

# Arquitectura: implementación secuencial

95.57/75.03 Organización del computador

---

**Docentes:** Patricio Moreno y Adeodato Simó

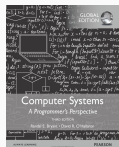
2.<sup>do</sup> cuatrimestre de 2020

Última modificación: Mon Jul 27 13:02:42 2020 -0300

Facultad de Ingeniería (UBA)

# Créditos

Para armar las presentaciones del curso nos basamos en:



R. E. Bryant and D. R. O'Hallaron, *Computer systems: a programmer's perspective*, Third edition, Global edition. Boston Columbus Hoboken Indianapolis New York San Francisco Cape Town: Pearson, 2015.



D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*, RISC-V edition. Cambridge, Massachusetts: Morgan Kaufmann Publishers, an imprint of Elsevier, 2017.



J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. 2017.

# Tabla de contenidos

---

1. Instrucciones Y86-64
2. Elementos básicos
3. Descripción de hardware
4. Implementación secuencial
  - Estructura
  - Etapas lógicas
  - Operación
  - Implementación

# Tabla de contenidos

---

1. Instrucciones Y86-64
2. Elementos básicos
3. Descripción de hardware
4. Implementación secuencial
  - Estructura
  - Etapas lógicas
  - Operación
  - Implementación

# Set de instrucciones Y86-64

Número de byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmovXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn	Dest							
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

# Tabla de contenidos

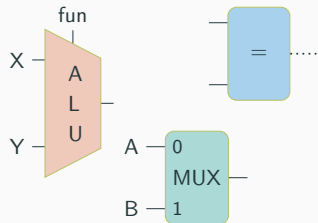
---

1. Instrucciones Y86-64
2. Elementos básicos
3. Descripción de hardware
4. Implementación secuencial
  - Estructura
  - Etapas lógicas
  - Operación
  - Implementación

# Bloques básicos

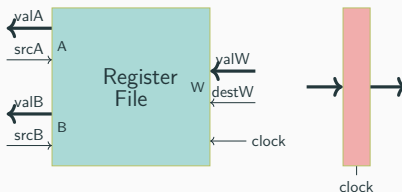
## Lógica combinacional

- Obtienen valores booleanos a partir de funciones
- Responden continuamente a la entrada
- Operan con los datos e implementan el control



## Elementos de almacenamiento

- Guardan bits
- Memoria direccionable
- Registros no direccionables
- Son elementos sincrónicos



# Tabla de contenidos

---

1. Instrucciones Y86-64
2. Elementos básicos
3. Descripción de hardware
4. Implementación secuencial
  - Estructura
  - Etapas lógicas
  - Operación
  - Implementación



# Hardware Control Language

---

- Lenguaje de descripción de hardware (HDL) muy simple
  - Sintaxis similar a las operaciones lógicas en C para las operaciones booleanas
- Diseñado por los docentes del curso de la CMU
- Permite expresar muy pocos aspectos del diseño
  - Lo mínimo que vamos a analizar
- Se utiliza para describir la lógica del procesador diseñado

# Hardware Control Language: tipos

## Tipos de datos

- `bool`: booleano
  - `a`, `b`, `c`, ...
- `int`: enteros
  - `A`, `B`, `C`, ...
  - No especifica el tamaño de la palabra—bytes, palabras de 64 bits, ...

## Declaraciones

- `bool a = expresión-booleana`
- `int A = expresión-entera`

# Hardware Control Language: operaciones

## Expresiones booleanas

- Operaciones lógicas
  - `a && b`, `a || b`, `!a`
- Comparaciones
  - `A == B`, `A != B`, `A < B`, `A <= B`, `A > B`, `A >= B`
- Operaciones con conjuntos
  - `A in {B, C, D}`
    - Equivalente a `(A == B || A == C || A == D)`

## Expresiones con palabras

- `case`: `[a : A; b : B; c : C]`
  - Evalúa las expresiones `a`, `b`, `c`, ... en esa secuencia
  - Retorna la expresión `A`, `B`, `C`, ... para la primera expresión (de las anteriores) que retorna verdadero

Clasifica los tipos de las expresiones en función del tipo de retorno

# Tabla de contenidos

---

1. Instrucciones Y86-64
2. Elementos básicos
3. Descripción de hardware
4. Implementación secuencial
  - Estructura
  - Etapas lógicas
  - Operación
  - Implementación

# Tabla de contenidos

---

1. Instrucciones Y86-64
2. Elementos básicos
3. Descripción de hardware
4. Implementación secuencial

Estructura

Etapas lógicas

Operación

Implementación

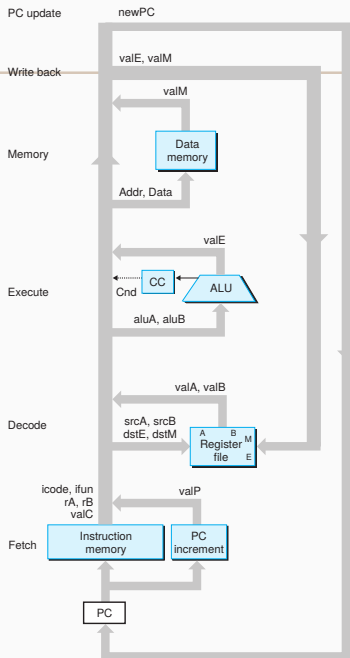
# Estructura secuencial

## Estado

- Registro del *program counter* (PC)
- Registro de condiciones (CC)
- Banco de registros (*Register File*)
- Memorias
  - Acceden al mismo espacio de memorias
  - Data: lee y escribe datos del programa
  - Instruction: lee instrucciones

## Flujo de ejecución

1. Leer instrucción de la dirección indicada por el PC
2. Procesar a través de cada etapa
3. Actualizar el PC



# Tabla de contenidos

---

1. Instrucciones Y86-64
2. Elementos básicos
3. Descripción de hardware
4. Implementación secuencial

Estructura

Etapas lógicas

Operación

Implementación

# Etapas del diseño SEQ

## Búsqueda (*Fetch*)

- Lee instrucciones de la memoria

## Decodificación (*Decode*)

- Lee registros del programa

## Ejecución (*Execute*)

- Calcula el valor o dirección necesario

## Acceso a memoria/Saltos (*Memory*)

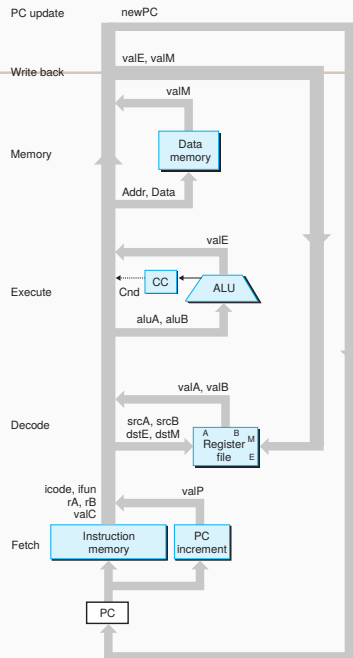
- Lee o escribe datos

## Postescritura (*Write-Back*)

- Escribe (actualiza) los registros del programa

## PC

- Actualiza el *programa counter*



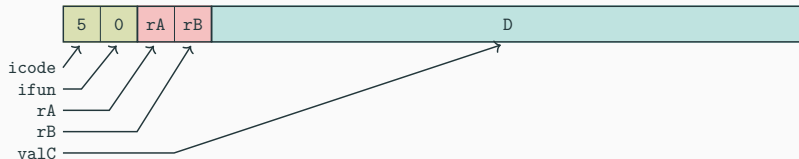


# Tabla de contenidos

---

1. Instrucciones Y86-64
2. Elementos básicos
3. Descripción de hardware
4. Implementación secuencial
  - Estructura
  - Etapas lógicas
  - Operación
  - Implementación

# Fetch y decodificación de una instrucción

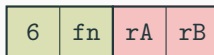


## Formato de las instrucciones

- Byte de instrucción icode:ifun
- Byte de registros (opcional) rA:rB
- qword de valor constante (opcional) valC

# Ejecución de una operación aritmético lógica

OPq rA, rB



## Búsqueda (*Fetch*)

- Lee 2 bytes

## Decodificación (*Decode*)

- Lee los registros que figuran como operandos

## Ejecución (*Execute*)

- Realiza la operación
- Establece los códigos de condición

## Acceso a memoria/Saltos (*Memory*)

- No hace nada

## Postescritura (*Write-Back*)

- Actualiza los registros

## PC Update

- Incrementa el PC en 2

# Operación en cada etapa: instrucción aritmética

	OPq rA, rB	
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	Lee el byte de instrucción
	$\text{rA:rB} \leftarrow M_1[\text{PC}+1]$	Lee el byte de registros
Decode	$\text{valP} \leftarrow \text{PC}+2$	Calcula el siguiente PC
	$\text{valA} \leftarrow R[\text{rA}]$	Lee el operando A
Execute	$\text{valB} \leftarrow R[\text{rB}]$	Lee el operando B
	$\text{valE} \leftarrow \text{valB OP valA}$	Realiza la operación de la ALU
Memory	Establecer CC	Modifica el registro CC
Write Back	$R[\text{rB}] \leftarrow \text{valE}$	Actualiza los registros
PC Update	$\text{PC} \leftarrow \text{valP}$	Actualiza el PC

# Ejemplo

```

1  0x000 : 30f20900000000000000 | irmovq $9, %rdx
2  0x00a : 30f31500000000000000 | irmovq $21, %rbx
3  0x014 : 6123                    | subq %rdx, %rbx

```

	OPq rA, rB	subq %rdx, %rbx
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC} + 2$	$\text{icode:ifun} \leftarrow M_1[0x14] = 6:1$ $\text{rA:rB} \leftarrow M_1[0x15] = 2:3$ $\text{valP} \leftarrow 0x14 + 2 = 0x16$
Decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$	$\text{valA} \leftarrow R[\%rdx] = 9$ $\text{valB} \leftarrow R[\%rbx] = 21$
Execute	$\text{valE} \leftarrow \text{valB OP valA}$ Establecer CC	$\text{valE} \leftarrow 21 - 9 = 12$ $\text{ZF} \leftarrow 0, \text{SF} \leftarrow 0, \text{OF} \leftarrow 0$
Memory		
Write Back	$R[\text{rB}] \leftarrow \text{valE}$	$R[\%rbx] \leftarrow \text{valE} = 12$
PC Update	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP} = 0x16$

# Ejecución de `rmmovq`

`rmmovq rA, D(rB)`



## Búsqueda (*Fetch*)

- Lee 10 bytes

## Decodificación (*Decode*)

- Lee los registros que figuran como operandos

## Ejecución (*Execute*)

- Calcula la dirección efectiva

## Acceso a memoria/Saltos (*Memory*)

- Escribe la memoria

## Postescritura (*Write-Back*)

- No hace nada

## PC Update

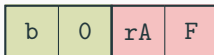
- Incrementa el PC en 10

# Operación en cada etapa: instrucción `rmmovq`

<code>rmmovq rA, D(rB)</code>		
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_8[PC+2]$ $valP \leftarrow PC+10$	Lee el byte de instrucción Lee el byte de registros Lee el desplazamiento D Calcula el siguiente PC
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$	Lee el operando A Lee el operando B
Execute	$valE \leftarrow valB + valC$	Calcula la dirección efectiva
Memory	$M_8[valE] \leftarrow valA$	Guarda el valor en memoria
Write Back		
PC Update	$PC \leftarrow valP$	Actualiza el PC

# Ejecución de popq

popq rA



## Búsqueda (*Fetch*)

- Lee 2 bytes

## Decodificación (*Decode*)

- Lee el *stack pointer*

## Ejecución (*Execute*)

- Incrementa en 8 el *stack pointer*

## Acceso a memoria/Saltos (*Memory*)

- Lee del valor anterior del *stack pointer*

## Postescritura (*Write-Back*)

- Actualiza el *stack pointer*
- Escribe lo leído en el registro destino

## PC Update

- Incrementa el PC en 2

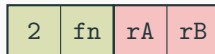


# Operación en cada etapa: popq

	popq rA	
Fetch	$icode:ifun \leftarrow M_1[PC]$	Lee el byte de instrucción
	$rA:rB \leftarrow M_1[PC+1]$	Lee el byte de registros
Decode	$valP \leftarrow PC+2$	Calcula el siguiente PC
	$valA \leftarrow R[\%rsp]$	Lee el <i>stack pointer</i>
Execute	$valB \leftarrow R[\%rsp]$	Lee el <i>stack pointer</i>
	$valE \leftarrow valB + 8$	Incrementa el <i>stack pointer</i>
Memory	$valM \leftarrow M_8[valA]$	Lee del <i>stack pointer</i> anterior
Write	$R[\%rsp] \leftarrow valE$	Actualiza el <i>stack pointer</i>
Back	$R[rA] \leftarrow valM$	Guarda lo leído
PC Update	$PC \leftarrow valP$	Actualiza el PC

# Ejecución de movimientos condicionales

cmovXX rA, rB



## Búsqueda (*Fetch*)

- Lee 2 bytes

## Decodificación (*Decode*)

- Lee los operandos

## Ejecución (*Execute*)

- Si !cnd, cambiar el registro de destino a 0xF

## Acceso a memoria/Saltos (*Memory*)

- No hace nada

## Postescritura (*Write-Back*)

- Actualiza el registro (o no)

## PC Update

- Incrementa el PC en 2

# Operación en cada etapa: cmovxx

cmovxx rA, rB		
Fetch	icode:ifun $\leftarrow M_1[PC]$	Lee el byte de instrucción
	rA:rB $\leftarrow M_1[PC+1]$	Lee el byte de registros
Decode	valP $\leftarrow PC+2$	Calcula el siguiente PC
	valA $\leftarrow R[rA]$ valB $\leftarrow 0$	Lee el operando A
Execute	valE $\leftarrow valB + valA$	Pasa valA a través de la ALU
	si !Cond(CC, ifun) rb $\leftarrow 0xF$	Deshabilita rB
Memory		
Write Back	R[rB] $\leftarrow valE$	Guarda el resultado en rB
PC Update	PC $\leftarrow valP$	Actualiza el PC

# Ejecución de llamadas a funciones



## Búsqueda (*Fetch*)

- Lee 9 bytes
- Incrementa el PC en 9

## Decodificación (*Decode*)

- Lee el *stack pointer* (*SP*)

## Ejecución (*Execute*)

- Resta 8 al SP

## Acceso a memoria/Saltos (*Memory*)

- Guarda el nuevo valor del PC en el SP modificado

## Postescritura (*Write-Back*)

- Actualiza el SP

## PC Update

- Asigna dest al PC

# Operación en cada etapa: call

call dest		
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	Lee el byte de instrucción
	$\text{valC} \leftarrow M_8[\text{PC}+1]$	Lee la dirección destino
	$\text{valP} \leftarrow \text{PC} + 9$	Calcula la dirección de retorno
Decode	$\text{valB} \leftarrow R[\%rsp]$	Lee el <i>stack pointer</i>
Execute	$\text{valE} \leftarrow \text{valB} + -8$	Decrementa el SP
Memory	$M_8[\text{valE}] \leftarrow \text{valP}$	Guarda la dirección de retorno
Write Back	$R[\%rsp] \leftarrow \text{valE}$	Actualiza el SP
PC Update	$\text{PC} \leftarrow \text{valC}$	Actualiza el PC

# Ejecución de retornos



## Búsqueda (*Fetch*)

- Lee 1 byte

## Decodificación (*Decode*)

- Lee el *stack pointer* (*SP*)

## Ejecución (*Execute*)

- Suma 8 al *SP*

## Acceso a memoria/Saltos (*Memory*)

- Lee la dirección de retorno del *SP* anterior

## Postescritura (*Write-Back*)

- Actualiza el *SP*

## PC Update

- Asigna la dirección de retorno al *PC*

# Operación en cada etapa: ret

ret		
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	Lee el byte de instrucción
Decode	$\text{valA} \leftarrow R[\%rsp]$	Lee el <i>stack pointer</i>
	$\text{valB} \leftarrow R[\%rsp]$	Lee el <i>stack pointer</i>
Execute	$\text{valE} \leftarrow \text{valB} + 8$	Incrementa el SP
Memory	$\text{valM} \leftarrow M_8[\text{valA}]$	Lee la dirección de retorno
Write Back	$R[\%rsp] \leftarrow \text{valE}$	Actualiza el SP
PC Update	$\text{PC} \leftarrow \text{valM}$	Actualiza el PC

# Ejecución de saltos

jxx dest



## Búsqueda (*Fetch*)

- Lee 9 bytes
- Incrementa el PC en 9

## Decodificación (*Decode*)

- No hace nada

## Ejecución (*Execute*)

- Determina si se salta en función de la condición de salto y los CC

## Acceso a memoria/Saltos (*Memory*)

- No hace nada

## Postescritura (*Write-Back*)

- No hace nada

## PC Update

- Asigna dest al PC si hay que saltar, o el valor incrementado si no



# Operación en cada etapa: ret

	jxx dest	
Fetch	icode:ifun $\leftarrow M_1[PC]$	Lee el byte de instrucción
	valC $\leftarrow M_8[PC + 1]$	Lee la dirección destino
	valP $\leftarrow PC + 9$	Incremento del PC
Decode		
Execute	Cnd $\leftarrow \text{Cond}(CC, \text{ifun})$	¿se debe efectuar el salto?
Memory		
Write Back		
PC Update	PC $\leftarrow \text{Cnd} ? \text{valC} : \text{valP}$	Actualiza el PC

# Operación en cada etapa: visión general

- Todas las instrucciones siguen el mismo patrón
- Cambia lo que se obtiene en cada etapa

	cómputo	OPq rA, rB	Descripción
Fetch	icode, ifun	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	Lee el byte de instrucción
	rA, rB	$\text{rA:rB} \leftarrow M_1[\text{PC}+1]$	Lee el byte de registros
	valC		[Lee la constante]
	valP	$\text{valP} \leftarrow \text{PC}+2$	Calcula el siguiente PC
Decode	valA, srcA	$\text{valA} \leftarrow R[\text{rA}]$	Lee el operando A
	valB, srcB	$\text{valB} \leftarrow R[\text{rB}]$	Lee el operando B
Execute	valE	$\text{valE} \leftarrow \text{valB OP valA}$	Realiza la operación de la ALU
	Cond Code	Establecer CC	Usa/Modifica el registro CC
Memory	valM		[Lectura/Escritura de la memoria]
Write Back	dstE	$R[\text{rB}] \leftarrow \text{valE}$	Guarda/Usa el resultado de la ALU
	dstM		[Guarda el resultado de la memoria]
PC Update	PC	$\text{PC} \leftarrow \text{valP}$	Actualiza el PC

# Operación en cada etapa: visión general

- Todas las instrucciones siguen el mismo patrón
- Cambia lo que se obtiene en cada etapa

	cómputo	call dest	Descripción
Fetch	icode, ifun rA, rB valC valP	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{valC} \leftarrow M_8[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 9$	Lee el byte de instrucción [Lee el byte de registros] Lee la constante Calcula el siguiente PC
Decode	valA, srcA valB, srcB	$\text{valB} \leftarrow R[\%rsp]$	[Lee el operando A] Lee el operando B
Execute	valE Cond Code	$\text{valE} \leftarrow \text{valB} + -8$	Realiza la operación de la ALU [Usa/Modifica el registro CC]
Memory	valM	$M_8[\text{valE}] \leftarrow \text{valP}$	Lectura/Escritura de la memoria
Write Back	dstE dstM	$R[\%rsp] \leftarrow \text{valE}$	Guarda/Usa el resultado de la ALU [Guarda el resultado de la memoria]
PC Update	PC	$\text{PC} \leftarrow \text{valP}$	Actualiza el PC

# Valores calculados

## Búsqueda (*Fetch*)

icode	código de instrucción
ifun	función de la instrucción
rA	registro A
rB	registro B
valC	constante
valP	PC incrementado

## Ejecución (*Execute*)

valE	resultado de la ALU
Cnd	flag de salto / mov. condicional

## Decodificación (*Decode*)

srcA	id. del registro A
srcB	id. del registro B
dstE	registro destino E
dstM	registro destino M
valA	valor del registro A
valB	valor del registro B

## Acceso a memoria/Saltos (*Memory*)

valM	valor leído de memoria
------	------------------------

# Tabla de contenidos

---

1. Instrucciones Y86-64
2. Elementos básicos
3. Descripción de hardware
4. Implementación secuencial

Estructura

Etapas lógicas

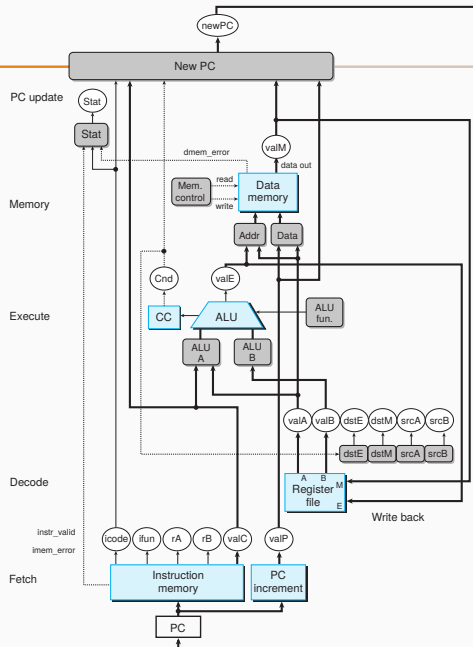
Operación

Implementación

# SEQ Hardware

## Referencia

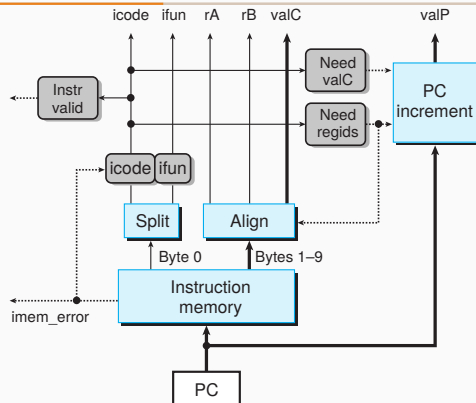
- En celeste: bloques básicos
  - Memorias, ALU, Register File, etc.
- En gris: lógica de control
  - Descripta en HCL
- Círculos blancos: etiquetas para las señales
- Líneas
  - gruesas: palabras de 64 bits
  - finas: palabras de 4/8 bits
  - punteadas: valores de 1 bit



# Lógica de la etapa *fetch*

## Bloques predefinidos

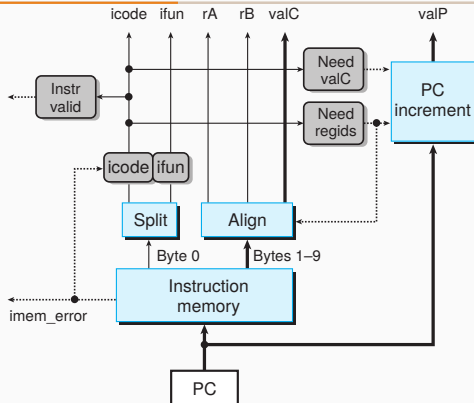
- PC: registro que contiene el PC
- Memoria de instrucciones: lee 10 bytes (de PC a PC+9)
  - Además señala intentos de acceso a direcciones inválidas
- Split: divide el byte la instrucción en icode e ifun
- align: obtiene los campos para rA, rB, y valC



# Lógica de la etapa *fetch*

## Lógica de control

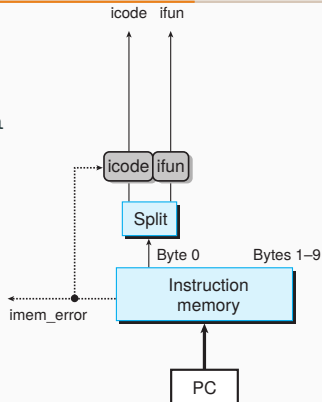
- *Instr. valid*: señala si la instrucción es válida o no.
- *icode*, *ifun*: generan no-op si la dirección es inválida.
- *need regids*: señala si la instrucción posee un byte de registros.
- *need valC*: señala si la instrucción posee un byte constante.





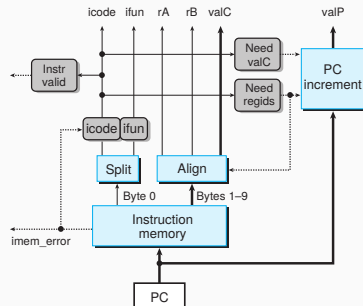
## Lógica de control de la etapa *fetch* en HCL

```
1  # Obtener el código de la
2  # instrucción
3  int icode = [
4      imem_error: INOP;
5      1: imem_icode;
6  ];
7
8  # Obtener la función
9  int ifun = [
10     imem_error: FNONE;
11     1: imem_ifun;
12 ];
```



# Lógica de control de la etapa *fetch* en HCL

Número de byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmovXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB						
rmmovq rA, D(rB)	4	0	rA	rB						
rrmmovq D(rB), rA	5	0	rA	rB						
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn								
call Dest	8	0								
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						



```

1  bool need_regids = icode in
2      {IRRMov, IOPQ, IPUSHQ, IPOPOPQ,
3        IIRMOVQ, IRMMOVQ, IMRMovQ };
4  bool instr_valid = icode in
5      {INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMovQ,
6        IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPOPQ };
  
```

# Lógica de la etapa *decode*

## Banco de registros (*Register File*)

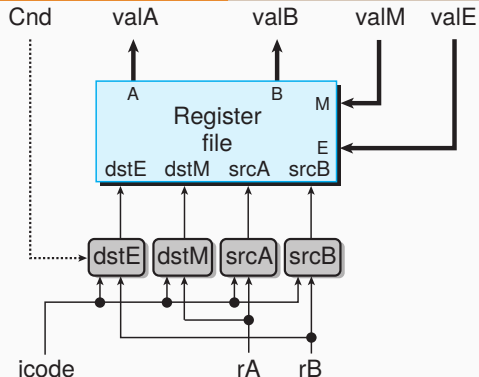
- Puertos de lectura A y B
- Puertos de escritura E y M
- Las direcciones son los IDs de los registros o 15 (0xF)

## Lógica de control

- srcA, srcB: direcciones de los puertos de lectura
- dstE, dstM: direcciones de los puertos de escritura

## Señales

- Cnd: indica si se debe realizar un movimiento condicional o no
  - Se calcula en la etapa de ejecución



## decode: obtención de srcA

Etap	Instrucción	Operación	Descripción	srcA
decode	OPq rA, rB	$valA \leftarrow R[rA]$	Leer operando A	rA
decode	cmovXX rA, rB	$valA \leftarrow R[rA]$	Leer operando A	rA
decode	rmmovq rA, D(rB)	$valA \leftarrow R[rA]$	Leer operando A	rA
decode	popq rA	$valA \leftarrow R[\%rsp]$	Leer stack pointer	%rsp
decode	jxx dest		No necesita registro	F
decode	call dest		No necesita registro	F
decode	ret	$valA \leftarrow R[\%rsp]$	Leer stack pointer	%rsp

```

1  int srcA = [
2      icode in {IRRMovQ, IRMMovQ, IOPQ, IPUSHQ} : rA;
3      icode in {IPOPOPQ, IRET} : RRSP;
4      1 : RNONE; # No se necesita un registro
5  ];

```

## write-back: obtención de dstE

Etapa	Instrucción	Operación	Descripción	dstE
write-back	OPq rA, rB	$R[rB] \leftarrow \text{valE}$	Guarda el resultado	rB
write-back	cmovXX rA, rB	$R[rB] \leftarrow \text{valE}$	Guarda el resultado condicionalmente	rB
write-back	rmmovq rA, D(rB)		Ninguno	F
write-back	popq rA	$R[\%rsp] \leftarrow \text{valE}$	Actualiza el SP	%rsp
write-back	jxx dest		Ninguno	F
write-back	call dest	$R[\%rsp] \leftarrow \text{valE}$	Actualiza el SP	%rsp
write-back	ret	$R[\%rsp] \leftarrow \text{valE}$	Actualiza el SP	%rsp

```

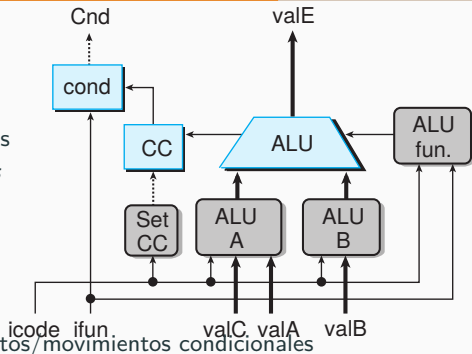
1  int dstE = [
2      icode in {IRRMovQ} && Cnd : rB;
3      icode in {IIRMOVQ, IOPQ} : rB;
4      icode in {IPUSHQ, IPOPQ, ICALL, IRET} : RRSP;
5      1 : RNONE;  # No se necesita un registro
6  ];

```

# Lógica de la etapa *execute*

## Unidades

- ALU
  - Implementa las 4 operaciones
  - Modifica los *condition codes*
- CC
  - Registro de 3 bits para las 3 condiciones
- cond: Computa la bandera de saltos/movimientos condicionales



## Lógica de control

- Set CC: indica si se debe cagar el reg. de códigos de condición
- ALU A: selector de entrada A de la ALU
- ALU B: selector de entrada B de la ALU
- ALU fun: indica la operación que debe realizar la ALU

## execute: entrada A de la ALU

Etapa	Instrucción	Operación	Descripción
execute	OPq rA, rB	$valE \leftarrow valB \text{ OP } valA$	Realizar la operación de la ALU
execute	cmovXX rA, rB	$valE \leftarrow 0 + valA$	Pasar valA a través de la ALU
execute	rmmovq rA, D(rB)	$valE \leftarrow valB + valC$	Calcular la dirección efectiva
execute	popq rA	$valE \leftarrow valB + 8$	Incrementar el SP
execute	jxx dest		no-op
execute	call dest	$valE \leftarrow valB + -8$	Decrementar el SP
execute	ret	$valE \leftarrow valB + 8$	Incrementar el SP

```

1  int aluA = [
2      icode in { IRRMOVQ, IOPQ } : valA;
3      icode in { IIRMOVQ, IRMMOVQ, IMRMVQ } : valC;
4      icode in { ICALL, IPUSHQ } : -8;
5      icode in { IRET, IPOPQ } : 8;
6      # Otras instrucciones no usan la ALU
7  ];

```

## *execute*: operación a realizar en la ALU

Etapa	Instrucción	Operación	Descripción
execute	OPq rA, rB	$\text{valE} \leftarrow \text{valB} \text{ OP } \text{valA}$	Realizar la operación de la ALU
execute	cmovXX rA, rB	$\text{valE} \leftarrow 0 + \text{valA}$	Pasar valA a través de la ALU
execute	rmmovq rA, D(rB)	$\text{valE} \leftarrow \text{valB} + \text{valC}$	Calcular la dirección efectiva
execute	popq rA	$\text{valE} \leftarrow \text{valB} + 8$	Incrementar el SP
execute	jxx dest		no-op
execute	call dest	$\text{valE} \leftarrow \text{valB} + -8$	Decrementar el SP
execute	ret	$\text{valE} \leftarrow \text{valB} + 8$	Incrementar el SP

```

1  int alufun = [
2      icode == IOPQ : ifun;
3      1 : ALUADD;
4  ];

```



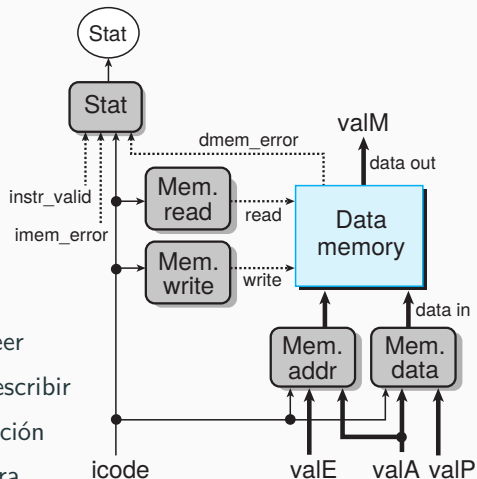
# Lógica de la etapa *memory*

## Memoria

- Lee o escribe en la memoria

## Lógica de control

- stat: computa el estado de la instrucción
- Mem. read: indica si hay que leer
- Mem. write: indica si hay que escribir
- Mem. addr.: selecciona la dirección
- Mem. data: selecciona la palabra



# memory: estado de ejecución

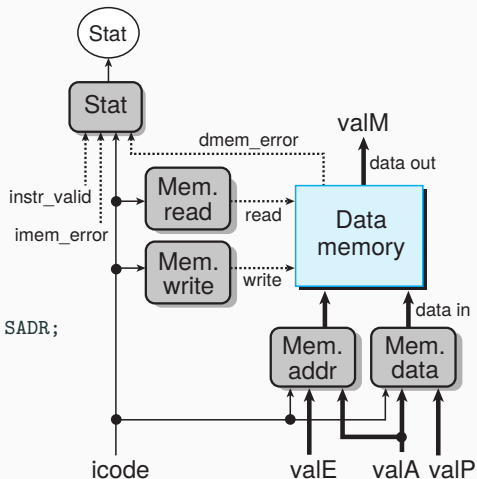
## Lógica de control

- stat: computa el estado del procesador

```

1  int Stat = [
2      imem_error || dmem_error : SADR;
3      !instr_valid : SINS;
4      icode == IHALT : SHLT;
5      1 : SAOK;
6  ];

```



## memory: obtención de la dirección

Etapa	Instrucción	Operación	Descripción
memory	OPq rA, rB		no-op
memory	cmovXX rA, rB		no-op
memory	rmmovq rA, D(rB)	$M_8[\text{valE}] \leftarrow \text{valA}$	Escribe en memoria
memory	popq rA	$\text{valM} \leftarrow M_8[\text{valA}]$	Leer del stack
memory	jxx dest		no-op
memory	call dest	$M_8[\text{valE}] \leftarrow \text{valP}$	Escribir en el stack
memory	ret	$\text{valM} \leftarrow M_8[\text{valA}]$	Leer la dirección de retorno

```

1  int mem_addr = [
2      icode in {IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ} : valE;
3      icode in {IPOPQ, IRET} : valA;
4      # Otras instrucciones no necesitan direcciones
5  ];

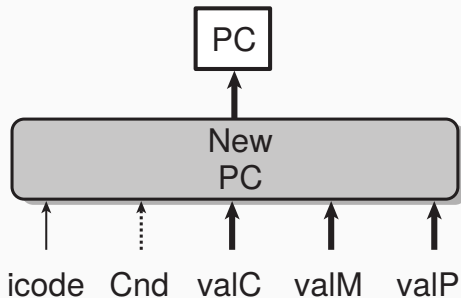
```

## memory: ¿hay que leer?

Etapa	Instrucción	Operación	Descripción
memory	OPq rA, rB		no-op
memory	cmovXX rA, rB		no-op
memory	rmmovq rA, D(rB)	$M_8[valE] \leftarrow valA$	Escribe en memoria
memory	popq rA	$valM \leftarrow M_8[valA]$	Leer del stack
memory	jxx dest		no-op
memory	call dest	$M_8[valE] \leftarrow valP$	Escribir en el stack
memory	ret	$valM \leftarrow M_8[valA]$	Leer la dirección de retorno

```
1  bool mem_read = icode in { IMRMOVQ, IPOPOPQ, IRET };
```

## Lógica de la etapa *PC update*



- Selecciona el siguiente valor del PC

## PC update

Instrucción	Operación	Descripción
OPq rA, rB	$PC \leftarrow valP$	Actualiza el PC
cmovXX rA, rB	$PC \leftarrow valP$	Actualiza el PC
rmmovq rA, D(rB)	$PC \leftarrow valP$	Actualiza el PC
popq rA	$PC \leftarrow valP$	Actualiza el PC
jxx dest	$PC \leftarrow Cnd ? valC : valP$	Actualiza el PC
call dest	$PC \leftarrow valC$	Carga el PC con dest
ret	$PC \leftarrow valM$	Carga el PC con la dirección de retorno

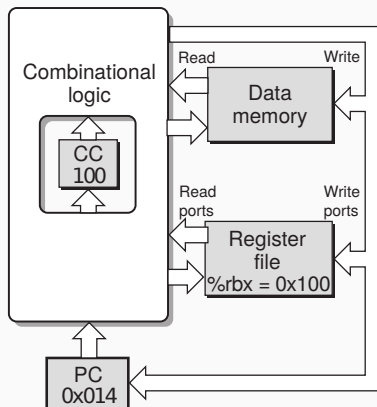
```

1  int new_pc = [
2      icode == ICALL : valC ;
3      icode == IJXX && Cnd : valC ;
4      icode == IRET : valM ;
5      1 : valP ;
6  ];

```

# Funcionamiento de la implementación SEQ

① Beginning of cycle 3



## Estado

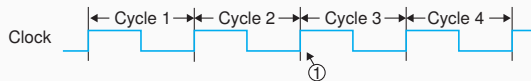
- Registro del PC
- Registro de los CC
- Memoria de datos
- Banco de registros

*Todos actualizados con el clock*

## Lógica combinacional

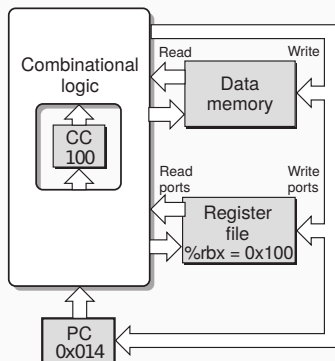
- ALU
- Lógica de control
- Lecturas de la memoria
  - Memoria de instrucciones
  - Banco de registros
  - Memoria de datos

# Funcionamiento de la implementación SEQ



Ciclo 1:	0x000:	irmovq \$0x100,%rbx	##%rbx <-- 0x100
Ciclo 2:	0x00a:	irmovq \$0x200,%rdx	##%rdx <-- 0x200
Ciclo 3:	0x014:	addq %rdx,%rbx	##%rbx <-- 0x300 CC <-- 000
Ciclo 4:	0x016:	je dest	#Not taken
Ciclo 5:	0x01f:	rmmovq %rbx,0(%rdx)	#M[0x200] <-- 0x300

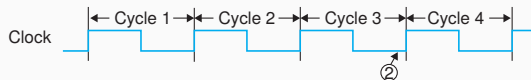
① Beginning of cycle 3



- El estado se establece según la segunda instrucción `irmovq`
- La lógica combinacional empieza a responder a los cambios de estado

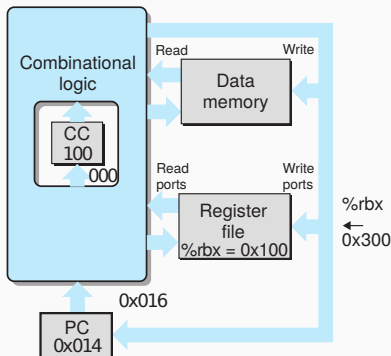


# Funcionamiento de la implementación SEQ



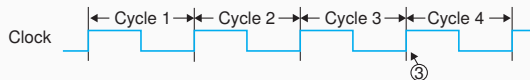
Ciclo 1:	0x000:	irmovq \$0x100,%rbx	##%rbx <-- 0x100
Ciclo 2:	0x00a:	irmovq \$0x200,%rdx	##%rdx <-- 0x200
Ciclo 3:	0x014:	addq %rdx,%rbx	##%rbx <-- 0x300 CC <-- 000
Ciclo 4:	0x016:	je dest	##Not taken
Ciclo 5:	0x01f:	rmmovq %rbx,0(%rdx)	##M[0x200] <-- 0x300

② End of cycle 3



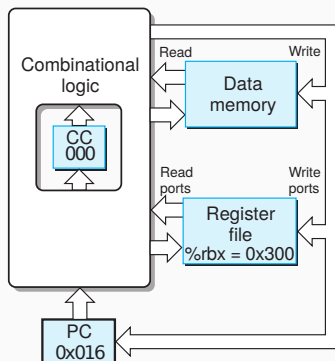
- El estado se establece según la segunda instrucción `irmovq`
- La lógica combinacional genera los resultados de la instrucción `addq`

# Funcionamiento de la implementación SEQ



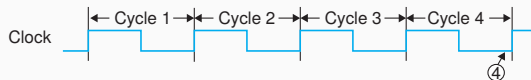
Ciclo 1:	0x000:	irmovq \$0x100,%rbx	##%rbx <-- 0x100
Ciclo 2:	0x00a:	irmovq \$0x200,%rdx	##%rdx <-- 0x200
Ciclo 3:	0x014:	addq %rdx,%rbx	##%rbx <-- 0x300 CC <-- 000
Ciclo 4:	0x016:	je dest	##Not taken
Ciclo 5:	0x01f:	rmmovq %rbx,0(%rdx)	##M[0x200] <-- 0x300

## ③ Beginning of cycle 4



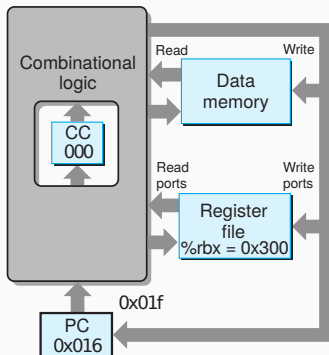
- El estado se establece según la instrucción `addq`
- La lógica combinacional empieza a responder a los cambios de estado

# Funcionamiento de la implementación SEQ



Ciclo 1:	0x000:	irmovq \$0x100,%rbx	﻿#%rbx <-- 0x100
Ciclo 2:	0x00a:	irmovq \$0x200,%rdx	﻿#%rdx <-- 0x200
Ciclo 3:	0x014:	addq %rdx,%rbx	﻿#%rbx <-- 0x300 CC <-- 000
Ciclo 4:	0x016:	je dest	﻿#Not taken
Ciclo 5:	0x01f:	rmmovq %rbx,0(%rdx)	﻿#M[0x200] <-- 0x300

④ End of cycle 4



- El estado se establece según la instrucción `addq`
- La lógica combinatorial genera los resultados de la instrucción `je`

# Licencia del estilo de beamer

Obtén el código de este estilo y la presentación demo en

`github.com/pamoreno/mtheme`

El estilo *en sí* está licenciado bajo la Creative Commons Attribution-ShareAlike 4.0 International License. El estilo es una modificación del creado por Matthias Vogelgesang, disponible en

`github.com/matze/mtheme`

