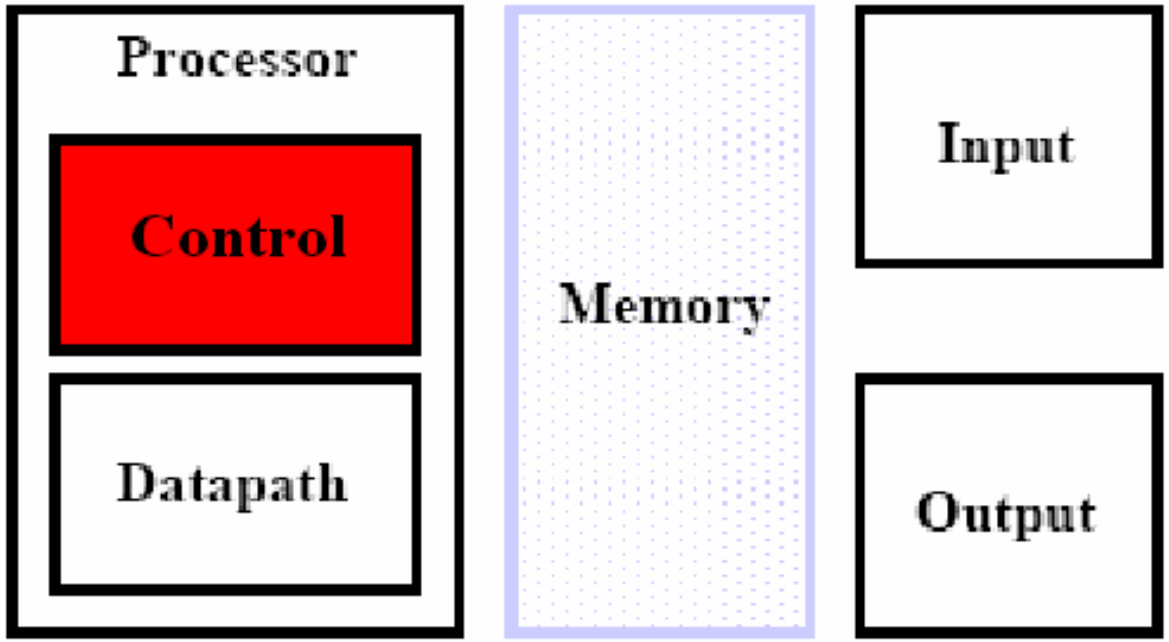
O Funcionamento do Processador

CES-25 – Arquiteturas para Alto Desmpenho
Prof. Paulo André Castro
pauloac@ita.br
Sala 110 – Prédio da Computação
www.comp.ita.br/~pauloac
IEC - ITA

Os cinco componentes clássicos de um Computador - Controle



O Processador em Funcionamento: Executando Instruções

- Em linhas gerais, a **execução de uma instrução** pode ser dividida nas seguintes fases:
  1. Recuperação do código da operação
  2. Decodificação do código da operação
  3. Recuperação dos operandos
  4. Execução propriamente dita
  5. Armazenamento dos resultados
- As fases que envolvem **acesso à memória** podem ser mais de **dez vezes mais lentas** que as demais fases

Tipos de Processadores segundo posição dos operandos e resultado

- Processadores com Acumulador: Operações envolvem um registrador especial e [em alguns casos] a memória.

- Processadores com Registradores de Propósito Geral: Os operandos podem estar em vários registradores ou mesmo na memória.

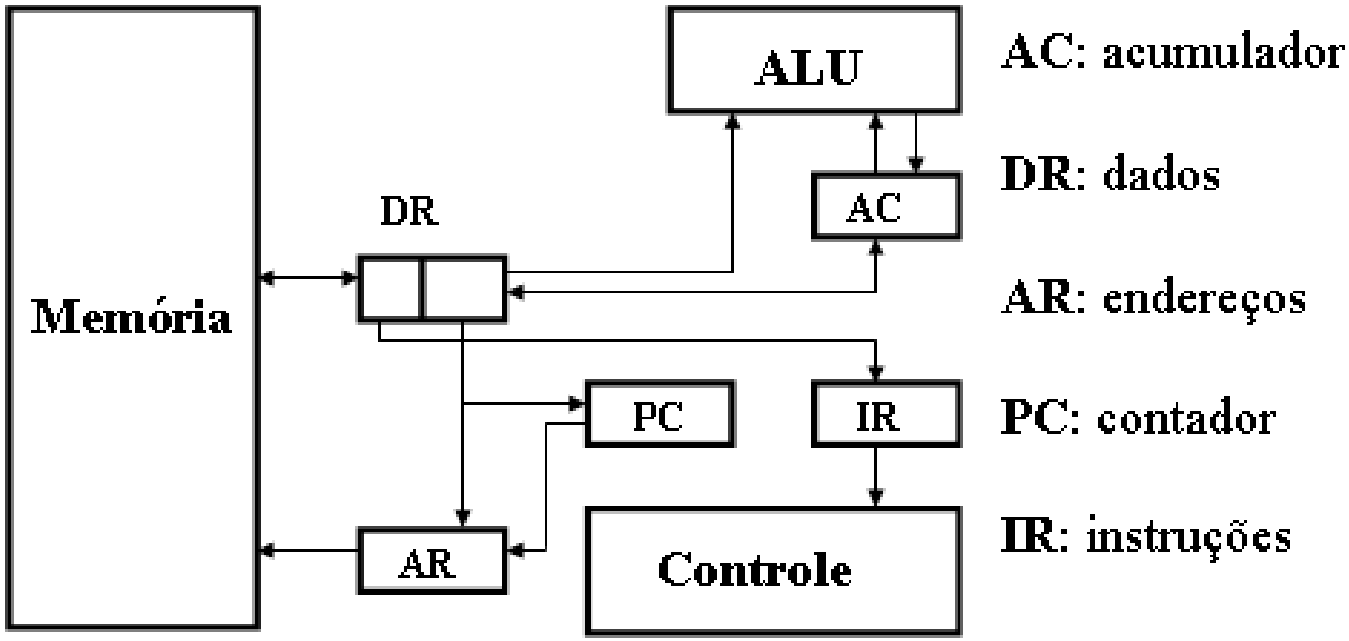
- Processadores de Pilha: instruções e operandos ficam armazenados em uma estrutura de dados do tipo Pilha na memória.

Processadores com Acumulador

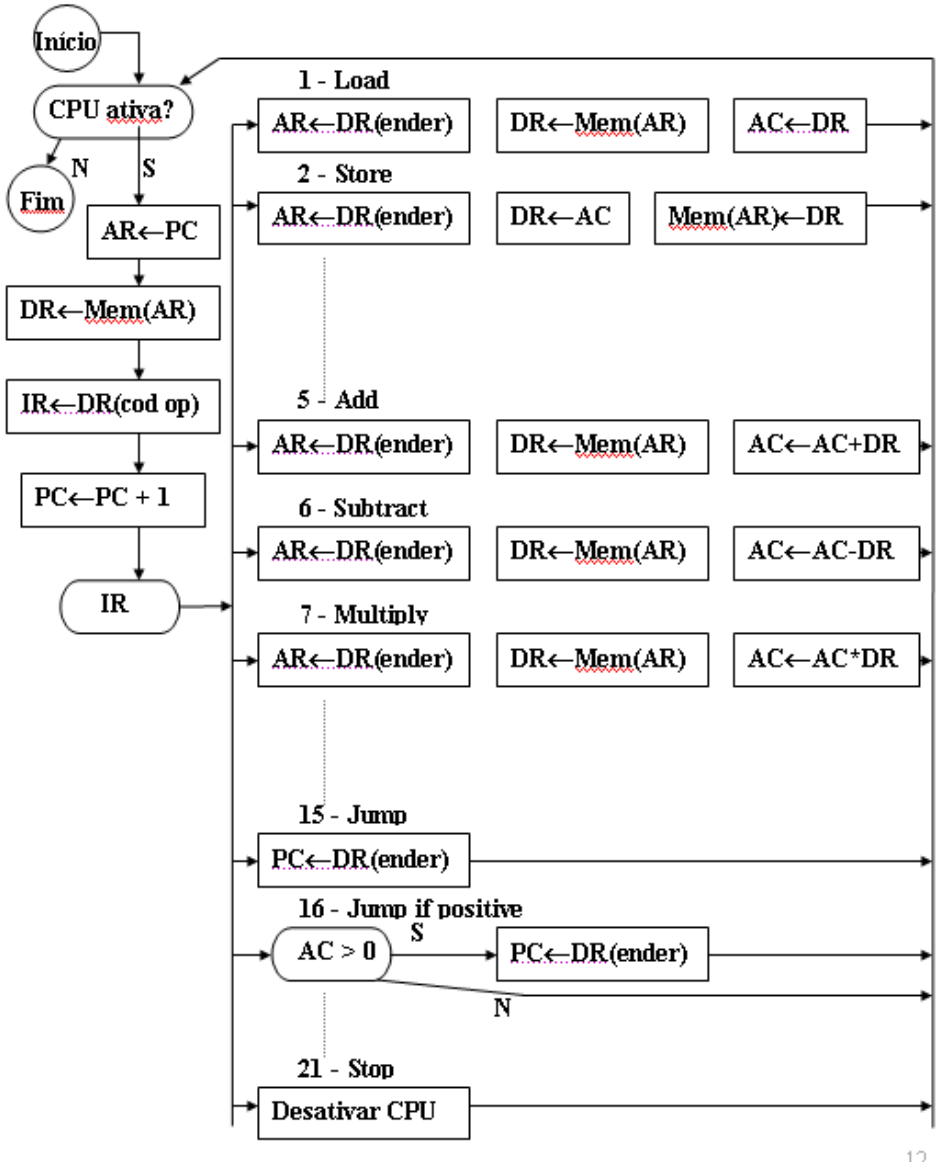
- Exemplo de Funcionamento de um Processador com Acumulador

- Comparação simples versus Processador com Registradores de Propósito Geral

Processador com Acumulador: Um Caso Simples



Microprograma da CPU com Acumulador



Exemplo de programa em Processador com Acumulador

- Pseudo-código:
  - T1=F+G
  - T1=(H-I)*T1
  - T2=E*F
  - X=A+B
  - X=((C-D)*X-T2)/T1
- Equivalente a:

$$X = \frac{(C-D)*(A+B)-(E*F)}{(H-I)*(F+G)}$$

- Como fazer o código assembly correspondente em um Processador com Acumulador ?

Pergunta...

- Considerando a seguinte divisão de fases:
    1. Recuperação do código da operação
    2. Decodificação do código da operação
    3. Recuperação dos operandos
    4. Execução propriamente dita
    5. Armazenamento dos resultados

- Há algum problema intrínseco de desempenho na arquitetura com acumulador? Qual?

Acumulador x Registradores de Propósitos Gerais

Um acumulador		Vários registradores	
Ld F	Ld A	Ld R1, F	Sub R4, D
Add G	Add B	Add R1, G	Mult R3, R4
Sto T1	Sto X	Ld R2, H	Sub R3, R2
Ld H	Ld C	Sub R2, I	Div R3, R1
Sub I	Sub D	Mult R1, R2	Sto X, R3
Mult T1	Mult X	Ld R2, E	
Sto T1	Sub T2	Mult R2, F	
Ld E	Div T1	Ld R3, A	
Mult F	Sto X	Add R3, B	
Sto T2		Ld R4, C	
19 instruções		15 instruções	
19 recuperações		15 recuperações	
19 acessos a operandos		11 acessos a operandos	
38 acessos à memória		26 acessos à memória	

Economia de 12/38 = 31,5%

Abordagens para Conjuntos de Instruções

Projetando Instruções

- Instruções com tempo de execução muito diferentes ou com número de fases muito diferentes não são adequadas para uma linha de produção (pipeline)
  - Porque não criar instruções simples com pequenas diferenças em tempo de ex. de fases (e mesmo número de fases)

- Não seria mais vantajoso:
  - Criar instruções poderosas que resolvessem problemas comuns ao invés de instruções simples que resolvem quase nada?

Abordagens para Conjuntos de Instruções

- Final dos anos 70, surge a idéia de Computadores de arquitetura de alto nível (HLLCA – High Level Language Computer Architecture)

- No ínicio dos anos 80, Ditzel e Patterson argumentavam que arquiteturas mais simples seriam o melhor caminho e apresentam a idéia do Reduced Instruction Set Computer (RISC)

- Na mesma época, alguns projetistas (VAX) refutaram a idéia e seguiram construindo computadores baseados em conjuntos de instrução complexos (Complex Instruction Set Computer, CISC)

Desenvolvimento

- Os desenvolvimentos RISC e CISC continuaram em paralelo,disputando mercado
  - A arquitetura teve três grandes projetos iniciais:
    - RISC de Berkeley(Patterson e outros)
    - IBM 801
    - MIPS de Stanford (Hennessy e outros)

RISC I,II e MIPS

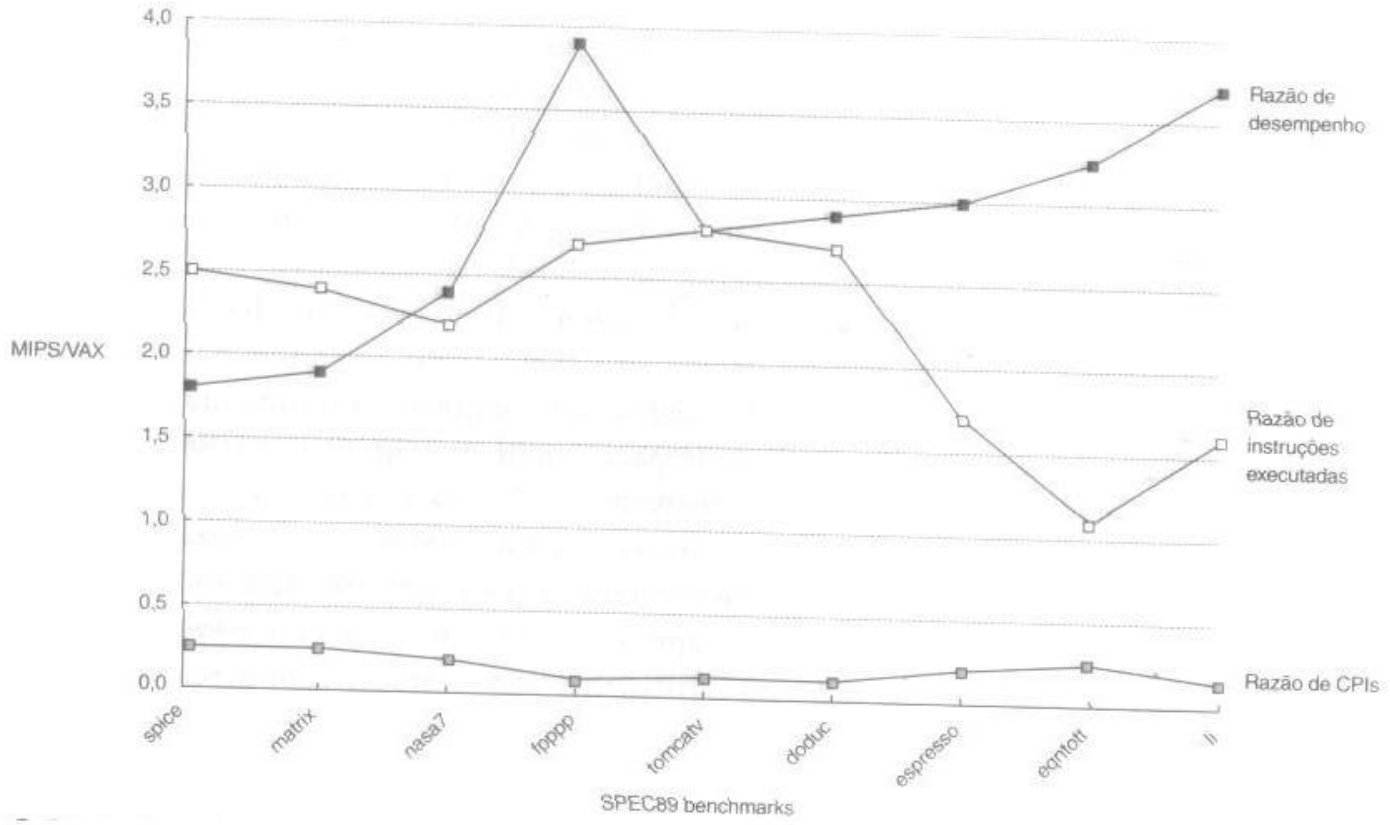
- RISC de Berkeley
  - RISC I e II (1980 – 1983/84)
  - Instruções de 16 e 32 bits
- MIPS (Microprocessor without Interlocked Pipeline Stages)
  - Instruções de 32 bits
- Após sua conclusão, os projetos das Universidades foram amplamente adotados pela indústria

- A IBM nunca chegou a lançar o IBM 801 no mercado, mas criou em 1990 o RS 6000 o primeiro RISC superescalar

RISC x CISC

- Para ajudar a resolver o debate entre RISC e CISC, os projetistas do VAX fizeram uma comparação entre o VAX 8700 e o MIPS M2000 no início dos anos 90.
  - VAX: Modos de endereçamentos poderosos, instruções eficientes, codificação de instrução eficiente e poucos registradores
  - MIPS M2000: Instruções simples, modos de endereçamentos simples, formato de instruções de comprimento fixo, grande número de registradores, pipelining
- Os computadores tinham organizações semelhantes e tempos de clock iguais

Gráfico de Desempenho
MIPS / VAX



RISC x CISC

- Finalmente, sobreviveu apenas um CISC a este embate: x86
  - Alto volume de chips
  - Compatibilidade binária com o software do PC
  - Conversão interna de CISC para RISC
  - Escala suficiente para suportar o hardware extra

RISC x CISC

- Mercado Embutido
  - Celulares, PDAs, eletrodomésticos,...
  - Crítico Custo e Energia: Não há espaço para conversão de hardware
  - Utiliza Compiladores e Arquiteturas RISC

- Em 2000, o número de processadores embutidos comercializados foi mais de duas vezes o número de processadores x86 (mais de 90% deles RISC)

Abordagens para Arquitetura de Conjuntos de Instruções

Accumulator:		
1 address	add A	acc ← acc + mem[A]
1+x address	addx A	acc ← acc + mem[A + x]
Stack:		
0 address	add	tos ← tos + next
General Purpose Register:		
2 address	add A B	EA(A) ← EA(A) + EA(B)
3 address	add A B C	EA(A) ← EA(B) + EA(C)
Load/Store:		
3 address	add Ra Rb Rc	Ra ← Rb + Rc
	load Ra Rb	Ra ← mem[Rb]
	store Ra Rb	mem[Rb] ← Ra

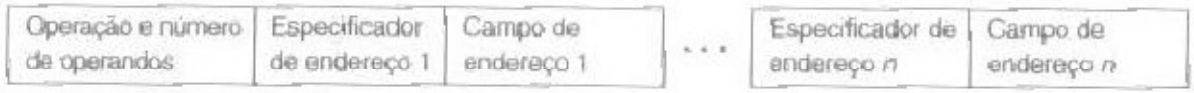
Registradores de Propósitos Gerais dominam

- Desde meados da década de 70, registradores de propósito geral são usados por todas as máquinas
- Vantagens:
  - Registradores são (muito) mais rápido do que memória
  - Registradores são mais eficientes para compiladores otimizarem código:
    - (A*B)-(C*D)+(E*F)….as multiplicações podem ser feitas em qualquer ordem, o que não seria possível com pilha
  - Registradores podem armazenar variáveis
    - O tráfego com a memória diminui
    - A densidade de código aumenta (são necessários menos bits para apontar um registrador do que uma posição de memória)

Alguns Exemplos

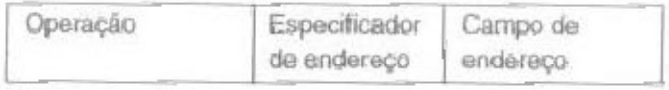
Number of memory addresses per typical ALU instruction	Maximum number of operands per typical ALU instruction	Examples
0	3	SPARC, MIPS, Precision Architecture, Power PC
1	2	Intel 80x86, Motorola 68000
2	2	VAX (also has 3-operand formats)
3	3	VAX (also has 2-operand formats)

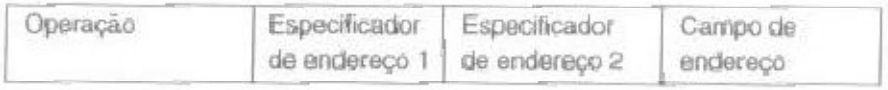
Exemplos de Formato de Conjunto de instruções

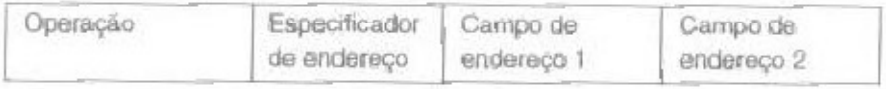


(a) Variável (por exemplo, VAX, Intel 80x86)

(b) Fixo (por exemplo, Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)







(c) Híbrido (por exemplo, IBM 360/70, MIPS16, Thumb, TI TMS320C54x)

Mundo RISC (CISC)

- Exploração do Paralelismo
  - Entre fases
  - Entre Instruções
  - Entre Threads
  - Entre programas
- Solução dos novos problemas trazidos pelo paralelismo

Controle de um Processador com Registradores de Propósito Geral

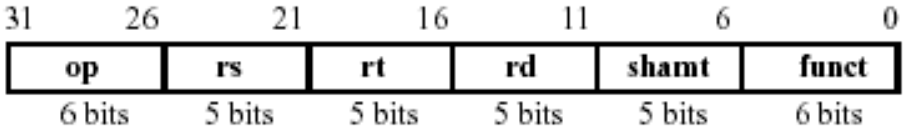
- MIPS (Microprocessor without Interlocked Pipeline Stages)
- Inspirou toda uma família de arquiteturas de processadores RISC, inclusive os atuais processadores ARM

Um Exemplo de Processador com Registradores de Propósito Geral: MIPS e suas Principais Instruções

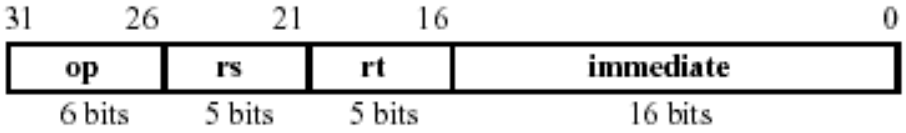
- **Adição**
  - **add R1,R2,R3; R1 = R2 + R3**
- **Subtração**
  - **sub R1,R2,R3; R1 = R2 – R3**
- **Adição de constante (add immediate)**
  - **addi R1,R2,100; R1 = R2 + 100**
- **Multiplicação (resultado em 64 bits)**
  - **mult R2,R3; Hi, Lo = R2 x R3**
- **Divisão (resultado em 64 bits)**
  - **div R2,R3; Lo = R2 ÷ R3, Hi = R2 mod R3**
  - **Lo = quotient, Hi = remainder**

Formato dos Conjunto de Instruções MIPS

- ADD and subtract
  - add rd, rs, rt
  - sub rd, rs, rt



- OR Imm:
  - ori rt, rs, imm16

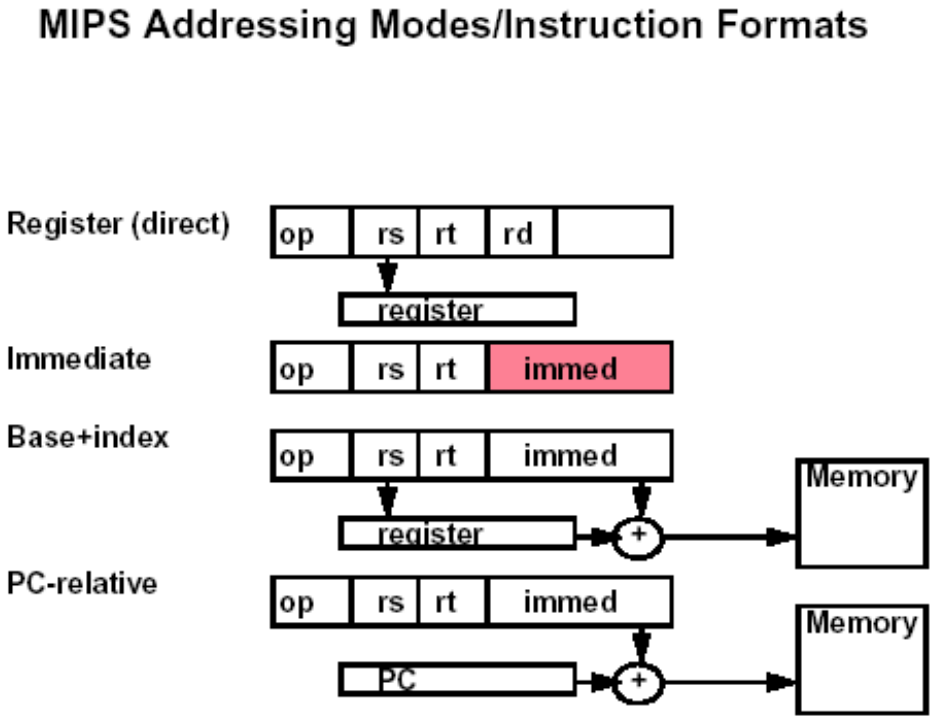


- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16
- JUMP:
  - j target

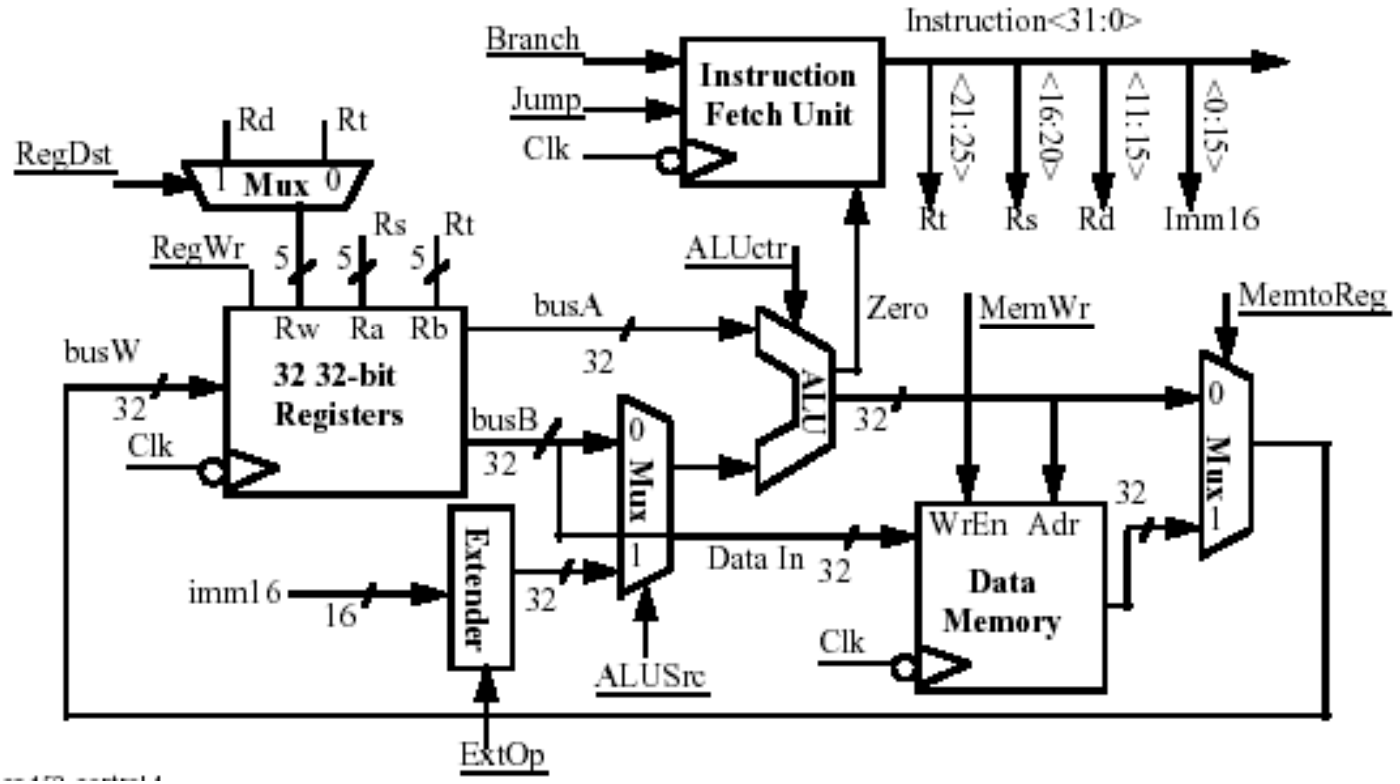
Um Exemplo: MIPS
Principais Instruções

- **Alterar memória (word)**
  - **SW R3, 500(R4) Mem[R4 + 500] =R3**
- **Ler memória (word)**
  - **LW R1, 30(R2) R1 = Mem[R2 + 30]**
- **Desvio Condicional**
  - **beq R1,R2,100 if (R1 == R2) go to PC+4+400**
- **Desvio incondicional (constante)**
  - **jump j 2500; go to 10000**
- **Desvio incondicional (registrador)**
  - **jr R31; go to R31**

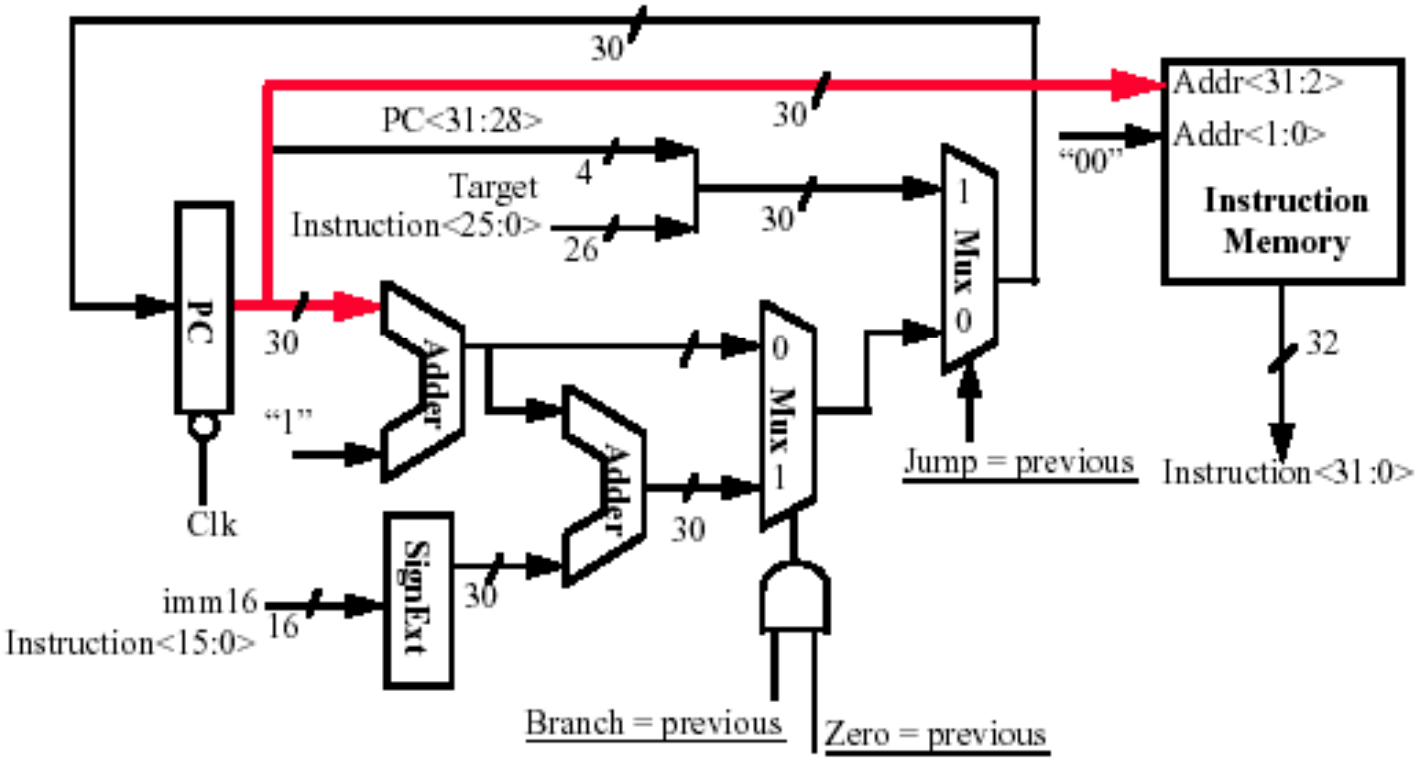
Formato de Instruções MIPS



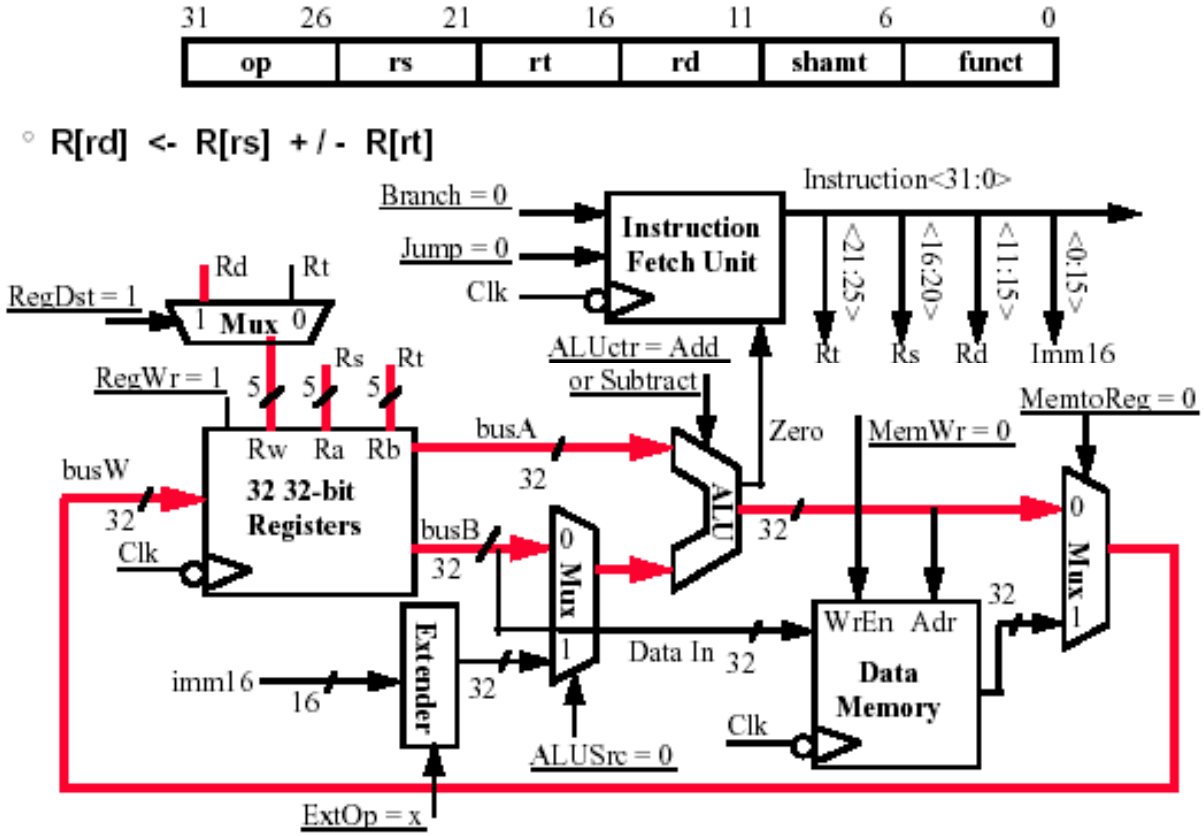
Unidades Funcionais e Controle no MIPS



Carregamento de Instrução

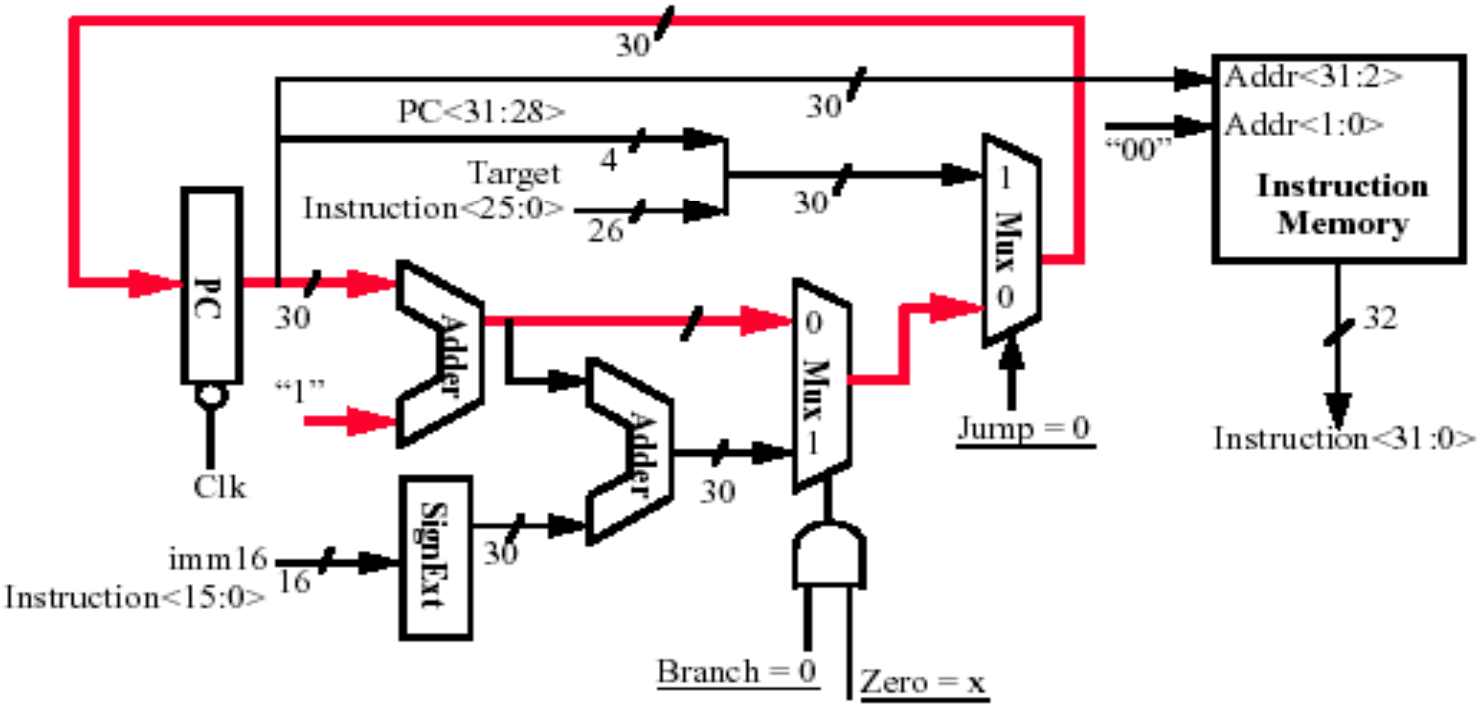


Controlando Operações R-type Adição/Subtração/And/or...

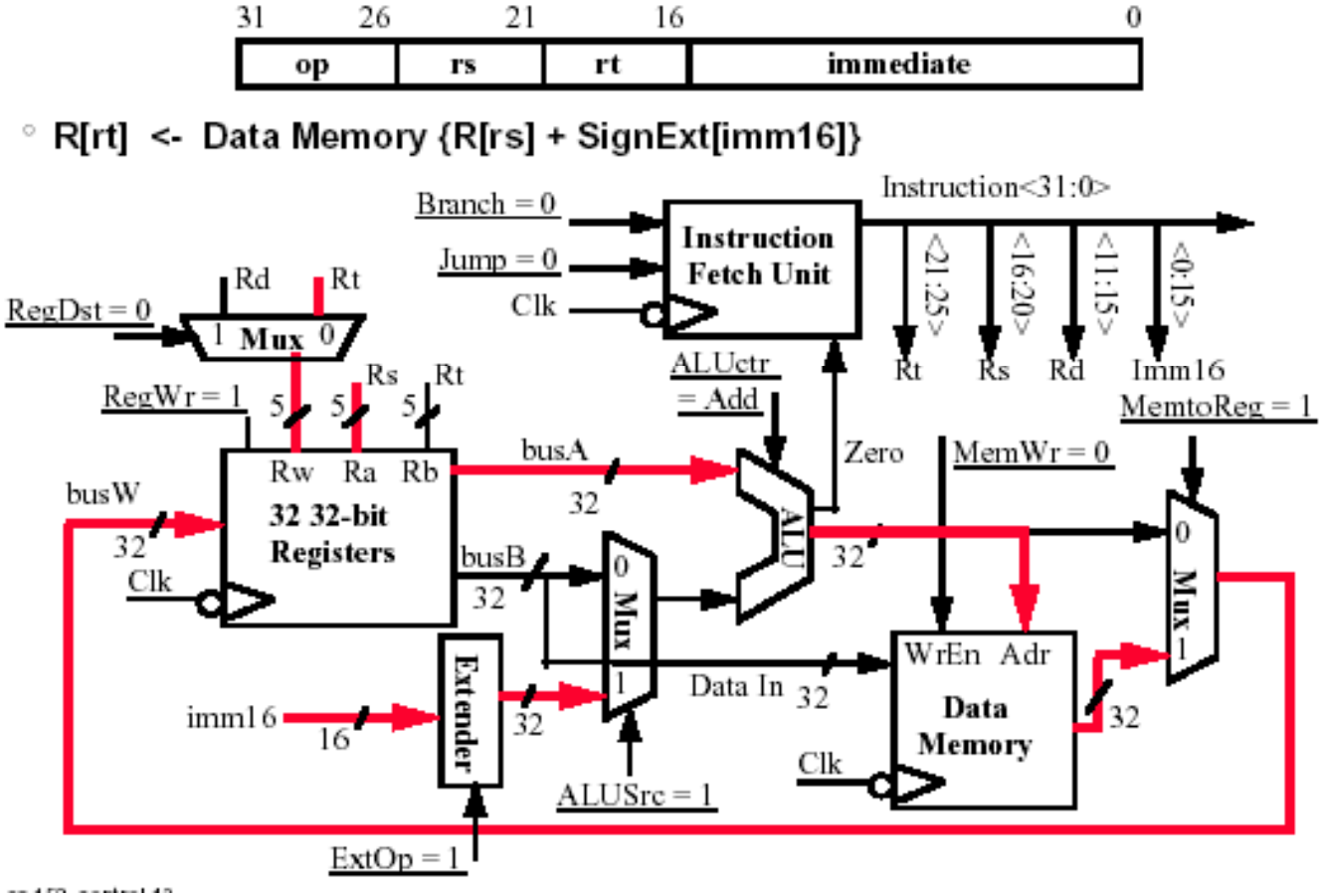


IF no fim de uma instrução

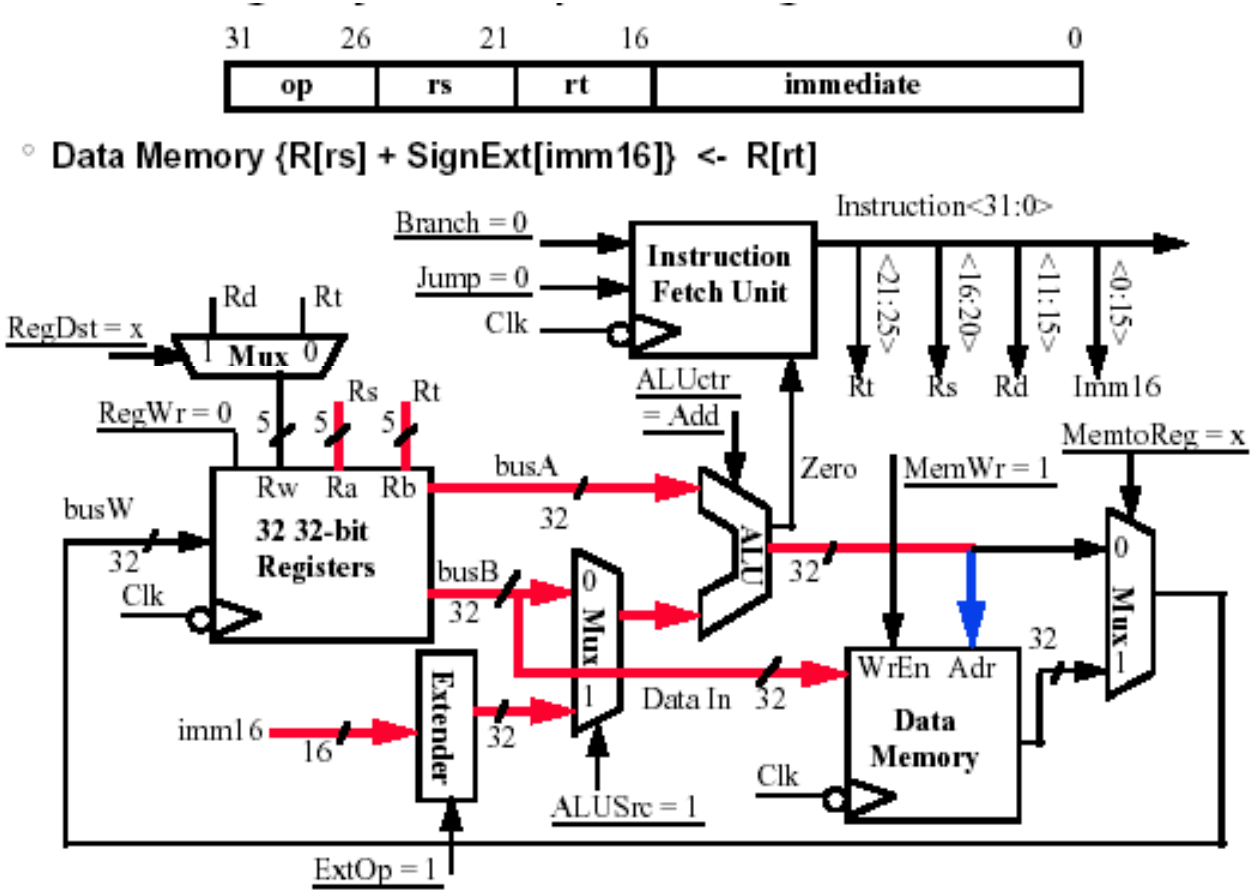
- PC <- PC + 4
  - This is the same for all instructions except: Branch and Jump



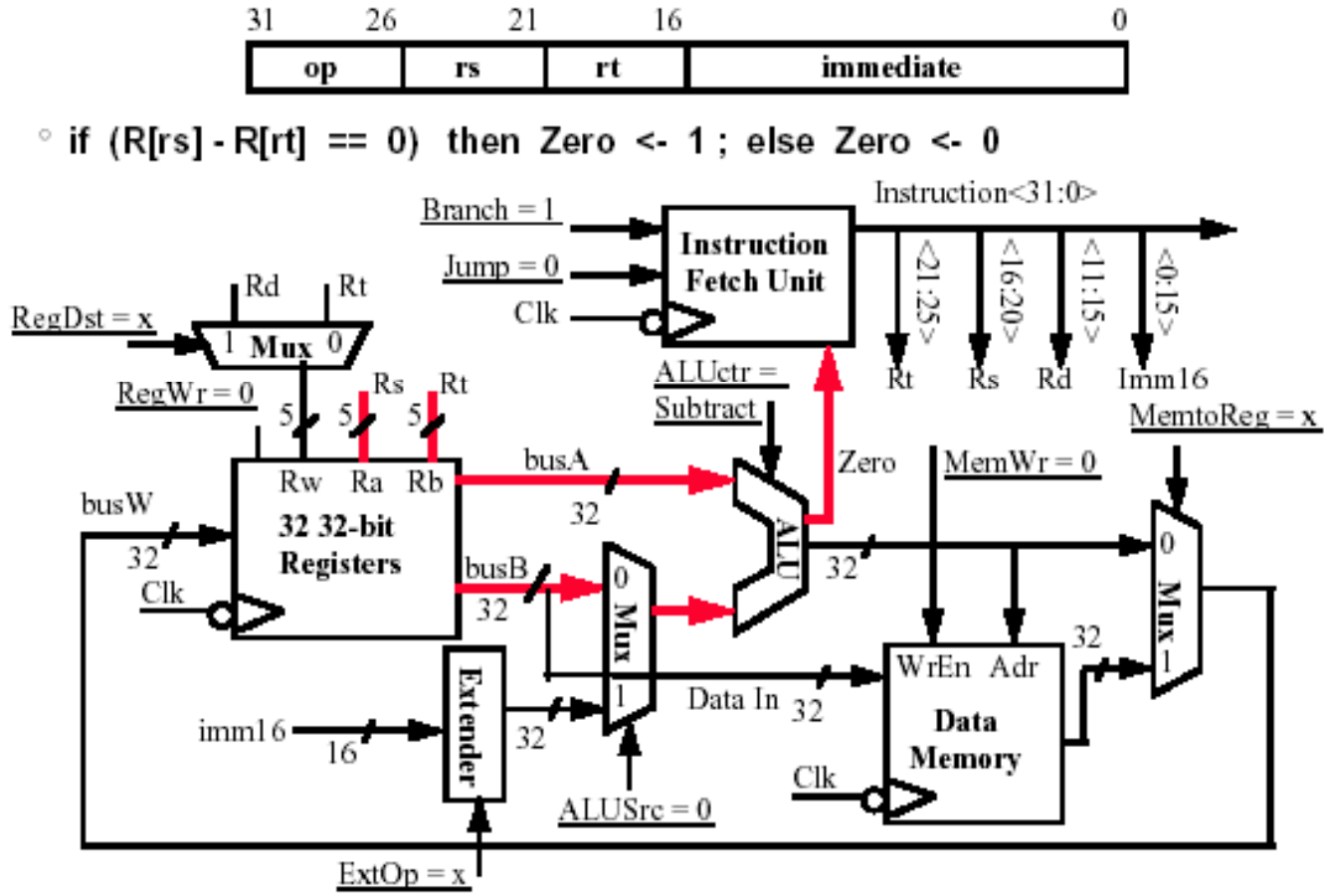
Instrução Load



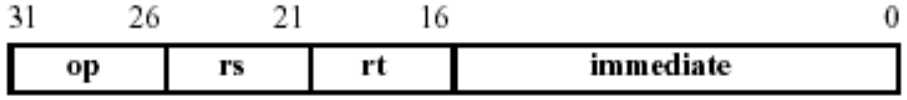
Instrução Store



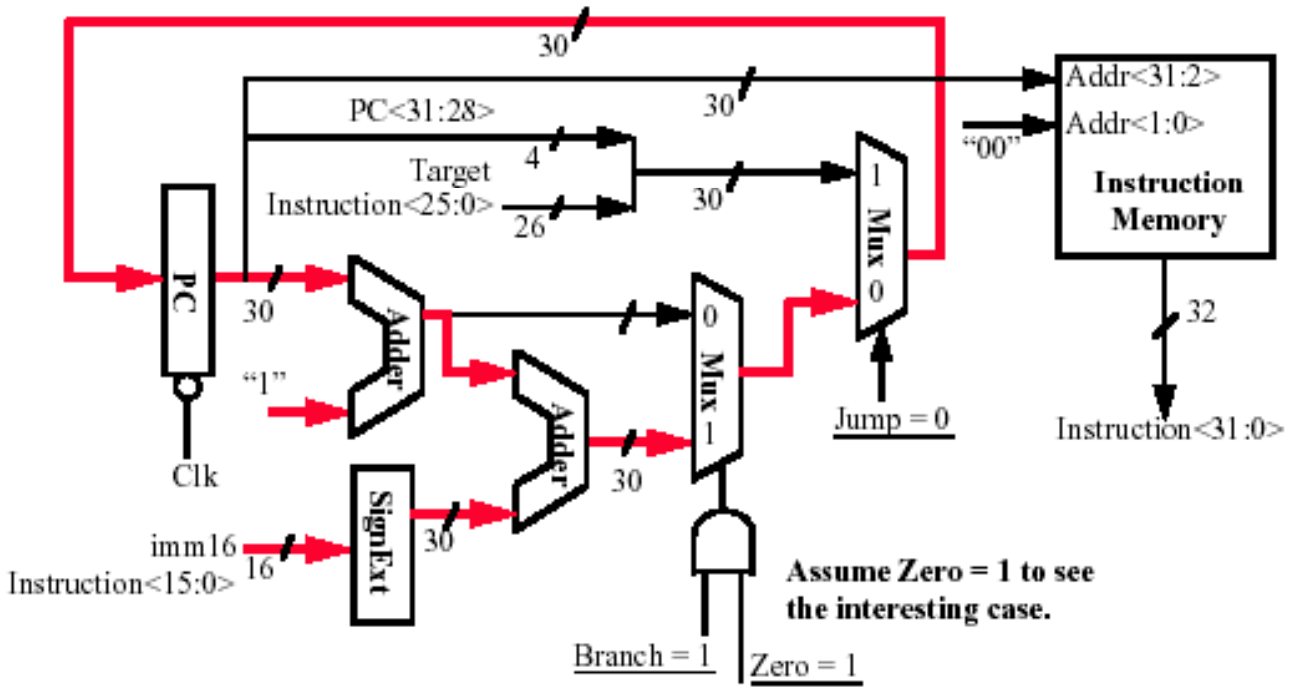
Instrução de Desvio Condicional



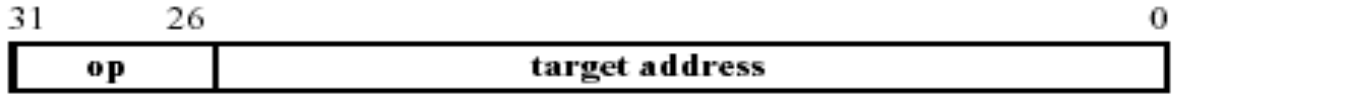
IF ao fim de uma instrução de desvio condicional



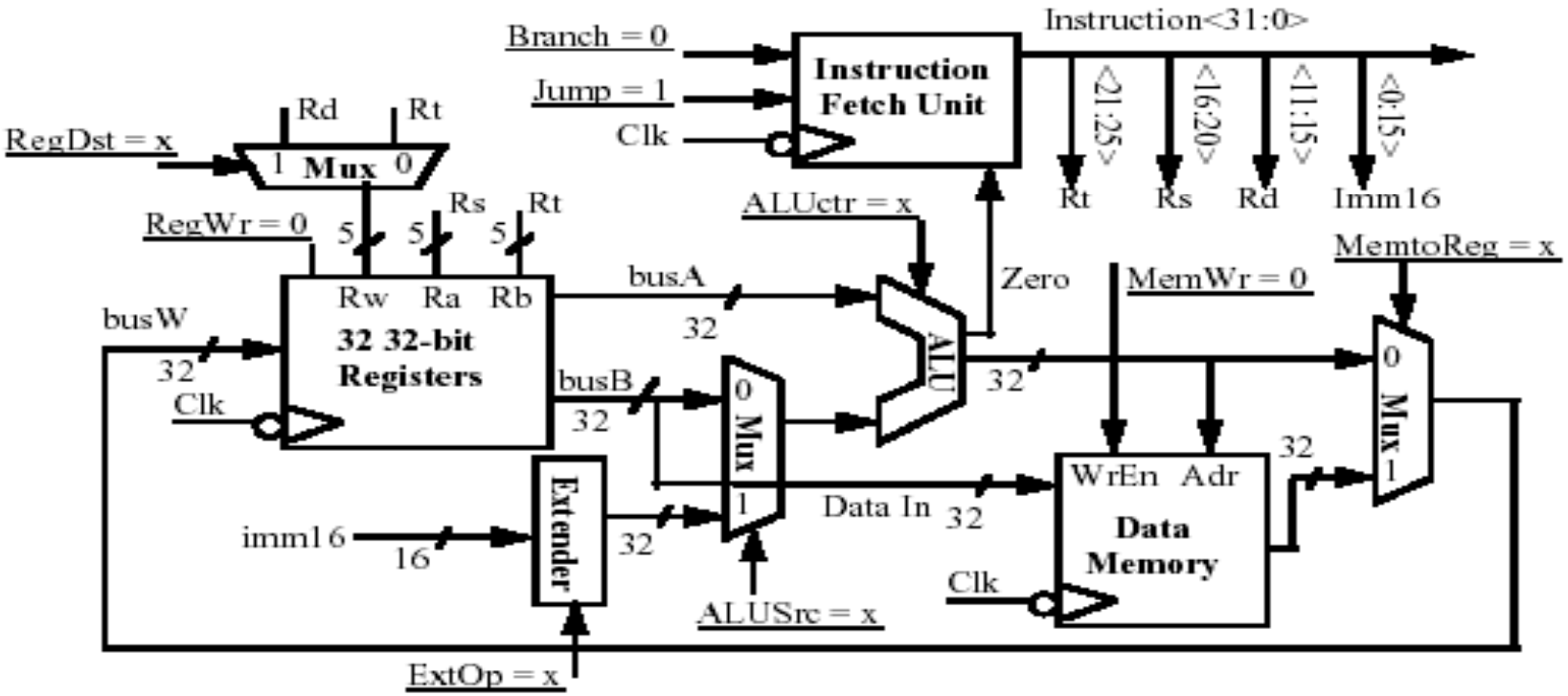
- if (Zero == 1) then PC = PC + 4 + SignExt[imm16]*4 ; else PC = PC + 4



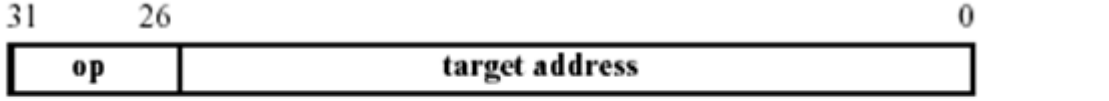
Desvio não condicional



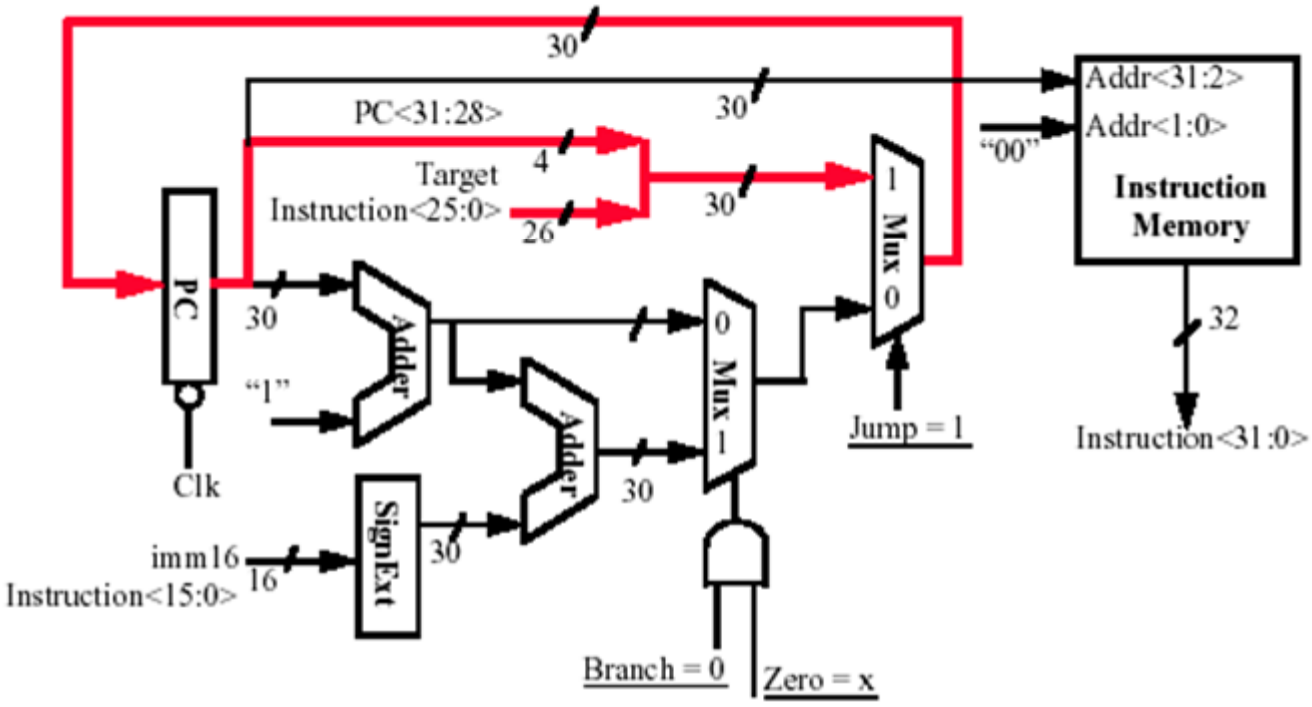
- **Nothing to do! Make sure control signals are set correctly!**



IF ao fim de uma desvio não-condicional

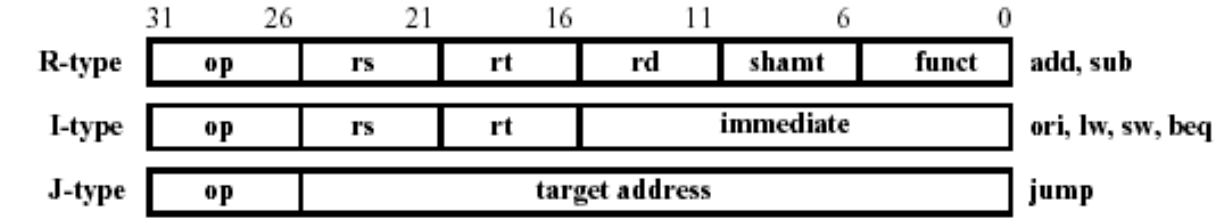


- PC <- PC<31:28> concat target<25:0> concat "00"



Sumário dos Sinais de Controle

See Appendix A							
func	10 0000	10 0010	**We Don't Care :-)**				
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	add	**sub**	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	0	x	x	x
ALUSrc	0	0	1	1	1	0	x
MemtoReg	0	0	0	1	x	x	x
RegWrite	1	1	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0
Branch	0	0	0	0	0	1	0
Jump	0	0	0	0	0	0	1
ExtOp	x	x	0	1	1	x	x
ALUctr<2:0>	Add	Subtract	Or	Add	Add	Subtract	xxx

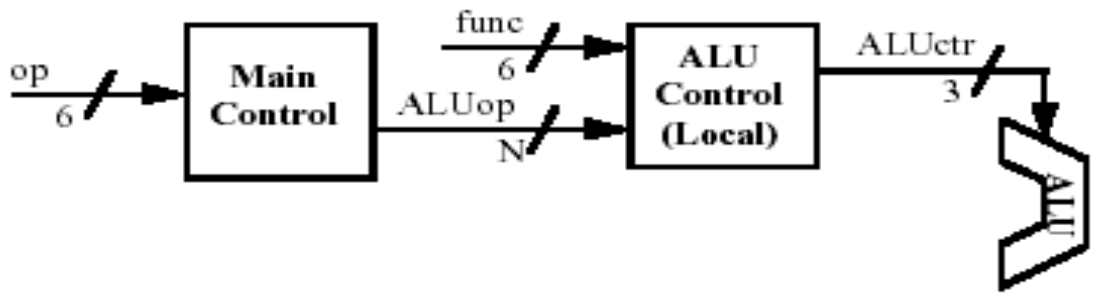


cs 152 control.19

©DAP & SIK 1995

Decodificação Local – Controle da ALU

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop<N:0>	"R-type"	Or	Add	Add	Subtract	xxx



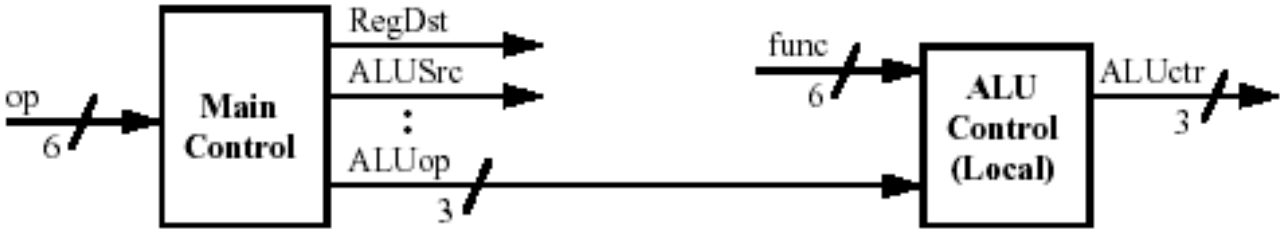
The Truth Table for ALUctr

ALUop (Symbolic)	R-type	ori	lw	sw	beq
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01

funct<3:0>	Instruction Op.
0000	add
0010	subtract
0100	and
0101	or
1010	set-on-less-than

ALUop			func				ALU Operation	ALUctr		
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>		bit<2>	bit<1>	bit<0>
0	0	0	x	x	x	x	Add	0	1	0
0	x	1	x	x	x	x	Subtract	1	1	0
0	1	x	x	x	x	x	Or	0	0	1
1	x	x	0	0	0	0	Add	0	1	0
1	x	x	0	0	1	0	Subtract	1	1	0
1	x	x	0	1	0	0	And	0	0	0
1	x	x	0	1	0	1	Or	0	0	1
1	x	x	1	0	1	0	Set on <	1	1	1

The "Truth Table" for the Main Control



op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUop <2>	1	0	0	0	0	x
ALUop <1>	0	1	0	0	0	x
ALUop <0>	0	0	0	0	1	x