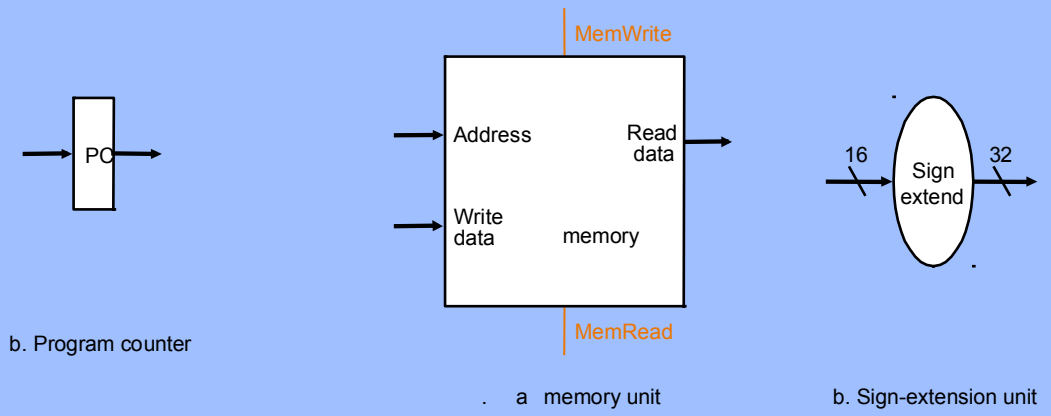
Organización del Computador

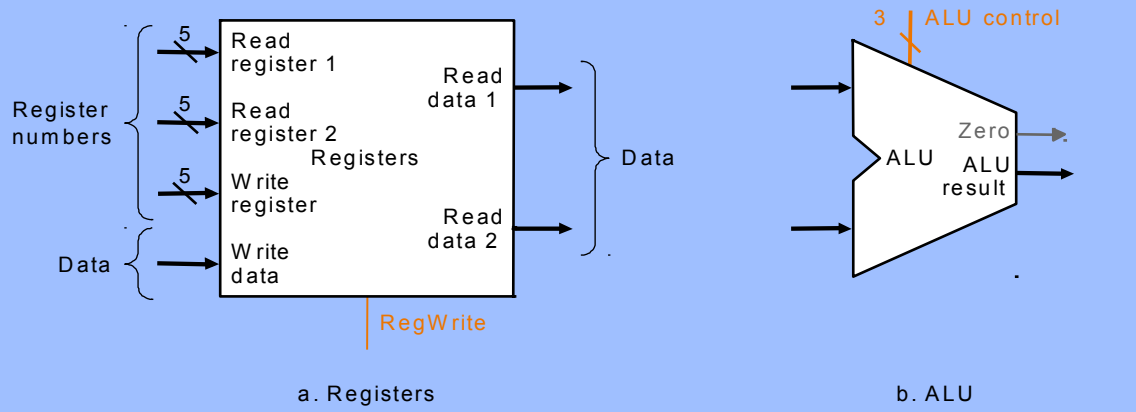
Microprogramación

Consideraciones

- Una única ALU, una única Memoria, un único Banco de Registros.
- Problemas con el uso de los recursos??
- Dos tipos de problemas:
  - Una instrucción utiliza un mismo recurso varias veces y se pierden los valores anteriores.
  - Una instrucción utiliza un mismo recurso en la misma etapa para dos o mas cosas distintas.

Algunos componentes disponibles





Diseño: Pasos necesarios

- **1er Paso**: Analizar el conjunto de instrucciones para determinar los requerimientos del Camino de Datos.
- **2o Paso**: Seleccionar los componentes.
- **3er Paso**: Construir el Camino de Datos según los requerimientos.
- **4o Paso**: Analizar la implementación de cada instrucción para determinar las señales de control necesarias.
- **5o Paso**: Construir el Control.

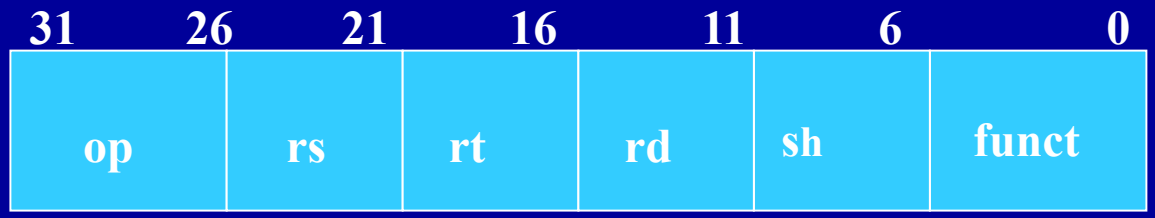
Cinco Etapas de Ejecución

1. Fetch de Instrucción (Fetch/IF)
2. Decodificación (y lectura de registros) (Decode/ID)
3. Ejecución (o Cálculo de Dirección de memoria) (Execution/EX)
4. Acceso a datos en memoria (Mem)
5. Escritura en registros (Write Back/WB)

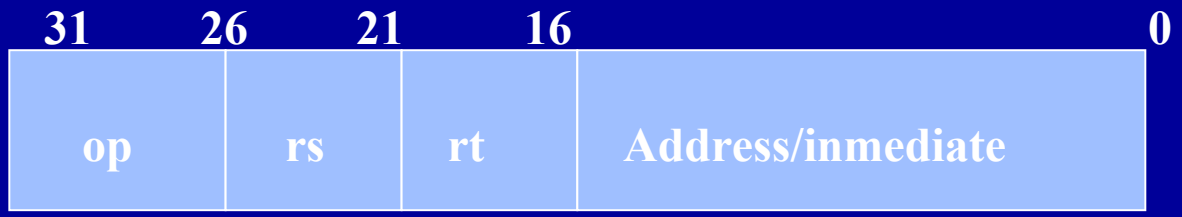
Formato MIPS de Instrucción

Son todas de 32 bits. Tres formatos:

- Tipo R
  - Aritméticas



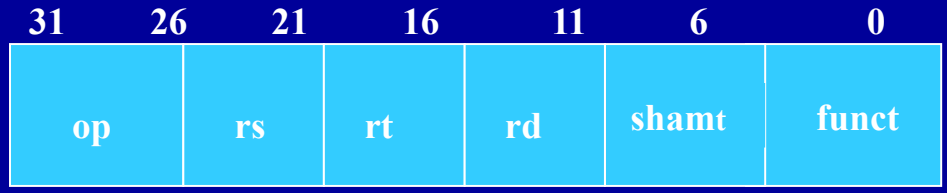
- Tipo I
  - Transferencia, salto
  - Operaciones con operando inmediato



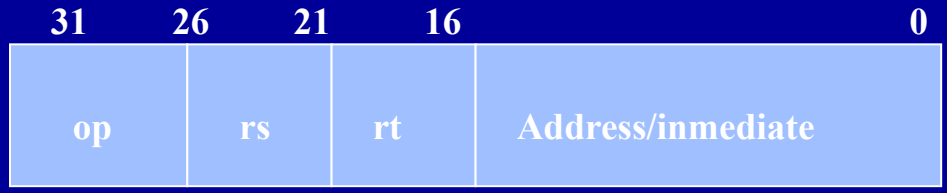
- Tipo J
  - Saltos

Formato MIPS

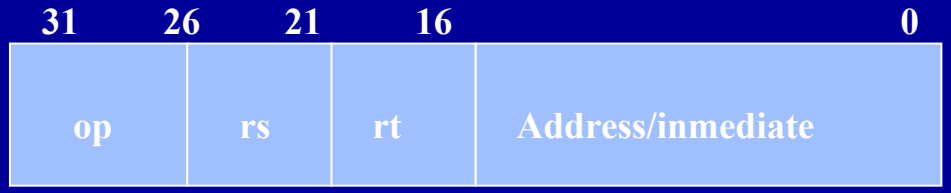
- ADD y SUB
  - (R[rd] = R[rs] op R[rt])
  - addu rd ,rs ,rt
  - subu rd, rs ,rt



- LOAD and STORE
  - lw rt, rs, inm16
    - R[rt] = Mem[R[rs]+sign_ext(Inm16)];
  - sw rt, rs, inm16
    - Mem[R[rs]+sign_ext(Inm16)]= R[rt];



- BRANCH
  - beq rs, rt, inm16
  - if (R[rs]==R[rt]) then
    PC=PC+(sign_ext(Inm16)*4)

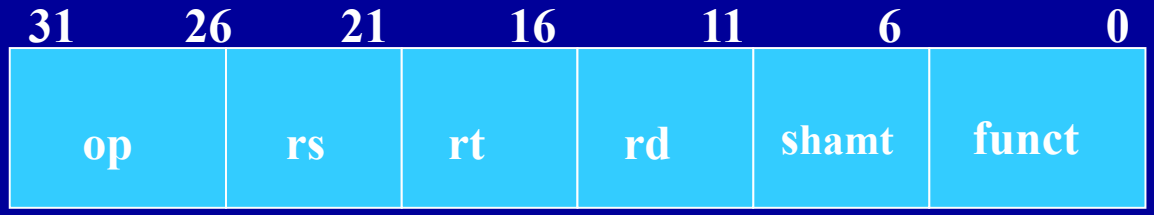


- JUMP
  - J target
  - PC<31:2> =PC<31:28>,(target <25:0> << 2

RTL

- Cada instrucción está formada por un conjunto de microoperaciones.
- RTL (Registrer Tranfer Language): se utilizada para determinar la secuencia exacta de microoperaciones.
- Ejemplo (Fetch en Marie):
  - t1: MAR <- (PC)
  - t2: MBR <- mem[MAR], PC <- (PC) +1
  - t3: IR <- (MBR)

1º Paso: Tipo R (add, sub..)



- R[rd] = R[rs] op R[rt] y PC=PC+4

- RTL

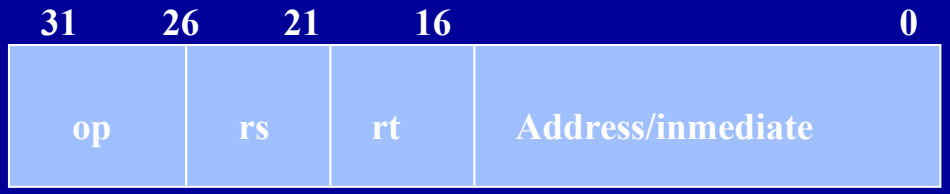
  T1: IR ← mem[PC]; PC←PC+4

  T2: A← R[rs]; B←R[rt]

  T3: ALUOut←A op B

  T4: R[rd] ← ALUOut

Branch



BEQ if (R[rs]==R[rt]) then PC=PC+(sign_ext(Inm16)*4)

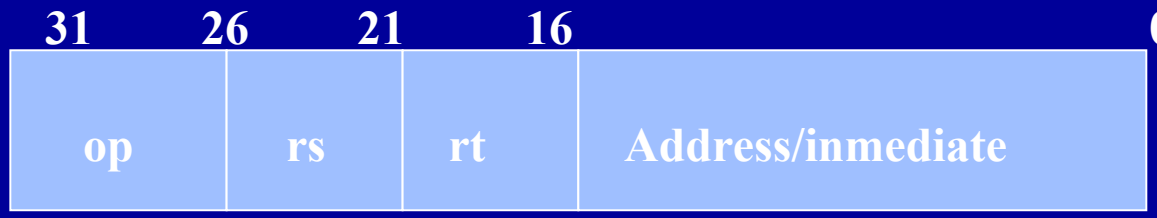
- RTL

T1: IR← mem[PC];
PC PC + 4

T2: A← R[rs] (Se guardan para el ciclo siguiente)
B ← R[rt]
ALUOut← PC + signextend(imm16) << 2 (calcula la dir. Del salto)

T3: Comparar A y B
PC ← ALUOut Si el flag Zero esta activo

LOAD



- LOAD R[rt] = Mem[R[rs]+sign_ext(Inm16)];

- RTL

  T1: IR ← mem[PC]
  PC ←PC + 4

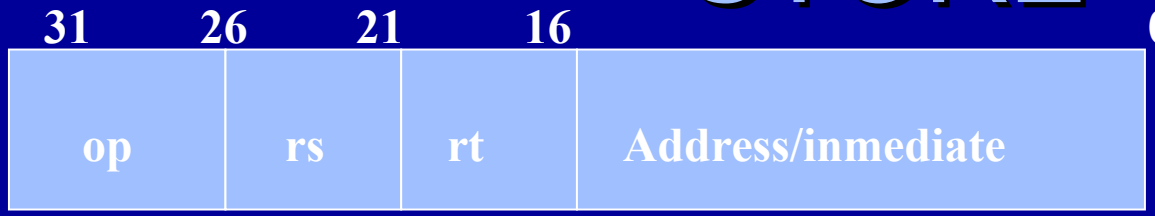
  T2: A ← R[rs]
  B ← R[rt] (B no se usa)

  T3: ALUOut ← A + signextend(imm16) (Calcula la dir.)

  T4: MBR ← Mem[ALUOut]

  T5: R[rt] ← MDR

STORE



- STORE Mem[R[rs]+sign_ext(Inm16)]<-- R[rt];

- RTL

  T1: IR ← mem[PC]
  PC ← PC + 4

  T2: A ← R[rs]
  B ← R[rt] (valor a escribir)

  T3: ALUOut ← A + signextend(imm16)

  T4: Mem[ALUOut] ← B

JUMP

- Jump: PC<31:2> ← PC<31:28>,(target <25:0> << 2)
  - Calcula la dirección concatenando los 26 bits del operando

- RTL

  T1: IR ← mem[PC]
  PC ← PC + 4

  T2: NADA!

  T3 PC<31:2> ← PC<31:28>, (IR<25:0> << 2)

Resumen de las etapas

Cycle	Instruction type	action
IF	all	`IR ← Memory[PC]` `PC ← PC + 4`

ID	all	`A ← Reg[rs]`
		`B ← Reg[rt]`
		`ALUOut ← PC + (imm16 <<2)`

EX	R-type
	Load/Store
	Branch
	Jump
		`ALUOut ← A op B`
		`ALUOut ← A + sign-extend(imm16)`
		`if (A == B) then PC ← ALUOut`
		`PC ← PC[31:28] || (IR[25:0] <<2)`

MEM	Load Store	`MDR ← Memory[ALUOut]` `Memory[ALUOut] ← B`
WB	R-type Load	`Reg[rd] ← ALUOut` `Reg[rt] ← MDR`

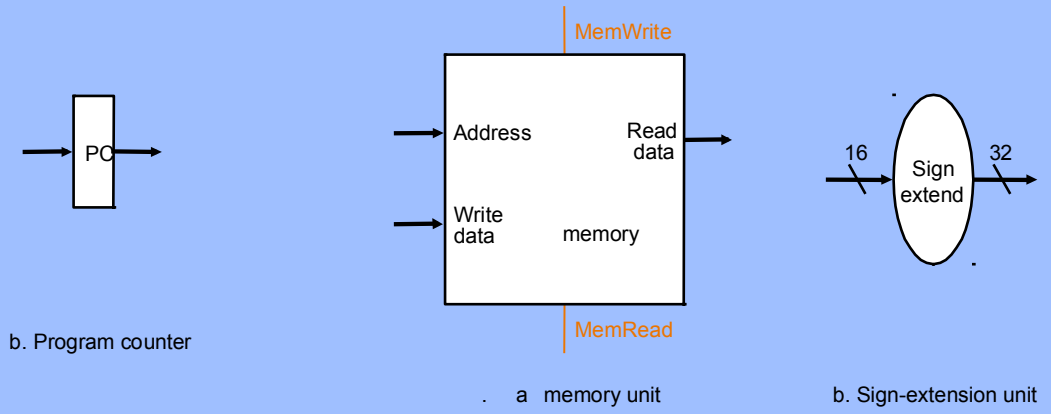
Número de Ciclos

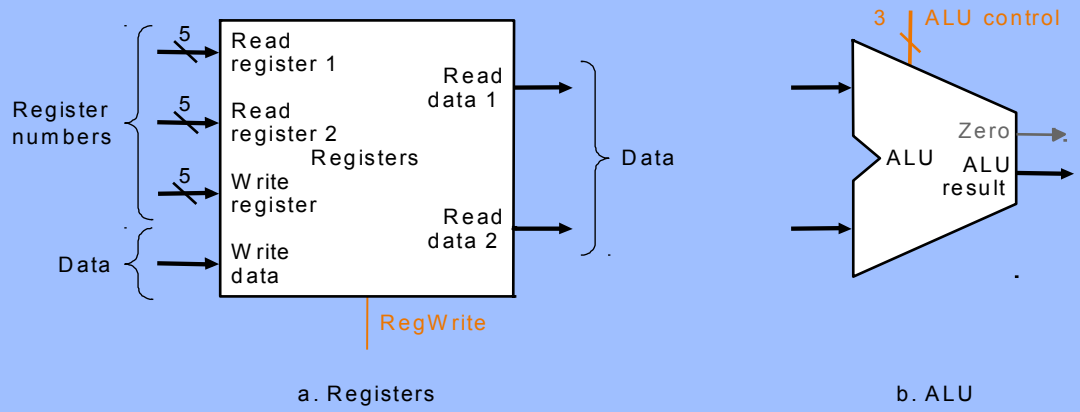
- Brach y Jump: 3 ciclos (IF,ID,EX)
- Las tipo R: 4 ciclos (IF,ID,EX,WB)
- STORE: 4 ciclos (IF,ID,EX,MEM)
- LOAD: 5 (IF,ID,EX,MEM,WB)

1er Paso: Requerimientos del Conjunto de Instrucciones

- Memoria
  - Para Instrucciones y Datos
- Registros (32x32)
  - Leer rs, Leer rt
  - Escribir rt o rd
- PC, MBR
- A, B para datos intermedios, ALUOut (retener salida ALU)
- Extensor de signo (16 a 32)
- Sumar y Restar registros y/o valores inmediatos
- Operaciones lógicas(and/or) registros y/o valores inmediatos
- Sumar 4 al PC o 4+inmediato extendido *4

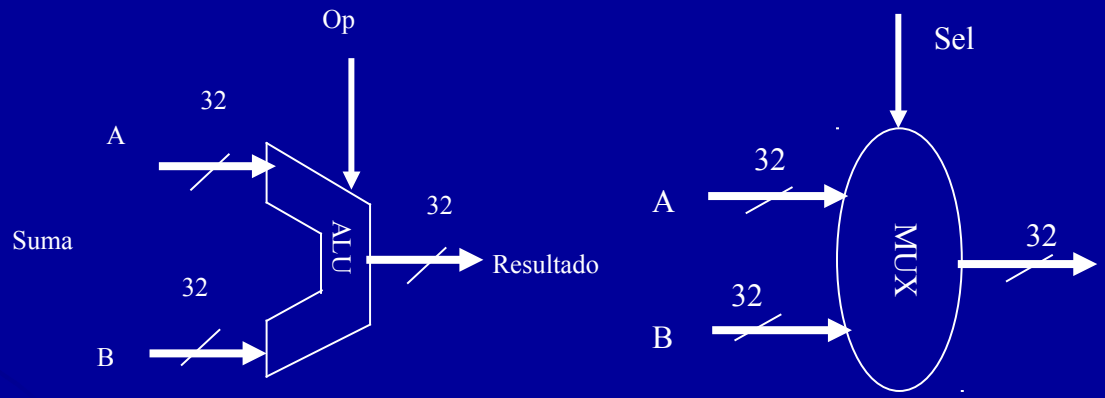
2º Paso: Componentes del Camino de Datos





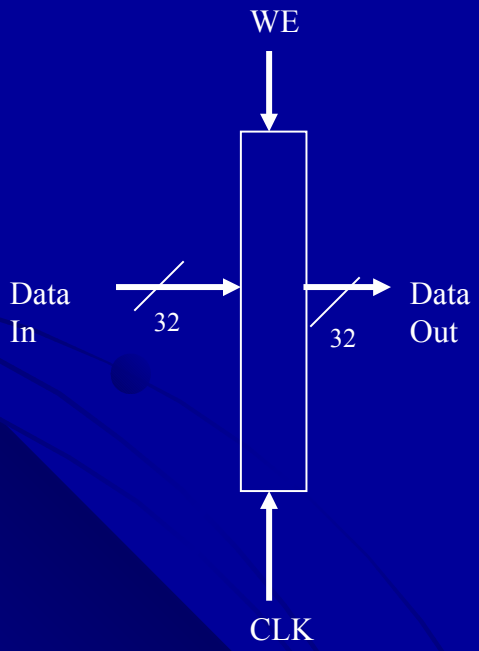
2º Paso: Componentes del Camino de Datos

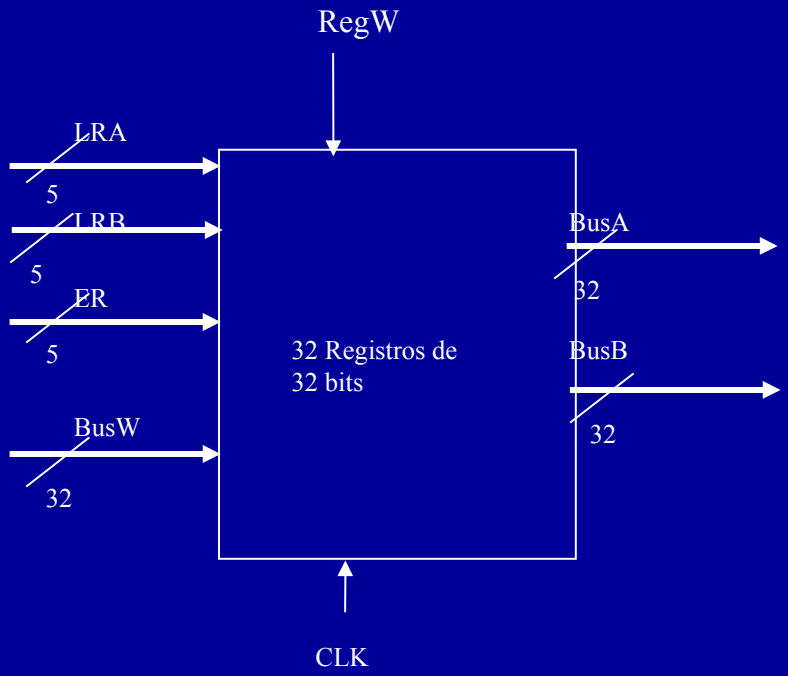
- Elementos Combinacionales
  - ALU y Multiplexor



2º Paso: Componentes del Camino de Datos

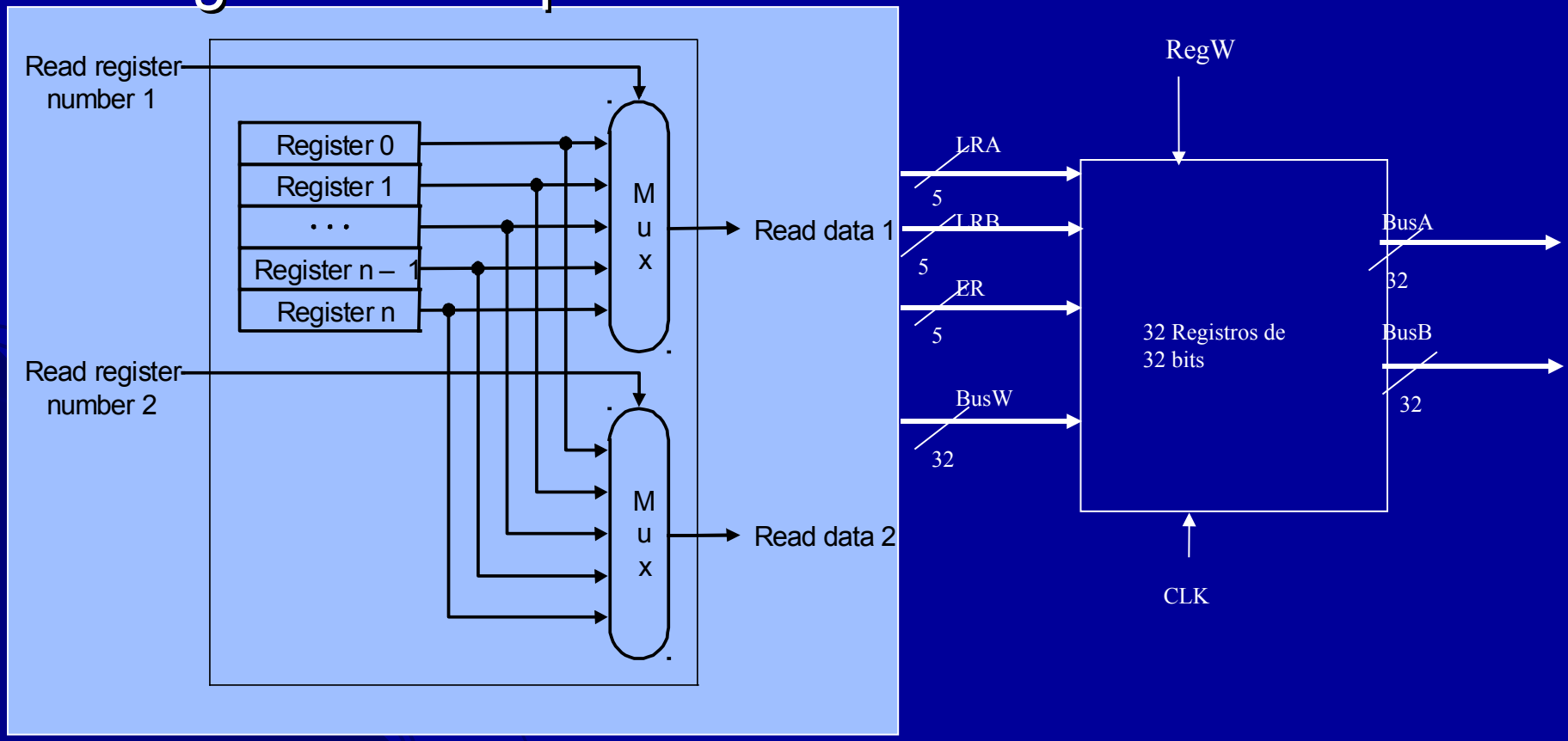
- Elementos de Almacenamiento: Banco de Registros





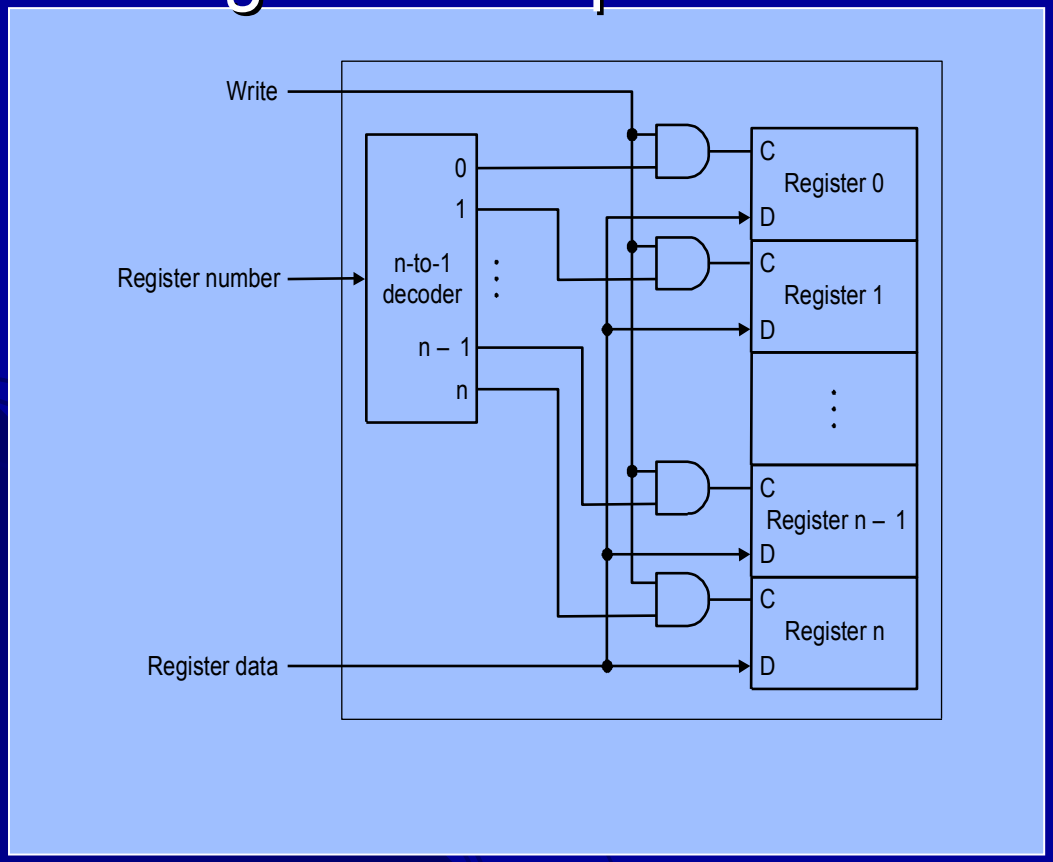
2º Paso: Componentes del Camino de Datos

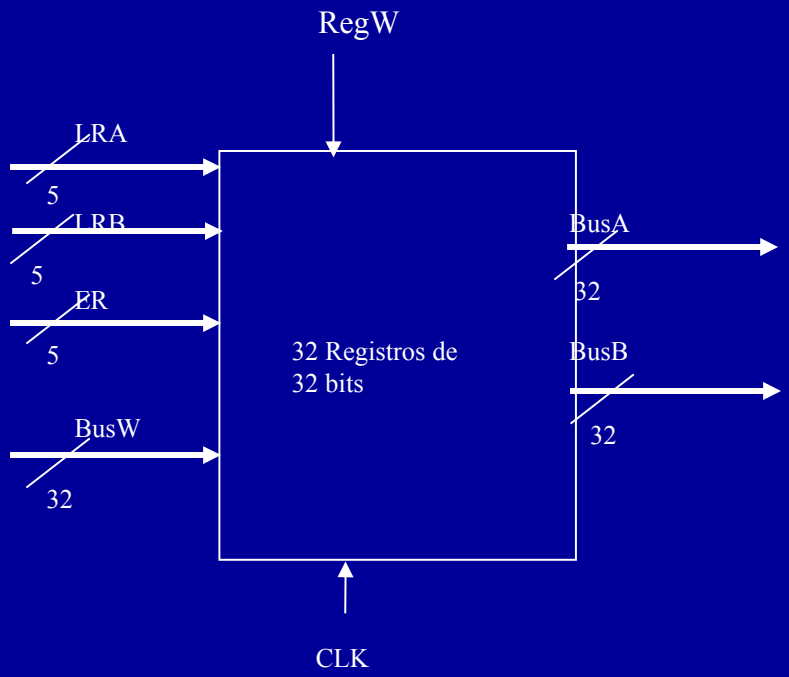
- Elementos de Almacenamiento: Banco de Registros: dos puertos de lectura



2º Paso: Componentes del Camino de Datos

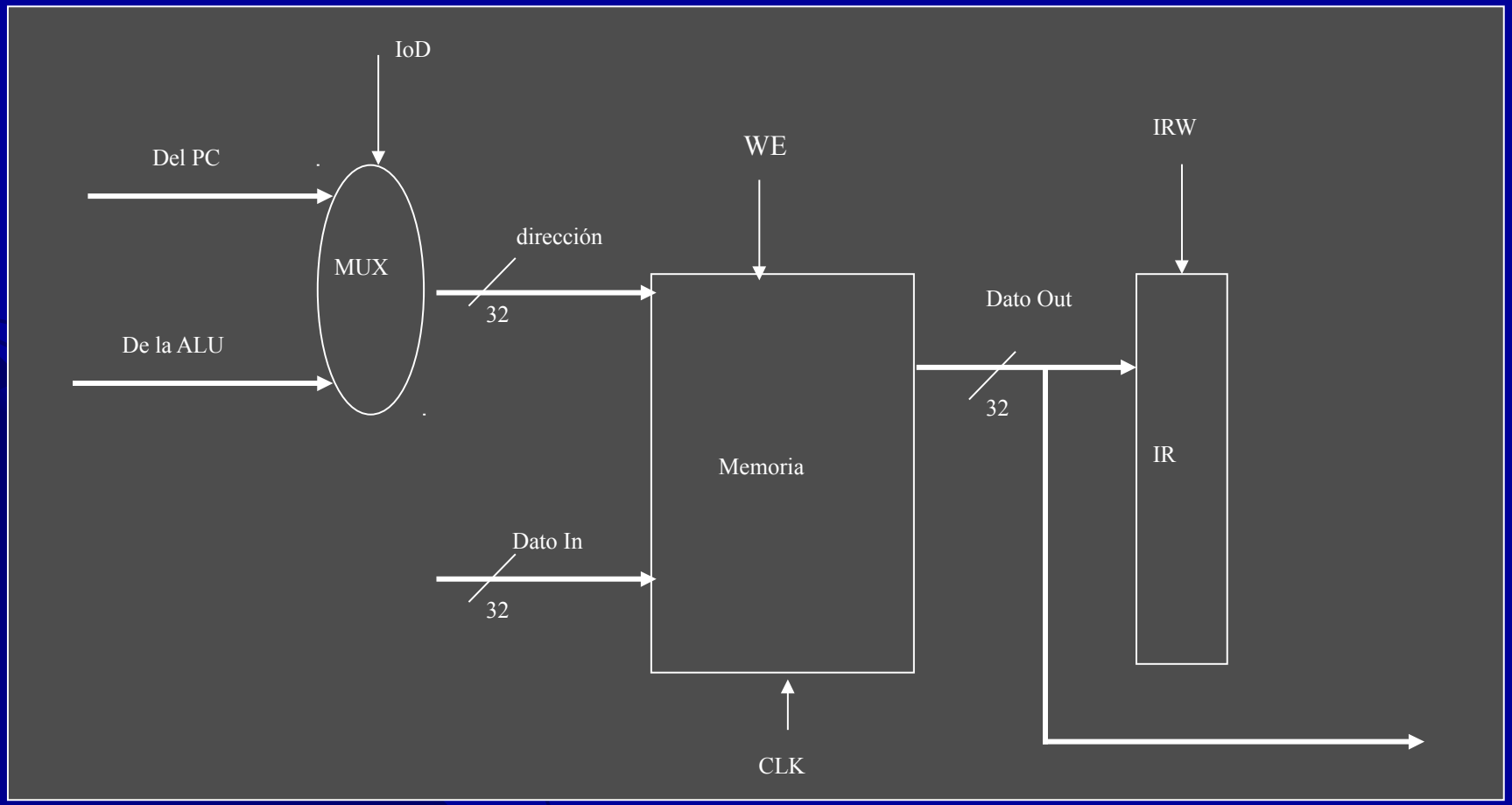
- Elementos de Almacenamiento: Banco de Registros: un puerto de escritura





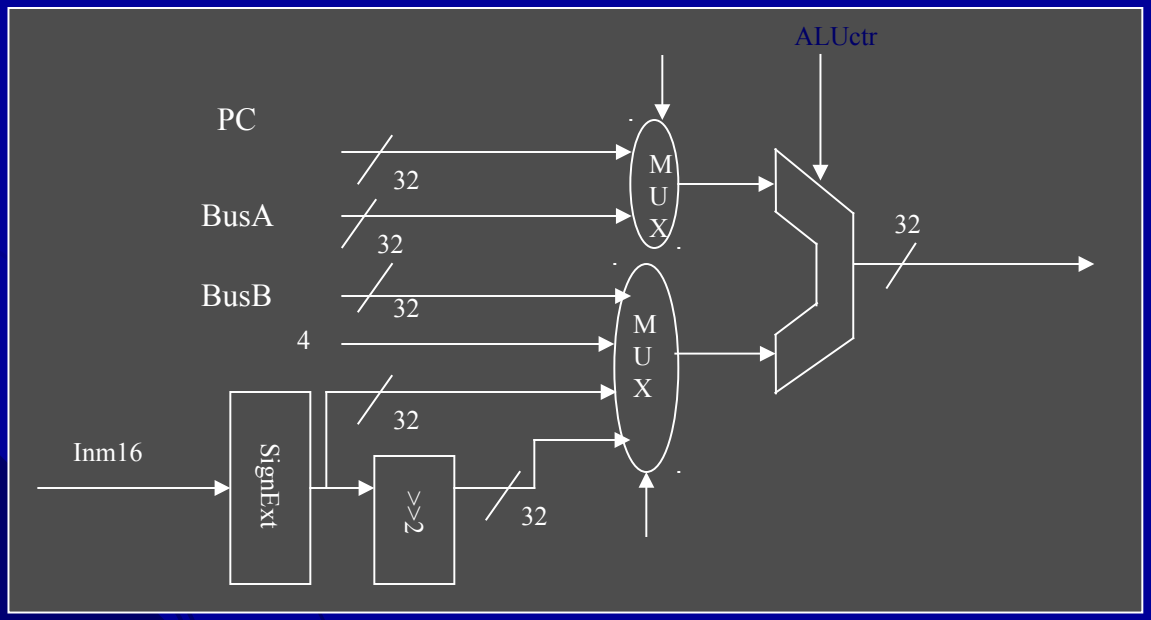
2º Paso: Componentes del Camino de Datos

- Una sola memoria para instrucciones y datos

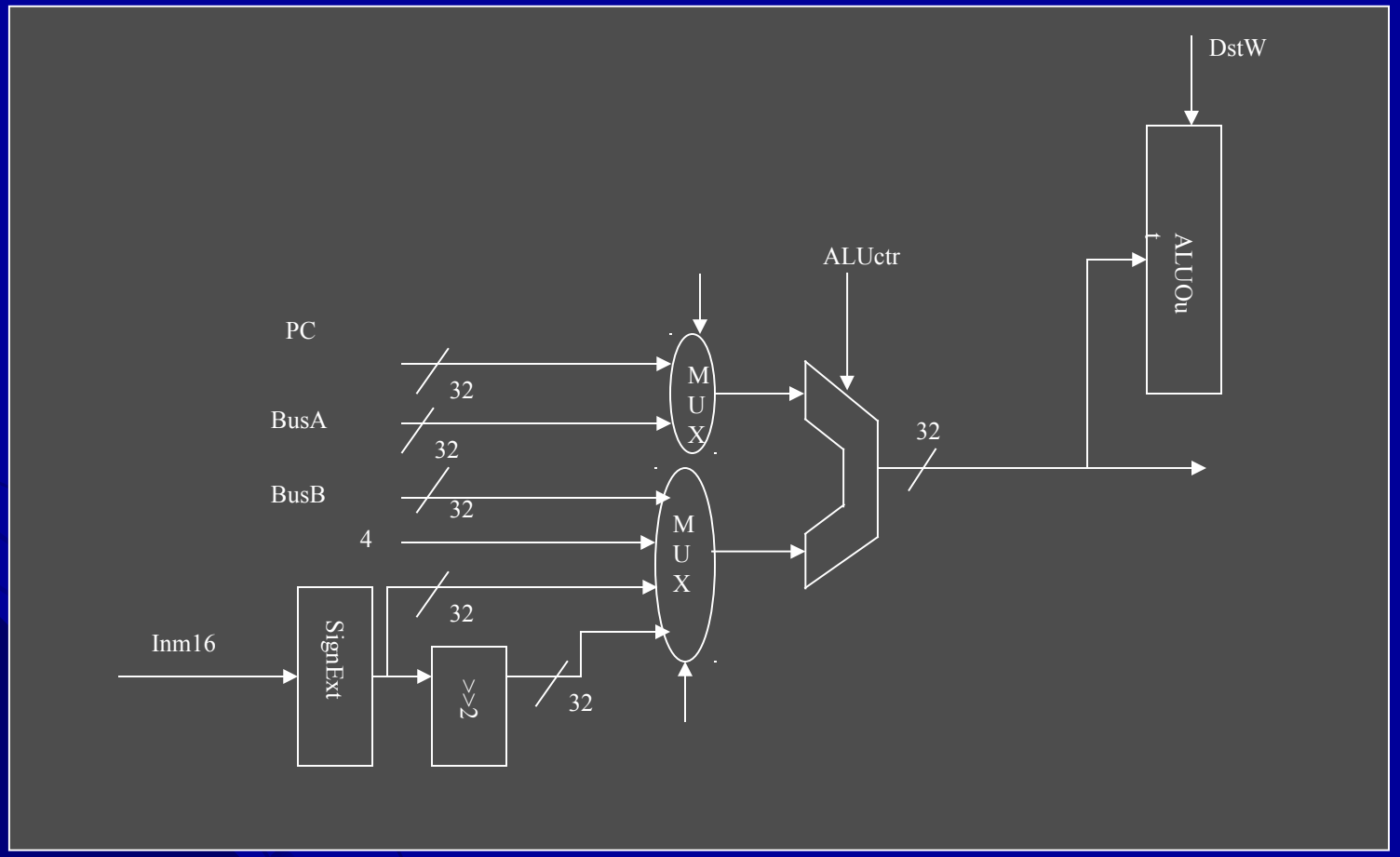


3er Paso: Reuso de Unidades Funcionales: ALU única

- ALU debe realizar
  - operaciones sobre registros
  - base + desplazamiento para loads y stores
  - dirección destino de salto: registro + signo_ext(inmm16) * 4
  - PC=PC+4

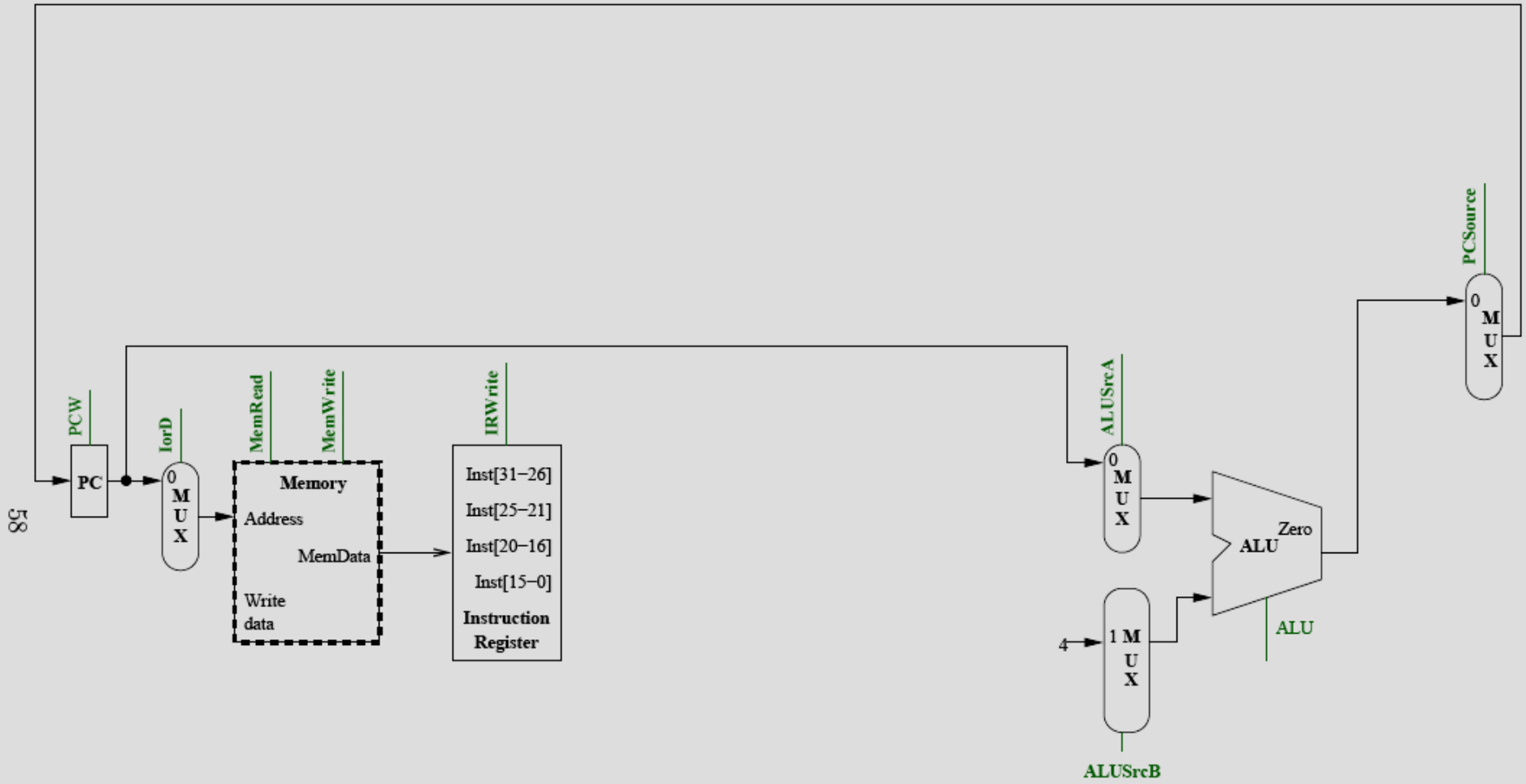


3er Paso: Registro ALUout

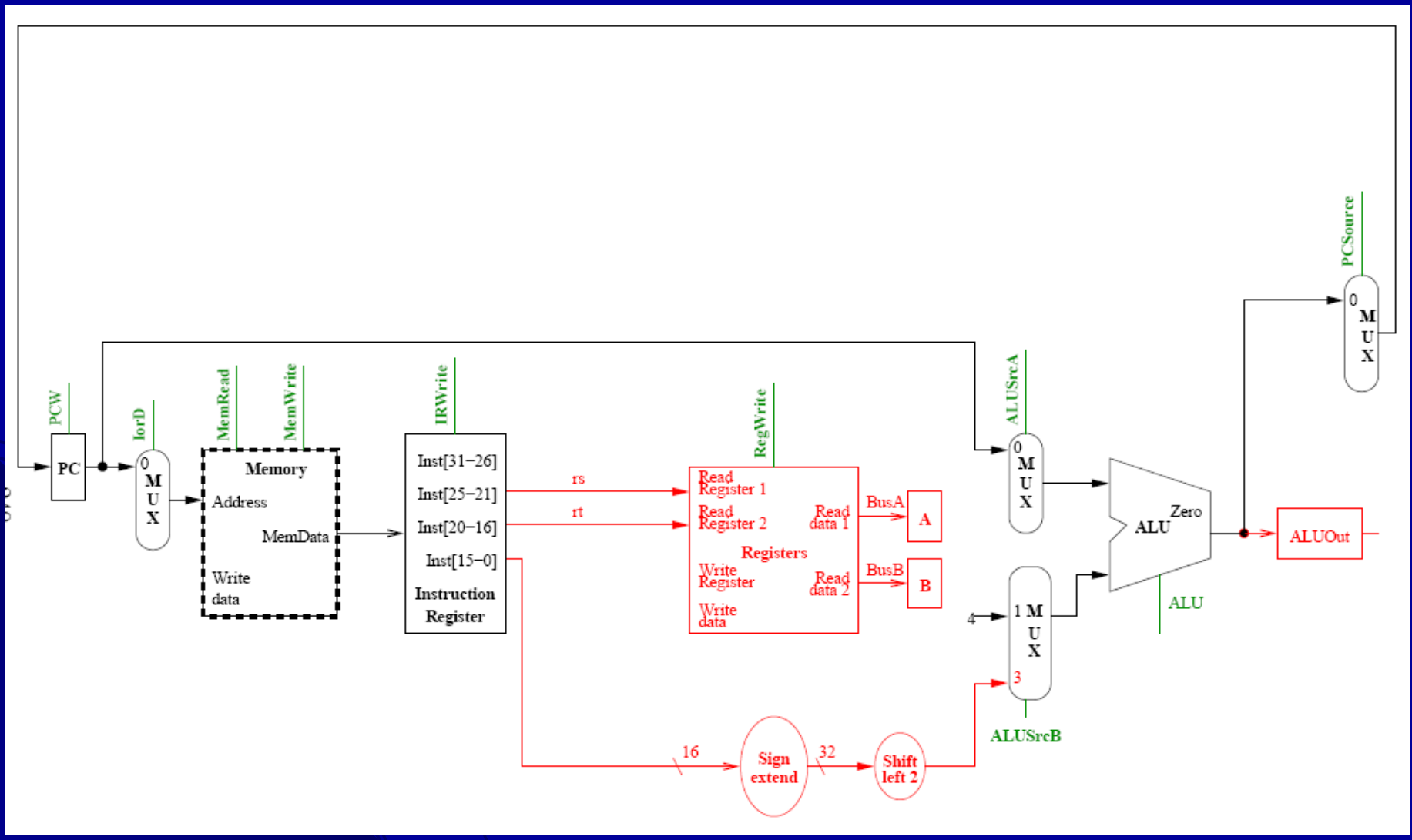


3er Paso: Fetch - IF

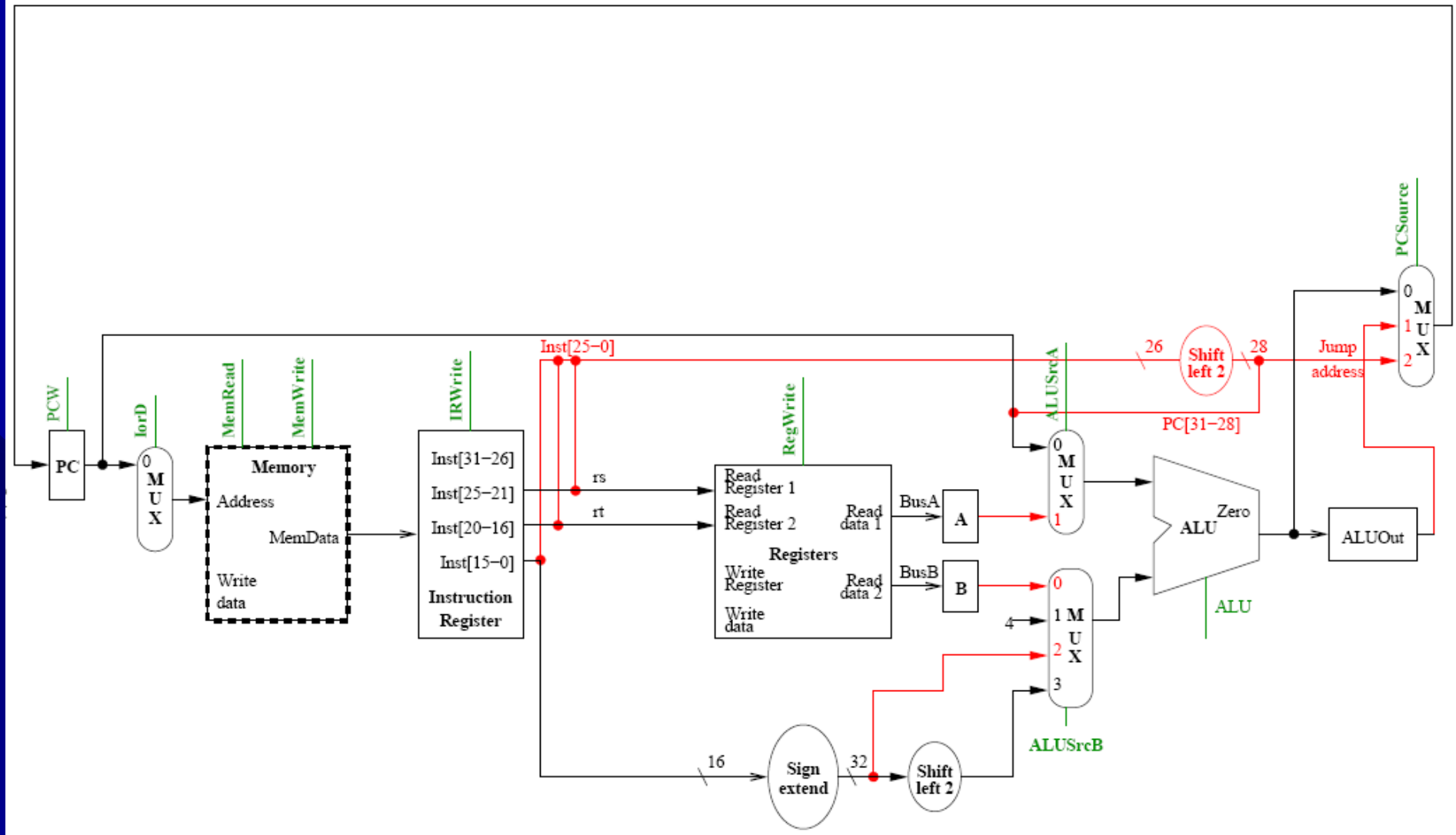
- Mem[PC]; PC<-- PC+4 (código secuencial)



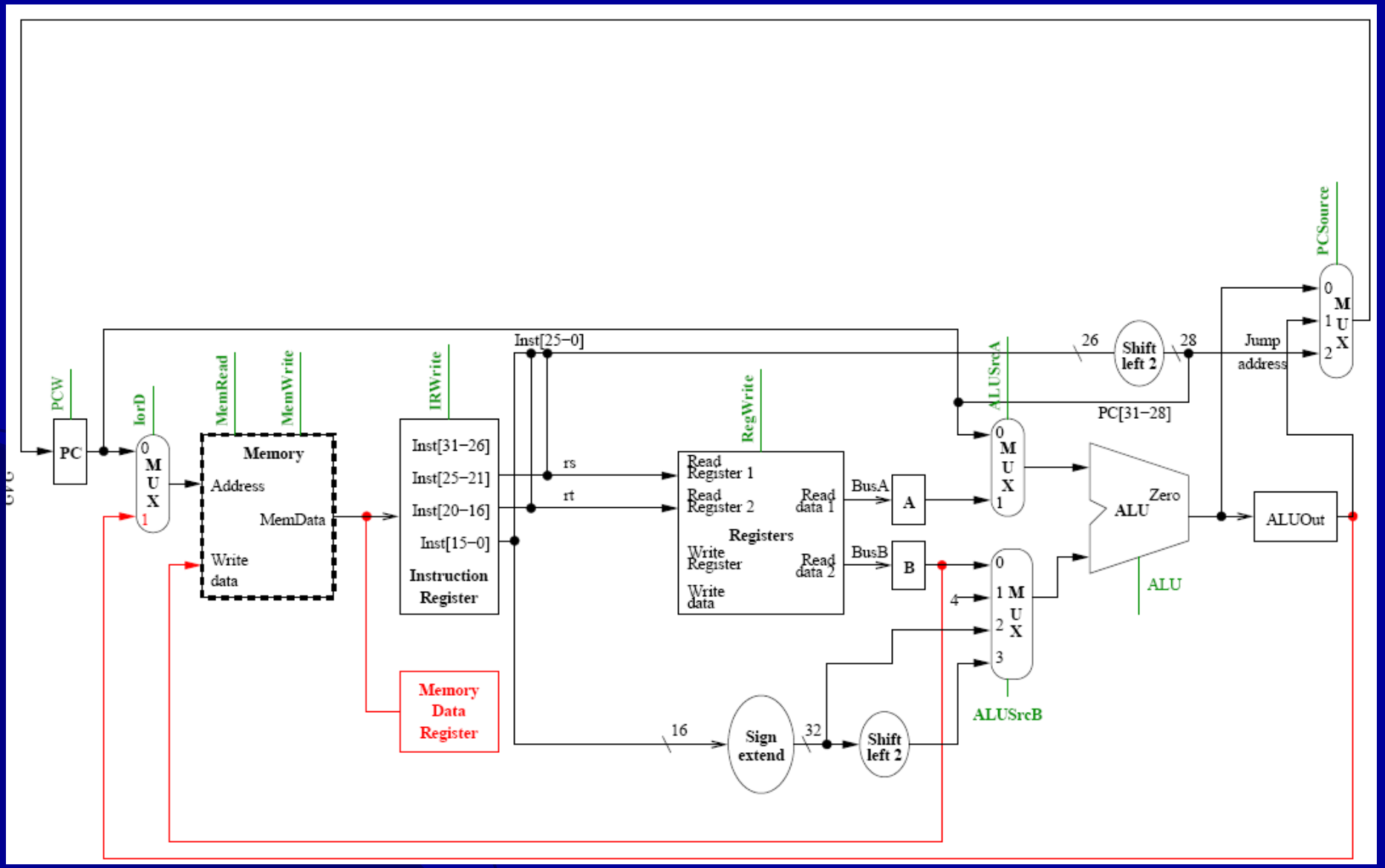
3er Paso: Decode - ID



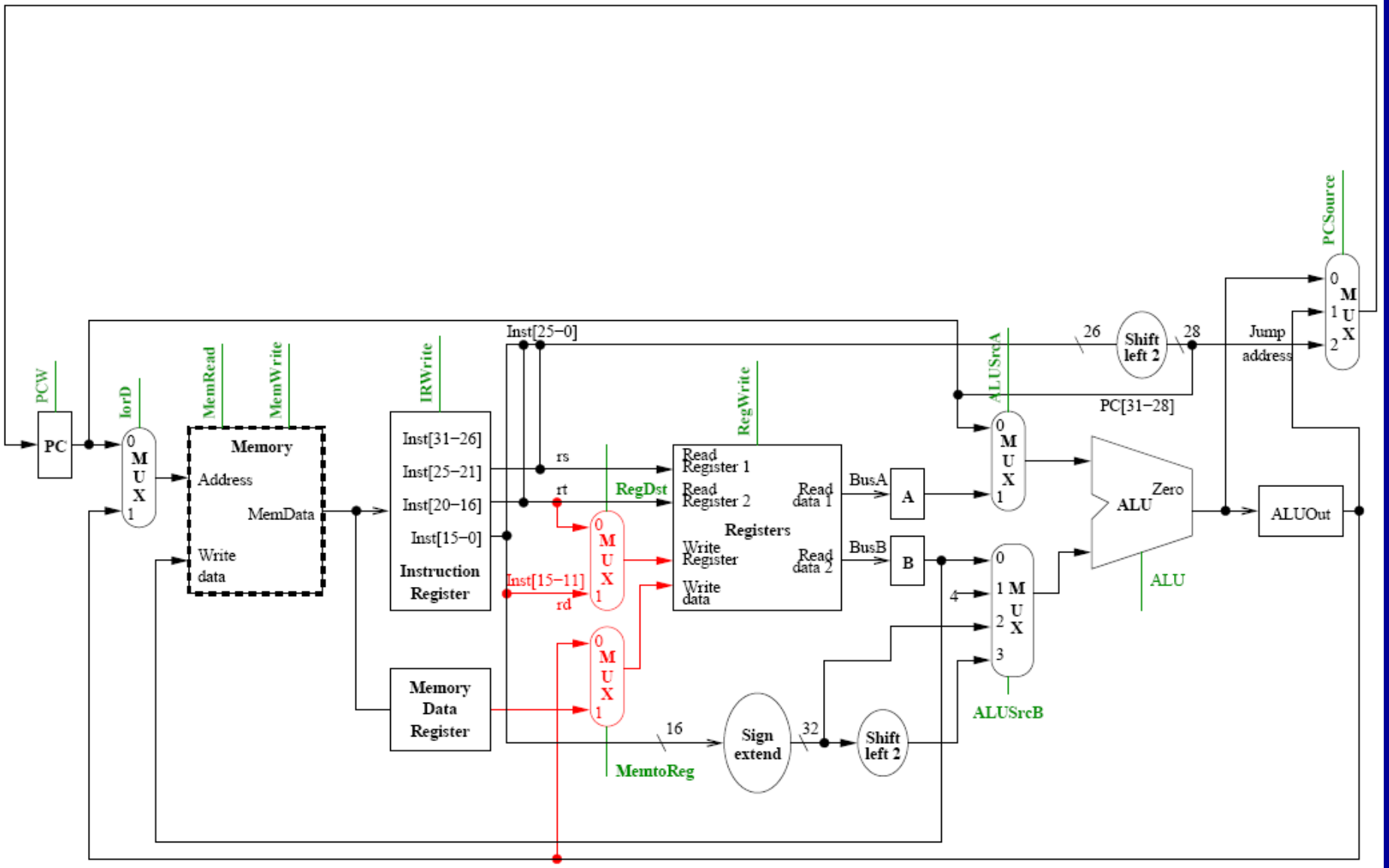
3er Paso: DataPath - Ex



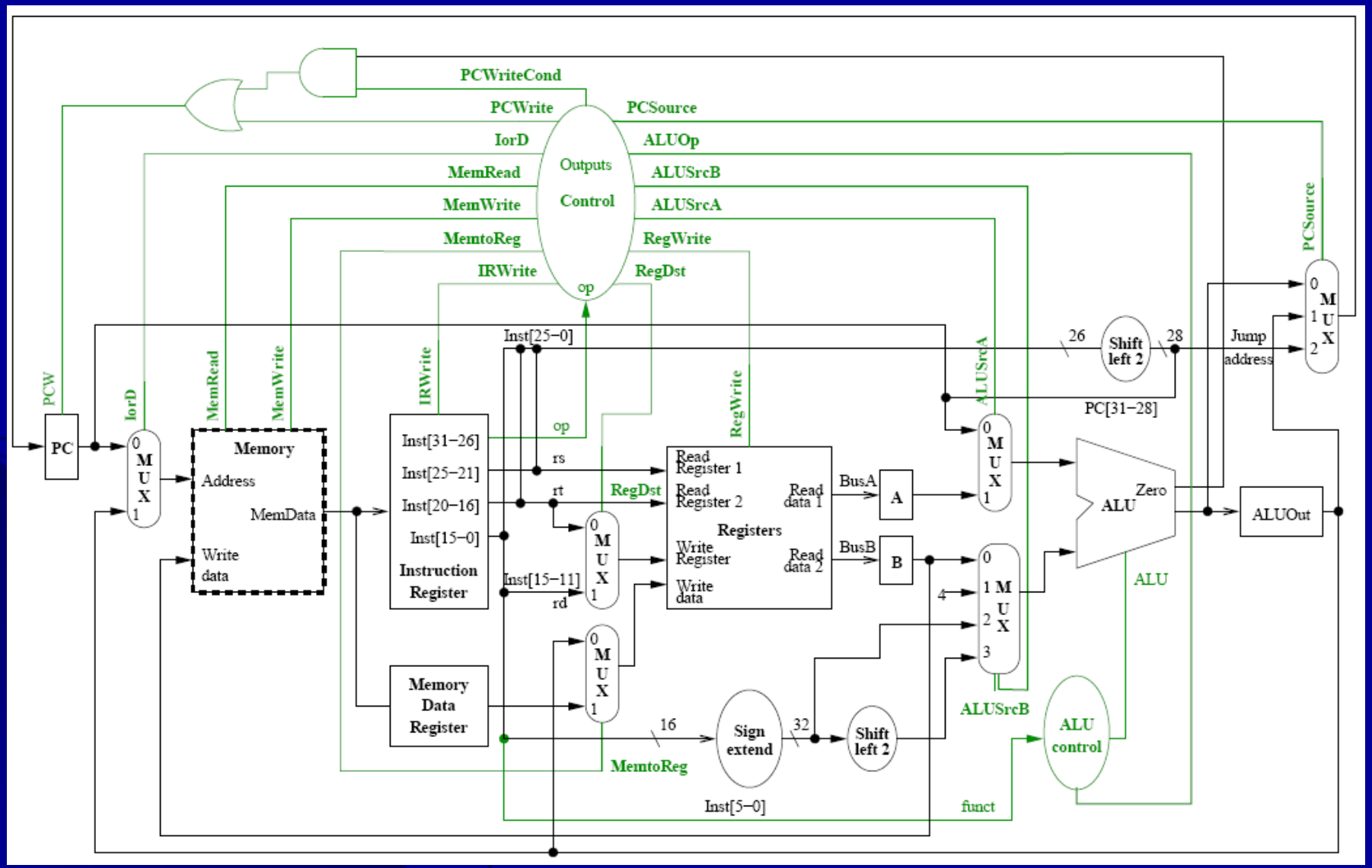
3er Paso: DataPath - MEM



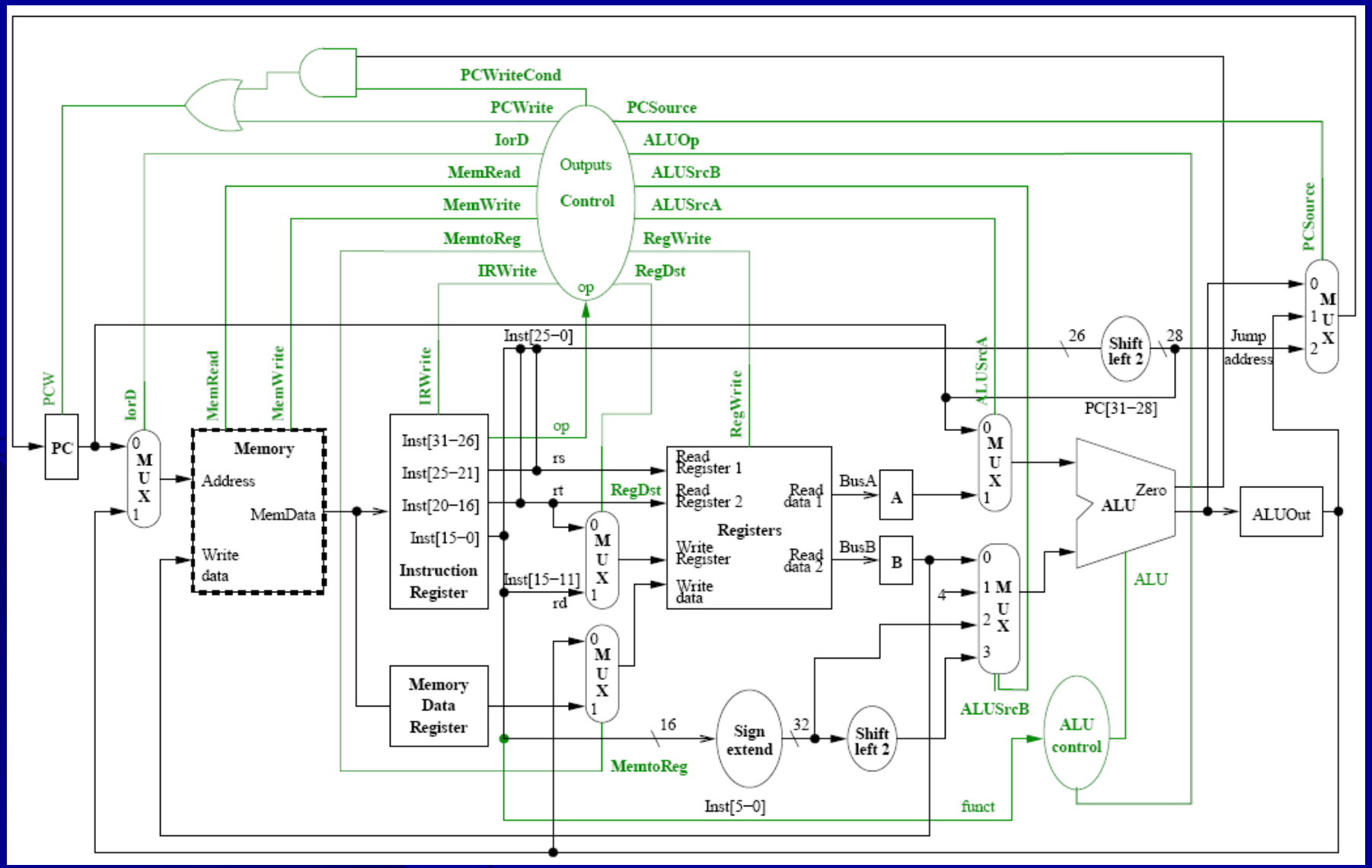
3er Paso: DataPath - WB



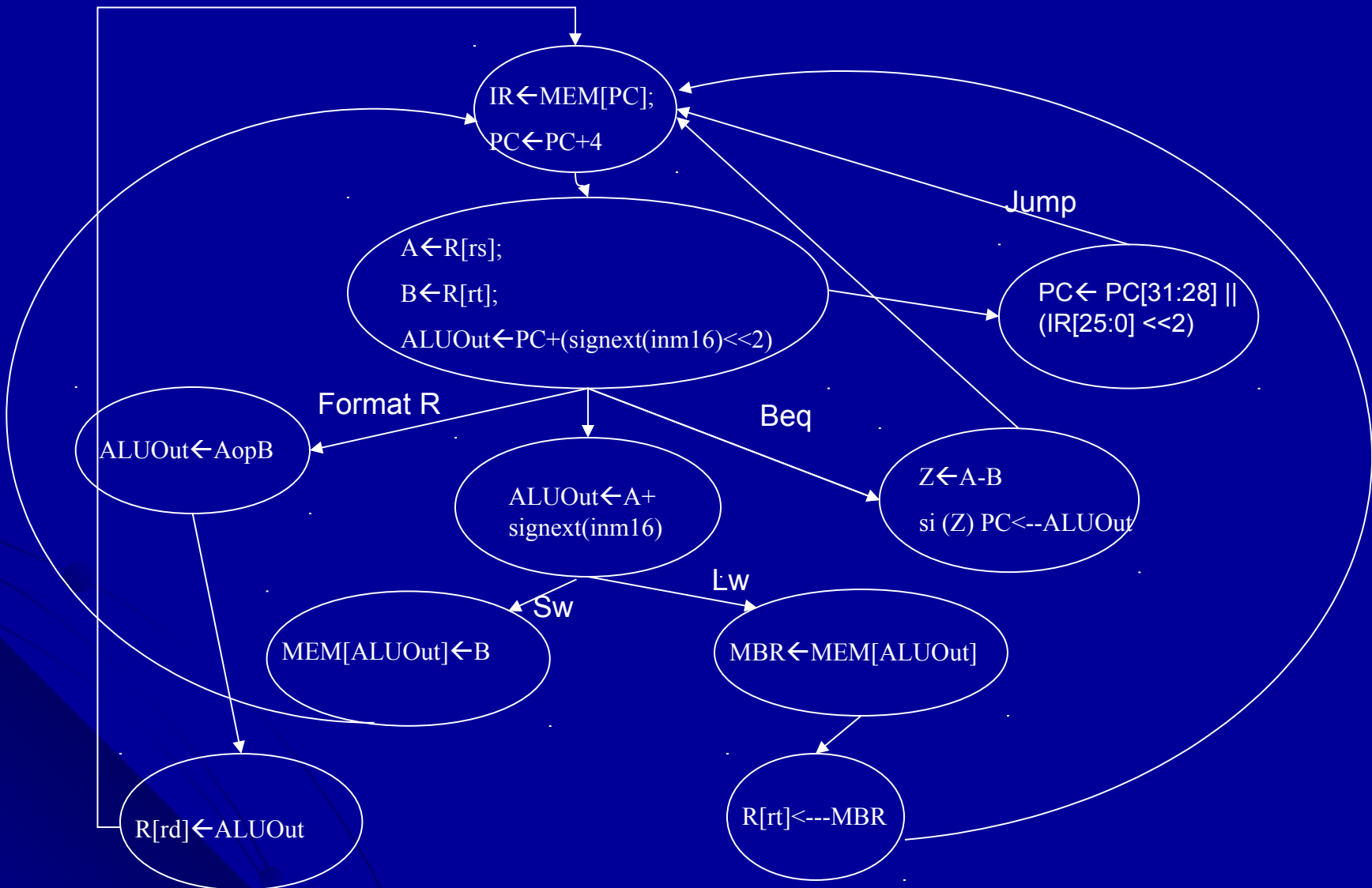
Señales de control



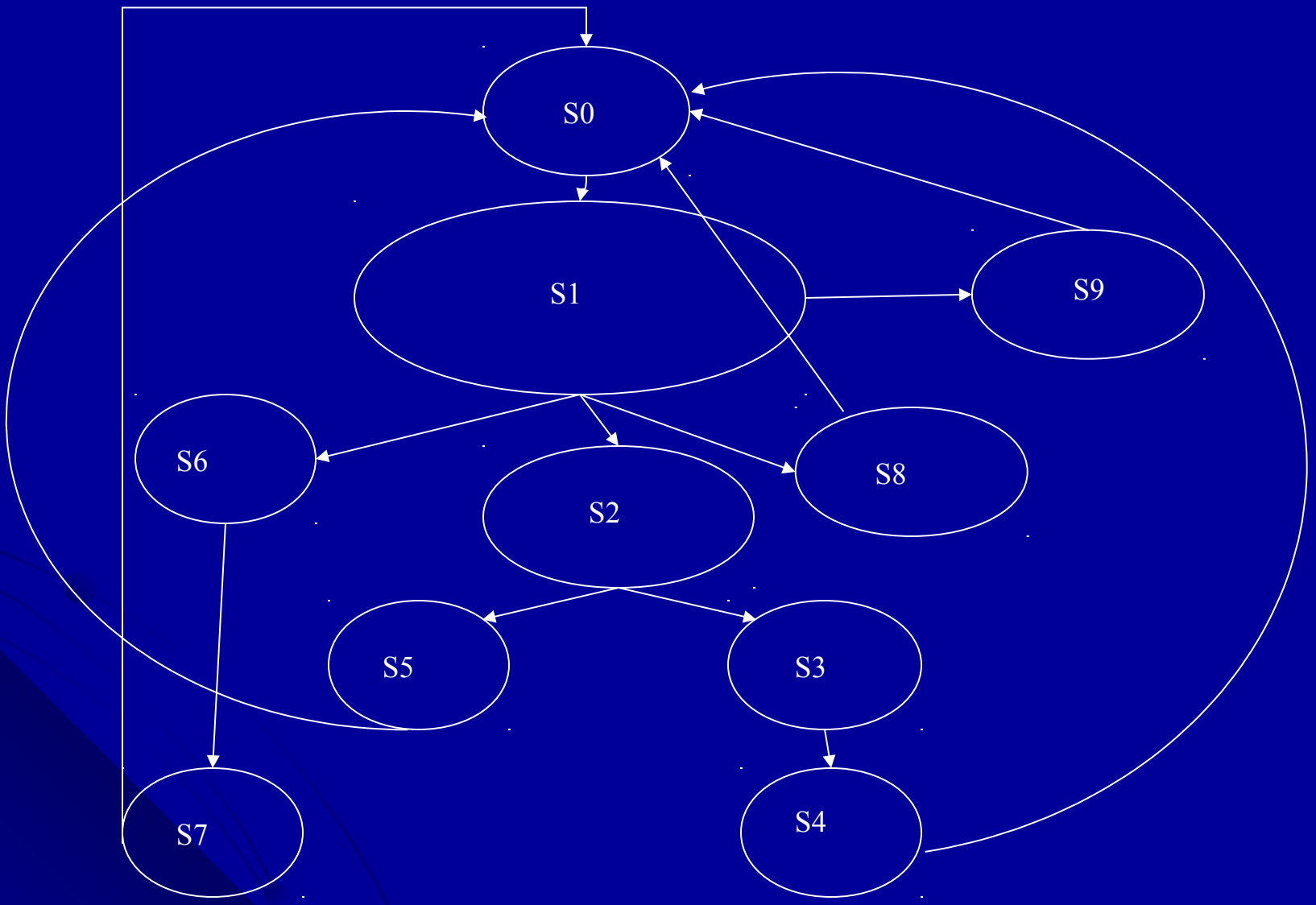
Señales de control



Grafo de Estados



Grafo de Estados



Control de Señales

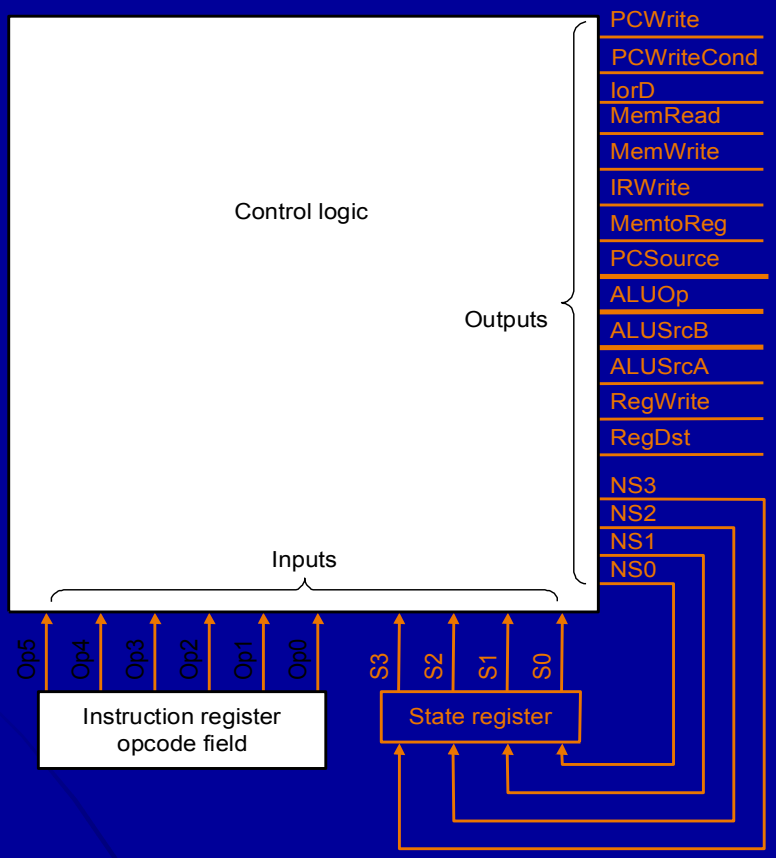
	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9
RegWrite										
IRWrite										
MemRead										
MemWrite										
PCWrite										
PCWCond										
PCSource										
ALUsrcA										
ALUsrcB										
MemToReg										
RegDst										
IoD										
ALUop										

Control de Señales

	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9
RegWrite	0	0	0	0	1	0	0	1	0	0
IRWrite	1	0	0	0	0	0	0	0	0	0
MemRead	1			1						
MemWrite	0	0	0	0	0	1	0	0	0	0
PCWrite	1	0	0	0	0	0	0	0	0	1
PCWCond									1	
PCSource	00								01	10
ALUsrcA	0	0	1	1	1	1	1	1	1	
ALUsrcB	01	11	10	10	10	10	00	00	00	00
MemToReg					1			0		
RegDst					0			1		
IoD	0			1	1	1				
ALUop	00	00	00				10		01	

Máquina de Estados Finitos

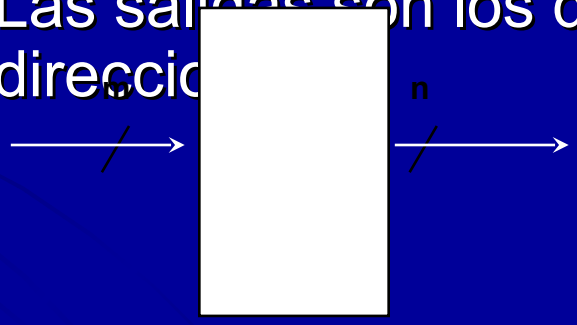
- Implementación:



- Hardwired ➔ Circuito Combinacional (Tabla de verdad!)

Implementación con ROM

- ROM = "Read Only Memory"
  - Se graba la memoria con valores fijos
- Se usa la ROM para implementar la Tabla de Verdad
  - Con direcciones de m-bits, podemos direccionar 2^m entradas en la ROM.
  - Las salidas son los datos (palabras) direcci

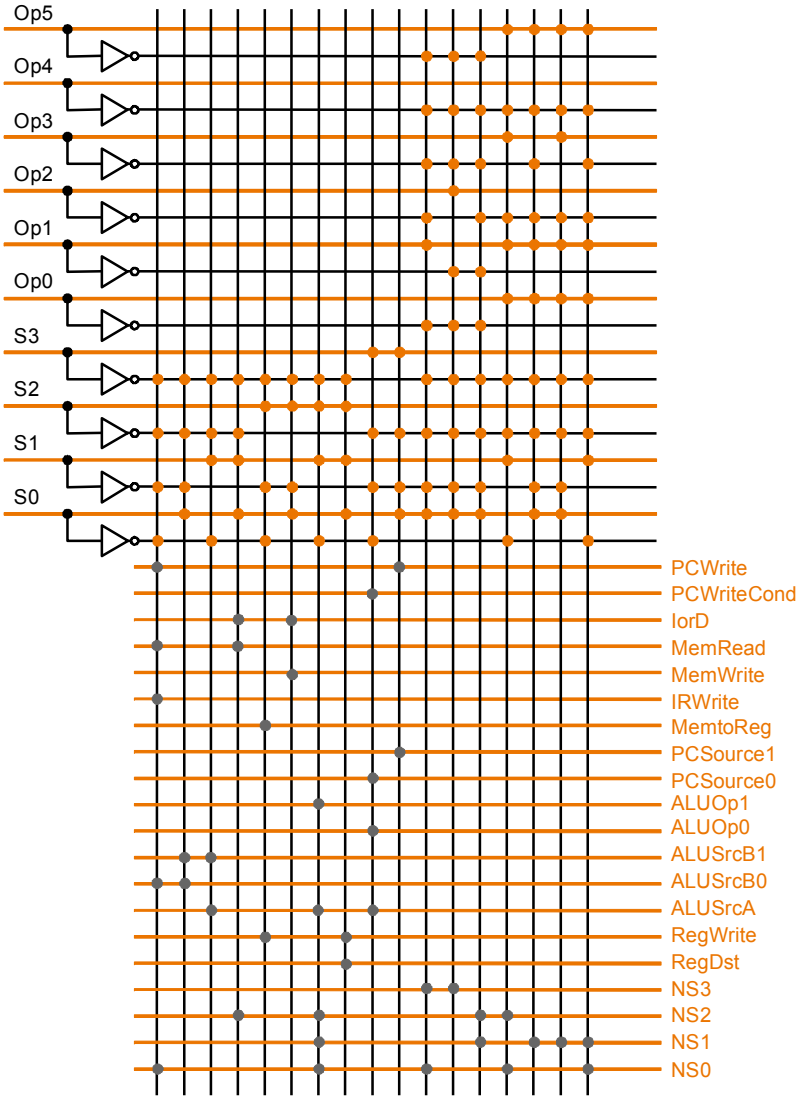


0	0	0	0	1	1	
0	0	1	1	1	0	0
0	1	0	1	1	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	1
1	1	0	0	1	1	0
1	1	1	0	1	1	1

Implementación con ROM

- ¿Cuantas entradas tenemos?
  - 6 bits para el opcode
  - 4 bits para el estado
  - = 10 líneas de direcciones (2^{10} = 1024 posibles direcciones)
- ¿Cuantas salidas?
  - 16 señales de control del camino de datos
  - 4 bits de estado
  - = 20 lineas de salida
- ➔ ROM de 2^{10} x 20bits = 20Kbits
- Problema: mucho desperdicio, ya que para muchisimas entradas, las salidas son idénticas.
  - Por ejemplo, el codígo de operación se ignora muchas veces

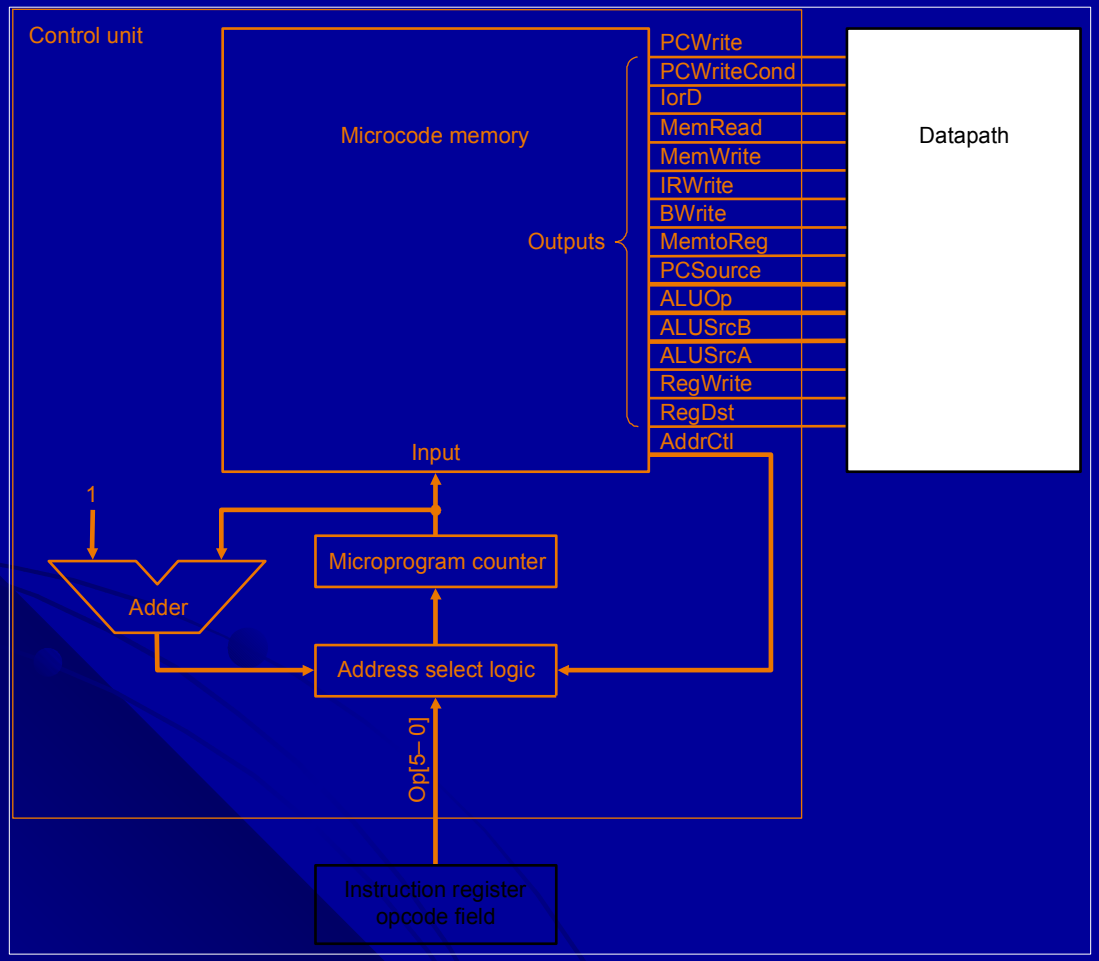
Implementación con PLA



ROM vs PLA

- Se podrían hacer dos ROM:
  — 4 bits de estado se usan como dirección de las palabras de salida: 2^4 x 16 bits= 256bits de ROM
  — 10 bits (6 opcode, 4 estado) se usan como dirección para la función de transición (nuevo estado): 2^{10} x 4 bits de ROM
  — Total: 4K bits de ROM
- PLA es mas pequeña
  — puede compartir términos producto
  — sólo utiliza las entradas que producen valores
  — puede considerar los “no importa”
- Tamaño (#inputs $\times$ #product-terms) + (#outputs $\times$ #product-terms)
  - En el ejemplo = (10x17)+(20x17) = 460 PLA cells
    Una celda de PLA es un poco mas grande que una de ROM

Microprogramación

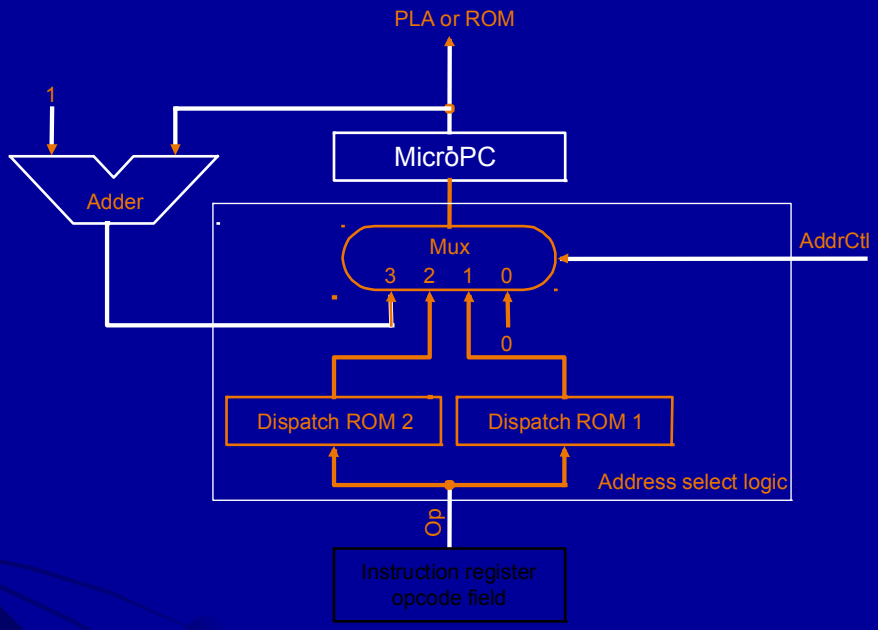


- La ROM es la memoria donde se guardan las instrucciones para el camino de datos (microinstrucciones)
- La dirección de la ROM (microPC) es el estado

Microprogramación

- Es una metodología de especificación
  - Apropiada para arquitecturas con cientos de instrucciones, modos, alto CPI, etc.
  - Las señales se especifican simbólicamente usando microinstrucciones
  - Se define el formato de la microinstrucción, estructurado en campos.
  - Luego cada campo se asocia a un conjunto de señales

Detalle de implementación



State number	Address-control action	Value of AddrCtl
0	Use incremented state	3
1	Use dispatch ROM 1	1
2	Use dispatch ROM 2	2
3	Use incremented state	3
4	Replace state number by 0	0
5	Replace state number by 0	0
6	Use incremented state	3
7	Replace state number by 0	0
8	Replace state number by 0	0
9	Replace state number by 0	0

Dispatch ROM 1			
OP	Name	Value	state
000000	R-type	Rformat1	0110
000010	`j`	JUMP1	1001
000100	`beq`	BEQ1	1000
100011	`lw`	Mem1	0010
101011	`sw`	Mem1	0010

Dispatch ROM 2			
OP	Name	Value	state
100011	`lw`	LW2	0011
101011	`sw`	SW2	0101

Diseño del Microcódigo

- Función básica: Proveer señales para el datapath
- Dos enfoques:
  - Horizontal:
    - La microinstruccion provee todas las señales de control necesarias para un ciclo
    - Paralelismo
  - Vertical
    - Más compacta
    - Las señales están codificadas para que ocupen menos bits
    - Menos paralelismo

Diseño de microinstrucciones

- Diferentes señales agrupadas por campos

Campo	Función
Alu Control	Que operación debe hacer la ALU en este ciclo
SRC1	Especifica el 1º operando de la ALU
SRC2	Especifica el 2º operando de la ALU
Register Ctrl	Especifica Lectura/Grabación de Registros, y la fuente para la grabación
Memoria	Especifica Lectura/Grabación. En lectura el registro de destino
PCWriteCtrl	Especifica la grabación del PC
Secuencia	Determina como elegir la proxima microinstrucción

Formato de Microinstrucción

Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
SRC2	B	ALUSrcB = 00	Register B is the second ALU input.
	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	Read PC	MemRead, IorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, IorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, IorD = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource = 00 PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

Microprogramación

- Microprogramando!

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

Microcódigo: Ventajas-Desventajas

- Ventajas en la especificación:
  - Fácil de diseñar: se escribe el microprograma
- Implementación en ROM (off-chip)
  - Fácil de cambiar
  - Puede emular otras arquitecturas
  - Puede usar registros internos
- Desventajas de la implementación
  - Control se implementa (hoy) en el mismo chip que el camino de datos
  - La ROM no es mas rápida que la RAM (CISC vs RISC)