

---

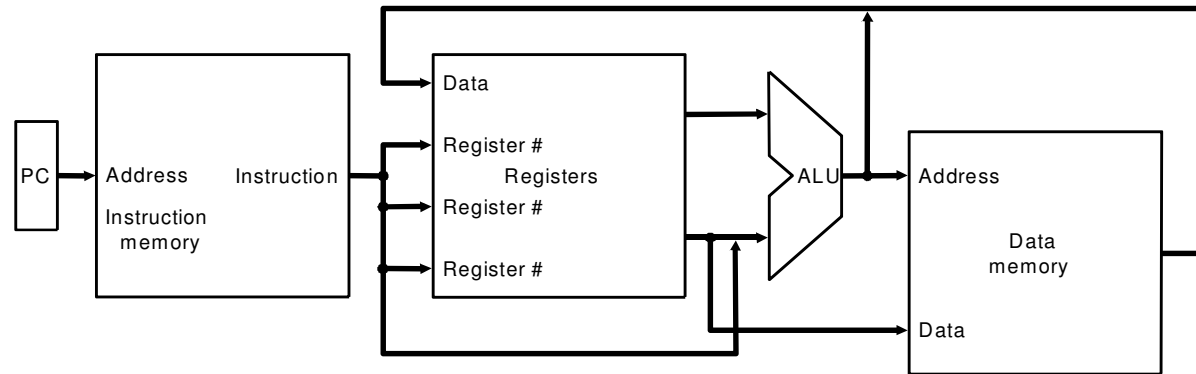
# L'unità di elaborazione e di controllo

# The Processor: Datapath & Control

---

- We're ready to look at an implementation of the MIPS
- Simplified to contain only:
  - memory-reference instructions: `lw`, `sw`
  - arithmetic-logical instructions: `add`, `sub`, `and`, `or`, `slt`
  - control flow instructions: `beq`, `j`
- Generic Implementation:
  - use the program counter (PC) to supply instruction address
  - get the instruction from memory
  - read registers
  - use the instruction to decide exactly what to do
- All instructions use the ALU after reading the registers
  - Why? memory-reference? arithmetic? control flow?

- **Abstract / Simplified View:**



## Two types of functional units:

- elements that operate on data values (combinational)
- elements that contain state (sequential)

# Convenzioni logiche e temporizzazioni

---

- *Segnale affermato* = livello logico vero (1)

- *Elementi combinatori:*

le uscite dipendono solo dagli ingressi nello stesso istante

- *Elementi sequenziali:*

le uscite dipendono sia dagli ingressi che dallo stato memorizzato ad un dato momento (memorie, registri, flip flop).

Sarebbe possibile far ripartire un elemento dopo un'interruzione caricando gli stessi elementi contenuti prima di essa.

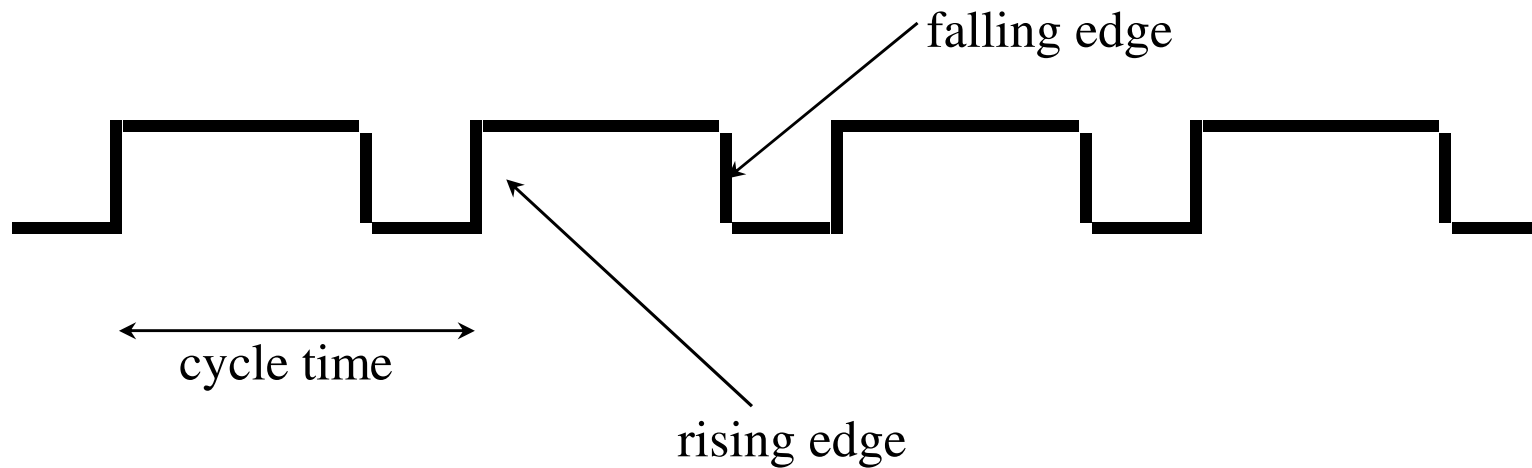
Almeno due ingressi: valore da scrivere e segnale di sincronismo (clock) per determinare la scrittura.

La lettura può avvenire in qualunque istante.

# State Elements

---

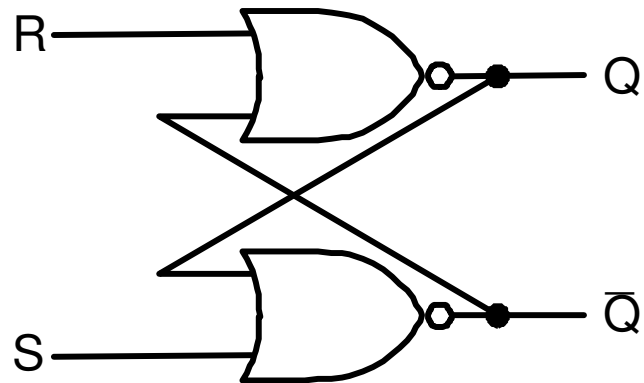
- Unlocked vs. Clocked
- Clocks used in synchronous logic
  - when should an element that contains state be updated?



# An unclocked state element

---

- The set-reset latch
  - output depends on present inputs and also on past inputs



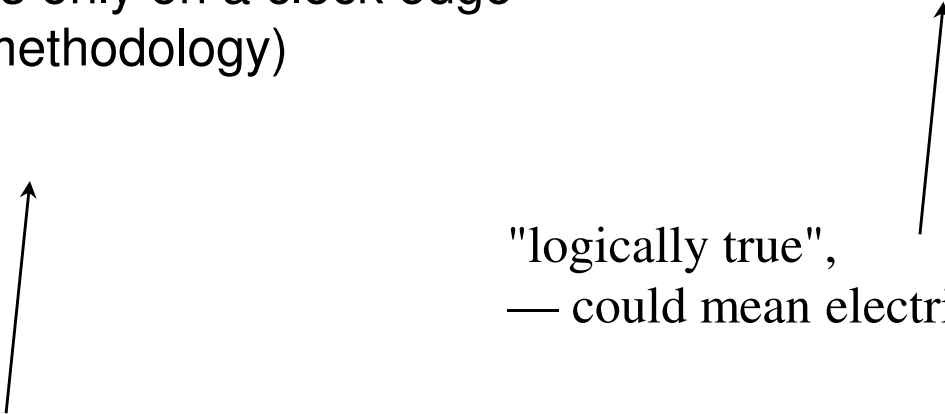
SET	RESET	Q	not Q
0	0	Q	not Q
0	1	0	1
1	0	1	0
1	1	non permessa	

# Latches and Flip-flops

---

- Output is equal to the stored value inside the element  
(don't need to ask for permission to look at the value)
- Change of state (value) is based on the clock
- Latches: whenever the inputs change, and the clock is asserted
- Flip-flop: state changes only on a clock edge  
(edge-triggered methodology)

"logically true",  
— could mean electrically low

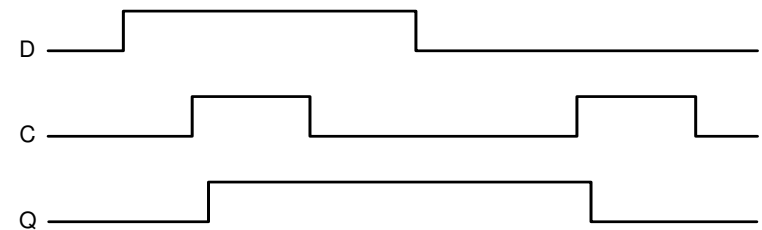
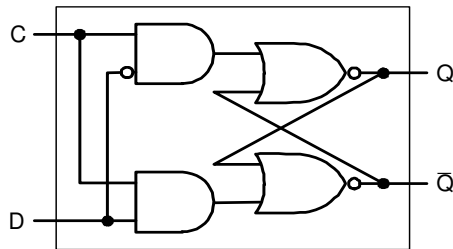


A clocking methodology defines when signals can be read and written  
— wouldn't want to read a signal at the same time it was being written

# D-latch

---

- Two inputs:
  - the data value to be stored (D)
  - the clock signal (C) indicating when to read & store D
- Two outputs:
  - the value of the internal state (Q) and it's complement

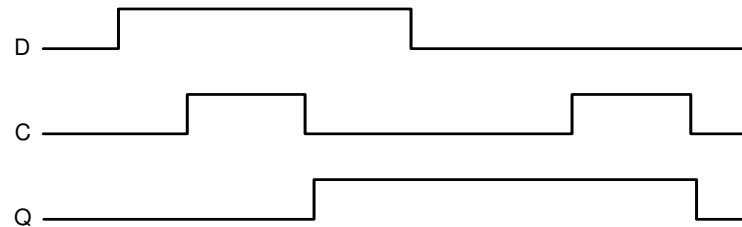
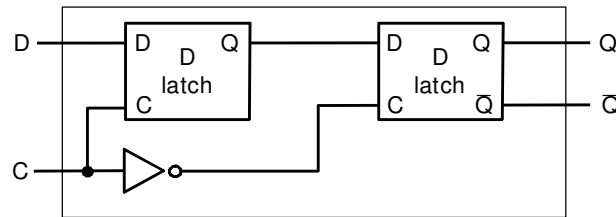




# D flip-flop

---

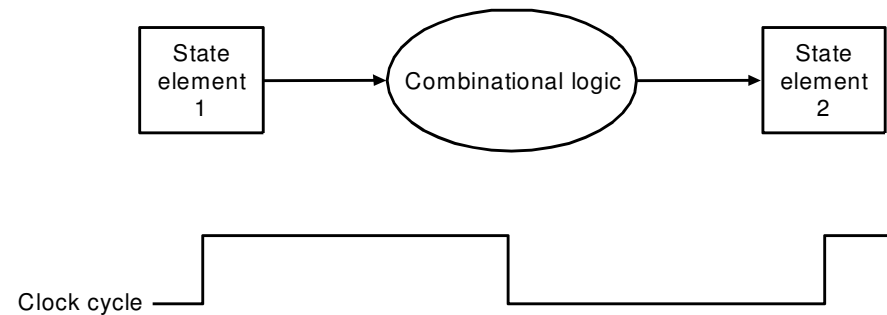
- Output changes only on the clock edge



# Our Implementation

---

- An edge triggered methodology
- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements



# Register File

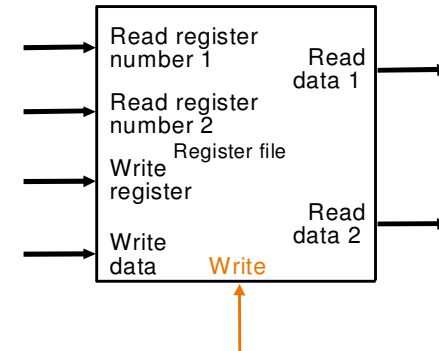
---

- Built using D flip-flops

Nell'architettura MIPS si ha un *register file* a 4 ingressi e 2 uscite: due ingressi rappresentano la coppia di registri in lettura, un ingresso è per selezionare il registro da scrivere ed uno per il dato da scrivere.

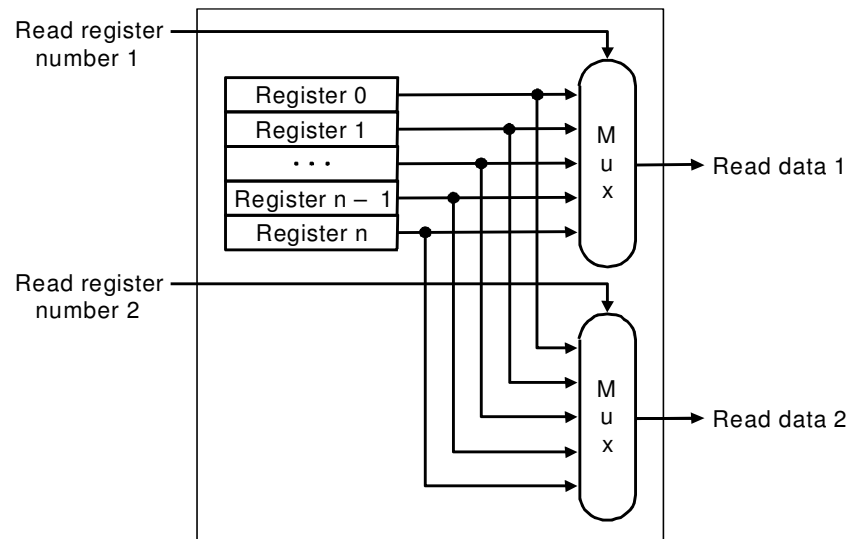
Le due uscite permettono di leggere 2 dati allo stesso tempo.

Inoltre c'è un segnale di sincronismo per la scrittura.



# Register File: lettura

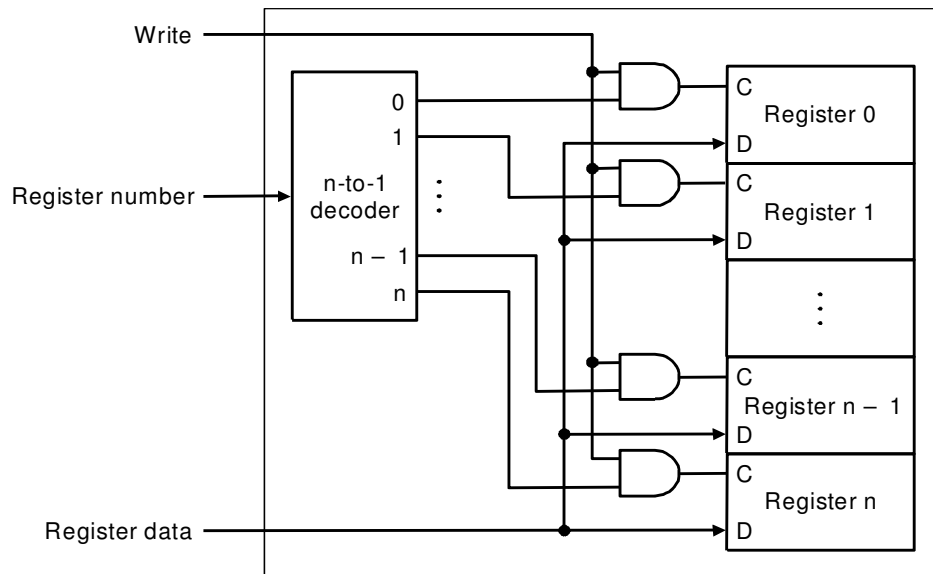
- Built using D flip-flops



Implementazione lettura 2 registri tramite due multiplexer (mux) in cui l'ingresso di selezione è il numero fornito in uno dei due ingressi al register file

# Register File: scrittura

- Note: we still use the real clock to determine when to write



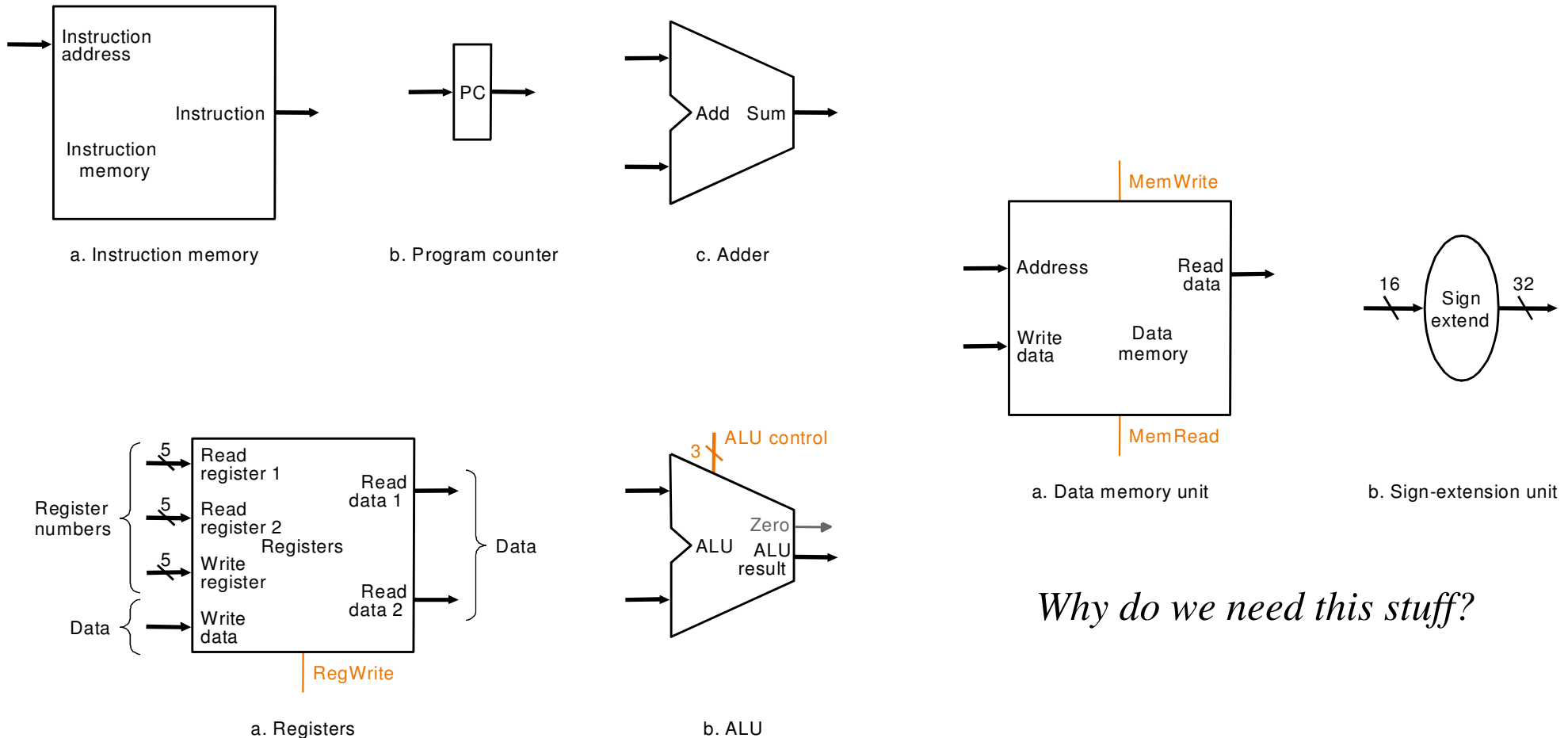
Il decodificatore seleziona il registro per la scrittura: il n° del registro fornito in ingresso seleziona solo una delle n uscite e quindi uno solo dei registri.

La scrittura avviene solo quando il write è affermato sul pin di clock del registro selezionato.

Non è possibile leggere e scrivere dallo stesso registro nello stesso ciclo di clock: o si legge il valore scritto nel ciclo precedente o si dovrà attendere un prossimo ciclo per poter leggere il valore appena scritto.

# Simple Implementation

- Include the functional units we need for each instruction

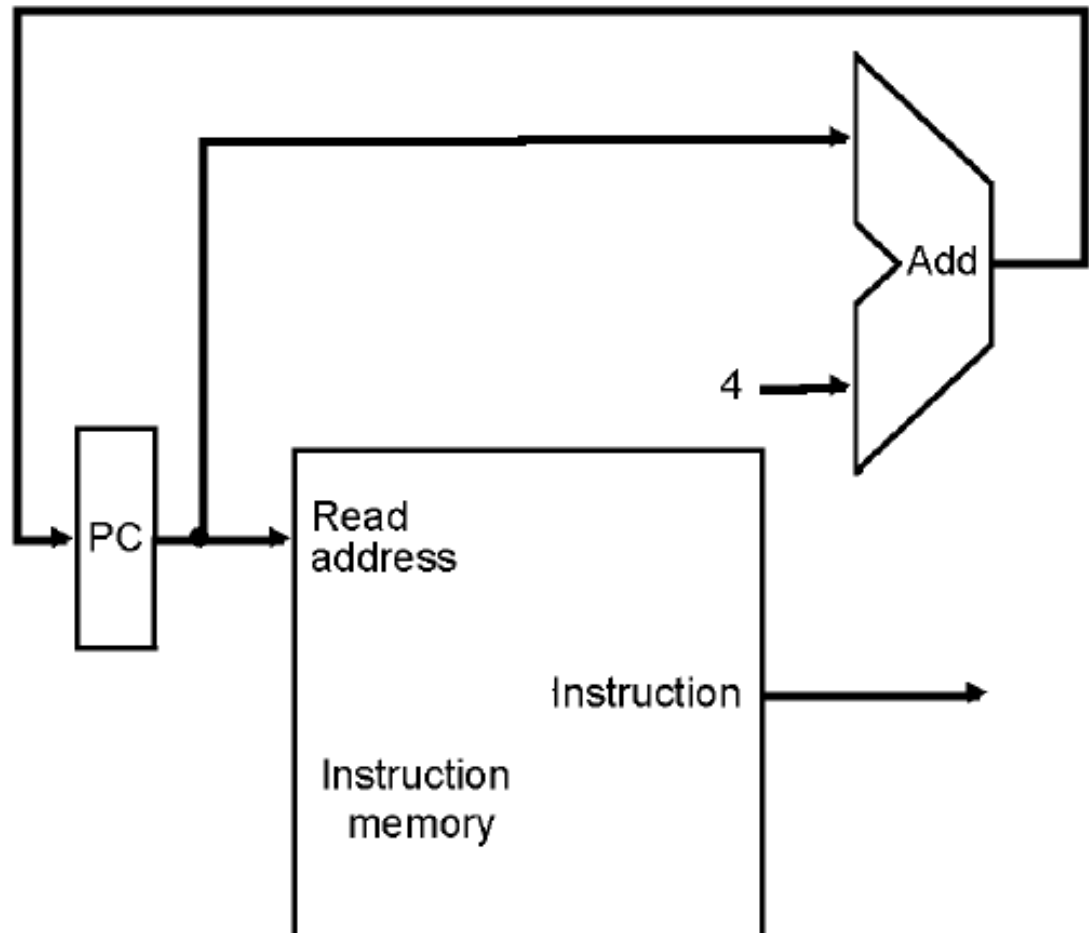


*Why do we need this stuff?*

# Instruction & PC

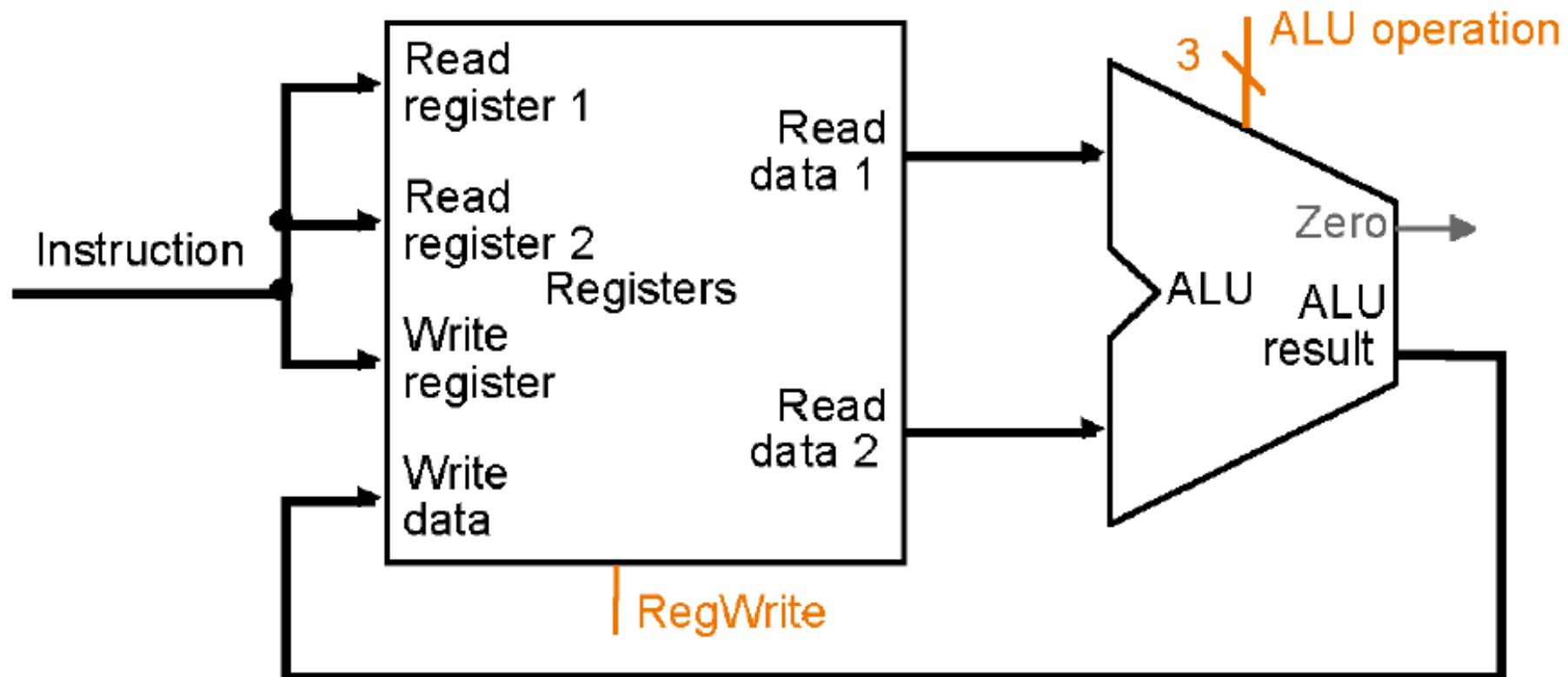
---

- **Increment:**  
add 4 to  
the PC to  
get to the  
address of  
the next  
instruction
- **Needs  
extension  
for Branch  
and Jump**



# Registers and ALU

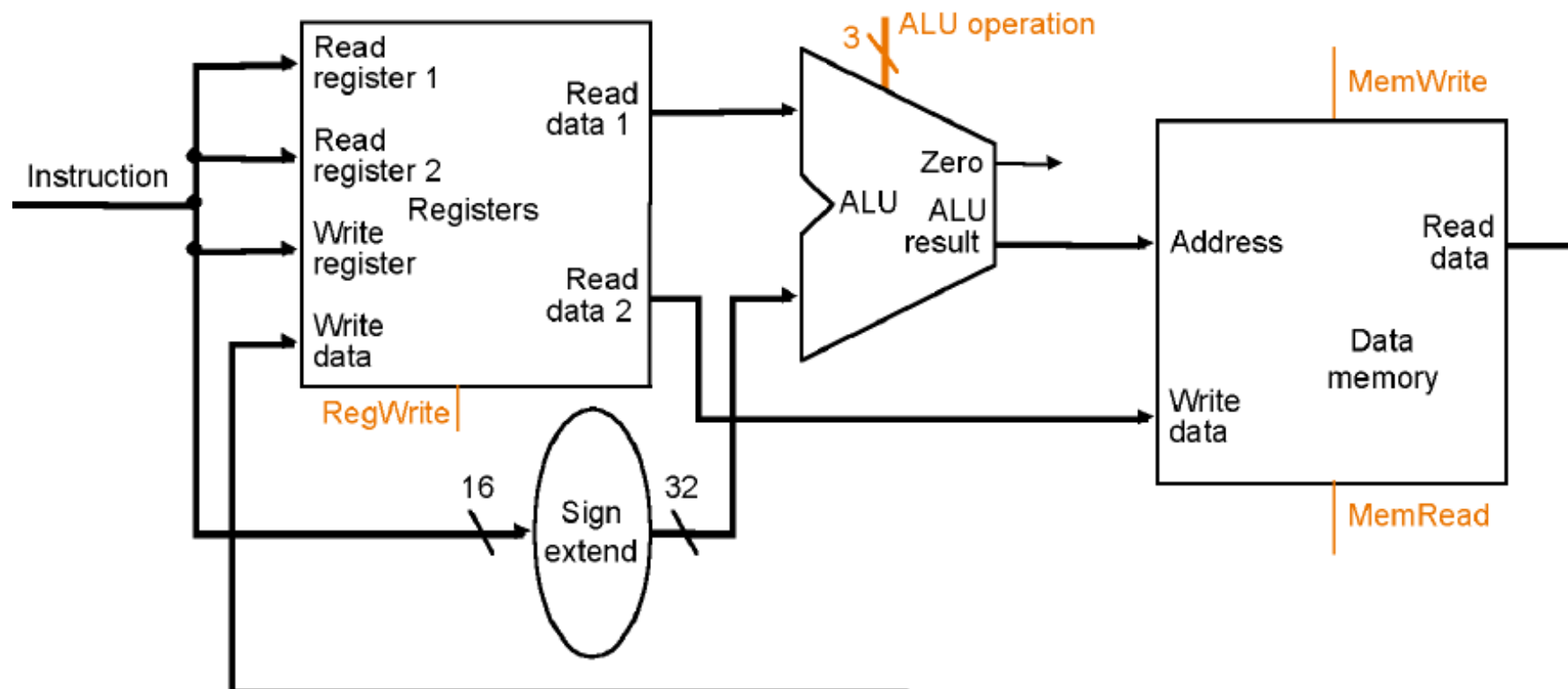
- Register file for R-format instructions as discussed previously
- ALU as designed in the previous chapter





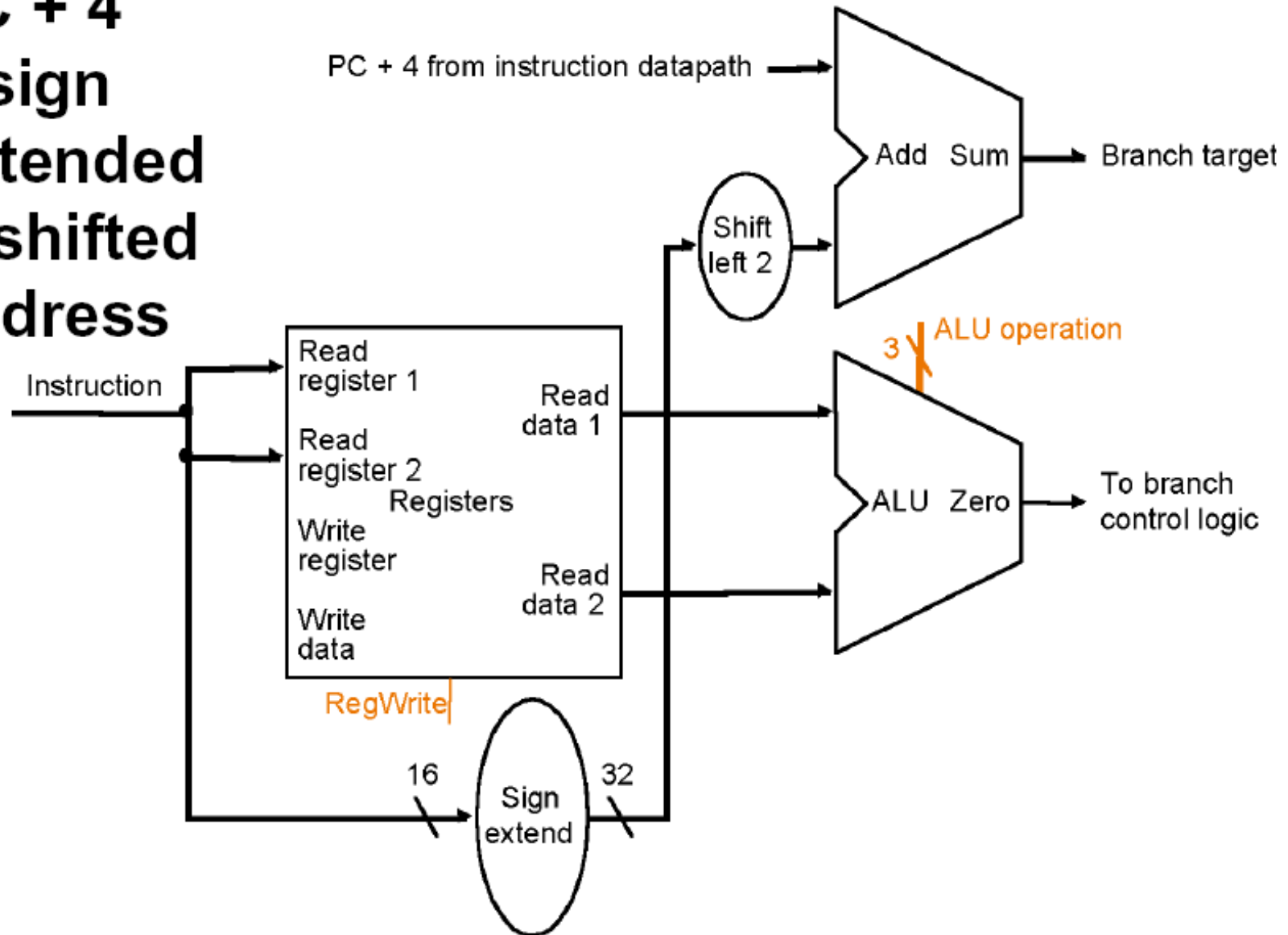
# Adding the data memory

- Transfer address to the memory
- Transfer data to the register file
- Sign extension unit for address calculation



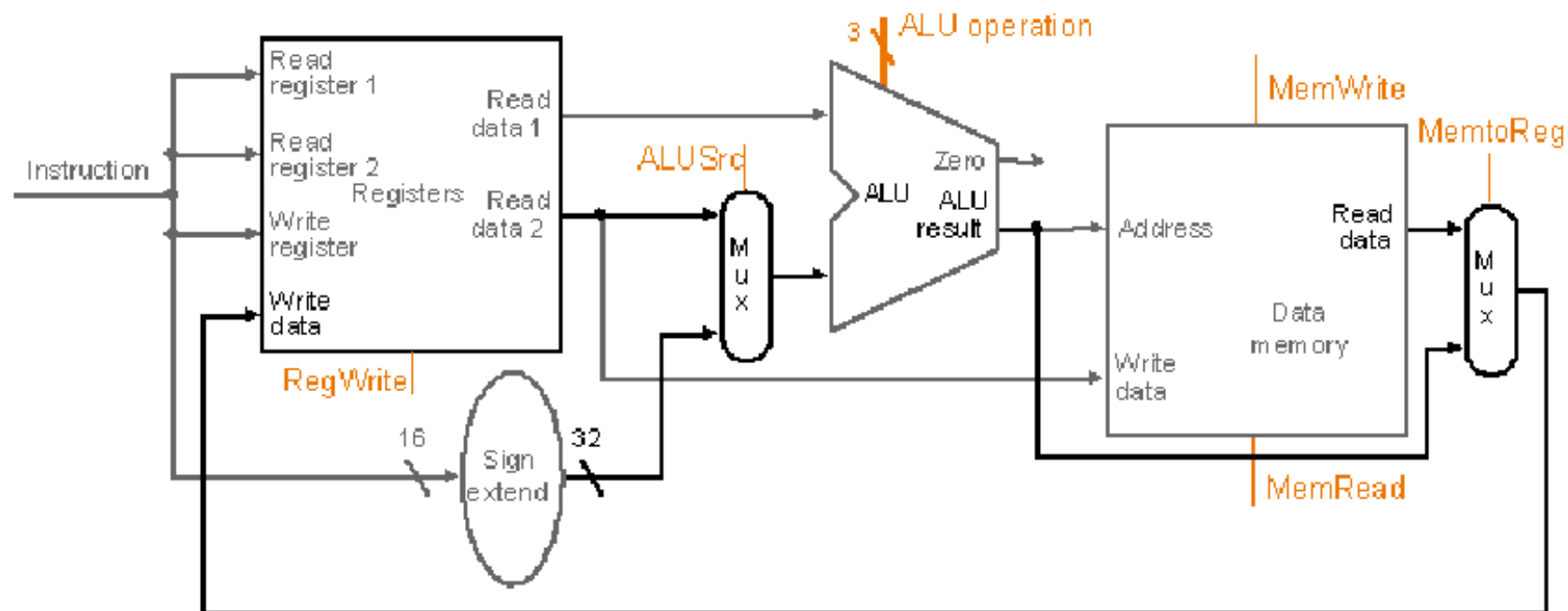
# Branch data path

■ **PC + 4**  
**+ sign**  
**extended**  
**& shifted**  
**address**



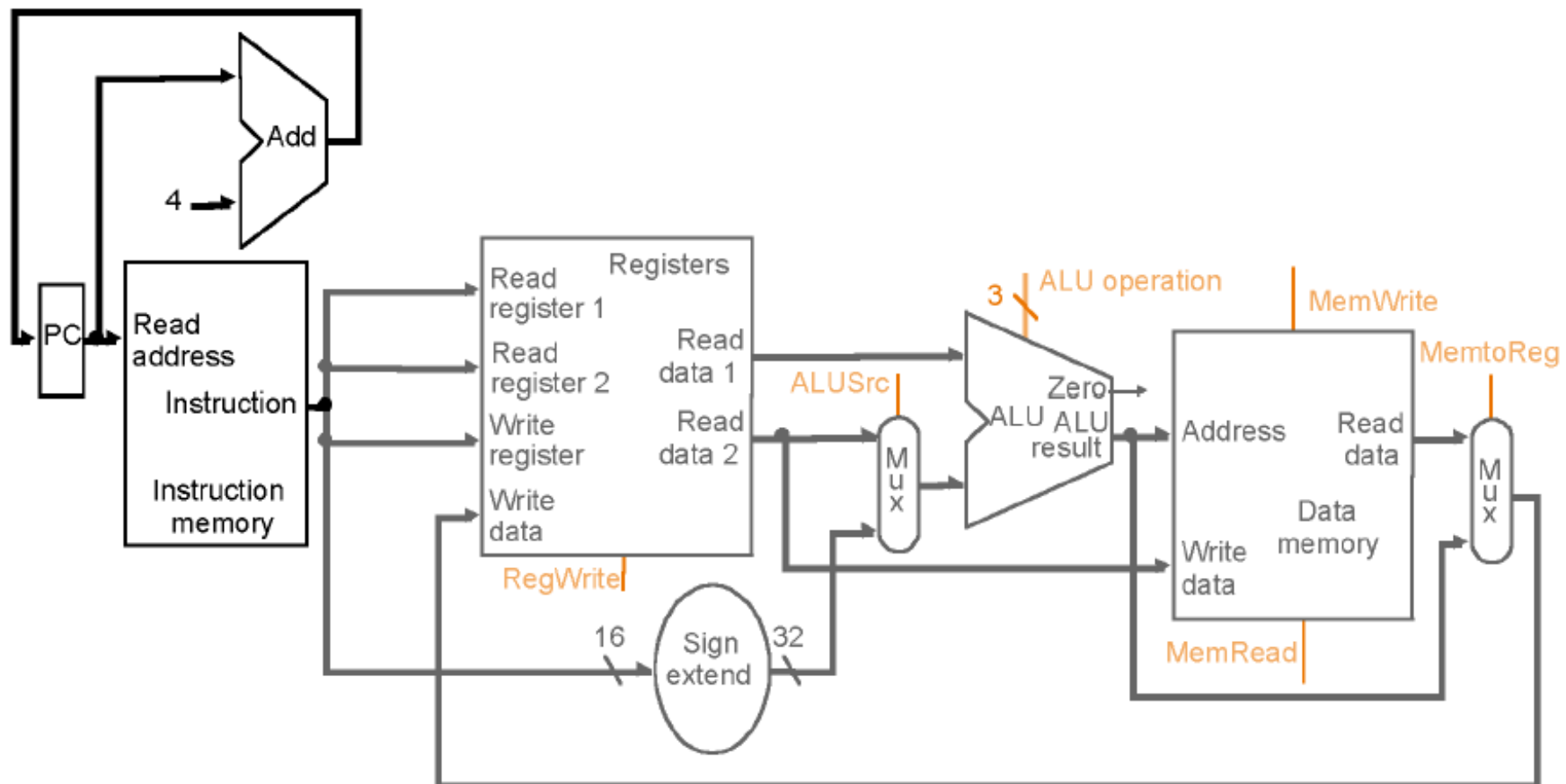
# Putting things together

- Register set
- ALU for operand and address calculation
- Data memory



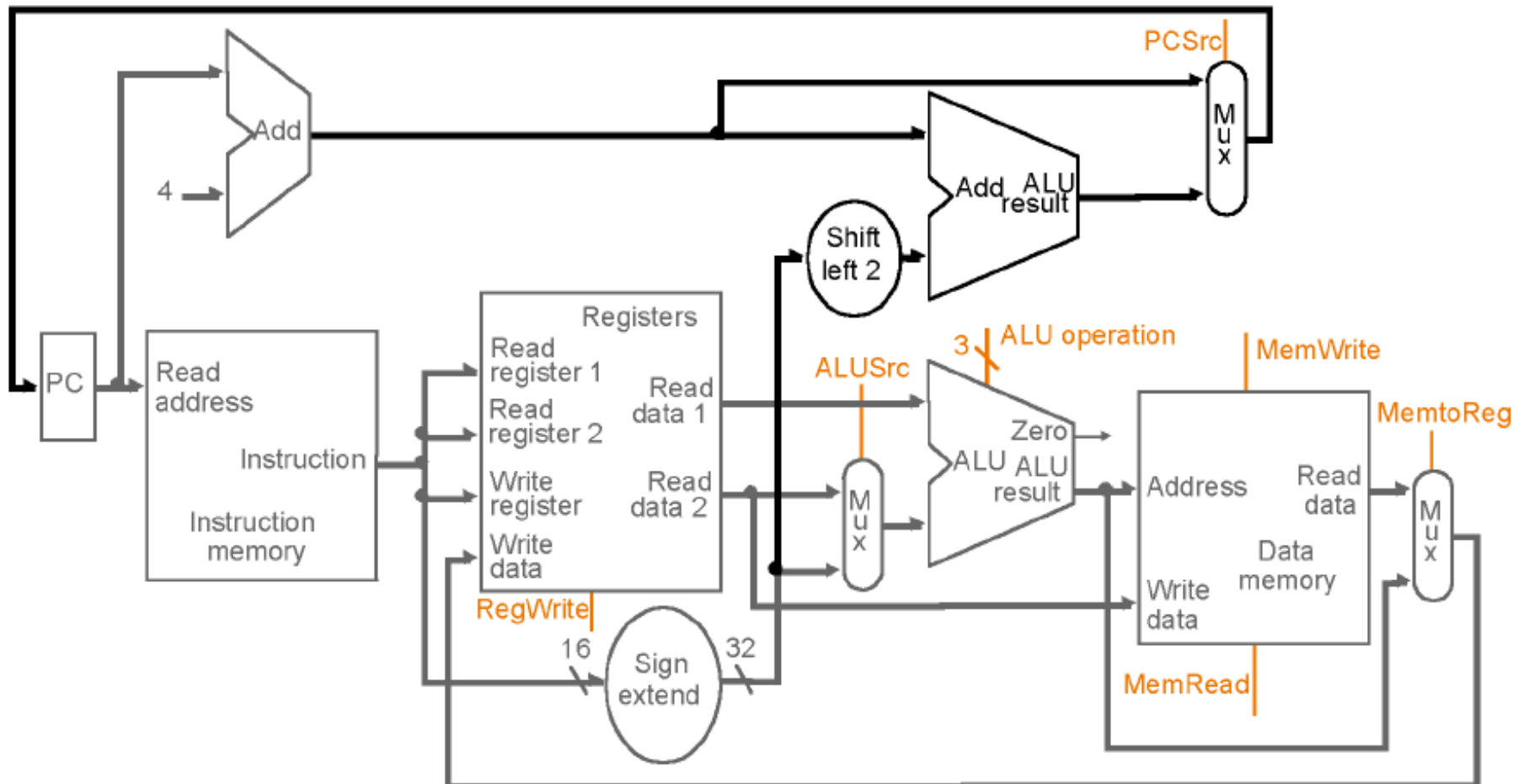
# Adding the instruction fetch unit

- The instruction fetch unit provides the instruction



# Adding branch logic

## ■ Supports now all basic instructions

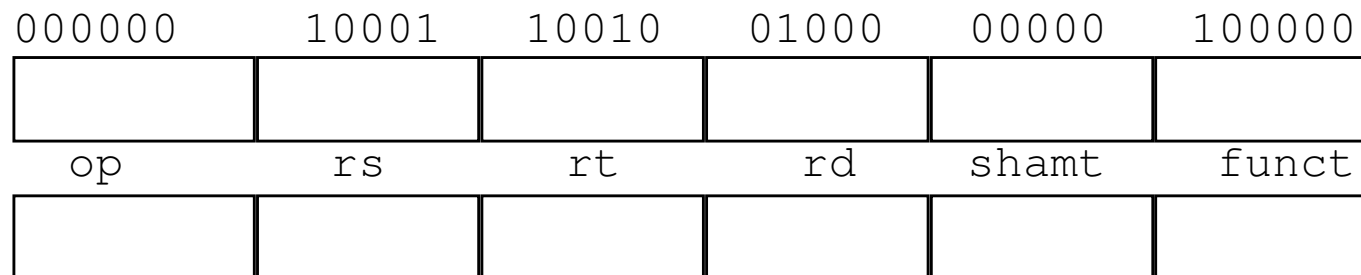


# Control

---

- Selecting the operations to perform (ALU, read/write, etc.)
- Controlling the flow of data (multiplexor inputs)
- Information comes from the 32 bits of the instruction
- Example:

add \$8, \$17, \$18    Instruction Format:



- ALU's operation based on instruction type and function code

# Control

---

- e.g., what should the ALU do with this instruction
- Example: lw \$1, 100(\$2)

35	2	1	100
----	---	---	-----

op	rs	rt	16 bit offset
----	----	----	---------------

- ALU control input

000	AND
001	OR
010	add
110	subtract
111	set-on-less-than

- Why is the code for subtract 110 and not 011?

## ■ Review of ALU functions / control lines

ALU Control lines	Function
000	And
001	Or
010	Add
110	Subtract
111	Set-on-less-than

## ■ ALU control bits: origin of

- ◆ 00: Addition (load & store word)
- ◆ 01 : Subtraction (branch on equal)
- ◆ 10 : Operations according the value in the function field (R-type instructions)



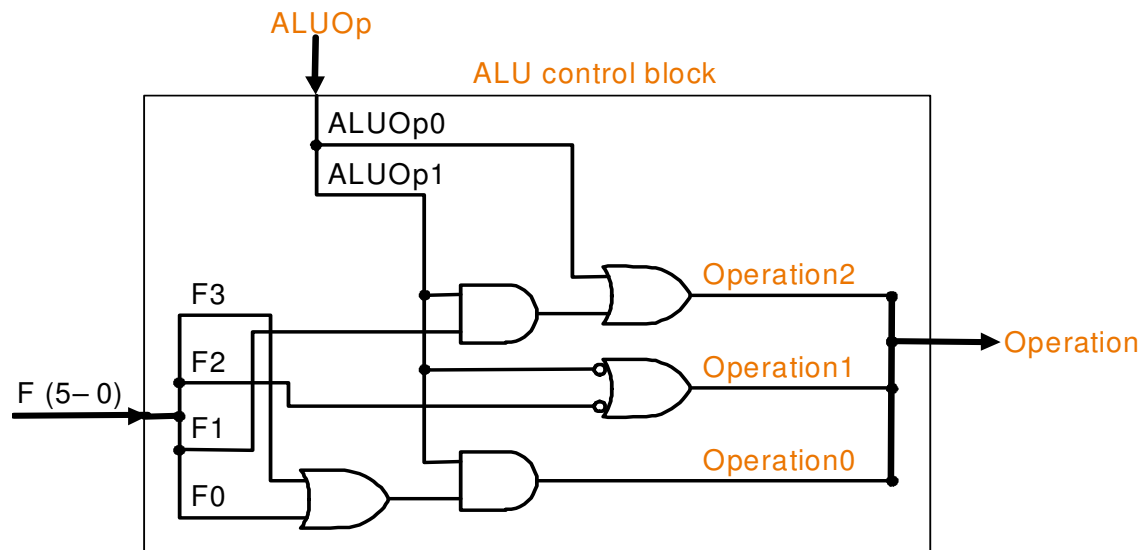
# ALU control

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	010
SW	00	store word	XXXXXX	add	010
Branch equal	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less than	101010	set on less than	111

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

# Funzione di controllo ALU

ALUop1	ALUop0	F5	F4	F3	F2	F1	F0	
x	1	x	x	x	x	x	x	Operation 2
1	x	x	x	x	x	1	x	
ALUop1	ALUop0	F5	F4	F3	F2	F1	F0	
0	x	x	x	x	x	x	x	Operation 1
x	x	x	x	x	0	x	x	
ALUop1	ALUop0	F5	F4	F3	F2	F1	F0	
1	x	x	x	x	x	x	1	Operation 0
1	x	x	x	1	x	x	x	



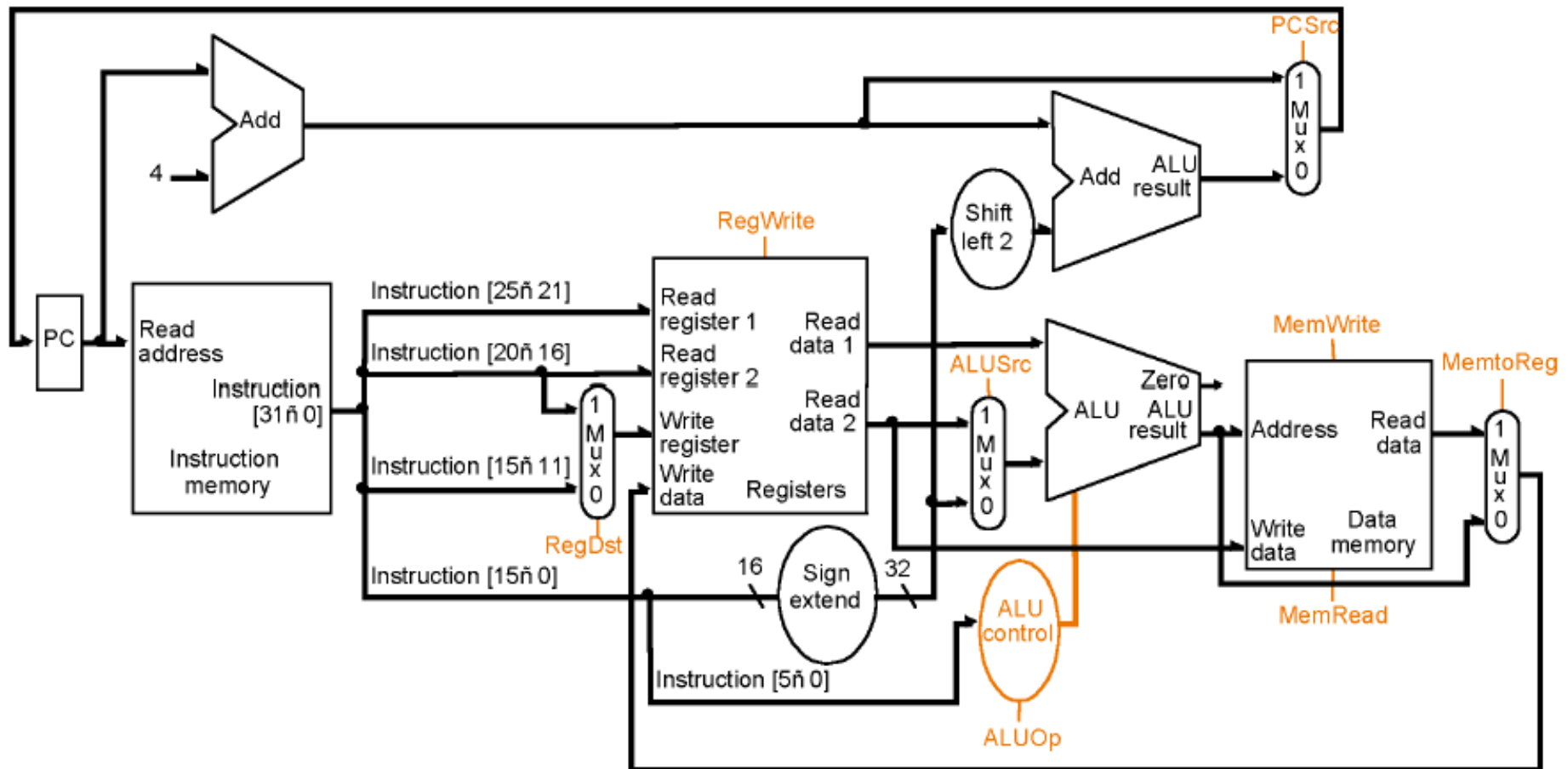
# Main control

---

- **Source information: instruction**
- **Operation code: Op [31-26]**
- **Read registers: rs [25-21], rt [20-16]**
- **Base register (LW, SW): rs [25-21]**
- **Destination register**
  - ◆ **Load: rt [20-16]**
  - ◆ **R-type: rd [15-11]**
  - ◆ **Requires a multiplexor**

# Extended data path

- Multiplexors for write register select
- All control lines



# Summary of control lines

---

- ◆ **RegDest**

Source of the destination register for the operation

- ◆ **RegWrite**

Enables writing a register in the register file

- ◆ **ALUsrc**

Source of second ALU operand, can be a register or part of the instruction

- ◆ **PCsrc**

Source of the PC (increment  $[PC + 4]$  or branch)

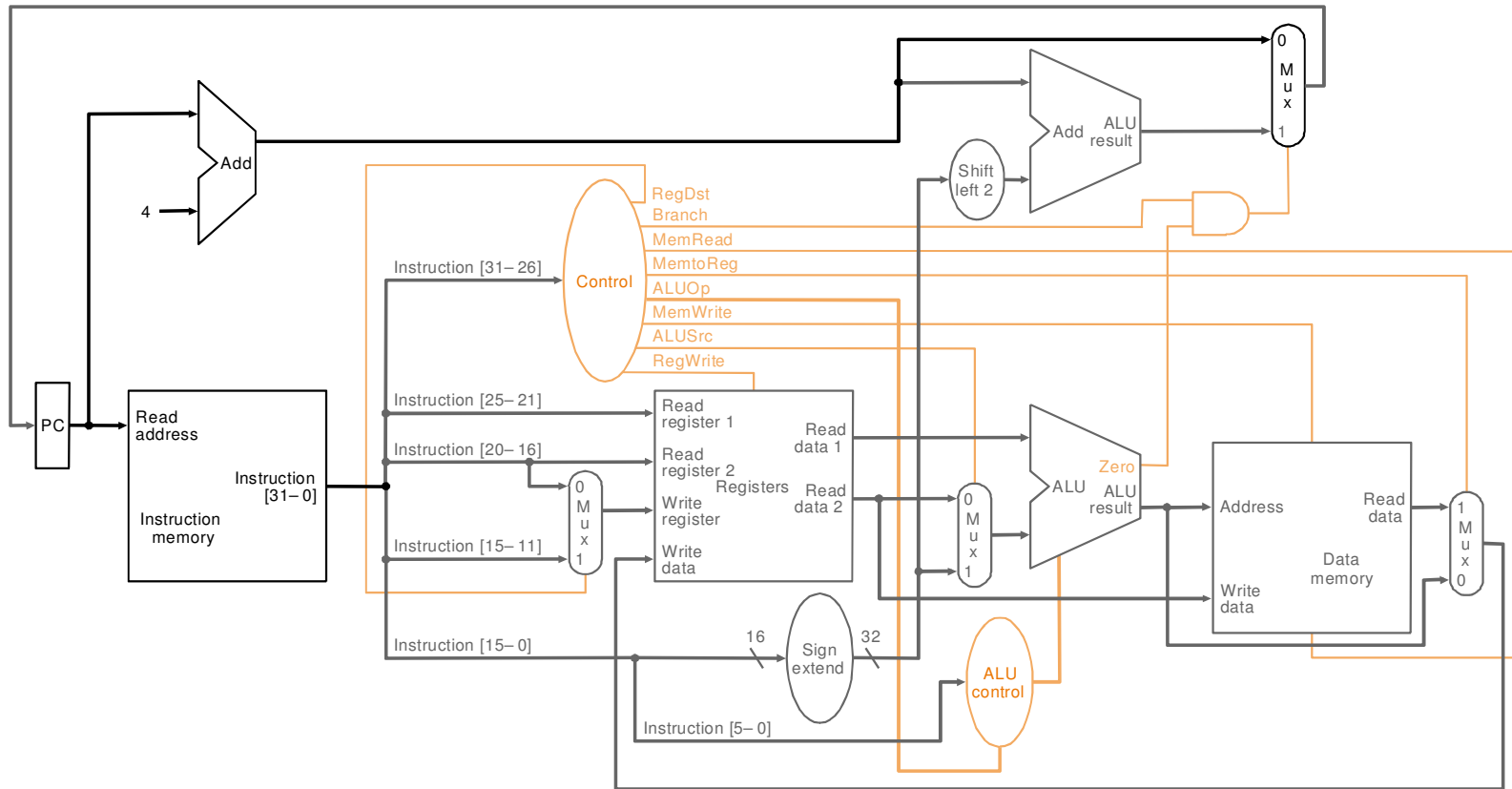
- ◆ **MemRead / MemWrite**

Reading / Writing from memory

- ◆ **MemtoReg**

Source of write register contents

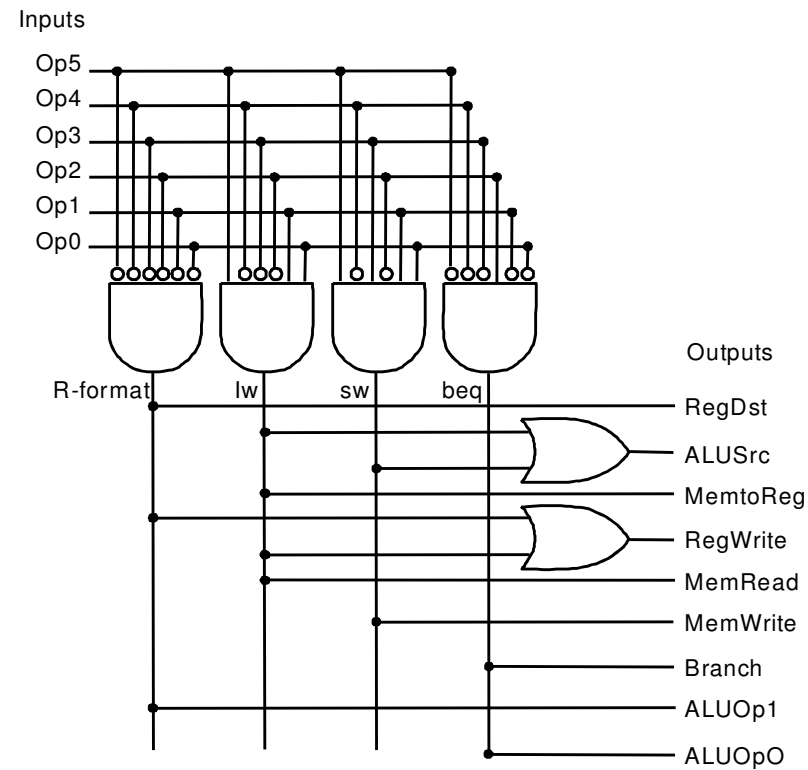
# Control



Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

# Main control

- Simple combinational logic (truth tables)



# Jump instruction

---

- **Extension of the previous architecture**
- **Jump address**
- **Lower 2 bits always 0**
- **Immediate field: bits 2-27**
- **Current PC: bits 28-31**

Instruction

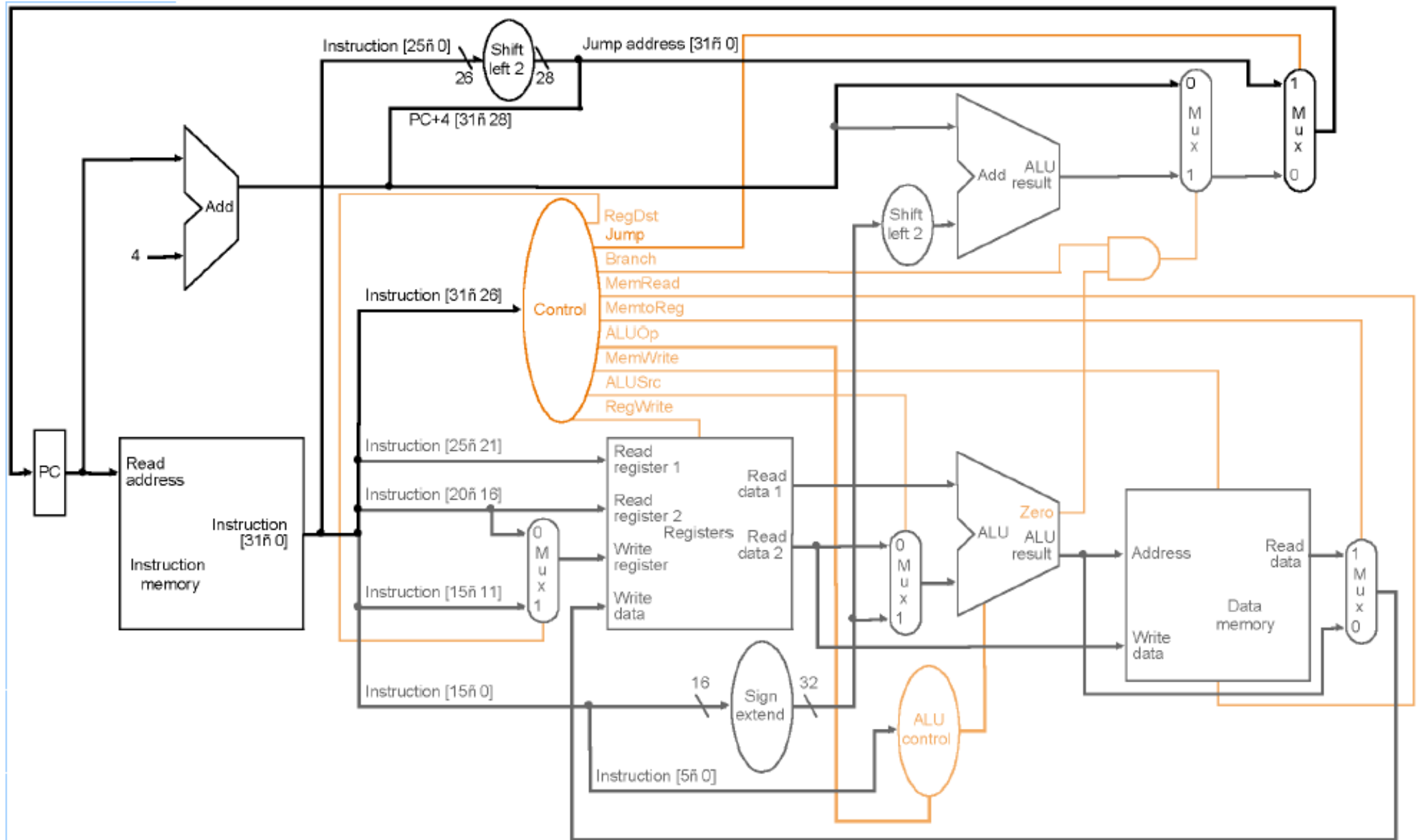
op	address
2	80000

Address

PC	jump address field	0
----	--------------------	---



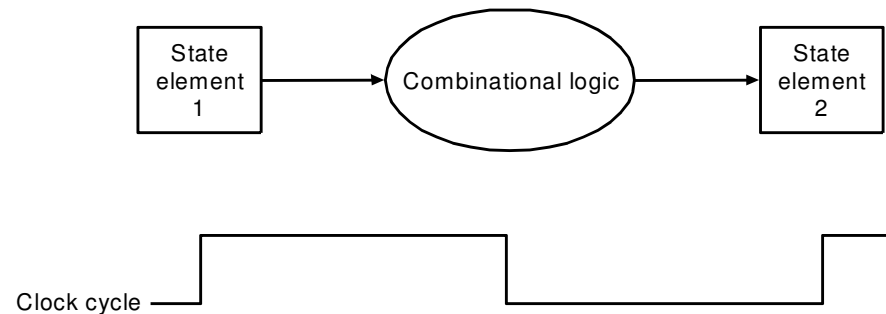
# Data path supporting jump



# Our Simple Control Structure

---

- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce “right answer” right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path

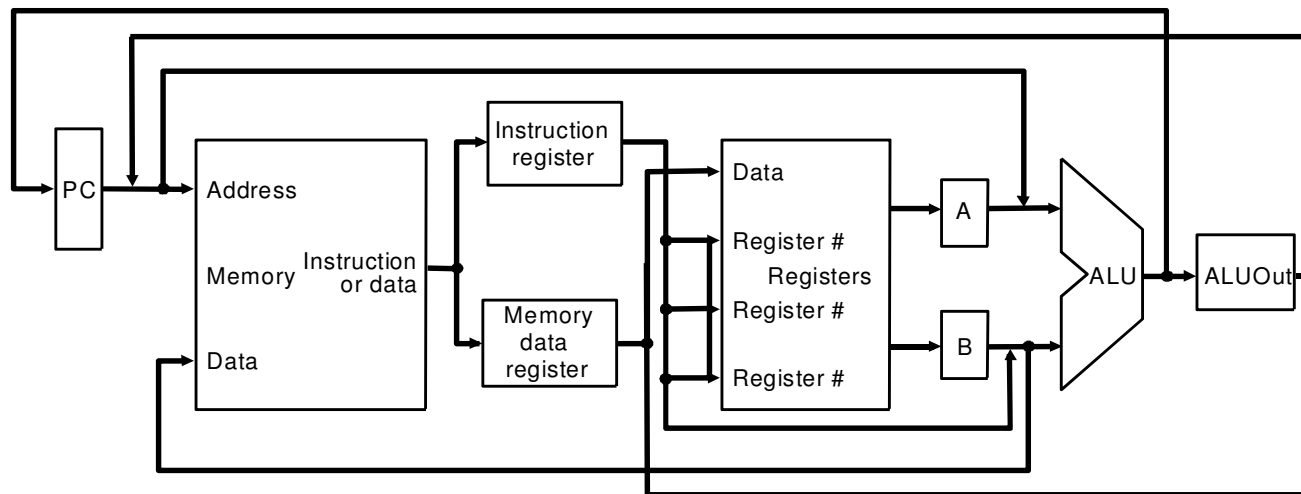


*We are ignoring some details like setup and hold times*

# Where we are headed

---

- Single Cycle Problems:
  - what if we had a more complicated instruction like floating point?
  - wasteful of area
- One Solution:
  - use a “smaller” cycle time
  - have different instructions take different numbers of cycles
  - a “multicycle” datapath:



# Multicycle Approach

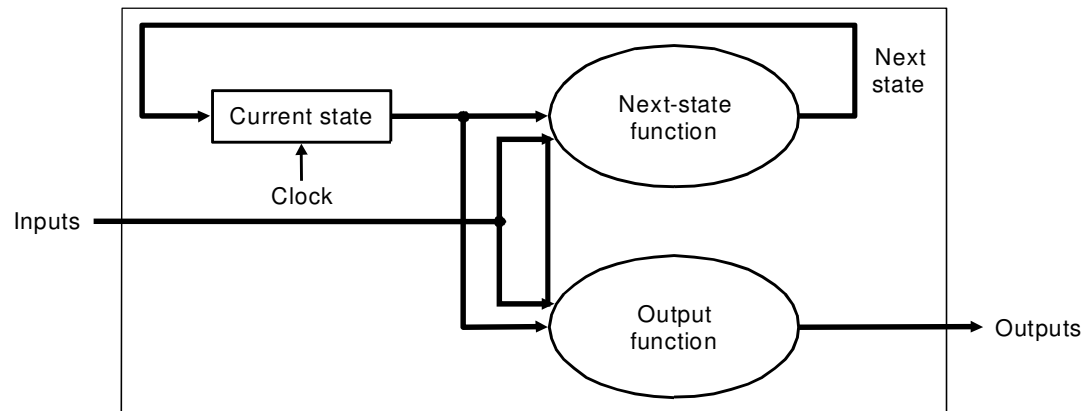
---

- We will be reusing functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- Our control signals will not be determined solely by instruction
  - e.g., what should the ALU do for a “subtract” instruction?
- We’ll use a finite state machine for control

# Review: finite state machines

---

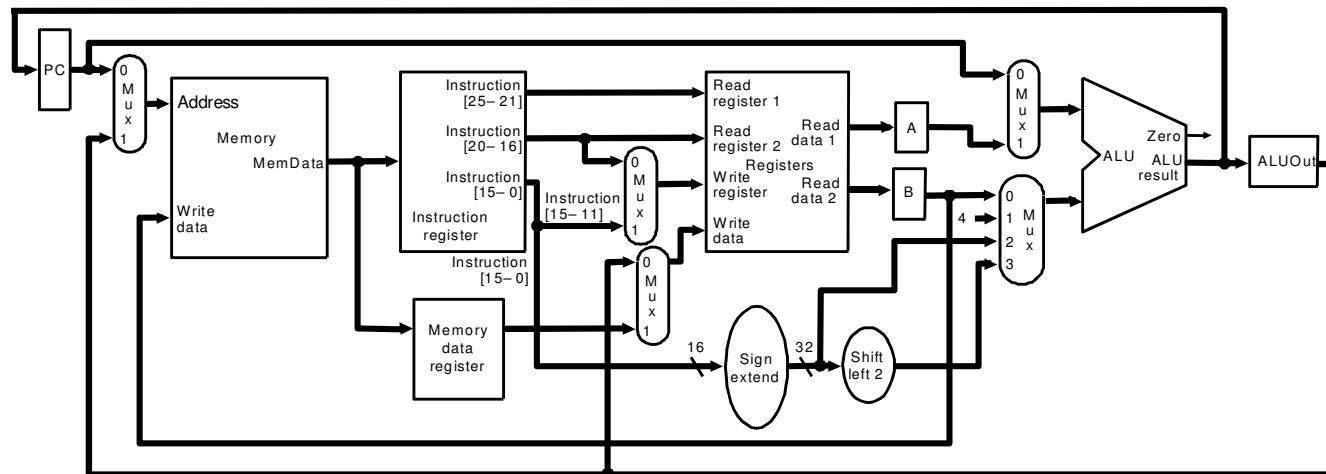
- Finite state machines:
  - a set of states and
  - next state function (determined by current state and the input)
  - output function (determined by current state and possibly input)



- We'll use a Moore machine (output based only on current state)

# Multicycle Approach

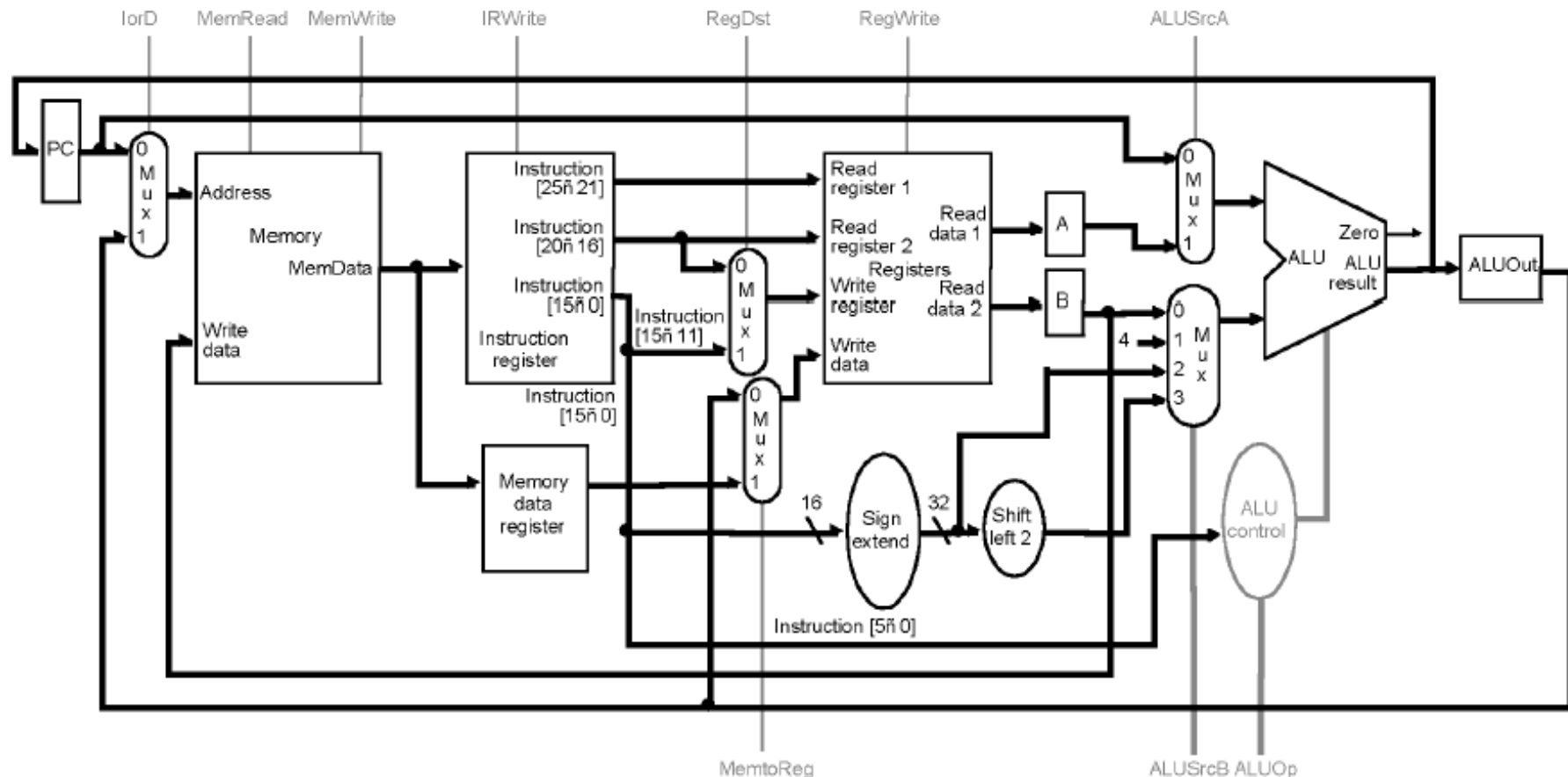
- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work to be done
  - restrict each cycle to use only one major functional unit
- At the end of a cycle
  - store values for use in later cycles (easiest thing to do)
  - introduce additional “internal” registers



# Data path

# Data path with control signals

- ◆ Memory address
- ◆ Memory R/W
- ◆ IR write
- ◆ Register write
- ◆ Write register destination
- ◆ Write register data source
- ◆ ALU source multiplexors
- ◆ ALU control





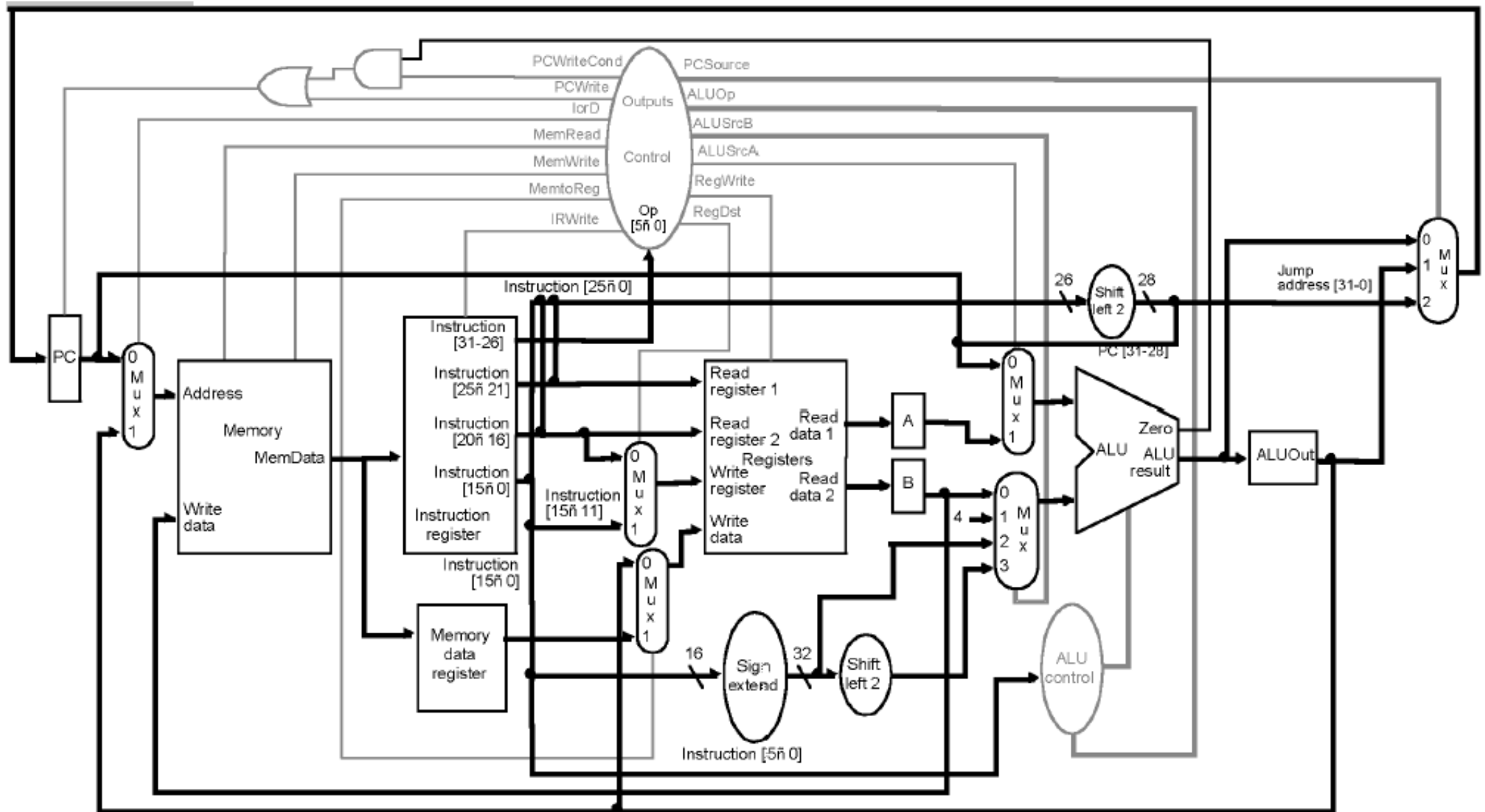
## ■ Multiplexor selecting the source for the PC

- ◆ ALU output after  $PC + 4$  (normal instruction)
- ◆ ALUOut after branch target address calculation
- ◆ New address with a jump instruction

## ■ Writing the PC

- ◆ Unconditionally after a normal instruction and jump
- ◆ Conditionally overwritten if a branch is taken
  - I PCWriteCond allows conditional loading
  - I Zero bit from ALU decides if the PC is reloaded with the branch target address

# Complete data path & control



# Control signals

---

**Actions of the 1-bit control signals**

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None	Memory contents at the location specified by the Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
lorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None	The output of the memory is written into the IR.
PCWrite	None	The PC is written; the source is controlled by PCSource.
PCWriteCond	None	The PC is written if the Zero output from the ALU is also active.

# Control signals

---

**Actions of the 2-bit control signals**

Signal name	Value	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left 2 bits.
PCSource	00	Output of the ALU ( $PC + 4$ ) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address ( $IR[25-0]$ shifted left 2 bits and concatenated with $PC + 4[31-28]$ ) is sent to the PC for writing.

# Breaking the instructions

---

## ■ Move from one-cycle to multi-cycle

- ◆ Identifying steps that take one cycle
- ◆ Equal distribution of execution time
- ◆ At most one operation for each of the modules
  - I ALU
  - I Register file
  - I Memory

## ■ New registers if

- ◆ The signal is computed in one cycle and used in another cycle
- ◆ The inputs of the block generating the signal may change in the second cycle

# Five Execution Steps

---

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back step

*INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!*

# Step 1: Instruction Fetch

---

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL "Register-Transfer Language"

```
IR = Memory[PC];  
PC = PC + 4;
```

*Can we figure out the values of the control signals?*

*What is the advantage of updating the PC now?*

## ■ Load instruction from memory

## IR = Memory [PC]

- ◆ Set Read address mux (lorD) = 0 select instruction
- ◆ Set MemRead = 1

## ■ Increment PC

**PC = PC + 4**

- ◆ Set ALUSrcA = 0 get operand from IR
- ◆ Set ALUSrcB = 01 get operand '4'
- ◆ Set ALUOp = 00 add
- ◆ Allow storing new PC in PC register



## Step 2: Instruction Decode and Register Fetch

---

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:

```
A = Reg[IR[25-21]];  
B = Reg[IR[20-16]];  
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```

- We aren't setting any control lines based on the instruction type (we are busy "decoding" it in our control logic)

## Step 2: Instruction Decode and Register Fetch

---

### ■ Switch registers to the output of the register block

**A = register [IR [25-21]]      rs**

**B = register [IR [20-16]]      rt**

◆ No signal setting required

### ■ Calculate the branch target address

**target = PC + (sign-ext. (IR [15-0]) << 2)**

◆ Stored in the ALUOut register

◆ Set      **ALUSrcB      = 11**

◆ Set      **ALUOp      = 00    add**

## Step 3 (instruction dependent)

---

- ALU is performing one of three functions, based on instruction type
- Memory Reference:

`ALUOut = A + sign-extend(IR[15-0]);`

- R-type:

`ALUOut = A op B;`

- Branch:

`if (A==B) PC = ALUOut;`

## Step 3 (instruction dependent)

---

- **Step depends on the instruction**
- **Selection performed by interpretation of the op + function field of the instruction**
- **Calculate memory reference address**  
 **$ALUOut = A + \text{sign-ext. (IR[15-0])}$** 
  - ◆ **Set ALUSrcA = 1** get operand from A
  - ◆ **Set ALUSrcB = 10** get operand from sign extension unit
  - ◆ **Set ALUOp = 00** add

## Step 3 (instruction dependent)

---

### ■ Arithmetic-logical instruction (R-type)

$$\text{ALUOut} = A \text{ op } B$$

- ◆ Set ALUSrcA = 1 get operand from A
- ◆ Set ALUSrcB = 00 get operand from B
- ◆ Set ALUOp = 10 code from IR

### ■ Branch: if (A == B)      PC = ALUOut

- ◆ Set ALUSrcA = 1 get operand from A
- ◆ Set ALUSrcB = 00 get operand from B
- ◆ Set ALUOp = 01 subtraction
- ◆ Write ALUOut to PC register

### ■ Jump:      PC = PC [31-28] || (IR[25-0] << 2)

## Step 4 (R-type or memory-access)

---

- Loads and stores access memory

```
MDR = Memory[ALUOut];  
    or  
Memory[ALUOut] = B;
```

- R-type instructions finish

```
Reg[IR[15-11]] = ALUOut;
```

*The write actually takes place at the end of the cycle on the edge*

## Step 4 (R-type or memory-access)

---

### ■ Memory access

- ◆ ALU controls must remain stable
- ◆ Set I or D = 1 address from ALU

**memory-data = memory [ALUOut]  
load from memory**

- ◆ Set MemRead = 1

**memory [ALUOut] = B  
store to memory**

- ◆ Set MemWrite = 1

## Step 4 (R-type or memory-access)

---

### ■ Arithmetic-logical instruction completion

**Register [IR [15-11]] = ALUOut**

- ◆ **Set      RegDst      = 1      Select write register**
- ◆ **Set      RegWrite    = 1      Allow write operation**
- ◆ **Set      MemToReg   = 0      Select ALU data**
  
- ◆ **ALUOp, ALUSrcA, ALUSrcB = constant**



# Write-back step

---

- $\text{Reg}[\text{IR}[20-16]] = \text{MDR};$

*What about all the other instructions?*

# Write-back step

---

## ■ Write data from memory to the register

**Reg [IR[20-16]] = memory-data**

- ◆ Set      RegDst      = 0      Select write rt  
as target register
- ◆ Set      RegWrite    = 1      Allow write operation
- ◆ Set      MemToReg   = 1      Select Memory data
  
- ◆ ALUOp, ALUSrcA, ALUSrcB = constant

# Summary

---

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	$IR = \text{Memory}[PC]$ $PC = PC + 4$			
Instruction decode/register fetch	$A = \text{Reg}[IR[25-21]]$ $B = \text{Reg}[IR[20-16]]$ $ALUOut = PC + (\text{sign-extend}(IR[15-0]) \ll 2)$			
Execution, address computation, branch/jump completion	$ALUOut = A \text{ op } B$	$ALUOut = A + \text{sign-extend}(IR[15-0])$	if $(A == B)$ then $PC = ALUOut$	$PC = PC[31-28] \parallel (IR[25-0] \ll 2)$
Memory access or R-type completion	$\text{Reg}[IR[15-11]] = ALUOut$	Load: $MDR = \text{Memory}[ALUOut]$ or Store: $\text{Memory}[ALUOut] = B$		
Memory read completion		Load: $\text{Reg}[IR[20-16]] = MDR$		

# Implementing the Control

---

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
- Use the information we've accumulated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming
- Implementation can be derived from specification

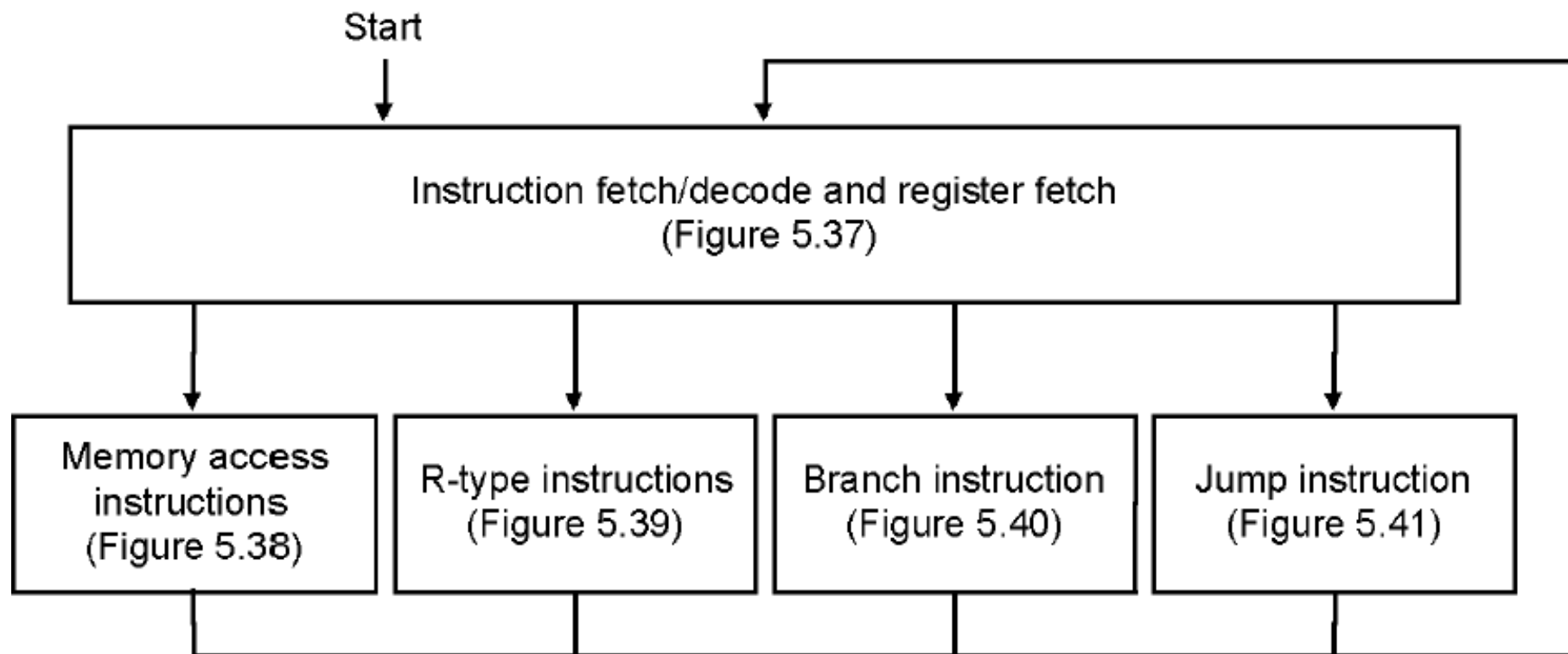
# General decomposition

---

## ■ Common part

- ◆ Instruction fetch
- ◆ Instruction decode/register fetch

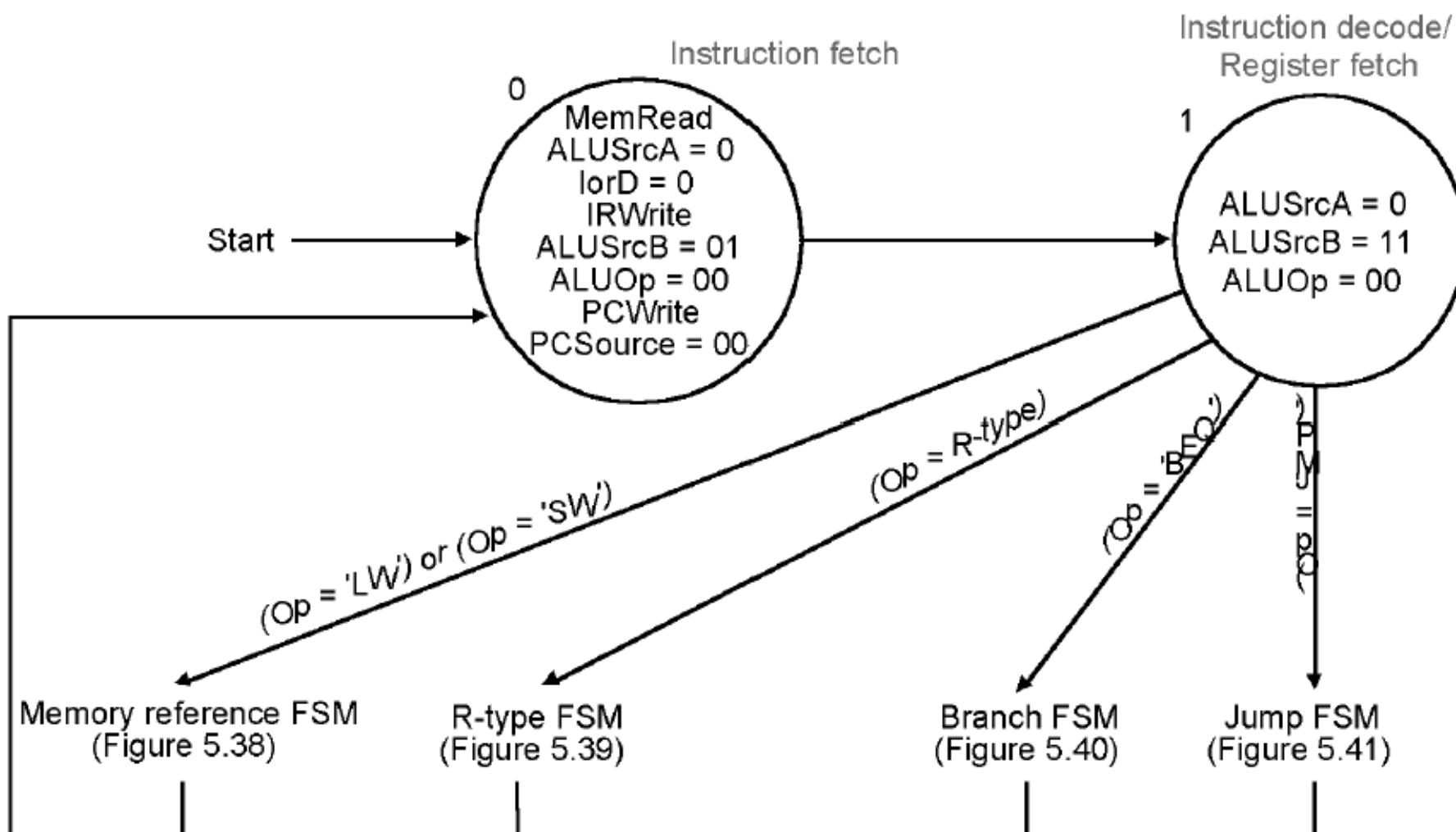
## ■ Instruction specific part



# Common part

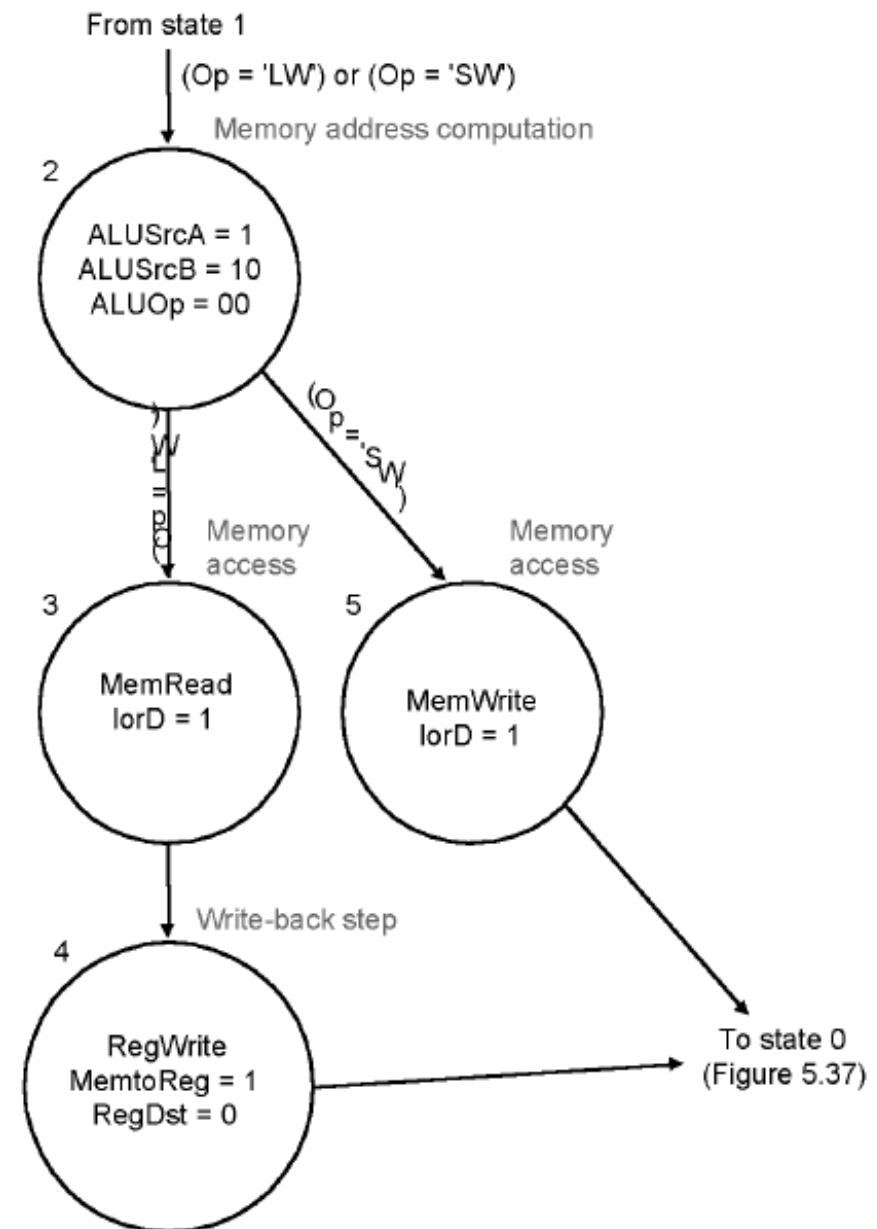
## ■ Instruction fetch

## ■ Instruction decode/register fetch



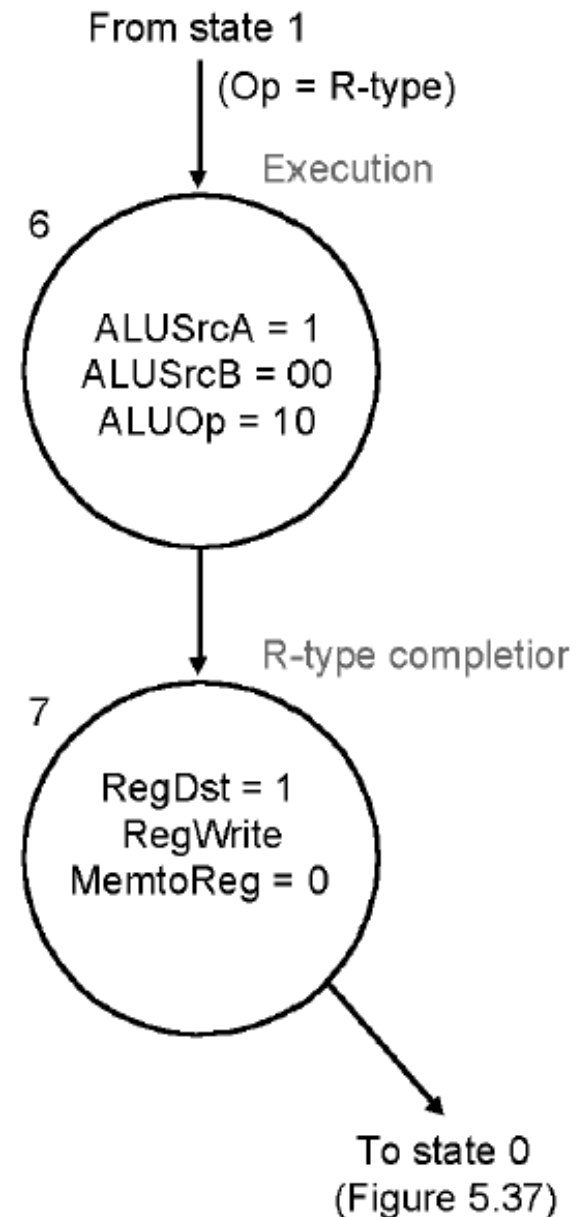
# Memory access

- **Address calculation**
- **Load sequence**
  - ◆ Read from memory
  - ◆ Store to register
- **Access memory**
- **Store sequence**  
**Write**



# R-type instruction

- Execution of instruction
- Completion  
Write result to register

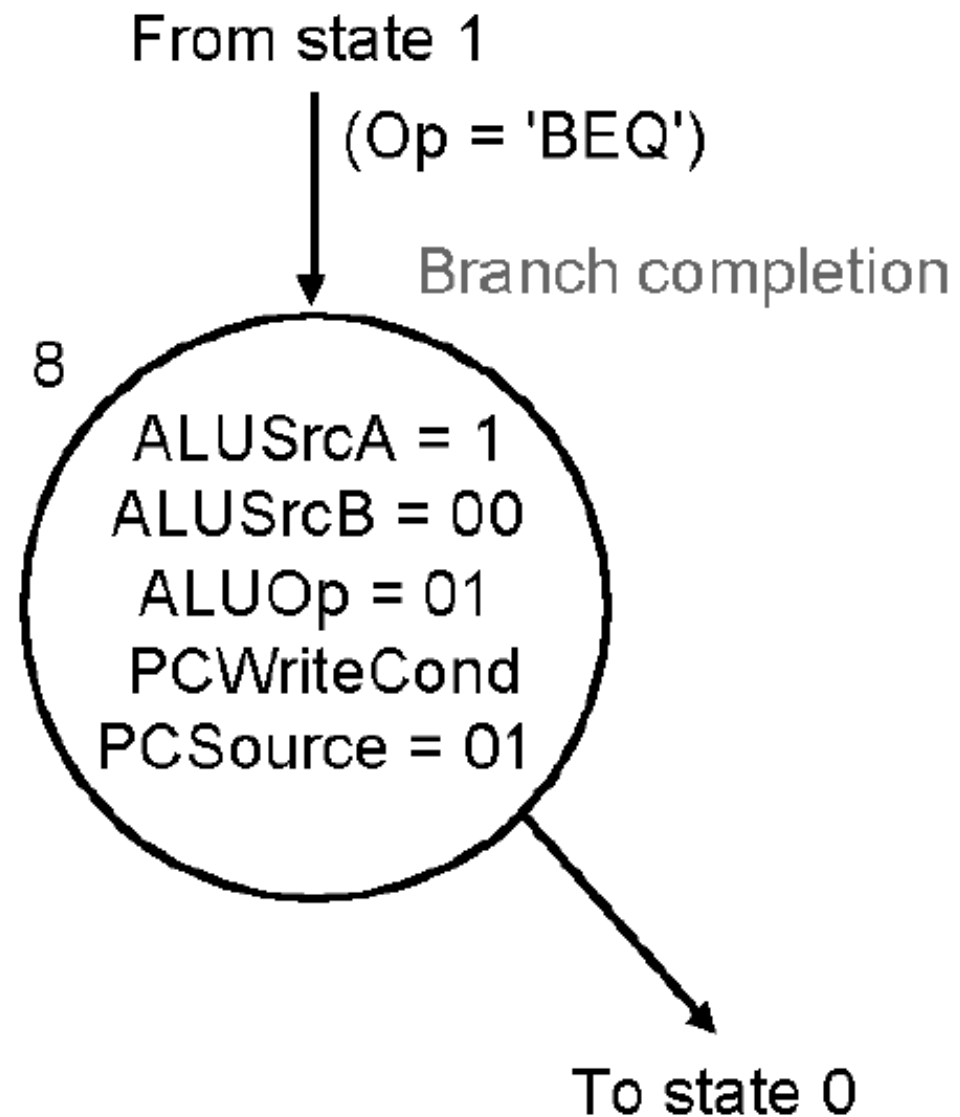




# Branch instruction

---

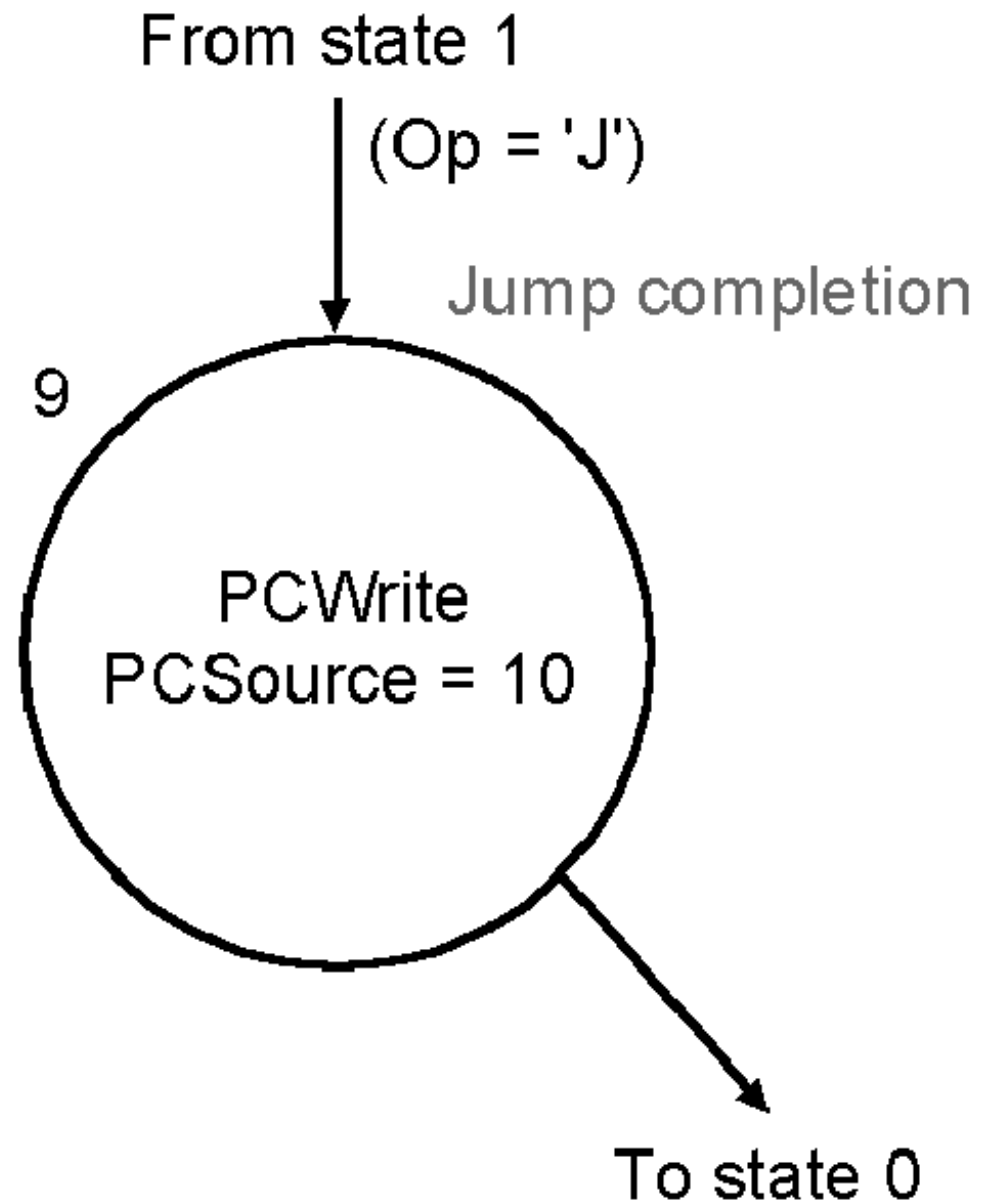
- **Single state**
- **PC is loaded with branch address**
- **Get next instruction**



# Jump instruction

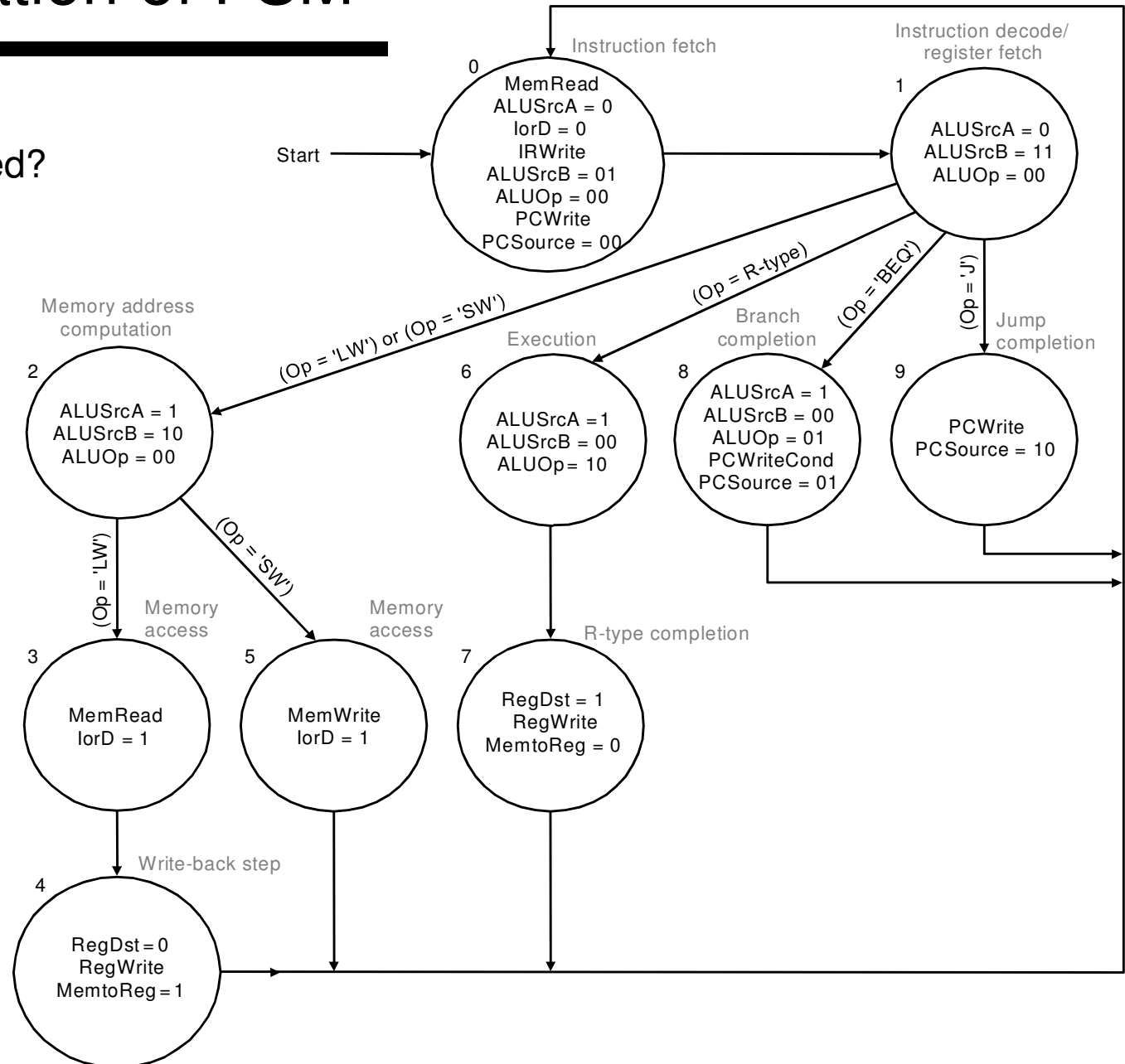
---

- **Load PC with new value**
- **Get next instruction**



# Graphical Specification of FSM

How many state bits will we need?



# Control signal summary

					States					
Signal Name	0	1	2	3	4	5	6	7	8	9
MemRead	1	0	0	1	1	0	0	0	0	0
MemWrite	0	0	0	0	0	1	0	0	0	0
ALUSrcA	0	0	1	1	1	1	1	1	1	-
ALUSrcB	01	11	10	10	10	10	00	00	00	-
ALUOp	00	00	00	00	00	00	10	10	01	-
RegDst	-	-	-	-	0	-	-	1	-	-
RegWrite	0	0	0	0	1	0	0	1	0	0
MemtoReg	-	-	-	-	1	-	-	0	-	-
lorD	0	-	-	1	1	1	-	-	-	-
IRWrite	1	0	0	0	0	0	0	0	0	0
PCWrite	1	0	0	0	0	0	0	0	0	1
PCWriteCond	0	0	0	0	0	0	0	0	1	0
TargetWrite	0	1	0	0	0	0	0	0	0	0

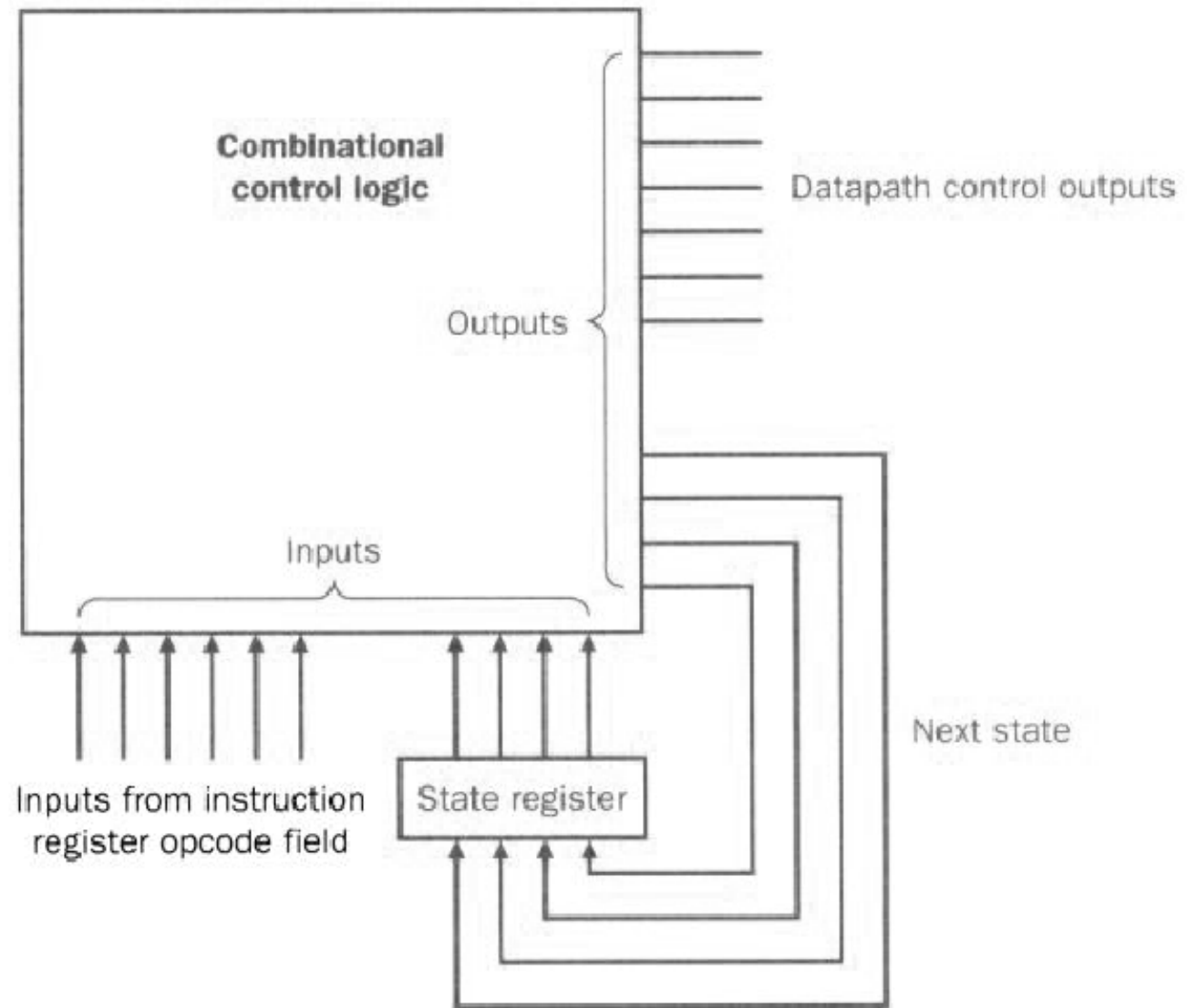
**I Black: active signals**

**I Red: inactive signals that must be set**

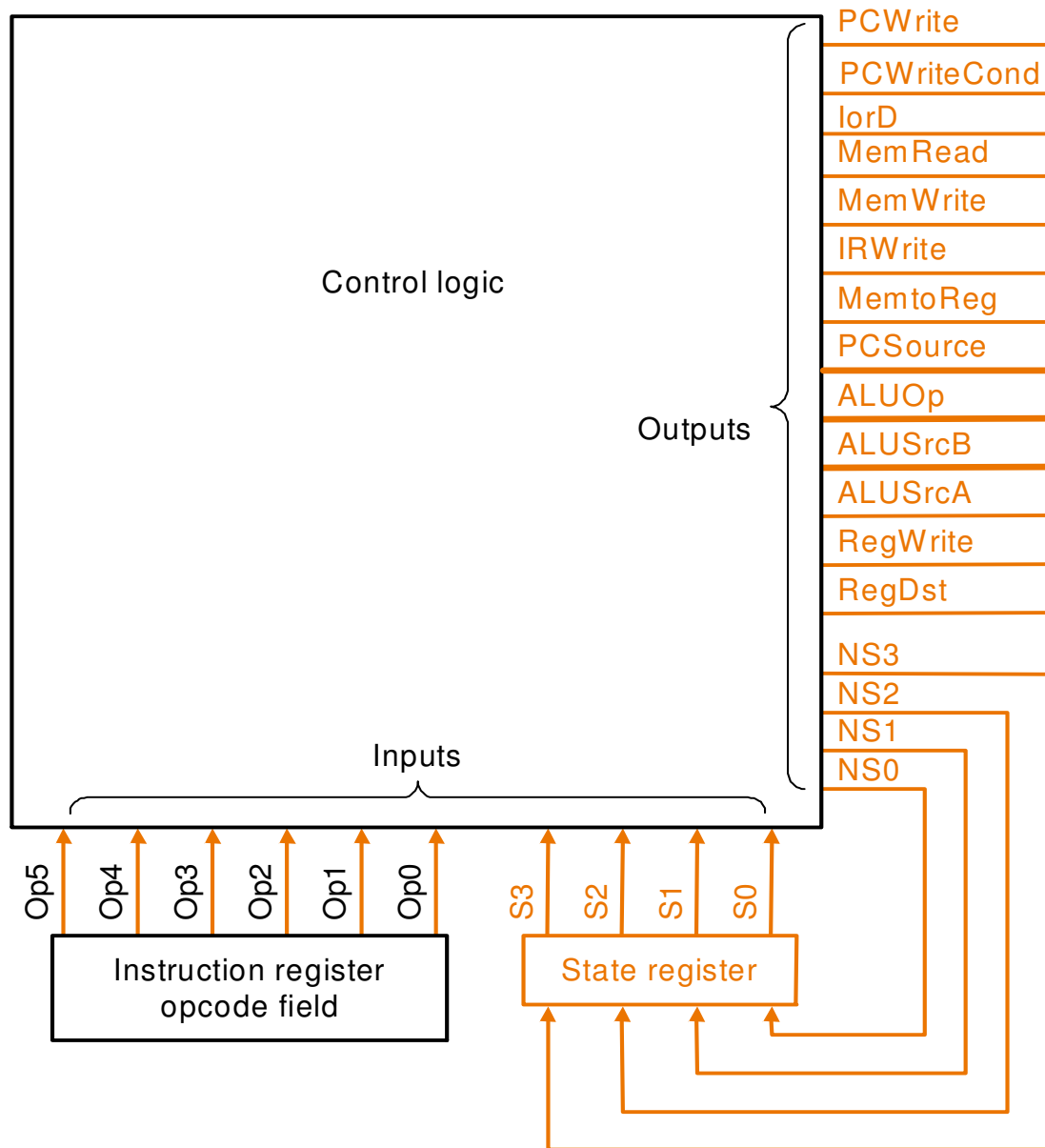
**I dash: don't care**

# Implementation possibilities

## ■ Moore machine: The outputs depend on inputs and state



# Unità di controllo multi-ciclo realizzata come FSM



S3, S2, S1, S0 bit usati per la codifica dello stato presente

NS3, NS2, NS1, NS0 bit usati per la codifica dello stato futuro

La logica di controllo implementa una tabella di verità con le uscite di controllo e lo stato futuro (vedi tabella seguente)

# Equazioni logiche per unità di controllo

<i>PC Write</i>	<i>PC Write Cond</i>	<i>lorD</i>	<i>Mem Read</i>	<i>Mem write</i>	<i>IR Write</i>	<i>Mem to reg</i>	<i>PC Src 1</i>	<i>PC Src 0</i>	<i>ALU Op1</i>	<i>ALU Op0</i>	<i>ALU SrcB 1</i>	<i>ALU SrcB 0</i>	<i>ALU SrcA</i>	<i>Reg Write</i>	<i>Reg Dst</i>
0+9	8	3+5	0+3	5	0	4	9	8	6	8	1+2	0+1	2+6+8	4+7	7

I valori dei segnali di controllo dipendono solo dagli stati correnti

Frammento di tabella della verità  
corrispondente al segnale PCWrite attivo

S3	S2	S1	S0
0	0	0	0
1	0	0	1

<i>Stato Futuro</i>	0	1	2	3	4	5	6	7	8	9
<i>Stato corrente</i>	4+5+7+8+9	0	1	2	3	2	1	6	1	1

# Esempio: tabella per bit 0 dello stato futuro (NS0)

---

NS0=1 negli stati futuri 1, 3, 5, 7, 9 (0001, 0011, 0101, 0111, 1001)

st. fut.1= st. corr. 0      st. fut.3= st. corr. 2 AND lw      st. fut.5=st. corr. 2 AND sw

st. fut.7= st. corr. 6      st. fut.9= st. corr. 1 AND jump

NS0 or di tutti questi stati

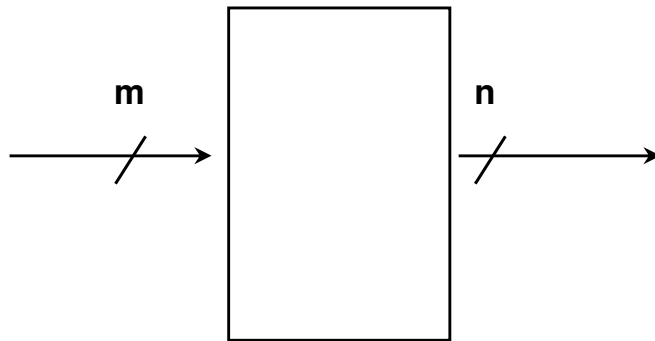
Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
x	x	x	x	x	x	0	0	0	0
1	0	0	0	1	1	0	0	1	0
1	0	1	0	1	1	0	0	1	0
x	x	x	x	x	x	0	1	1	0
0	0	0	0	1	0	0	0	0	1



# ROM Implementation

---

- ROM = "Read Only Memory"
  - values of memory locations are fixed ahead of time
- A ROM can be used to implement a truth table
  - if the address is  $m$ -bits, we can address  $2^m$  entries in the ROM.
  - our outputs are the bits of data that the address points to.



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

$m$  is the "height", and  $n$  is the "width"

# ROM Implementation

---

- How many inputs are there?  
6 bits for opcode, 4 bits for state = 10 address lines  
(i.e.,  $2^{10} = 1024$  different addresses)
- How many outputs are there?  
16 datapath-control outputs, 4 state bits = 20 outputs
- ROM is  $2^{10} \times 20 = 20\text{K}$  bits (and a rather unusual size)
- Rather wasteful, since for lots of the entries, the outputs are the same  
— i.e., opcode is often ignored

# Implementazione tramite ROM1 - uscite controlli

Uscite/Ingressi	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
PCWrite	1	0	0	0	0	0	0	0	0	1
PCWrite Cond	0	0	0	0	0	0	0	0	1	0
IorD	0	0	0	1	0	1	0	0	0	0
MemRD	1	0	0	1	0	0	0	0	0	0
MemWR	0	0	0	0	0	1	0	0	0	0
IRWrite	1	0	0	0	0	0	0	0	0	0
Memtoreg	0	0	0	0	1	0	0	0	0	0
PCSrc1	0	0	0	0	0	0	0	0	0	1
PCSrc0	0	0	0	0	0	0	0	0	1	0
AluOp0	0	0	0	0	0	0	1	0	0	0
AluOp1	0	0	0	0	0	0	0	0	1	0
AlusrcB1	0	1	1	0	0	0	0	0	0	0
AlusrcB0	1	1	0	0	0	0	0	0	0	0
AlusrcA	0	0	1	0	0	0	1	0	1	0
RegWr	0	0	0	0	1	0	0	1	0	0
RegDst	0	0	0	0	0	0	0	1	0	0

ROM da 20 Kbit =  $2^{10} \times 20$ : 10 ingressi (6 opcode + 4 bit di stato corrente), 20 uscite (16 linee di controllo + 4 bit di stato futuro)

Nel nostro caso 10 ingressi implicano 1024 possibili righe: complesso.

IDEA: separare controllo e stato futuro e ottenere 2 ROM più facilmente gestibili

Il codice operativo non conta sul valore dei controlli. Per questo motivo se usassimo una ROM unica avremmo le stesse combinazioni uscite ripetute 64 volte (duplicazione e spreco).

# Implementazione tramite ROM2 - stato futuro

Stato corrente/ Opcode	ADD 000000	Jump 000010	Branch 000100	LW	SW	Altri valori
0000	0001	0001	0001	0001	0001	0001
0001	0110	1001	1000	0010	0010	ILLEG.
0010	xxxx	xxxx	xxxx	0011	0101	ILLEG.
0011	0100	0100	0100	0100	0100	ILLEG.
0100	0000	0000	0000	0000	0000	ILLEG.
0101	0000	0000	0000	0000	0000	ILLEG.
0110	0111	0111	0111	0111	0111	ILLEG.
0111	0000	0000	0000	0000	0000	ILLEG.
1000	0000	0000	0000	0000	0000	ILLEG.
1001	0000	0000	0000	0000	0000	ILLEG.

L'ultima colonna corrisponde a codici operativi non permessi

Con 2 ROM risparmio:

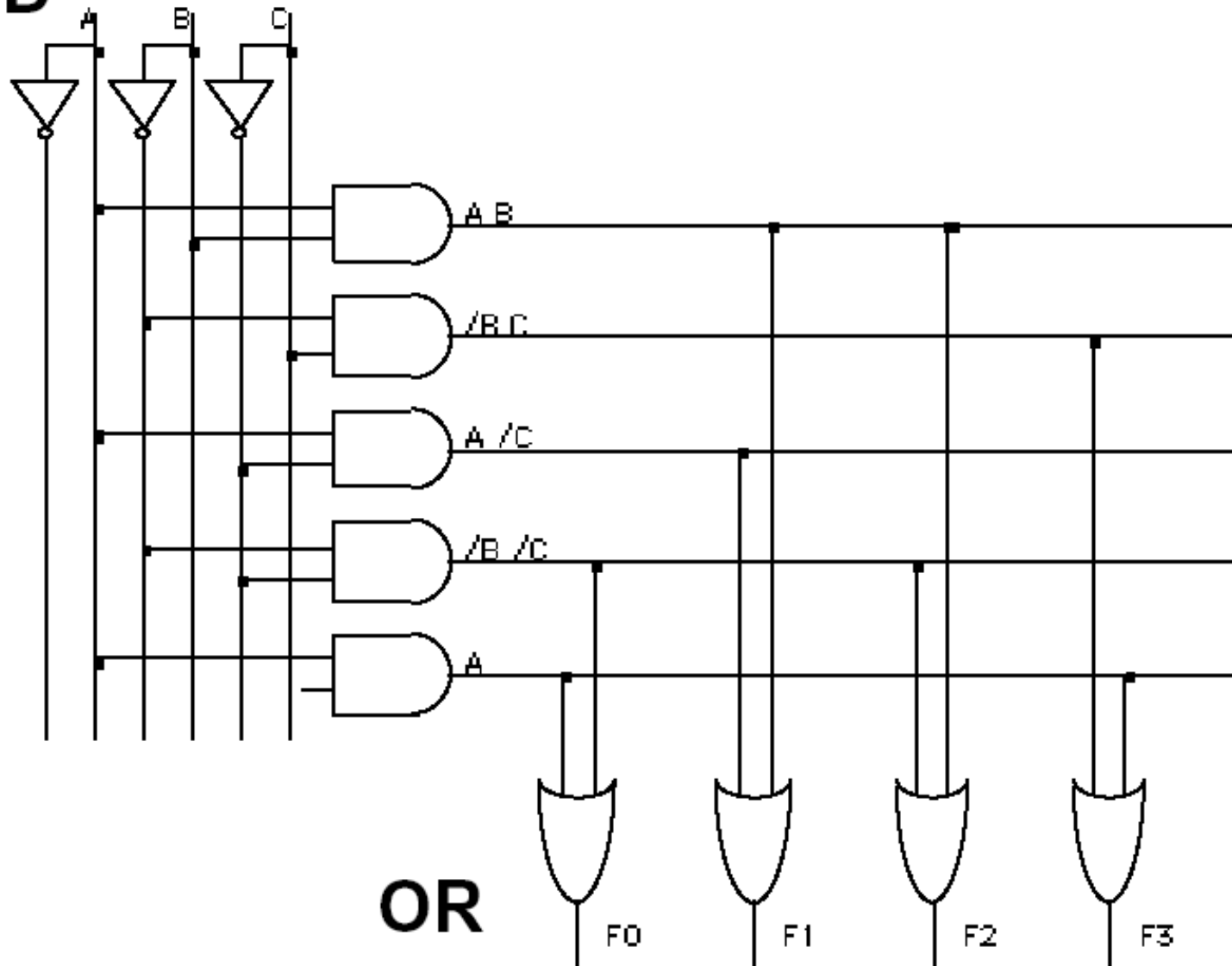
$$2^4 \times 16 + 2^{10} \times 4 = 4.3 \text{ Kbit}$$

anziché 20 Kbit

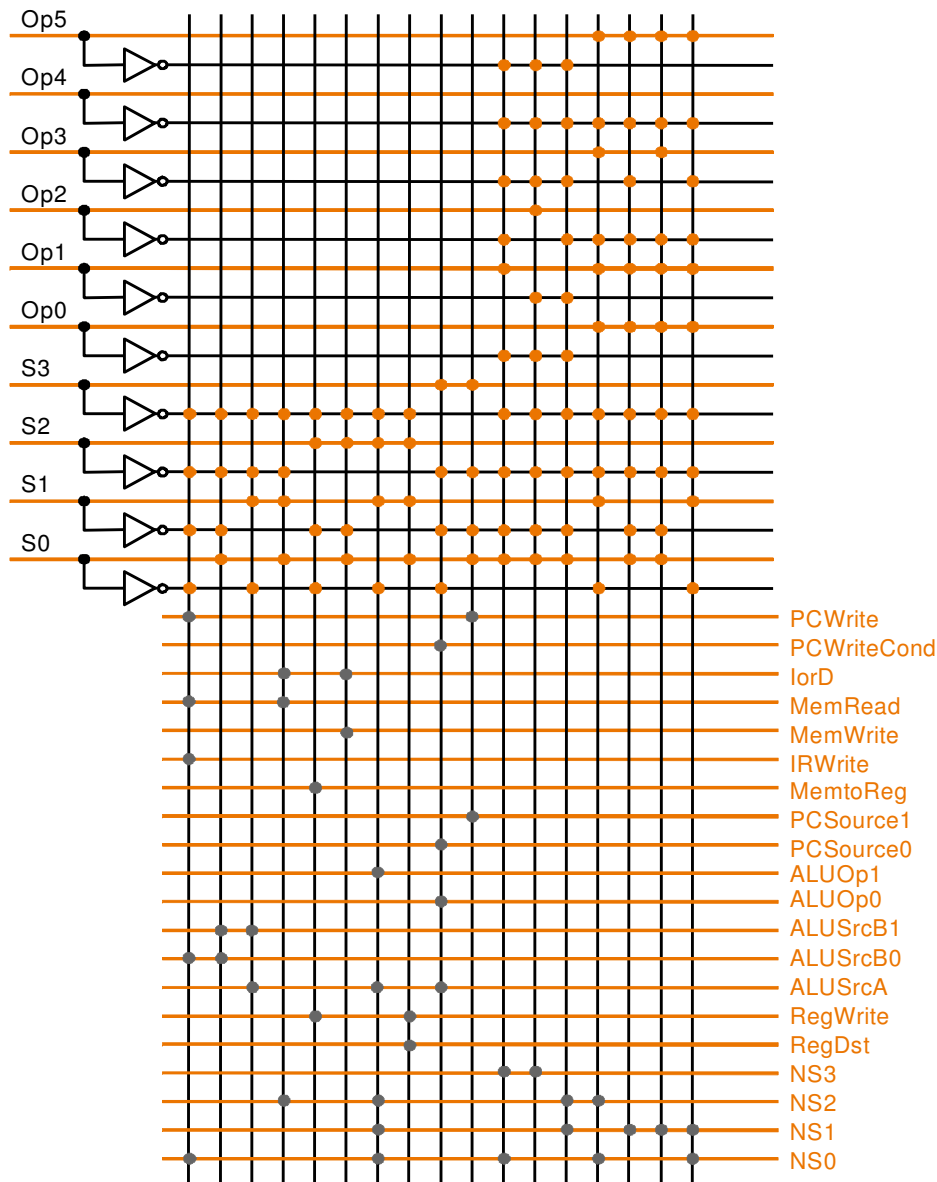
No ottimizzazione: codici don't care  $\Rightarrow$  ancora spreco (< di prima però).

# PLA

## ■ AND



# Implementazione tramite PLA



Meno memoria ma circuito di decodifica indirizzi più complesso.

Ogni uscita è l'or logico (pallino scuro) di più mintermini (colonne verticali).

17 mintermini di cui 10 dipendenti dallo stato corrente ed altri 7 che dipendono da stato corrente e opcode.

Ogni mintermine è l'and dei termini corrispondenti (pallini chiari).

Dimensione = piano AND + piano OR =  $(10 \times 17) + (20 \times 17) = 510$  bit.

Si potrebbe suddividerla in 2 minimizzando ulteriormente: 10 ingressi con 7 mintermini e 4 uscite di stato futuro + 4 ingressi e 10 mintermini a generare le uscite di controllo.

# ROM vs PLA

---

- Break up the table into two parts
  - 4 state bits tell you the 16 outputs,  $2^4 \times 16$  bits of ROM
  - 10 bits tell you the 4 next state bits,  $2^{10} \times 4$  bits of ROM
  - Total: 4.3K bits of ROM
- PLA is much smaller
  - can share product terms
  - only need entries that produce an active output
  - can take into account don't cares
- Size is  $(\text{\#inputs} \times \text{\#product-terms}) + (\text{\#outputs} \times \text{\#product-terms})$   
For this example =  $(10 \times 17) + (20 \times 17) = 460$  PLA cells
- PLA cells usually about the size of a ROM cell (slightly bigger)

- **Limitation of FSM's based on state diagrams is the complexity**
  - ◆ **MIPS: more than 100 instructions**
  - ◆ **Instruction length: 1 - 20 cycles**
  - ◆ **Number of states: enormously growing**
- **One alternative: Microprogramming**
  - ◆ **Implementation of the MIPS instructions as a sequence of simpler instructions**
  - ◆ **Design of the micro instruction format**
  - ◆ **Writing the program**



# Micro instructions

---

## ■ What do we need?

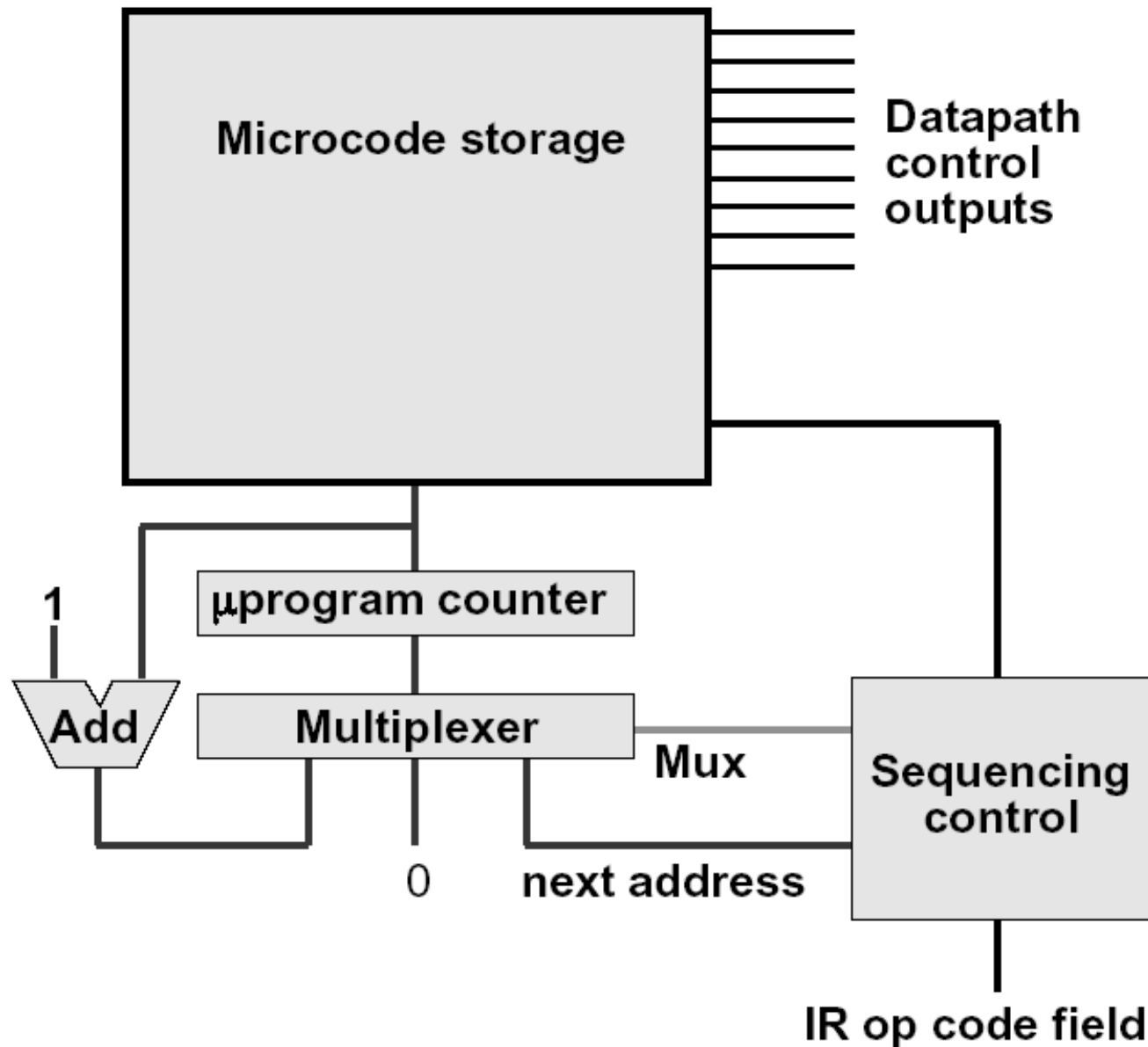
- ◆ ALU control signals
- ◆ Program counter control signals
- ◆ Complete data path

## ■ What do we do?

- ◆ Analyse what is happening at each step
- ◆ Write it down in a column oriented way
- ◆ Translate it to binary
- ◆ Store it in an appropriate form

# A simple implementation

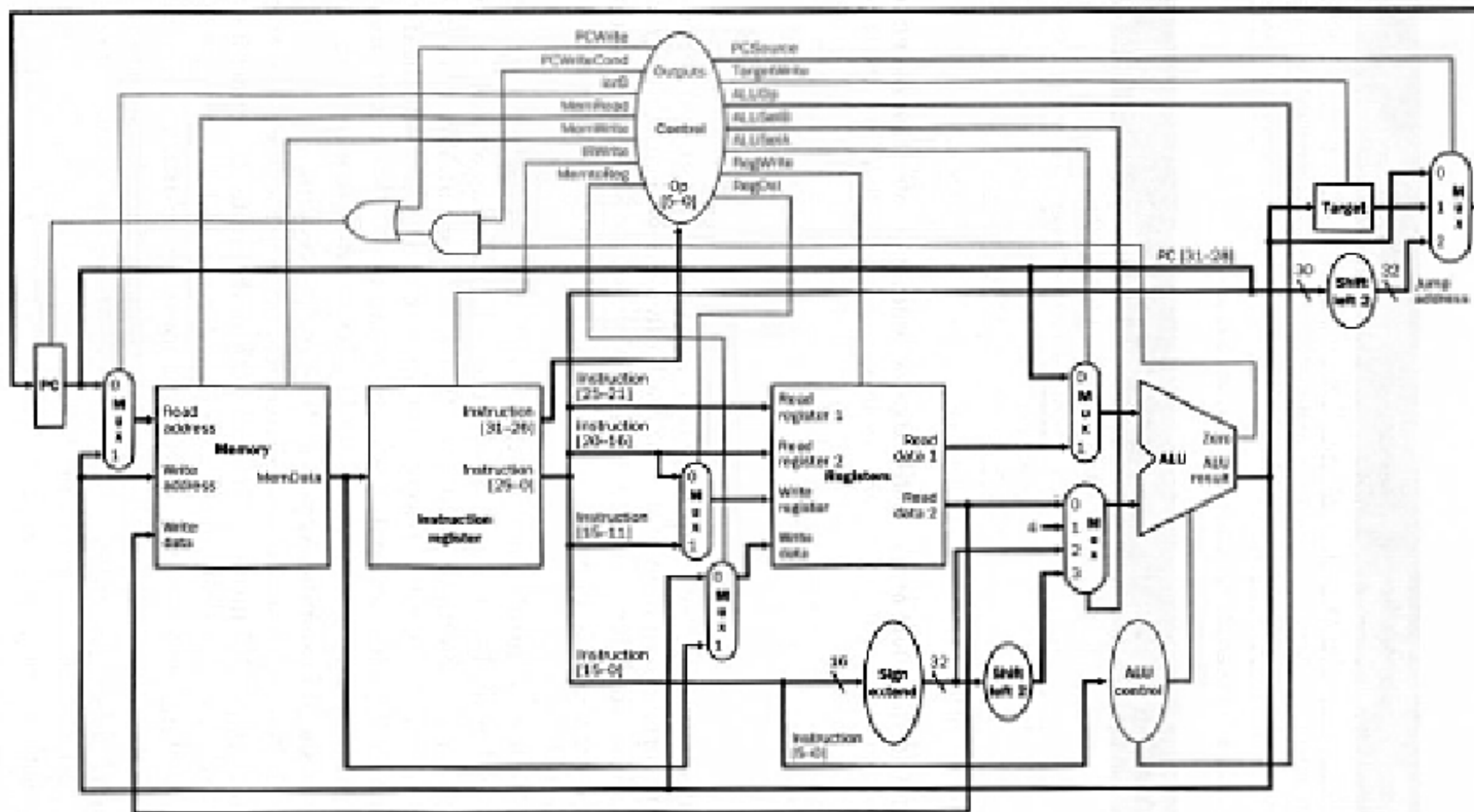
---



- **Microcode storage (ROM or PLA)**
  - ◆ Datapath control outputs
  - ◆ Sequencing control out
- **Output decoder (optional)**
- **Microprogram counter / Mux**
  - ◆ Reset
  - ◆ Increment
  - ◆ New address
- **Sequencing control**
  - ◆ Mux control
  - ◆ Next address calculation

# Data path

## ■ Data path and control



# Micro instruction format

---

- **Sort the signals according to their function**
- **Signals that are never used at the same time can be shared**
- **These signals have to be set !**

Field name	Function of field
ALU control	Specify the operation being done by the ALU during this clock; the result is always written in ALUOut.
SRC1	Specify the source for the first ALU operand.
SRC2	Specify the source for the second ALU operand.
Register control	Specify read or write for the register file, and the source of the value for a write.
Memory	Specify read or write, and the source for the memory. For a read, specify the destination register.
PCWrite control	Specify the writing of the PC.
Sequencing	Specify how to choose the next microinstruction to be executed.

# Microprogram

---

- **Physical implementation: ROM or PLA**
- **Each micro instruction has an address**
- **Sequentially ordered**
- **Each sequence step is one cycle**
- **Selection of next instruction**
  - ◆ **Address increment (sequencing field = seq)**
  - ◆ **Fetch: begins the fetch of the next MIPS micro instruction (sequencing field = fetch)**
  - ◆ **Dispatch: jump to the next micro instruction the number i indicates the location in the dispatch table.**

# Control signals

Field Name	Function	Effect
ALU Control	Add	Addition operation
	Subt	Subtraction operation
	Func code	Function depends on the contents of the code field in the IR
SRC 1	PC	Use the PC as the first ALU input.
	A	Register A is the first ALU input.
SRC 2	B	Register B is the second ALU input.
	4	The second ALU operand is the constant 4
	Extend	The second ALU operand is the sign-extended bits of IR
	Extshft	The second ALU operand is the sign-extended & shifted bits of IR
	Read	Read two registers using rd field of the IR and place the contents in A & B.
Register control	Write ALU	Write Register with data from ALU
	Write MDR	Write Register with data from the Memory Data Register
Memory	Read PC	Read memory using the PC address
	Read ALU	Read memory using the ALU output as address
	Write ALU	Write memory using the ALU output as address
PCWrite control	ALU	Write to the PC the ALU output
	Target cond	Write to the PC the contents of target IF Zero = active
	jump address	Write to the PC the jump address from instruction
Sequencing	Seq	Choose the next microinstruction
	Fetch	Go to the first microinstruction and begin a new instruction

# Creating the microprogram

---

## ■ Start of each instruction

- ◆ Fetch instruction
- ◆ Calculate next PC
- ◆ Decode instruction
- ◆ Calculate branch PC

Label	ALU Con	SRC 1	SRC 2	RegCntl	Mem	PC Write	Sequence
Fetch	Add	PC	4		ReadPC	ALU	Seq
	Add	PC	Extshft	read			Dispatch 1

## ◆ Blank field

- I Multiplexor: we don't care
- I Control signal for register access or function: it is set to 0



# The start sequence

---

Label	ALU Con	SRC 1	SRC 2	RegCntl	Mem	PC Write	Sequence
Fetch	Add	PC	4		ReadPC	ALU	Seq
	Add	PC	Extshft	read			Dispatch 1

## ◆ First instruction

Fields	Effect
ALU control, SRC 1, SRC 2	Compute PC + 4
Memory	Fetch the instruction to IR
PCWrite control	Output of ALU is loaded into PC
Sequencing	Go to the next microinstruction

## ◆ Second instruction

Fields	Effect
ALU control, SRC 1, SRC 2	Store PC + sign extension (IR [15-0])
	<< 2 into ALUOut
Register control	Moves the data from the register file to A & B
Sequencing	Use dispatch table 1 for next address

# Memory access

---

## ■ Microinstructions

Label	ALU Con	SRC 1	SRC 2	RegCntl	Mem	PC Write	Sequence
Mem 1	Add	A	Exend				Dispatch 2
LW2					Read ALU		seq
				Write MDR			Fetch
SW2					Write ALU		Fetch

## ■ Common part

Fields	Effect
ALU control, SRC 1, SRC 2	Memory address = rs + sign extend (IR[15-0])
	result available at the ALU output
Sequencing	Use 2nd dispatch tabel for continuation
	at SW 2 or LW 2

# Memory access

---

## ■ Load word specific, 1

Fields	Effect
Memory	Read memory and write to MDR
Sequencing	Go to the next microinstruction

## ■ Load word specific, 2

Fields	Effect
Register control	Write the contents of the MDR to register rt
Sequencing	Go to the label Fetch

## ■ Store word specific

Fields	Effect
Memory	Write memory with ALUOut as address
Sequencing	Go to the label Fetch

# R-format

---

## ■ Microinstruction

Label	ALU Con	SRC 1	SRC 2	RegCntl	Mem	PC Write	Sequence
Rfor1	FuncCode	A	B				seq
				Write ALU			Fetch

## ■ Operation

Fields	Effect
ALU control, SRC 1, SRC 2	ALU processes A & B according to function
Sequencing	Go to the next microinstruction

## ■ Finishing the operation

Fields	Effect
register control	Write the ALUOut int the register file
Sequencing	Go to the label Fetch

# Branch & Jump

---

## ■ Branch

Label	ALU Con	SRC 1	SRC 2	RegCntl	Mem	PC Write	Sequence
BEQ1	Subtr	A	B			AluOutC	Fetch

Fields	Effect
ALU control, SRC 1, SRC 2	ALU subtracts operands for Zero output
PC Write control	Load PC with ALUOut if Zero = 1
Sequencing	Go to the label Fetch

## ■ Jump

Label	ALU Con	SRC 1	SRC 2	RegCntl	Mem	PC Write	Sequence
Jump1						JpAdr	Fetch

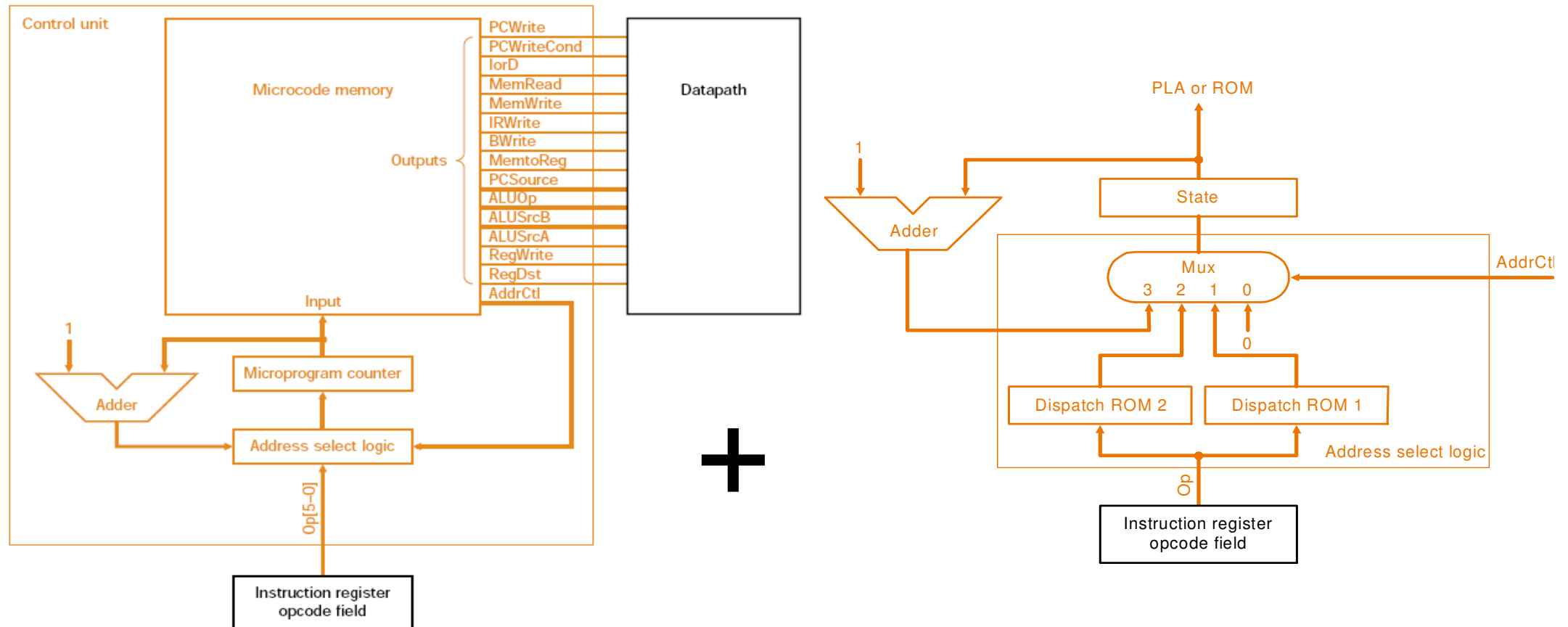
Fields	Effect
PC Write control	PC loaded with jump target address

# Complete microprogram

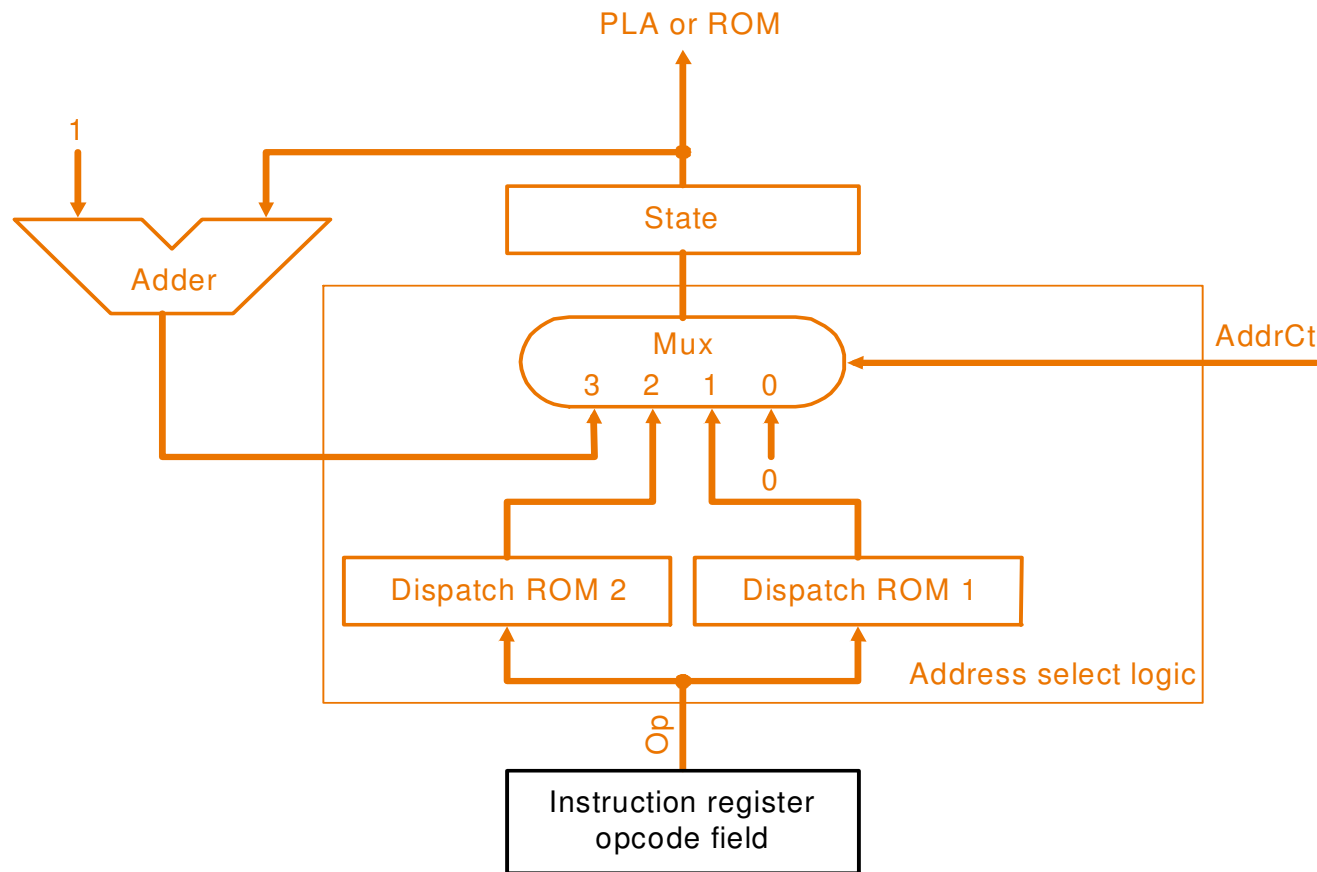
---

Label	ALU Con	SRC 1	SRC 2	RegCntl	Mem	PC Write	Sequence
Fetch	Add	PC	4		ReadPC	ALU	Seq
	Add	PC	Extshft	read			Dispatch 1
Mem 1	Add	A	Exend				Dispatch 2
LW2					Read ALU		seq
				Write Mem			Fetch
SW2					Write ALU		Fetch
Rfor1	FuncCode	A	B				seq
				Write ALU			Fetch
BEQ1	Subtr	A	B			AluOutC	Fetch
Jump1						JpAdr	Fetch

# Architettura unità di controllo per microistruzioni (1)



# Architettura unità di controllo per microistruzioni (2)



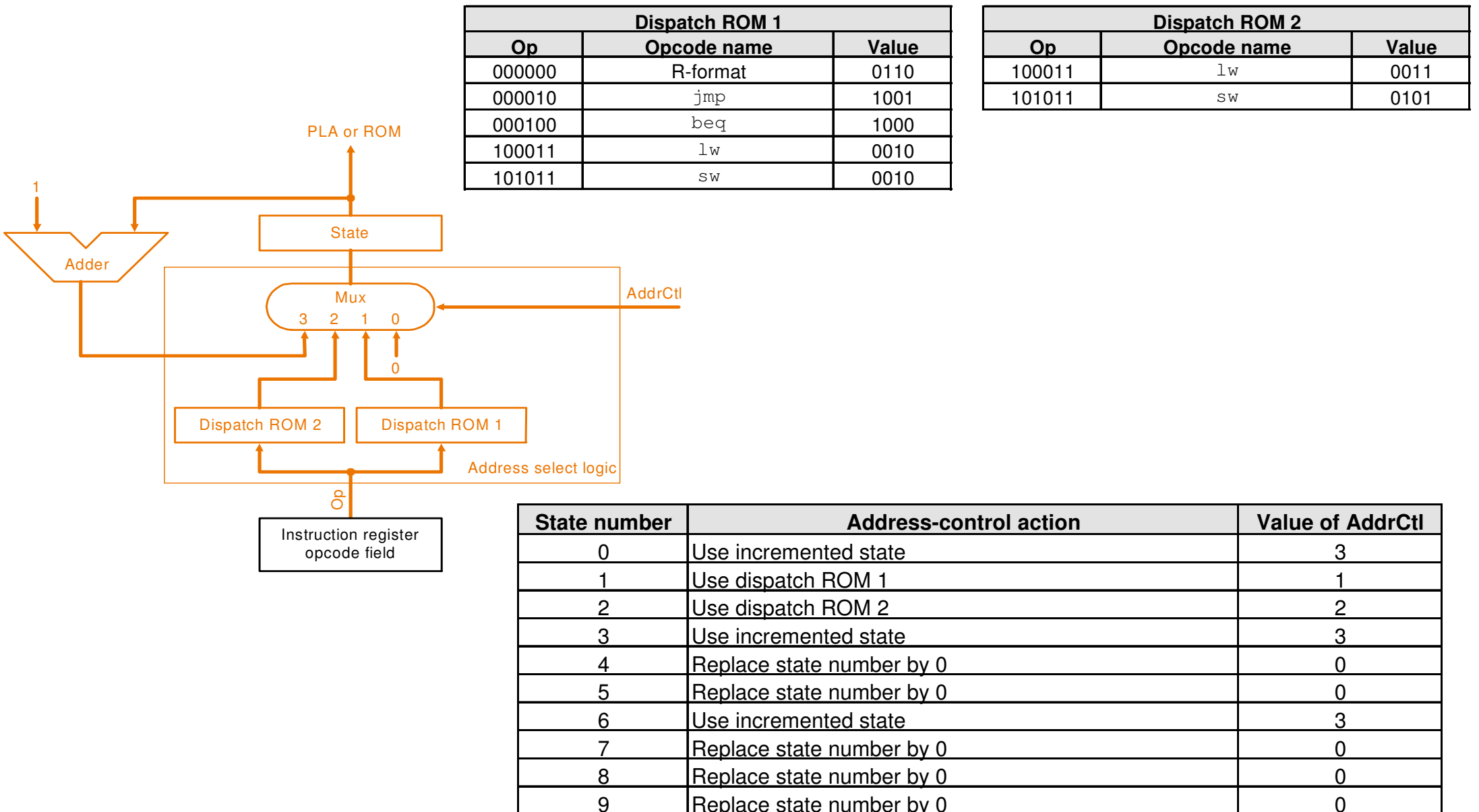
## GESTIONE SALT

Il salto dispatch verrà gestito da una ROM/PLA esterna indirizzata da linee apposite della logica di controllo. Il salto allo stato 0 codificato esplicitamente.

La selezione tra le diverse possibilità viene fatta dalle linee AddrCtl della logica di controllo.



# Architettura unità di controllo per microistruzioni (3)



# Untreated situations

---

## ■ Interrupts

- ◆ I/O device request

## ■ Exception handling

- ◆ Invocation of operating system from user
- ◆ Arithmetic overflow
- ◆ Undefined instruction
- ◆ Hardware malfunction

## ■ Error handling

- ◆ Recovery
- ◆ Abort operation
- ◆ Abort process
- ◆ reboot

# Exception detection

---

## ■ Undefined instruction

- ◆ Check the value of the op code field
- ◆ On all values that do not belong to the valid set generate an interrupt

## ■ Arithmetic overflow

- ◆ Is detected and generated by the ALU

## ■ Action

- ◆ Save the PC of the current instruction (PC-4) in the exception program counter (EPC)
- ◆ Continue with OS with error handling code
- ◆ Return to the original code at the next instruction for restart

# Exception handling

---

## ■ Version 1:

**Use of a status register (cause register)**

- ◆ Single entry point
- ◆ OS analyses the status register

## ■ Version 2:

**Vectored interrupt**

- ◆ For each interrupt cause a separate vector is stored
- ◆ Each interrupt vector targets at the appropriate piece of code

## ■ Registers

### ◆ Cause register, 32 bits

- | Individual bits are used for coding the cause, let's assume
- | Undefined instruction: bit 0 = 0
- | Arithmetic overflow: bit 0 = 1

### ◆ EPC, 32 bits, loadable from ALU output

## ■ Control signals

### ◆ CauseWrite

### ◆ EPCWrite

### ◆ 1 bit signals IntCause

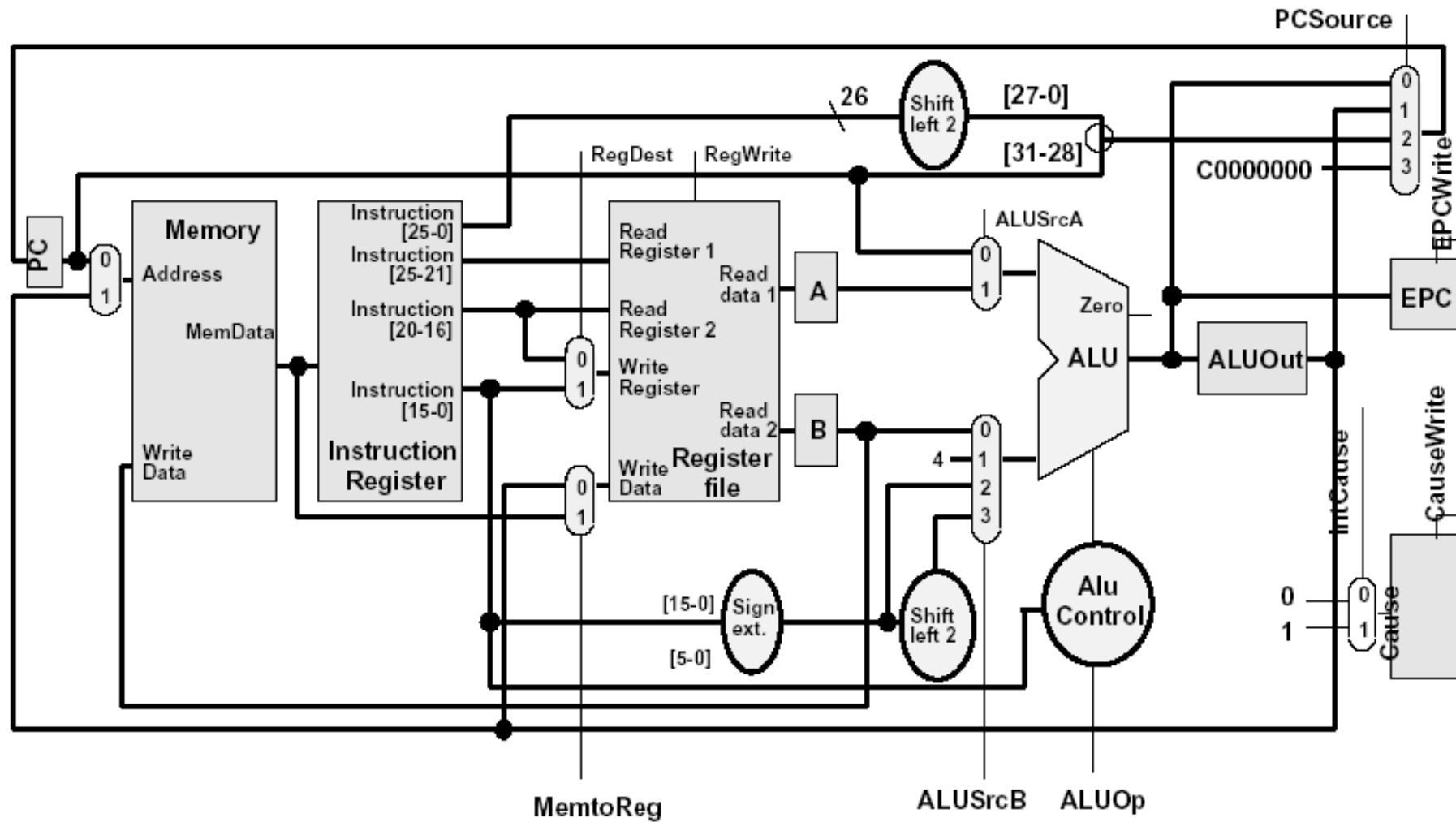
### ◆ Exception address, C0000000<sub>hex</sub>

## ■ PC Mux extension

- ◆ ALU
- ◆ Branch
- ◆ Jump
- ◆ Exception

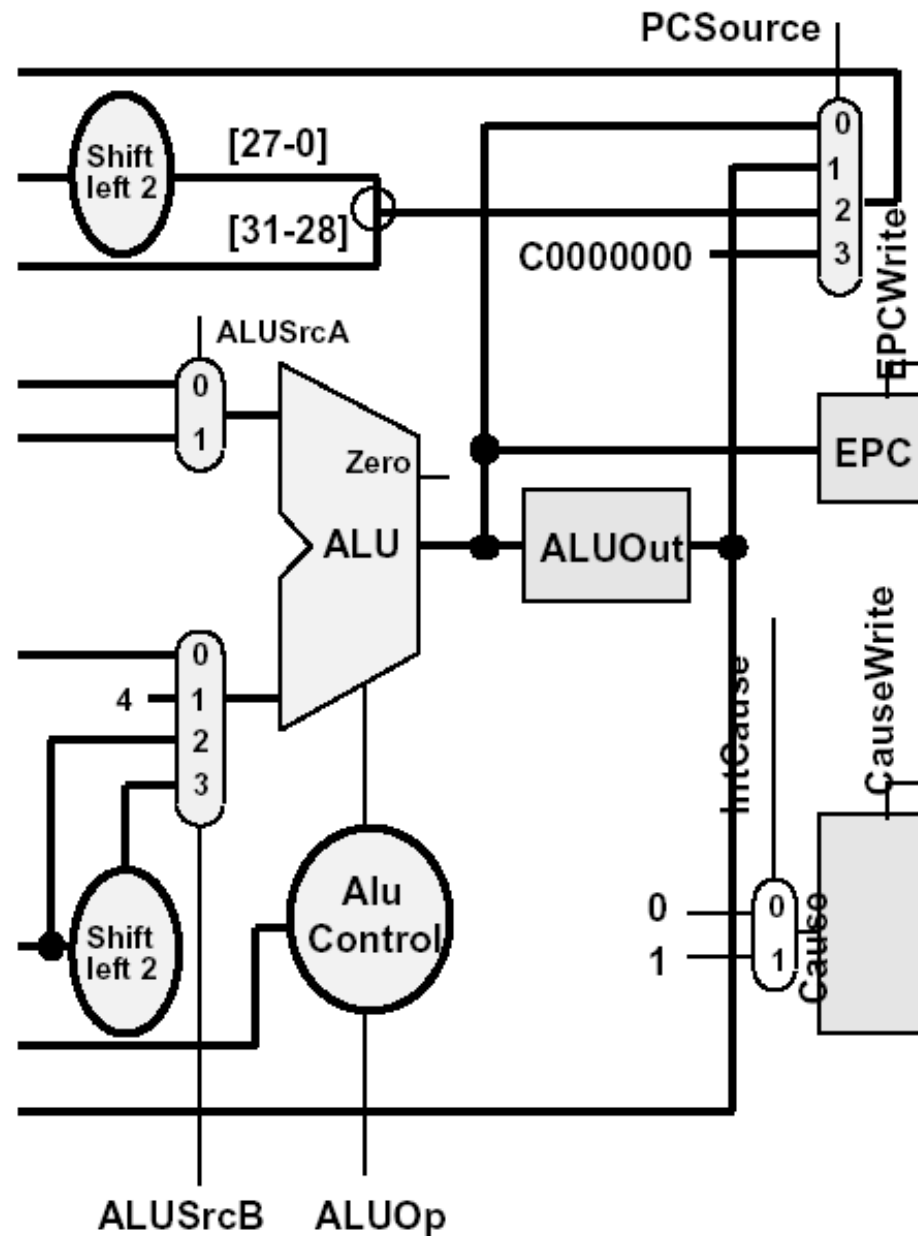
## ■ PC decrement by 4 to get address of current instruction

# Extended architecture



# Extended architecture

- Cause register
- Cause write
- EPC
- EPCWrite
- PCSource extension





# Exception handling

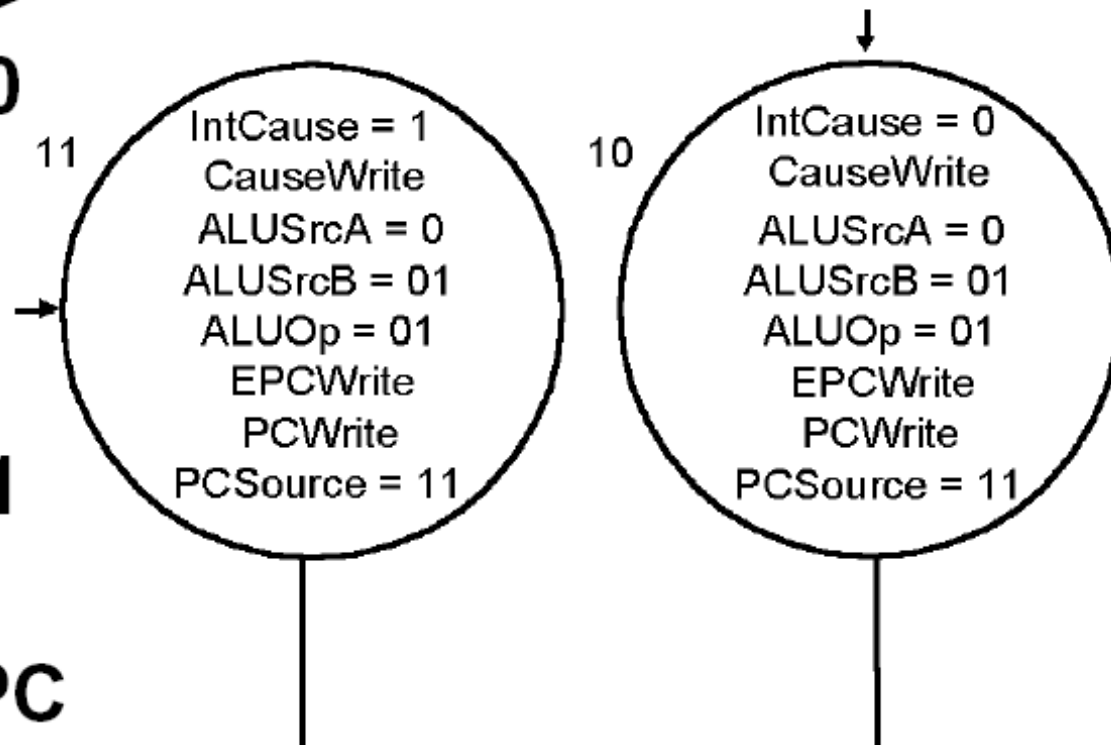
---

■ Undefined instruction >  
IntCause = 0  
(State 10)

■ Arithmetic overflow >  
IntCause = 1  
(State 11)

■ Load the EPC

■ Load PC with  
interrupt  
register



# State machine with exception

