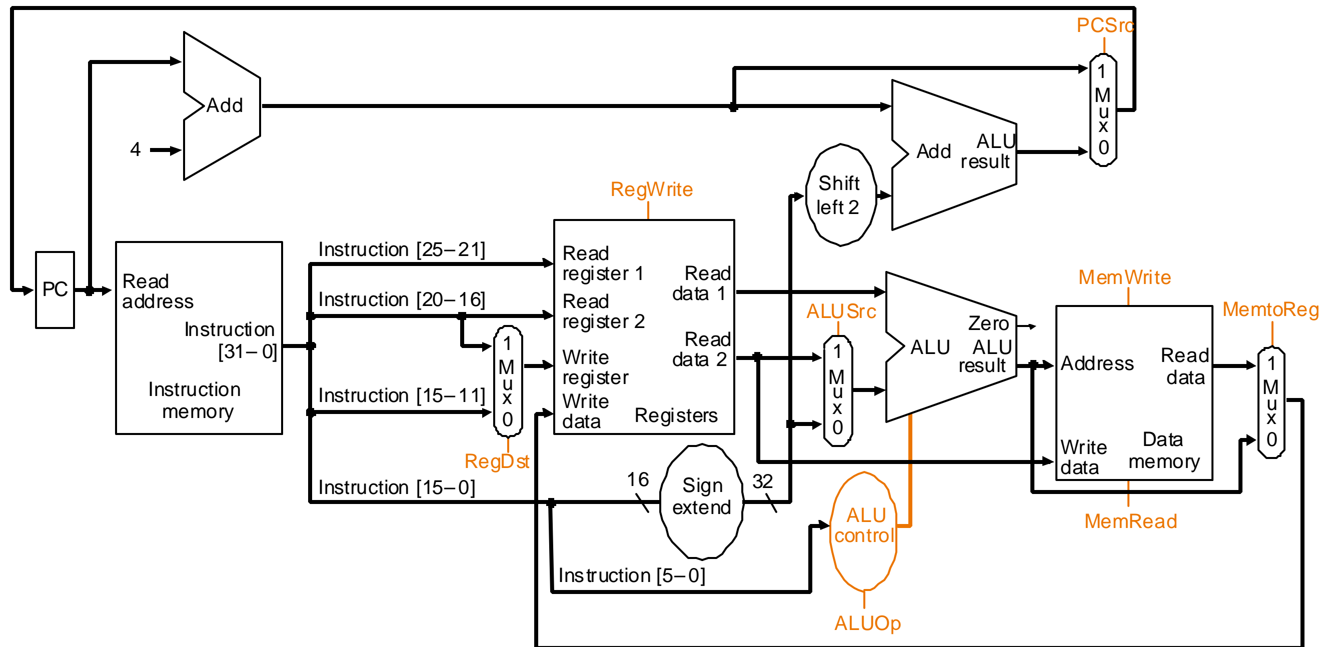
b. Load or store instruction

Field	4	rs	rt	address
Bit positions	31-26	25-21	20-16	15-0

c. Branch instruction

- **O campo opcode ➔ bits 31-26**
- **Os dois registradores a serem lidos ➔ rs e rt ➔ bits 25-21 e 20-16**
- **O registrador base (load e store) ➔ bits 15-0**
- **O valor a ser guardado no PC que pode vir do cálculo de um endereço de salto ou simplesmente do PC + 4.**
- **O registrador destino (a ser escrito), que deverá ser selecionado dentre 2 opções, o que requer um multiplexador:**
  - **para um load ➔ bits 20-16 (rt)**
  - **para instruções R-type ➔ bits 15-11 (rd)**
- **O valor guardado no banco de registradores que pode vir da ALU (R-type) ou da memória (sw)**

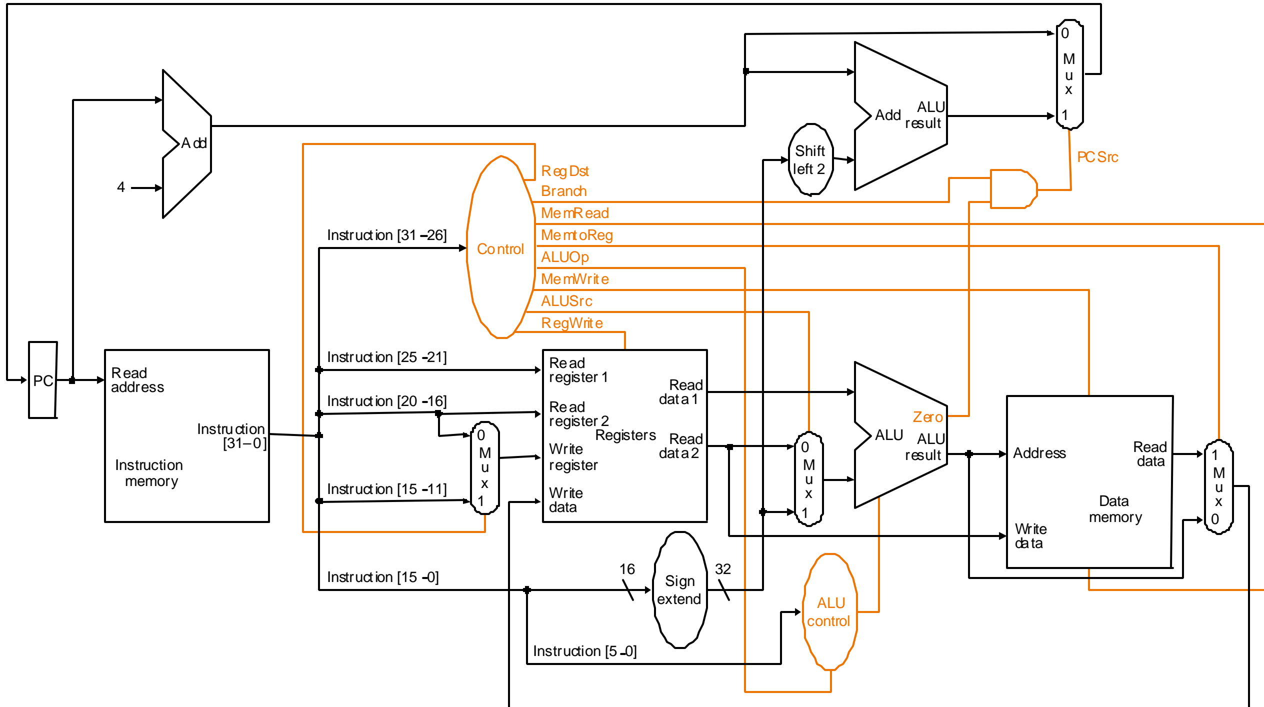
- **Figura 5.17 - Datapath com os multiplexadores necessários**



- **Sinais de controle – Tabela da figura 5.18**

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20–16).	The register destination number for the Write register comes from the rd field (bits 15–11).
RegWrite	None	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

- **Figura 5.19 - Datapath com a Unidade de Controle**

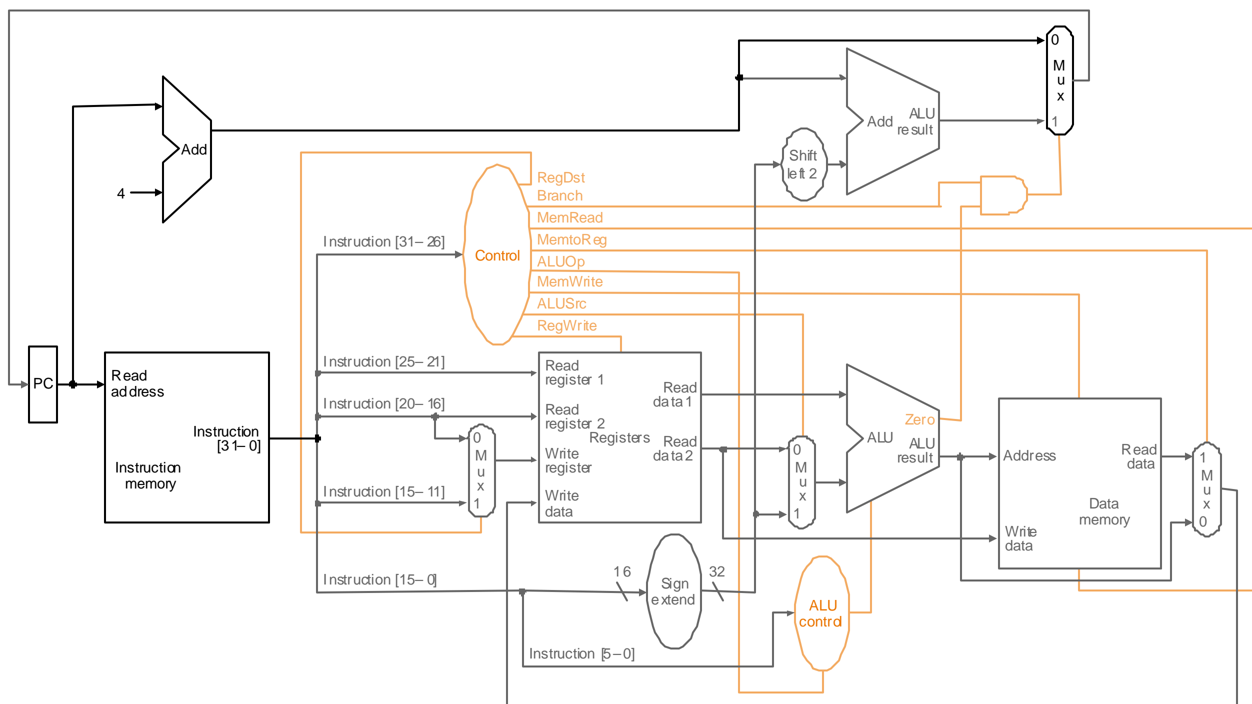


- **Sinais de controle – Tabela da figura 5.20**

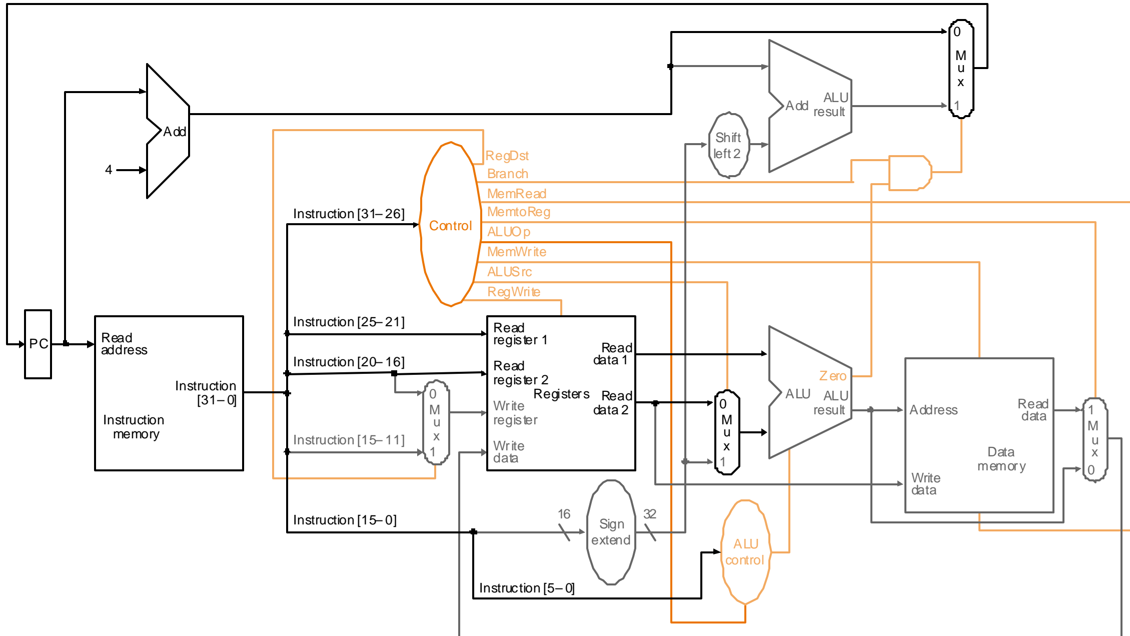
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

- **Operação do Datapath**
  - **instrução R-type ➔ add $t1,$t2,$t3**
    - **fetch da instrução**
    - **leitura de $t2 e $t3**
    - **operação da ULA com os dados lidos**
    - **resultado da ULA escrito em $t1**

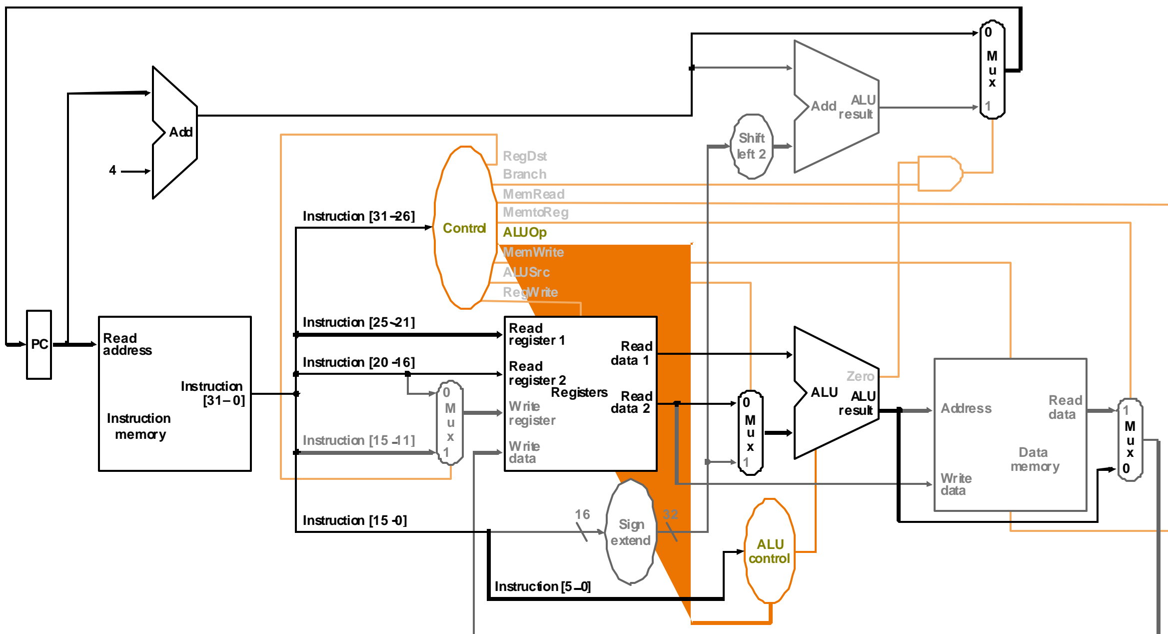
- **Figura 5.21 – Fetch e incremento de PC – instrução R-TYPE**



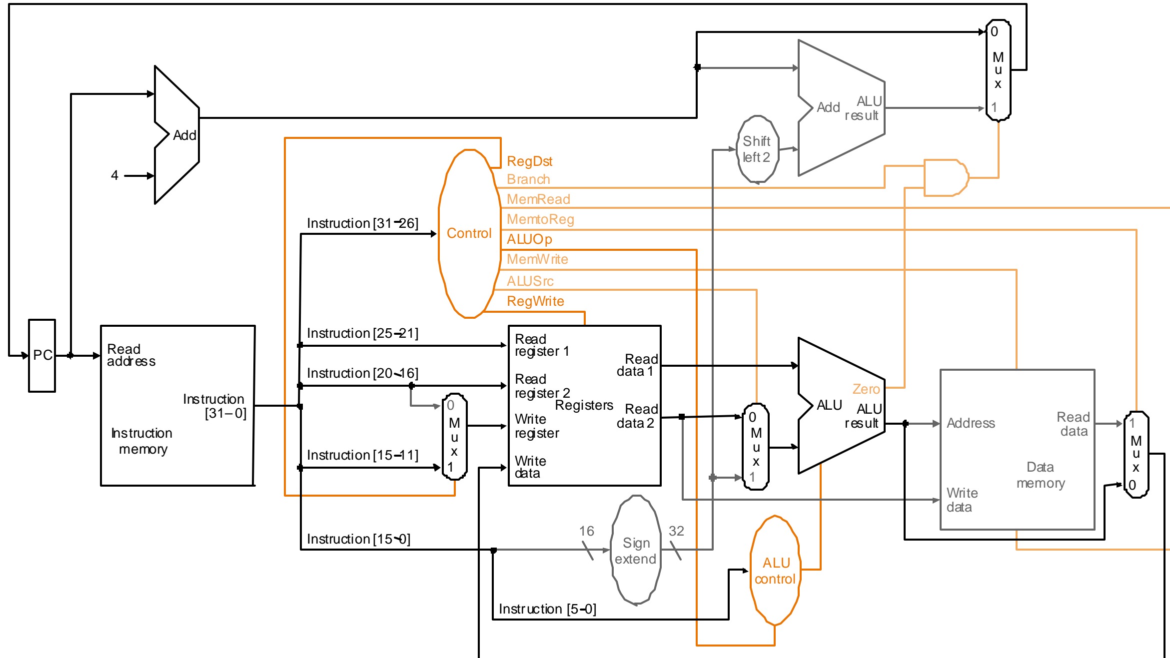
- **Figura 5.22 – Leitura dos registradores e decodificação – instrução R-TYPE**



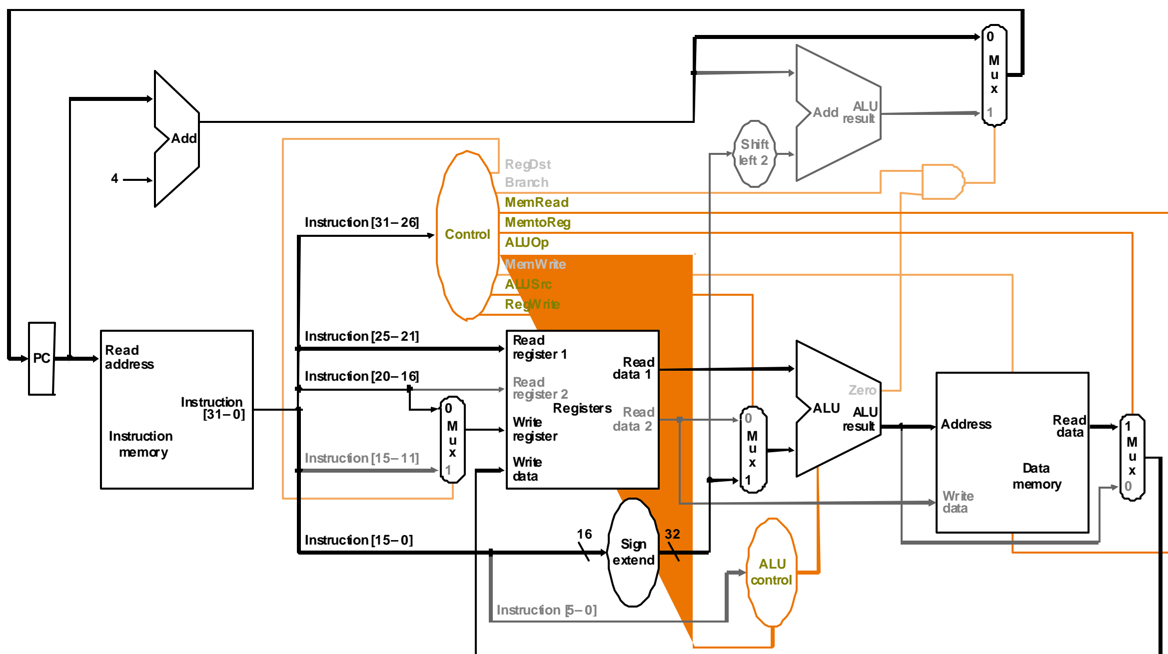
- **Figura 5.23 – operação da ULA – instrução R-TYPE**



• **Figura 5.24 – Escrita no registrador – instrução R-TYPE**

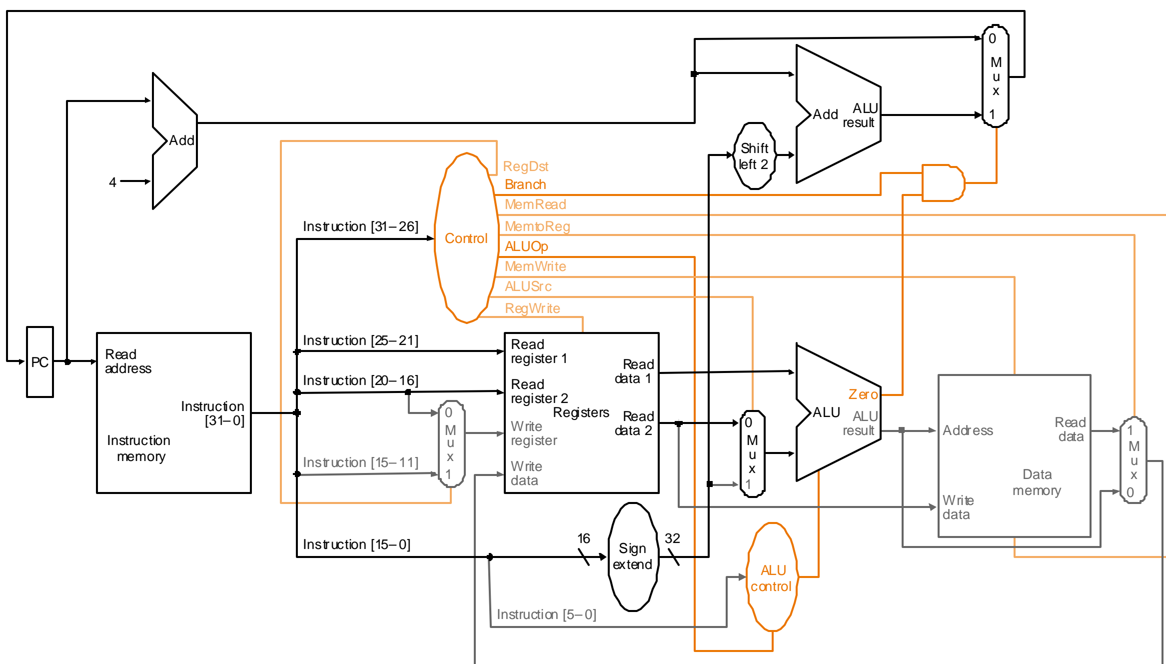


- **Instrução load word**
    - **lw $t1, offset($t2)**
        - **Instruction Fetch**
        - **$t2 é lido**
        - **ULA calcula a soma do valor lido e o imediato de 16 bits**
        - **O resultado é usado como endereço da memória de dados**
        - **O dado da memória de dados é escrito no register file**
- **Figura 5.25 – Operação de um lw com um esquema simples de controle.**



- **Instrução de branch**
    - **beq $t1,$t2,offset**
        - **Fetch da instrução**
        - **$t1 e $t2 são lidos**
        - **ULA faz subtração dos valores lidos. PC+4 é adicionado ao imediato de 16 bits, deslocado de 2 bits ➔ resultado é o endereço do desvio**
        - **A saída Zero é usada para decidir qual endereço será armazenado em PC**

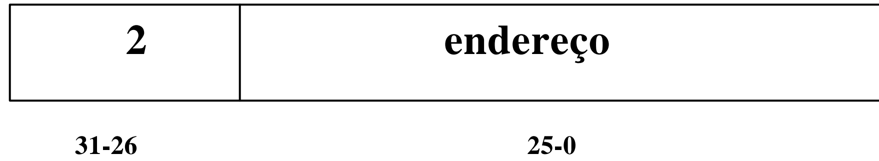
- **Figura 5.26 – Datapath para a instrução beq**



- **Tabela da figura 5.27 – Tabela verdade para as funções de controle.**

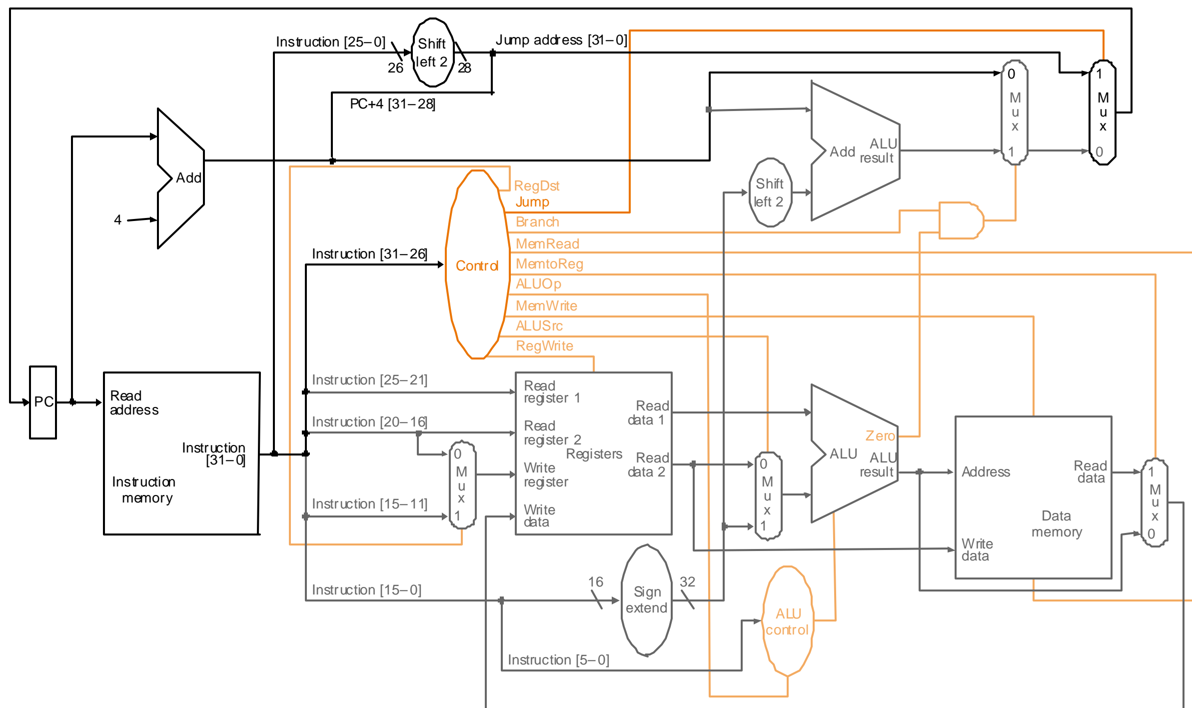
Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

- **Instrução de jump**



- **PC é formado pelos 4 bits mais significativos + 26 bits do imediato + 00 (deslocamento de 2)**

- **Figura 5.29 – Datapath para instrução de jump**



- **Implementação em um ciclo de clock → não usada**

  - **Funciona corretamente mas não é eficiente**

  - **Para single-cycle →o ciclo do clock deve ter o mesmo comprimento para todas as instruções → CPI = 1 → para isto, o ciclo de clock é determinado pelo maior caminho no datapath da máquina (instrução de load que usa 5 unidades funcionais em série: instruction memory, register file, ULA, data memory e register file)**

Exemplo.

Sejam os seguintes tempos de operação:

- **unidade de memória : 2 ns**
- **ULA e somadores: 2ns**
- **register file (read ou write) : 1 ns**
- **unidade de controle, multiplexadores, acesso ao PC, circuito para extensão do sinal e linhas não tem atraso, quais das seguintes implementações seria mais rápida e quanto ?**

  1. **Uma implementação na qual cada instrução opera em um ciclo de clock de tamanho fixo**
  2. **Uma implementação onde cada instrução é executada usando clock de tamanho de ciclo variável (exatamente do tamanho necessário para a execução da respectiva instrução → também não utilizado na prática)**

Para comparar a performance, assuma que o seguinte conjunto de instruções : 24% de loads, 12% de store, 44% instruções tipo R, 18% de branches e 2% de jumps.

Solução:

tempo de execução CPU = número de instruções X CPI X período do clock

CPI = 1 ➔ tempo de execução CPU = número de instruções X período do clock

Temos que encontrar o período do clock para as duas implementações, pois o número de instruções e a CPI são iguais para ambas implementações.

O caminho crítico para cada classe de instrução é:

Classe da Instrução	Unidades funcionais envolvidas				
R-TYPE	Inst. fetch	Reg. access	ALU	Reg. access	
LW	Inst. fetch	Reg. access	ALU	Mem access	Reg access
SW	Inst. fetch	Reg. access	ALU	Mem access	
BRANCH	Inst. fetch	Reg. access	ALU		
JUMP	Inst. fetch				

Usando o caminho crítico, podemos computar o comprimento do ciclo de clock necessário para cada classe de instrução:

Classe da Instrução	Memória Instrs.	Leitura Regs.	ALU oper.	Memória Dados	Escrita Regs.	Total
R-TYPE	2	1	2	0	1	6 ns
LW	2	1	2	2	1	8 ns
SW	2	1	2	2		7 ns
BRANCH	2	1	2			5 ns
JUMP	2					2 ns

O período do clock para a máquina com um único clock é determinado pela maior instrução → 8ns

A máquina com clock variável, terá seus períodos de clock variando entre 2ns e 8ns. O clcock médio será:

Tempo de clock da CPU = 8 X 24% + 7 X 12% + 6 X 44% + 5 X 18 % + 2 X 2% = 6.3ns

CPU perfomance$_{vc}$ / CPU perfomance$_{sc}$ =

= tempo de execução CPU$_{sc}$ / tempo de execução CPU$_{vc}$ =

= IC X período de clock da CPU$_{sc}$ / IC X período de clock da CPU$_{vc}$ =

= período de clock da CPU$_{sc}$ / período de clock da CPU$_{vc}$ =

= 8 / 6.3 = 1.27

- **Exemplo**

  - **FP unit → 8ns para add e 16ns para mul**
  - **todos os loads tem o mesmo tempo e são 33% das instruções**
  - **todos os stores tem mesmo tempo e são 21% das instruções**
  - **instruções tipo R são 27 % das instruções**
  - **Branches são 5 % e jumps 2% das instruções**
  - **FP add e sub tem o mesmo tempo e juntos tem 7% das instruções**
  - **FP mul e div tem o mesmo tempo e &5 das instruções**

Solução

CPU perfomance$_{vc}$ / CPU perfomance$_{sc}$ =
= tempo de execução CPU$_{sc}$ / tempo de execução CPU$_{vc}$

Para a máquina de único clock → período do clock = ciclo da instrução FP mul (mais longa) → 2+1+16+1 = 20ns

Para a máquina de clock variável:

período do clock CPU = 8 X 31% + 7 X 21% + 6 X 27% + 5 X 5% = 7.0 ns

Portanto:

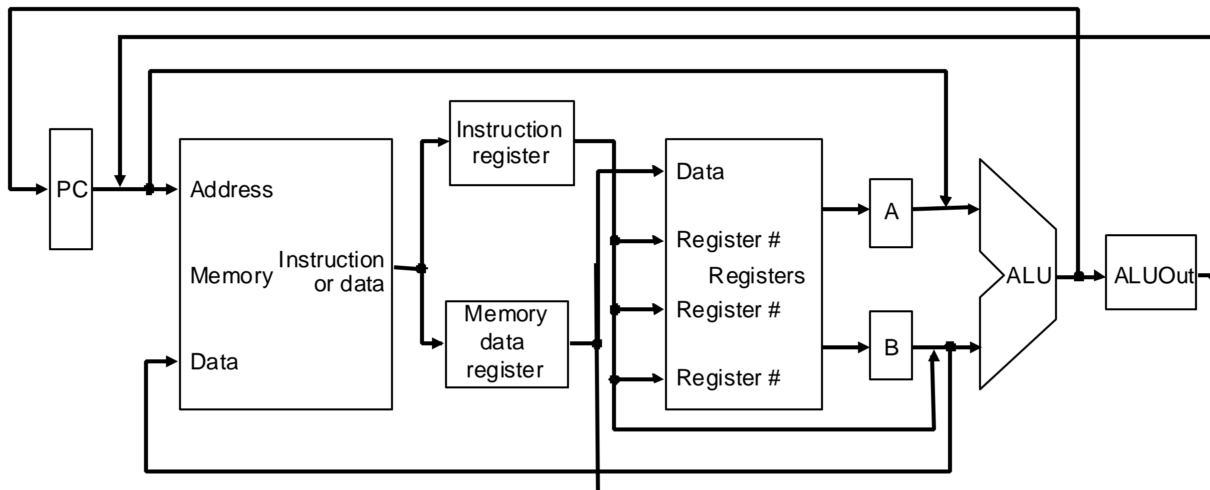
CPU perfomance$_{vc}$ / CPU perfomance$_{sc}$ =
= tempo de execução CPU$_{sc}$ / tempo de execução CPU$_{vc}$ = 20/7 = 2.9

- **Implementação Multiciclos**

  - **a execução de cada instrução é dividida em etapas, onde cada etapa corresponde a uma operação de uma unidade funcional**

  - **implementação multiciclos ➔ cada etapa é executada em um ciclo, isto permite que uma unidade funcional possa ser usada por mais de uma instrução ➔ compartilhamento pode diminuir a quantidade de HW necessária**

- **Figura 5.30 – Datapath multiciclos**



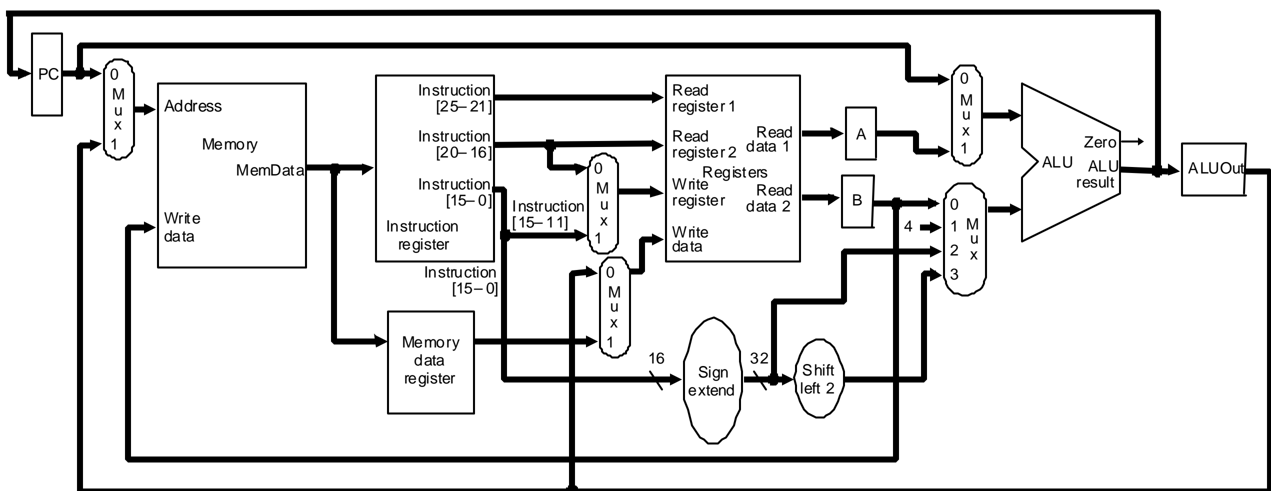
- **Diferenças com a versão single-cycle**

  - **Uma só memória usada para dados e instruções**
  - **Uma ULA em vez de uma ULA e dois somadores**
  - **Redução de área ➔ redução de custos**
  - **alguns registradores a mais para manter a saída de uma unidade funcional para outra unidade funcional, em um ciclo de clock posterior:**

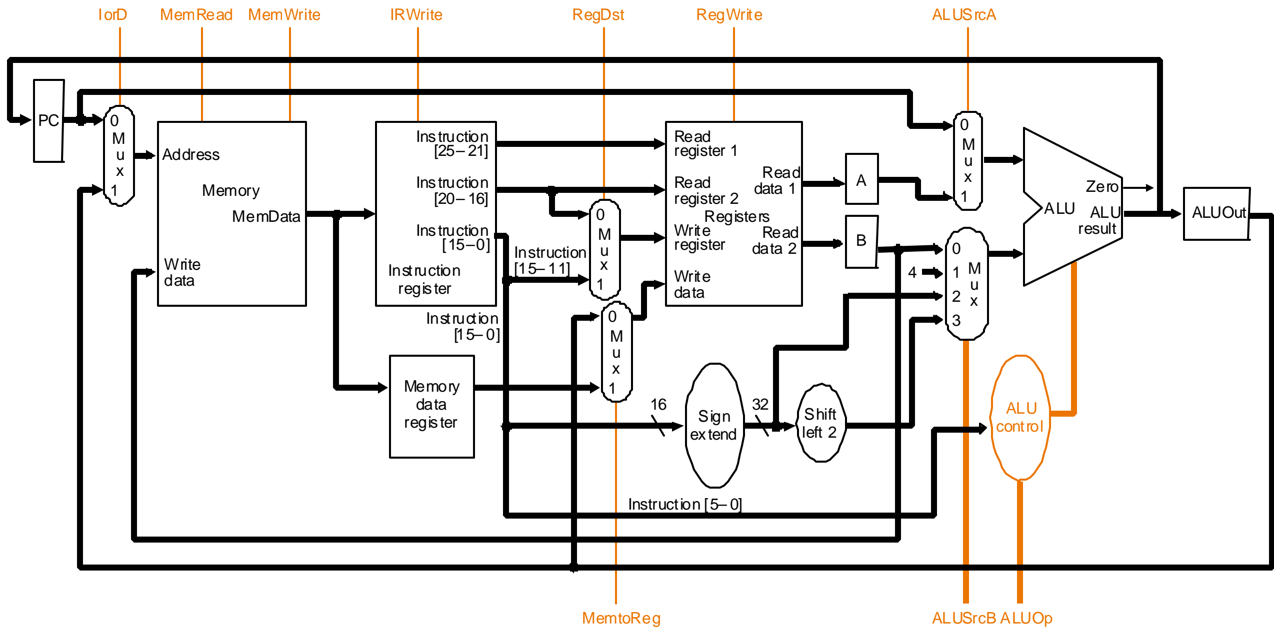
    - **Instruction Register – IR ➔ armazenar uma instrução lida da memória (a instrução deve ser mantida até o fim de sua execução)**
    - **Memory Data Register – MDR ➔ armazenar dados lidos da memória**
    - **registrador A e registrador B usados para manter os operandos lidos do register file**
    - **registrador ALUOut para manter a saída da ULA**

- **Inclusão de multiplexadores**

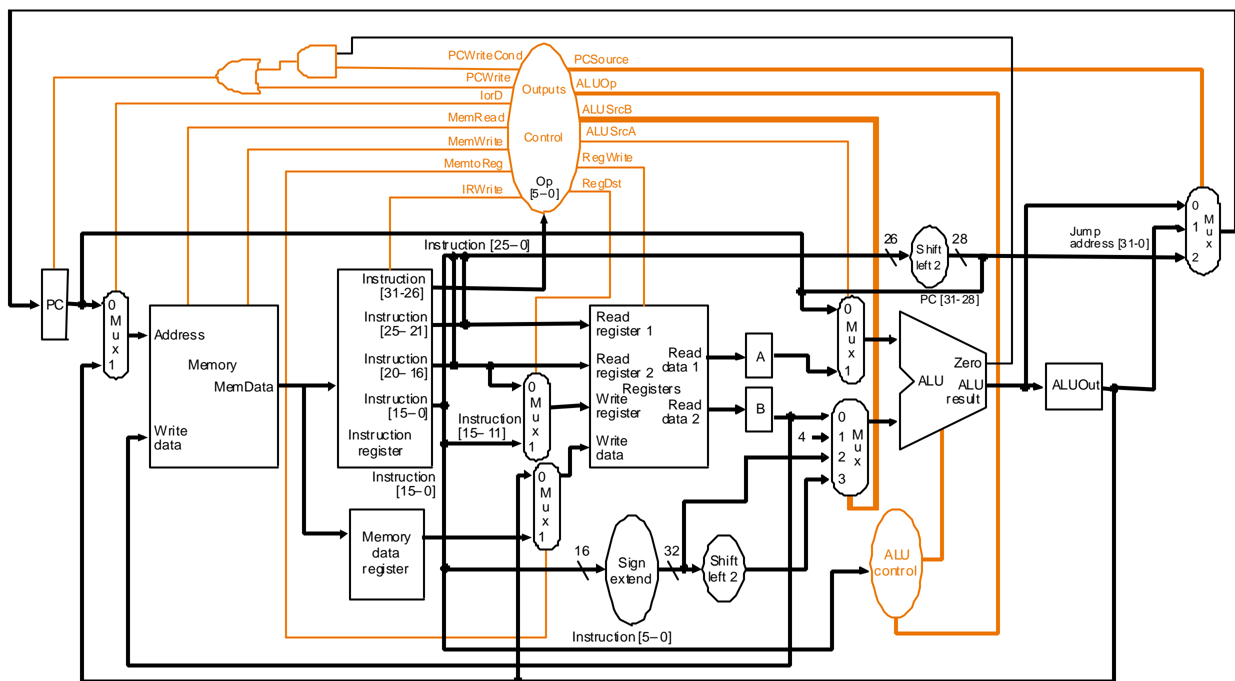
- **Figura 5.31 – Datapth multiciclos para as instruções básicas**



- **Figura 5.32 - Datapath para implementação multiciclos com sinais de controle**



- **Figura 5.33 - Datapath para implementação multiciclos com sinais de controle – completo incluindo atualização do PC**



- **Figura 5.34 tabela com Sinais de Controle**

Actions of the 1-bit control signals

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None	Memory contents at the location specified by the Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
IorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None	The output of the memory is written into the IR.
PCWrite	None	The PC is written; the source is controlled by PCSource.
PCWriteCond	None	The PC is written if the Zero output from the ALU is also active.

Actions of the 2-bit control signals

Signal name	Value	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left 2 bits.
PCSource	00	Output of the ALU (PC + 4) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address (IR[25–0] shifted left 2 bits and concatenated with PC + 4[31–28]) is sent to the PC for writing.

- **Divisão da execução de uma instrução em ciclos de clock e respectivos sinais de controle**

1. **Instruction fetch**

   O incremento do PC e leitura da da instrução ➔ em paralelo:
   IR = Memory[PC];
   PC = PC + 4;
   Sinais ativados ➔ MemRead (1), Irwrite (1), IorD (0 – PC como endereço), ALUSrcA (0 – PC para ULA), ALUSrcB (01 – 4 para a ULA), ALUOp (00 – add) e PCWrite (1 – o novo valor de PC não é visível até o próximo ciclo de clock).

2. **Instruction decode e register fetch**

   Como temos regularidade no formato das instruções, podemos, sem saber a natureza da instrução, fazer:

   - **ler dois registradores do register file, mesmo que eles não sejam utilizados e armazená-los nos registradores A e B;**
   - **computar os endereços de branch e guardá-los e ALUOut, mesmo que a instrução não venha a ser um branch.**

   A = Reg[IR[25-21]];
   B = Reg[IR[20-16]];
   ALUOut = PC + (sign-extend (IR[15-0] << 2)

   Sinais ativados: ALUSrcA (0 – PC vai para a ULA), ALUSrcB (11- sign extended e shifted enviado a UAL) e ALUOp (00 – add).

3. **Execution, memory address computation ou branch completion**

   Primeira etapa determinada pelo tipo de instrução. Para cada classe temos:

   - **Referência à memória:**

     ALUOut = A + sign-extend (IR[15-0]);

     Sinais ativados: ALUSrcA (1 - A para ULA), ALUSrcB (10 – saida da unidade sign-extension para a ULA) e ALUOp (00 - add).

   - **Instruções Aritméticas-lógicas (R-type)**

     ALUOut = A op B;

     Sinais ativados: ALUSrcA (1 - A para ULA), ALUSrcB (00 – B para a ULA) e ALUOp (10 – o campo funct é usado para determinar os sianis de controle da ULA).

   - **Branch**

     if (A == B) PC = ALUOut;

     Sinais ativados: ALUSrcA (1 - A para ULA), ALUSrcB (00 – B para a ULA) e ALUOp (01 – sub para teste de igualdade), PCCondWrite (1 se Zero=1) e PCSource (01 – PC recebe dado vindo de ALUOut).

- **Jump**

  PC = PC[31-28] || (IR[25-0]<<2)

  Sinais ativados: PCSource(10- jump address para PC) e PCWrite (1).

4. **Memory access or R-type instruction completion**

   Nesta etapa, uma instrução de load ou store acessa a memória ou uma instrução R-type escreve seu resultado.

- **Referência à memória**

  MDR = Memory [ALUOut]; - load

  ou

  Memory [ALUOut] = B; - store

  Sinais ativados: MemRead (1 - para load) ou MemWrite (1 - para store), IorD (1 para load, para que o endereço de memória venha da ULA).

- **instruções R-type**

  Reg[IR[15-11]] = ALUOut;

  Sinais ativados: RegDst (1 – campo rd (15-11) usado como entrada do register file para escrita), RegWrite (1) e Memto Reg (0 – para saida da ULA ser escrita).

5. **Memory read completion**

- **Load**

  Reg[IR[20-16]] = MDR;

  Sinais ativados: MemtoReg (1 – para escrver o resultado da memória), RegWrite (1- escrita no register file) e RegDst (0 – para escolher rt (20-16) como número do registrador).

- **Figura 5.35 - Resumo das etapas de execução para as diversas classes**

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR = Memory[PC] PC = PC + 4			
Instruction decode/register fetch	A = Reg [IR[25–21]] B = Reg [IR[20–16]] ALUOut = PC + (sign-extend (IR[15–0]) << 2)			
Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15–0])	if (A == B) then PC = ALUOut	PC = PC [31–28] II (IR[25–0]<<2)
Memory access or R-type completion	Reg [IR[15–11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20–16]] = MDR		

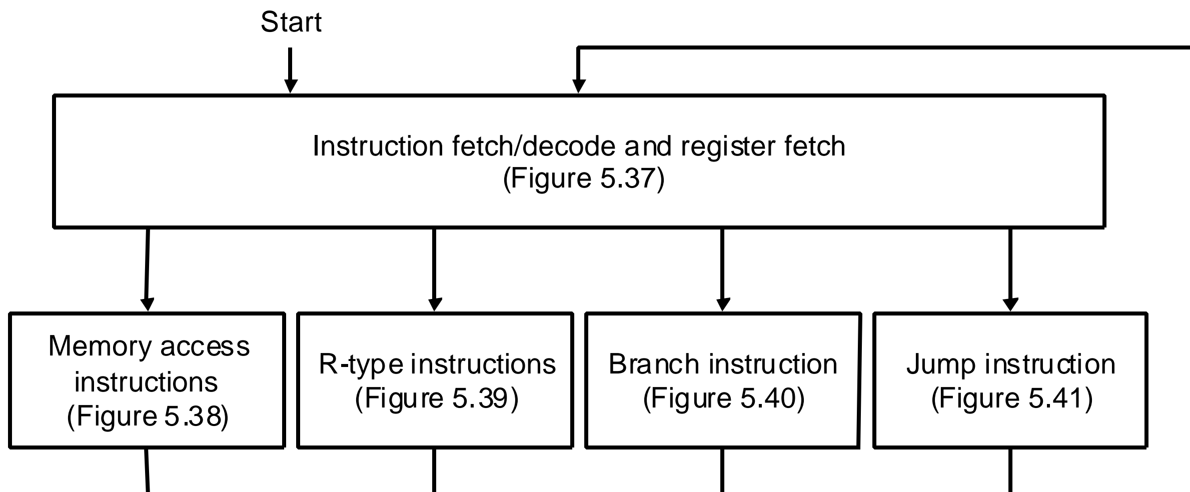
- **Projeto da Unidade de Controle**

  - **Máquinas de estados finitos → conjunto de estados e como estes estados podem mudar (esta mudança → próximo estado).**

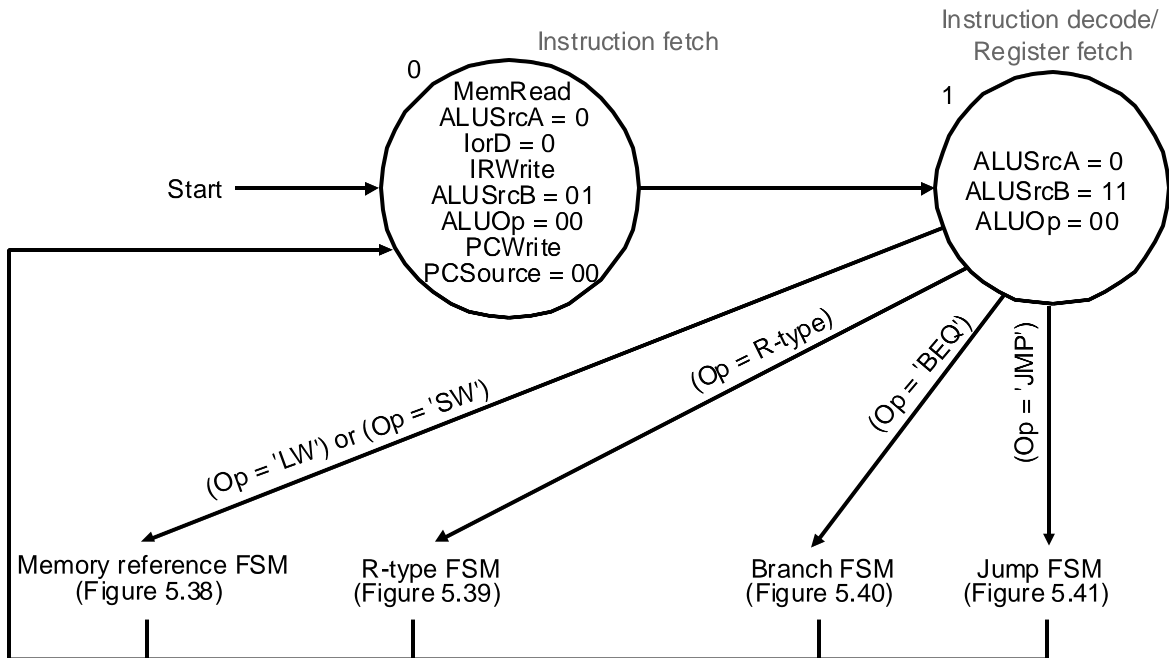
  - **Controle de estados finitos corresponde às cinco etapas de execução de uma instrução, onde cada etapa é executada em um ciclo de clock.**

  - **Unidade de Controle implementada como máquina de estados finitos:**

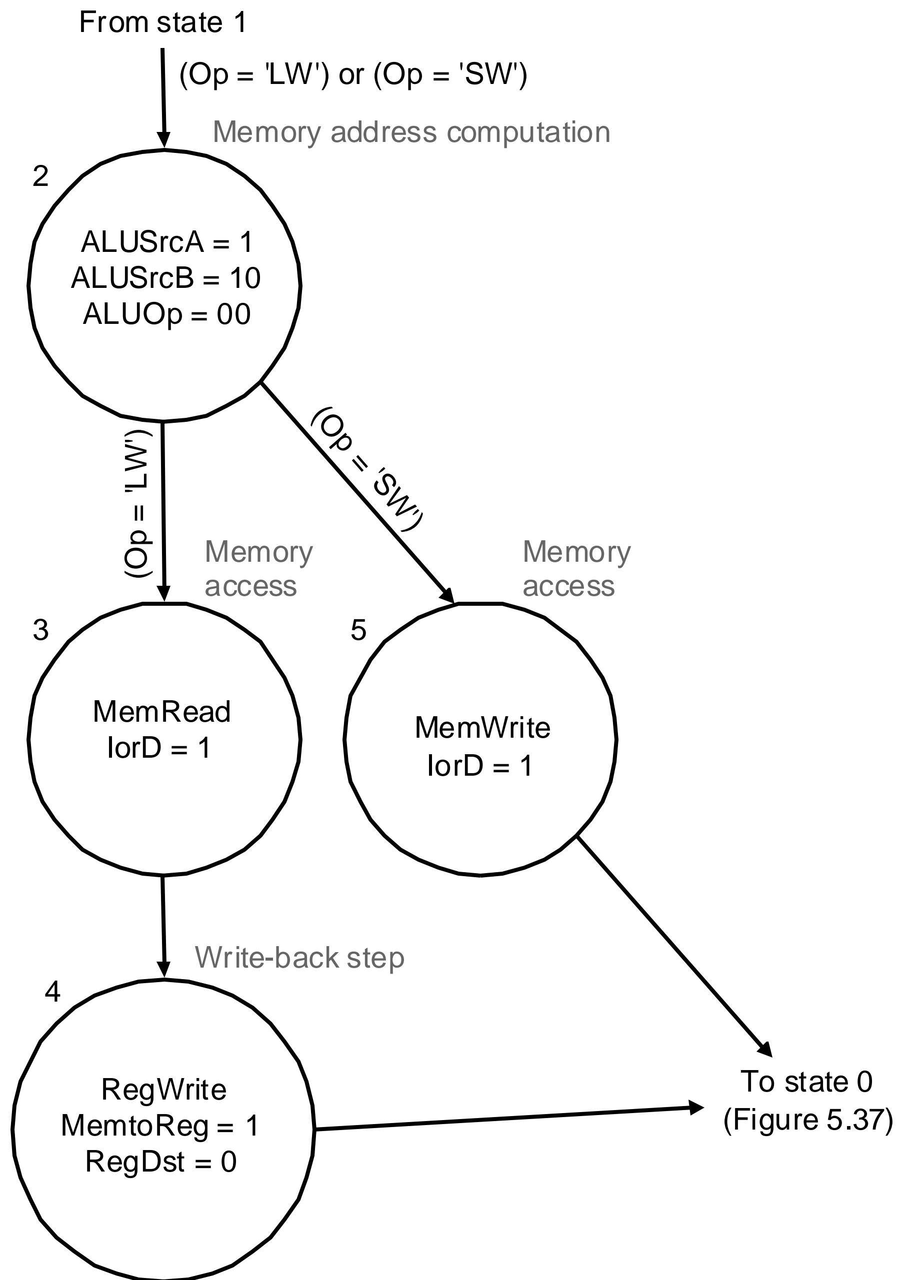
- **Figura 5.36 – Unidade de Controle – Máquinas de Estados Finitos**



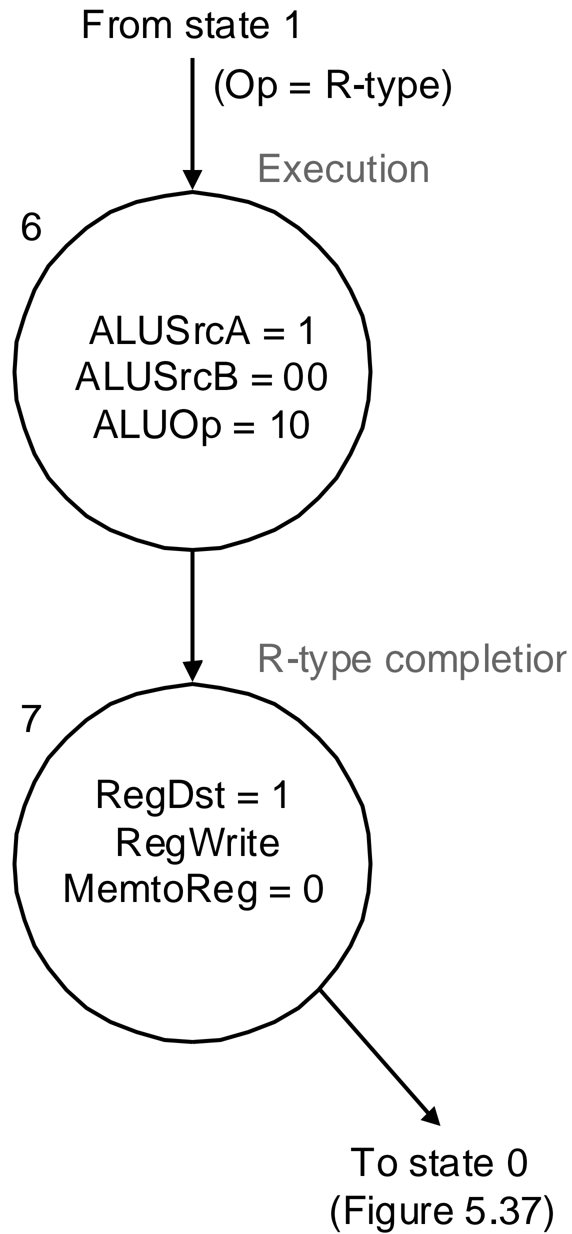
- **Figura 5.37 – Diagrama de estados - Instruction fetch e instruction decode/register fetch**



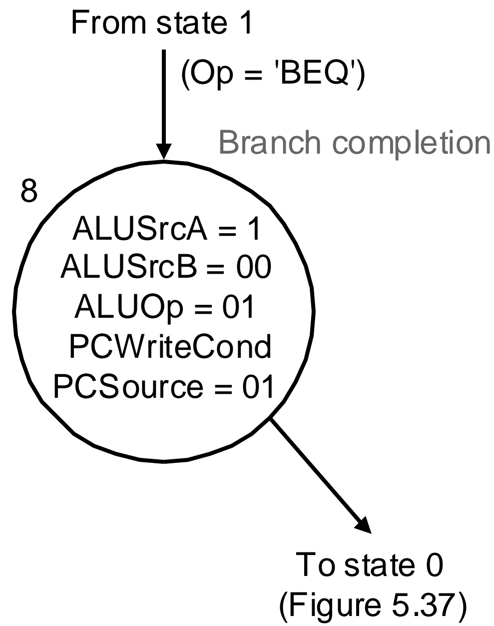
- **Figura 5.38 – Diagrama de estados - Memory reference instructions**



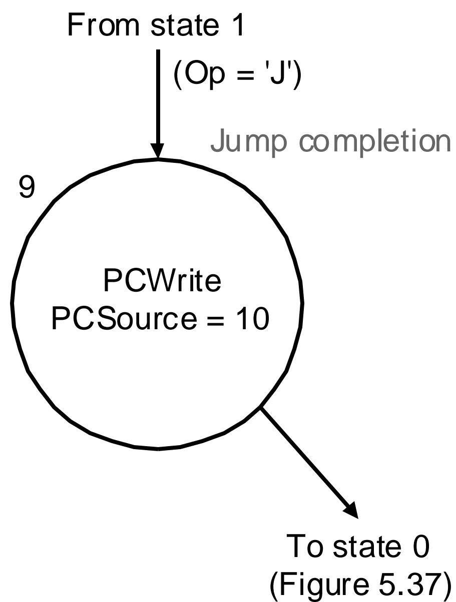
- **Figura 5.39 – Diagrama de estados - R-type instructions**



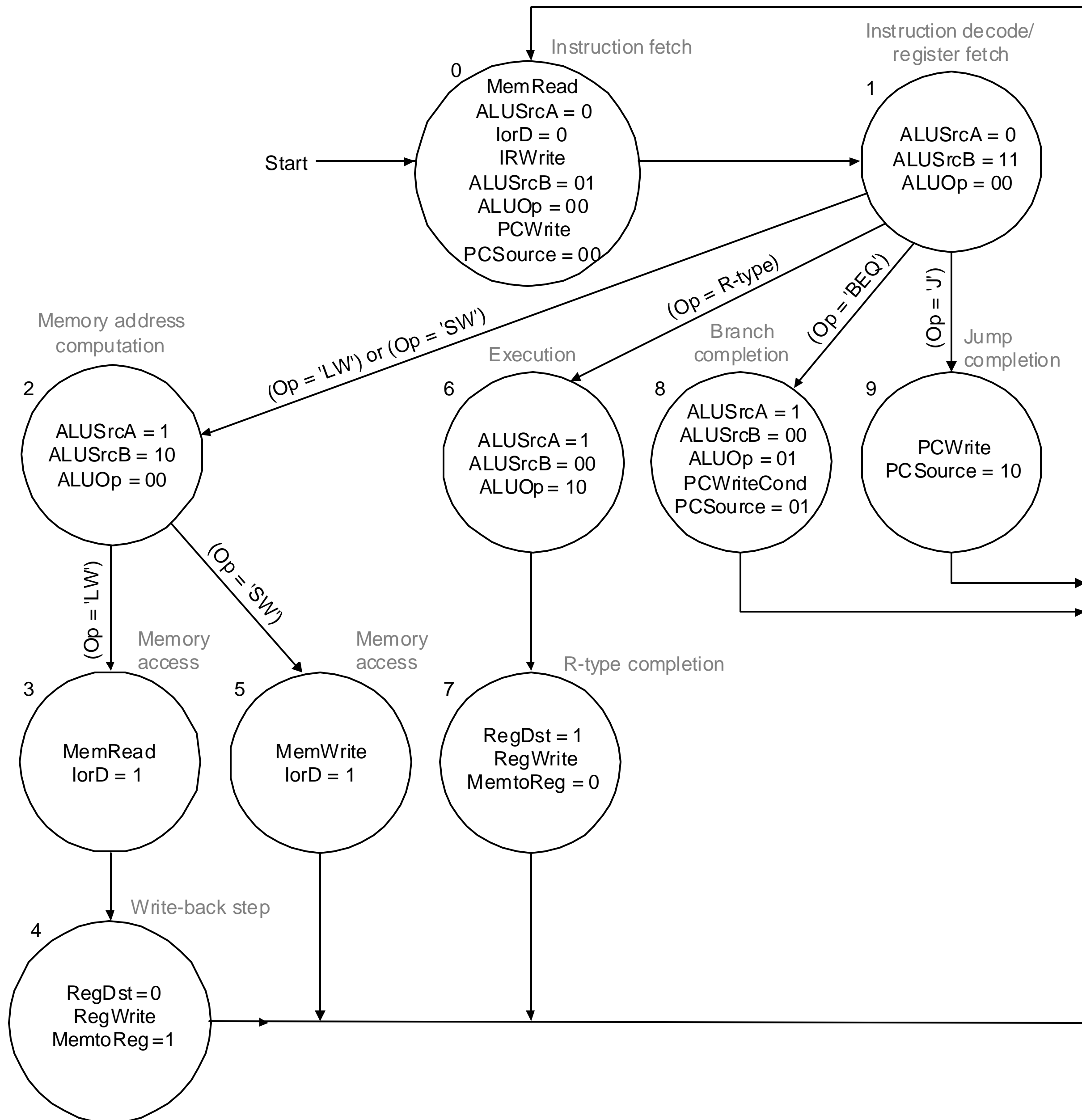
- **Figura 5.40 – Diagrama de estados - Branch instructions**



- **Figura 5.41 – Diagrama de estados - Jump instructions**



- **Figura 5.42 – Diagrama de estados - Figura Unidade de Controle completa**



- **CPI em uma CPU multicycles**

Exemplo:

Usando a figura anterior o conjunto de instruções abaixo, qual a CPI, assumindo que cada estado necessita de 1 clico de clock ?

conjunto de instruções: 22% loads, 11% de stores, 49% de R-type, 16% de branches e 2% de jumps (gcc).

Solução:

O número de ciclos de clocks para cada classe é (da figura anteiror):

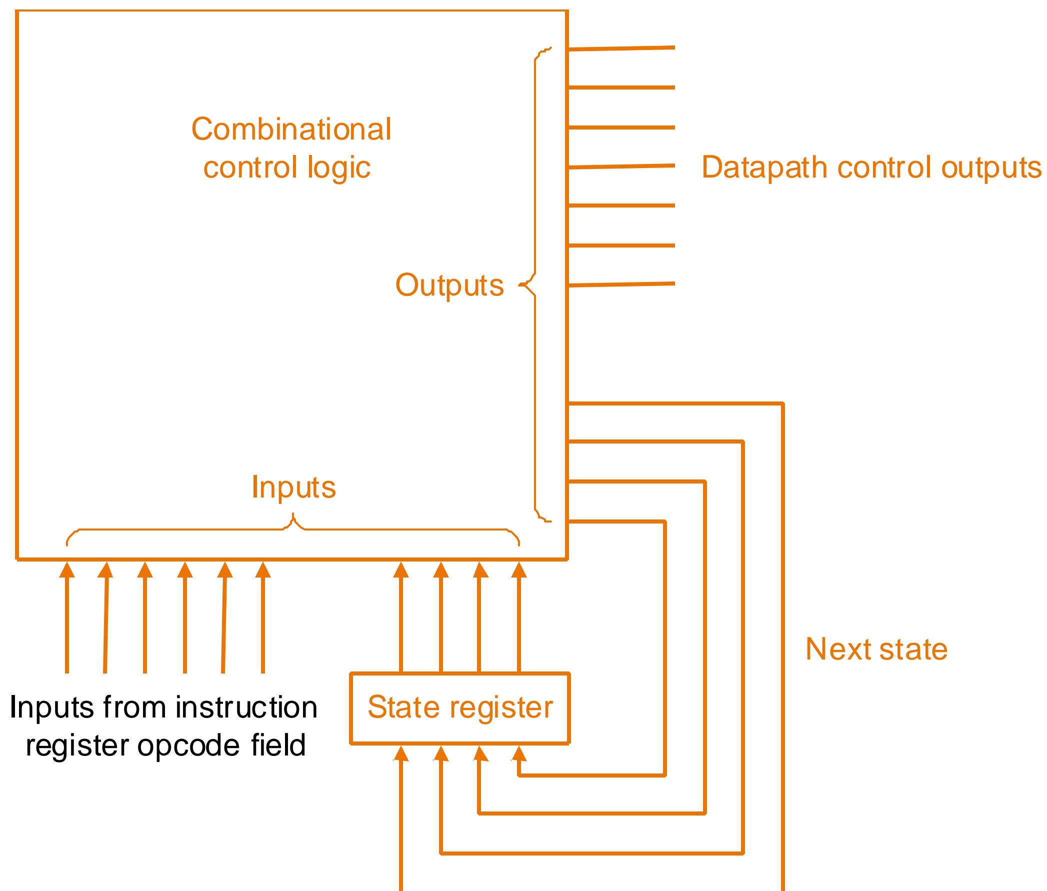
- **loads: 5**
- **stores: 4**
- **R-type : 4**
- **branches : 3**
- **jumps: 3**

CPI = ciclos de clock da CPU / número de instruções ➔
CPI = Σ (num. de instruções$_i$ X CPI$_i$) / num. de instruções
CPI = Σ ((num. de instruções$_i$/ num. de instruções) X CPI$_i$)

CPI = 0.22 X 5 + 0.11 X 4 + 0.49 X 4 + 0.16 X 3 + 0.02 X 3
CPI = 4.04

4.04 é melhor que a CPI de uma CPU em que todas as instruções tivessem o mesmo número de ciclos de clock (CPI pior caso = 5).

- **Figura 5.43 - Implementação de uma Unidade de Controle com máquina de estados finitos**



- **Microprogramação – Projeto simplificado da Unidade de Controle**

  - **Representação gráfica útil para pequenas máquinas de estados.**

  - **Cada microinstrução tem o efeito de ativar os sinais de controle especificados por elas**

  - **Um microprograma é uma representação simbólica da unidade de controle que será traduzida por um programa para a lógica de controle.**

  - **O formato da microinstrução deve ser escolhida de tal forma a simplificar sua representação.**

  - **Para evitar microinstruções inconsistentes, cada campo da microinstrução deve especificar um conjunto de sinais que não acontecem simultaneamente.**

- **Figura 5.44 – Tabela com os sete campos da microinstrução**

Field name	Function of field
ALU control	Specify the operation being done by the ALU during this clock; the result is always written in ALUOut.
SRC1	Specify the source for the first ALU operand.
SRC2	Specify the source for the second ALU operand.
Register control	Specify read or write for the register file, and the source of the value for a write.
Memory	Specify read or write, and the source for the memory. For a read, specify the destination register.
PCWrite control	Specify the writing of the PC.
Sequencing	Specify how to choose the next microinstruction to be executed.

- **As microinstruções são colocadas em uma ROM ou PLA.**

- **O endereçamento da microinstrução é dado seqüencialmente.**

- **Existem 3 modos diferentes para escolher a próxima microinstrução:**

1. **Incrementar o endereço da microinstrução corrente para obter ao endereço da próxima microinstrução. Este comportamento é indicado, colocando Seq no campo de Seqüência.**

2. **Desviar para a microinstrução que inicia a execução da próxima instrução (estado 0 - fetch). Este comportamento é indicado colocando Fetch no campo de Seqüência.**

3. **Escolher a próxima microinstrução baseada em entradas da unidade de controle. Este comportamento é chamado dispatch e é usualmente implementado criando uma tabela (implementada em ROM ou PLA) contendo os endereços das microinstruções alvo. Geralmente existe muitas tabelas, neste caso temos duas a do estado 1 e a do estado 2. Isto é indicado no campo Seqüência por Dispatch i, onde i é o número da tabela.**

- **Figura 5.45 – Tabela com os valores dos campos de uma microinstrução**

Field name	Values for field	Function of field with specific value
Label	Any string	Used to specify labels to control microcode sequencing. Labels that end in a 1 or 2 are used for dispatching with a jump table that is indexed based on the opcode. Other labels are used as direct targets in the microinstruction sequencing. Labels do not generate control signals directly but are used to define the contents of dispatch tables and generate control for the Sequencing field.
ALU control	Add	Cause the ALU to add.
	Subt	Cause the ALU to subtract; this implements the compare for branches.
	Func code	Use the instruction's funct field to determine ALU control.
SRC1	PC	Use the PC as the first ALU input.
	A	Register A is the first ALU input.
SRC2	B	Register B is the second ALU input.
	4	Use 4 for the second ALU input.
	Extend	Use output of the sign extension unit as the second ALU input.
	Extshft	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read	Read two registers using the rs and rt fields of the IR as the register numbers, putting the data into registers A and B.
	Write ALU	Write the register file using the rd field of the IR as the register number and the contents of ALUOut as the data.
	Write MDR	Write the register file using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	Read PC	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	Read memory using ALUOut as address; write result into MDR.
	Write ALU	Write memory using the ALUOut as address; contents of B as the data.
PCWrite control	ALU	Write the output of the ALU into the PC.
	ALUOut-cond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	Jump address	Write the PC with the jump address from the instruction.
Sequencing	Seq	Choose the next microinstruction sequentially.
	Fetch	Go to the first microinstruction to begin a new instruction.
	Dispatch i	Dispatch using the ROM specified by i (1 or 2).

- **Microprograma**
  - **Fetch e decode**

Primeira tabela página 404

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1

- **Primeira microinstrução**

Segunda tabela página 404

Fields	Effect
ALU control, SRC1, SRC2	Compute PC + 4. (The value is also written into ALUOut, though it will never be read from there.)
Memory	Fetch instruction into IR.
PCWrite control	Causes the output of the ALU to be written into the PC.
Sequencing	Go to the next microinstruction.

- **Segunda microinstrução**

Terceira tabela página 404

Fields	Effect
ALU control, SRC1, SRC2	Store PC + sign extension (IR[15–0]) << 2 into ALUOut.
Register control	Use the rs and rt fields to read the registers placing the data in A and B.
Sequencing	Use dispatch table 1 to choose the next microinstruction address.

- **Dispatch table 1 é usada para selecionar uma das 4 seqüências de microinstruções:**

  - **Mem1 → instruções de referência à memória**
  - **Rformat1 → instruções R-type**
  - **BEQ1 → instruções beq**
  - **JUMP1 → instruções de jump**

- **Microprograma para Mem1**

Tabelas da página 405 e primeira tabela da página 406

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch

Fields	Effect
ALU control, SRC1, SRC2	Compute the memory address: Register (rs) + sign-extend (IR[15–0]), writing the result into ALUOut.
Sequencing	Use the second dispatch table to jump to the microinstruction labeled either LW2 or SW2.

Fields	Effect
Memory	Read memory using the ALUOut as the address and writing the data into the MDR.
Sequencing	Go to the next microinstruction.

Fields	Effect
Register control	Write the contents of the MDR into the register file entry specified by rt.
Sequencing	Go to the microinstruction labeled Fetch.

Fields	Effect
Memory	Write memory using contents of ALUOut as the address and the contents of B as the value.
Sequencing	Go to the microinstruction labeled Fetch.

- **Microprograma para Rformat1**

As três últimas tabelas da página 406

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch

Fields	Effect
ALU control, SRC1, SRC2	The ALU operates on the contents of the A and B registers, using the function field to specify the ALU operation.
Sequencing	Go to the next microinstruction.

Fields	Effect
Register control	The value in ALUOut is written into the register file entry specified by the rd field.
Sequencing	Go to the microinstruction labeled Fetch.

- **Microprograma para BEQ1**

As duas primeiras tabelas da página 407

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
BEQ1	Subt	A	B			ALUOut-cond	Fetch

Fields	Effect
ALU control, SRC1, SRC2	The ALU subtracts the operands in A and B to generate the Zero output.
PCWrite control	Causes the PC to be written using the value already in ALUOut, if the Zero output of the ALU is true.
Sequencing	Go to the microinstruction labeled Fetch.

- **Microprograma para JUMP1**

As duas últimas tabelas da página 407

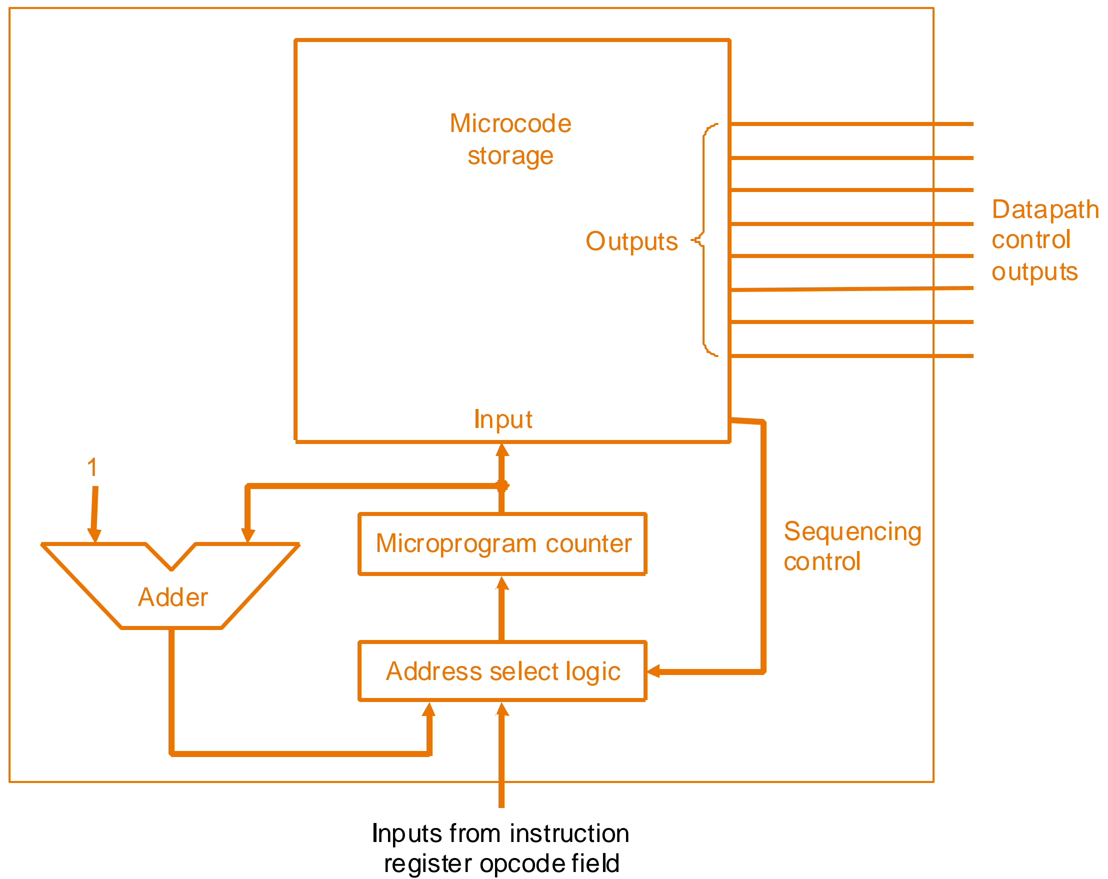
Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
JUMP1						Jump address	Fetch

Fields	Effect
PCWrite control	Causes the PC to be written using the jump target address.
Sequencing	Go to the microinstruction labeled Fetch.

- **Figura 5.46 - Microprograma para unidade de controle**

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

- **Figura 5.47 - Implementação, em ROM, da Unidade de controle microprogramada.**



- **Exceções e interrupções**

  - **Exceção é um evento inesperado, interno ao processador (p. ex. overflow), que causa uma mudança no fluxo normal de execução das instruções.**

  - **Interrupção é um evento inesperado, externo ao processador (p. ex. interrupção de I/O), que causa uma mudança no fluxo normal de execução das instruções.**

Tabela da página 411

Tipo de Evento	Fonte	Terminologia MIPS
Requisição de I/O	Externa	Interrupção
Chamada	Interna	Exceção
Overflow aritimético	Interna	Exceção
Uso de instrução não definida	Interna	Exceção
Malfuncionamento do hardware	Ambos	Exceção ou Interrupção

- **Detecção de exceção**

  - **Dois tipos de exceção serão tratados: execução de uma instrução não definida e overflow aritmético.**

  - **Quando ocorre uma exceção, o endereço da instrução afetada é guardada no EPC (exception program counter) e o controle transferido ao sistema operacional em um endereço especificado. Dependendo da atitude do SO e do tipo de exceção, o programa pode ser interrompido ou reiniciado a partir do endereço armazenado em EPC.**

  - **Para o SO tratar a exceção, ele tem que conhecer qual é a exceção. Há duas maneiras:**

    - **status register (Cause), que tem um campo que especifica a exceção e**
    - **vectored interrupts ➔ o SO sabe a razão da exceção pelo endereço de entrada no vetor. O SO sabe a razão da exceção pelo endereço passado para ele.**

Tabela da página 412

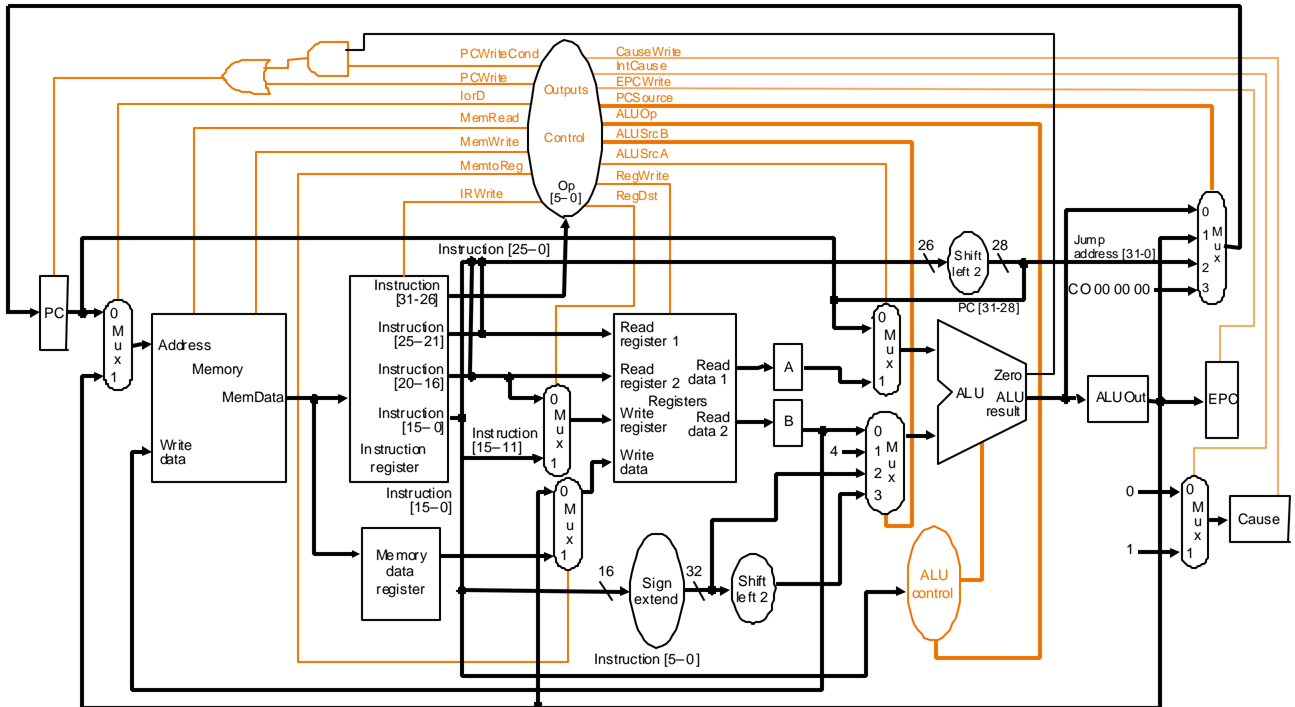
Tipo de exceção	**Endereço no vetor de exçeções(hexa)**
Intrução não definida	**C0 00 00 00_{hex}**
Overflow aritimético	**C0 00 00 20_{hex}**

- **Para tratar a exceção vamos supor o primeiro método.**

  - **Temos 2 registradores extra: EPC de 32 bits, usado para manter o endereço da instrução afetada e Cause (32 bits), usado para armazenar a causa da exceção. Vamos assumir que o bit menos significativo seja usado para codificar a causa da exceção (instrução indefinida = 0 e overflow = 1)**
  - **Precisamos de dois sinais de controle EPCWrite e CauseWrite, o sinal do bit menos significativo de Cause – IntCause. Precisamos também escrever o endereço de exceção, no qual o SO entra para tratar a mesma (supor $C0000000_{16}$).**
  - **O multiplexador de entrada de PC dever ser alterado de 3 para 4 entradas (entrada $C0000000_{16}$).**

  - **A ULA deve subtrair 4 do valor de PC para poder guardar em EPC.**

- **Figura 5.48 - Datapath que trata exceções**

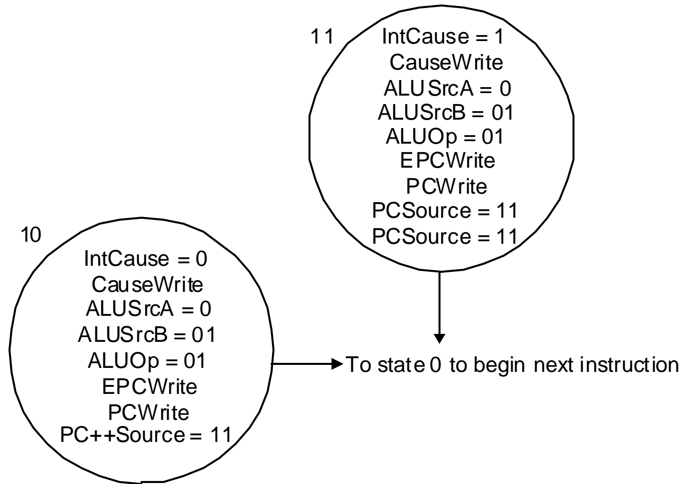


- **Para detectar as exceções e transferir o controle para o estado apropriado temos:**

  - **Instrução indefinida: é detectada quando o próximo estado é definido do estado 1 para o valor op. Tratamos definindo o próximo estado, para qualquer valor de op, como estado 10.**

  - **Overflow aritmético: o sinal de overflow (saída da ULA) é usado para modificar a máquina de estados para especificar um próximo estado para o estado 7.**

- **Figura 5.49 – Estados para tratar exceções.**



- **Figura 5.50 - Unidade de controle modificada para tratar exceção**