Arquitectura de Computadores

Y86
Sequencial

Créditos

Randal E. Bryant

http://csapp.cs.cmu.edu

Y86 - Jogo de Instruções

Octeto	0	1	2	3	4	5
nop	0	0				
halt	1	0				
rrmovl rA, rB	2	0	rA	rB		
irmovl V, rB	3	0	8	rB	V	
rmmovl rA, D(rB)	4	0	rA	rB	D	
mrmovl D(rB), rA	5	0	rA	rB	D	
OPl rA, rB	6	fn	rA	rB		
jXX Dest	7	fn	Dest			
call Dest	8	0	Dest			
ret	9	0				
pushl rA	A	0	rA	8		
popl rA	B	0	rA	8		

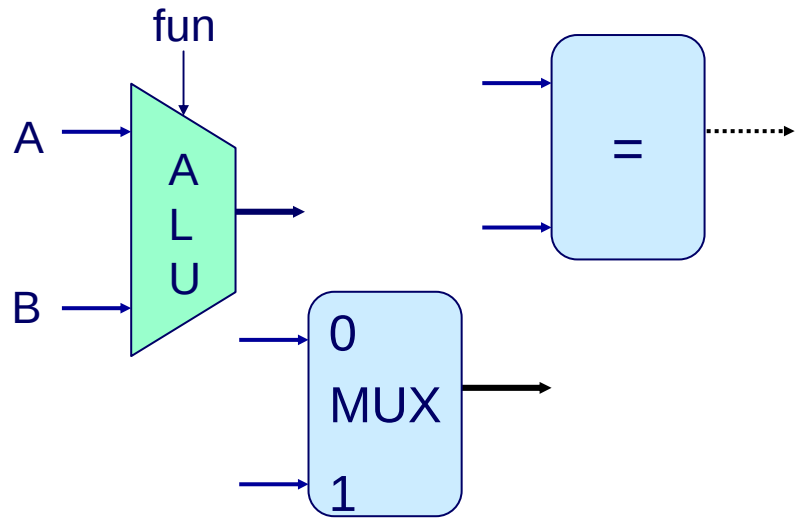
addl	6	0
subl	6	1
andl	6	2
xorl	6	3

jmp	7	0
jle	7	1
jl	7	2
je	7	3
jne	7	4
jge	7	5
jg	7	6

Elementos de Base

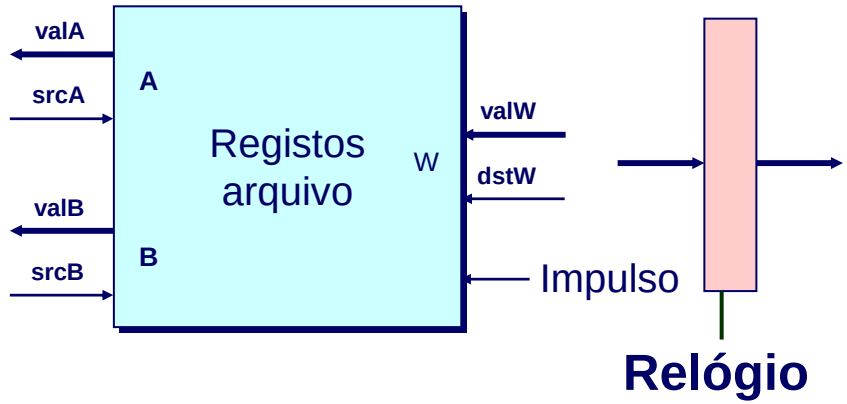
Lógica Combinacional

- Funções Booleanas e entradas
- Reage a alteração das entradas
- Operação e controlo de dados



Elementos de Armazenamento

- Bits
- Memória endereçável
- Registos não-endereçáveis
- Sincronização ao impulso



Linguagem de Controlo

- **Linguagem de descrição de *hardware* muito simples**
- **HCL-*Hardware Control Language*** (anglo-saxónico)
- **Descreve apenas alguns aspectos da operação do hardware**
  - **Partes a explorar e a modificar**

Tipos de Dados

- **`bool`: Boolean**
  - **`a`, `b`, `c`, …**
- **`int`: palavras**
  - **`A`, `B`, `C`, …**
  - **Não especifica tamanhos --- octetos, palavras de 32-bit , …**

Declarações

- **`bool a =` *`bool-expr`* `;`**
- **`int A =` *`int-expr`* `;`**

HCL - Operações

- Classificação por tipo e valor de retorno

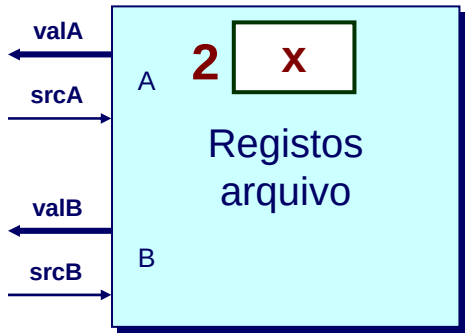
Expressões Booleanas

- Operações Lógicas
  - `a && b, a || b, !a`
- Comparação de palavras
  - `A == B, A != B, A < B, A <= B, A >= B, A > B`
- Inclusão de conjunto
  - `A em { B, C, D }`
    - » Idêntico a `A == B || A == C || A == D`

Expressões

- caso
  - `[ a : A; b : B; c : C ]`
  - Avaliação expressões de teste a, b, c, … na sequência
  - Retorno do valor A, B, C, … a primeira a satisfazer a expressão

Arquivo de Registos

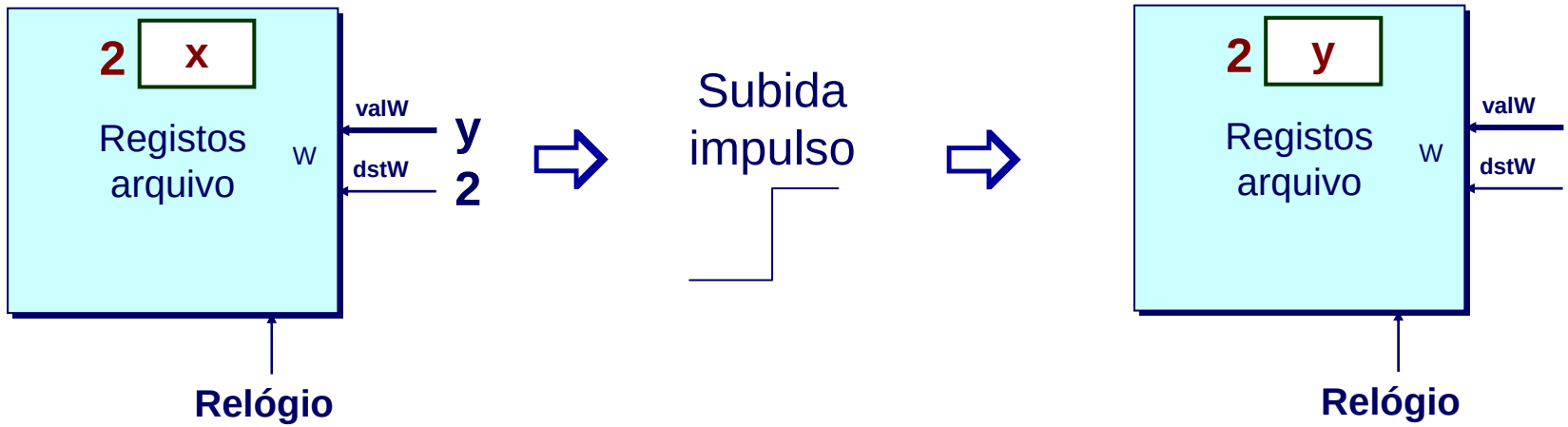


Leitura

- Lógica combinacional
- Dados à saída seleccionados pelos endereços à entrada
  - Atraso de propagação

Escrita

- Como num registo
- Na subida do impulso



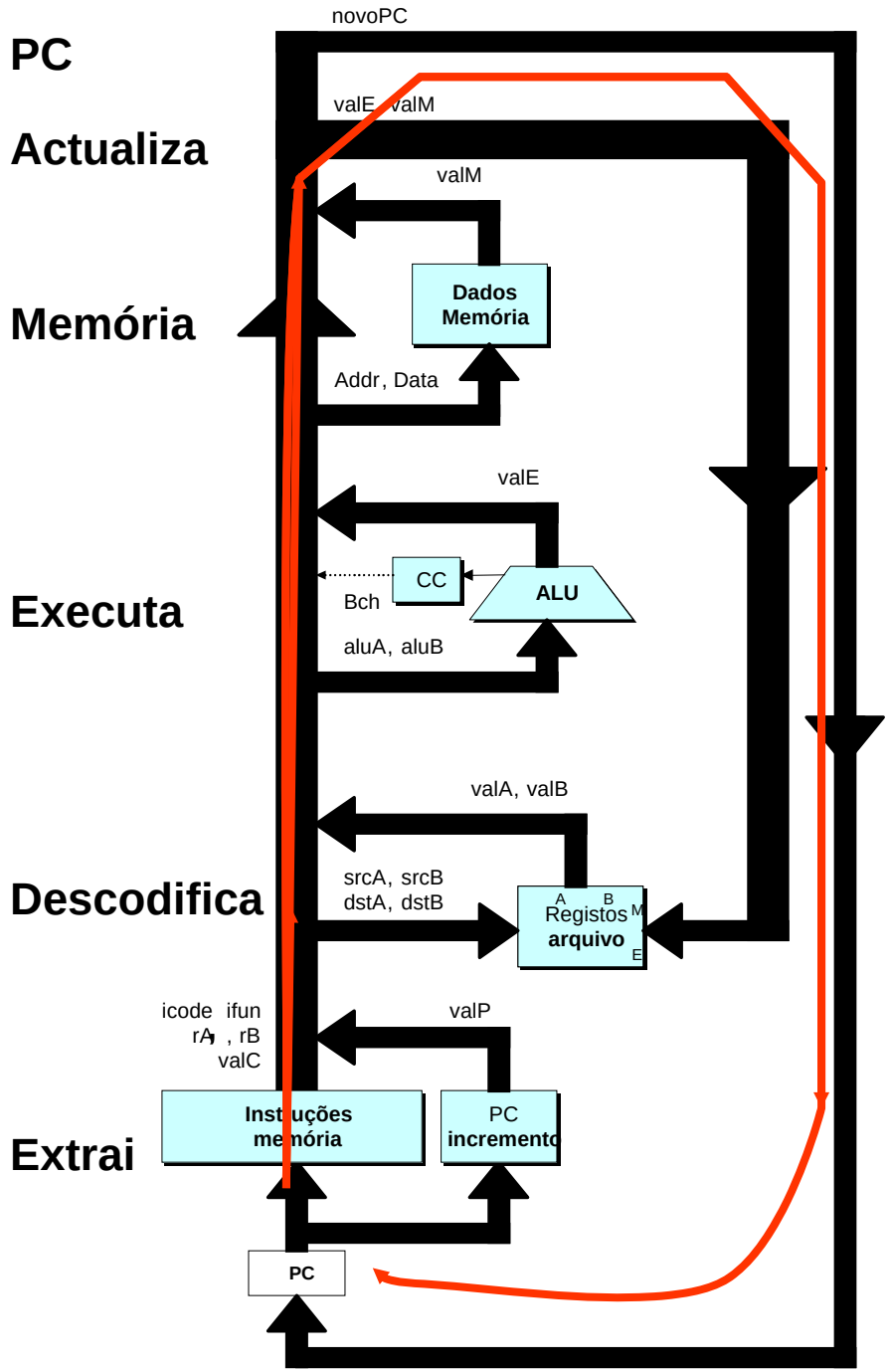
SEQ - Estrutura *Hardware*

Estado

- Contador de programa (PC)
- Códigos de condição (CC)
- Arquivo de registos
- Memórias
  - Espaço de memória único
  - Dados
    - » Leitura/escrita
  - Instruções
    - » Leitura

Fluxo de Instruções

- Instruções referenciadas pelo PC
- Processamento em estágios
- Actualização do PC



SEQ - Estágios

Extrai

- Lê instruções da memória

Descodifica

- Registos de programa

Executa

- Cálculo de valores e endereços

Memória

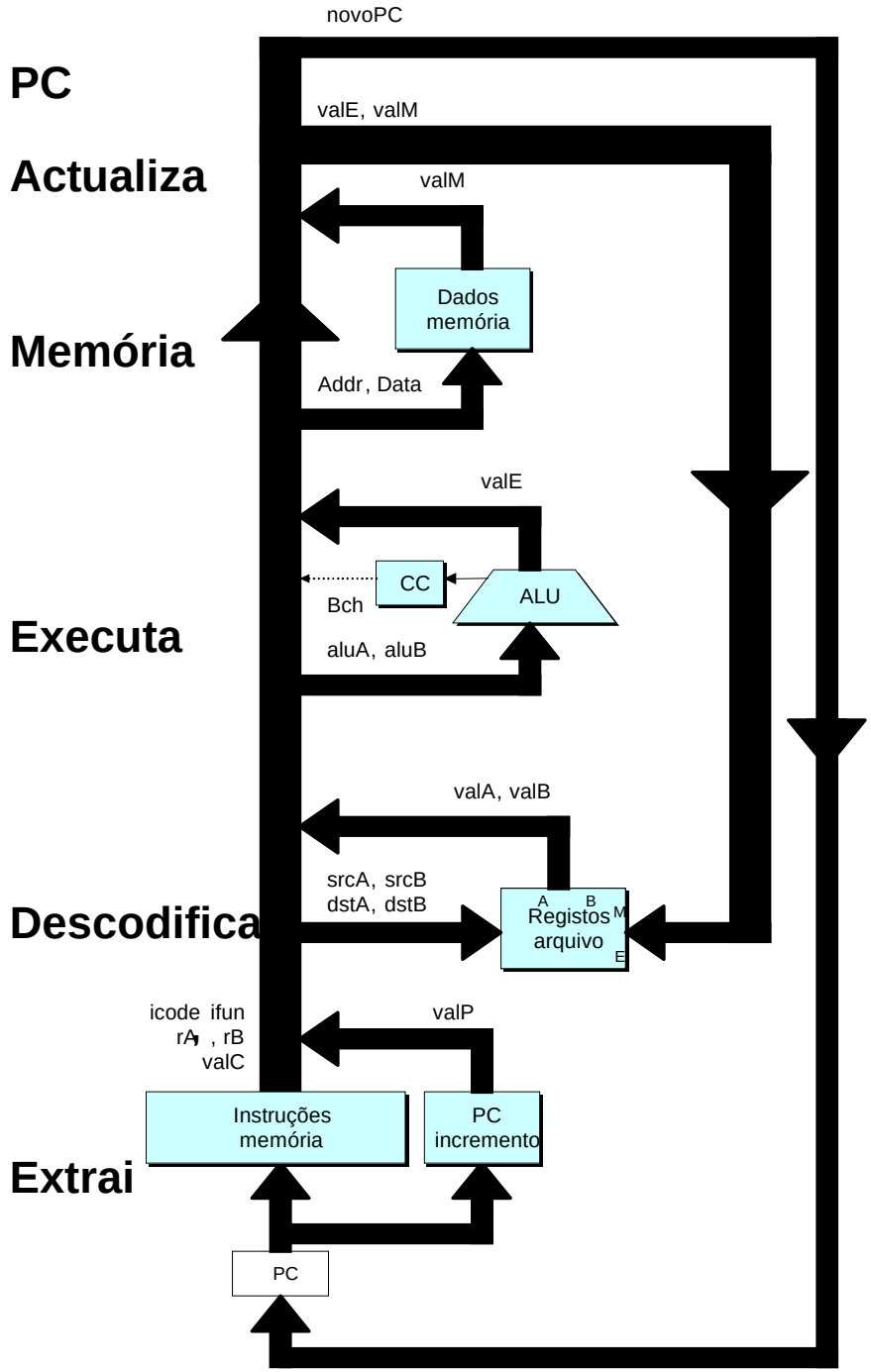
- Leitura/escrita dados

Actualiza

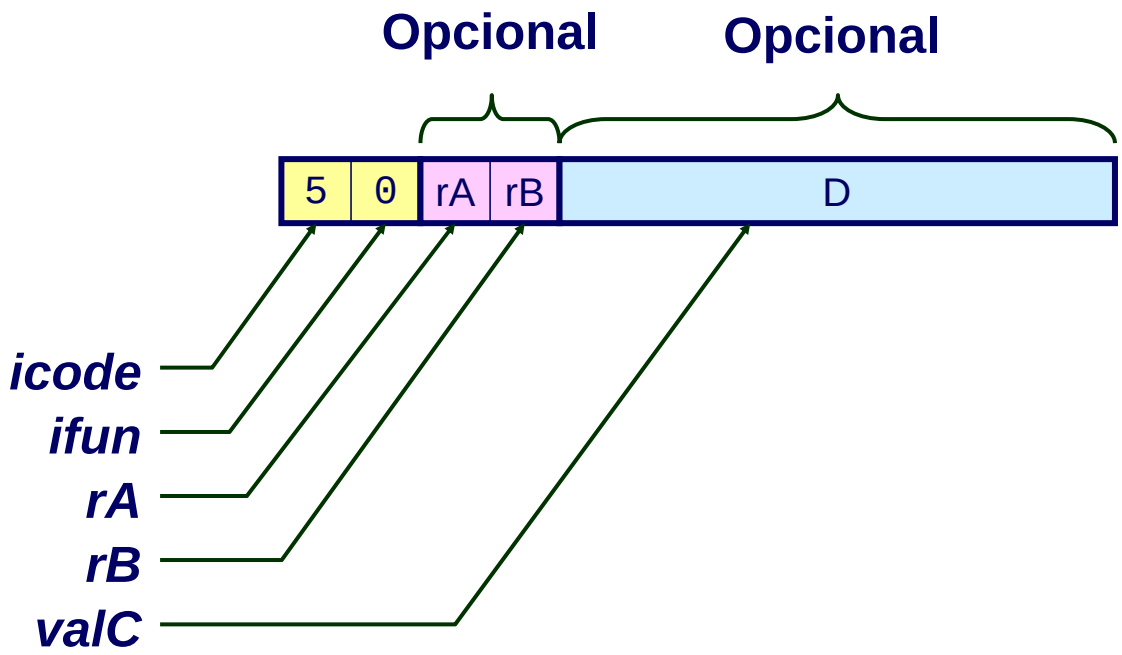
- Actualiza registos

PC

- Actualiza PC



Descodificação de Instruções



Formato de Instruções

- Instrução - octeto *icode:ifun*
- Registo opcional - octeto *rA:rB*
- Constante opcional – palavra *valC*

Execução - Operações Arit./Lógicas

`OPl rA, rB`

Extrai

- lê 2 octetos

Descodifica

- Lê operandos em registos

Executa

- Efectua a operação
- Ajusta códigos de condição

Memória

- Não faz nada

Actualiza

- Escreve no registo de destino

PC

- Actualiza PC (+2)

Estágios - Operações: Arit./Lógicas

	OPl rA, rB	
Extrai	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	Lê instrução – 1 octeto Lê registo – 1 octeto Calcula valor PC
Descodifica	valA ← R[rA] valB ← R[rB]	Lê operando A Lê operando B
Executa	valE ← valB OP valA Ajusta CC	Efectua operação na ALU Ajusta códigos de condição
Memória		
Actualiza	R[rB] ← valE	Actualiza registo destino
PC	PC ← valP	Actualiza PC

- Instruções executadas numa sequência de passos simples
- Idêntico para todas as operações aritméticas/lógicas

Execução - rmmovl

Extrai

- Lê 6 octetos

Descodifica

- Lê operandos em registos

Executa

- Calcula o endereço efectivo

Memória

- Escreve na memória

Actualiza

- Não faz nada

PC

- Actualiza PC (+6)

Estágios : `rmmovl`

	`rmmovl` rA, D(rB)	
Extrai	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_4[PC+2] valP ← PC+6	Lê instrução – 1 octeto Lê registo – 1 octeto Lê desfasamento D Calcula valor PC
Descodifica	valA ← R[rA] valB ← R[rB]	Lê operando A Lê operando B
Executa	valE ← valB + valC	Calcula endereço efectivo
Memória	M_4[valE] ← valA	Escreve valor na memória
Actualiza		
PC	PC ← valP	Actualiza PC

- Cálculo de endereços através da ALU

Execução - popl

Extrai

- Lê 2 octetos

Descodifica

- Lê apontador de pilha

Executa

- Calcula novo valor para o apontador pilha (+4)

Memória

- Lê da memória no endereço antigo do apontador de pilha

Actualiza

- Actualiza apontador de pilha
- Actualiza registo destino

PC

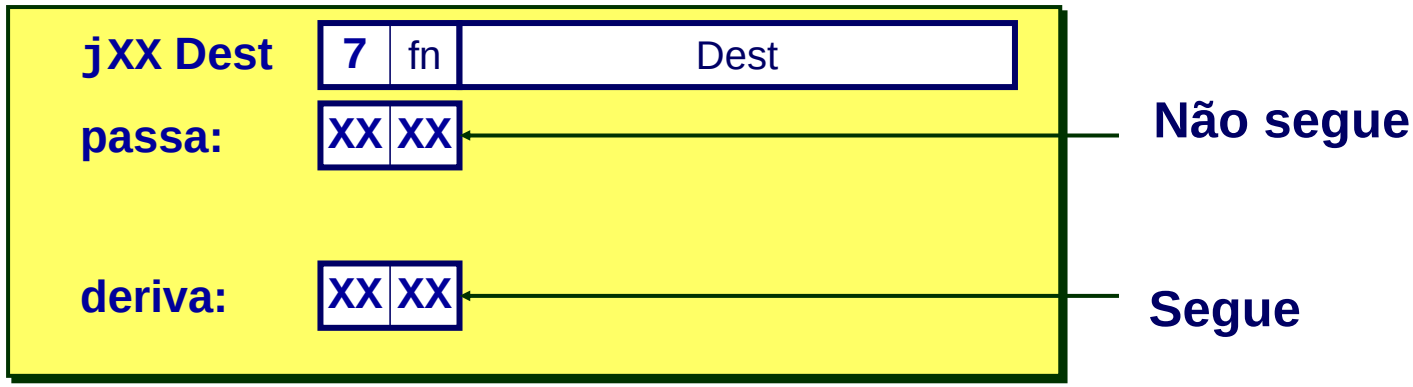
- Actualiza PC (+2)

Estágios : `popl`

	`popl rA`	
Extrai	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	Lê instrução – 1 octeto Lê registo – 1 octeto Calcula valor PC
Descodifica	valA ← R[%esp] valB ← R [%esp]	Lê valor apont. pilha Lê valor apont. pilha
Executa	valE ← valB + 4	Novo valor apont. pilha
Memória	valM ← M_4[valA]	Lê valor da pilha
Actualiza	R[%esp] ← valE R[rA] ← valM	Actualiza apontador pilha Registo com valor resultado
PC	PC ← valP	Actualiza PC

- Uso da ALU para actualizar o apontador pilha
- Necessário actualizar dois registos
  - Valor extraído da memória
  - Novo valor apontador de pilha

Execução - Saltos



Extrai

- Lê 5 octetos
- Calcula PC (+ 5)

Descodifica

- Não faz nada

Executa

- Decide o salto comparando a condição de salto com os códigos de condição

Memória

- Não faz nada

Actualiza

- Não faz nada

PC

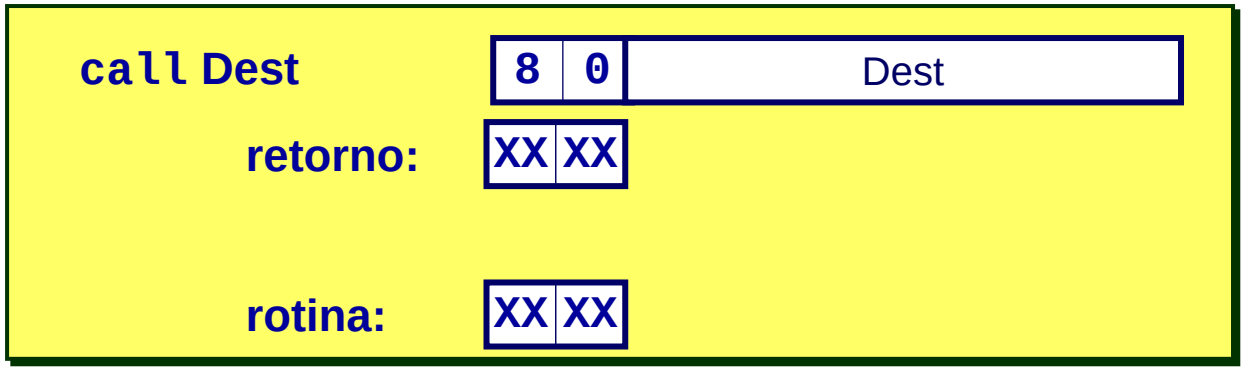
- Actualiza PC com Dest se "Segue" ou com valor antes calculado se "Não segue"

Estágios : Saltos

	jXX Dest	
Extrai	icode:ifun ← M_1[PC] valC ← M_4[PC+1] valP ← PC+5	Lê instrução – 1 octeto Lê endereço destino Calcula valor PC
Descodifica		
Executa	Bch ← Cond(CC,ifun)	Determina a opção a usar
Memória		
Actualiza		
PC	PC ← Bch ? valC : valP	Actualiza PC

- Calcula dois endereços para o PC
- Decisão baseada na condição na instrução e nos CC

Execução - `call`



Extrai

- Lê 5 octetos
- Calcula novo PC (+ 5)

Descodifica

- Lê apontador pilha

Executa

- Calcula novo valor para apontador pilha (-4)

Memória

- Salva novo PC no novo endereço do apontador pilha

Actualiza

- Actualiza o apontador pilha

PC

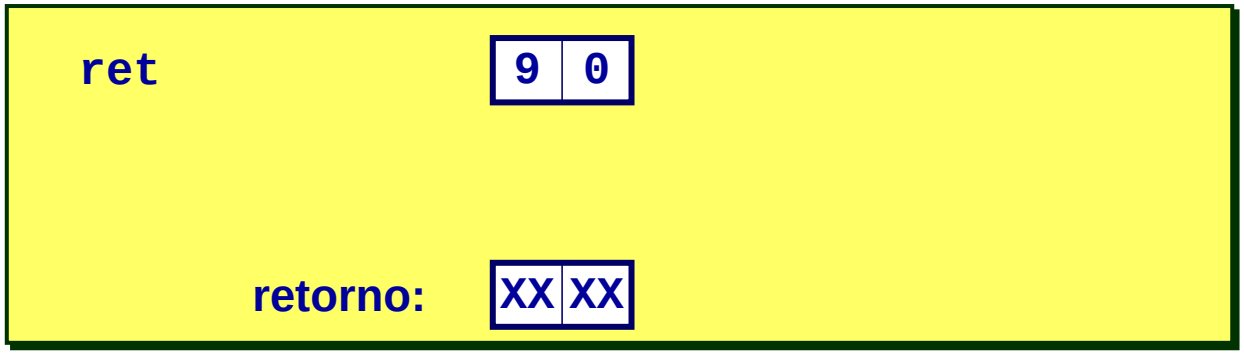
- Actualiza PC com valor Dest

Estágios : `call`

	`call` Dest	
Extrai	icode:ifun ← M_1[PC]	Lê instrução – 1 octeto
	valC ← M_4[PC+1]	Lê endereço de destino
	valP ← PC+5	Calcula o endereço retorno
Descodifica	valB ← R[%esp]	Lê apontador de pilha
Executa	valE ← valB + –4	Decrementa o apont. pilha
Memória	M_4[valE] ← valP	Salva endereço de retorno
Actualiza	R[%esp] ← valE	Actualiza apontador de pilha
PC	PC ← valC	PC com endereço destino

- Uso da ALU para actualizar o apontador de pilha
- Actualiza o valor do PC

Execução - ret



Extrai

- Lê 1 octeto

Descodifica

- Lê apontador de pilhar

Executa

- Calcula novo valor para apontador pilha (+4)

Memória

- Obtém endereço de retorno da posição apontada pelo antigo apontador de pilha

Actualiza

- Actualiza o apontador de pilha

PC

- Actualiza PC com valor retorno:

Estágios : `ret`

	`ret`	
Extrai	icode:ifun ← M_1[PC]	Lê instrução – 1 octeto
Descodifica	valA ← R[%esp] valB ← R[%esp]	Lê operando apont. pilha Lê operando apont. pilha
Executa	valE ← valB + 4	Novo valor para apont. pilha
Memória	valM ← M_4[valA]	Lê endereço retorno
Actualiza	R[%esp] ← valE	Actualiza apontador pilha
PC	PC ← valM	PC com endereço de retorno

- Uso da ALU para actualizar o apontador de pilha
- Lê o endereço de retorno da memória

Passos - Computação

		OPl rA, rB	
Extrai	icode,ifun	icode:ifun ← M_1[PC]	Lê instrução – 1 octeto
	rA,rB	rA:rB ← M_1[PC+1]	Lê registo – 1 octeto
	valC		[Lê constante - 1 palavra]
	valP	valP ← PC+2	Calcula valor PC
Descodifica	valA, srcA	valA ← R[rA]	Lê operando A
	valB, srcB	valB ← R[rB]	Lê operando B
Executa	valE	valE ← valB OP valA	Efectua operação na ALU
	Cond code	Set CC	Ajusta códigos condição
Memória	valM		[Lê/escreve na memória]
Actualiza	dstE	R[rB] ← valE	Registo com resultado
	dstM		[Memória com resultado]
PC	PC	PC ← valP	Actualiza PC

- Todas as instruções seguem o mesmo padrão
- A diferença está no que é calculado em cada passo

Passos - Computação

		`call` Dest	
Extrai	icode,ifun	icode:ifun ← M_1[PC]	Lê instrução – 1 octeto
	rA,rB		[Lê registo – 1 octeto]
	valC	valC ← M_4[PC+1]	Lê constante - 1 palavra
	valP	valP ← PC+5	Calcula valor PC
Descodifica	valA, srcA		[Lê operando A]
	valB, srcB	valB ← R[%esp]	Lê operando B
Executa	valE	valE ← valB + –4	Efectua operação na ALU
	Cond code		[Ajusta códigos condição.]
Memória	valM	M_4[valE] ← valP	[Lê/escreve na memória]
Actualiza	dstE	R[%esp] ← valE	[Registo com resultado]
	dstM		Memória com resultado]
PC	PC	PC ← valC	Actualiza PC

- Todas as instruções seguem o mesmo padrão
- Diferente apenas o que é calculado em cada passo

Valores Calculados

Extracção

icode	Instrução - código
ifun	Instrução - função
rA	Instrução - registo A
rB	Instrução - registo B
valC	Instrução - constante
valP	PC - Aumentado

Descodificação

srcA	Registo ID - A
srcB	Registo ID - B
dstE	Registo Destino - E
dstM	Registo Destino - M
valA	Registo valor - A
valB	Registo valor - B

Execução

- valE
  - ALU - resultado
- Bch
  - Condição - segue

Memória

- valM
  - Memória - valor

Passos – Computação: Exemplo

```
 1   0x000: 308209000000 |     irmovl $9,  %edx
 2   0x006: 308315000000 |     irmovl $21, %ebx
 3   0x00c: 6123         |     subl %edx, %ebx          # subtrai
 4   0x00e: 308480000000 |     irmovl $128,%esp         #
 5   0x014: 404364000000 |     rmmovl %esp, 100(%ebx) # salvaguarda
 6   0x01a: a028         |     pushl %edx               # insere na pilha
 7   0x01c: b008         |     popl  %eax               #
 8   0x01e: 7328000000   |     je done                  # Não segue
 9   0x023: 8029000000   |     call proc                #
10   0x028:              | done:
11   0x028: 10           |     halt
12   0x029:              | proc:
13   0x029: 90           |     ret                      # Retorna
```

- **Sequência de instruções**
- **Traçado do processamento**
  - **através dos diferentes estágios**

Passos – Computação: opl

Estágio	Genérico	Específico
	OPl rA, rB	subl %edx, %ebx
Extrai	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1] valP $\leftarrow$ PC + 2	icode:ifun $\leftarrow$ M_1[0x00c] = 6:1 rA:rB $\leftarrow$ M_1[0x00d] = 2:3 valP $\leftarrow$ 0x00c + 2 = 0x00e
Descodifica	valA $\leftarrow$ R[rA] valB $\leftarrow$ R[rB]	valA $\leftarrow$ R[%edx] = 9 valB $\leftarrow$ R[%ebx] = 21
Executa	valE $\leftarrow$ valB OP valA Set CC	valE $\leftarrow$ 21 − 9 = 12 ZF $\leftarrow$ 0, SF $\leftarrow$ 0, OF $\leftarrow$ 0
Memória		
Actualiza	R[rB] $\leftarrow$ valE	R[%ebx] $\leftarrow$ valE = 12
PC	PC $\leftarrow$ valP	PC $\leftarrow$ valP = 0x00e

- **Inicial**
  - **%edx=9; %ebx=21; PC=0x00C**
- **Efeito**
  - **%ebx=12; CC=000; PC+=2**

Passos – Computação: rrmovl

Estágio	Genéric	Específico
	`rmmovl rA, D(rB)`	`rmmovl %esp, 100(%ebx)`
Extrai	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_4[PC + 2] valP ← PC + 6	icode:ifun ← M_1[0x014] = 4:0 rA:rB ← M_1[0x015] = 4:3 valC ← M_4[0x016] = 100 valP ← 0x014 + 6 = 0x01a
Descodifica	valA ← R[rA] valB ← R[rB]	valA ← R[%esp] = 128 valB ← R[%ebx] = 12
Executa	valE ← valB + valC	valE ← 12 + 100 = 112
Memória	M_4[valE] ← valA	M_4[112] ← 128
Actualiza		
PC	PC ← valP	PC ← 0x01a

- **Efeito**
  - **M[112]=128; PC+=6**

Passos – Computação: pushl

Estágio	Genérico	Específico
	`pushl rA`	`pushl %edx`
Extrai	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2	icode:ifun ← M_1[0x01a] = a:0 rA:rB ← M_1[0x01b] = 2:8 valP ← 0x01a + 2 = 0x01c
Descodifica	valA ← R[rA] valB ← R[%esp]	valA ← R[%edx] = 9 valB ← R[%esp] = 128
Executa	valE ← valB + (−4)	valE ← 128 + (−4) = 124
Memória	M_4[valE] ← valA	M_4[124] ← 9
Actualiza	R[%esp] ← valE	R[%esp] ← 124
PC	PC ← valP	PC ← 0x01c

- **Efeito**
  - **%esp=124; M[124]=9; PC+=2**

Passos – Computação: je

Estágio	Genérico	Específico
	`jXX` Dest	`je 0x028`
Extrai	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{valC} \leftarrow M_4[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC}+5$	$\text{icode:ifun} \leftarrow M_1[\texttt{0x01e}] = 7:3$ $\text{valC} \leftarrow M_4[\texttt{0x01f}] = \texttt{0x028}$ $\text{valP} \leftarrow \texttt{0x01e}+5 = \texttt{0x023}$
Descodifica		
Executa	$\text{Bch} \leftarrow \text{Cond}(\text{CC}, \text{ifun})$	$\text{Bch} \leftarrow \text{Cond}(\langle 0,0,0\rangle, 3) = 0$
Memória		
Actualiza		
PC	$\text{PC} \leftarrow \text{Bch} ? \text{valC} : \text{valP}$	$\text{PC} \leftarrow 0 ? \texttt{0x028} : \texttt{0x023} = \texttt{0x023}$

- **Inicial**
  - **CC = 000;**
- **Efeito**
  - **Não segue; PC+=5**

Passos – Computação: ret

Estágio	Genéric	Específico
	`ret`	`ret`
Extrai	icode:ifun ← M_1[PC] valP ← PC + 1	icode:ifun ← M_1[0x029] = 9:0 valP ← 0x029 + 1 = 0x02a
Descodifica	valA ← R[%esp] valB ← R[%esp]	valA ← R[%esp] = 124 valB ← R[%esp] = 124
Executa	valE ← valB + 4	valE ← 124 + 4 = 128
Memória	valM ← M_4[valA]	valM ← M_4[124] = 0x028
Actualiza	R[%esp] ← valE	R[%esp] ← 128
PC	PC ← valM	PC ← 0x028

- **Efeito**
  - **PC=0x028**

SEQ - *Hardware*

Caixas azuis
- blocos pré-definidos
  - E.g., memórias, ALU

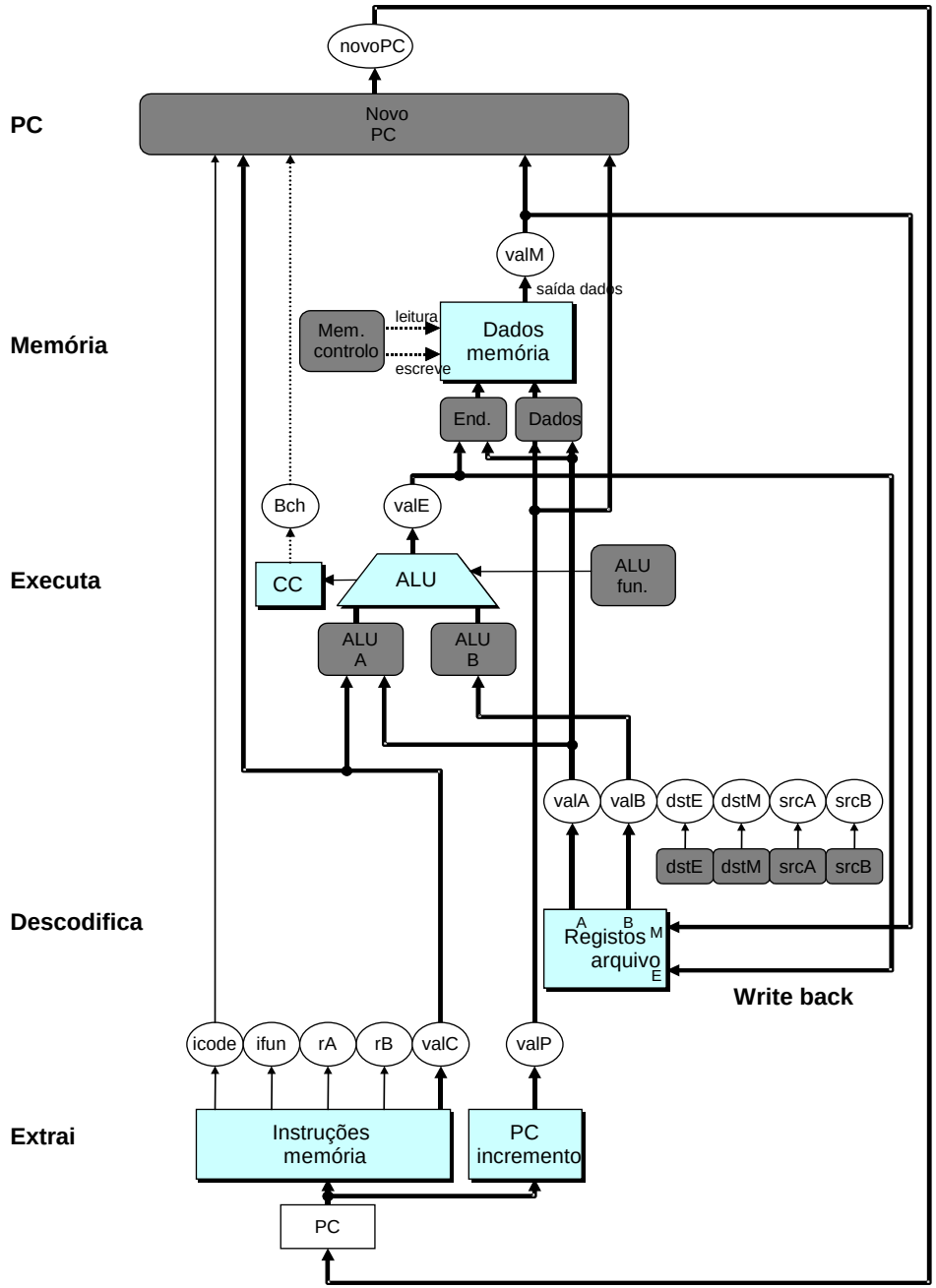
Caixas cinzentas:
- lógica de controlo
  - Especificada em HCL

Ovais brancas
- etiquetas para sinais

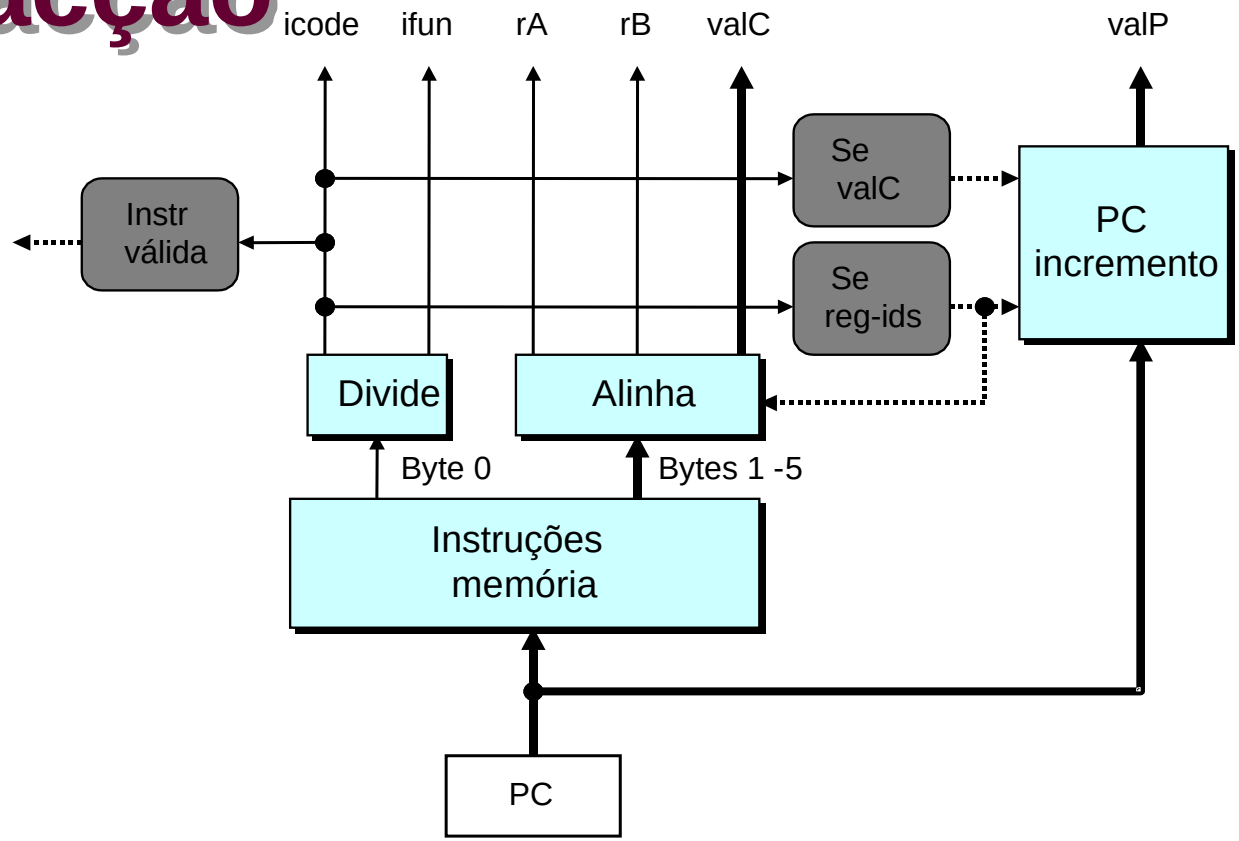
Linhas espessas
- palavras de 32-bits

Linhas finas:
- valores de 4-8 bits

Linhas ponteadas
- valores de 1-bit



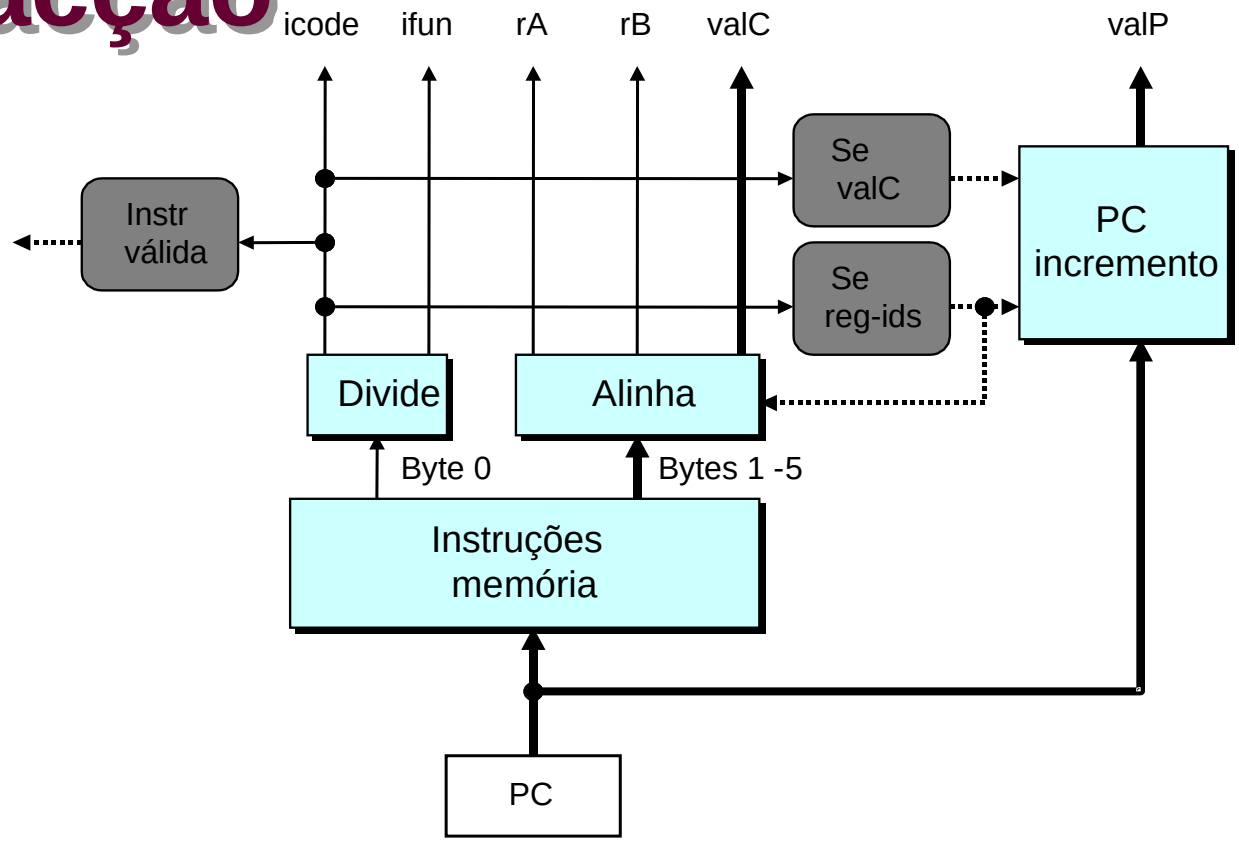
Lógica - Extracção



Blocos Pré-definidos

- **PC: Registo com contador de programa**
- **Memória de Instruções: Leitura de 6 octetos (PC to PC+5)**
- **Divide: Divide a instrução (octeto) em *icode* e *ifun***
- **Alinha: Obtém os campos para *rA*, *rB*, e *valC***

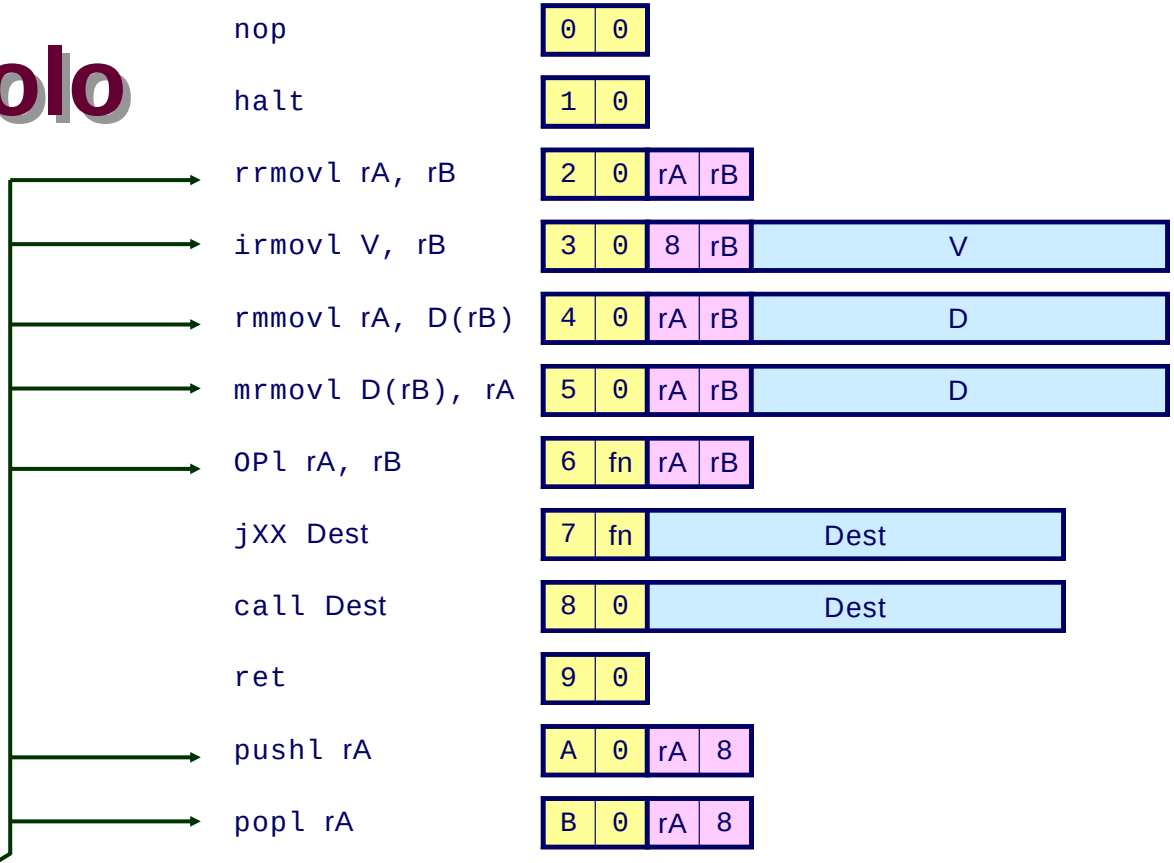
Lógica - Extracção



Lógica Controlo

- **Instr. válida:**
  - **Esta instrução é válida?**
- **Se regids:**
  - **Esta instrução referencia registos?**
- **Se valC:**
  - **Esta instrução refere valores constantes?**

Lógica de Controlo Extracção



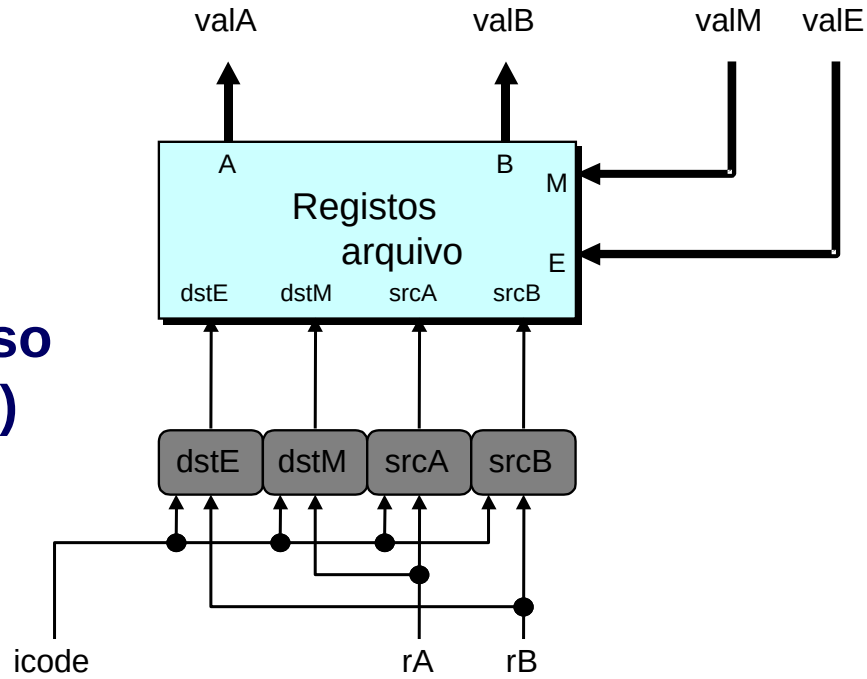

```
bool need_regids =
       icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
                  IIRMOVL, IRMMOVL, IMRMOVL };

bool instr_valid = icode in
       { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
              IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
```

Lógica Descodificação

Arquivo de Registos

- Lê portas A, B
- Escreve portas E, M
- Endereços são IDs para acesso a registos (8 para não-acesso)



Lógica de Controlo

- srcA, srcB
  - lê endereço da porta
- dstA, dstB
  - escreve no endereço da porta

Origem - A

	OPl rA, rB	
Descodifica	valA ← R[rA]	Lê operando A
	`rmmovl` rA, D(rB)	
Descodifica	valA ← R[rA]	Lê operando A
	`popl` rA	
Descodifica	valA ← R[%esp]	Lê apontador pilha
	jXX Dest	
Descodifica		Sem operando
	`call` Dest	
Descodifica		Sem operando
	`ret`	
Descodifica	valA ← R[%esp]	Lê apontador pilha

```
int srcA = [
        icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : rA;
        icode in { IPOPL, IRET } : RESP;
        1 : RNONE; # Don't need register ];
```

Destino E

	OPl rA, rB	
Actualiza	R[rB] ← valE	Actualiza com resultado
	`rmmovl` rA, D(rB)	
Actualiza		Não faz nada
	`popl` rA	
Actualiza	R[%esp] ← valE	Actualiza apont. pilha
	jXX Dest	
Actualiza		Não faz nada
	`call` Dest	
Actualiza	R[%esp] ← valE	Actualiza apont. pilha
	`ret`	
Actualiza	R[%esp] ← valE	Actualiza apont. pilha

```
int dstE = [
        icode in { IRRMOVL, IIRMOVL, IOPL} : rB;
        icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE;  # Don't need register];
```

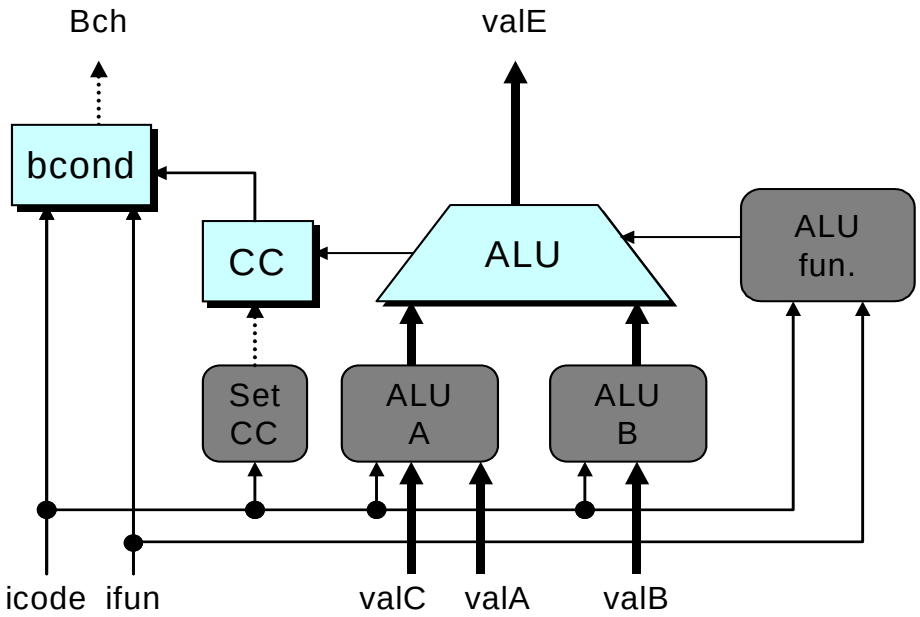
Lógica - Execução

Unidades

- ALU
  - Executa 4 funções
  - Gera códigos de condição
- CC
  - Registo com 3 bits de condições
- *bcond*
  - Calcula bit de salto

Lógica de controlo

- Ajusta CC
  - É necessário actualizar os códigos de condição?
- ALU A
  - Entrada A para a ALU
- ALU B
  - Entrada B para a ALU
- ALU fun
  - Qual a função efectuada pela ALU?



ALU – Entrada A

	OPl rA, rB	
Executa	valE ← valB OP valA	Efectua operação ALU
	`rmmovl` rA, D(rB)	
Executa	valE ← valB + valC	Calcula endereço efectivo
	`popl` rA	
Executa	valE ← valB + 4	Incrementa apont. pilha
	jXX Dest	
Executa		Não faz nada
	`call` Dest	
Executa	valE ← valB + –4	Decrementa apont. pilha
	`ret`	
Executa	valE ← valB + 4	Incrementa apont. pilha

```
int aluA = [
        icode in { IRRMOVL, IOPL } : valA;
        icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
        icode in { ICALL, IPUSHL } : -4;
        icode in { IRET, IPOPL } : 4;
        # Other instructions don't need ALU];
```

ALU - Operação

	OPl rA, rB	
Executa	valE ← valB OP valA	Efectua operação ALU

	`rmmovl` rA, D(rB)	
Executa	valE ← valB + valC	Calcula endereço efectivo

	`popl` rA	
Executa	valE ← valB + 4	Incrementa apont. pilha

	jXX Dest	
Executa		Não faz nada

	`call` Dest	
Executa	valE ← valB + –4	Decrementa apont. pilha

	`ret`	
Executa	valE ← valB + 4	Incrementa apont. pilha

```
int alufun = [
        icode == IOPL : ifun;
        1 : ALUADD;];
```

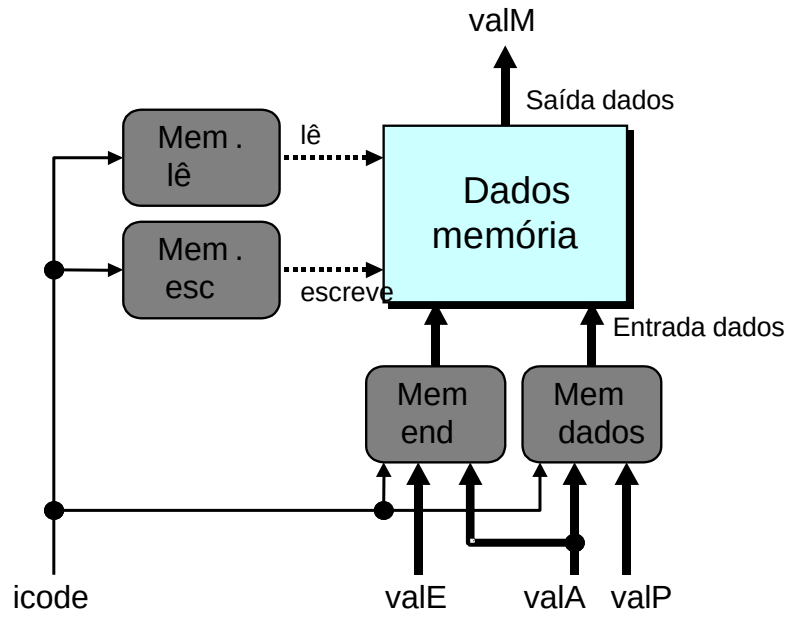
Lógica Memória

Memória

- Lê/escreve palavras

Lógica Controlo

- Mem. esc
  - deve ser lida uma palavra?
- Mem. lê
  - Deve ser escrita uma palavra?
- Mem. end
  - Selecciona endereço
- Mem. Dados
  - Selecciona dados



Memória – Endereçamento

	OPl rA, rB	
Memória		Não faz nada

	`rmmovl` rA, D(rB)	
Memória	M_4[valE] ← valA	Escreve valor na memória

	`popl` rA	
Memória	valM ← M_4[valA]	Lê valor da pilha

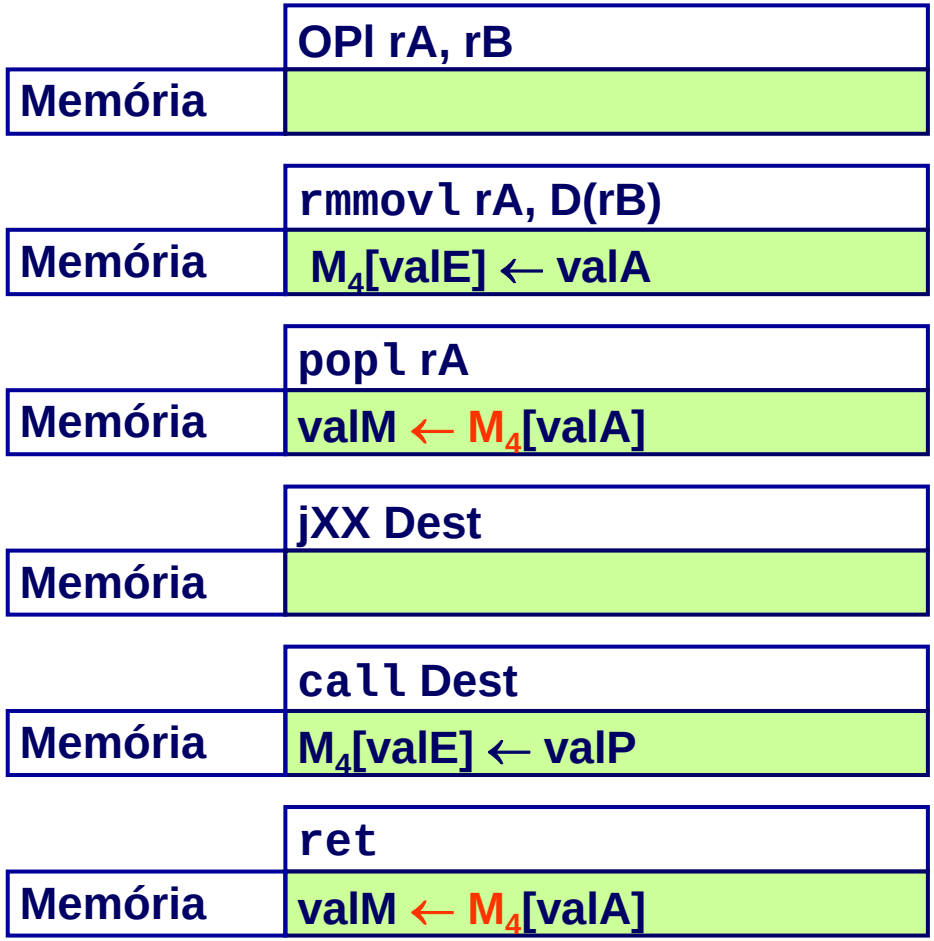
	jXX Dest	
Memória		Não faz nada

	`call` Dest	
Memória	M_4[valE] ← valP	Escreve valor de retorno na memória

	`ret`	
Memória	valM ← M_4[valA]	Lê endereço retorno

```
int mem_addr = [
        icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
        icode in { IPOPL, IRET } : valA;
        # Other instructions don't need address];
```

Memória - Leitura

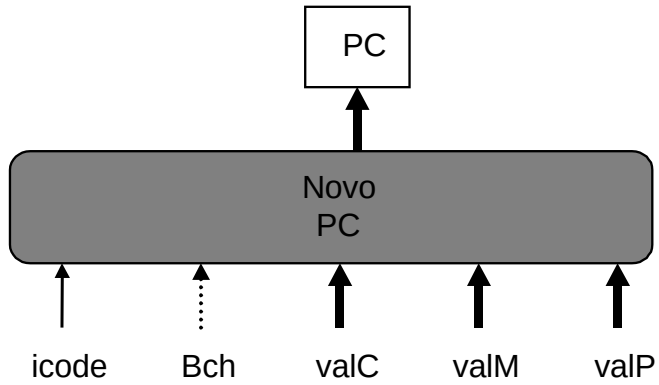



```
bool mem_read = icode in { IMRMOVL, IPOPL, IRET };
```

PC – Lógica para actualização

PC novo

- Determina o novo valor do PC



PC Actualiza

	OPl rA, rB	
PC	PC ← valP	Actualiza PC
	`rmmovl` rA, D(rB)	
PC	PC ← valP	Actualiza PC
	`popl` rA	
PC	PC ← valP	Actualiza PC
	jXX Dest	
PC	PC ← Bch ? valC : valP	Actualiza PC
	`call` Dest	
PC	PC ← valC	Ajusta PC para destino
	`ret`	
PC	PC ← valM	Ajusta PC para destino

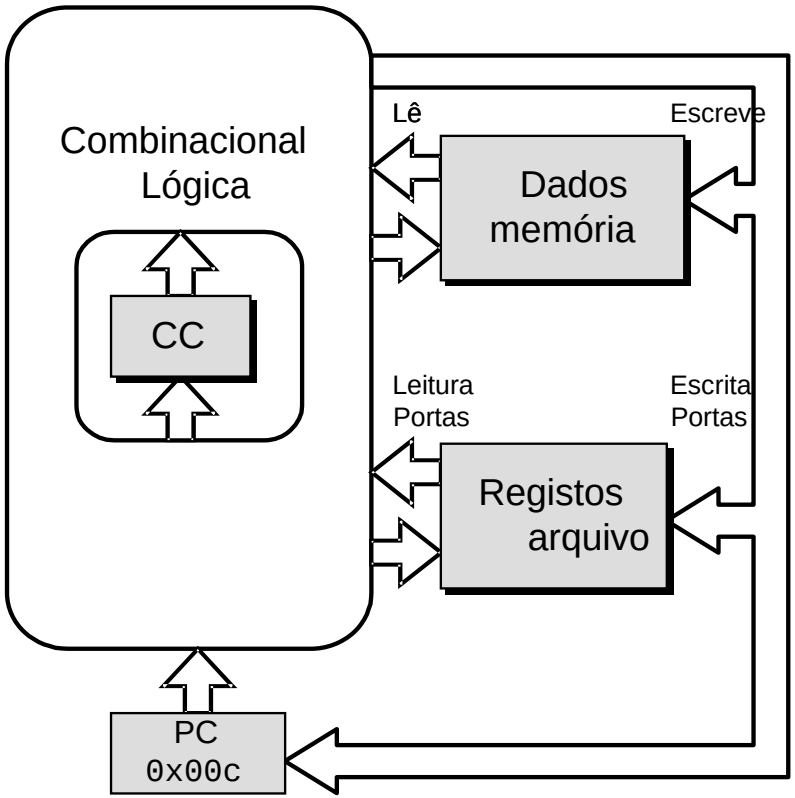
```
int new_pc = [
        icode == ICALL : valC;
        icode == IJXX && Bch : valC;
        icode == IRET : valM;
        1 : valP;];
```

SEQ – Traçado de instruções

```
1    0x000:    irmovl $0x100,%ebx   # %ebx <-- 0x100
2    0x006:    irmovl $0x200,%edx   # %edx <-- 0x200
3    0x00c:    addl %edx,%ebx       # %ebx <-- 0x300 CC <-- 000
4    0x00e:    je dest              # Não Segue
5    0x013:    rmmovl %ebx,0(%edx)  # M[0x200] <-- 0x300
6    0x019: dest: halt
```

- **Processamento**
  - **CC=100 (ZF,SF,OF);**
  - **addl %edx, %ebx; je dest**

SEQ - *Hardware*



Estado

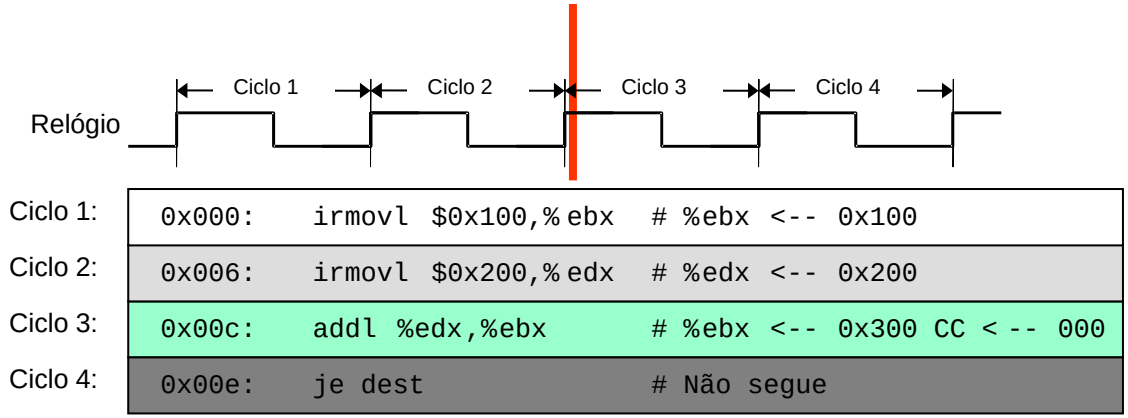
- Registo PC
- Registo de códigos condição
- Memória de dados
- Registos em arquivo

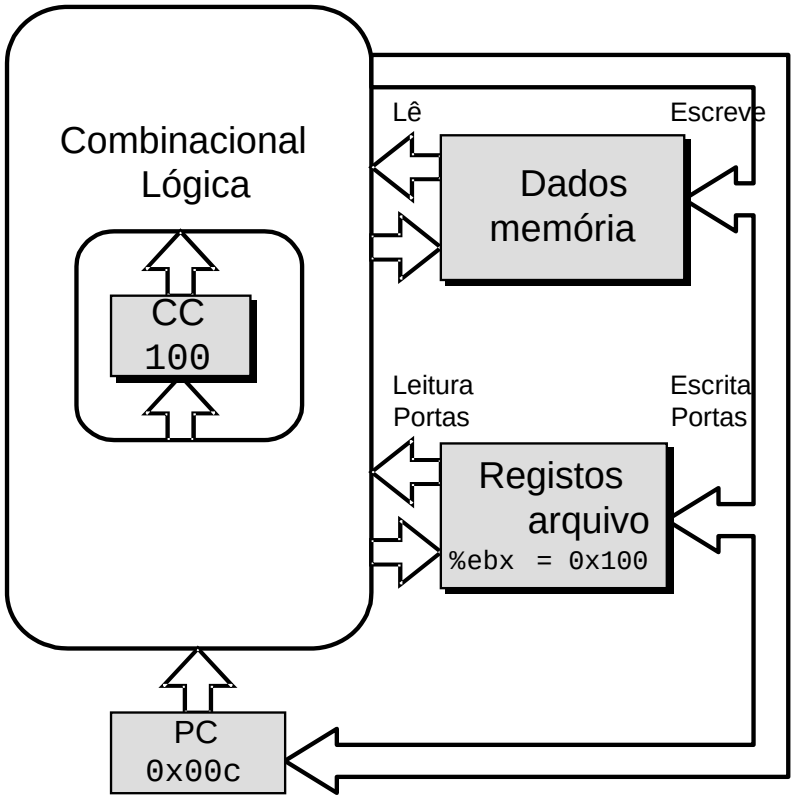
Actualizados à subida do impulso

Lógica combinacional

- ALU
- Lógica de Controlo
- Leituras de Memória
  - Instruções
  - Arquivo de registos
  - Dados

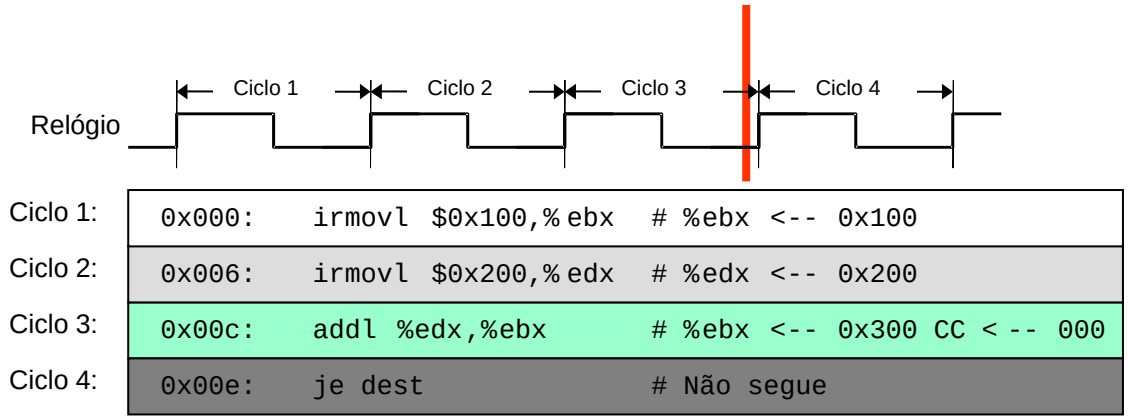
SEQ Operação #2

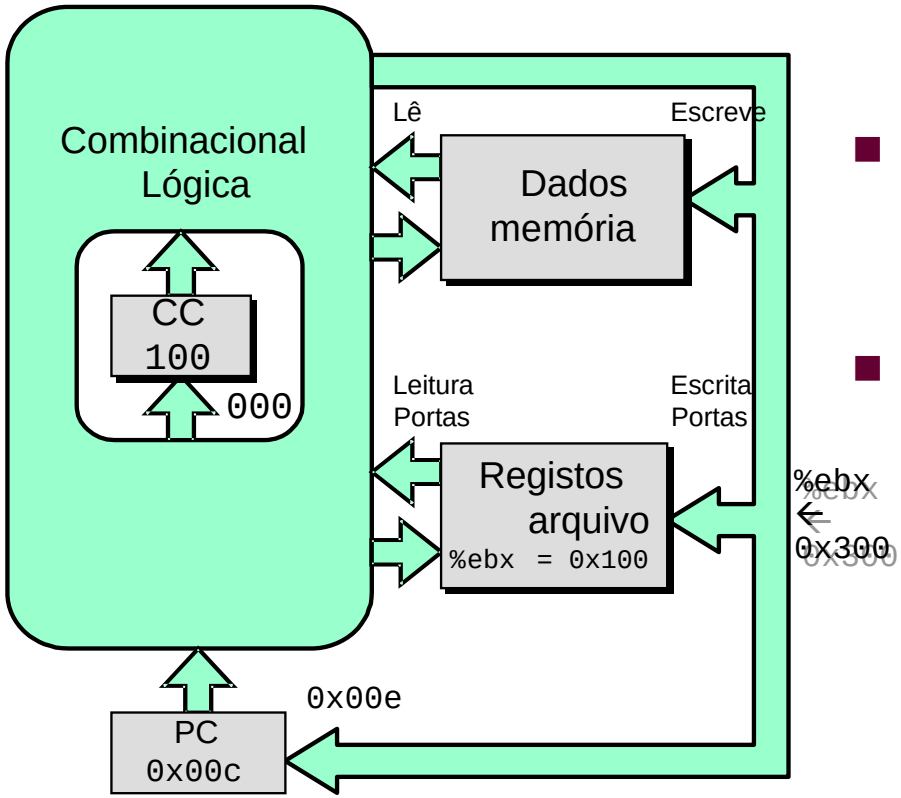




- **Ajuste do estado de acordo com a 2ª instrução `irmovl`**
- **Lógica combinacional começa a reagir à alteração do estado**

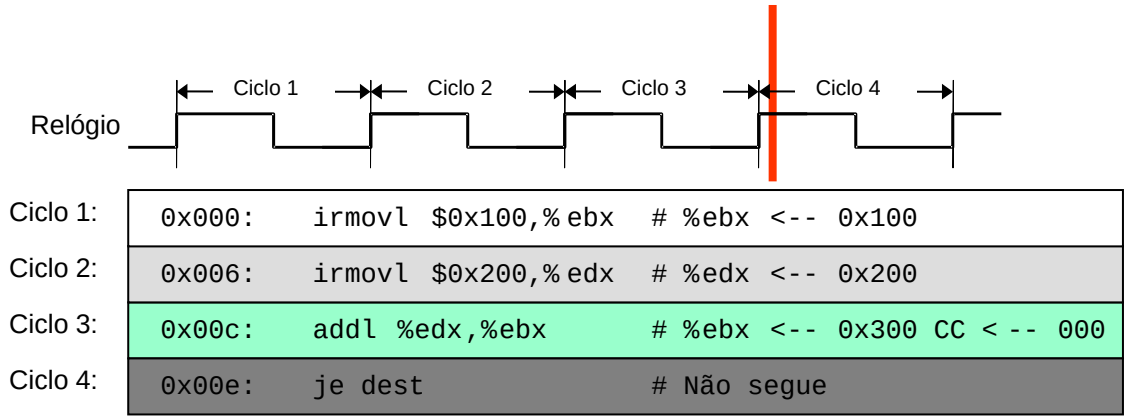
SEQ Operação #3

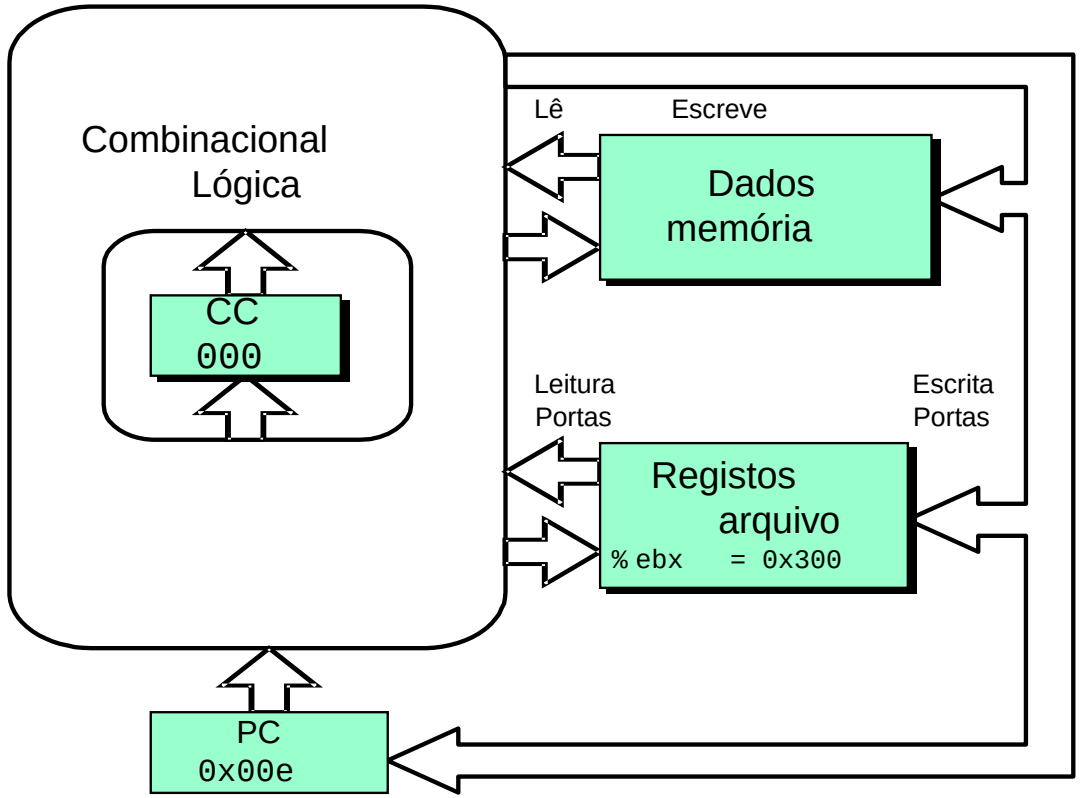




- **Ajuste do estado de acordo com a 2ª instrução `irmovl`**
- **Lógica combinacional gera resultados para a instrução `addl`**

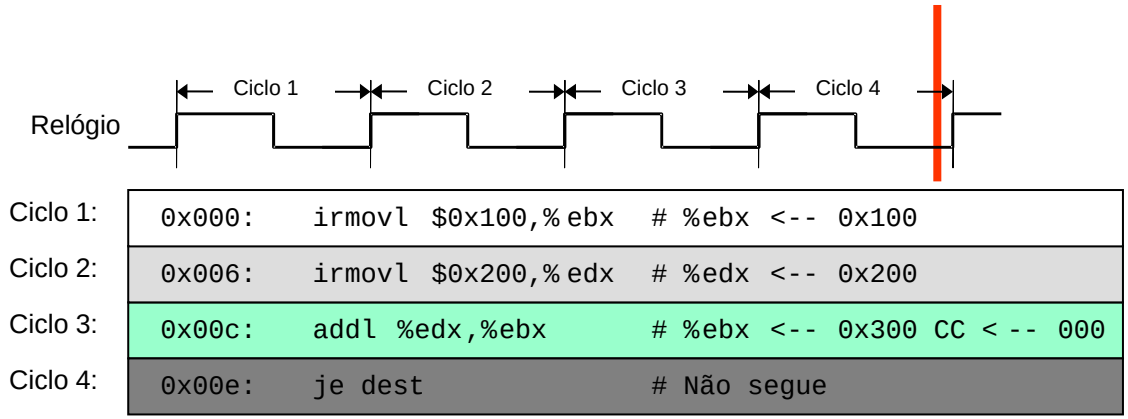
SEQ Operação #4

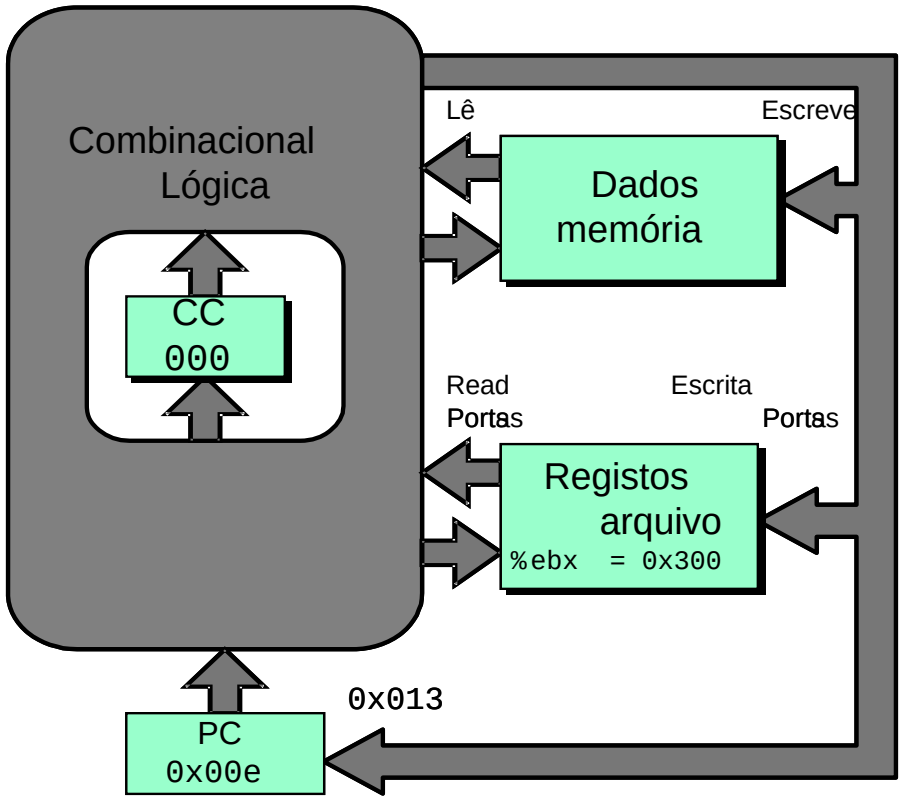




- **Ajuste do estado de acordo com a instrução `addl`**
- **Lógica combinacional começa a reagir à alteração do estado**

SEQ Operation #5





- **Ajuste do estado de acordo com a instrução `addl`**
- **Lógica combinacional gera resultados para a instrução `je`**

SEQ - Discussão

Realização

- Toda a instrução consiste numa sequência de passos simples
- Segue o modelo geral de fluxo em cada tipo de instrução
- Agrupa registos, memória e blocos combinacionais pré-desenhados
- Conexão através de lógica de controlo

Limitações

- Demasiado lenta para ser praticável
- Um único ciclo para propagação através da memória de instruções, do arquivo de registos, ALU, e da memória de dados
- Seria necessário um relógio extremamente lento
- As unidades de *Hardware* estariam activas apenas durante uma fracção do ciclo de relógio

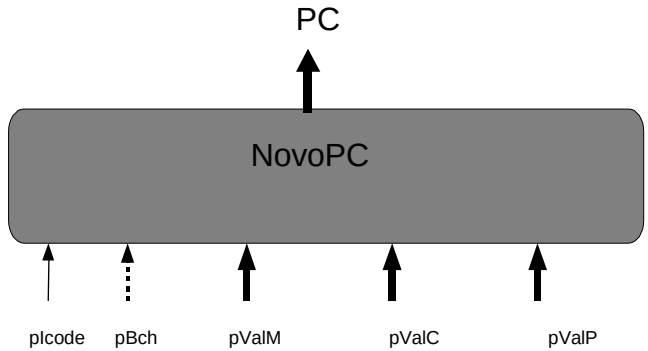
SEQ+ Rearranjo de estágios

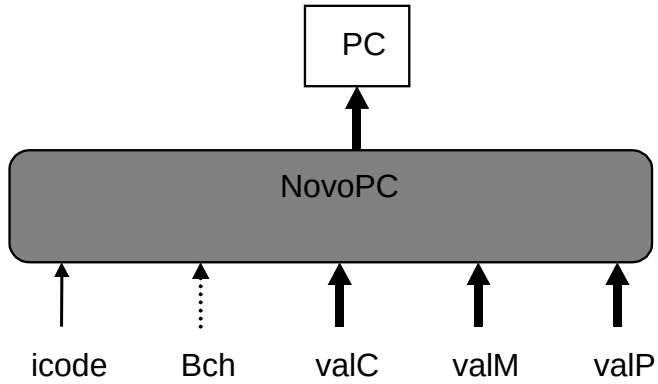
Cálculo PC

- SEQ :último estágio da instrução anterior
- SEQ+ :no início da instrução corrente

Hardware

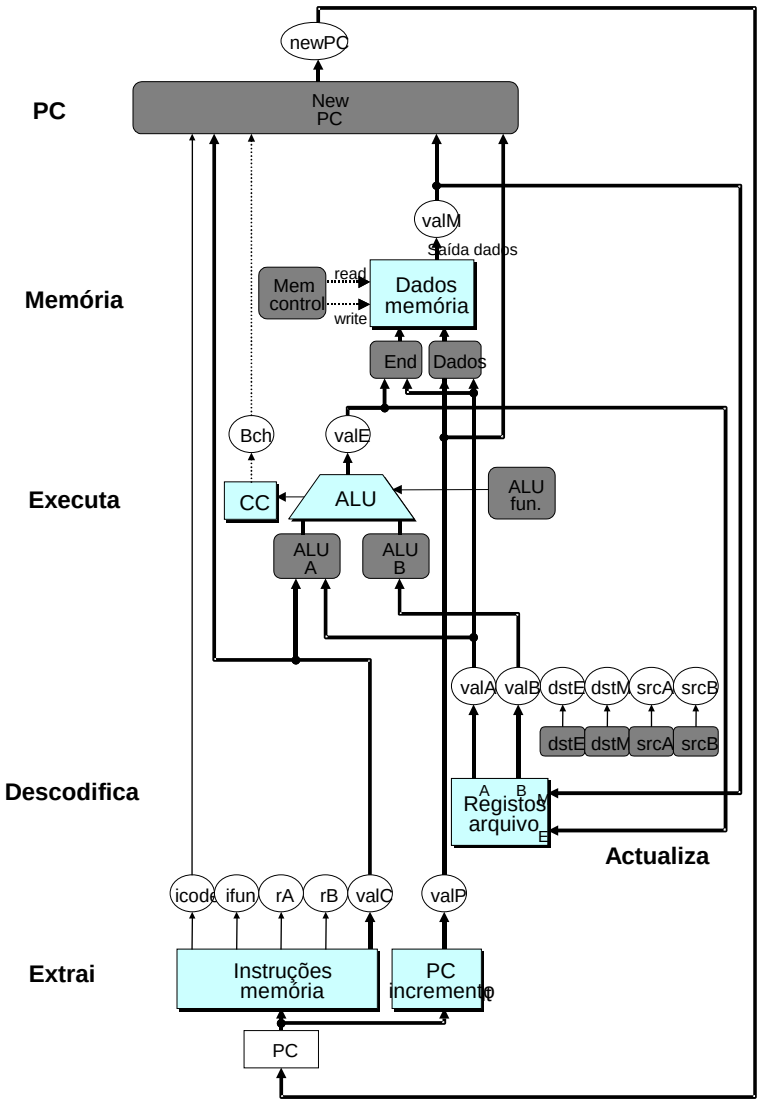
- Mesmas unidades e blocos de controlo
- SEQ+
  - Registos armazenam estado anterior (p)

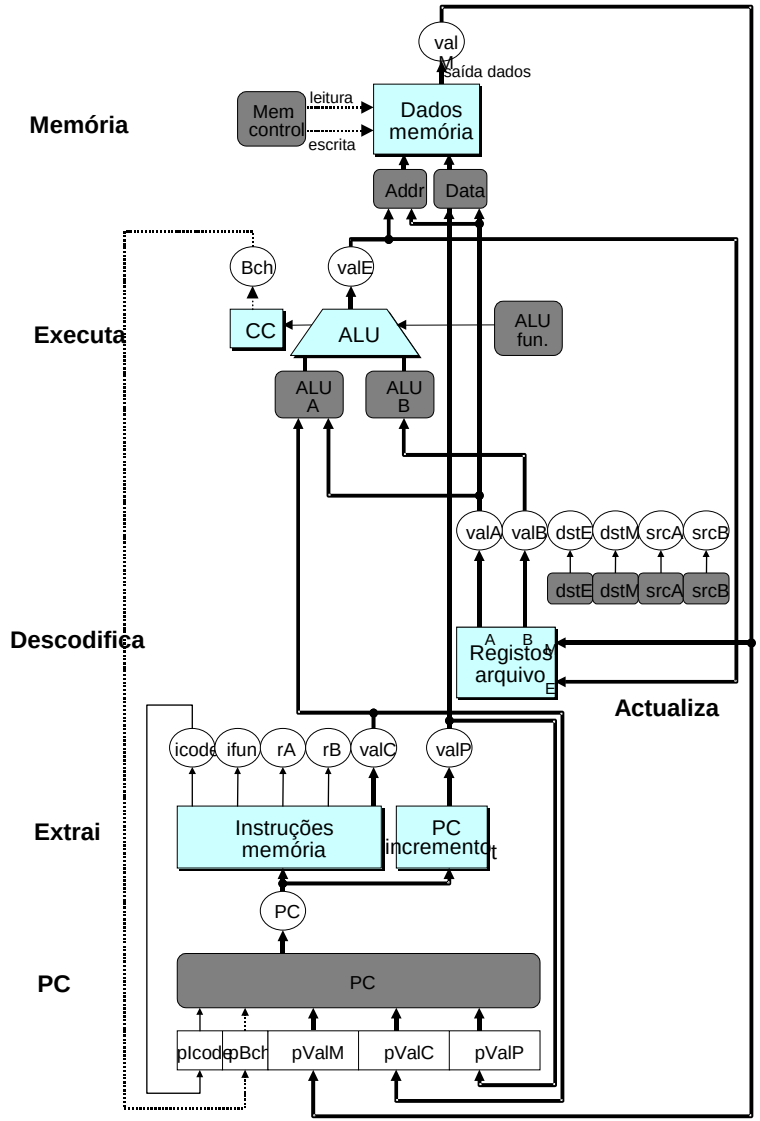





```
int pc = [
        # Call.  Usa constante na instrução
        pIcode == ICALL : pValC;
        # Segue salto. Usa constante na instrução
        pIcode == IJXX && pBch : pValC;
        # Terminação da instrução RET.  Usa valor da pilha
        pIcode == IRET : pValM;
        # Omissão: Usa valor do PC actualizado
        1 : pValP;
];
```

SEQ/SEQ+


newPC
PC
New PC
valM
Saída dados
Memória
Mem control
read
write
Dados memória
End
Dados
Bch
valE
Executa
CC
ALU
ALU fun.
ALU A
ALU B
valA
valB
dstE
dstM
srcA
srcB
Descodifica
Registos arquivo
Actualiza
icode
ifun
rA
rB
valC
valP
Extrai
Instruções memória
PC incrementor
PC


valM
Saída dados
Memória
Mem control
leitura
escrita
Dados memória
Addr
Data
Bch
valE
Executa
CC
ALU
ALU fun.
ALU A
ALU B
valA
valB
dstE
dstM
srcA
srcB
Descodifica
Registos arquivo
Actualiza
icode
ifun
rA
rB
valC
valP
Extrai
Instruções memória
PC incrementor
PC
PC
pIcode
pBch
pValM
pValC
pValP