Microprogramación

Consideraciones en el diseño de una CPU

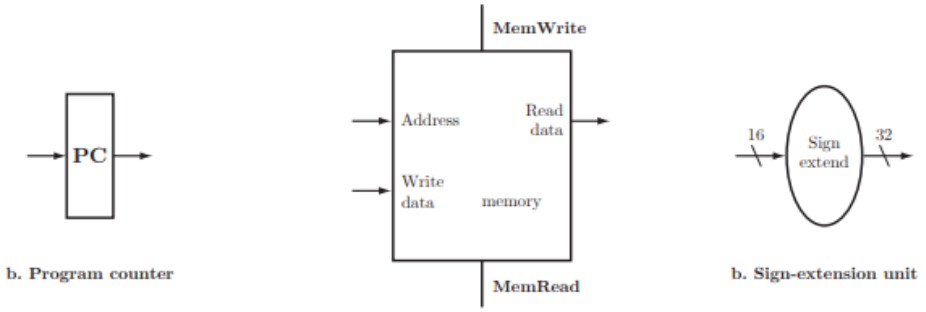
Ï Una única ALU, una única Memoria, un único Banco de Registros.

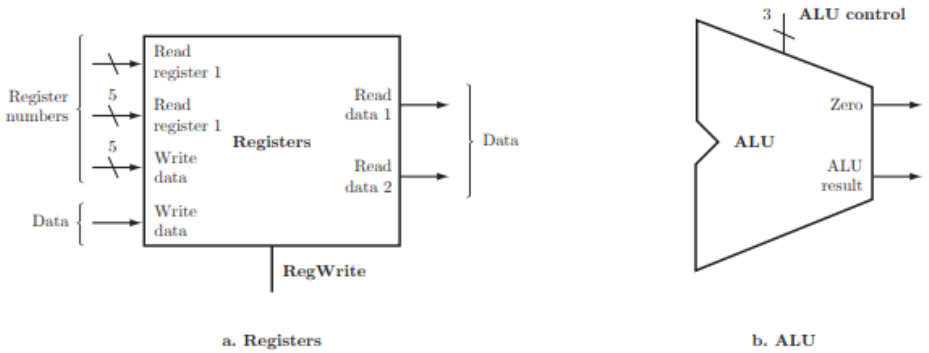
Ï ¿Problemas con el uso de recursos?

Ï Una instrucción utiliza un mismo recurso varias veces y se pierden los valores anteriores.

Ï Una instrucción utiliza un mismo recurso en la misma etapa para dos o más tareas diferentes.

Recursos disponibles





Diseño: pasos necesarios

1. Analizar el conjunto de instrucciones para determinar los requerimientos del camino de datos.
2. Seleccionar los componentes.
3. Construir el camino de datos según los requerimientos.
4. Analizar la implementación de cada instrucción para determinar las señales de control necesarias.
5. Construir el control.

Etapas de ejecución de una instrucción

1. Fetch de la instrucción: **IF** (Fetch)
2. Decodificación (y lectura de registros: **ID** (Decode)
3. Ejecución ó cálculo de dirección de memoria: **EX** (Execution)
4. Acceso a datos en memoria: **MEM**
5. Escritura en registros: **WB** (Write Back)

Formato de instrucción MIPS

- **Tipo R**
  - Aritméticas

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

- **Tipo J**
  - Saltos

- 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)

Register Transfer Language

Ï Cada instrucción está formada por un conjunto de microoperaciones.
Ï El *Register Transfer Language* (RTL) se utiliza para describir la secuencia exacta de las microoperaciones.
Ï Ejemplo (`Fetch` en Marie):

- Ï t1: MAR <- [PC]
- Ï t2: MBR <- mem[MAR], PC <- PC + 1
- Ï t3: IR <- [MBR]

Ejemplo Paso 1: Tipo R (add, sub...)

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

RTL

```
Ï t1: IR <- mem[PC], PC <- PC + 4
Ï t2: A <- R[RS], B <- R[RT]
Ï t3: ALUout <- A op B
Ï t4: R[RD] <- ALUout
```

Ejemplo Paso 1: Branch

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

RTL
Ï t1: IR <- mem[PC], PC <- PC + 4
Ï t2: A <- R[RS], B <- R[RT]
Ï t3: ALUout <- PC + signextend(imm16) < < 2
Ï t4: Comparar A y B, PC <- ALUout (si Z=1)

Ejemplo Paso 1: Load

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

RTL
Ï t1: IR <- mem[PC], PC <- PC + 4
Ï t2: A <- R[RS], B <- R[RT] (B no se usa)
Ï t3: ALUout <- PC + signextend(imm16) < < 2
Ï t4: MBR <- mem[ALUout]
Ï t5: R[RT] <- MBR

Ejemplo Paso 1: Store

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

RTL
Ï t1: IR <- mem[PC], PC <- PC + 4
Ï t2: A <- R[RS], B <- R[RT] (dato a escribir)
Ï t3: ALUout <- PC + signextend(imm16) < < 2
Ï t4: mem[ALUout] <- B

Ejemplo Paso 1: Jump

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

RTL
Ï t1: IR <- mem[PC], PC <- PC + 4
Ï t2: NADA !
Ï t3: PC<31:2> <- PC<31:28>, IR<25:0> < < 2

Resumen para cada etapa del ciclo de instrucción

Etapa	Tipo de instrucción	acciones
IF	todas	IR <- mem[PC] PC <- PC + 4
ID	todas	A <- R[Rs] B <- R[RT] ALUout <- PC + (inm16 < <2)
EX	tipo R Load/Store Branch Jump	ALUout <- A op B ALUout <- A + signextend(inm16) if(A==B) then PC <- ALUout PC<31:2> <- PC<31:28>, IR<25:0> < < 2
MEM	Load Store	MBR <- mem[ALUout] mem[ALUout] <- B
WB	tipo R Load	R[RD] <- ALUout R[RT] <- MBR

Etapas según el tipo de instrucción

Ï Tipo R: 4 etapas (IF, ID, EX y WB)
Ï Branch y Jump: 3 etapas (IF, ID y EX)
Ï Store: 4 etapas (IF, ID, EX y MEM)
Ï Load: 5 etapas (IF, ID, EX, MEM y WB)

Etapa 1: Fetch (IF)

RTL
Ï IR <- mem[PC]
Ï PC <- PC + 4

Etapa 2: Decode (ID)

RTL
Ï Opción lectura de registros:
    Ï A <- R[IR[25:21]]
    Ï B <- R[IR[20:16]]
Ï Opción cálculo de saltos:
    Ï ALUout <- PC + signextend(IR[15:0]) < < 2

Etapa 3: Execute (EX)

RTL
Ï Opción aritmética/lógica:
    Ï `ALUout <- A op B`
    Ï `ALUout <- A op signextend(IR[15:0])`
Ï Opción dirección (Load ó Store):
    Ï `ALUout <- A + signextend(IR[15:0])`
Ï Salto condicional:
    Ï `Si A = B, PC <- ALUout`
Ï Jump:
    Ï `PC[31:28] <- PC || IR[25:0] < < 2`

Etapa 3: Memoria (MEM)

RTL sólo para Load y Store
Ï Opción lectura:
    Ï `MBR <- mem[ALUout]`
Ï Opción escritura:
    Ï `mem[ALUout] <- B`

Etapa 3: Escritura (WB)

RTL

Ï Opción tipo R ó aritmética inmediata:

- Ï `R[IR[15:11]] <- ALUout`

Ï Opción escritura:

- Ï `R[IR[20:16]] <- MBR`

Paso 1: requerimientos de la ISA

Ï Memoria

- Ï Para instrucciones y datos

Ï Registros 32x32

- Ï Leer RS, leer RT
- Ï Escribir RT ó RD

Ï PC, MDR

Ï A, B para datos intermedios, ALUout (retiene salida de la ALU)

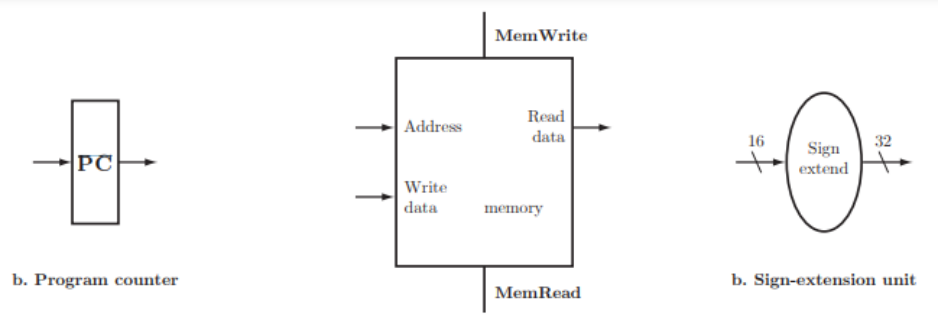
Ï Extensor de signo de 16 a 32 bits

Ï Sumar y restar registros y/o valores inmediatos

Ï Operaciones lógicas (and/or) con registros y/o valores inmediatos

Ï Sumar 4 al PC ó 4+inmediato extendido * 4

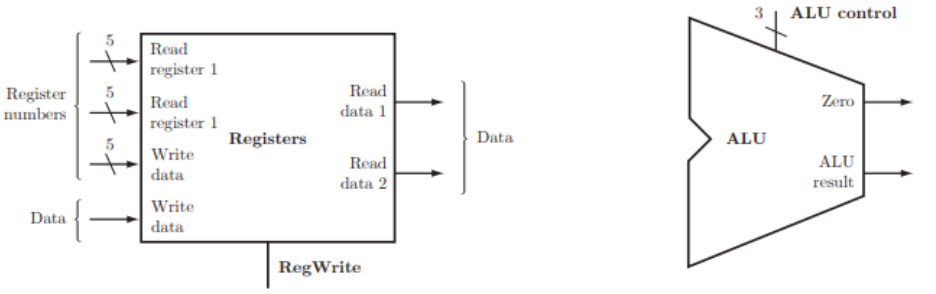
Paso 2: componentes del camino de datos



b. Program counter

a. memory unit

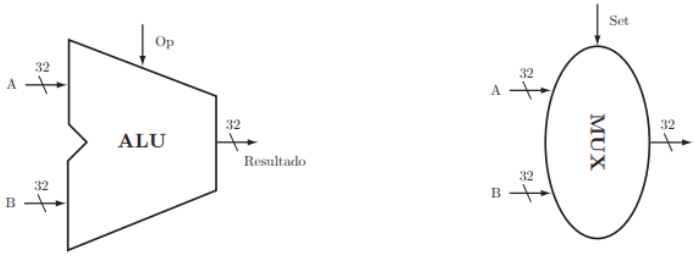
b. Sign-extension unit



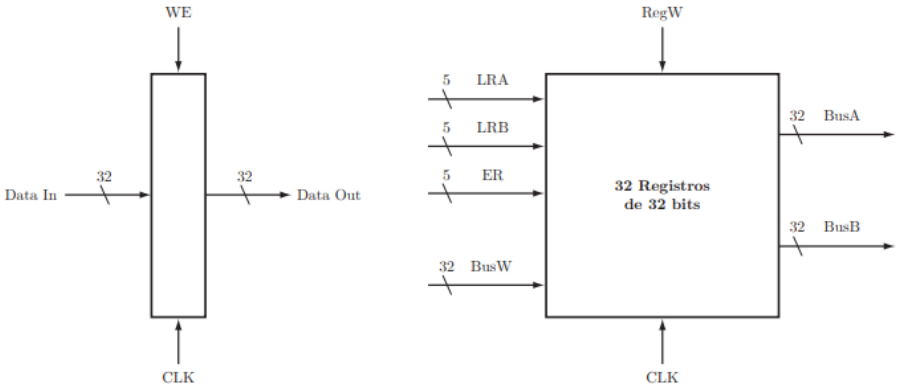
a. Registers

b. ALU

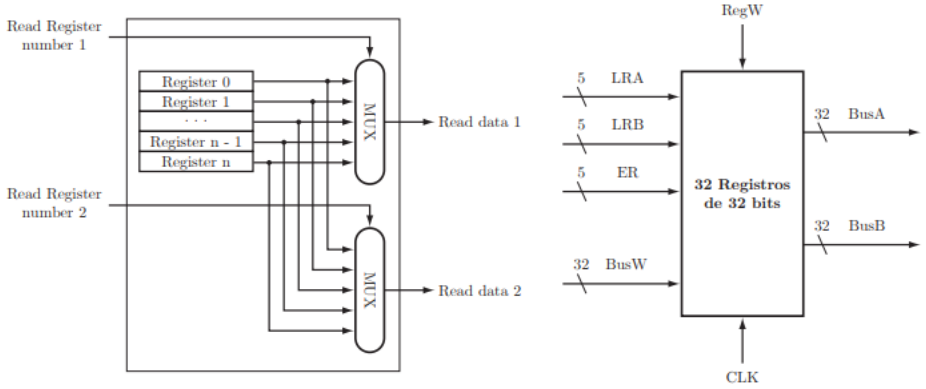
Elementos de lógica combinacional: ALU y multiplexor



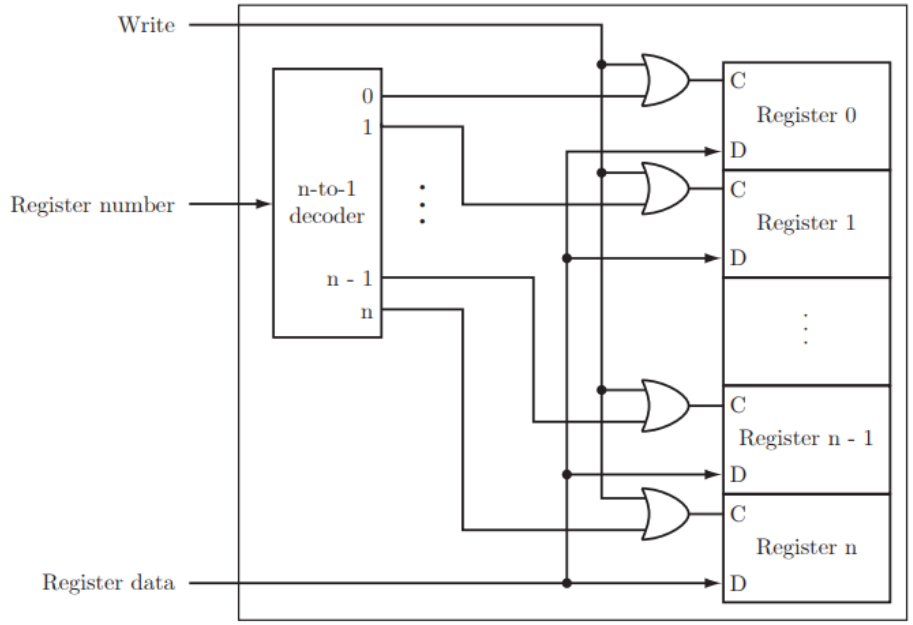
Elementos de almacenamiento: banco de registros

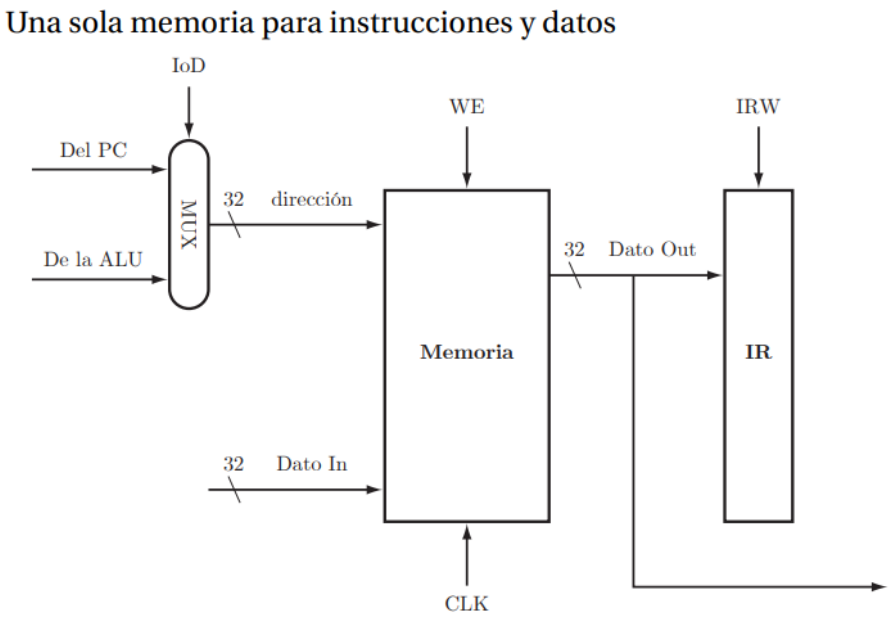


Elementos de almacenamiento: banco de registros con dos puertos de lectura

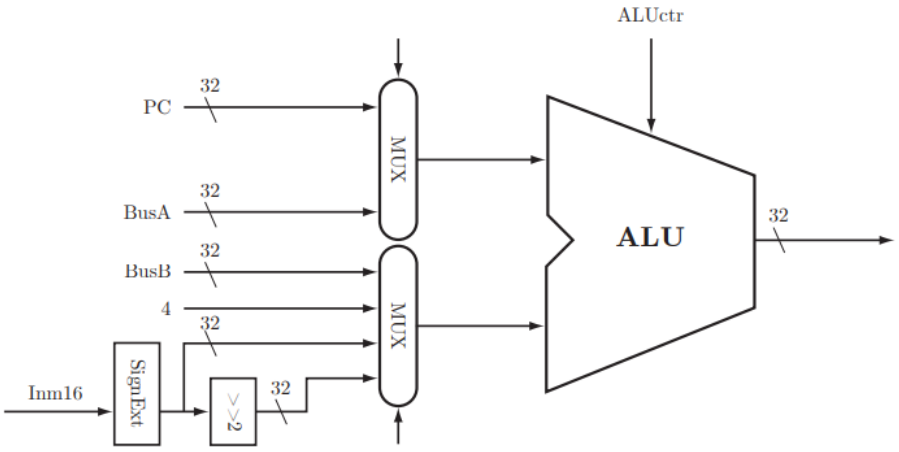


Elementos de almacenamiento: banco de registros con un puerto de escritura

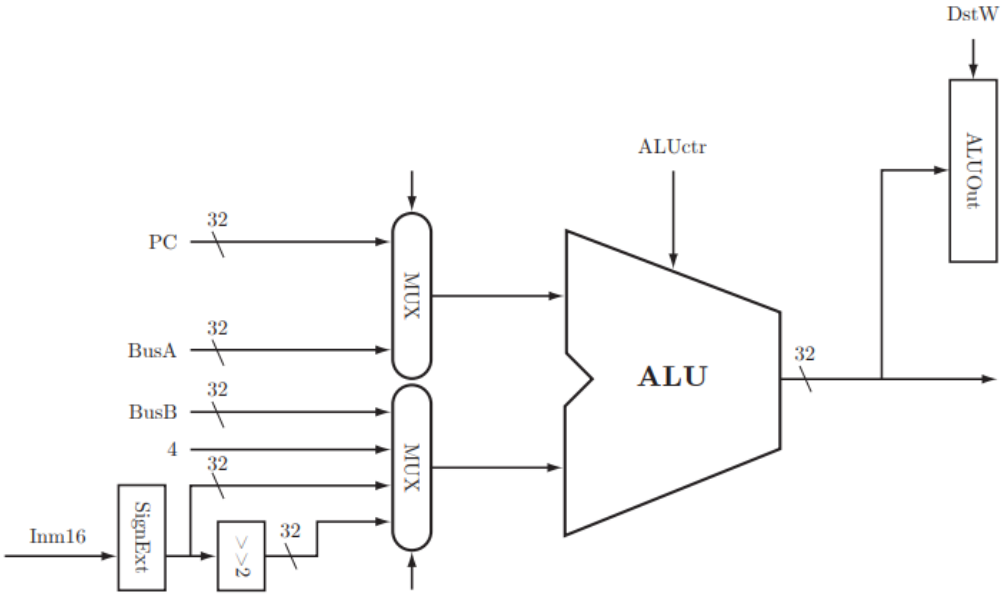




Paso 3: reutilización de una sola ALU



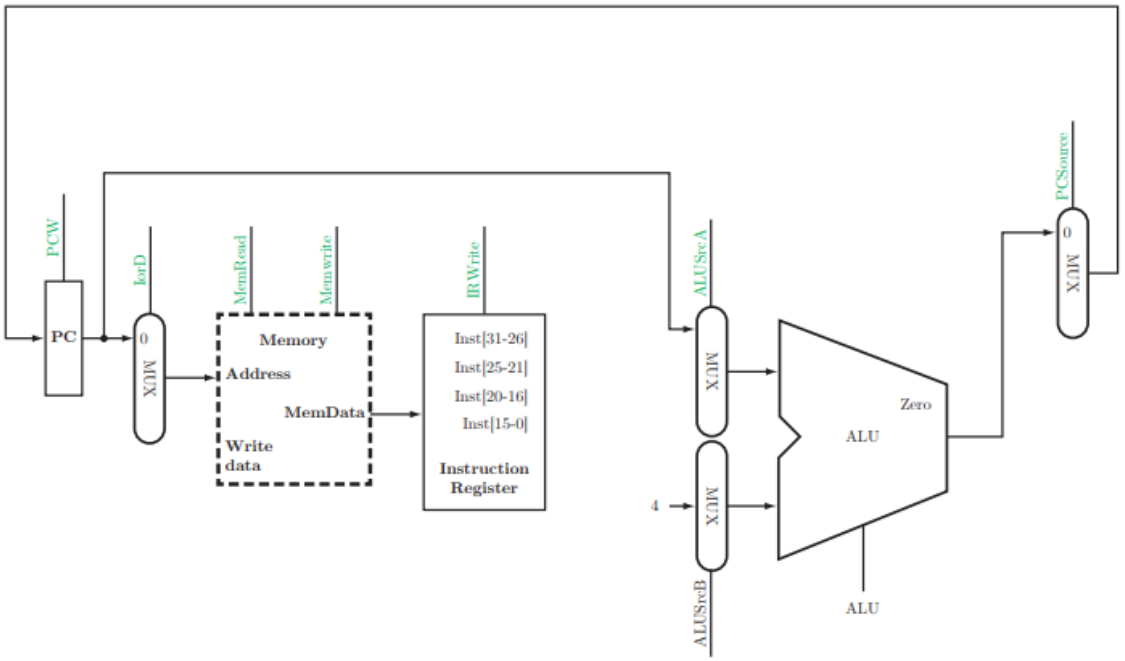
Paso 3: Registro ALUout



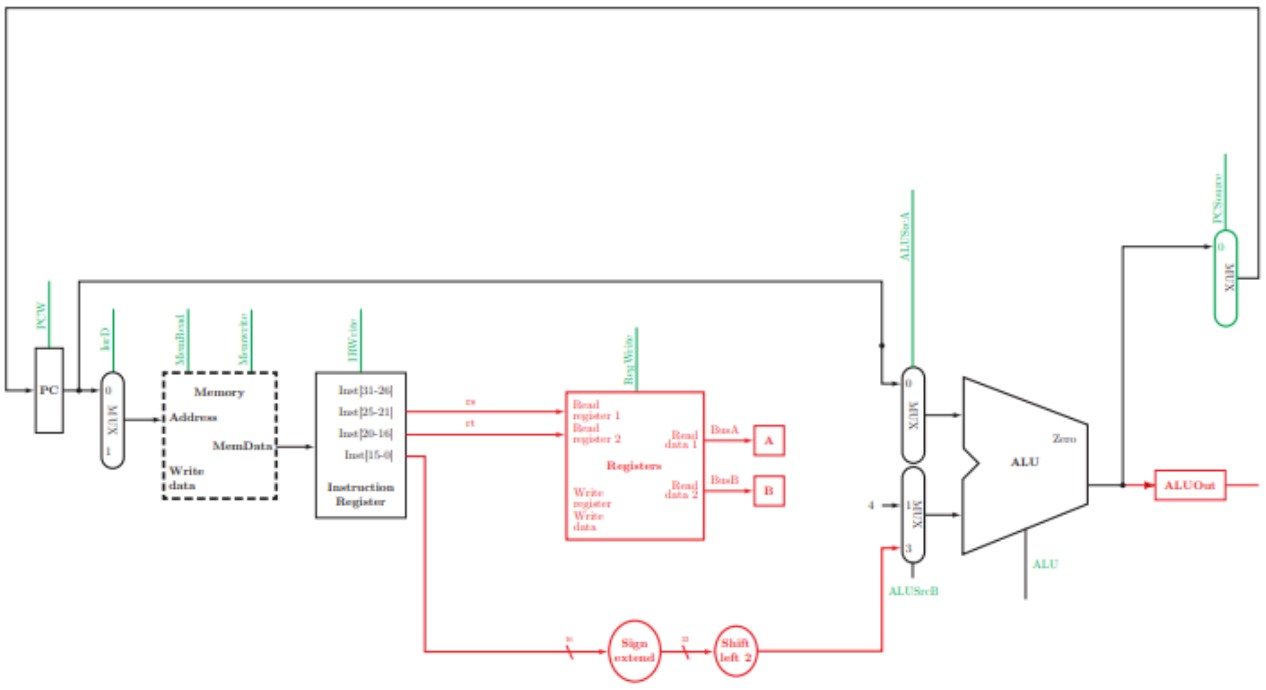
Paso 3: Fetch (IF)

IR <- Mem[PC]

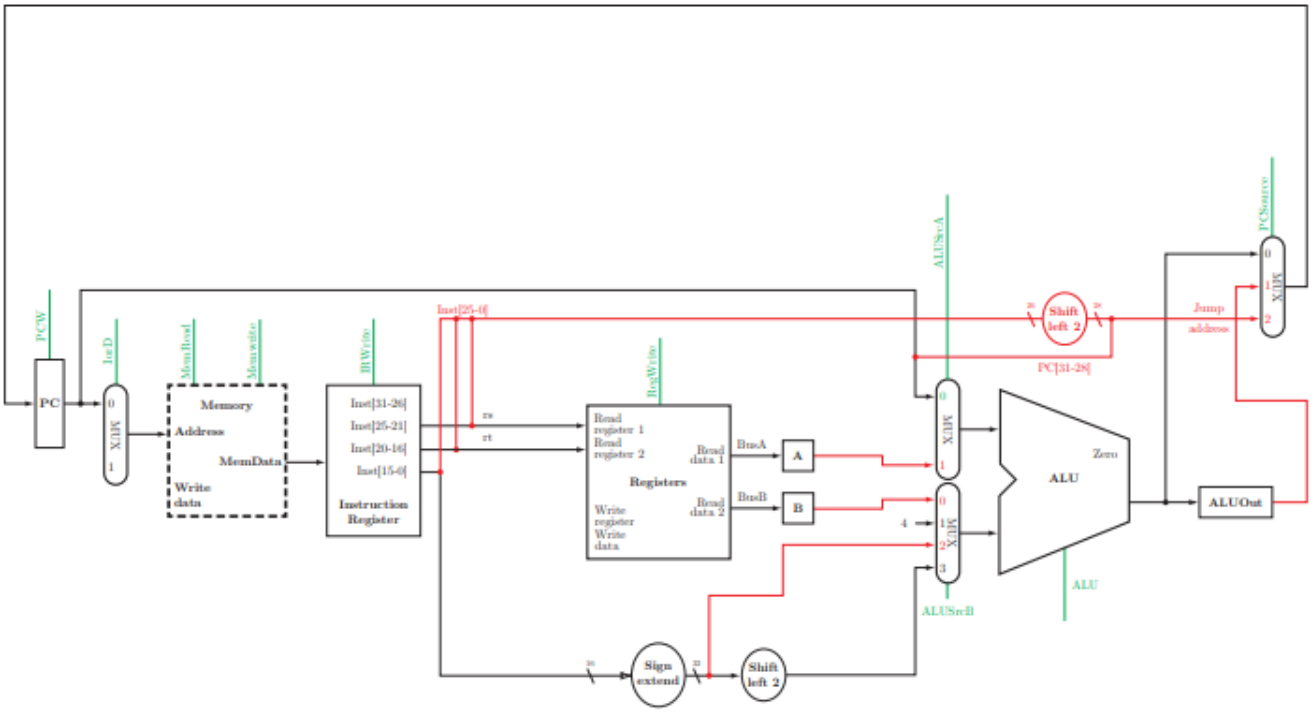
PC <- PC + 4 (código secuencial)



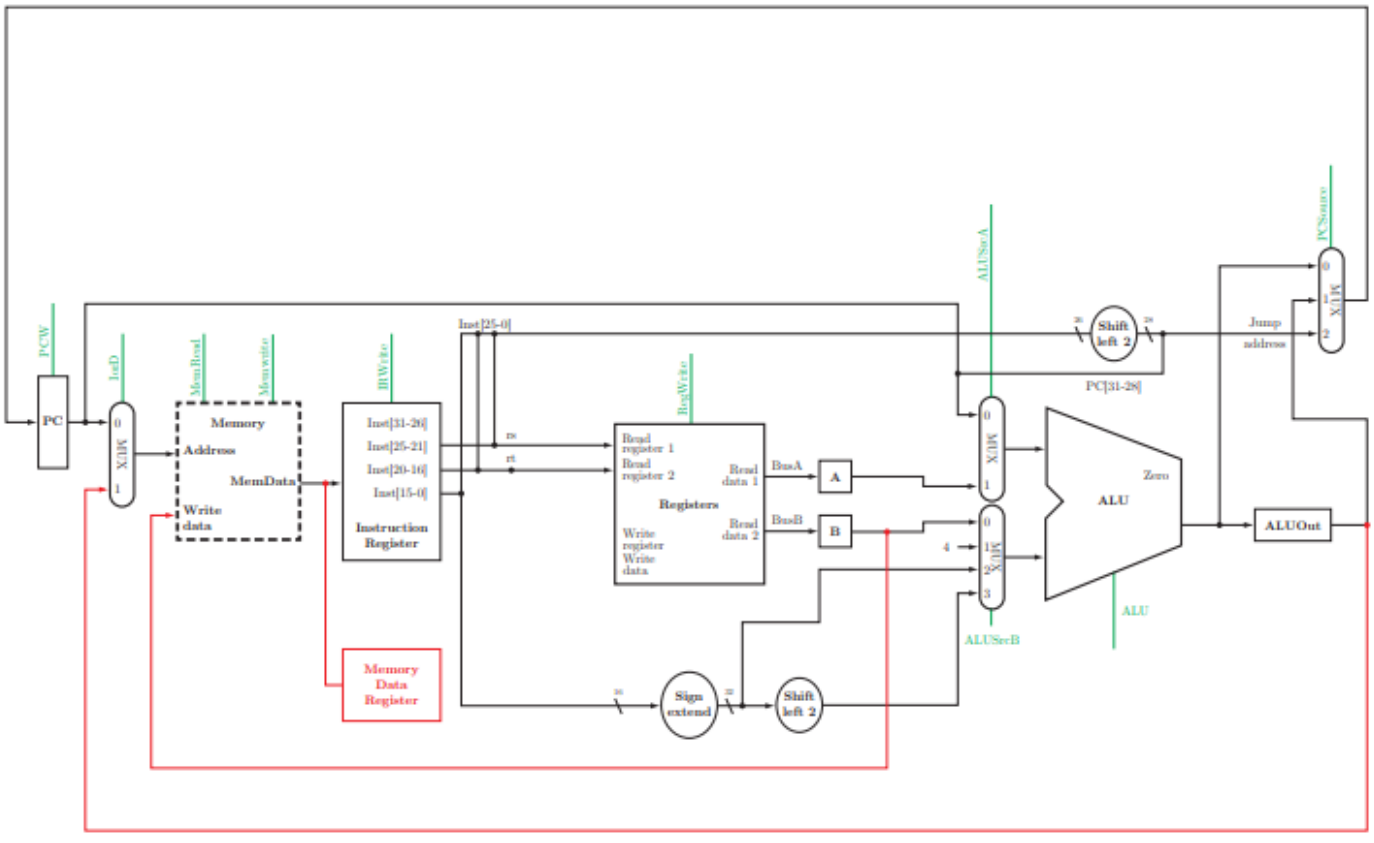
Paso 3: Decode (ID)



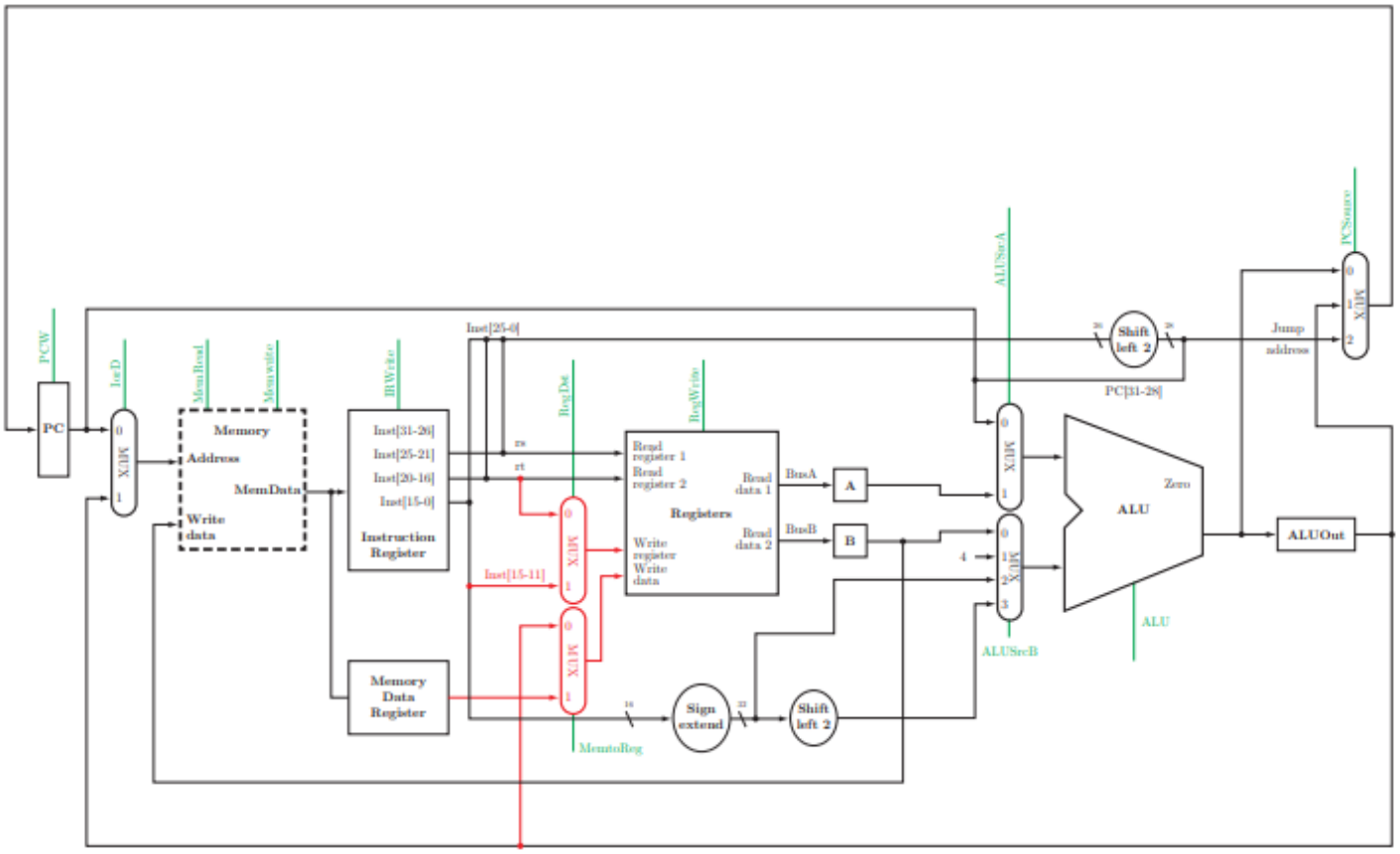
Paso 3: Data path (EX)



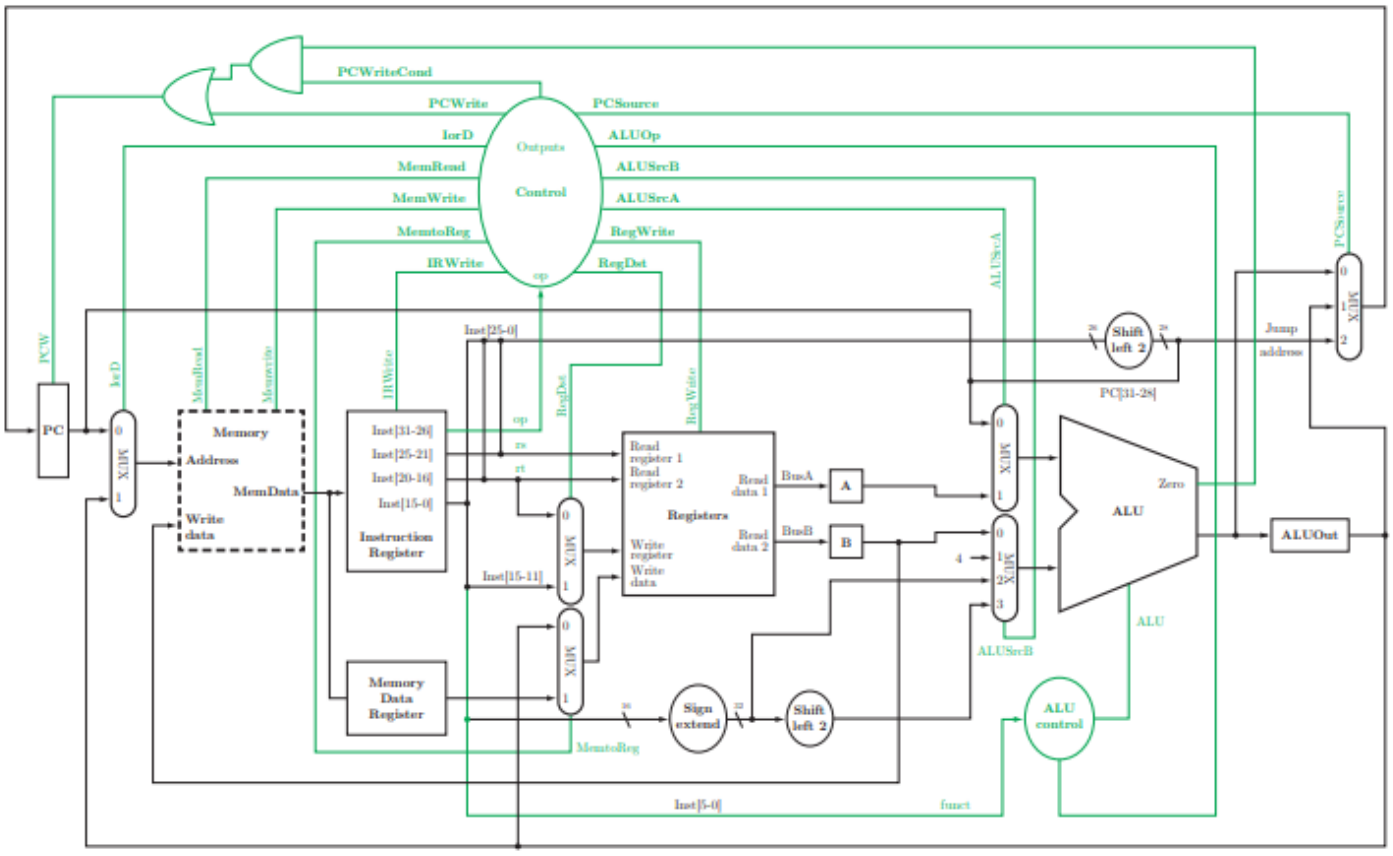
Paso 3: Data path (MEM)



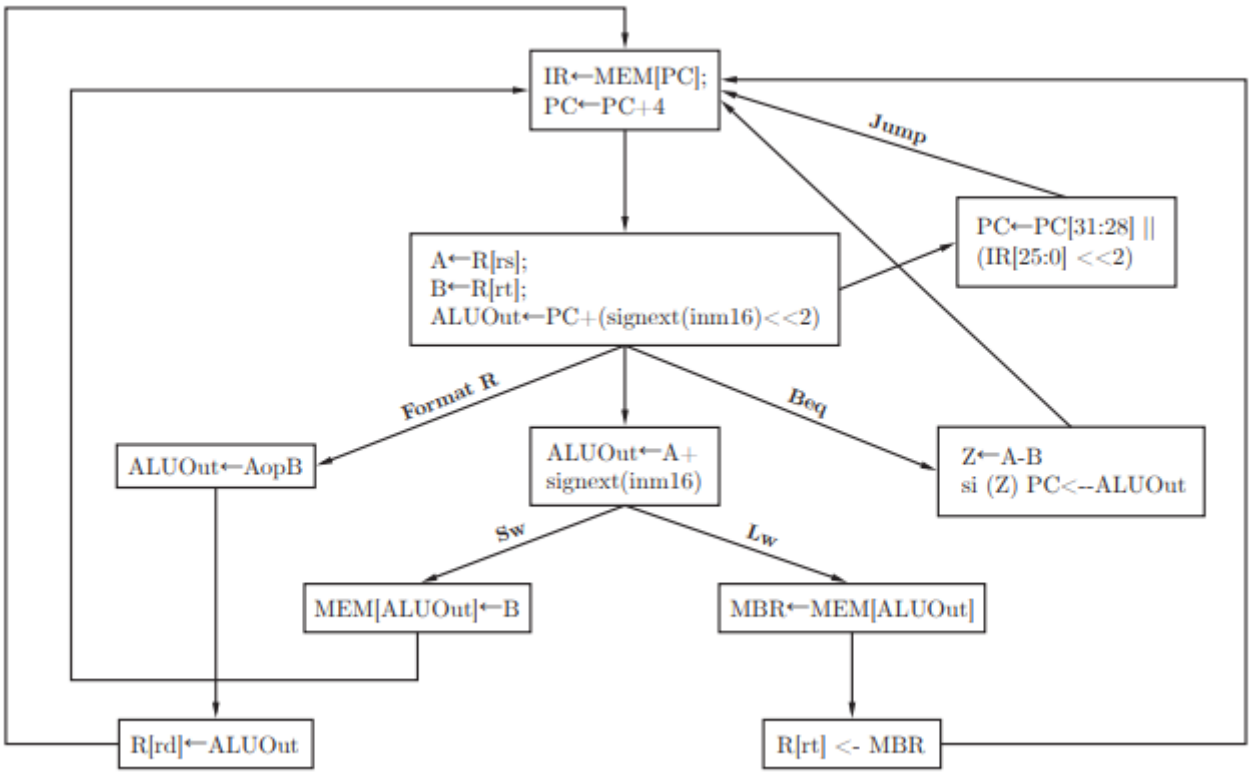
Paso 3: Data path (WB)



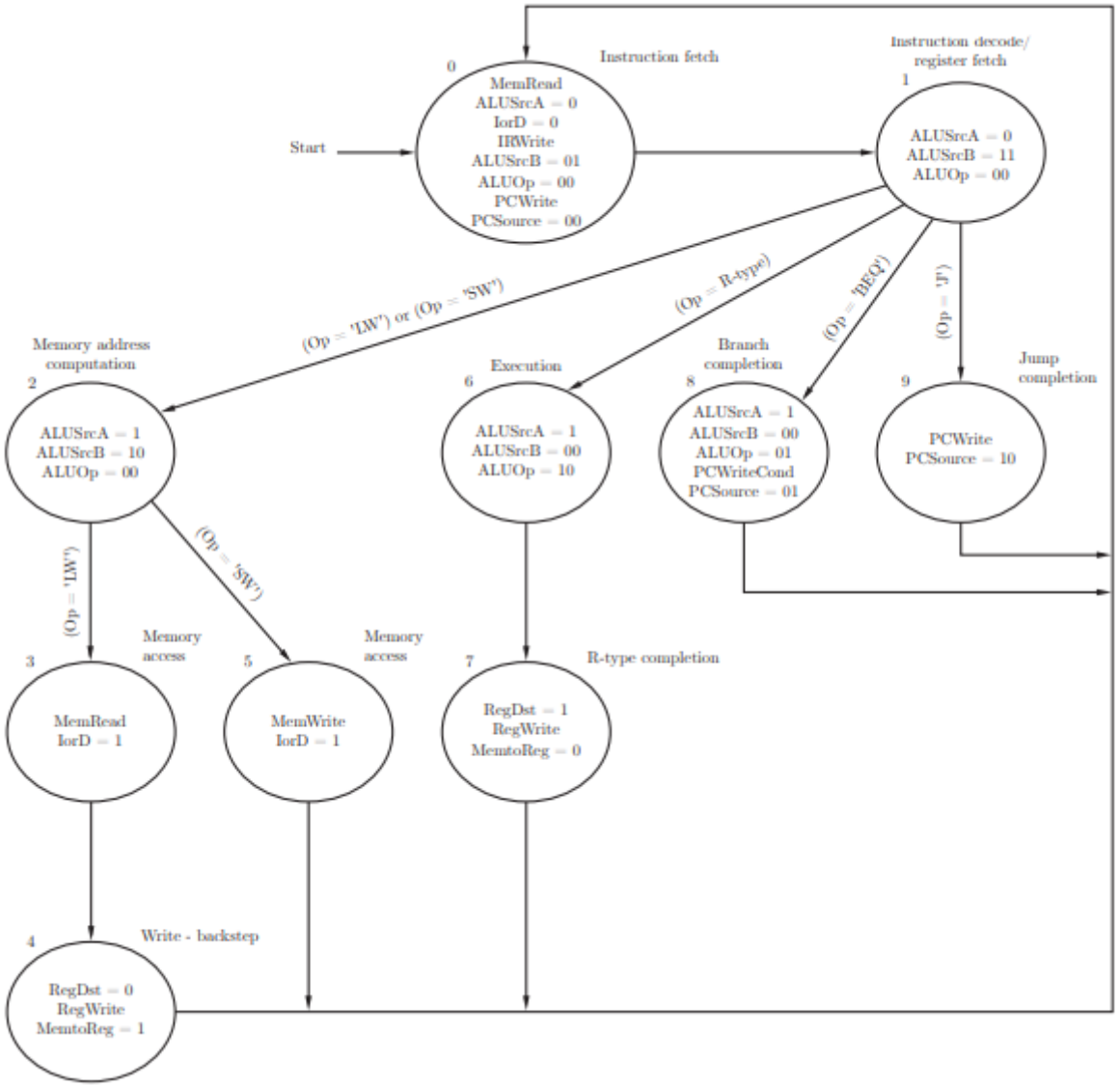
Señales de control



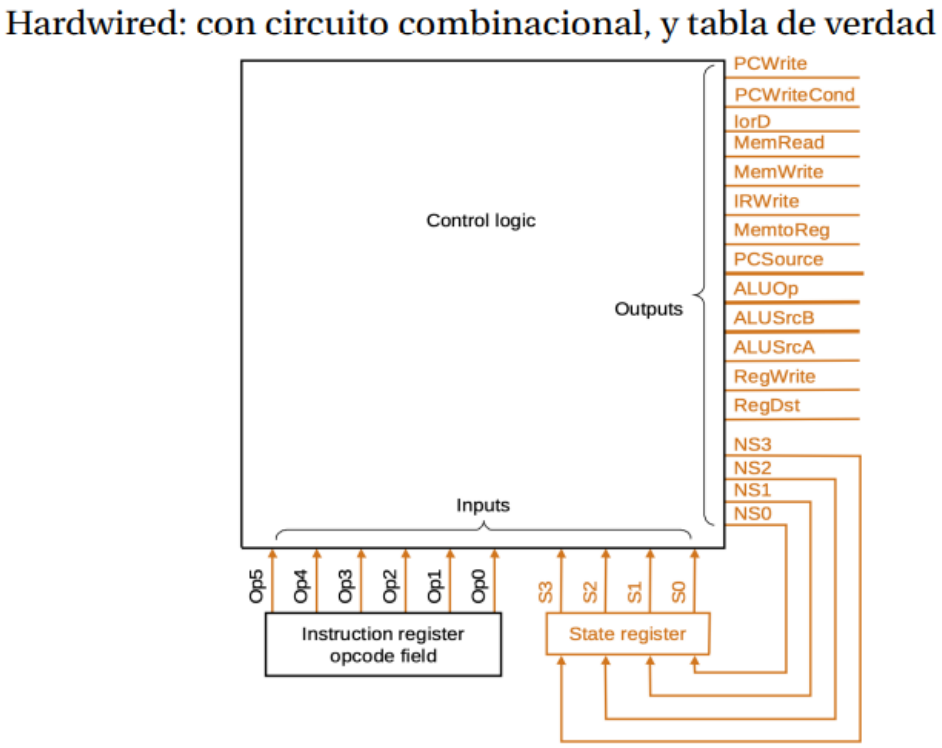
Grafo de estados



Control de señales



Máquina de estados finitos



Microprogramada con una memoria ROM