Módulo 6 :: AC :: LEI

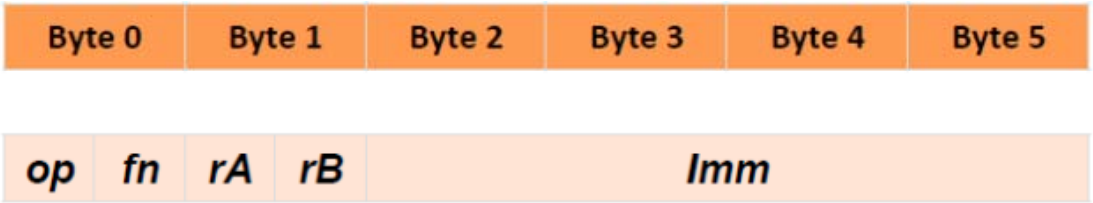
Y86 Sequencial

05 de Novembro 2013

Instruction **S**et **A**rchitecture (ISA) do Y86

Instrução	Octetos	Comentários
`nop`	1	Nenhuma operação
`halt`	1	Parar execução
`rrmovl rA, rB`	2	Mover conteúdo de registo **rA** para registo **rB**
`irmovl V, rB`	6	Mover valor imediato **V** para registo **rB**
`rmmovl rA, D(rB)`	6	Mover conteúdo de **rA** para o endereço de memória **rB+D**
`mrmovl D(rB), rA`	6	Mover o conteúdo da posição de memória **rb+D** para **rA**
`addl rA, rB`	2	Adicionar **rB** com **rA** colocando o resultado em rB
`subl rA, rB`	2	A **rB** subtrair **rA**, colocando o resultado em **rB**
`andl rA, rB`	2	Conjunção de **rA** com **rB**, resultado em **rB**
`xorl rA, rB`	2	Disjunção exclusiva de **rA** com **rB**, resultado em **rB**
`jmp Dest`	5	Salto incondicional para **Dest**
`jle Dest`	5	Salto se menor ou igual (SF=1 ou ZF=1) para `Dest`
`jl Dest`	5	Salto se menor (SF=1) para `Dest`
`je Dest`	5	Salto se igual (ZF=1) para `Dest`
`jne Dest`	5	Salto se diferente (ZF=0) para `Dest`
`jge Dest`	5	Salto se maior ou igual (SF=0 ou ZF=1) para `Dest`
`jg Dest`	5	Salto se maior (SF=0) para `Dest`
`call Dest`	5	Salta para `Dest`, guarda o endereço de retorno no topo da pilha
`ret`	1	Salta para o endereço que se encontra no topo da pilha
push *rA*	**2**	**Decrementa %esp e depois guarda o conteúdo de *rA* na pilha**
popl *rA*	**2**	**Lê da pilha para *rA* e depois incrementa %esp**

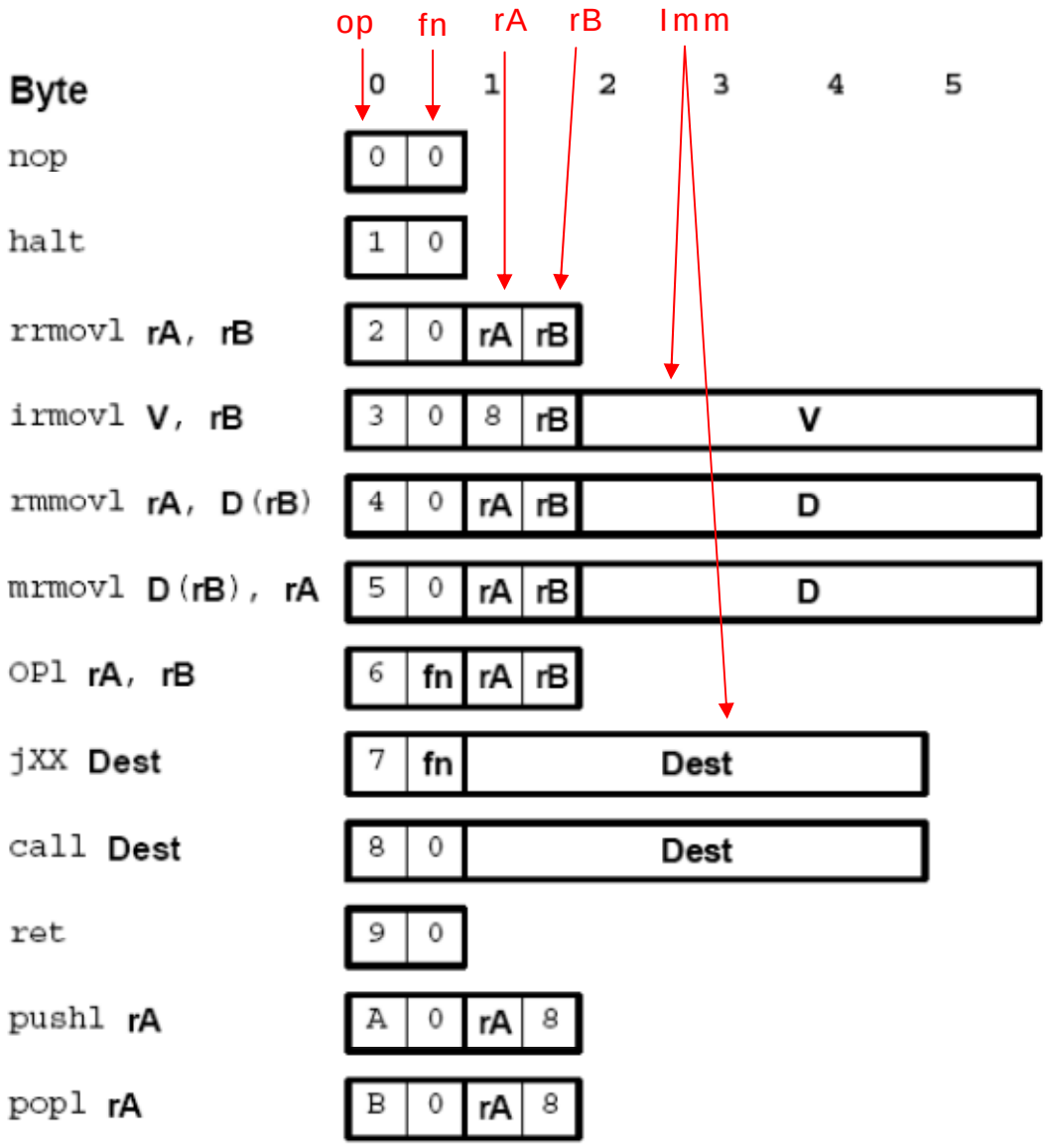
- 8 **registos** de 32 bits: %EAX, %EBC, %ECX, %EDX, %ESI, %EDI, %ESP, %EBP.
- 3 **flags**: **OF** – resultado com *overflow*, **ZF** – resultado nulo, **SF** – resultado com sinal.
- registo **PC**: contém o endereço da próxima instrução a executar.
- **valores imediatos**, **deslocamentos** e **endereços**: ocupam 4 bytes em formato *little endian* (byte menos significativo primeiro/no endereço menor).
- **Formato das instruções** (1 a 6 bytes):



- ***op*** → código de operação (*opcode*): identifica a instrução
- ***fn*** → função: identifica a operação

} existem em todas as instruções

- ***rA, rB*** → indicam quais os registos a utilizar
- ***Imm*** → valor imediato (constante)



rA, rB:

%eax	0 (**0000**)	%esp	4 (**0100**)
%ecx	1 (**0001**)	%ebp	5 (**0101**)
%edx	2 (**0010**)	%esi	6 (**0110**)
%ebx	3 (**0011**)	%edi	7 (**0111**)

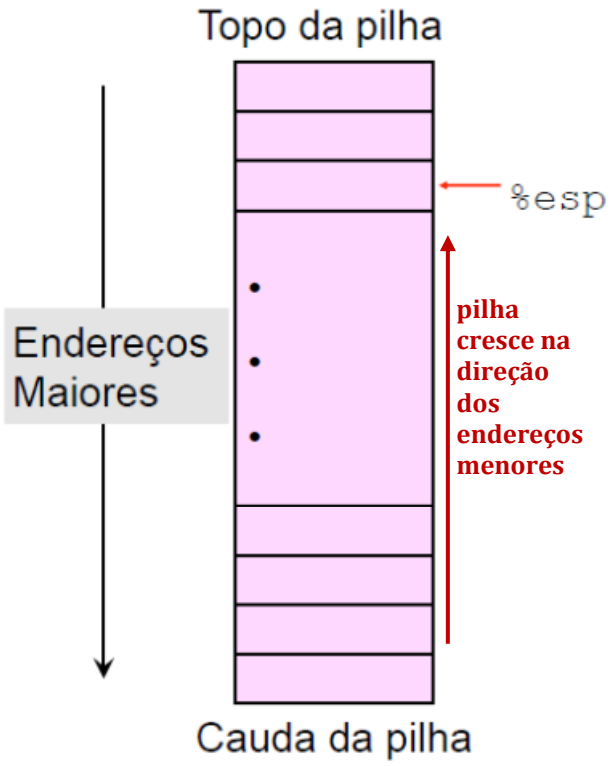
Quando o campo correspondente a **rA** ou **rB** não é usado → usa-se o código **1000**

OPl	op	fn
addl	0110	0000
subl	0110	0001
andl	0110	0010
xorl	0110	0011

MOV	op	fn
rrmov	0010	0000
irmov	0011	0000
rmmov	0100	0000
mrmov	0101	0000

Branch	op	fn
jmp	0111	0000
jle	0111	0001
jl	0111	0010
je	0111	0011
jne	0111	0100
jge	0111	0101
jg	0111	0110

Nos **MOV**'s existe apenas um modo endereçamento: **base+deslocamento**.
Nas operações aritméticas e lógicas (**OPl**) os operandos e resultado estão em registos.



- **%esp** aponta para o topo da pilha
- Pilha cresce na direção dos endereços menores:
  - Topo fica no endereço menor
  - **push** → primeiro subtrai 4 ao %esp, depois escreve na pilha
  - **pop** → lê da pilha e depois adiciona 4 ao %esp

Y86: Estrutura dos programas

- Os programas começam no endereço **0.**
- Tem que se inicializar a pilha **(%esp)** → cuidado para não sobrepor a pilha ao código.
- Tem que se inicializar os dados.
- Diretivas do *assembly* Y86:
  - **.pos address** → o código/dados que vierem a seguir são colocados a partir do endereço de memória ***address***
  - **.align Num** → alinhar os dados em múltiplos de ***Num*** bytes
  - **.long Imm** → reserva **4** bytes para um inteiro e inicializa-o com o valor ***Imm***

Para cada instrução do código *assembly* Y86 fornecido, assinalada com ***, apresente:

(a) A codificação da instrução, a sua disposição na memória, indicando o endereço onde está armazenada (lembre-se de que se trata de uma arquitetura *little endian*).

Endereço	Instrução	Codificação	Memória
	.pos 0		
	init:		
	irmovl stack, %esp # ***		
	irmovl stack, %ebp		
	jmp main # ***		
	main:		
	irmovl $4, %eax		
	Pushl %eax		
	irmovl data, %ebx # ***		
	mrmovl $4(%ebx), %eax		
	pushl %eax # ***		
	call soma # ***		
	halt		
	soma:		
	pushl %ebp		
	rrmovl %esp, %ebp # ***		
	mrmovl $8(%ebp), %eax		
	mrmovl $12(%ebp), %ebx # ***		
	addl %ebx, %eax # ***		
	rrmovl %ebp, %esp		
	popl %ebp # ***		
	ret # ***		
	.pos 0x100		
	data: .long 10		
	.long 24		
	.pos 0x200		
	stack: # Inicio da pilha		

1º passo - calcular o endereço de cada instrução e das etiquetas

- substituir cada etiqueta do programa pelo respetivo endereço

2º passo - codificar cada instrução

3º passo - mostrar a disposição dos vários bytes de cada instrução e dos dados em memória

Endereço	Instrução	Codificação	Memória
	.pos 0		
	init:		
0x000	irmovl 0x200, %esp # ***	30 84 00 00 02 00	30 84 00 02 00 00
0x006	irmovl 0x200, %ebp		
0x00C	jmp 0x011 # ***	70 00 00 00 11	70 11 00 00 00
	main:		
0x011	irmovl $4, %eax		
0x017	pushl %eax		
0x019	irmovl 0x100, %ebx # ***	30 83 00 00 01 00	30 83 00 01 00 00
0x01F	mrmovl $4(%ebx), %eax		
0x025	pushl %eax # ***	A0 08	A0 08
0x027	call 0x02D # ***	80 00 00 00 2D	80 2D 00 00 00
0x02C	halt		
	soma:		
0x02D	pushl %ebp		
0x02F	rrmovl %esp, %ebp # ***	20 45	20 45
0x031	mrmovl $8(%ebp), %eax		
0x037	mrmovl $12(%ebp), %ebx # ***	50 35 00 00 00 0C	50 35 0C 00 00 00
0x03D	addl %ebx, %eax # ***	60 30	60 30
0x03F	rrmovl %ebp, %esp		
0x041	popl %ebp # ***	B0 58	B0 58
0x043	ret # ***	90	90
	.pos 0x100		
0x100	data: .long 10		0A 00 00 00
0x104	.long 24		18 00 00 00
	.pos 0x200		
0x200	stack: # Inicio da pilha		

- **irmovl 0x200, %esp**
  codificação: 30 8 codigo_%esp Imm ⇔ 30 84 00 00 02 00
- **jmp 0x011**
  codificação:70 address ⇔ 70 00 00 00 11
- **irmovl 0x100, %ebx**
  codificação: 30 8 codigo_%ebx Imm ⇔ 30 83 00 00 01 00
- **pushl %eax**
  codificação: A0 codigo_%eax 8 ⇔ A0 08
- **call 0x02D**
  codificação: 80 address ⇔ 80 00 00 00 2D
- **rrmovl %esp, %ebp**
  codificação: 20 codigo_%esp codigo_%ebp ⇔ 20 45
- **mrmovl $12(%ebp), %ebx**
  codificação: 50 codigo_%ebx codigo_%ebp offset ⇔ 50 35 00 00 00 0C
- **addl %ebx, %eax**
  codificação: 60 codigo _%ebx codigo_%eax ⇔ 60 30
- **popl %ebp**
  codificação: B0 codigo_%ebp 8 ⇔ B0 58
- **ret**
  codificação: 90

Utilizar o simulador do Y86 para verificar se a codificação das instruções feita manualmente está correta e para executar o código máquina gerado:

- Copiar o código *assembly* da 1ª página do enunciado para um ficheiro → **soma.ys**
- Gerar código máquina para este ficheiro usando o programa *assembler* do Y86 **yas**:

  yas soma.ys

  que gera o ficheiro objeto **soma.yo**
- Comparar a codificação das instruções feita manualmente com o código gerado pelo **yas** (ficheiro **soma.yo**)
- Executar o código máquina gerado usando o simulador do processador Y86 (SEQ) **ssim**:

  copiar a script **/usr/local/sim/seq/seq.tcl** para a diretoria de trabalho

  ssim –g soma.yo

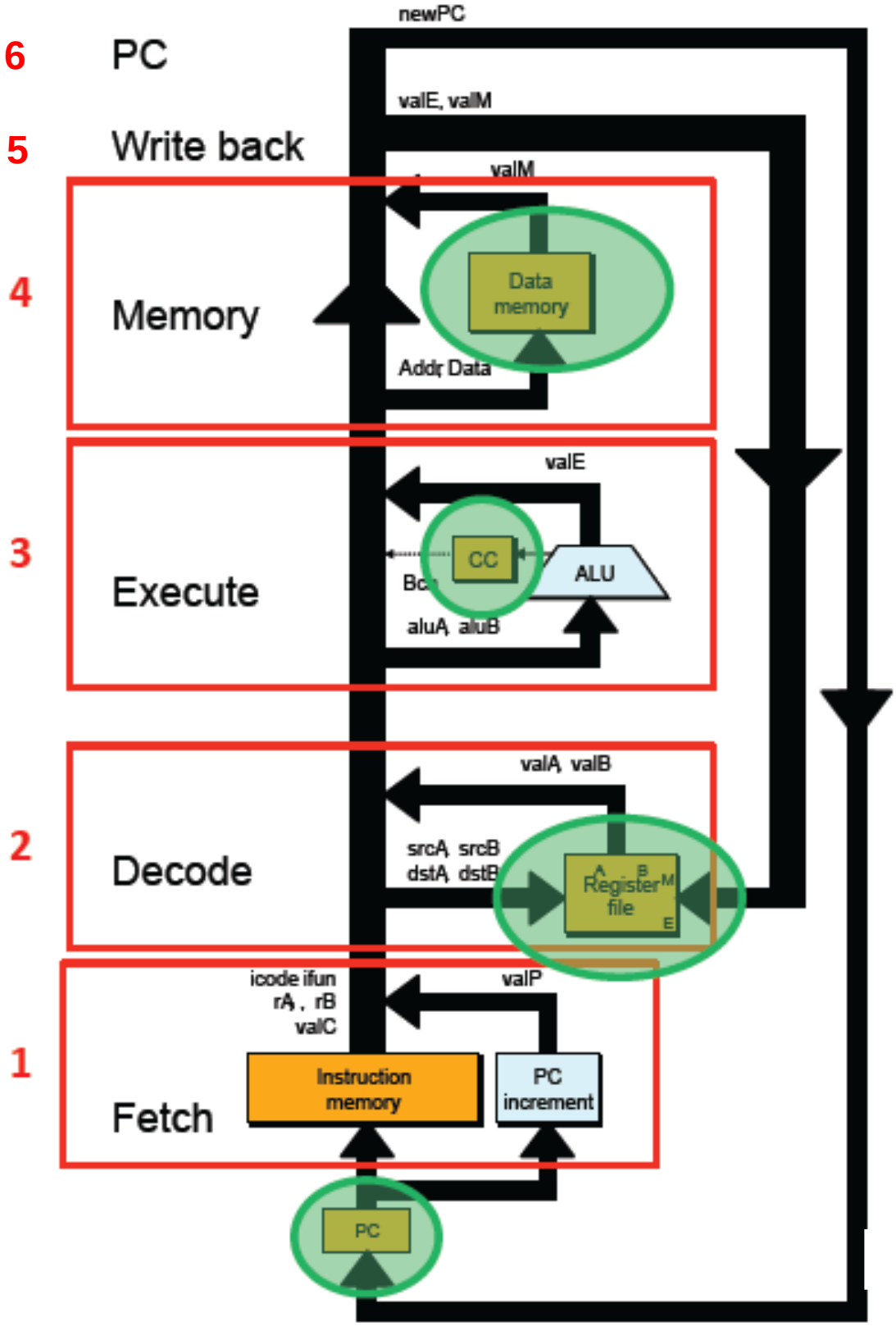
  com a opção **"-g"** selecionamos executar o simulador com interface gráfica (GUI).

Para cada instrução assinalada com ***, apresente:

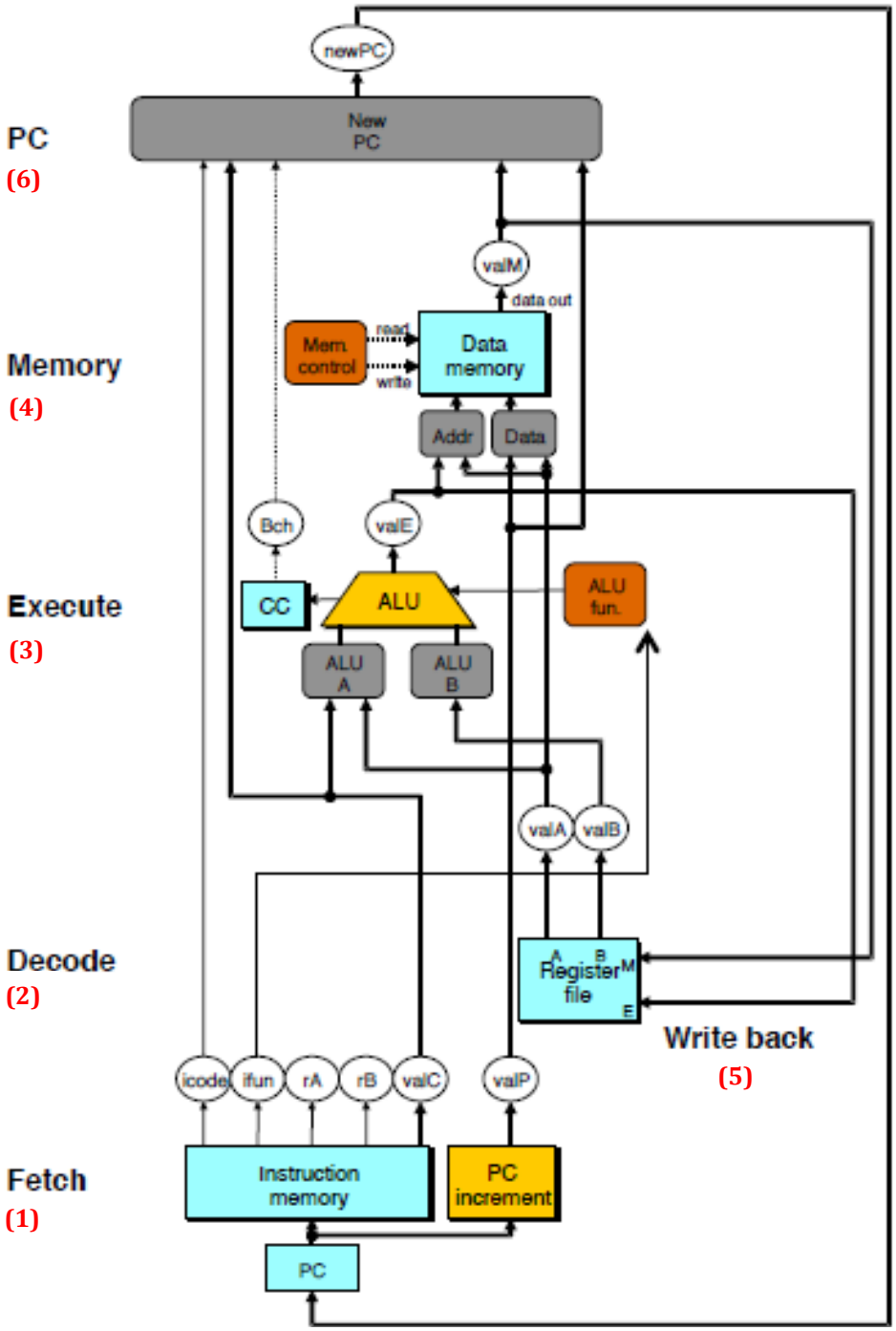
(b) Uma tabela com os valores dos sinais de controlo da organização sequencial. Apresente os sinais diferenciados pelo estágio em que são relevantes/gerados. Para cada tipo de instrução que surja pela primeira vez apresente os seus valores genéricos e os valores específicos para este programa.

Temporização da execução das instruções no Y86 SEQ:

1. Busca / extração
2. Descodificação
3. Execução
4. Leitura da memória
5. Escritas (as escritas ocorrem todas no fim do ciclo e em simultâneo):
   - Escrita nas *flags* (CC)
   - Escrita na memória
   - Escrita nos registos genéricos
   - (6) Escrita no PC.

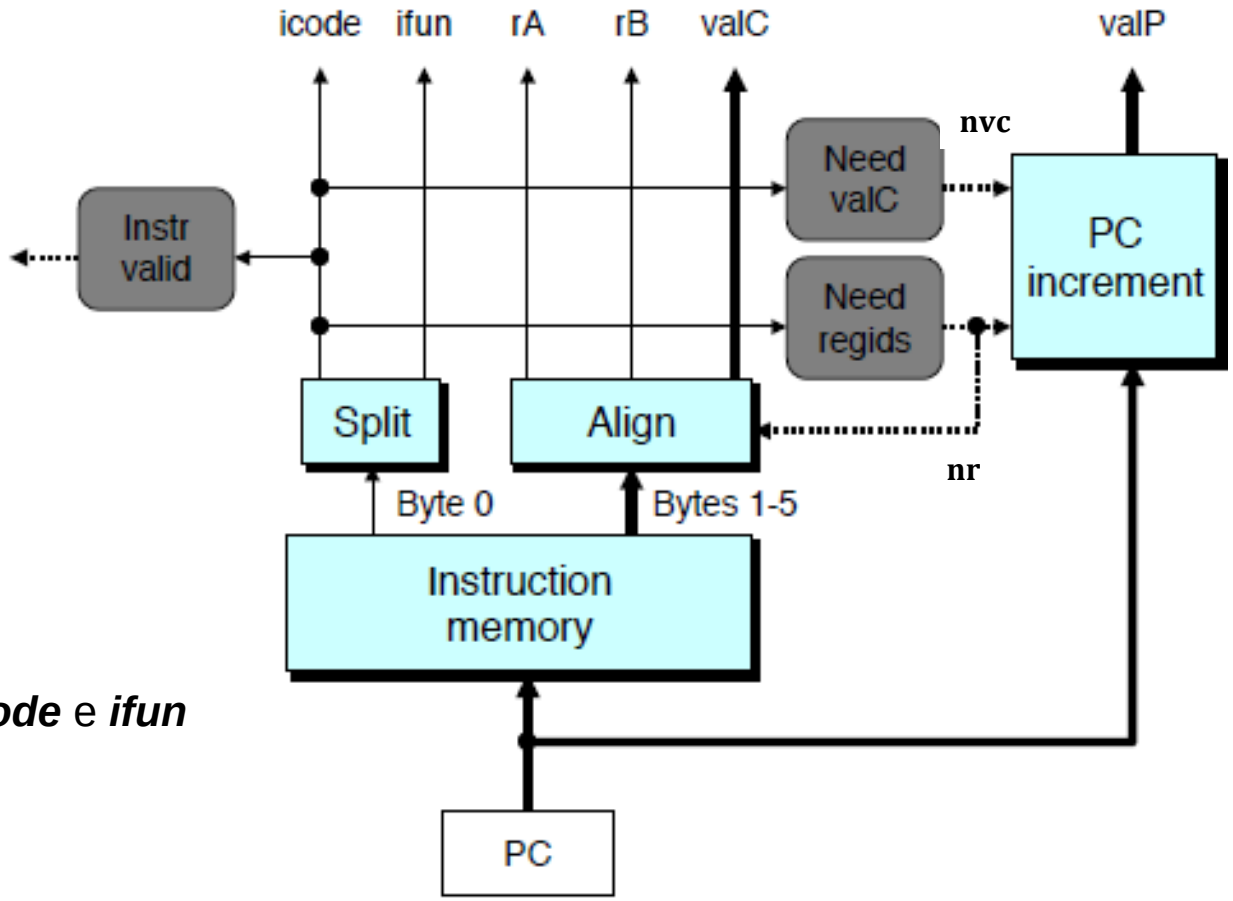

6
PC
newPC
5
Write back
valE, valM
valM
4
Memory
Data memory
Addr Data
3
Execute
valE
CC
ALU
aluA aluB
2
Decode
valA valB
srcA srcB
dstA dstB
Register file
1
Fetch
icode ifun
rA , rB
valC
valP
Instruction memory
PC increment
PC

- **Ovais brancas**: sinais internos
- **Caixas azuis**: lógica sequencial
- **Caixas cinzentas**: multiplexadores
- **Caixas laranja**: cálculo (ALU e "incrementar PC")
- **Caixas castanhas**: controlo
- **Traço grosso**: **32** bits
- **Traço fino**: **4** bits
- **Tracejado**: **1** bit



ARQUITETURA DO Y86 SEQUENCIAL

Y86: *Fetch*



Blocos

- **PC**: Registo
- **Memória Instruções**: Ler 6 bytes (PC→PC+5)
- **Split**: Dividir o Byte 0 em ***icode*** e ***ifun***
- **Align**: Obter rA, rB e valC
- -- **PC increment**: obter valP

Lógica de Controlo (sinais obtidos a partir de icode)

- **Instr. Valid**: esta instução é válida?
- **Need regids**: esta instrução tem os campos rA:rB?
- **Need valC**: esta instrução tem um valor imediato?

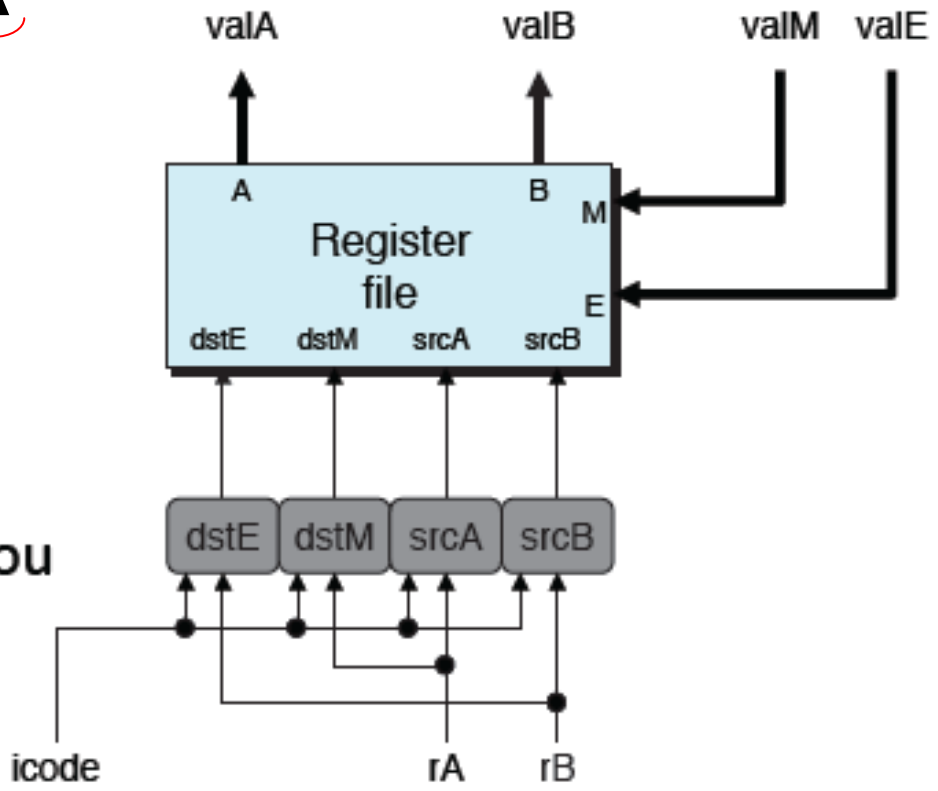
nr	valC
0	M[PC+1]
1	M[PC+2]

nvc	nr	valP (próximo PC)
0	0	PC+1
0	1	PC+2
1	0	PC+5
1	1	PC+6

Y86: *Decode* & *Write back*

2 5

- Banco de Registos
  - 2 – Ler portas A, B
  - 5 – Escrever portas E, M
  - – Endereços são os IDs do registos ou 8 (não aceder)



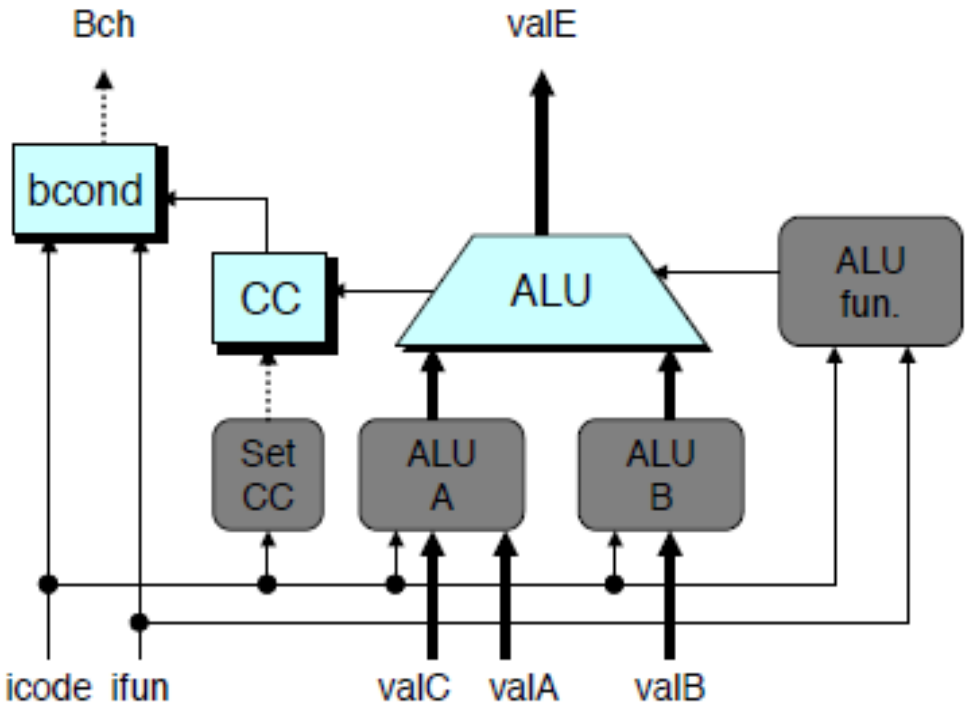
Lógica de Controlo

- 2 – srcA, srcB: registos a ler
- 5 – dstE, dstM: registos a escrever

Y86: *Execute*

Unidades

- ALU
  - Implementa 4 funções
  - Gera códigos de condição
- CC
  - Registo com 3 bits
- bcond
  - Calcular se o salto é tomado (*Bch)*



Lógica de Controlo

- **Set CC**: Alterar CC?
- **ALU A**: Entrada A para ALU
- **ALU B**: Entrada B para ALU
- **ALU fun**: Qual a função a calcular?

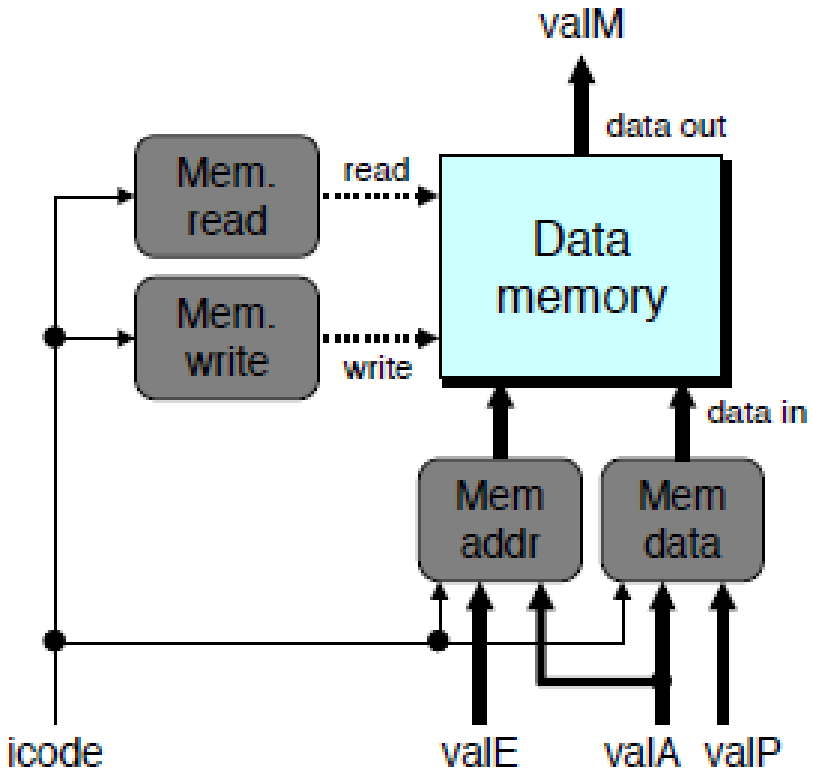
Y86: *Memory*

Memória

- Ler ou escrever uma palavra

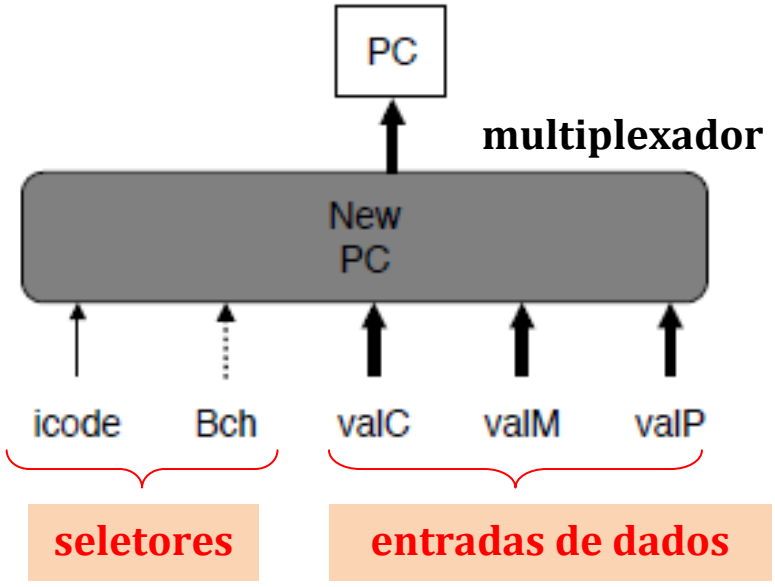
Lógica de Controlo

- **Mem.read:** leitura?
- **Mem.write**: escrita?
- **Mem.addr**: Selecciona addr
- **Mem.data**: Selecciona dados



- Novo PC
  - Selecciona próximo valor do PC

	OPl; `XXmovl; popl ; …`
PC update	PC ← valP
	jXX Dest
PC update	PC ← Bch ? valC : valP
	`call` Dest
PC update	PC ← valC
	`ret`
PC update	PC ← valM



Notas: **M1[add]** ⇔ 1 byte correspondente ao conteúdo da posição de memória ***add***
M4[add] ⇔ 4 bytes correspondentes ao conteúdo das posições de memória ***add*** a ***add+3***
R[codeR] ⇔ conteúdo do registo com código ***codeR***

irmovl stack, %esp

Código em memória: 30 84 00 02 00 00 (**6** bytes)
PC: 0x0

Estágio	**Genérico** **irmovl Imm, rb**	**Específico** **irmov 0x200, %esp**
Extração/busca	icode|ifun = M1[PC] (lê **6** bytes) rA|rB = M1[PC+1] valC = M4[PC+2] (porque nr=1) valP = PC+**6** (novo PC)	**icode|ifun** = M1[0] = **3 | 0** **rA|rB** = M1[1] = **8 | 4** **valC** = M4[2] = **0x200** **valP** = **0**+**6** = **6**
Descodificação	-------- (não há leitura de registos)	
Execução	valE = valC + 0 (passa a constante *Imm* para poder ser escrita nos registos)	**valE** = **0x200**
Memória (leitura)	-------- (não há leitura de memória)	--------
Atualização (escrita)	R[rB] = valE (escreve no registo *rb*)	R[**4**]=R[%esp]= **valE**= 0x200
PC (escrita)	PC = valP (atualiza o PC)	**PC** = valP = **6**

nr=1
nvc=1
valP=
=PC+6
=0+6

jmp main

Código em memória: 70 11 00 00 00 (**5** bytes)
PC: 0xC

Estágio	**Genérico** **jmp dest**	**Específico** **jmp 0x11**
Extração	icode|ifun = M1[PC] rA|rB ---------- valC = M4[PC+1] (porque nr=0) valP = PC + **5** (valor não usado)	**icode|ifun** = M1[0xC] = **7 | 0** ---------- **valC** = M4[0xD] = **0x11** **valP** = **0xC**+**5** = **0x11**
Descodificação		
Execução		
Memória		
Atualização		
PC	PC = valC	**PC** = **0x11**

nr=0
nvc=1
valP=
=PC+5
=0xC+5
=0x11

irmovl 0x100, %ebx

Código em memória: 30 83 00 01 00 00 (6 bytes)
PC = 0x19

Estágio	Genérico	Específico
	irmovl Imm, rb	**irmovl 0x100, %ebx**
Extração	icode|ifun = M1[PC] (lê **6** bytes) rA|rB = M1[PC+1] valC = M4[PC+2] valP = PC+**6** (novo PC)	**icode|ifun** = M1[0x19] = **3 | 0** **rA|rB** = M1[0x1A] = **8 | 3** **valC** = M4[0c1B] = **0x100** **valP** = **0x19**+**6** = **0x1F**
Descodificação		
Execução	valE = valC + 0	**valE** = **0x100**
Memória		
Atualização	R[rB] = valE	**%ebx** = valE = 0x100
PC	PC = valP	**PC** = valP = **0x1F**

pushl %eax

Código em memória: A0 08

Antes da instrução	Depois da instrução
%eax = 0x18	--------
%esp = 0x1FC	%esp = 0x1F8
PC = **0x25**	PC = 0x27

Estágio	Genérico	Específico
	push rA	**push %eax**
Extração	icode|ifun = M1[PC] rA|rB = M1[PC+1] valC ---------- valP = PC +2	**icode|ifun** = M1[0x25] = **A | 0** **rA|rB** = M1[0x26]= **0 | 8** --------- **valP** = **0x25**+2 = **0x27**
Descodificação	valA = R[rA] (lê registo de origem rA) valB = R[%esp] (lê registo implícito ESP)	**valA** = R[%eax] = **0x18** **valB** = R[%esp] = **0x1FC**
Execução	valE = valB + (-4) (calcula o novo topo da pilha)	**valE** = 0x1FC + (-4) = 0x1F8
Memória	M4[valE] = valA (escreve na pilha)	M4[0x1F8] = **0x18**
Atualização	R[%esp] = valE (escreve registo ESP)	R[%esp] = **0x1F8**
PC	PC = valP (atualiza o PC)	**PC** = **0x27**

nr=1
nvc=0

valP=
=PC+2
=0x25+2
=0x27

call 0x2D

(guarda o endereço de retorno na pilha = endereço do ***halt***)

Código em memória: **80 2D 00 00 00**

Antes da instrução	Depois da instrução
%esp = **0x1F8**	%esp = 0x1F4
PC = **0x27**	PC = 0x2D

Estágio	**Genérico**	**Específico**
	call Dest	**call 0x2D**
Extração	icode|ifun = M1[PC] valC = M4[PC+1] (nr=0) (endereço da rotina) valP = PC+5 (endereço de retorno)	**icode|ifun** = M1[0x27] = **8 | 0** **valC** = M4[0x28] = **0x2D** **valP** = 0x27+5 = **0x2C**
Descodificação	valB = R[%esp] (lê registo implícito ESP)	**valB** = R[%esp] = **0x1F8**
Execução	valE = valB+(**-4**) (calcula novo topo da pilha)	**valE** = valB + (**-4**) = **0x1F4**
Memória	M4[valE] = valP (escreve o endereço de retorno na pilha)	M4[0x1F4] = 0x2C
Atualização	R[%esp] = valE (atualiza topo da pilha)	R[%esp] = **0x1F4**
PC	PC = valC (atualiza PC com end. da rotina)	PC = **0x2D**

nr=0
nvc=1

valP=
=PC+5
=0x27+5
=0x2C

rrmovl %esp, %ebp

Código em memória: **20 45**

Antes da instrução	Depois da instrução
%esp = **0x1F0**	----------
%ebp = 0x200	%ebp = 0x1F0
PC = **0x2F**	PC = 0x31

Estágio	**Genérico**	**Específico**
	rrmovl rA, rB	**rrmovl rA, rB**
Extração	icode|ifun = M1[PC] rA|rB = M1[PC + 1] valP = PC+2 (novo PC)	**icode|ifun** = M1[0x2F] = **2 | 0** **rA|rB** = M1[0x30] = **4 | 5** **valP** = **0x2F**+2 = **0x31**
Descodificação	valA = R[rA] (lê registo origem rA)	**valA** = R[**4**] = R[%esp] = **0x1F0**
Execução	valE = 0 + valA (mover rA→rB implica passar pela ALU)	**valE** = **0x1F0**
Memória	-------	--------
Atualização	R[rB] = valE (escreve registo destino rB)	R[**5**] = R[%ebp] = **0x1F0**
PC	PC = valP (atualiza PC)	PC = **0x31**

nr=1
nvc=0

valP=
=PC+2
=0x2F+2
=0x31

mrmovl $12(%ebp), %ebx

Código em memória: 50 35 0C 00 00 00

Antes da instrução	Depois da instrução
%ebp = 0x1F0	----------
%ebx = 0x100	%ebx = 0x4
PC = 0x37	PC = 0x3D

Estágio	Genérico	Específico
	mrmovl D(rB), rA	**mrmovl $12(%ebp), %ebx**
Extração	icode|ifun = M1[PC] rA|rB = M1[PC+1] valC = M4[PC+2] (deslocamento) valP = PC+6 (novo PC)	**icode|ifun** = M1[0x37]= **5 | 0** **rA|rB** = M1[0x38]= **3 | 5** **valC** = M4[0x39] = **0x0C** **valP** = 0x37+6 = **0x3D**
Descodificação	valB = R[rB] (lê registo origem rB ⇔ base do endereço)	**valB** = R[**5**]=R[%ebp] =**0x1F0**
Execução	valE = valB + valC (base+deslocamento)	**valE** = **0x1F0** + **0xC** = **0x1FC**
Memória	valM = M4[valE] (lê memória base+desl)	**valM** = M4[0x1FC] = **0x4**
Atualização	R[rA] = valM (escreve no reg. destino rA)	R[**3**]=R[%ebx] = **0x4**
PC	PC = valP (atualiza PC)	PC = **0x3D**

add %ebx, %eax

(%eax=%eax+%ebx)

Código em memória: 60 30

Antes da instrução	Depois da instrução
%eax = 0x18	%eax =0x1C
%ebx = 0x4	----------
PC = 0x3D	PC = 0x3F

Estágio	Genérico	Específico
	add rA,rB (rB=rB+rA)	**add %ebx, %eax**
Extração	icode|ifun = M1[PC] rA|rB = M1[PC+1] valP = PC+2	**icode|ifun** = M1[0x3D] = **6 | 0** **rA|rB** = M1[0x3E] = **3 | 0** **valP** = 0x3D+2 = **0x3F**
Descodificação	valA = R[rA] (lê registo de origem rA) valB = R[rB] (lê registo de origem rB)	**valA** = R[**3**] = R[%ebx] = **0x04** **valB** = R[**0**] = R[%eax] = **0x18**
Execução	valE = valA + valB (soma operandos A e B)	**valE** = 0x04 + 0x18 = **0x1C**
Memória		
Atualização	R[rB] = valE (escreve registo destino rB)	R[**0**] = R[%eax] = **0x1C**
PC	PC = valP (atualiza PC)	PC = **0x3F**

pop %ebp

Código em memória: B0 58

Antes da instrução	Depois da instrução
%ebp = **0x1F0**	%ebp =0x200
%esp = **0x1F0**	%esp =0x1F4
PC = **0x41**	PC = 0x43

Estágio	**Genérico**	**Específico**
	pop rA	**pop %ebp**
Extração	icode|ifun = M1[PC] rA|rB = M1[PC+1] valP = PC+2	**icode|ifun** = M1[0x41] = **B | 0** **rA|rB** = M1[0x42] = **5 | 8** **valP** = 0x41+2 = **0x43**
Descodificação	valA = R[%esp] (lê reg. origem implícito) * valB = R[%esp] (lê reg. origem implícito) +	**valA** = R[%esp] = **0x1F0** **valB** = R[%esp] = **0x1F0**
Execução	valE = valB +4 (calcula novo topo da pilha)	**valE** = valB +4 = **0x1F4**
Memória	valM = M4[valA] (lê do topo da pilha)	**valM** = M4[**0x1F0**] = **0x200**
Atualização	R[%esp]=valE (atualiza o topo da pilha) R[rA] = valM (escreve registo destino rA)	R[%esp] = **0x1F4** R[**5**] = R[%ebp] = **0x200**
PC	PC = valP (atualiza PC)	PC = **0x43**

(*) valA é usado como endereço na leitura da pilha.
(+) valB é usado para calcular o novo valor do topo da pilha.

ret

Código em memória: 90

Antes da instrução	Depois da instrução
%esp = **0x1F4**	%esp =0x1F8
PC = **0x43**	PC = 0x2C

Estágio	**Genérico**	**Específico**
	ret	**ret**
Extração	icode|ifun = M1[PC] valP = PC+1	**icode|ifun** = M1[0x43] = **9 | 0** **valP** = 0x43+1 = 0x44
Descodificação	valA = R[%esp] (lê reg. origem implícito) * valB = R[%esp] (lê reg. origem implícito) +	**valA** = **0x1F4** **valB** = **0x1F4**
Execução	valE = valB + 4 (calcula novo topo pilha)	**valE** = **0x1F4**+4 = **0x1F8**
Memória	valM=M4[valA] (lê do topo da pilha ⇔e.r.)	**valM**=M4[**0x1F4**] = **0x2C**
Atualização	R[%esp]=valE (atualiza o topo da pilha)	R[%esp] = **0x1F8**
PC	PC = valM (atualiza PC com end. retorno)	PC = **0x2C**

PILHA

0x1E4		
0x1E8		
0x1EC		
0x1F0	**0x200**	(valor EBP salvaguardado)
0x1F4	**0x2C**	(endereço retorno ao main)
0x1F8	**0x18**	(2º PUSH do main)
0x1FC	**0x04**	(1º PUSH do main)
0x200		