Y86:
Datapath Sequencial

Arquitectura de Computadores

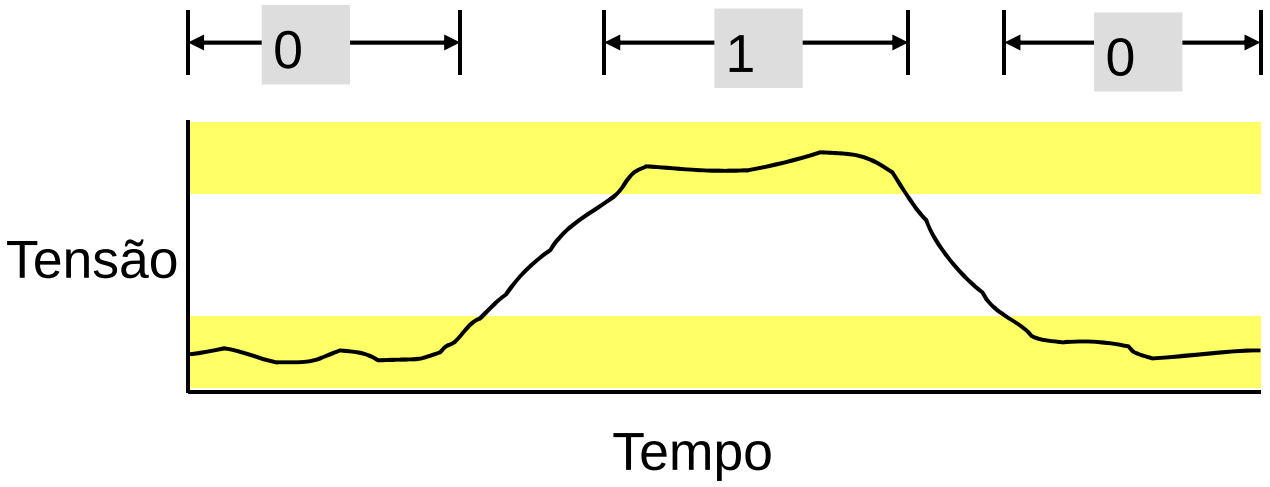
Lic. em Engenharia Informática

Luís Paulo Santos

Y86: *Datapath* Sequencial

Conteúdos	**9 – Organização do Processador**
	9.1 – Datapath sequencial
Resultados de Aprendizagem	R9.1 – Analisar e descrever organizações sequenciais de processadores elementares

Circuitos Lógicos



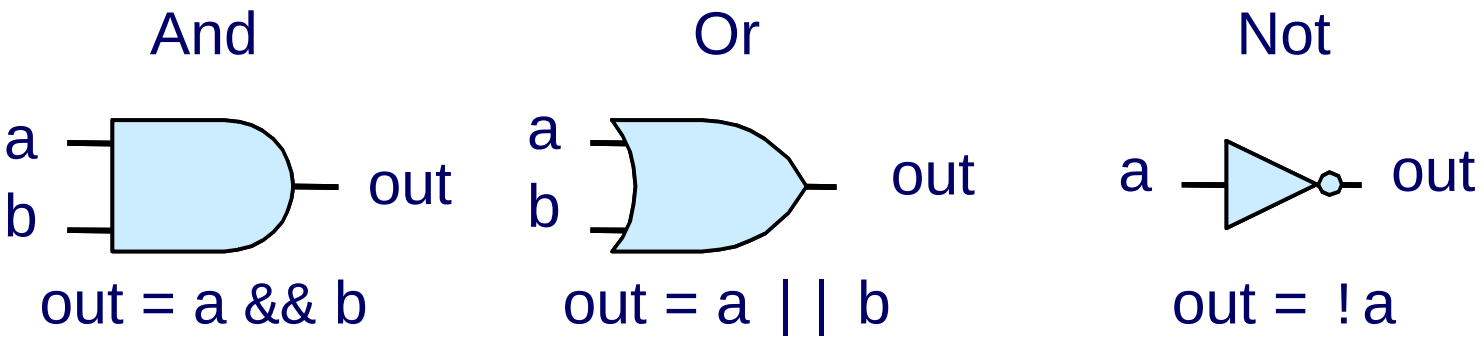
- Funcionam com valores discretos extraídos de um sinal contínuo
- Circuitos binários:
  - Cada sinal tem o valor 0 ou 1

Circuitos Lógicos

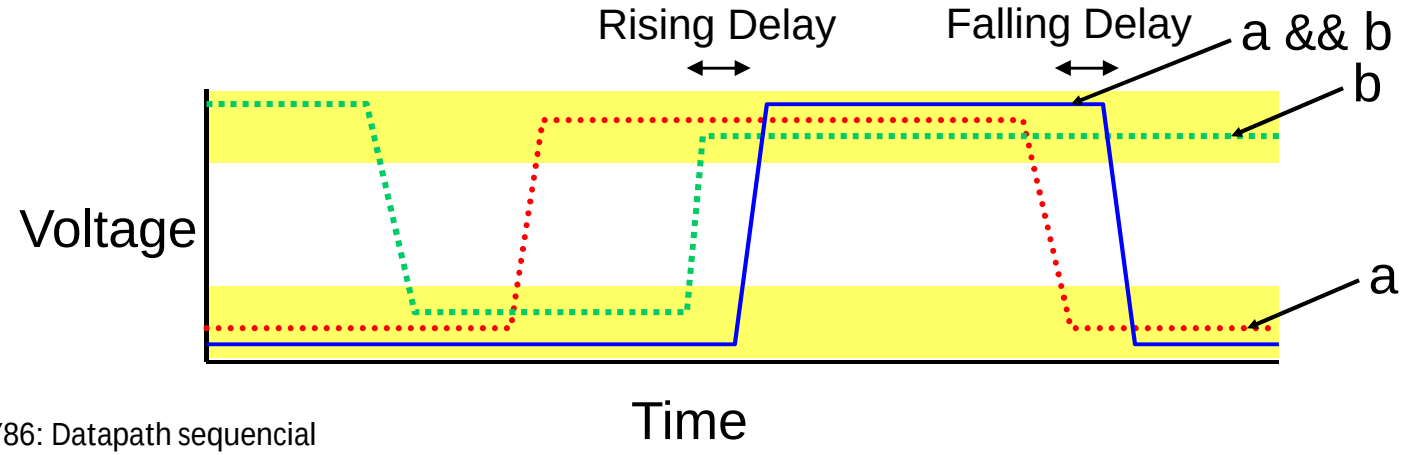
Existem 2 grandes tipos de componentes ou circuitos:

- Circuitos combinatórios
  - Reagem continuamente a mudanças nos valores das entradas (com um atraso de propagação)
- Circuitos sequenciais
  - Capazes de memorizar informação
  - As suas saídas só se alteram quando um sinal de temporização (relógio ou *clock*) o autoriza

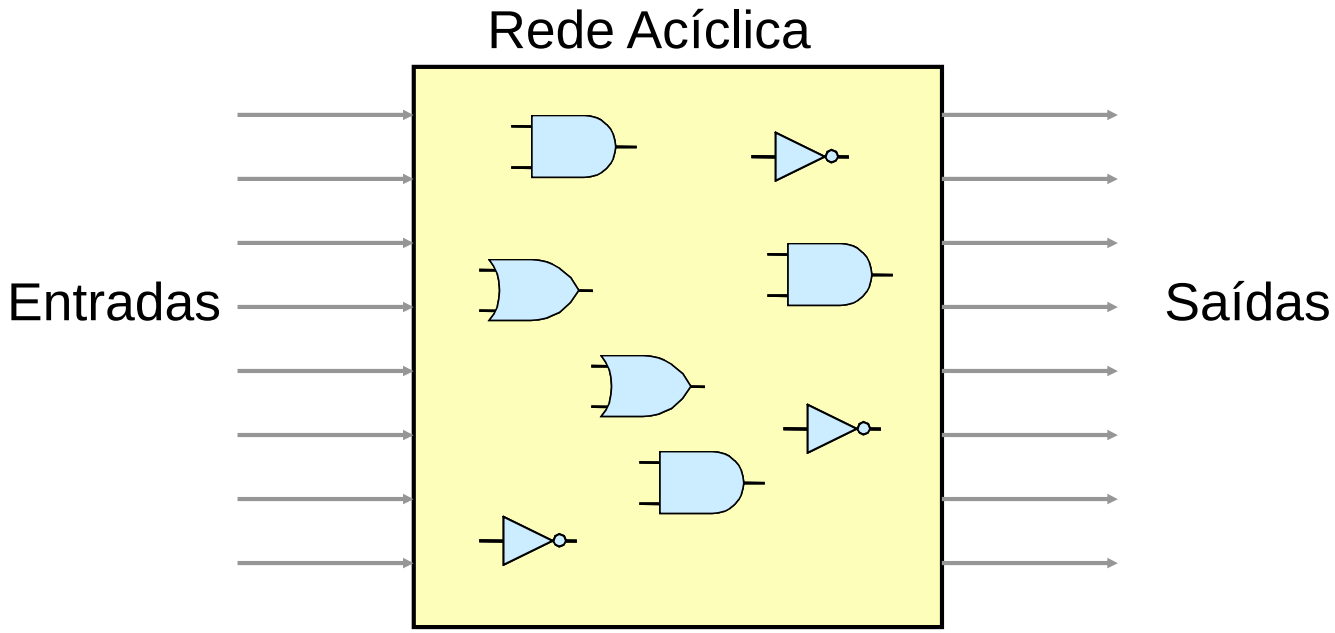
Circuitos Combinatórios: *Gates*



- As saídas são funções Booleanas das entradas
- Respondem continuamente a mudanças nas entradas
  - Com algum atraso

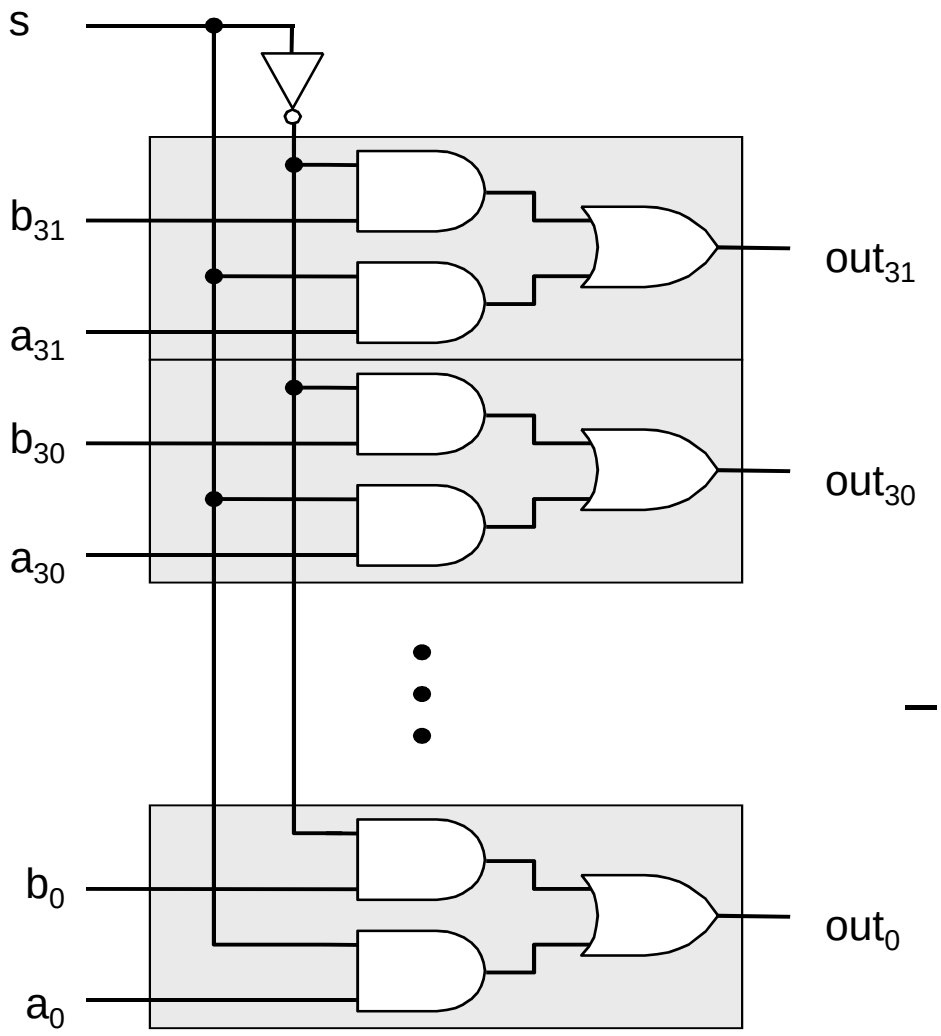


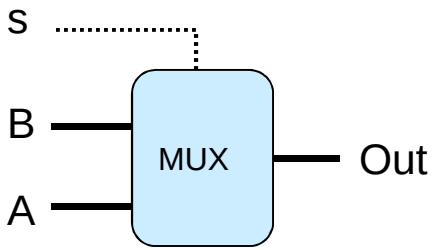
Circuitos Combinatórios



- Rede Acíclica de Componentes Lógicos
  - Respondem continuamente a mudanças nas entradas
  - As saídas transformam-se (após um atraso)em funções Booleanas das entradas

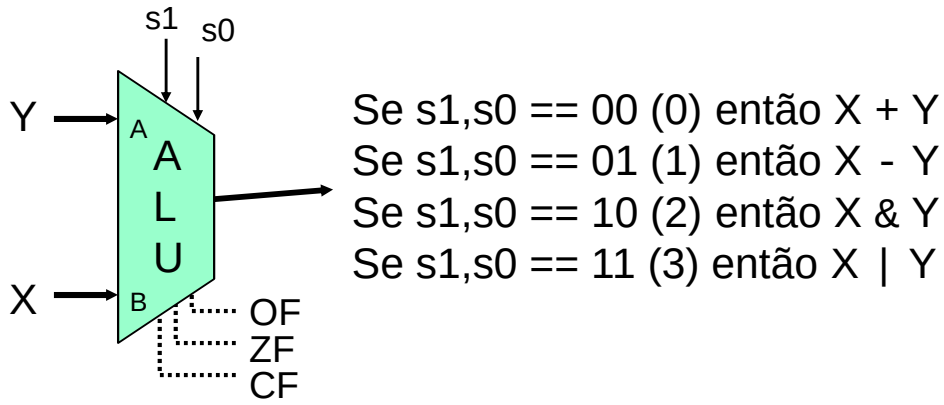
Circuitos combinatórios: *multiplexer*





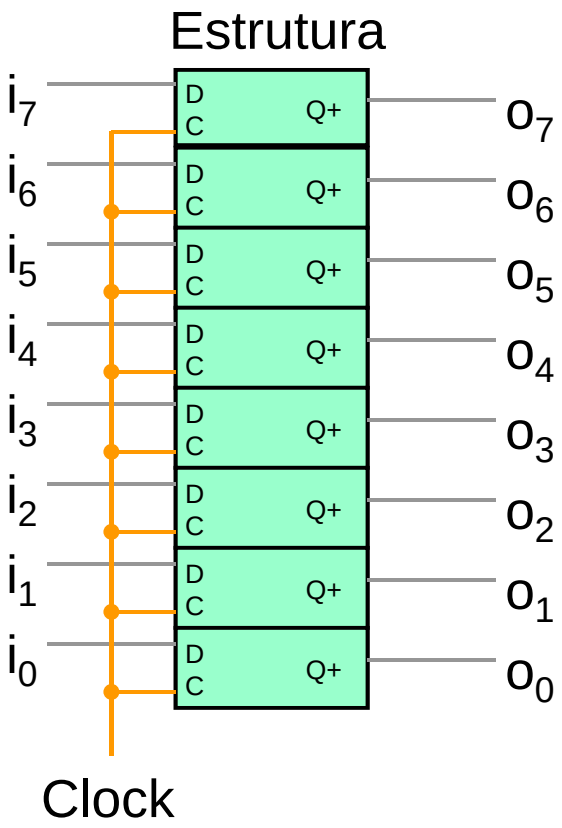
- Selecciona palavra de entrada A ou B dependendo do sinal de controlo s

Circuitos combinatórios: ALU

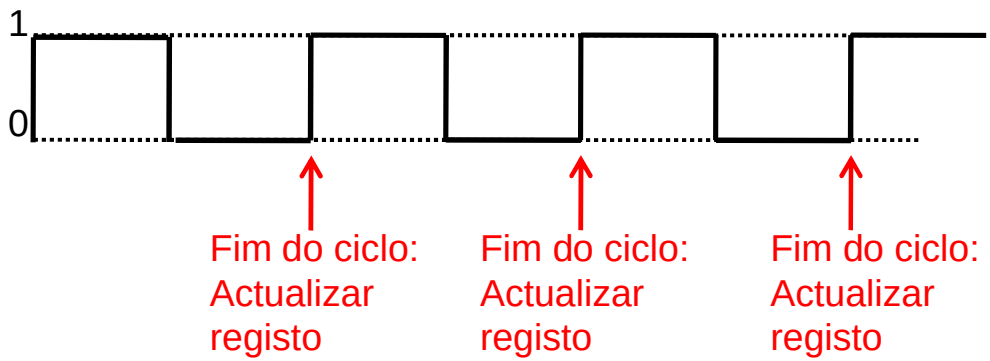


- Lógica Combinatória
  - Responde continuamente às entradas
- Sinais de controlo (s1,s0) seleccionam função a computar
  - Correspondem às 4 instruções lógico-aritméticas do Y86
- Também calcula os códigos de condição

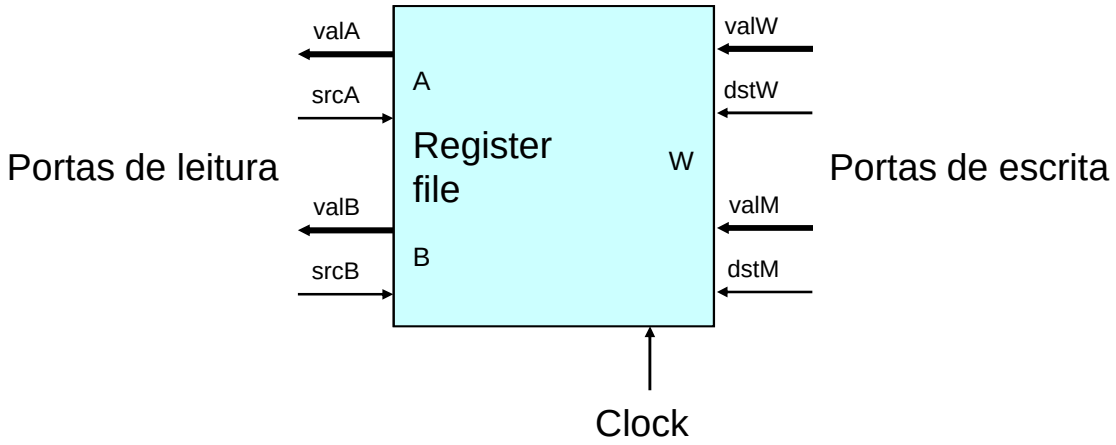
Circuitos Sequenciais: Registos



- Armazenam dados
- As saídas só variam quando o valor do sinal do *clock* sobe de 0 para 1



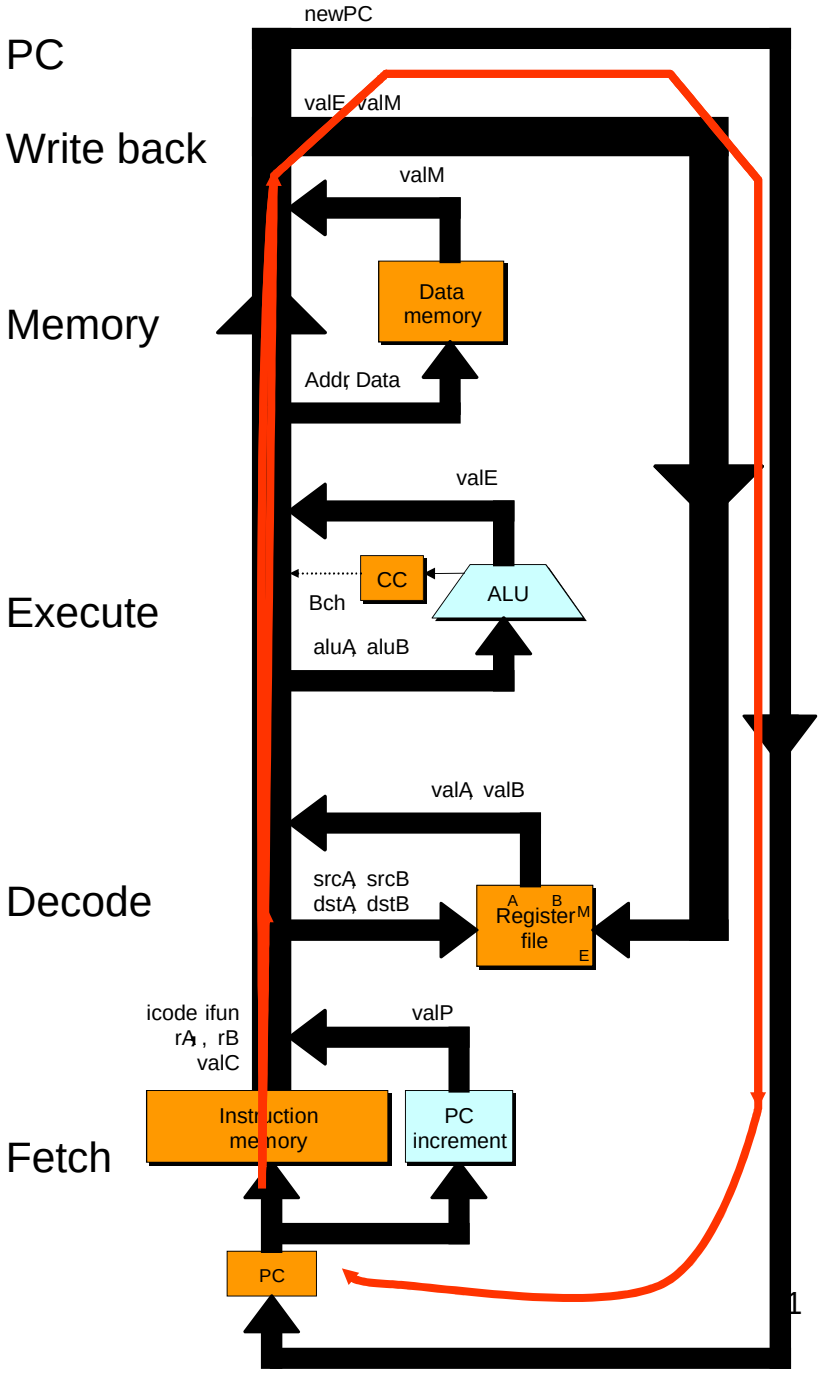
Circuitos Sequenciais: Banco de Registos



- Armazena múltiplas palavras de dados (múltiplos registos)
  - Endereços especificam quais os registos a ler e/ou escrever
    - Endereços nos sinais `srcA`, `srcB`, `dstW`, `dstM`
  - Mantem os valores dos registos: `%eax, %ebx, %esp,` etc.
  - O ID do registo serve como endereço
    - ID 8 implica que não é feita leitura ou escrita
- Múltiplas Portas
  - Pode ler (`srcA, srcB`) e/ou escrever (`dstW, dstM`) múltiplos registos num ciclo

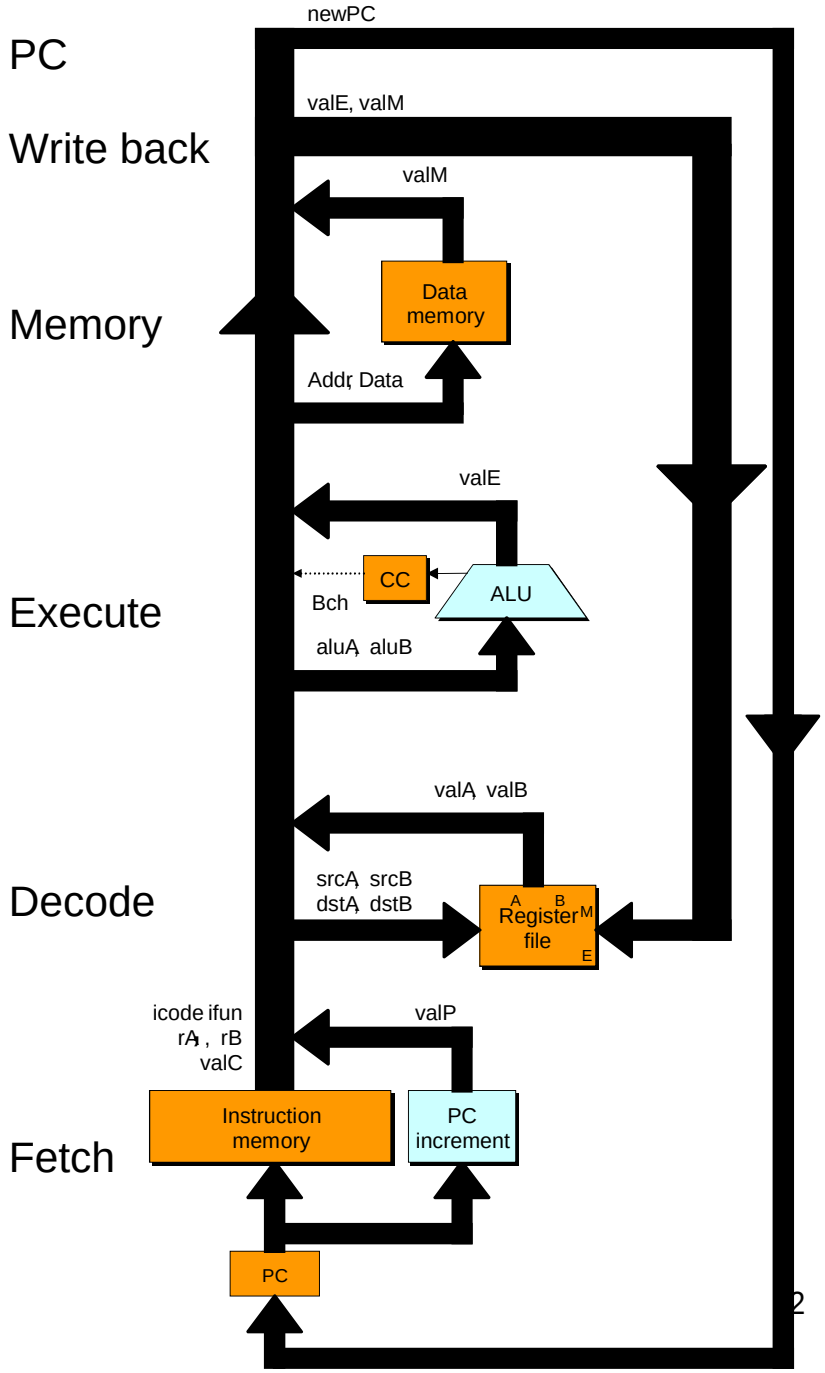
Y86: Organização Sequencial

- Estado
  - Program counter (PC)
  - Códigos de Condição (CC)
  - Banco de Registos
  - Memória
    - Data: para escrita/leitura de dados
    - Instruction: para leitura de instruções
- Fluxo de Instruções
  - Ler instrução no endereço dado pelo PC
  - Processar nos vários estágios
  - Escrita:
    - Actualizar registos genéricos
    - Escrever na memória de dados
    - Actualizar PC



Y86: Organização Sequencial

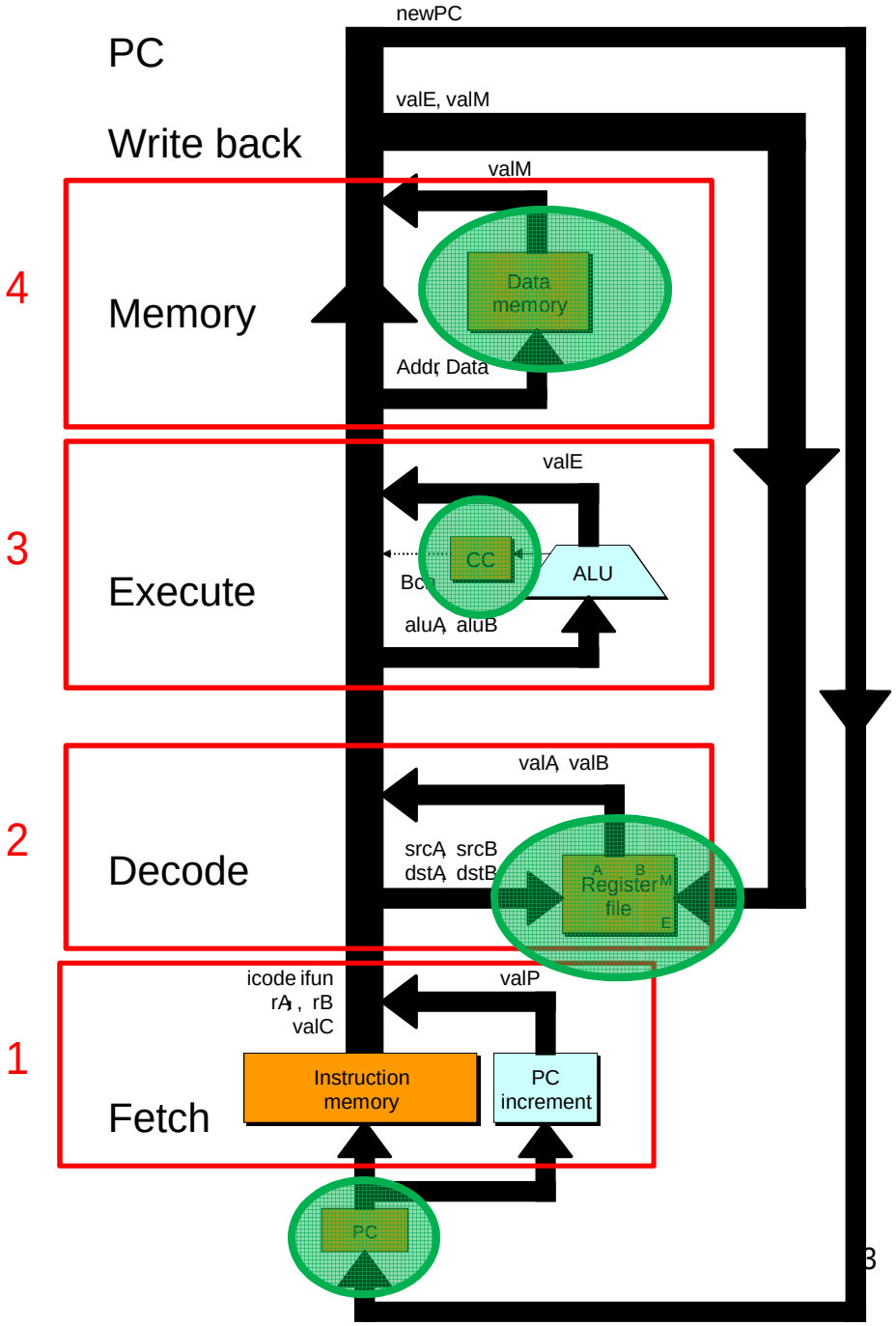
- Busca (*Fetch*)
  - Ler instruções da memória
  - `icode:ifun ; rA:rB ; valC ; valP`
- Descodificação *(Decode*)
  - Ler registos: `valA:valB`
  - Determina ID de regs. a alterar:
    - `srcA ; srcB ; dstE ; dstM`
- Execução (*Execute)*
  - Calcular valor ou endereço / CC
    - `valE  / CC (Bch )`
- Memória (*Memory*)
  - Ler ou escrever dados
    - `valM`
- Actualização (*Write Back)*
  - Escrita nos registos genéricos
- Actualização PC



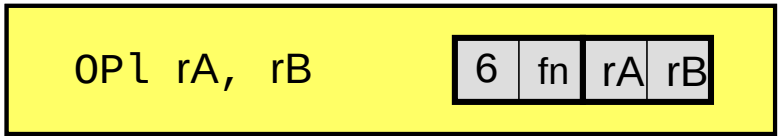
Y86: Organização Sequencial

Atenção à temporização:

1. Busca (início do ciclo)
2. Descodificação
3. Execução
4. Leitura da Memória
5. Escritas
   TODAS as escritas ocorrem SIMULTANEAMENTE no FIM do CICLO
   - Escrita nos CC
   - Escrita na memória
   - Escrita nos regs. genéricos
   - Escrita no PC



Instrução Y86: operações lógico-aritméticas



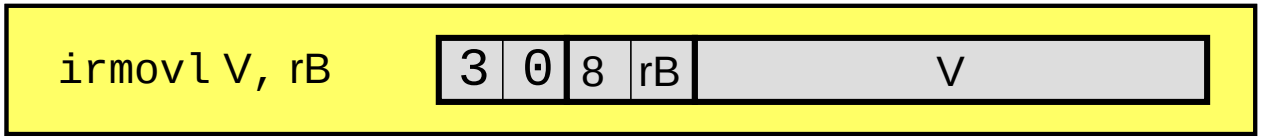
- Fetch
  - Ler 2 bytes
    `icode:ifun ; rA:rB`
  - Calcular próximo PC
    `valP = PC + 2`
- Decode
  - Ler registos
    `valA = R[rA]`
    `valB = R[rB]`
- Execute
  - Realizar operação
    `valE = valA OP valB`
  - Códigos de condição
    `cc = f(valE)`
- Memory
  - Nada a fazer
- Write back
  - Actualizar registo
    `R[rB] = valE`
- PC Update
  - Incrementar PC
    `PC = valP`

Instrução Y86: operações lógico-aritméticas

	OPl rA, rB	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	Ler byte de instrução Ler byte dos registos Calcular próximo PC
Decode	valA ← R[rA] valB ← R[rB]	Ler operando A Ler operando B
Execute	valE ← valB OP valA Set CC	Realizar operação na ALU Códigos de condição
Memory		
Write back	R[rB] ← valE	Escrever resultado
PC update	PC ← valP	Actualizar PC

- Formular execução de instruções como uma sequência de passos simples
- Usar a mesma forma geral para todas as instruções

Instrução Y86: irmovl

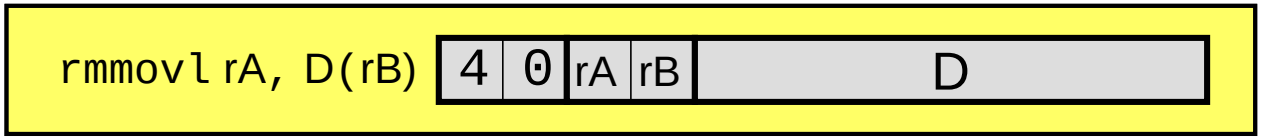


- Fetch
  - Ler 6 bytes
    `icode:ifun ; rA:rB ; valC`
  - Calcular próximo PC
    `valP = PC + 6`
- Decode
  - Nada a fazer
- Execute
  - Passar constante
    `valE = valC`
- Memory
  - Nada a fazer
- Write back
  - Escrever registo
    `R[rB] = valE`
- PC Update
  - Incrementar PC
    `PC = valP`

Instrução Y86: irmovl

	`irmovl` V, rB	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_4[PC+2] valP ← PC+6	Ler byte de instrução Ler byte de registos Ler deslocamento Calcular próximo PC
Decode		
Execute	valE ← valC	Passar constante
Memory		
Write back	R[rB] ← valE	Escrever resultado
PC update	PC ← valP	Actualizar PC

Instrução Y86: rmmovl

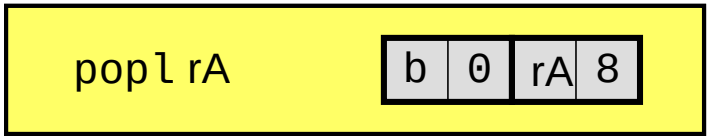


- Fetch
  - Ler 6 bytes
    `icode:ifun ; rA:rB ; valC`
  - Calcular próximo PC
    `valP = PC + 6`
- Decode
  - Ler registos
    `valA = R[rA]`
    `valB = R[rB]`
- Execute
  - Calcular endereço
    `valE = valB + valC`
- Memory
  - Escrever na memória
    M_4`[valE] = valA`
- Write back
  - Nada a fazer
- PC Update
  - Incrementar PC
    `PC = valP`

Instrução Y86: rmmovl

	`rmmovl` rA, D(rB)	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_4[PC+2] valP ← PC+6	Ler byte de instrução Ler byte de registos Ler deslocamento Calcular próximo PC
Decode	valA ← R[rA] valB ← R[rB]	Ler operando A Ler operando B
Execute	valE ← valB + valC	Calcular endereço
Memory	M_4[valE] ← valA	Escrever na memória
Write back		
PC update	PC ← valP	Actualizar PC

Instrução Y86: popl



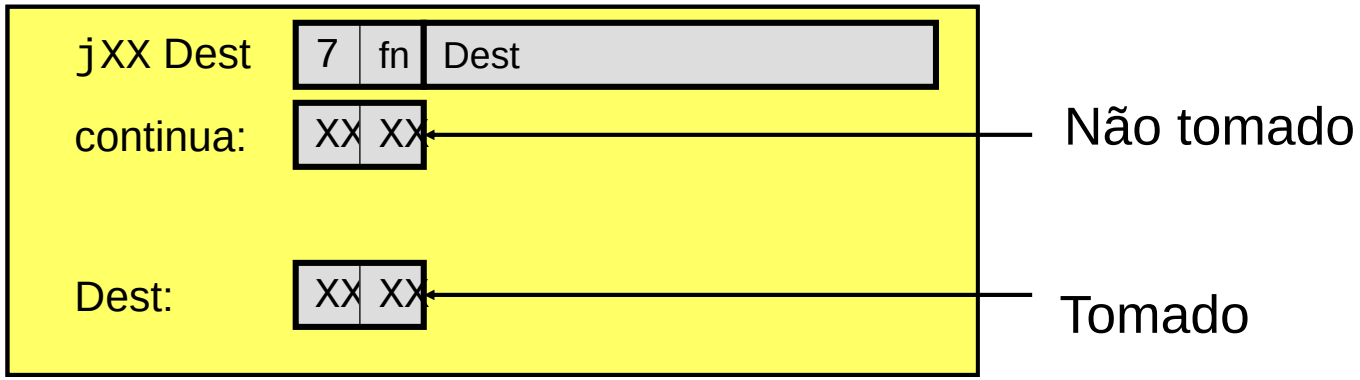
- Fetch
  - Ler 2 bytes
    `icode:ifun ; rA:rB`
  - Calcular próximo PC
    `valP = PC + 2`
- Decode
  - Ler registos
    `valA = R[%esp]`
    `valB = R[%esp]`
- Execute
  - Calcular próximo topo da pilha
    `valE = valB + 4`
- Memory
  - Ler da memória
    `valM = M4[valA]`
- Write back
  - Escrever no registo
    `R[rA] = valM`
    `R[%esp] = valE`
- PC Update
  - Incrementar PC
    `PC = valP`

Instrução Y86: popl

	`popl` rA	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	Ler byte de instrução Ler byte de registo Calcular próximo PC
Decode	valA ← R[%esp] valB ← R[%esp]	Ler stack pointer Ler stack pointer
Execute	valE ← valB + 4	Incrementar stack pointer
Memory	valM ← M_4[valA]	Ler topo da pilha
Write back	R[%esp] ← valE R[rA] ← valM	Actualizar stack pointer Escrever resultado
PC update	PC ← valP	Actualizar PC

- Usar ALU para incrementar stack pointer
- Actualizar dois registos: rA e %esp

Instrução Y86: Jump



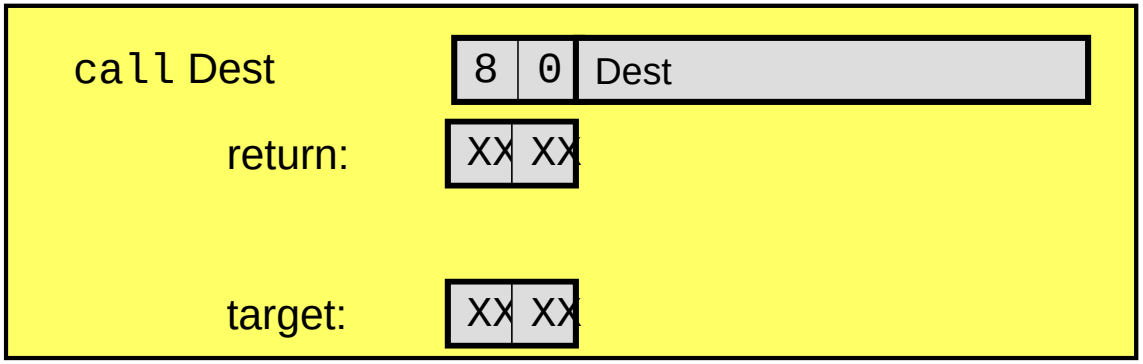
- Fetch
  - Ler 5 bytes
    `icode:ifun ; valC`
  - Calcular próximo PC
    `valP = PC + 5`
- Decode
  - Nada a fazer
- Execute
  - Saltar ??
    `Bch = f(ifun, CC)`
- Memory
  - Nada a fazer
- Write back
  - Nada a fazer
- PC Update
  - Novo valor depende de Bch
    `PC = (Bch ? valC : valP)`

Instrução Y86: Jump

	`jxx` Dest	
Fetch	icode:ifun ← M_1[PC] valC ← M_4[PC+1] valP ← PC+5	Ler byte de instrução Ler destino do salto (se tomado) Calcular próximo PC (se ñ tomado)
Decode		
Execute	Bch ← f(ifun, CC)	Saltar?
Memory		
Write back		
PC update	PC ← (Bch ? valC : valP)	Actualizar PC

- Calcular ambos os endereços
- Escolher destino baseado no tipo de salto e nos códigos de condição

Instrução Y86: call

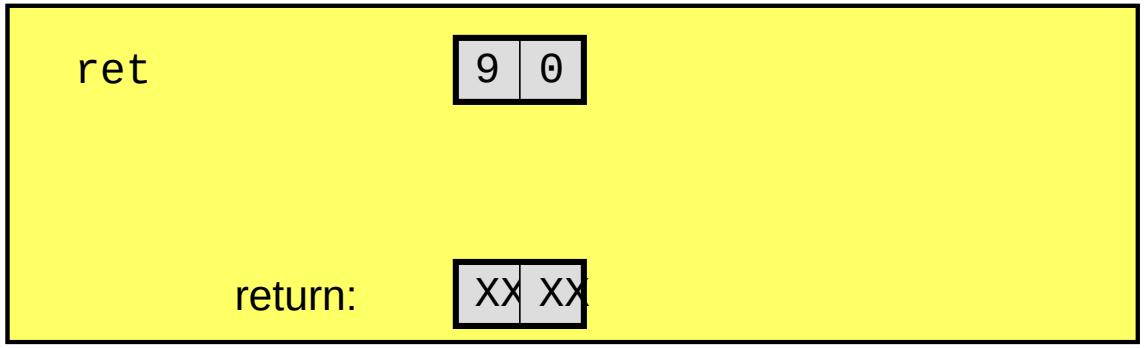


- Fetch
  - Ler 5 bytes
    `icode:ifun ; valC`
  - Calcular próximo PC (end. de retorno)
    `valP = PC + 5`
- Decode
  - Le registo
    `valB = R[%esp]`
- Execute
  - Calcular novo topo da pilha
    `valE = valB - 4`
- Memory
  - Push do end. de retorno
    M_4`[valE] = valP`
- Write back
  - Actualizar topo da pilha
    `R[%esp] = valE`
- PC Update
  - `PC = valC`

Instrução Y86: call

	`call Dest`	
Fetch	icode:ifun ← M_1[PC] valC ← M_4[PC+1] valP ← PC+5	Ler byte de instrução Ler destino do salto Calcular endereço de retorno
Decode	valB ← R[%esp]	Ler addr do topo da pilha
Execute	valE ← valB - 4	Novo topo da pilha
Memory	M_4[valE] ← valP	Guardar endereço de retorno
Write back	R[%esp] ← valE	Actualizar topo da pilha
PC update	PC ← valC	Actualizar PC

Instrução Y86: ret



- Fetch
  - Ler 1 byte `icode:ifun`
  - Calcular próximo PC
    `valP = PC + 1`
- Decode
  - Ler registo
    `valA = R[%esp]`
    `valB = R[%esp]`
- Execute
  - Calcular novo topo da pilha
    `valE = valB + 4`
- Memory
  - Ler end. de retorno
    `valM = M4[valA]`
- Write back
  - Actualizar topo da pilha
    `R[%esp] = valE`
- PC Update
  - `PC = valM`

Instrução Y86: ret

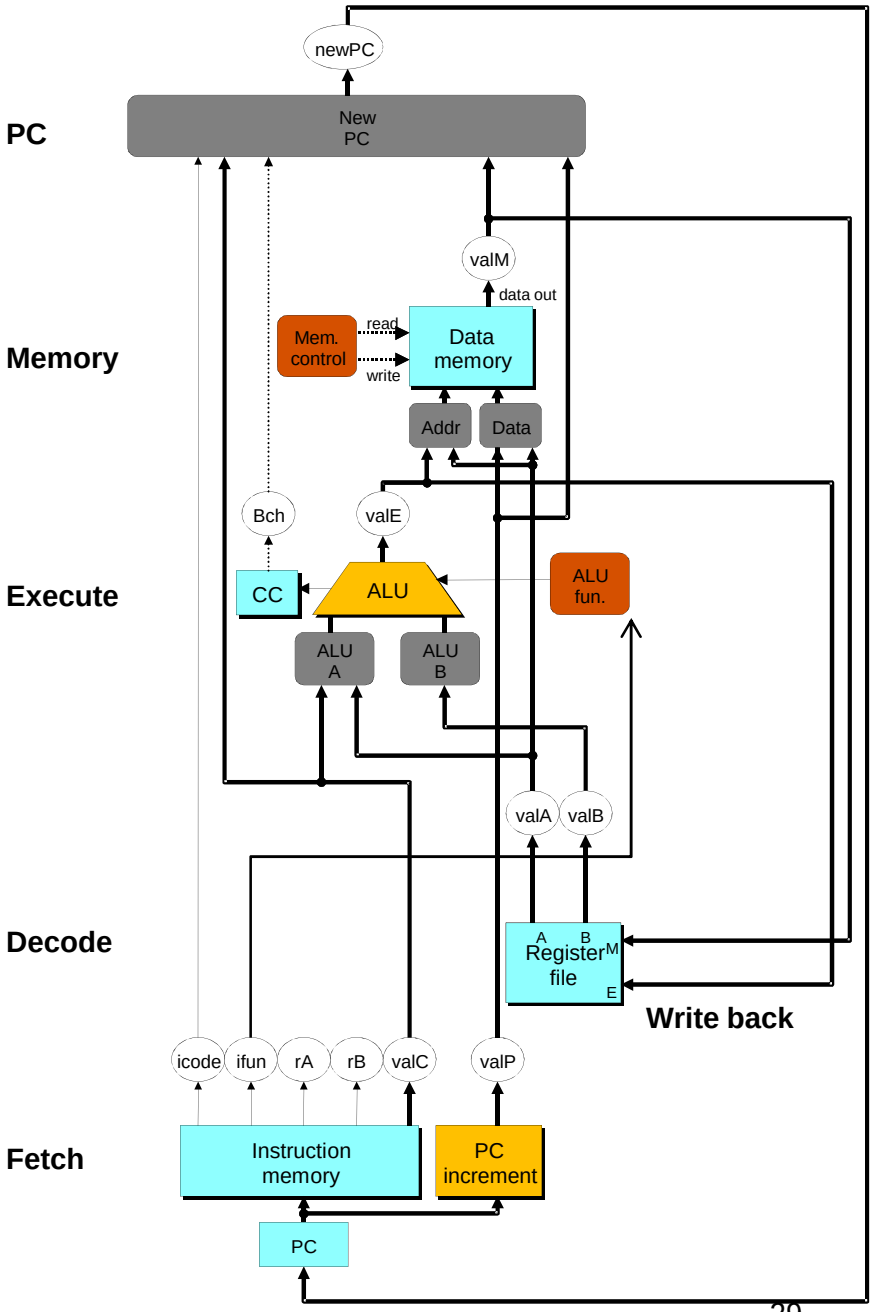
	`ret`	
Fetch	icode:ifun ← M_1[PC] valP ← PC+1	Ler byte de instrução Calcular próximo PC
Decode	valA ← R[%esp] valB ← R[%esp]	Ler addr do topo da pilha
Execute	valE ← valB + 4	Novo topo da pilha
Memory	valM ← M_4[valEA	Ler endereço de retorno
Write back	R[%esp] ← valE	Actualizar topo da pilha
PC update	PC ← valM	Actualizar PC

Y86: Valores calculados

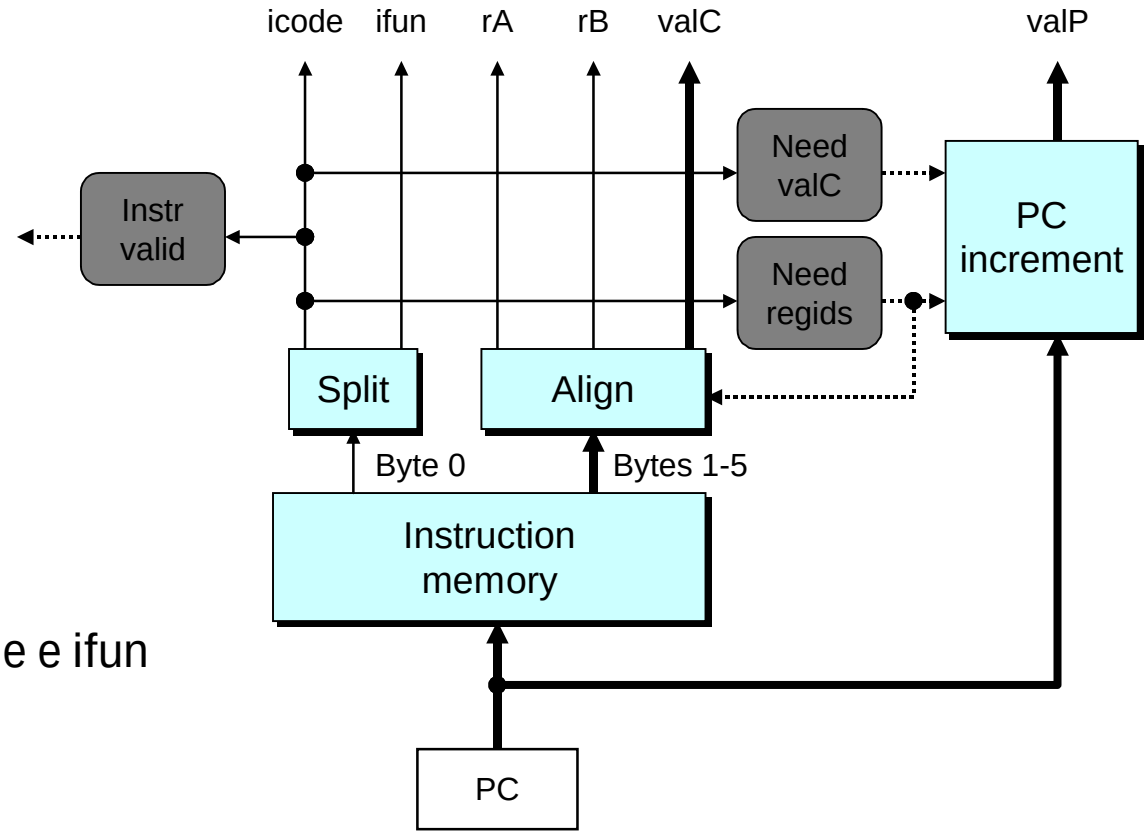
- Fetch
  - icode Instruction code
  - ifun Instruction function
  - rA Instr. Register A
  - rB Instr. Register B
  - valC Instruction constant
  - valP Incremented PC
- Decode
  - valA Register value A
  - valB Register value B
- Execute
  - valE ALU result
  - Bch Branch flag
- Memory
  - valM Value from memory

Y86: SEQ

- Ovais brancas: sinais internos
- Caixas azuis: lógica sequencial
  escrita controlada pelo *clock*
- Caixas cinzentas: multiplexers
- Caixas laranja: Cálculo
  ALU e "incrementar PC"
- Caixas castanhas: controlo
- Traço grosso: 32 bits
- Traço fino: 4 bits
- Tracejado: 1 bit



Y86: *Fetch*

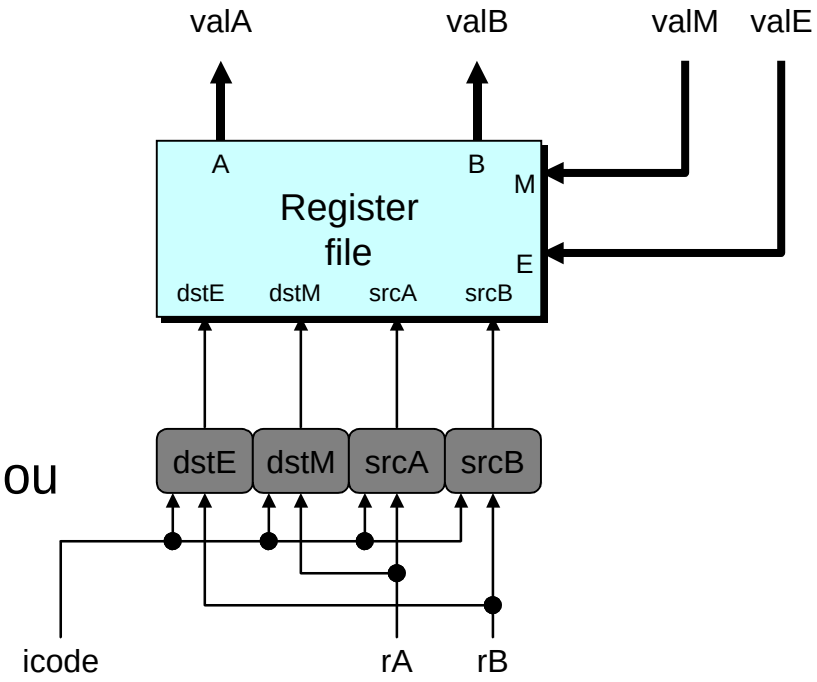


- Blocos
  - PC: Registo
  - Memória Instruções:
    Ler 6 bytes (PC to PC+5)
  - Split: Dividir byte em icode e ifun
  - Align: Obter rA, rB e valC

- 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?

Y86: *Decode*

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



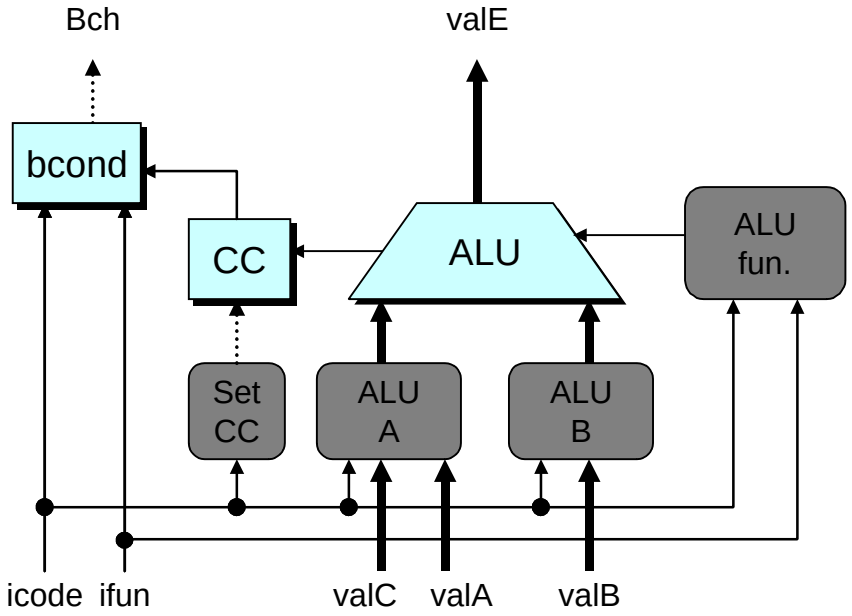
Lógica de Controlo

- srcA, srcB: registos a ler
- dstA, dstB: registos a escrever

Y86: *Execute*

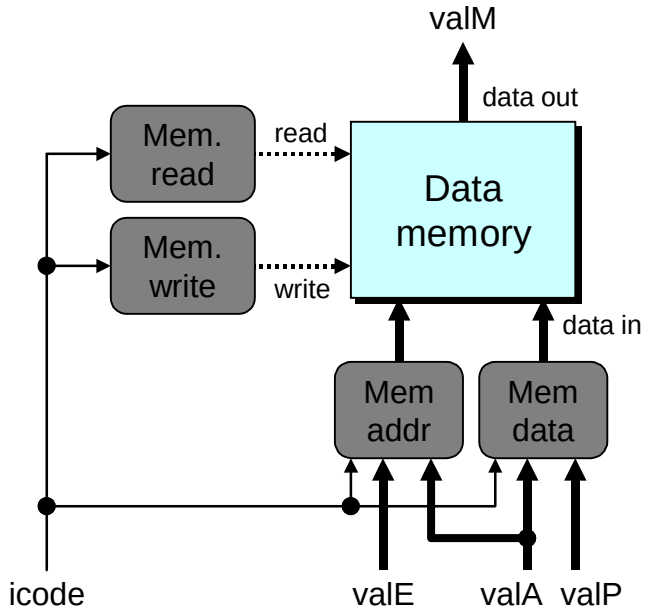
- Unidades
  - ALU
    - Implementa 4 funções
    - Gera códigos de condição
  - CC
    - Registo com 3 bits
  - bcond
    - Calcular se o salto é tomado (*Bch)*

- 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*

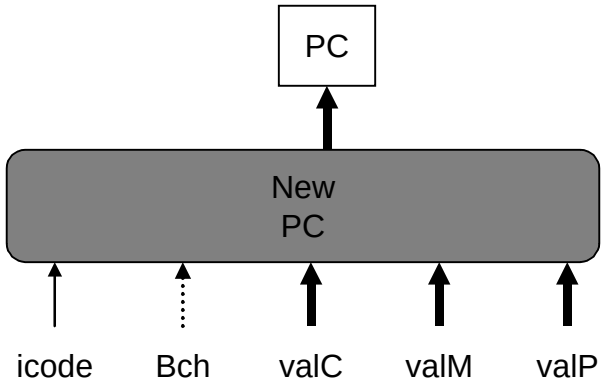
- Memória
  - Ler ou escrever uma palavra
- Lógica de Controlo
  - Mem.read: leitura?
  - Mem.write: escrita?
  - Mem.addr: Selecciona addr
  - Mem.data: Selecciona dados



Y86: *PC*

- 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



Y86: SEQ resumo

- Implementação
  - Cada instrução expressa como uma sequência de passos elementares
  - Mesma sequência geral para todos os tipos de instruções
  - Identificar blocos combinatórios e sequenciais básicos
  - Ligar e controlar com a lógica de controlo

Y86: SEQ Limitações

- Em cada ciclo do relógio os sinais têm que se propagar desde o PC, memória de instruções, banco de registos, ALU e memória até aos registos de novo:
  - O período do relógio tem que ser suficientemente longo para permitir esta propagação
  - O período do relógio é constante, logo tem que ser tão grande quanto necessário para a instrução mais lenta
  - O relógio é, portanto, muito lento
- Cada unidade de hardware só está activa (é utilizada) durante uma fracção pequena do período do relógio
  - Logo, desperdício das verdadeiras capacidades do equipamento

Y86: SEQ+

- Estágio PC reordenado: passa a acontecer ao princípio
- Estágio PC
  - Selecciona PC para a instrução actual
  - Baseado nos resultados da última instrução
  - Estes são guardados num registo e consistem em: `pIcode; pBch; pvalM ; pvalC ; pvalP`
  - PC não está guardado em nenhum registo