Computer Structure

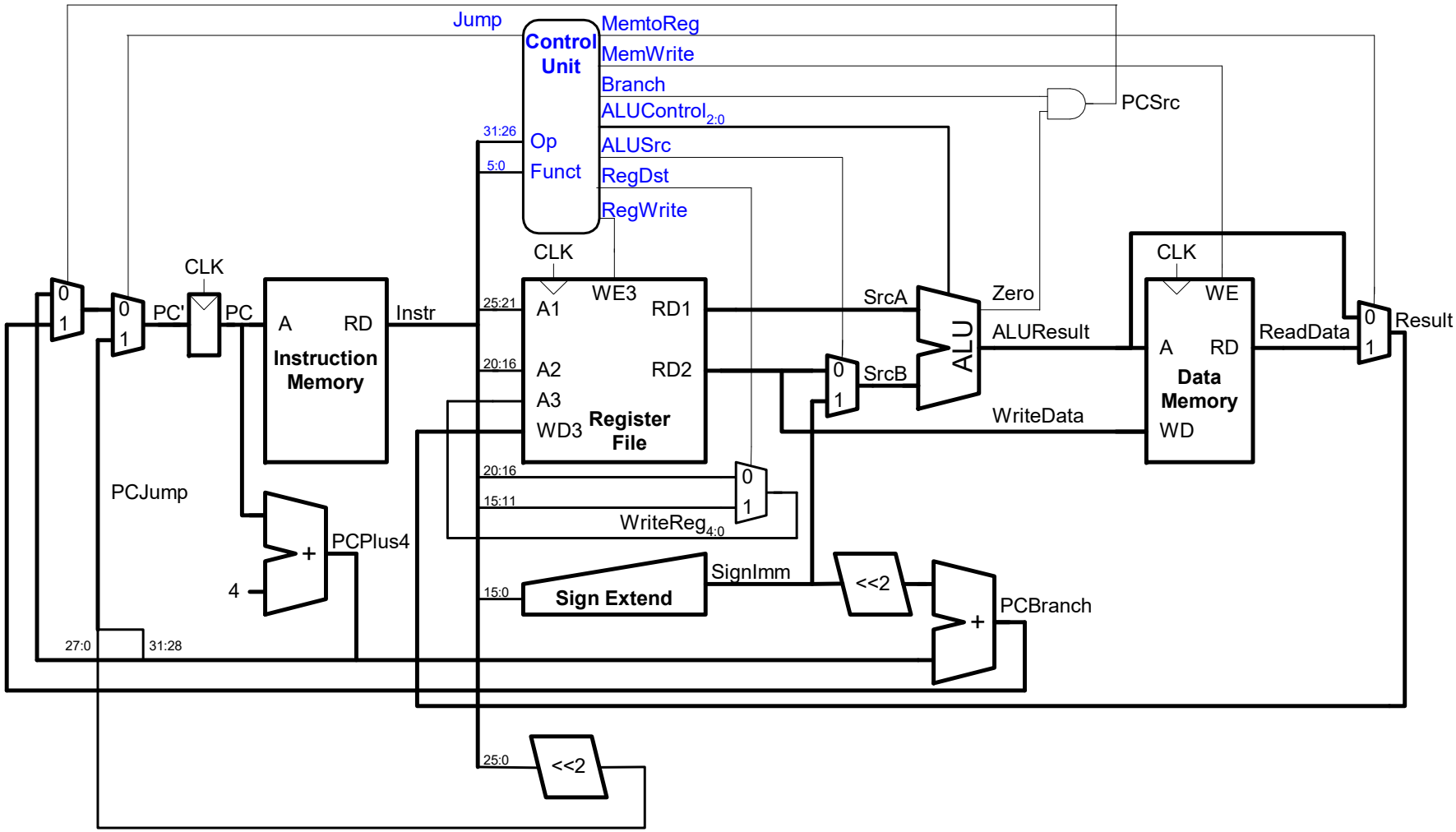
Unit 5. Processor III: Design and control of the datapath. Multicycle architecture

Escuela Politécnica Superior - UAM

Outline

- **Summary single-cycle MIPS**
- Multicycle datapath
- Multicycle control path
- Adding more instructions

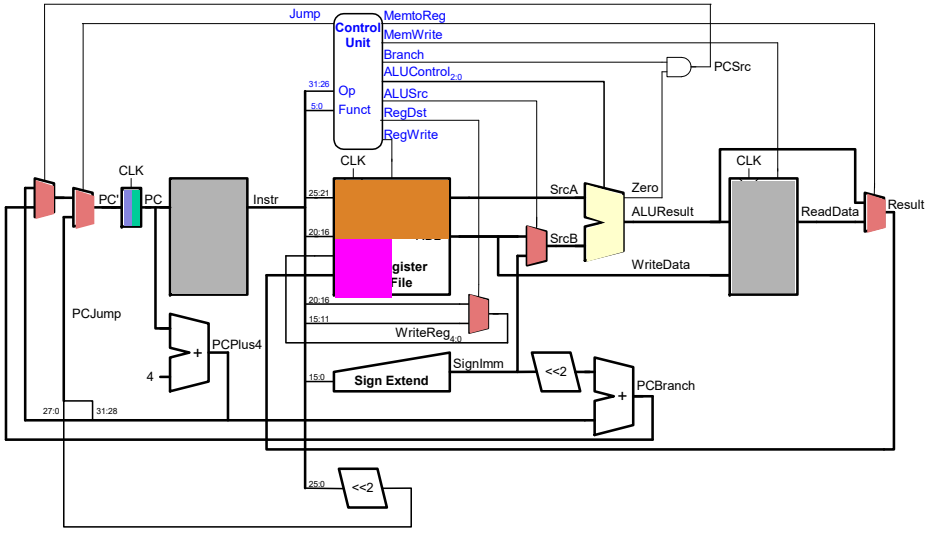
Single-cycle MIPS



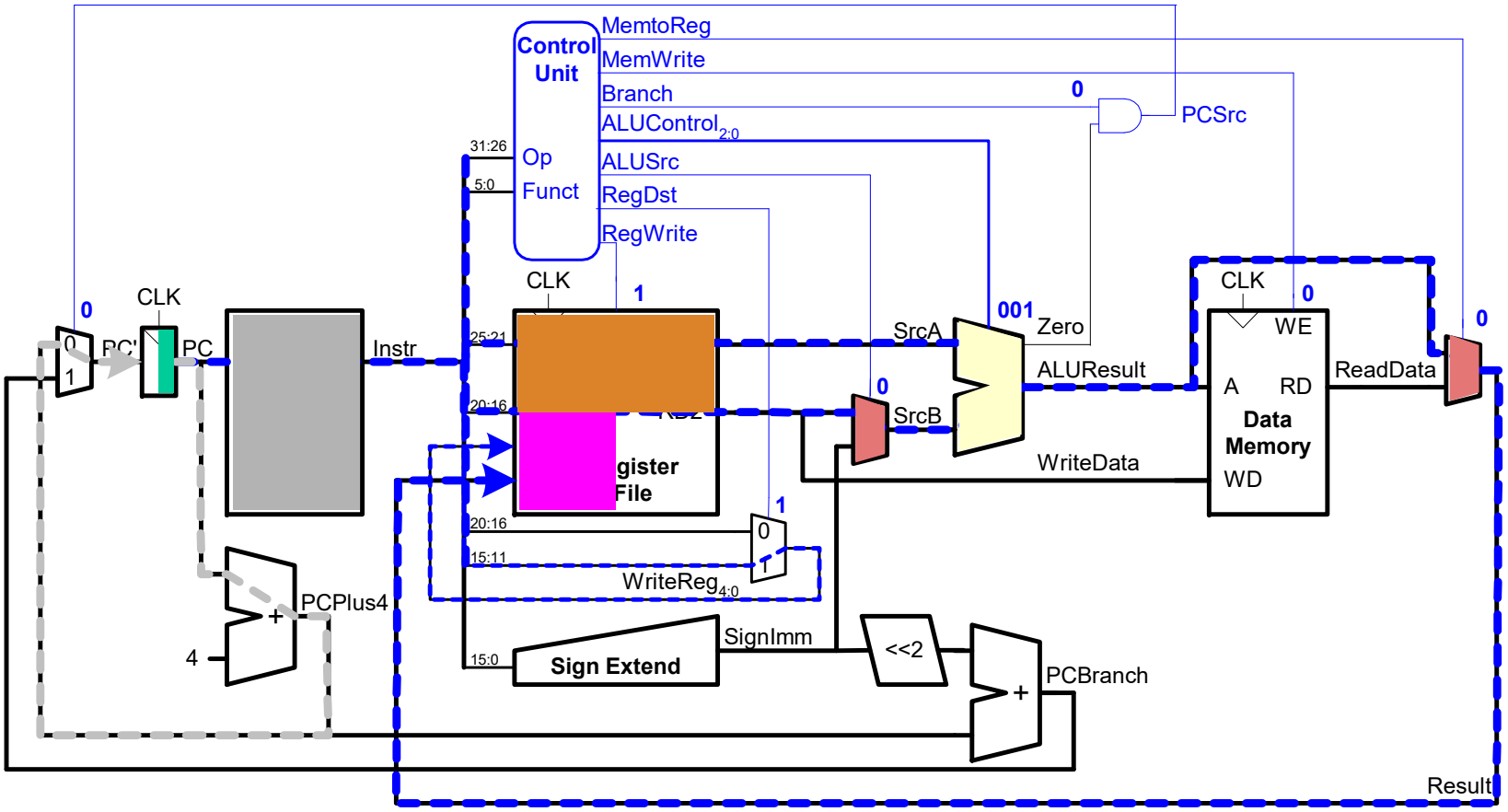
Single-cycle clock cycle

- All instructions use the same clock cycle, which must be long enough for the slowest instruction.

Element	Parameter	Delay (ps)
$T_{CLK \rightarrow Q}$ Register	t_{pcq_PC}	30
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Register file setup	$t_{RFsetup}$	20
Register setup	t_{setup}	20



Single-cycle clock cycle
Example: path for ***add***



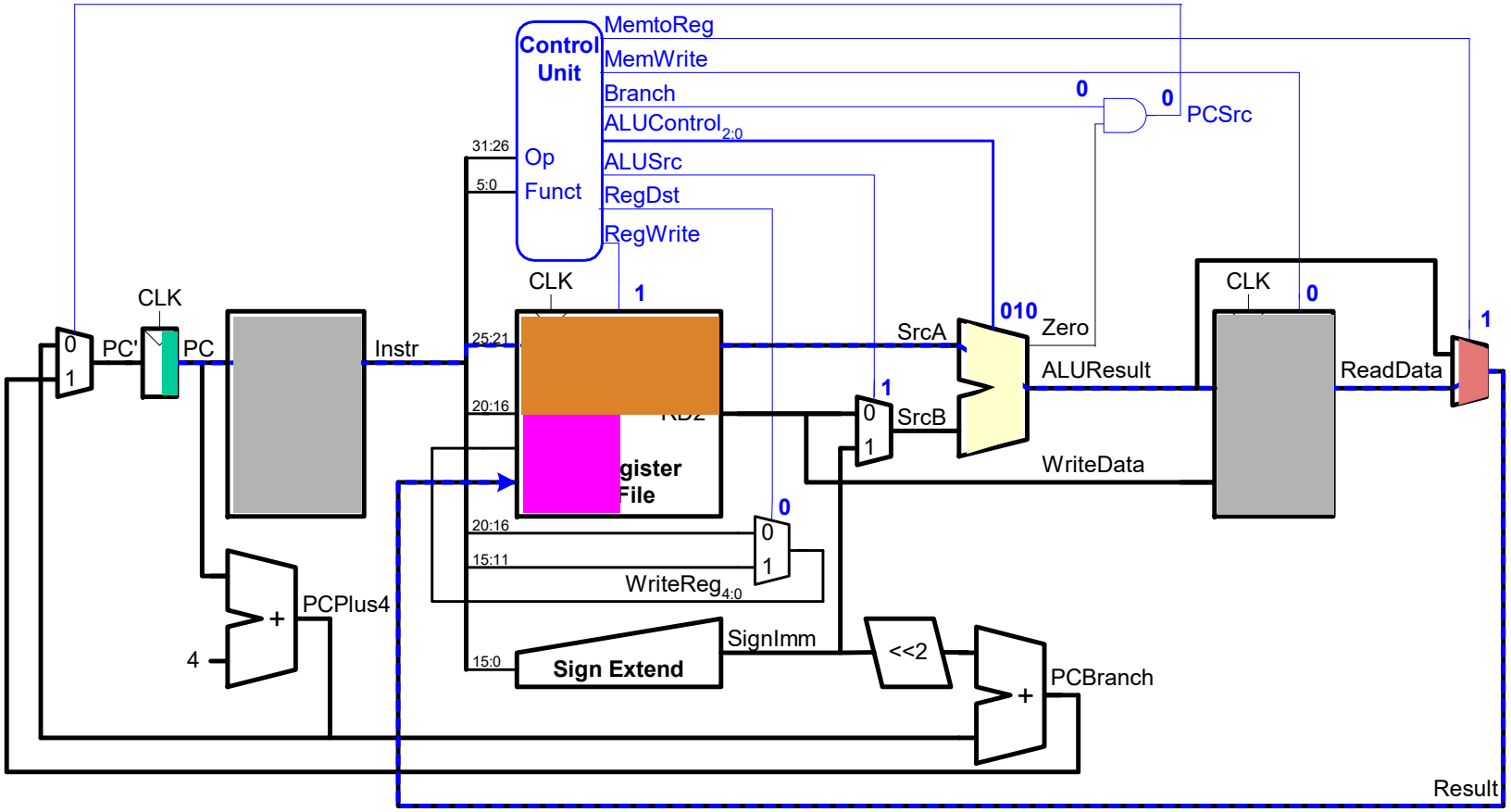
Total delay in add:

$T_c = t_{pcq\_PC} + t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{mux} + t_{RFsetup} =$

$= [30 + 250 + 150 + 25 + 200 + 25 + 20]$ ps = 700 ps (1,4 GHz)

"The writing in the register file is done the next clock edge"

Single-cycle clock cycle
Example: path for *lw*



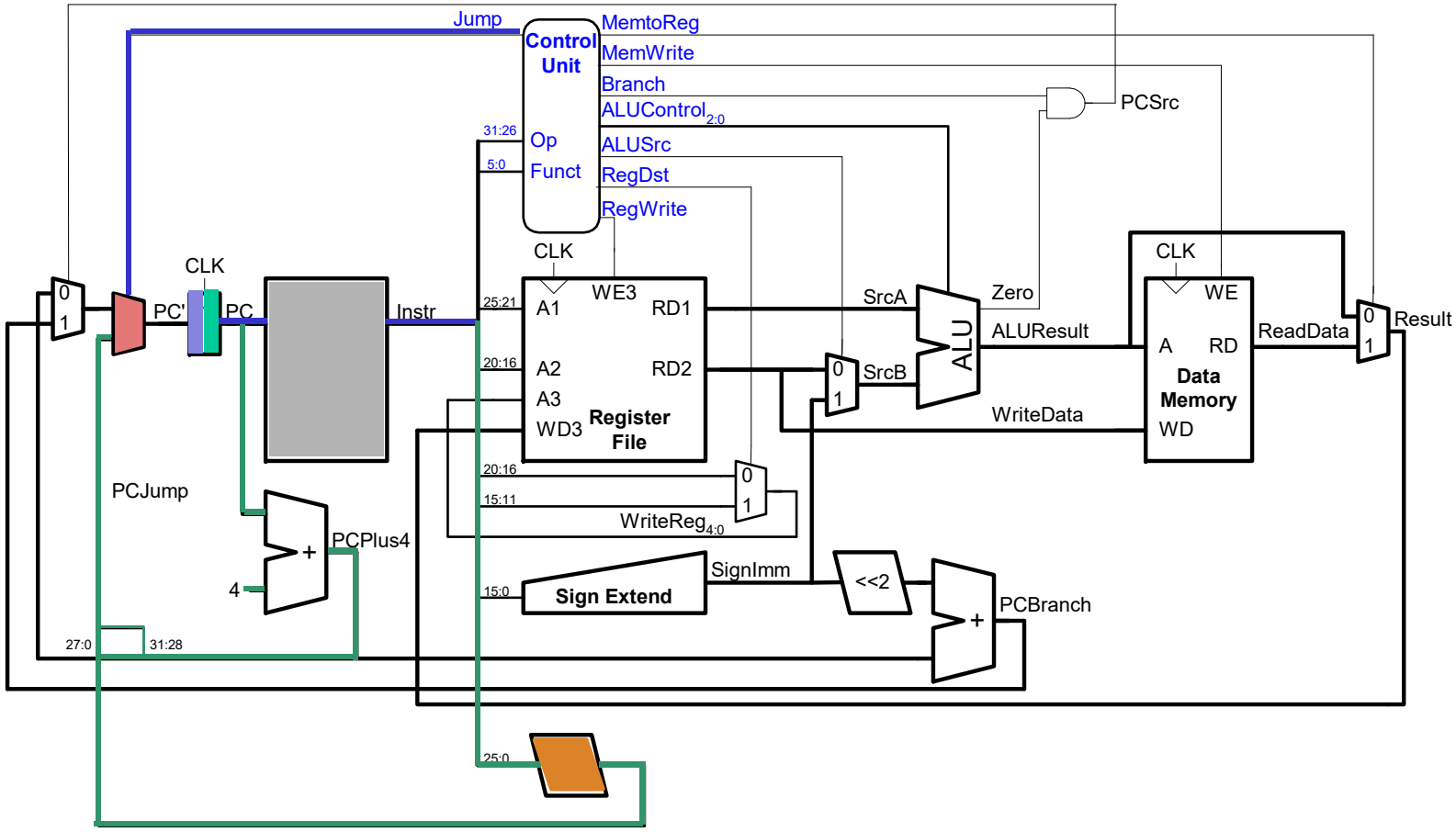
Total delay in lw:

$T_c = t_{pcq\_PC} + t_{mem} + t_{RFread} + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup} =$

$= [30 + 250 + 150 + 200 + 250 + 25 + 20]$ ps = 925 ps (1,1 GHz)

"The writing in the register file is done the next clock edge"

Single-cycle clock cycle
Example: path for *jump (j)*



Total delay in *jump* (j):

$T_c = t_{pcq\_PC} + t_{mem} + t_{<<} + t_{mux} + t_{PCsetup} =$

$= [30 + 250 + 150 + 25 + 20]$ ps = 475 ps (~2,1 GHz)

MIPS single-cycle vs multicycle

	t_{pcq_PC}	t_{mem}	$T_{<<}$	t_{RFread}	t_{mux}	t_{ALU}	t_{mem}	t_{mux}	$t_{RFsetup}/t_{PCsetup}$	T_{total} (ps)
add	30	250	--	150	25	200	--	25	20/20	700 ps
lw	30	250	--	150	--	200	250	25	20/20	**925 ps**
j	30	250	150	--	--	--	--	25	20	475 ps

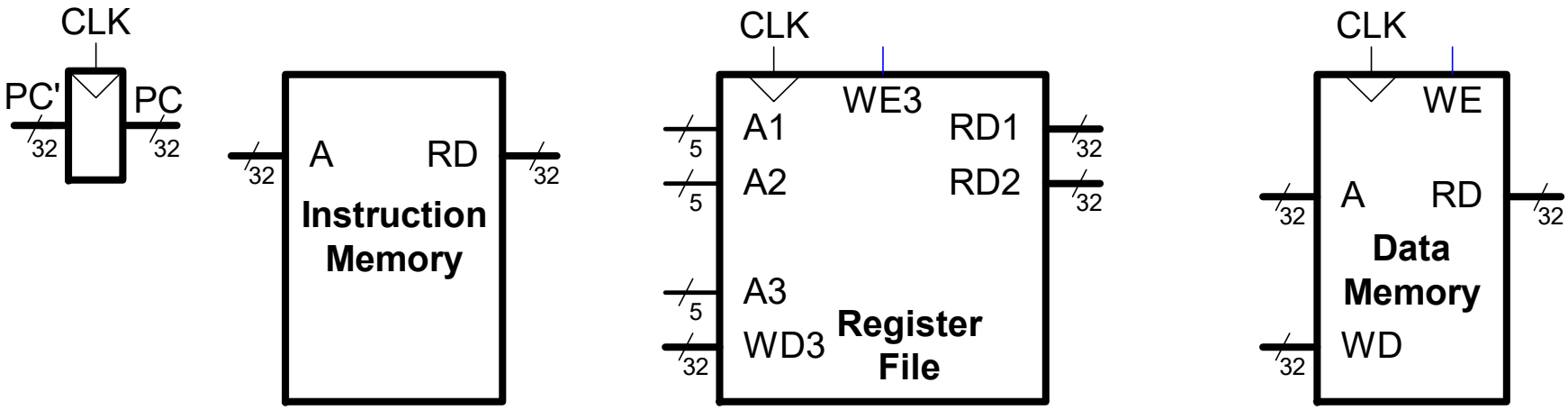
- Single-cycle microarchitecture:
  - + Simple
  - - Cycle time limited by longest instruction (**lw**)
  - - 3 adders/ALUs and 2 memories (instructions and data)
- Muticycle microarchitecture:
  - + Higher clock speed
  - + Simpler instructions run faster (less cycles)
  - + Reuse expensive hardware on multiple cycles
  - - Sequencing overhead paid many times

Outline

- Summary single-cycle MIPS
- **Multicycle datapath**
- Multicycle control path
- Adding more instructions

Recap single-cycle Architectural State

- These elements are enough for determining the state of a processor:
  - PC
  - Register file (32 registers)
  - Memory (text and data)
- First elements to take into account:



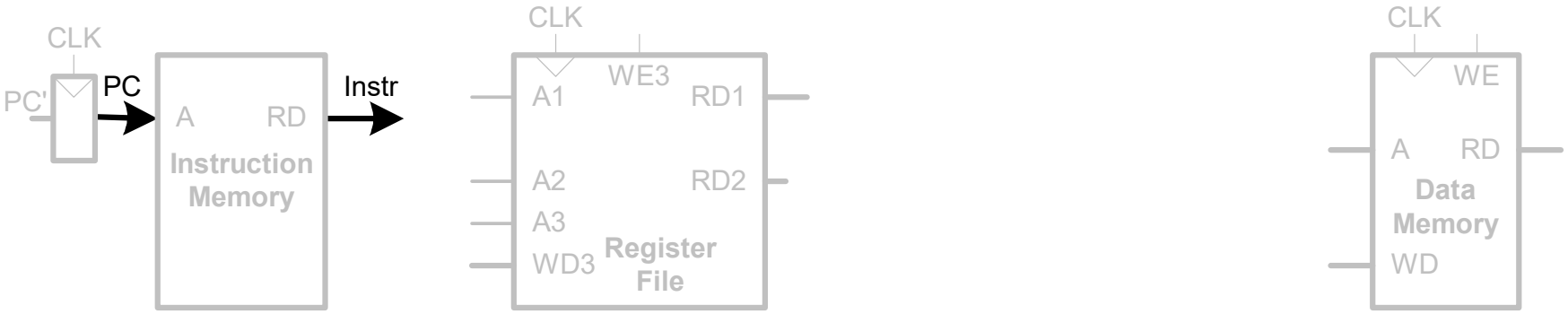
Recap single-cycle Single-Cycle Datapath: lw fetch

Let's consider the instruction:

(**0x8C112000**) **`lw $s1,0x2000($0)`**

and the steps to perform it:

- **STEP 1:** fetch instruction (*fetch*)

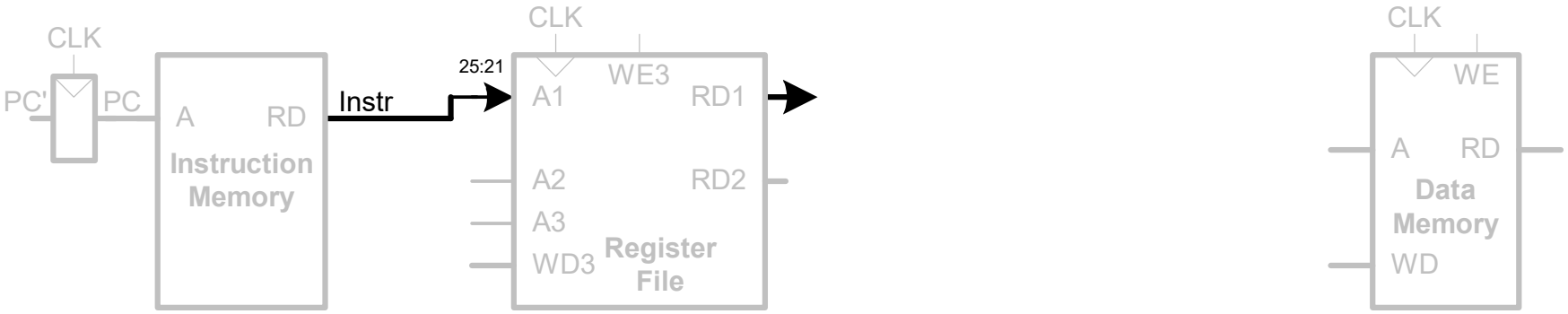


Instr <= "100011 00000 10001 0010000000000000"

op: Instr [31:26] =	"100011"	=>	**`lw`**
rs: Instr [25:21] =	"00000"	=>	**`$0`**
rt: Instr [20:16] =	"10001"	=>	**`$s1`**
imm: Instr [15:0] =	"0010000000000000"	=>	**`0x2000`**

Recap single-cycle Single-Cycle Datapath: lw register read

- **STEP 2:** read source operands from register file

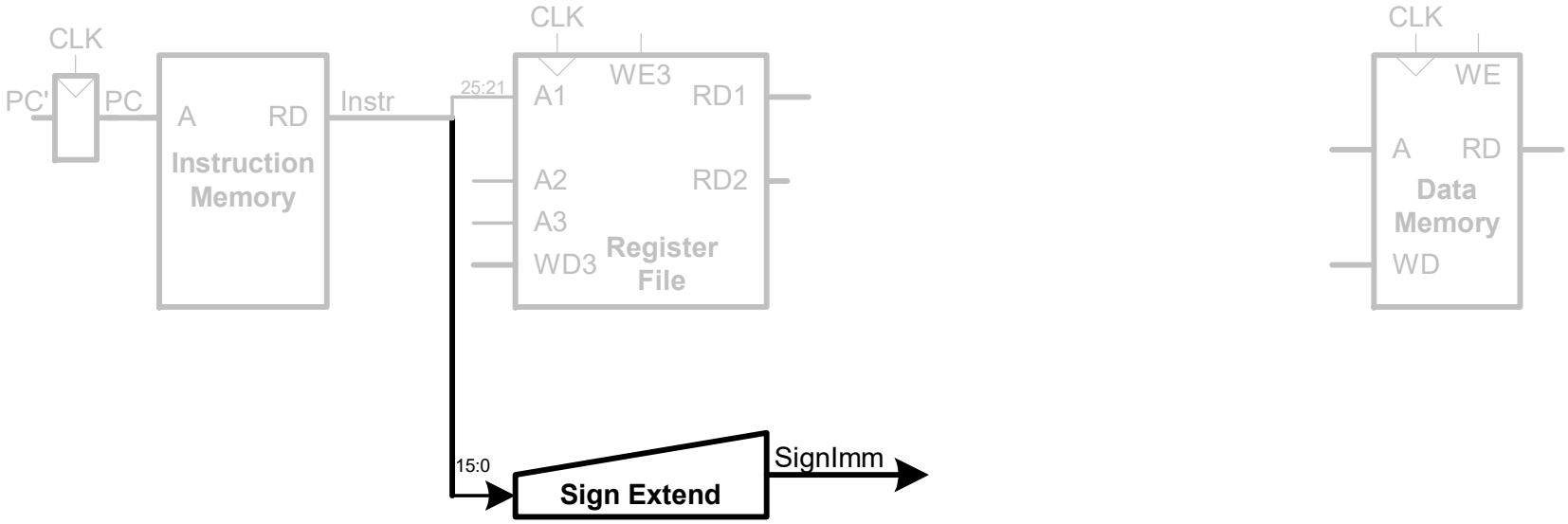


A1: Instr [25:21] = “00000”; RD1 <= “0x00000000” = ($0)

```
lw $s1,0x2000($0)
```

Recap single-cycle Single-Cycle Datapath: lw immediate

- **STEP 3:** sign-extend the immediate

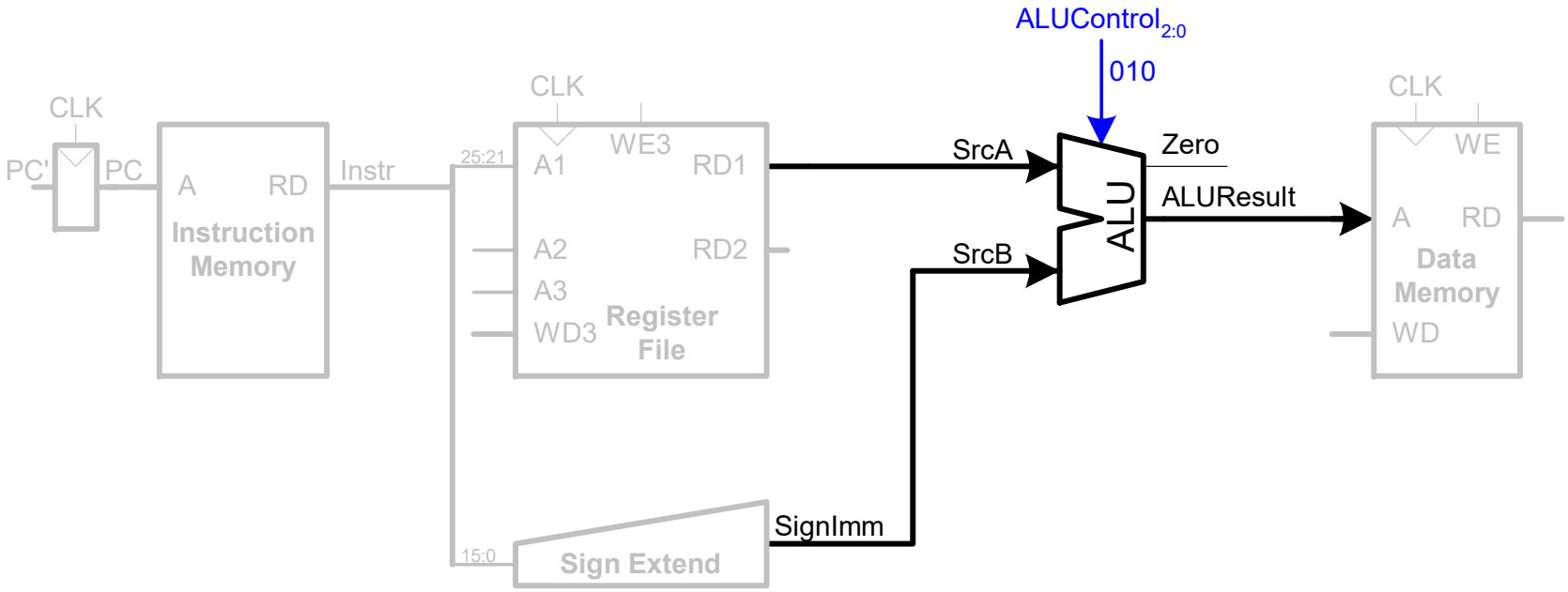


SignImm <= 0x00002000 = Sign Extend (0x2000)

```
lw $s1,0x2000($0)
```

Recap single-cycle Single-Cycle Datapath: lw address

- **STEP 4:** compute the memory address ([rs] + SignImm) in the ALU

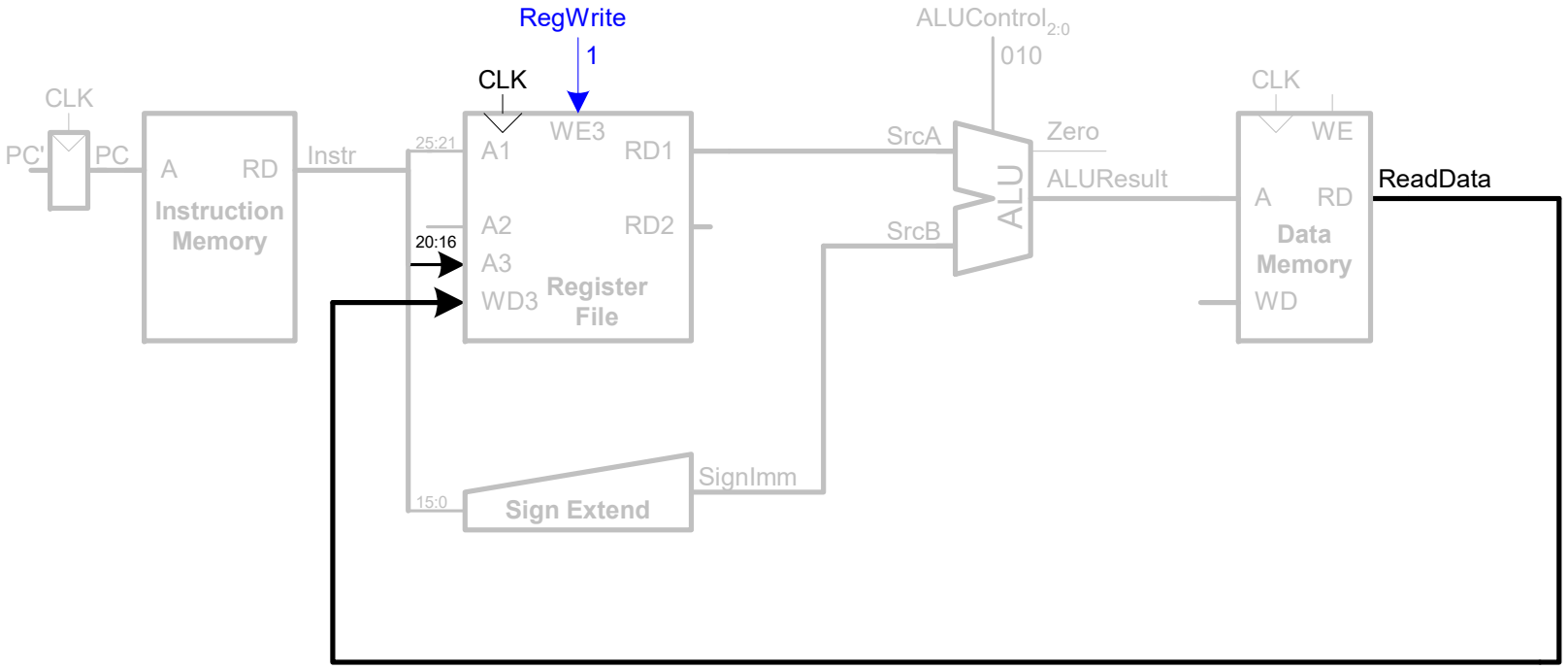


ALUResult <= 0x00002000 = 0x00000000 + 0x00002000

```
lw $s1,0x2000($0)
```

Recap single-cycle Single-Cycle Datapath: lw memory read

- **STEP 5:** read data from memory and write it back to register file, rt



A3: Instr [20:16] = “10001” ($s1)

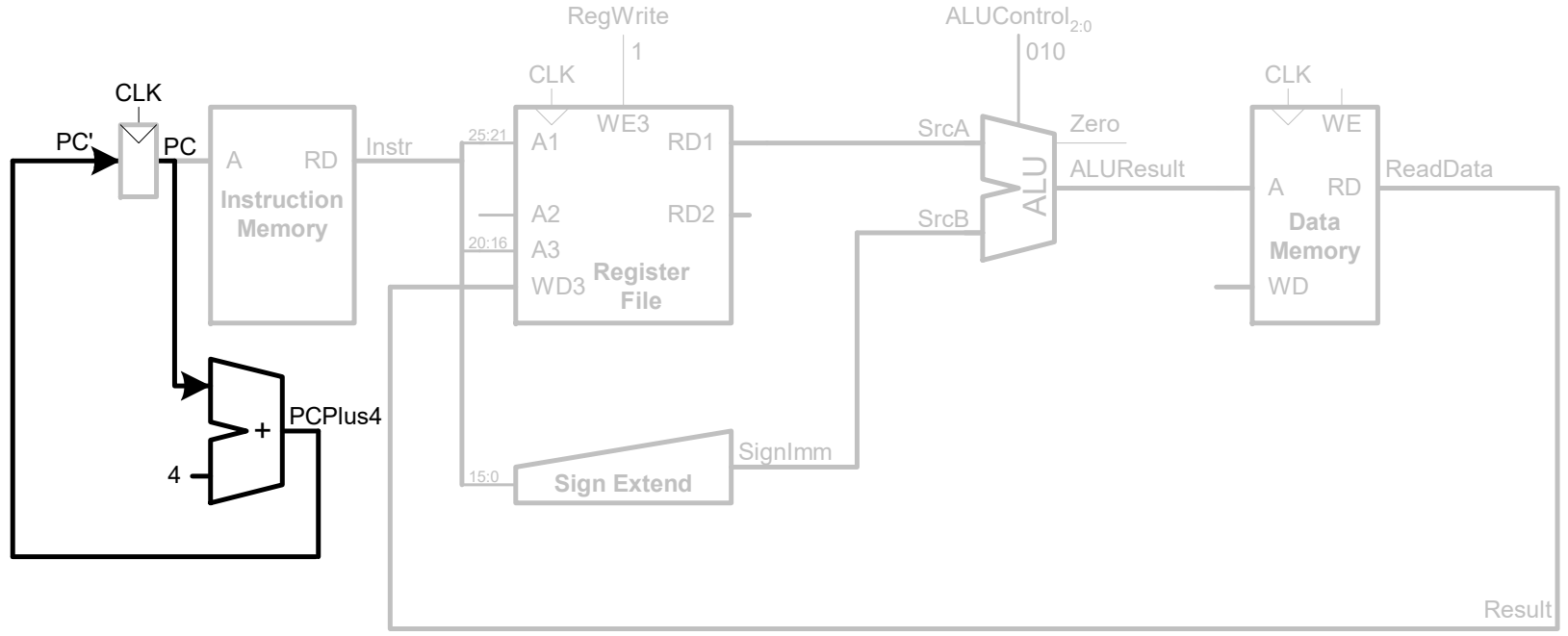
$s1 <= WD3 = MEM[0x00002000]

```
lw $s1,0x2000($0)
```

Recap single-cycle
Single-Cycle Datapath: lw PC increment

P

- **STEP 6:** determine the address of the next instruction (PC+4)



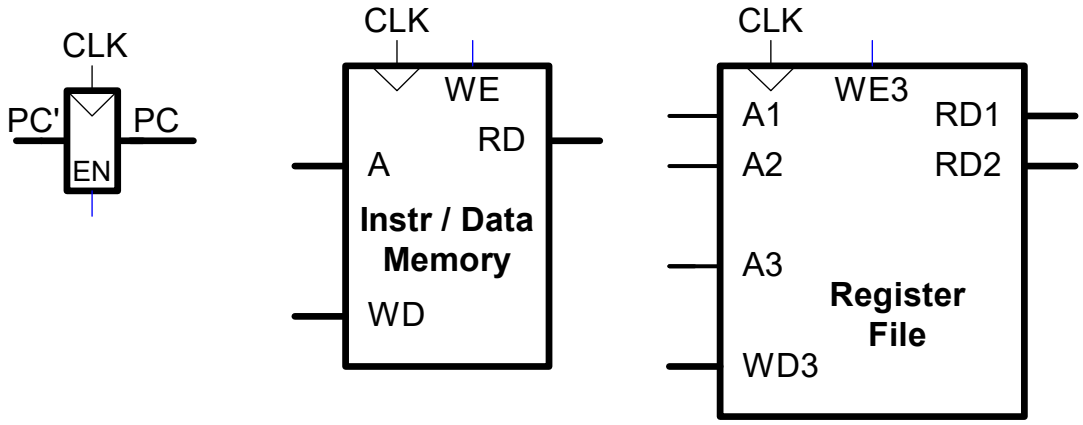
$PC <= $PC + 4

```
lw $s1,0x2000($0)
```

Muticycle State Elements

Replace Instruction and Data memories with a single unified memory.

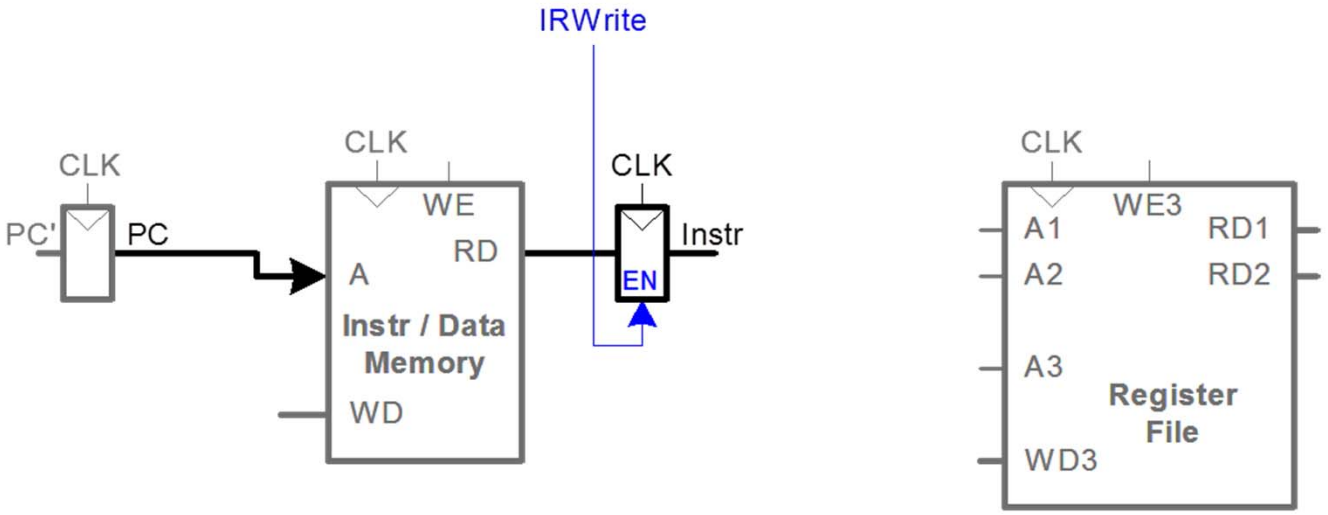
- Each part is accessed in a different clock cycle.



Muticycle Datapath: instruction fetch lw

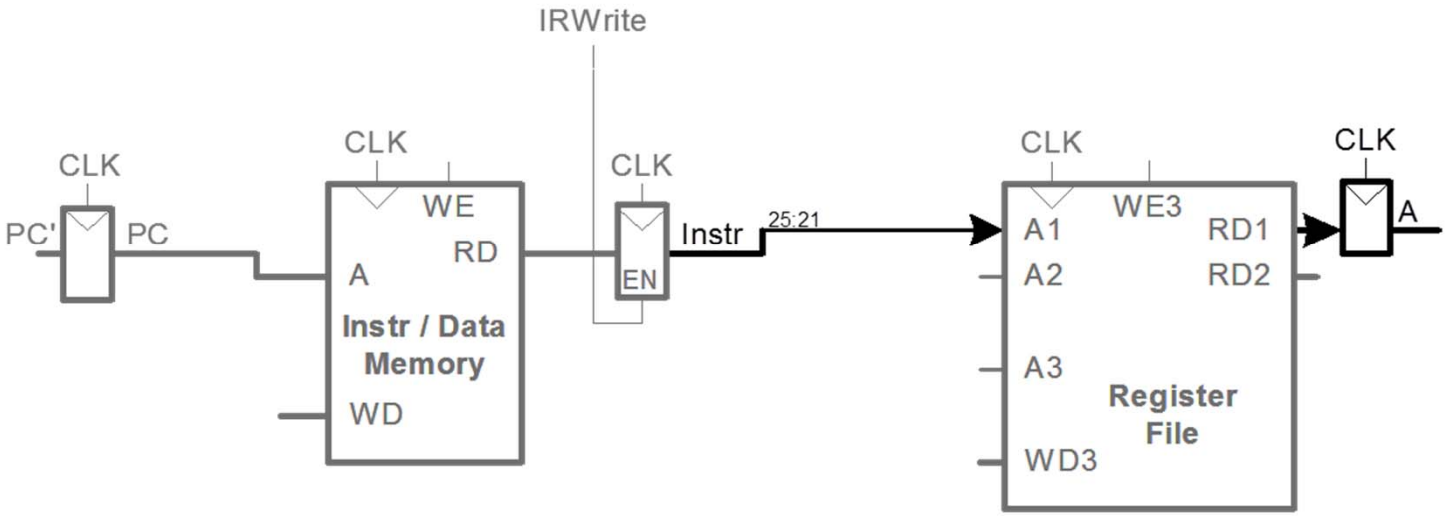
Again, consider executing lw

- **Cycle 1:** instruction fetch.
  - Each step will correspond to a clock cycle, but shorter than in single-cycle.
  - The result of each step must be registered (**Instr** register) enabling it (IRWrite, *enable*) in the clock cycle the information becomes available.



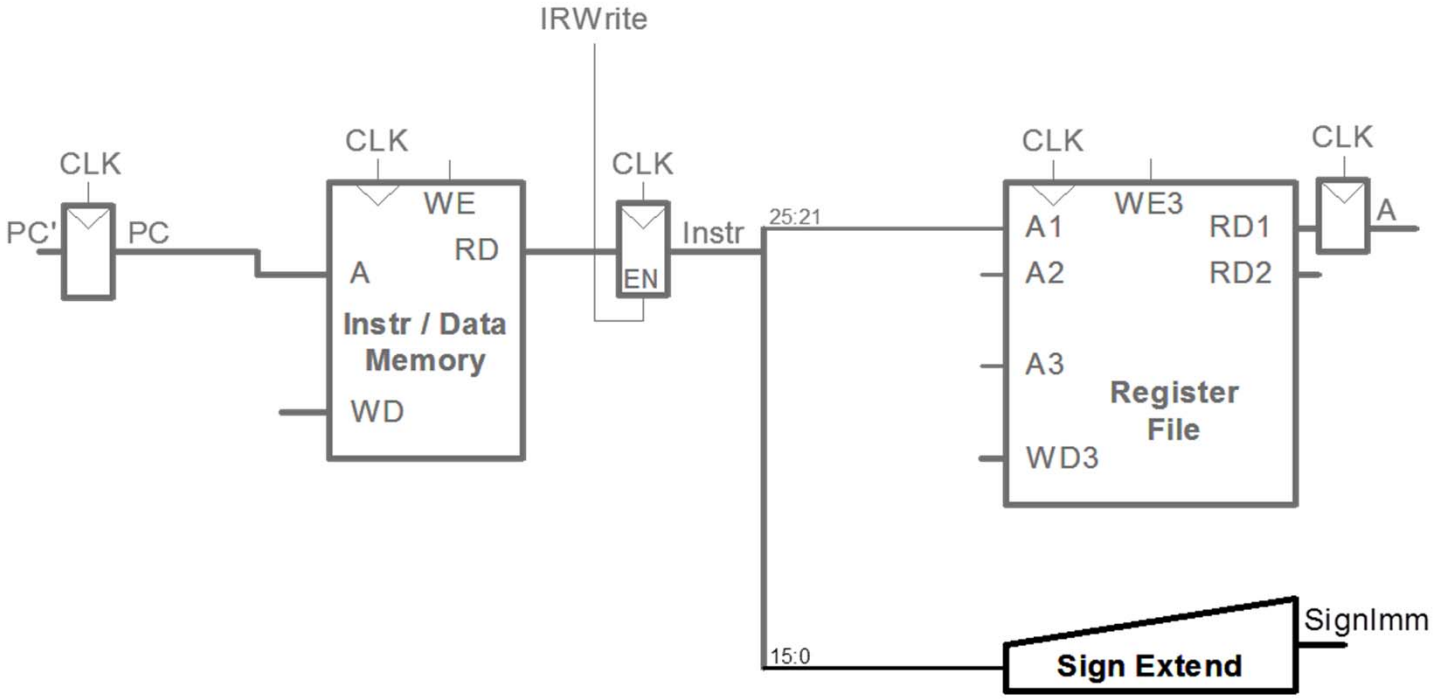
Muticycle Datapath: lw register read

- **Cycle 2:** read source from register file.
  - The result is registered (**A**). No enable, so it is updated all clock cycles (rs does not change during the instruction).



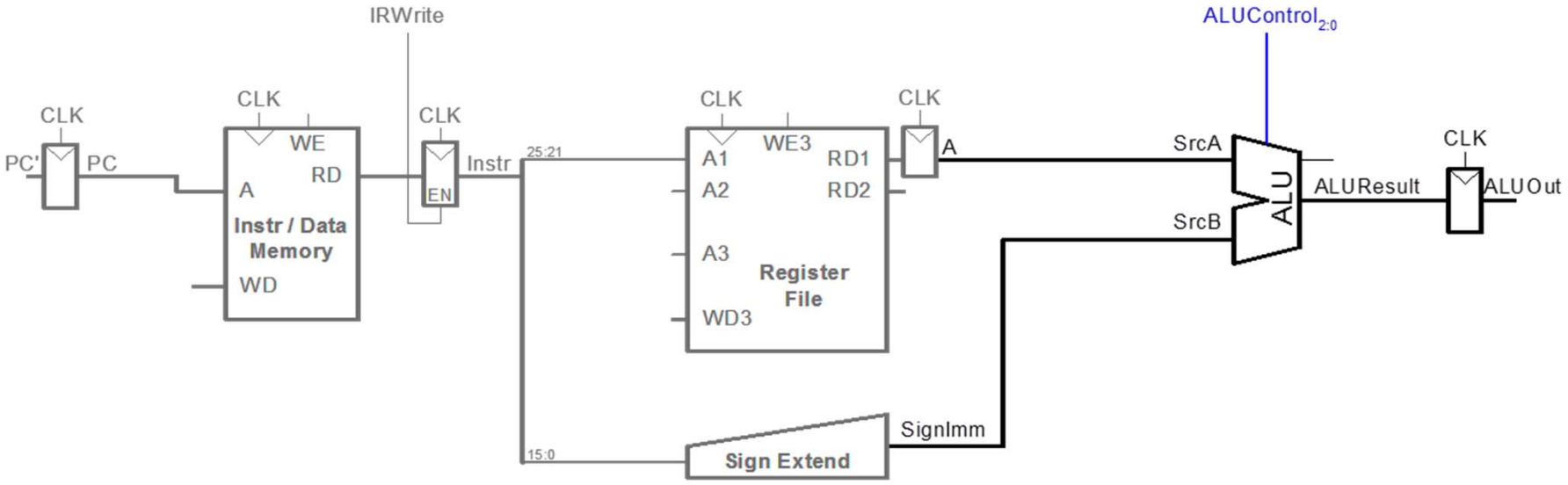
Muticycle Datapath: lw immediate

- **Cycle 2:** second operand (immediate) read in parallel to the register.
  - No new register since Instr is already a register.



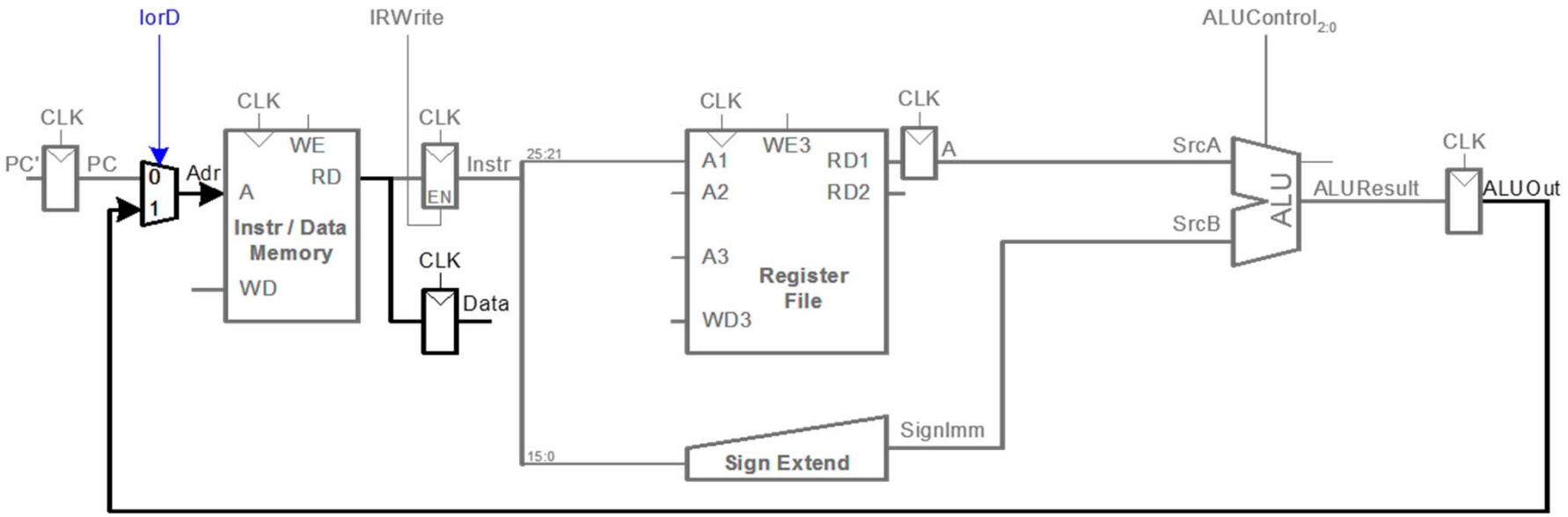
Muticycle Datapath: lw address

- **Cycle 3:** the address is obtained adding the register and the immediate.
  - Result registered (**ALUOut**) to keep it for the next clock cycle.



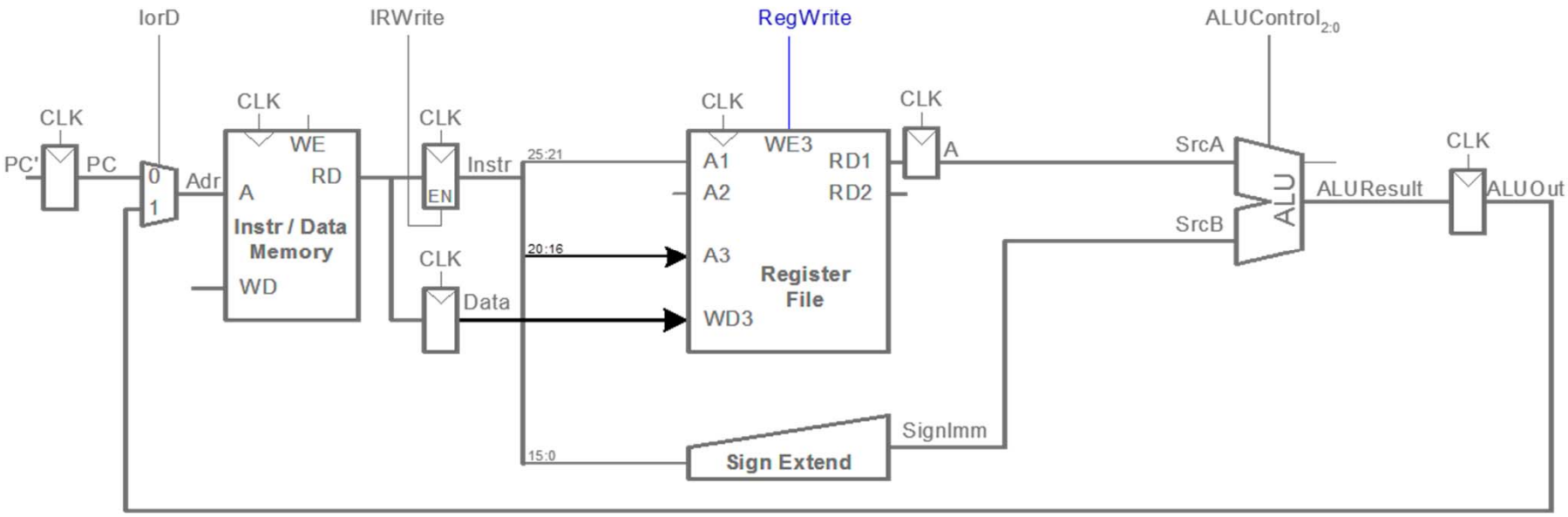
Muticycle Datapath: lw memory read

- **Cycle 4:** the address calculated in step 3 is used for reading the memory. New control signal IorD (instruction or data).
  - The same memory can be used for instruction and date because in this clock cycle there is no instruction fetch.
  - The result is registered (**Data**).



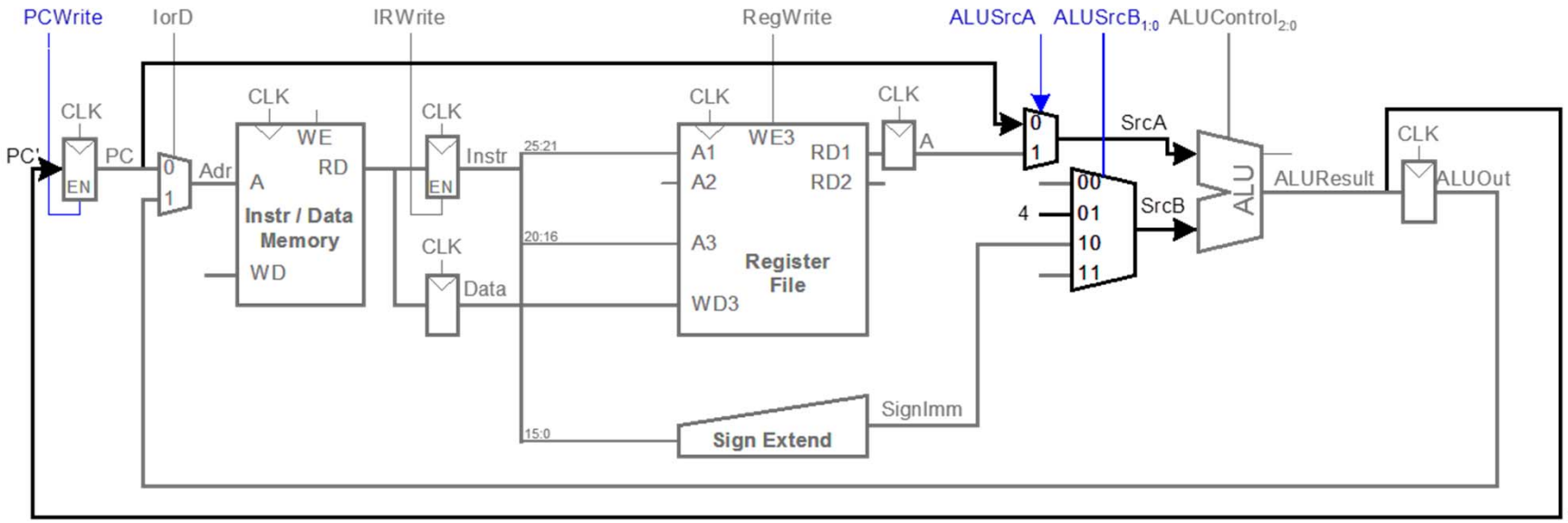
Muticycle Datapath: lw write register

- **Cycle 5:** Data is stored in the destination register.
  - No additional register at the end of the clock cycle, the register file is the register.



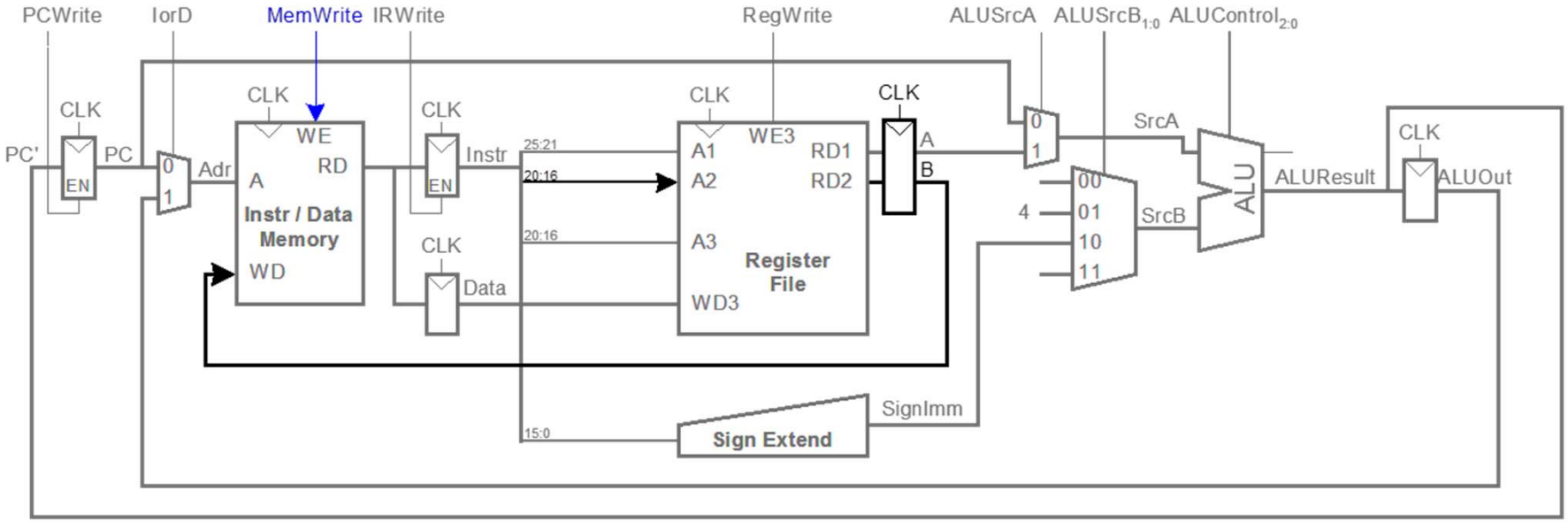
Muticycle Datapath: lw increment PC

- **Cycle 1:** during the first clock cycle of each instruction, the new PC (PC+4) is generated and registered in PC. This may be later modified in branches / jumps.
  - The ALU is used as adder since the ALU is not doing any other thing this clock cycle.
  - PC and 4 must reach the ALU operands (more multiplexers).



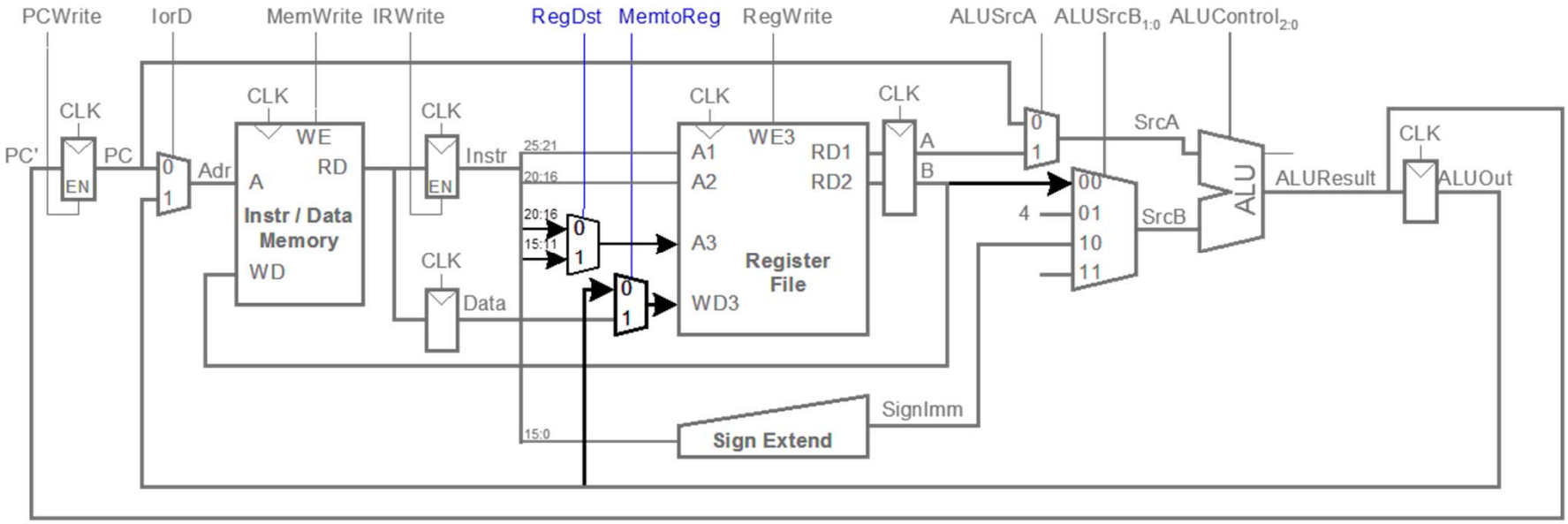
Muticycle Datapath: sw

- **Cycle 2:** the second source operand is registered (register **B**) because the register read and memory write are executed in different cycles.
- **Cycle 3:** memory address calculated and registered (**ALUOut**).
- **Cycle 4:** rt register (stored in B) is written into memory (MemWrite), so it must be sent to the memory input WD.



Muticycle Datapath: R-type

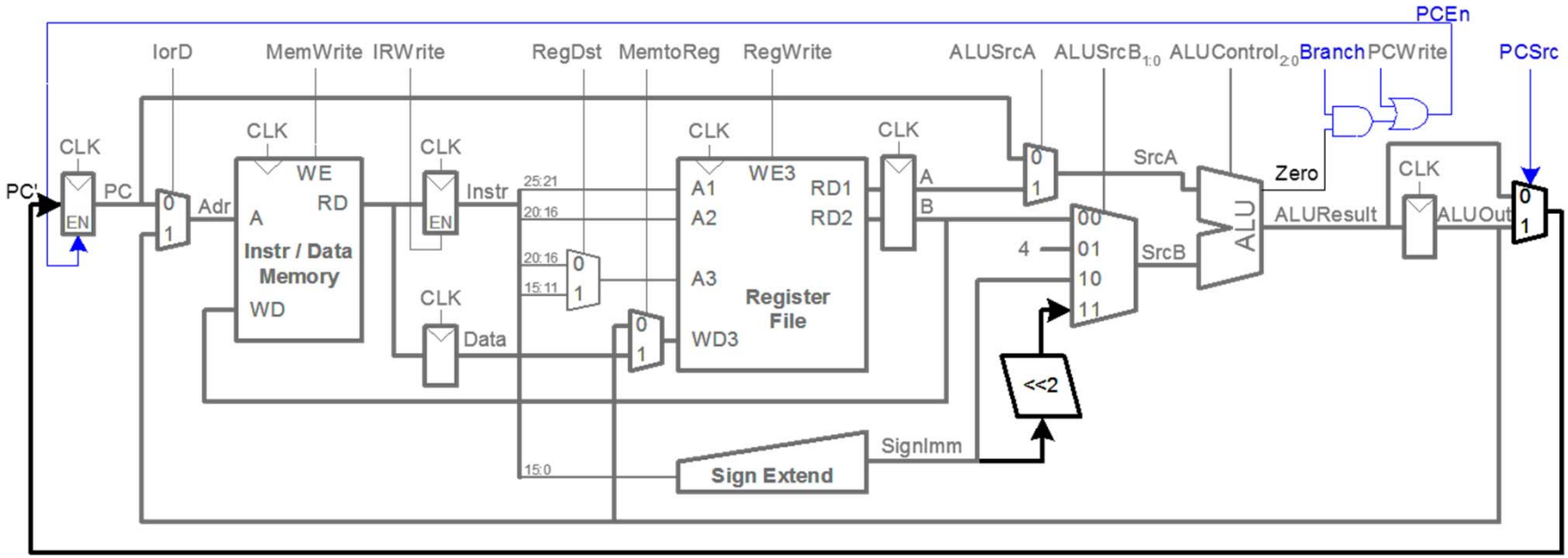
- **Cycle 2:** read rs and rt, the two source operands.
- **Cycle 3:** ALUResult registered in *ALUOut.*
- **Cycle 4:** ALUOut content written in the register file (MemtoReg=‘0’). Destination register is rd instead of rt (RegDst=‘1’).



Muticycle Datapath: beq

Calculate branch target address: BTA = (PC+4) + (Sign Imm << 2)

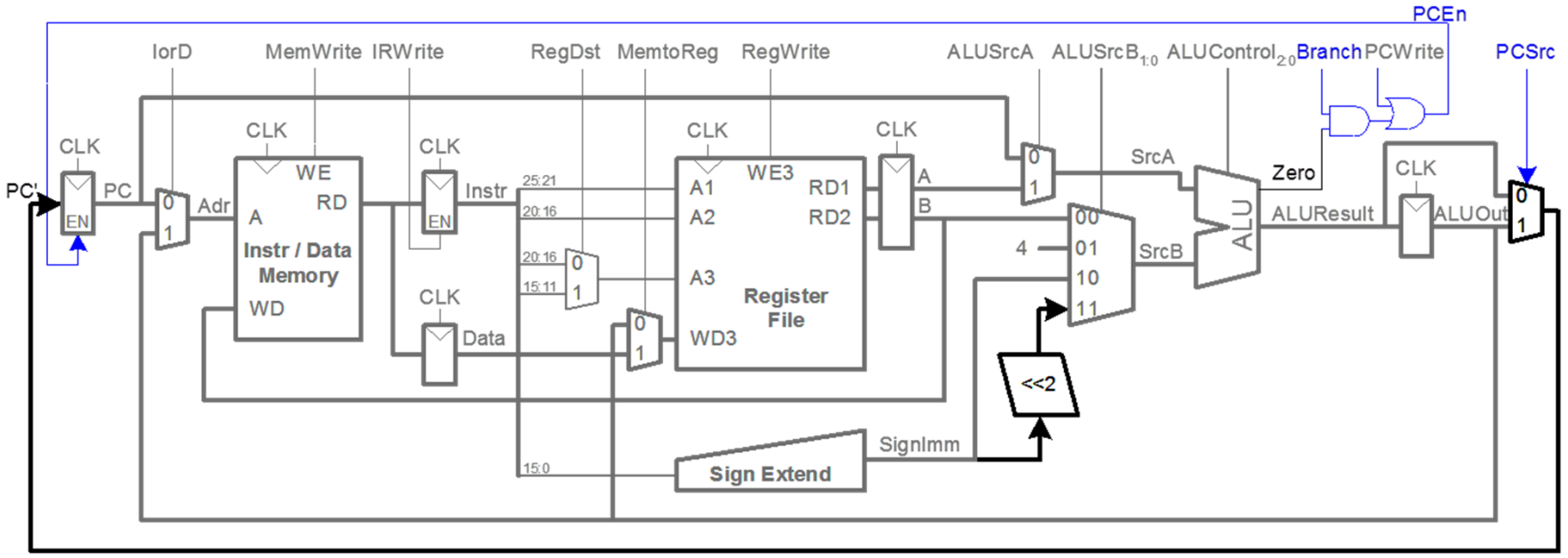
- **Cycle 1:** PC <= PC+4 (using ALU), enable PC (PCWrite='1').
- **Cycle 2:** ALUOut <= (PC+4) + (Sign Imm << 2) (using ALU).
- **Ciclo 3:** registers subtracted to obtain Z. ALUOut is sent to the PC (PCSrc='1') just in case the branch is taken (Branch AND Z = '1').



Muticycle Datapath: beq (decision)

- Branch taken or not depending on flag Z (Zero).
- Two PC updates:
  - ✓ PCWrite='1', cycle 1 in order to obtain PC+4 (**always**)
  - ✓ Zero and Branch (Z='1' and beq instruction).

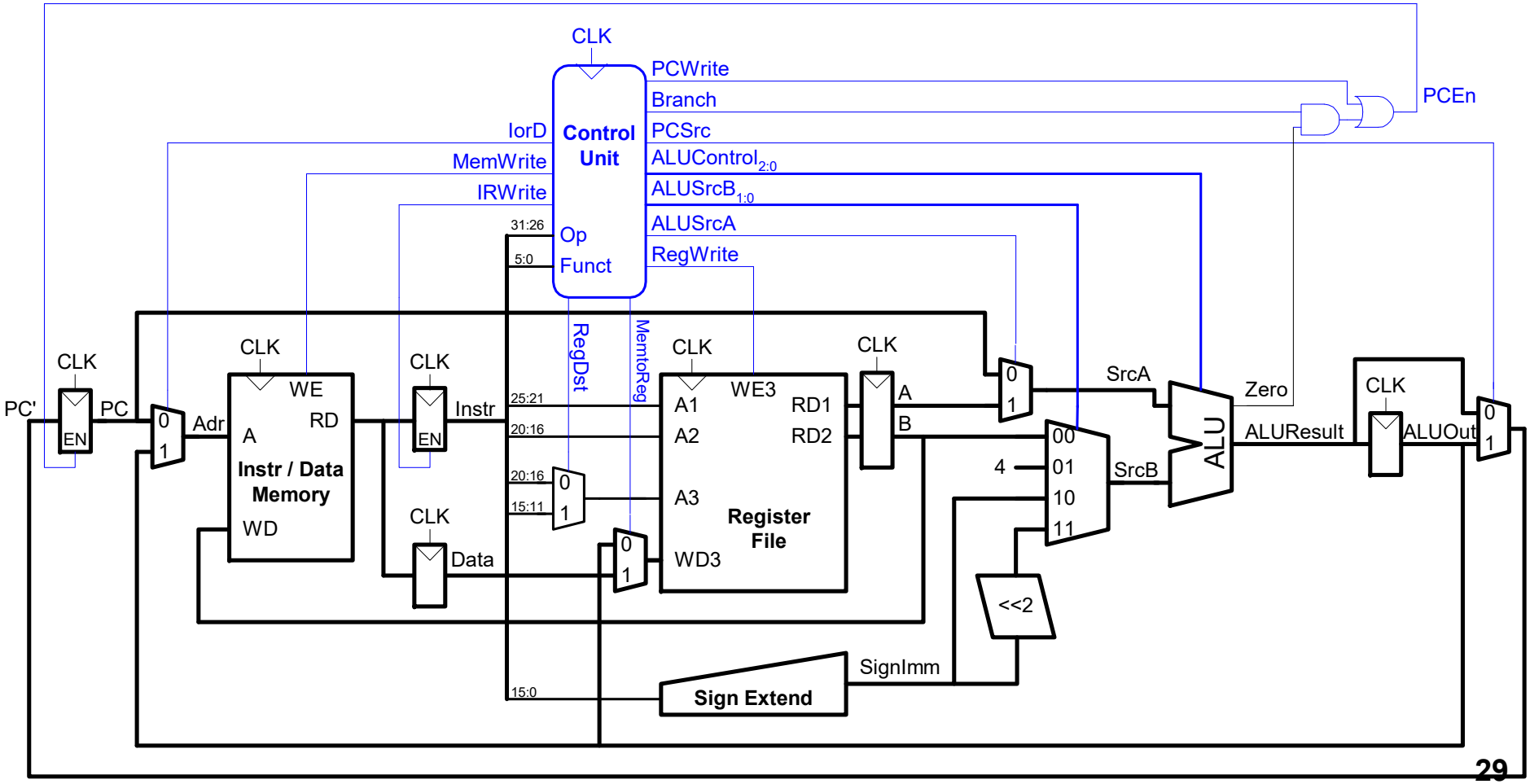
    Branch must be generated in cycle 3 of beq, branch taken or not depending on Zero.



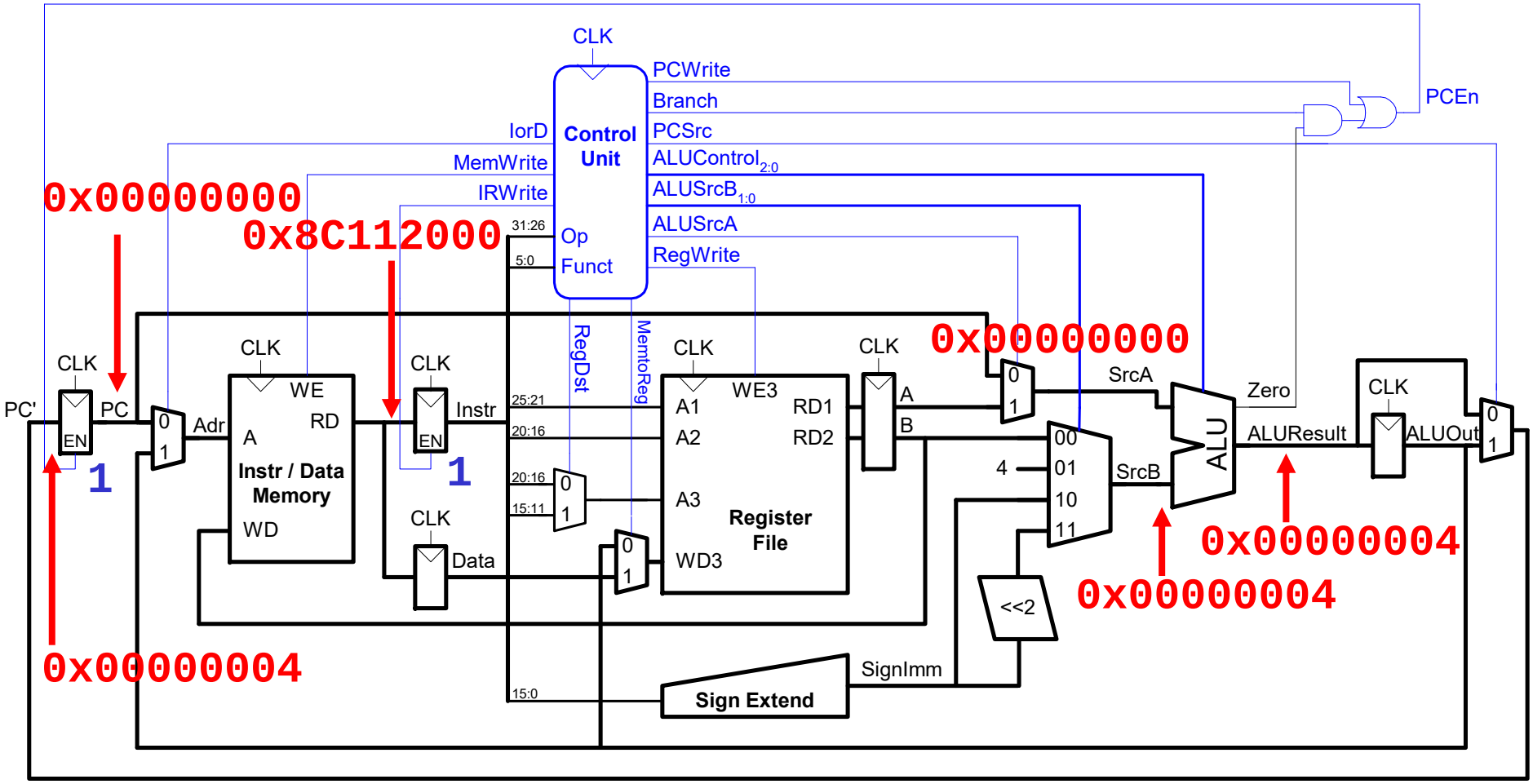
Multicycle Processor (Datapath and control)

P

In the multicycle processor, the control signals are generated depending on the instruction (decoding opcode and funct) **and on the clock cycle**.



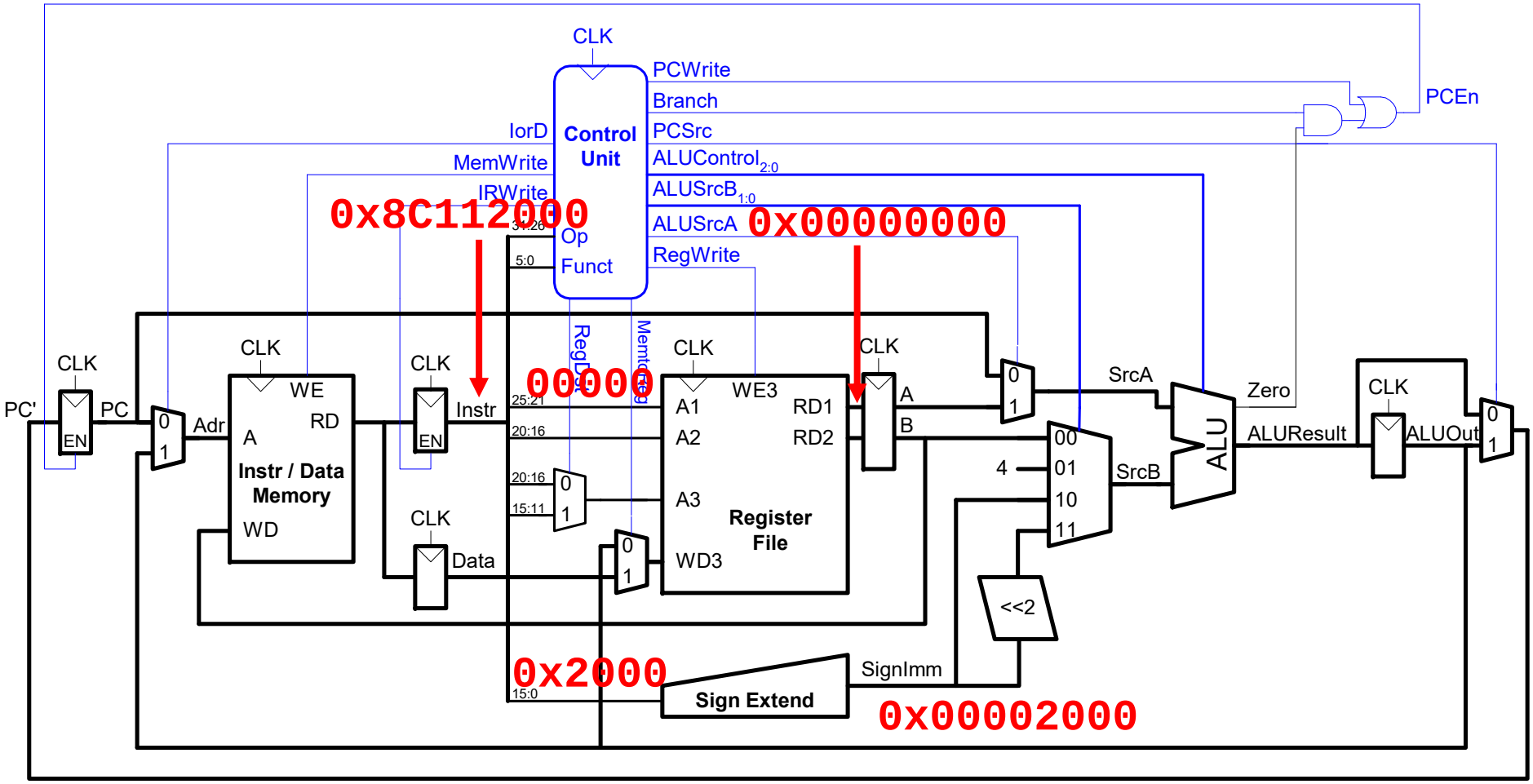
Muticycle datapath
lw example, cycle 1



MEM[0x00000000]=0x8C112000 => lw $s1,0x2000($0)

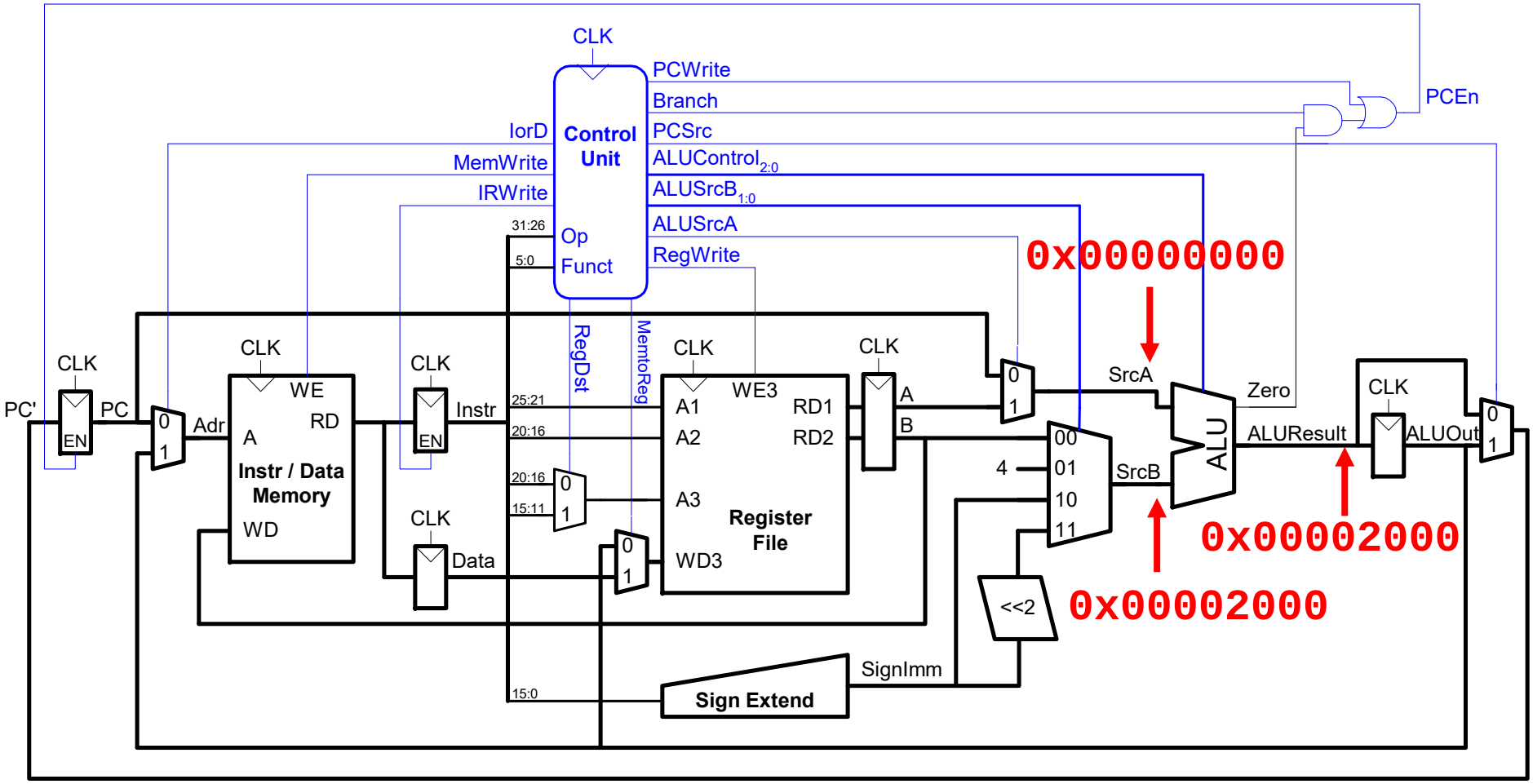
Supposing that MEM[0x00002000]=0x12345678

Muticycle datapath
lw example, cycle 2



MEM[0x00000000]=0x8C112000 => lw $s1,0x2000($0)
Supposing that MEM[0x00002000]=0x12345678

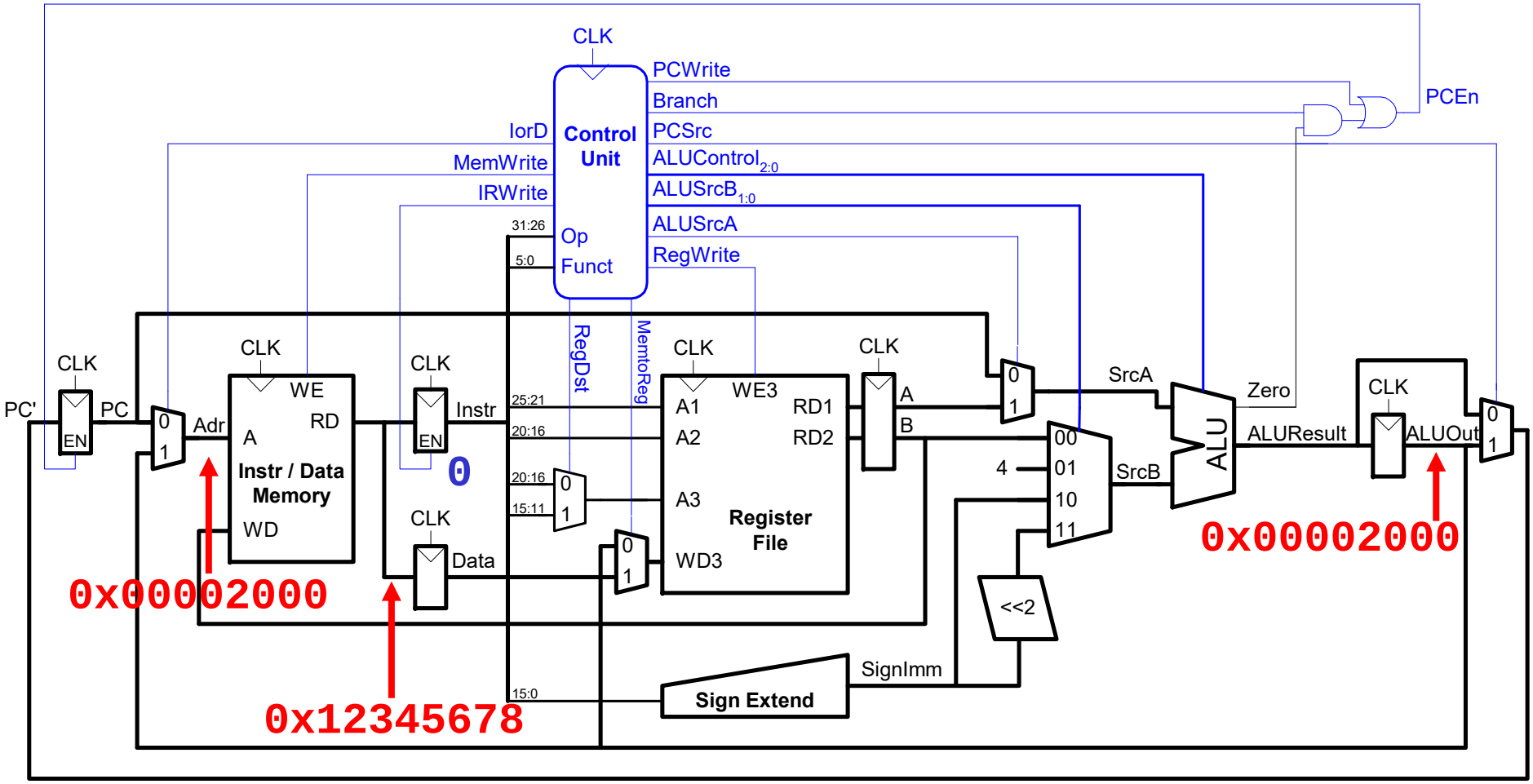
Muticycle datapath
lw example, cycle 3



MEM[0x00000000]=0x8C112000 => lw $s1,0x2000($0)

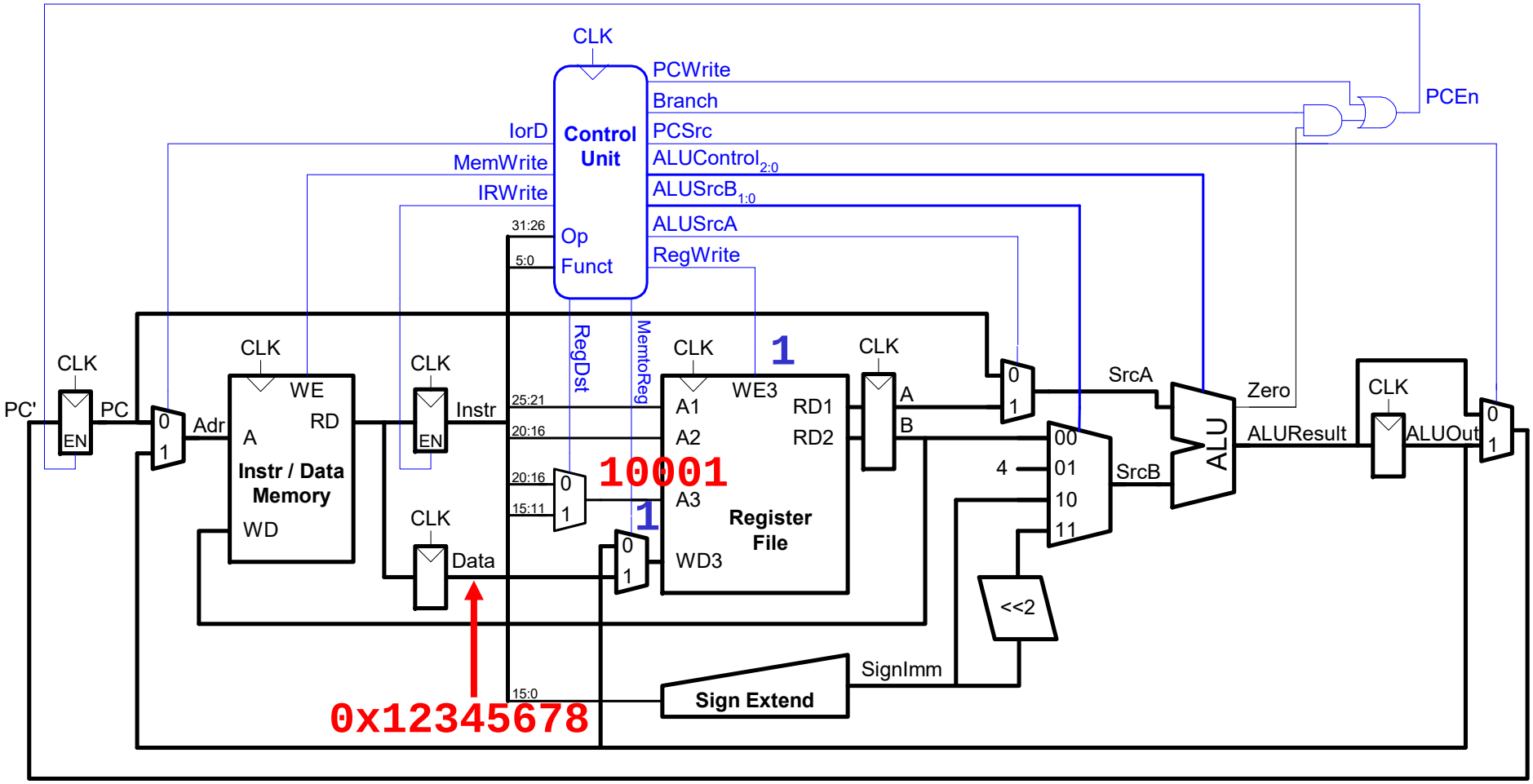
Supposing that MEM[0x00002000]=0x12345678

Muticycle datapath
lw example, cycle 4



MEM[0x00000000]=0x8C112000 => lw $s1,0x2000($0)
Supposing that MEM[0x00002000]=0x12345678

Muticycle datapath
lw example, cycle 5



MEM[0x00000000]=0x8C112000 => lw $s1,0x2000($0)
Supposing that MEM[0x00002000]=0x12345678

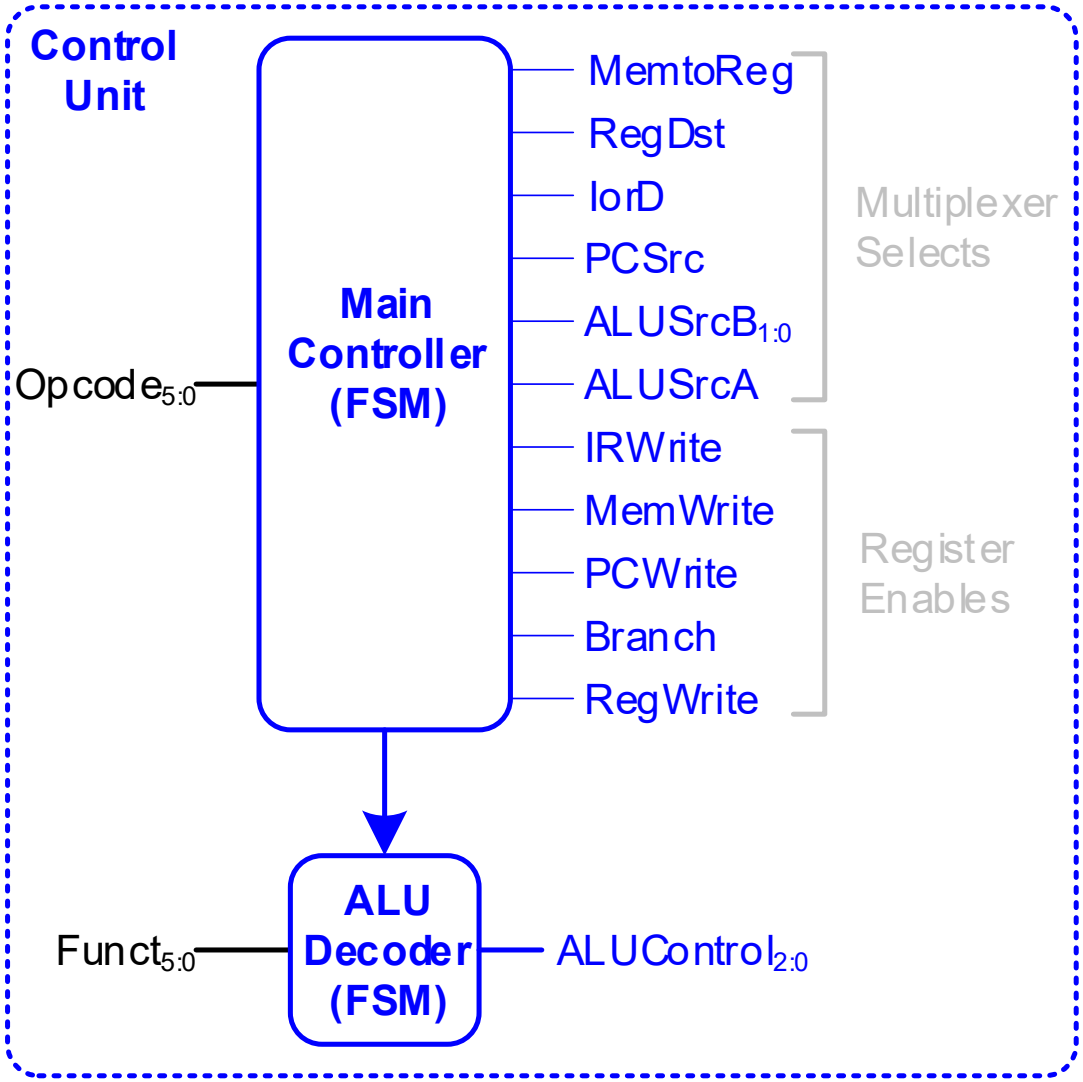
Outline

- Summary single-cycle MIPS
- Multicycle datapath
- **Multicycle control path**
- Adding more instructions

Instructions planning

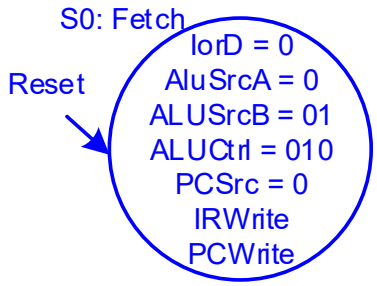
	lw	sw	R-type	beq
Cycle 1	Instruction fetch. PC <= PC + 4			
Cycle 2	Operands read.			**a.** Operands read. **b.** BTA calculated (not writting PC)
Cycle 3	Data Memory address obtained		ALU operation	**a.** Subtract in ALU **b.** If Z=1, update PC
Cycle 4	Data Memory read	Data Memory write	Register write	
Cycle 5	Register write			

Control Unit

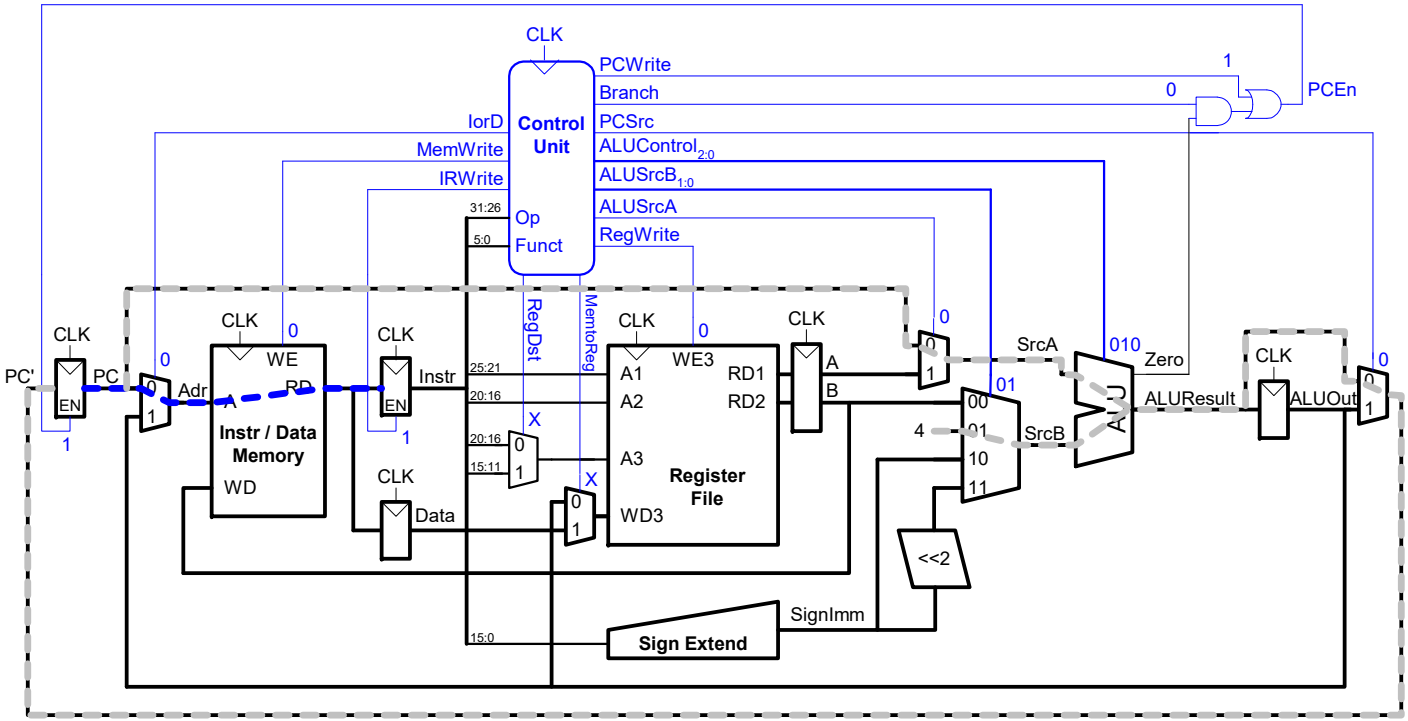


- Control signals different each clock cycle => can not be combinational logic.
- ALU Decoder remains the same. If the ALU does several operations during an indstruction, ALUControl is changed in the corresponding cycle.
- ***Main Controller*** is a finite state machine (FSM).

Main Controller FSM: cycle 1

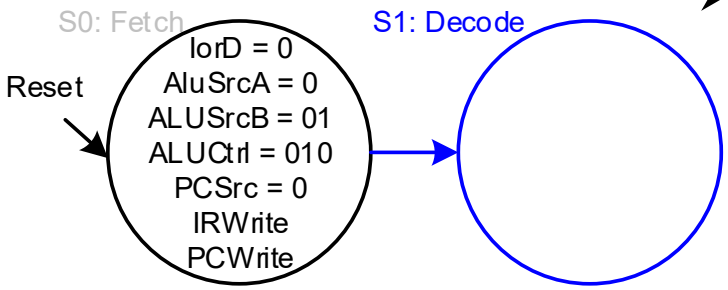


- **Cycle-1:** the same for all instructions.
  - ✓ Instruction fetch (instruction memory).
  - ✓PC updated to PC+4 (ALU adds).

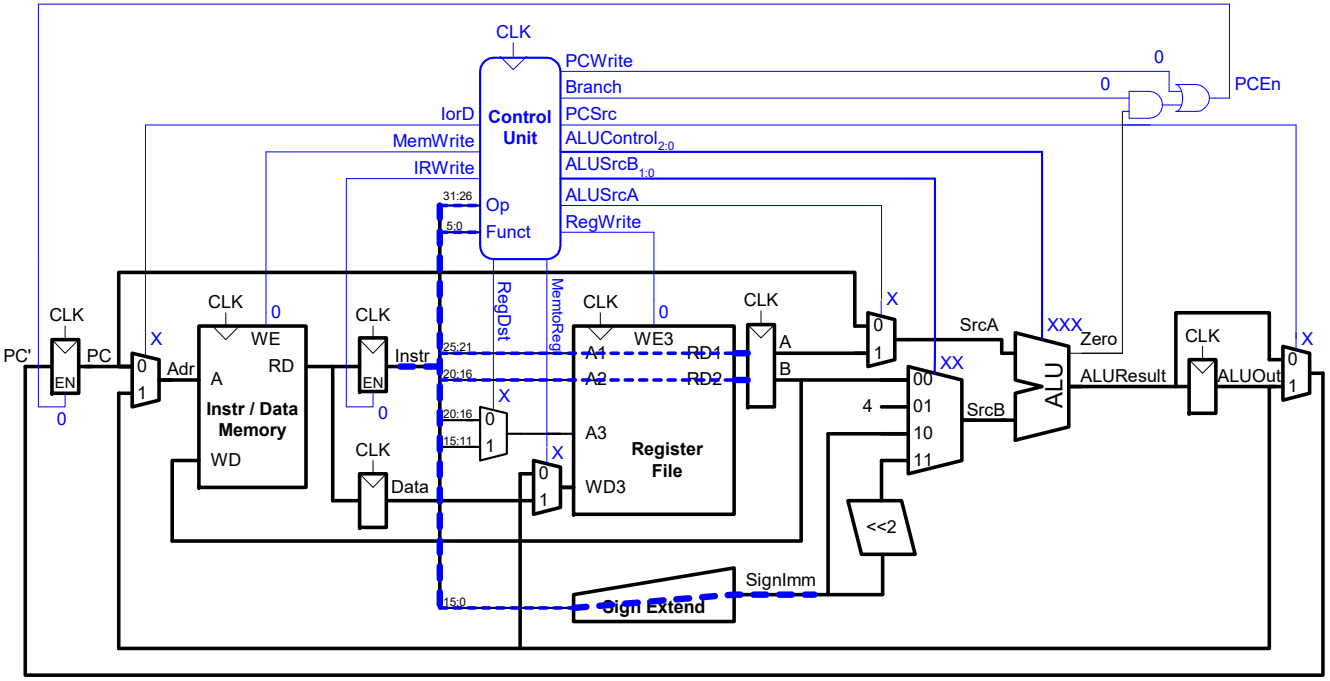


Note: only the relevant mux signals and register enabled are shown. Otherwise, mux can be at X and enables are at '0'.

Main Controller FSM: cycle 2

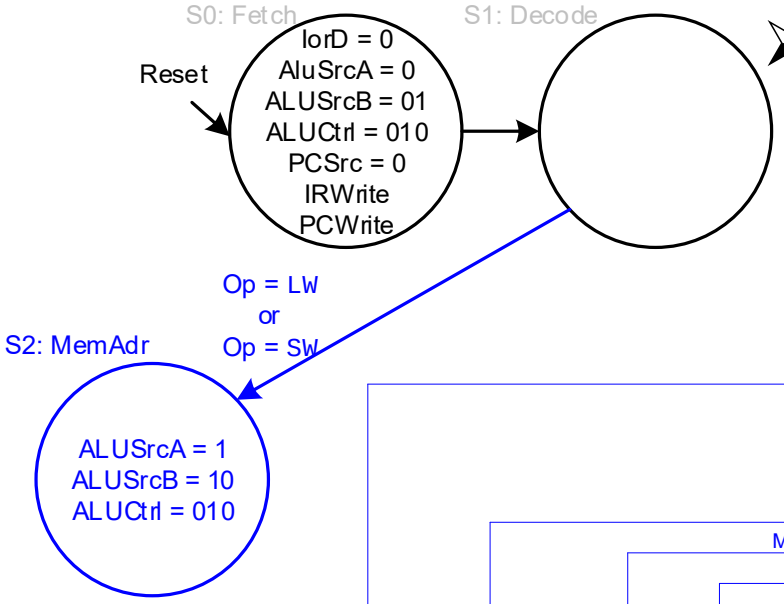


- **Cycle-2:** instruction decode and operands read.
  - ✓ This cycle is also common for all instructions. A and B are registered (even if not used later). No enable in A and B, always registered.

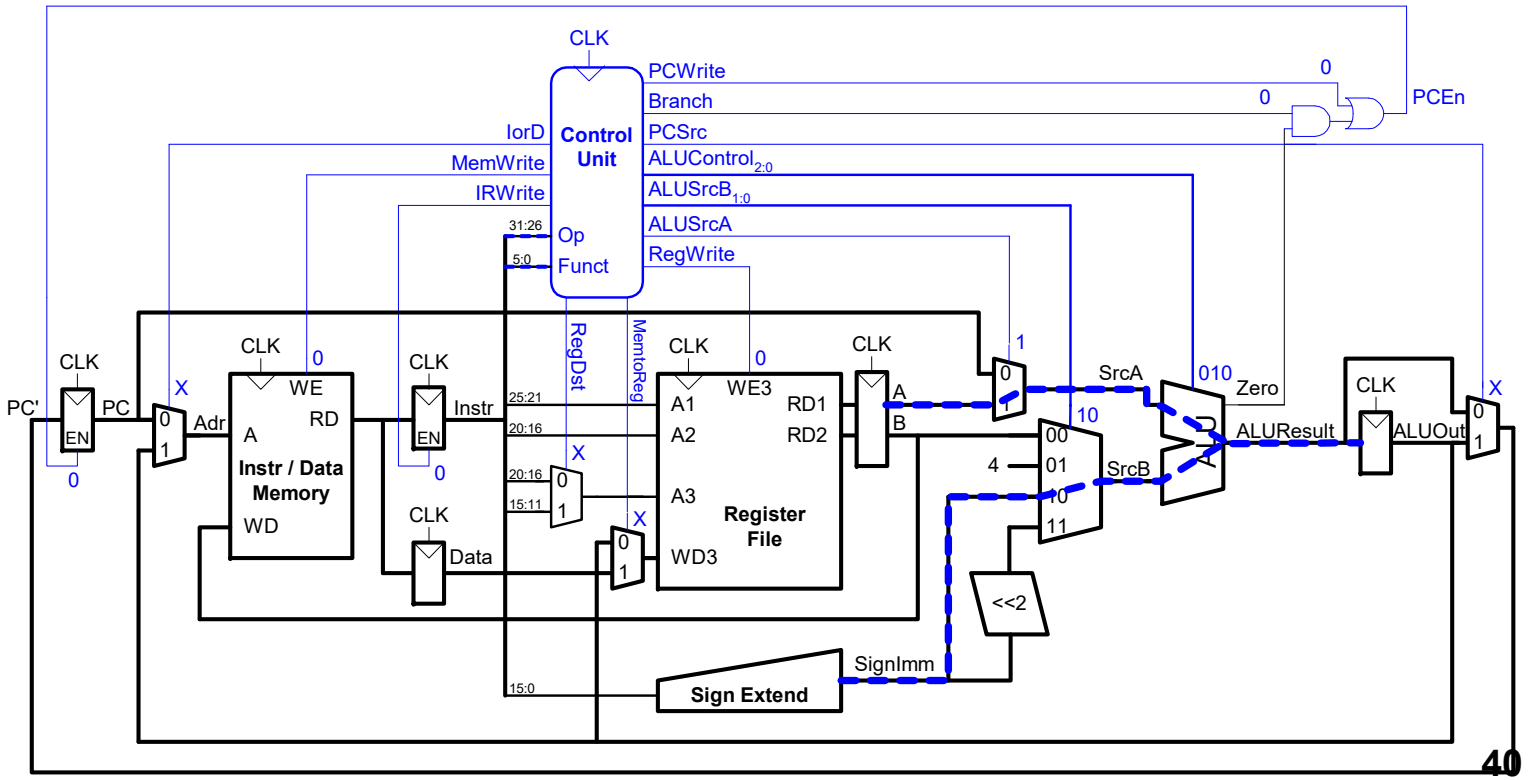


✓From this point on, different evolution for each instruction.

Main Controller FSM: cycle 3 lw/sw



Cycle-3 (for lw or sw): the data memory address is calculated and registered in ALUOut (no enable, always registered).

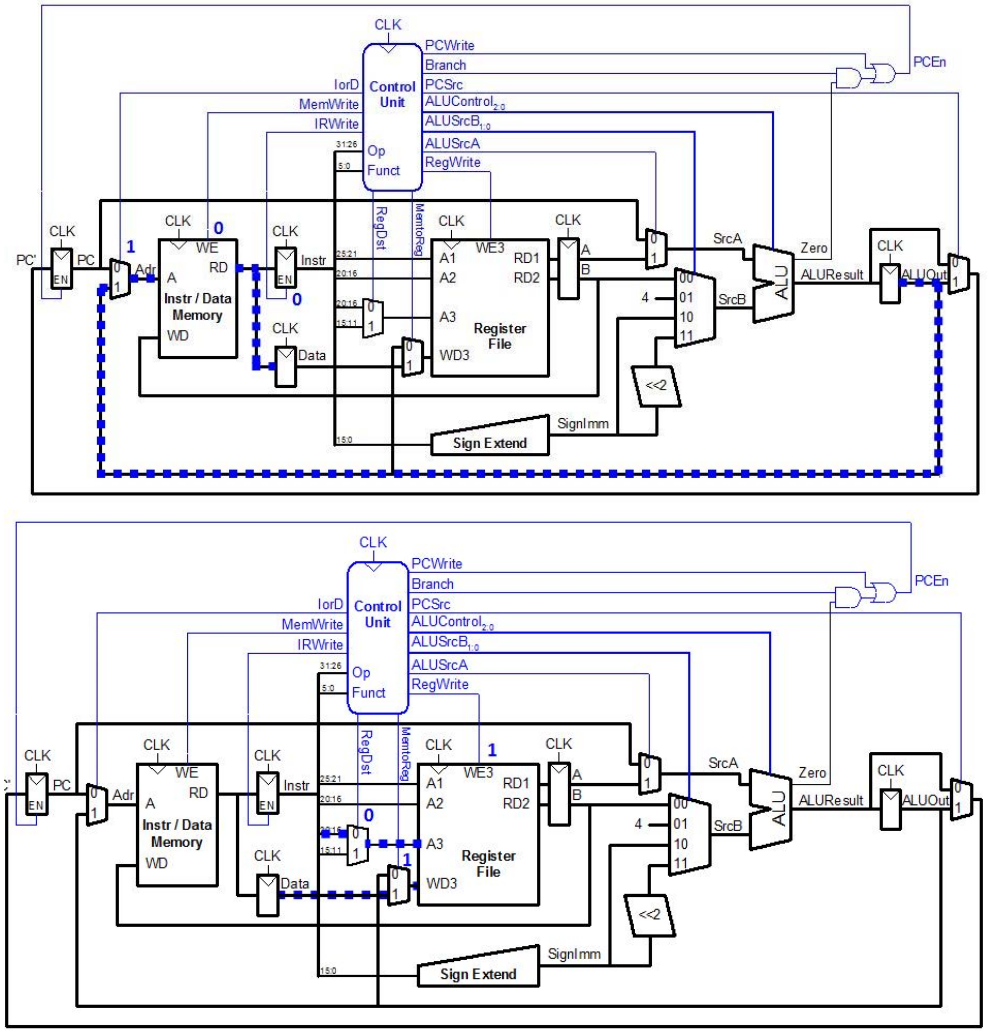


Main Controller FSM: cycles 4, 5 lw

- **Cycle-4:** the address (ALUOut) is sent to memory (IorD='1') and the read data is registerd in Data (no enable, always registered).

- **Cycle-5:** the data (Data) is stored in the register file (rt is the destination register).

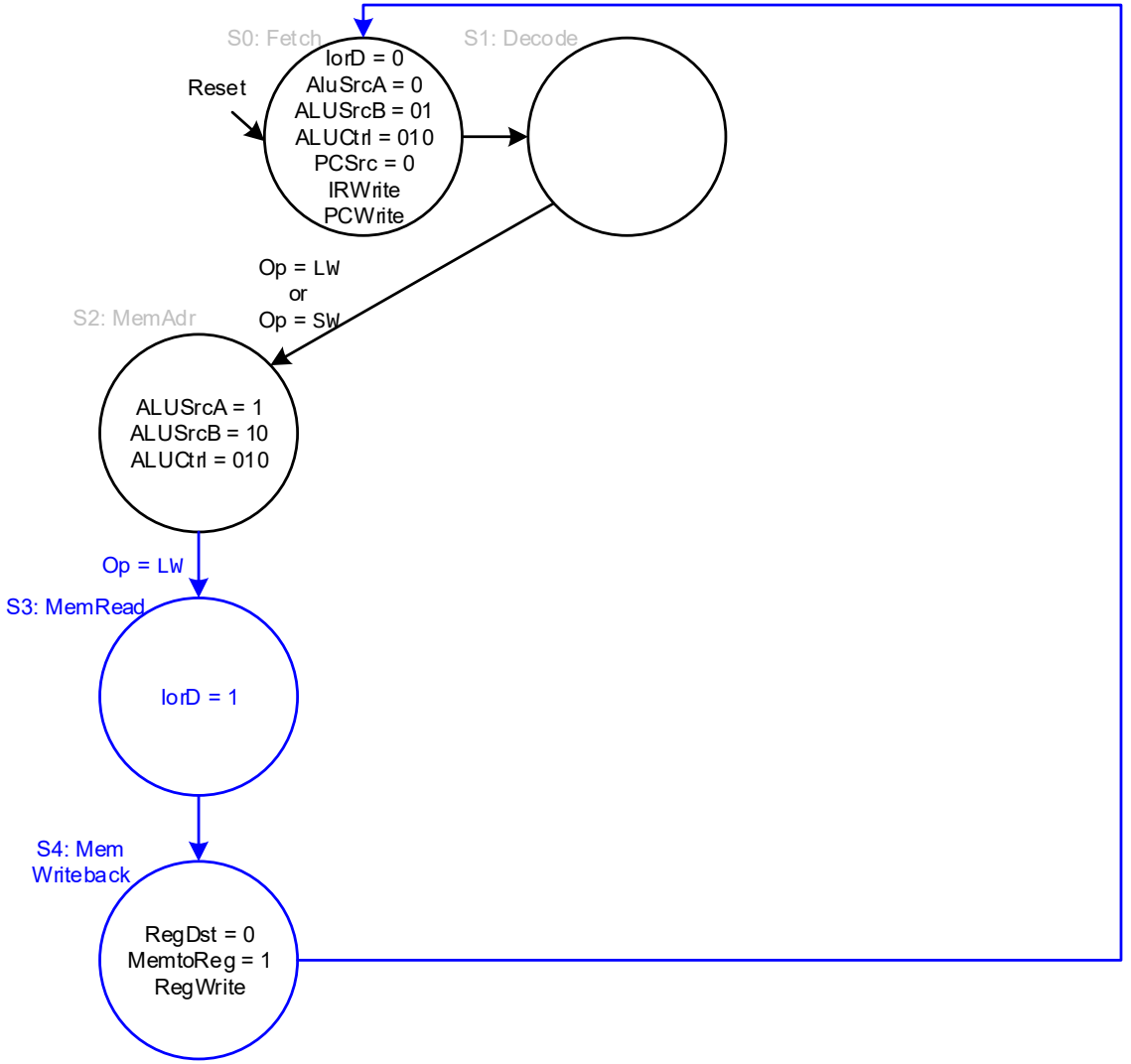
  End of instruction (back to S0).



Main Controller FSM: cycles 4, 5 lw

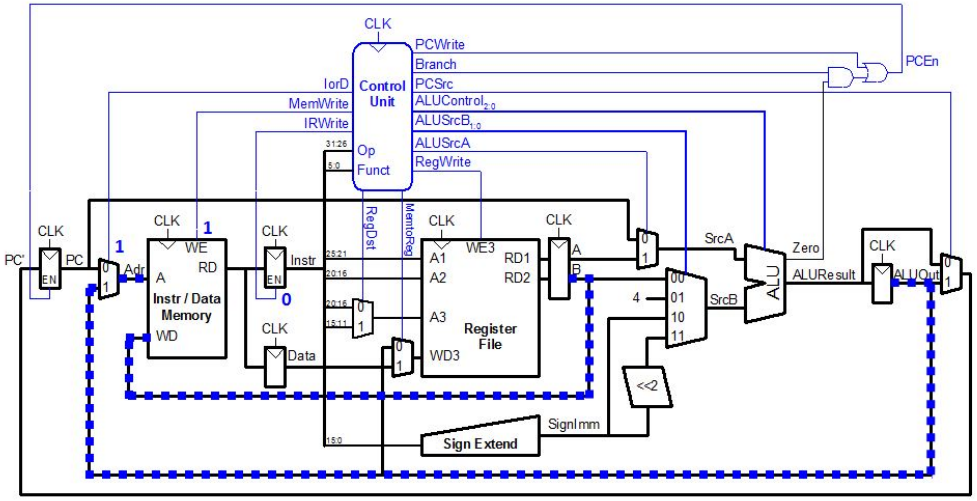
- **Cycle-4:** the address (ALUOut) is sent to memory (IorD='1') and the read data is registerd in Data (no enable, always registered).
- **Cycle-5:** the data (Data) is stored in the register file (rt is the destination register).

  End of instruction (back to S0).



Main Controller FSM: cycle 4 sw

- **Cycle-4:** the address (ALUOut) is sent to memory (IorD=‘1’) to store the data coming from register rt, stored in B. MemWrite=‘1’.



End of instruction (back to S0).

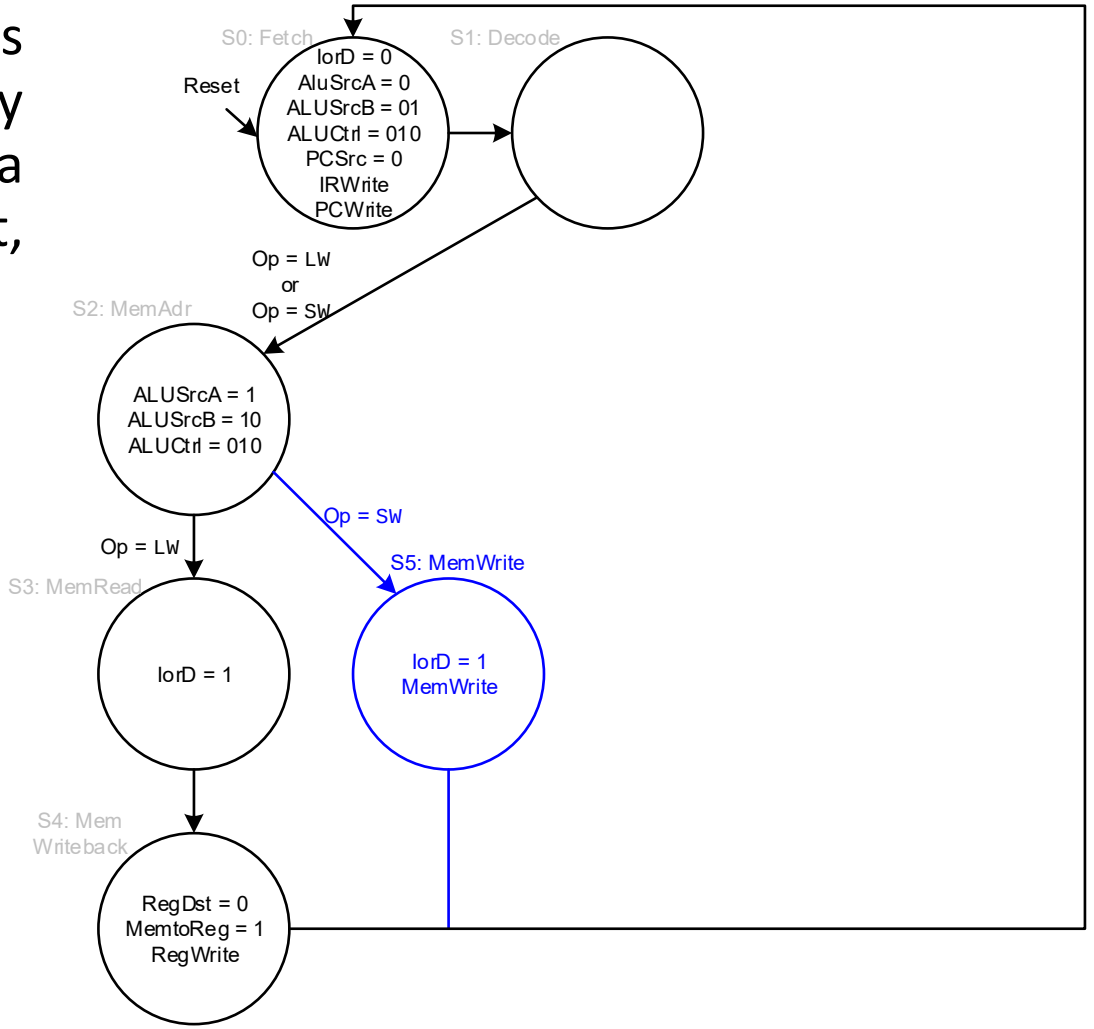
sw needs 4 cycles, not 5.

Main Controller FSM: cycle 4 sw

- **Cycle-4:** the address (ALUOut) is sent to memory (IorD=‘1’) to store the data coming from register rt, stored in B. MemWrite=‘1’.

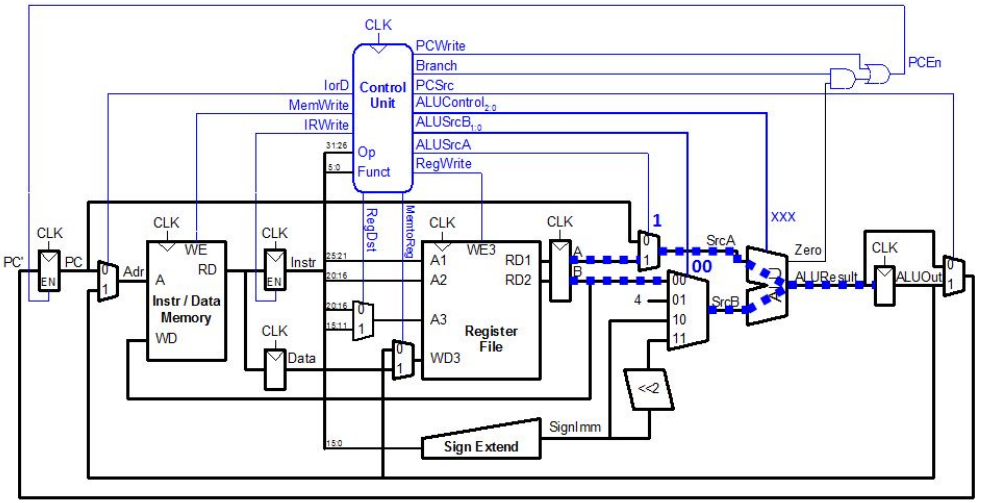
End of instruction (back to S0).

sw needs 4 cycles, not 5.

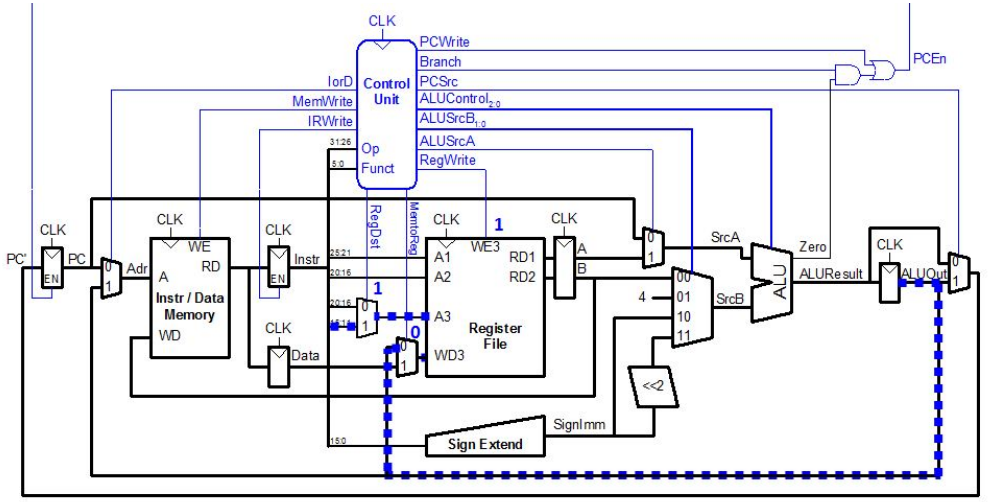


Main Controller FSM: cycles 3, 4 R-type

- **Cycle-3:** the ALU makes the operation. A and B (from rs and rt) are used. The operation depends on funct.



- **Cycle-4:** the result, registered in ALUOut, is stored in rd.



End of instruction (back to S0).

4 cycles needed.

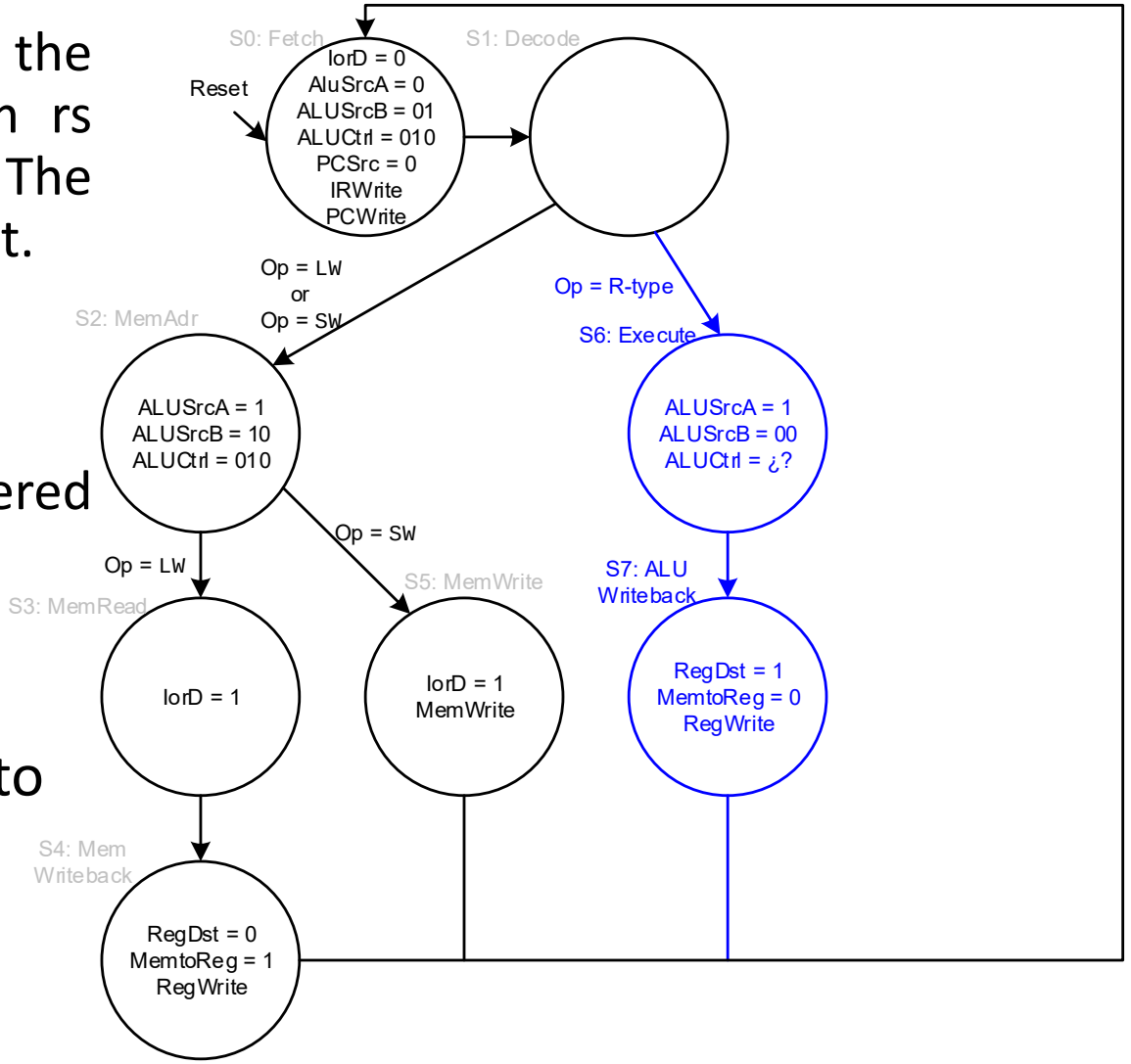
Main Controller FSM: cycles 3, 4 R-type

- **Cycle-3:** the ALU makes the operation. A and B (from rs and rt) are used. The operation depends on funct.

- **Cycle-4:** the result, registered in ALUOut, is stored in rd.

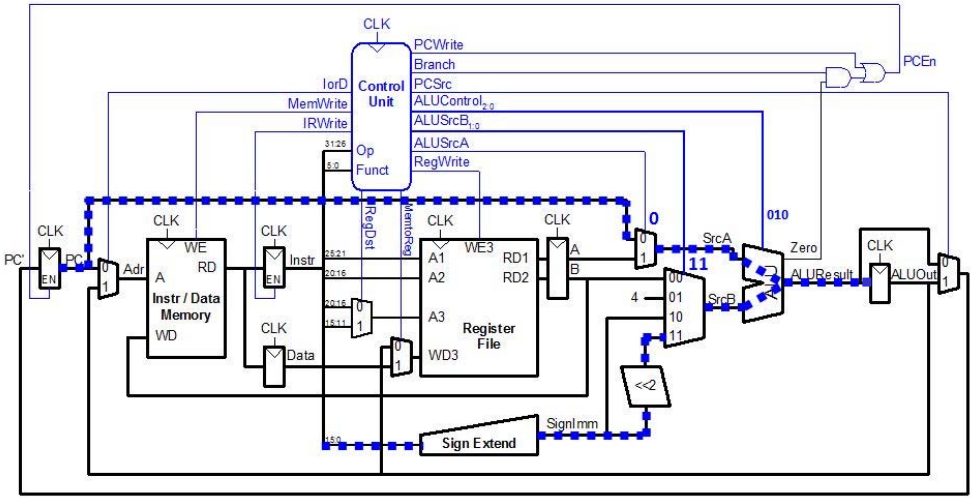
End of instruction (back to S0).

4 cycles needed.

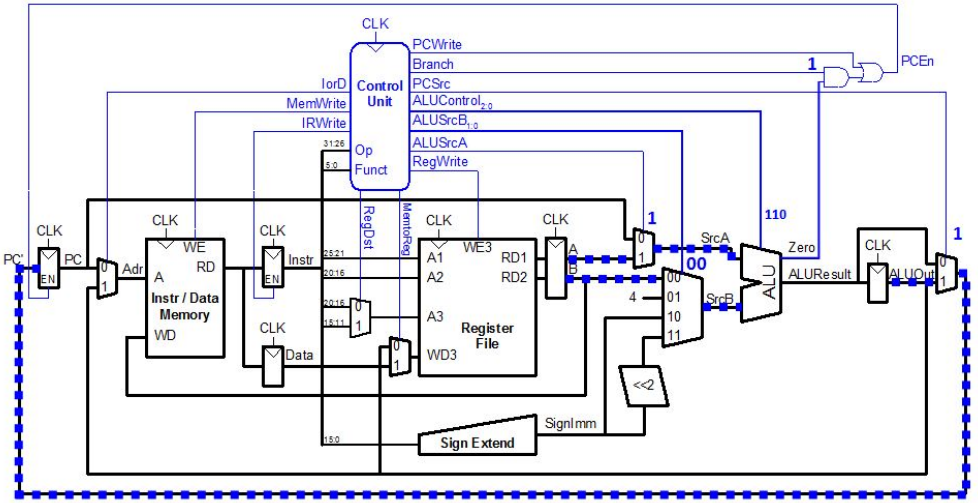


Main Controller FSM: cycles 2, 3 beq

- **Cycle-1:** ALU calculates PC+4
- **Cycle-2:** BTA is calculated (SrcA=0 and SrcB=11) and stored in ALUOut. Done in **all** instructions. If it is not a beq, ALUOut from cycle 2 will be just discarded.



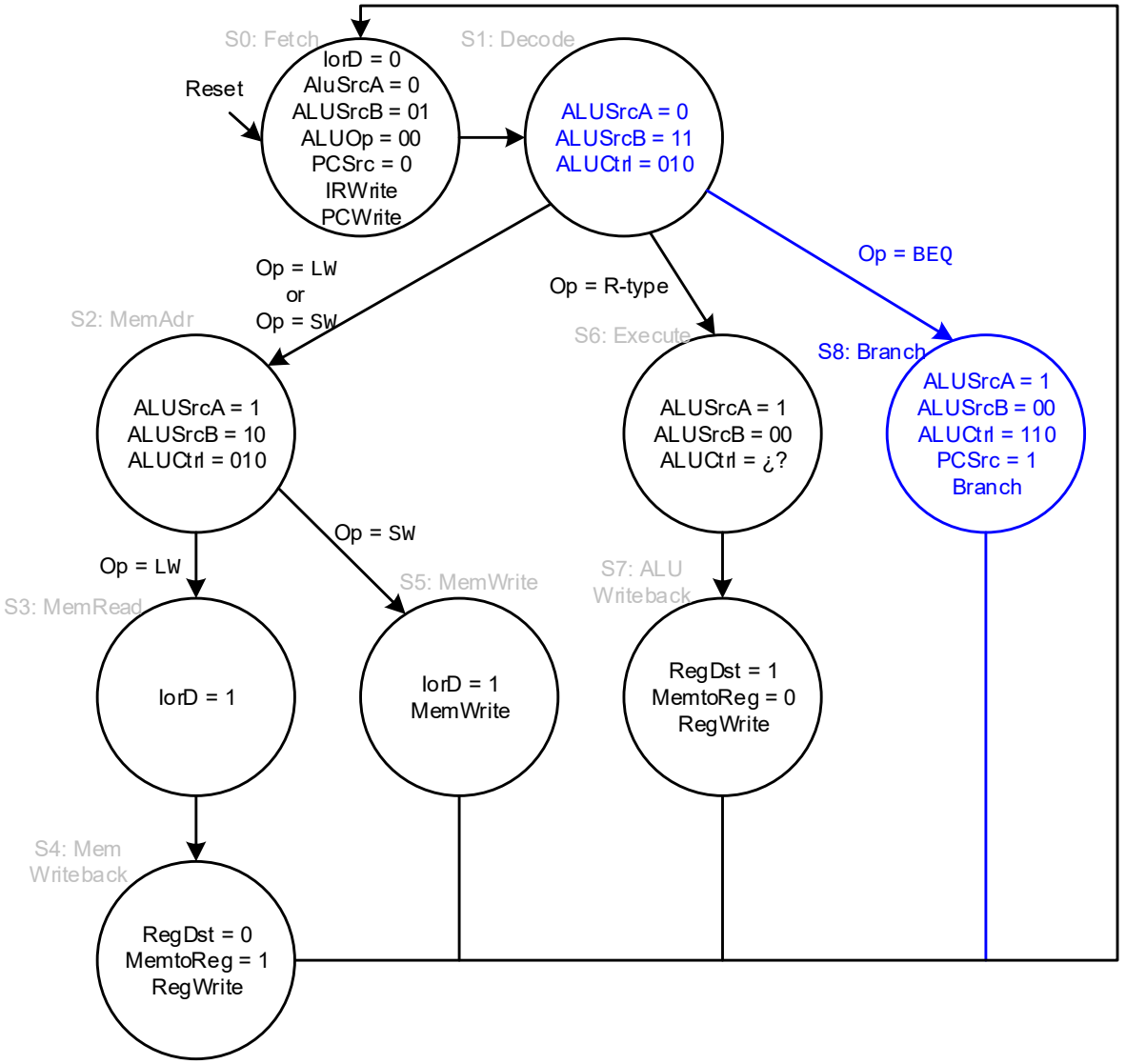
- **Cycle-3:** Branch signal active. The ALU subtracts the registers to be compared. Flag Z goes to '0' if not branching, to '1' if branching. PCEn=Z this cycle.



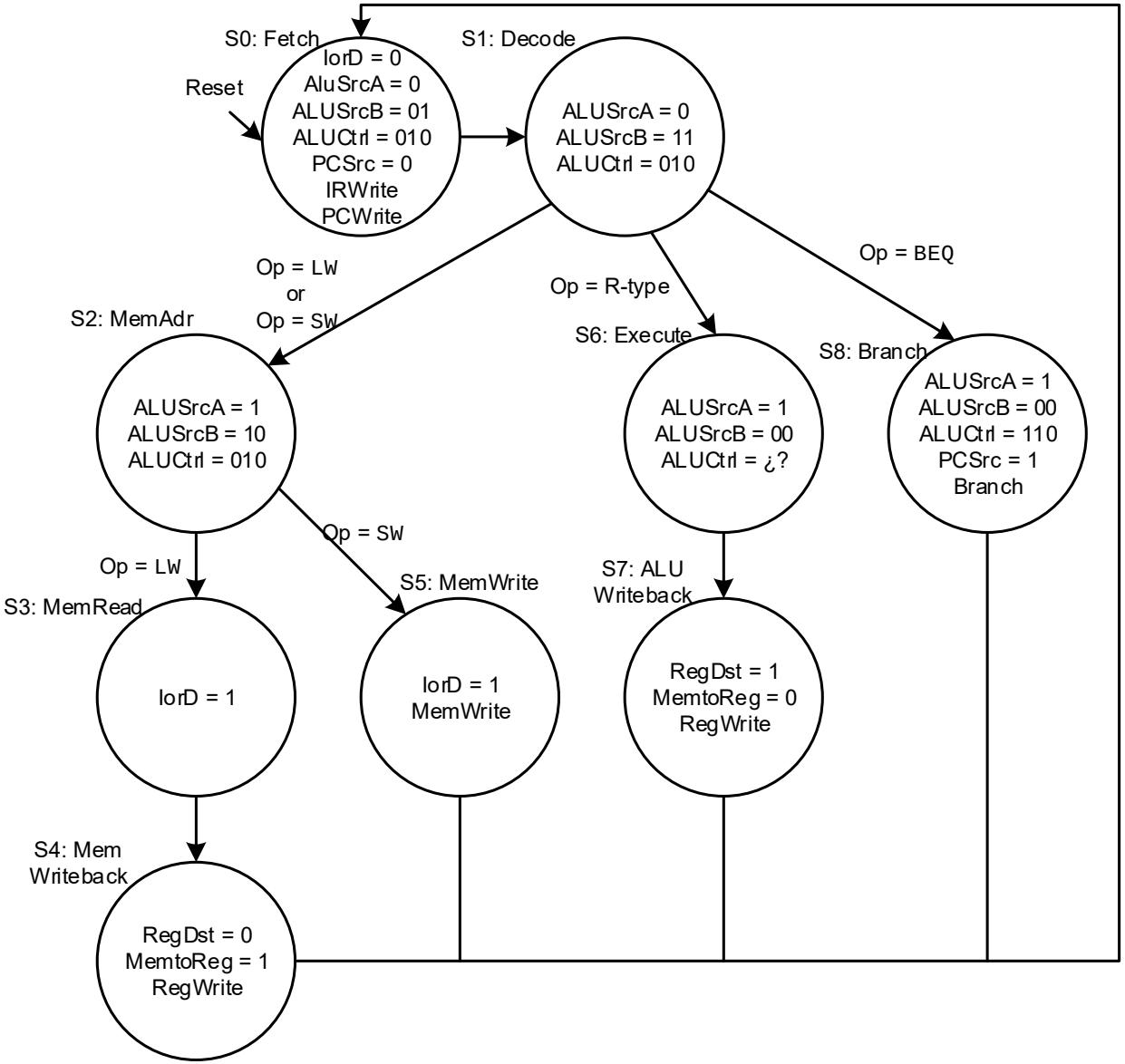
Main Controller FSM: cycles 2, 3 beq

✓Only 3 cycles used. Cycle 2 is made in all instructions, even in not beq instructions.

✓If branch is taken, PCEn will be at '1'. PC is updated using ALUOut, which has the BTA calculated in cycle 2.



Main Controller FSM: summary (without addi and j)



Instructions planning

P

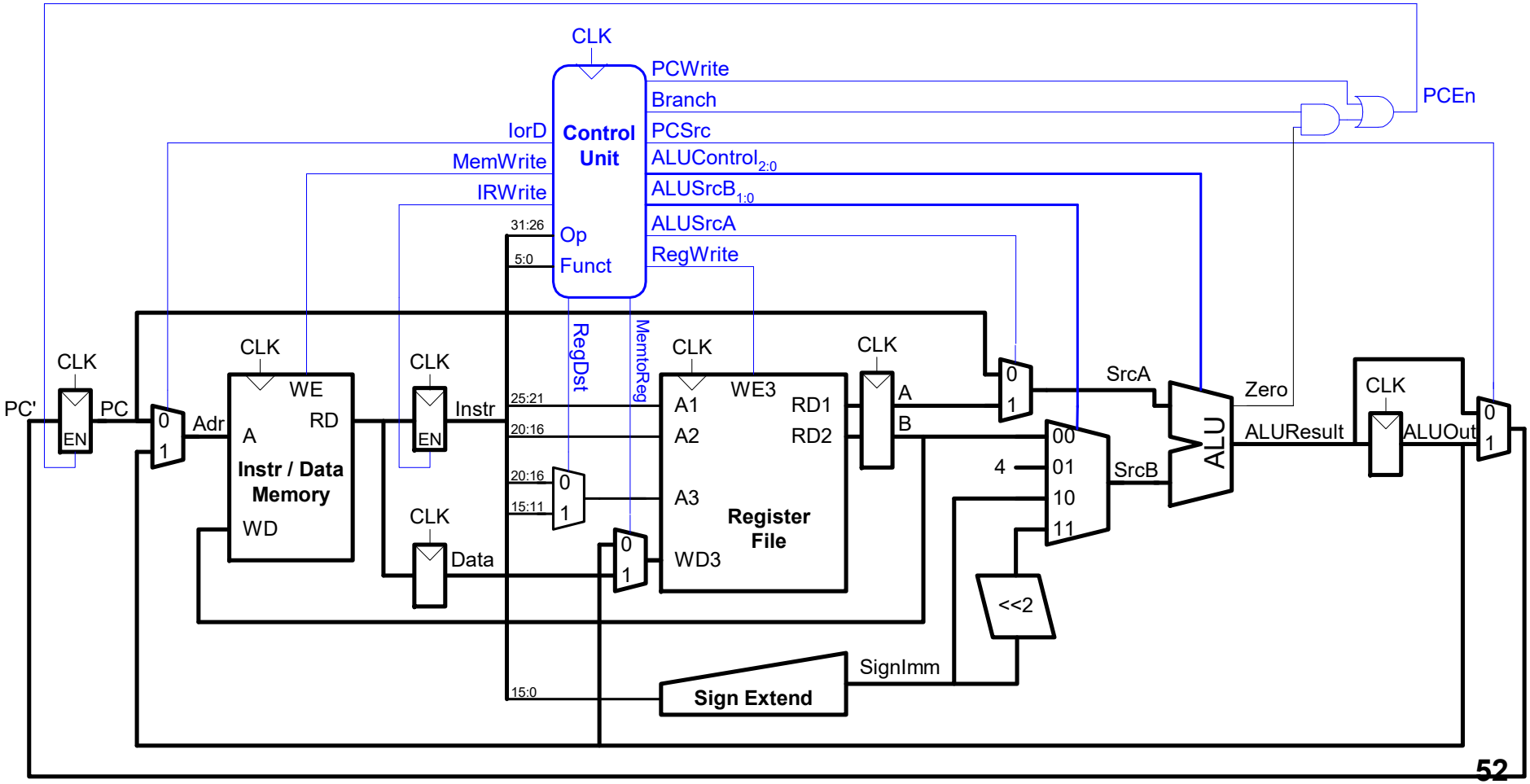
	lw	sw	R-type	beq
Cycle 1	Instruction fetch. PC <= PC + 4			
Cycle 2	**a.** Operands read. **b.** BTA calculated (not writting PC)			
Cycle 3	Data Memory address obtained		ALU operation	**a.** Subtract in ALU **b.** If Z=1, update PC
Cycle 4	Data Memory read	Data Memory write	Register write	
Cycle 5	Register write			

Outline

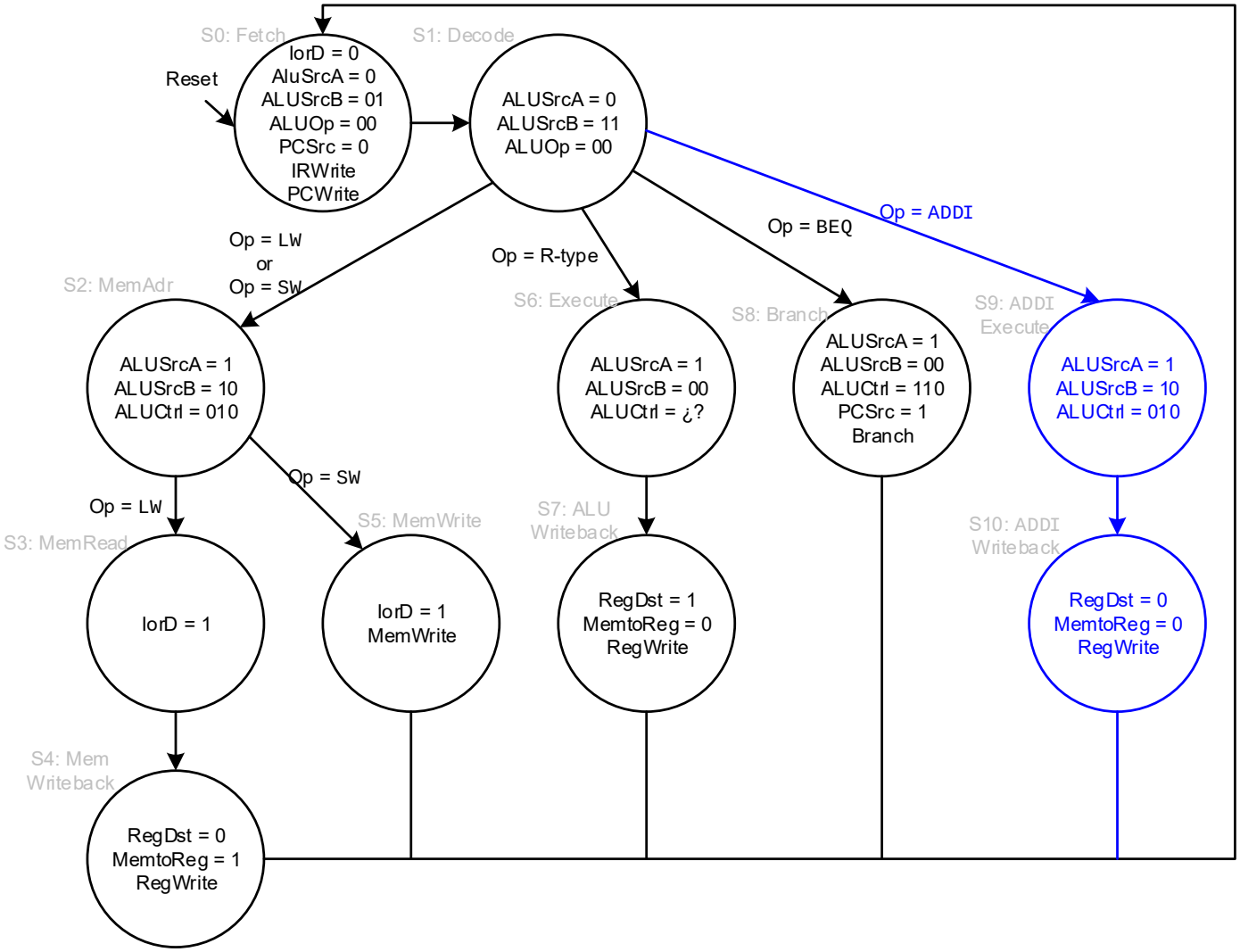
- Summary single-cycle MIPS
- Multicycle datapath
- Multicycle control path
- **Adding more instructions**

Adding addi

- No datapath changes (a register and an immediate can be added, as in lw, sw). Only control changes are necessary.



Control: changes for addi

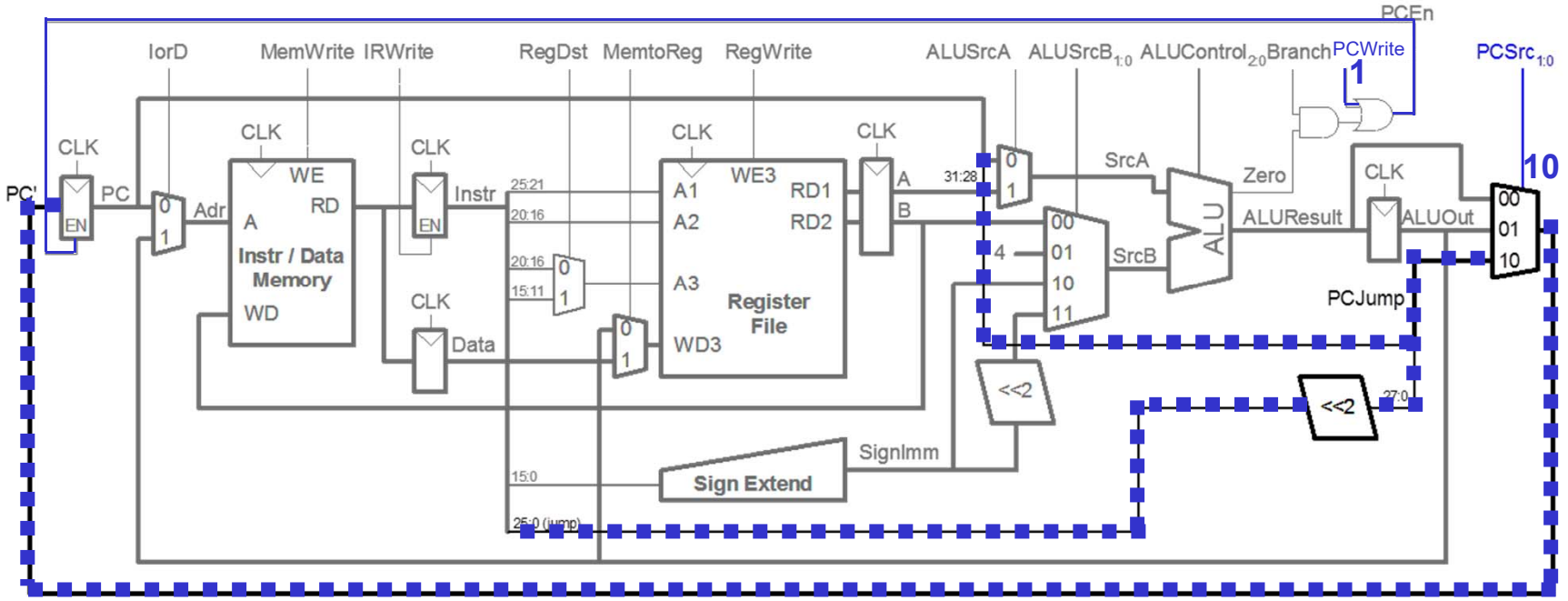


Adding j

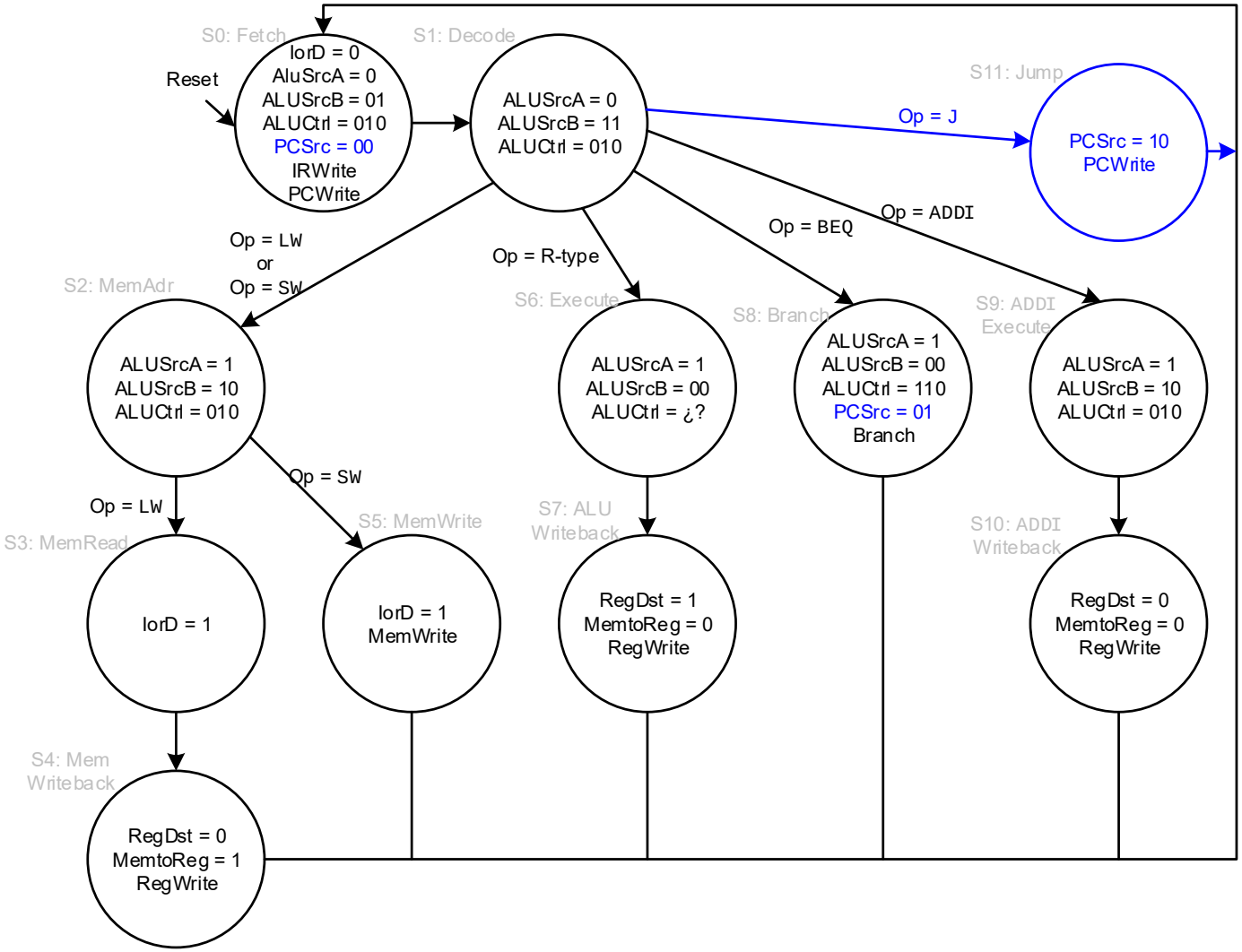
- Datapath needs changes in order to generate JTA
  JTA = (PC+4)[31:28] & addr & “00”

  Cycle-1 PC <= (PC+4)

- **Cycle-3:** addr shifted left 2. PCSrc becomes a two bits signal, since there are three possibilities to chose from.



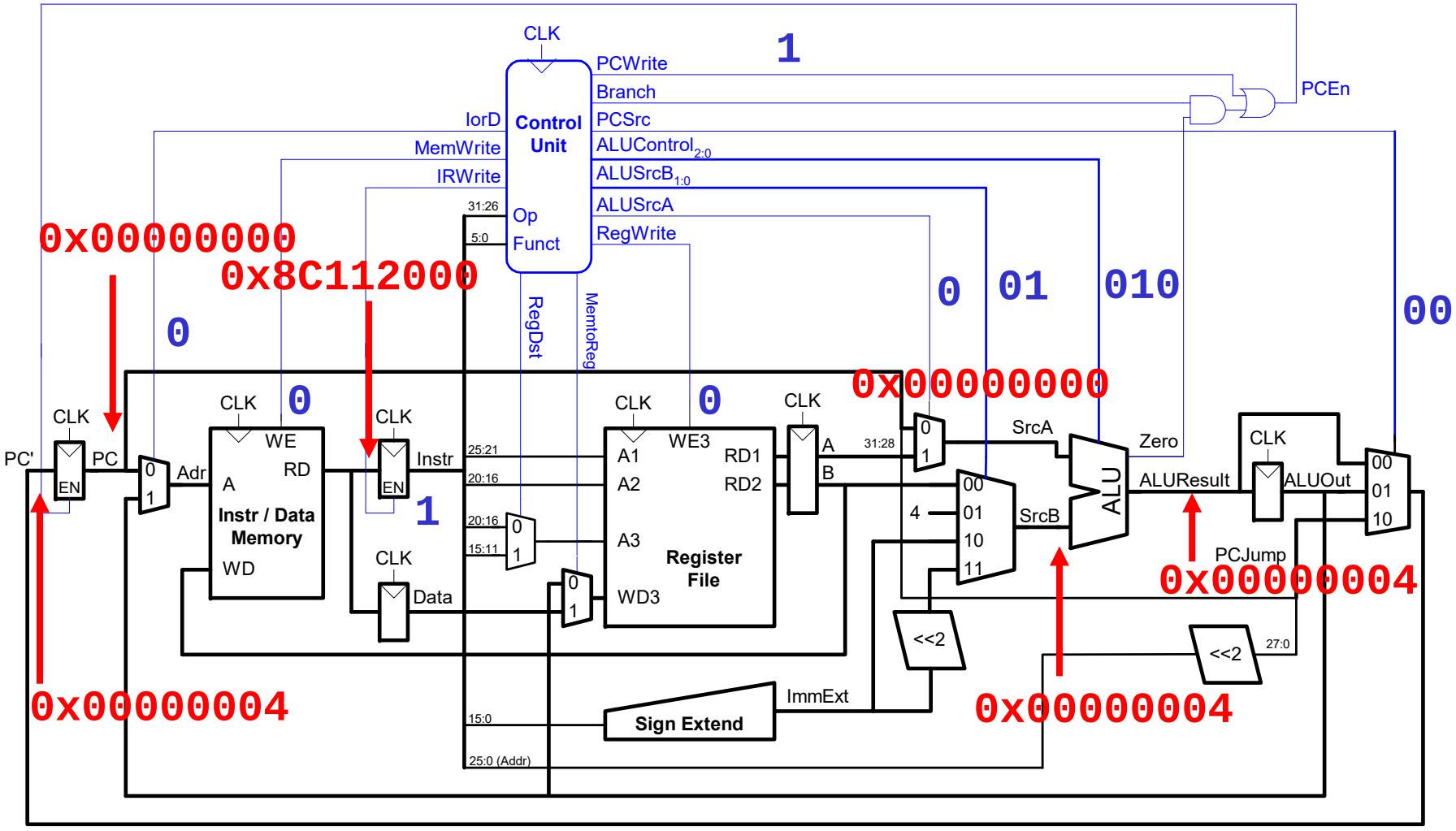
Control: changes for j



Instructions planning

	lw	sw	R-type	beq	addi	j
Cycle 1	Instruction fetch. PC <= PC + 4					
Cycle 2	a. Operands read. b. BTA calculated (not writting PC)					
Cycle 3	Data Memory address obtained		ALU operation	a. Subtract in ALU b. If Z=1, update PC	ALU operation	Jumps to JTA
Cycle 4	Data Memory read	Data Memory write	Register write		Register write	
Cycle 5	Register write					

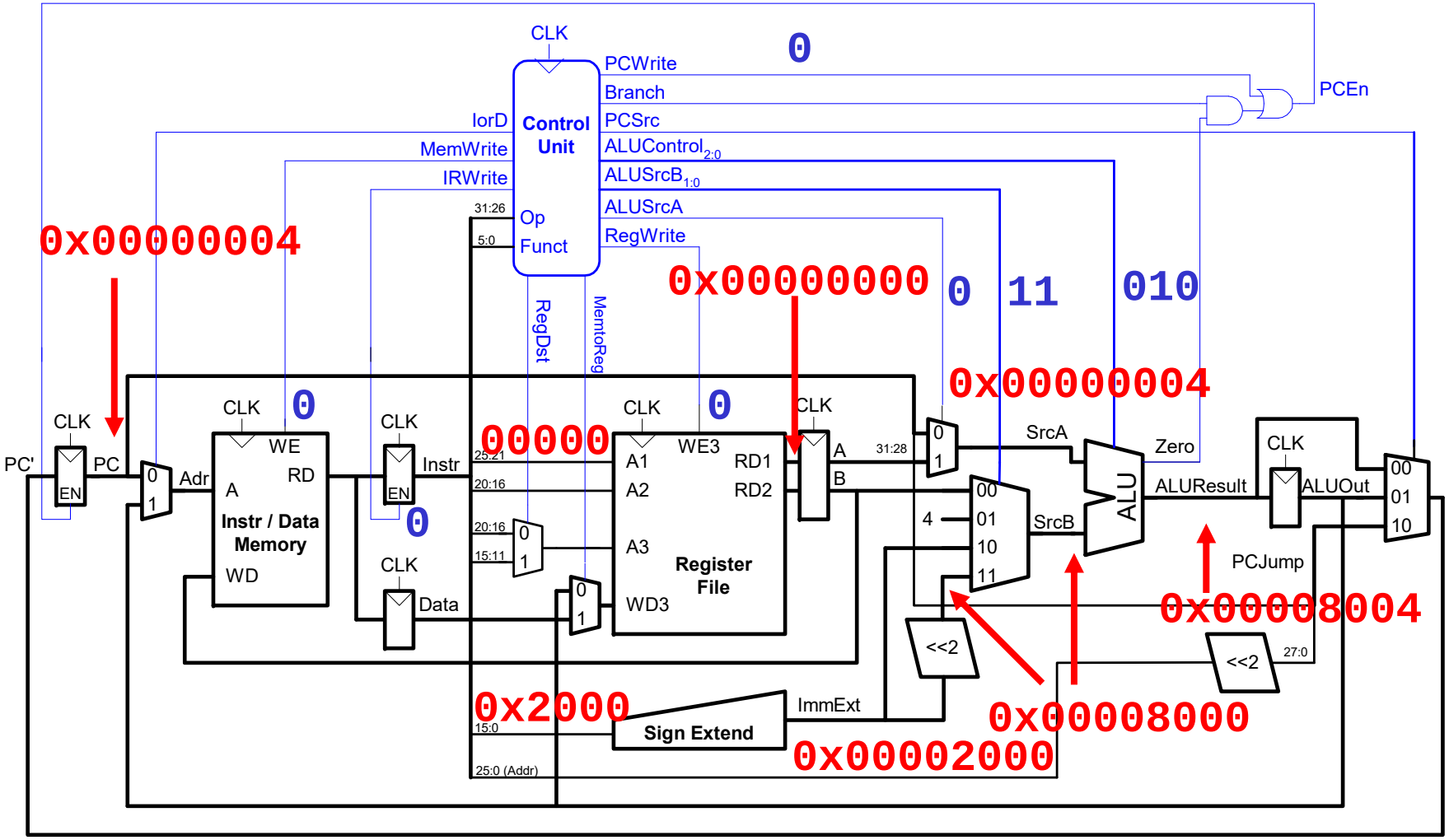
Muticycle datapath and control
lw example, cycle 1



MEM[0x00000000]=0x8C112000 => lw $s1,0x2000($0)

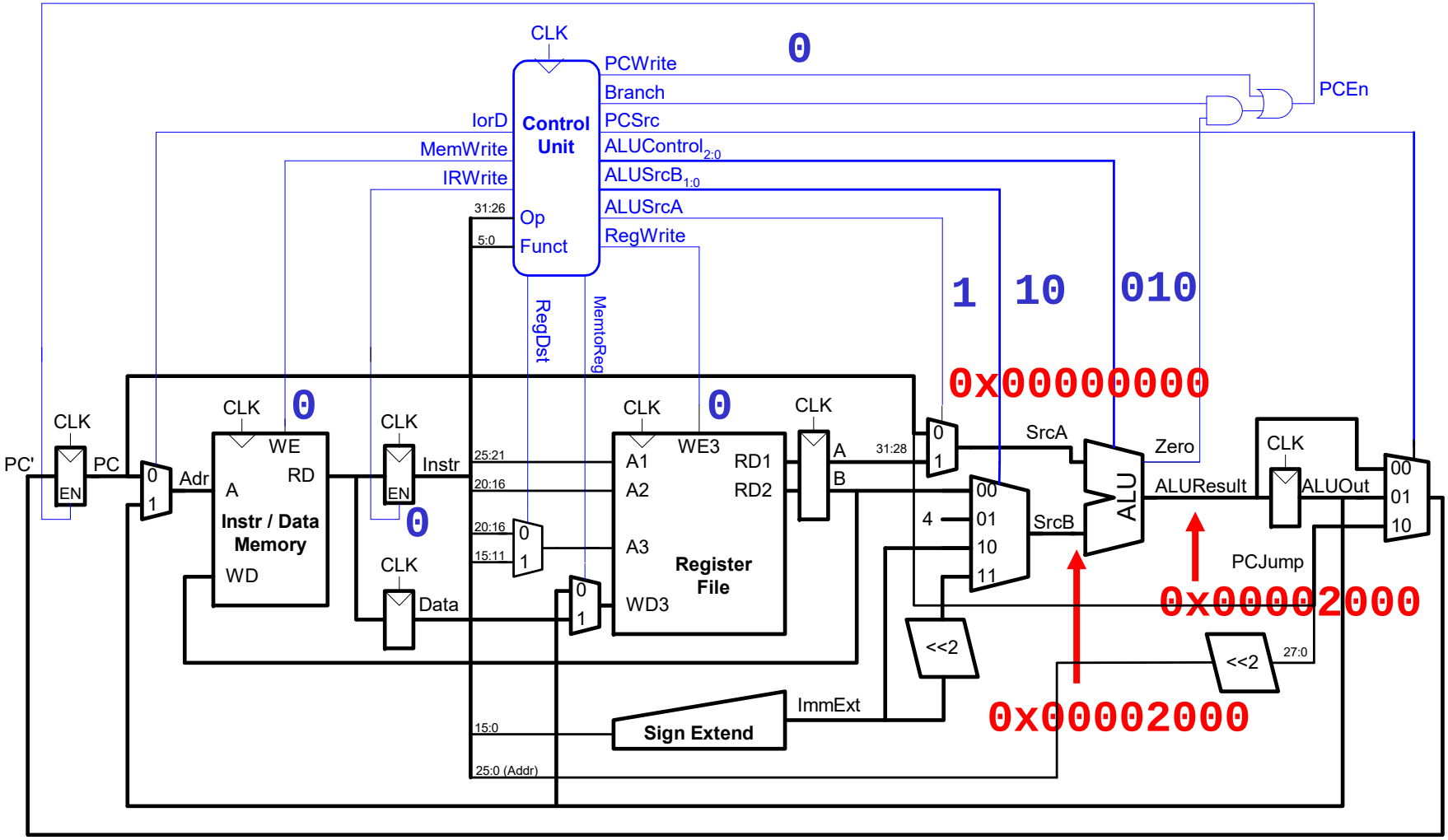
Supposing that MEM[0x00002000]=0x12345678

Muticycle datapath and control
lw example, cycle 2



MEM[0x00000000]=0x8C112000 => lw $s1,0x2000($0)
Supposing that MEM[0x00002000]=0x12345678

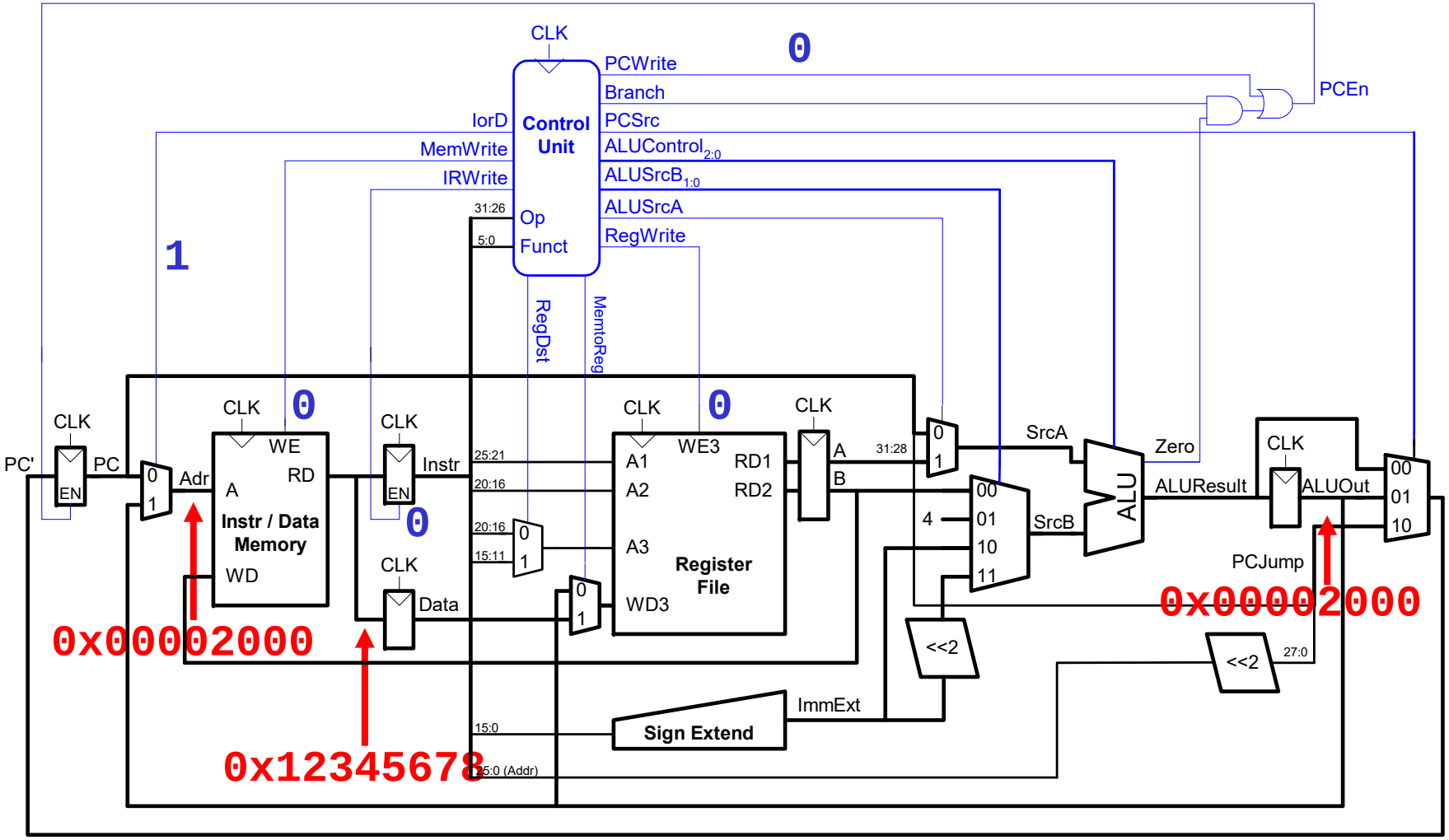
Muticycle datapath and control
lw example, cycle 3



MEM[0x00000000]=0x8C112000 => lw $s1,0x2000($0)

Supposing that MEM[0x00002000]=0x12345678

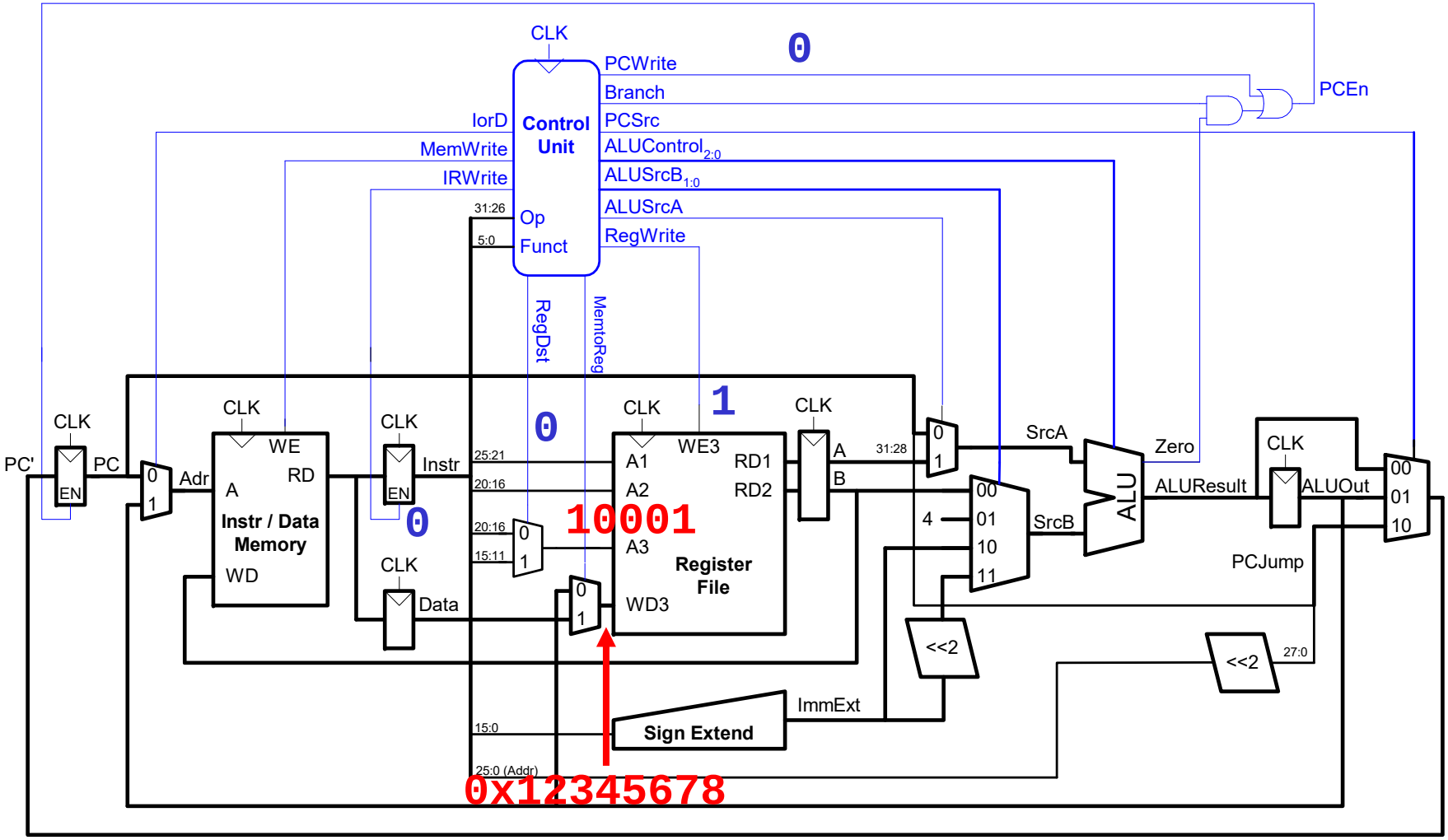
Muticycle datapath and control
lw example, cycle 4



MEM[0x00000000]=0x8C112000 => lw $s1,0x2000($0)

Supposing that MEM[0x00002000]=0x12345678

Muticycle datapath and control
lw example, cycle 5

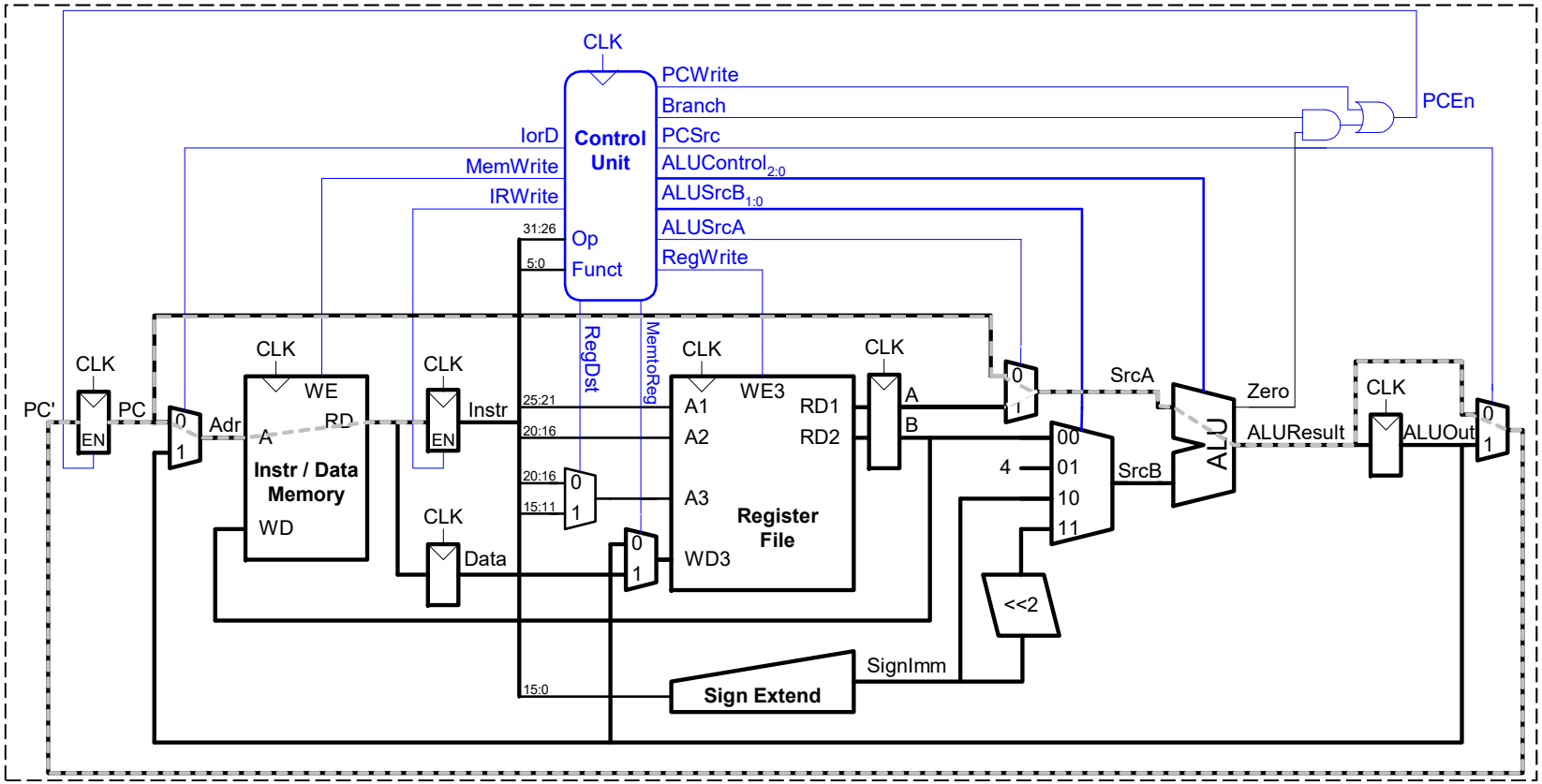


MEM[0x00000000]=0x8C112000 => lw $s1,0x2000($0)
Supposing that MEM[0x00002000]=0x12345678

Clock cycle in multicycle

- Set by the slowest clock path (cycle 1):

$$T_c = t_{pcq} + t_{mux} + t_{mem} + t_{setup}$$



Clock cycle in multicycle

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

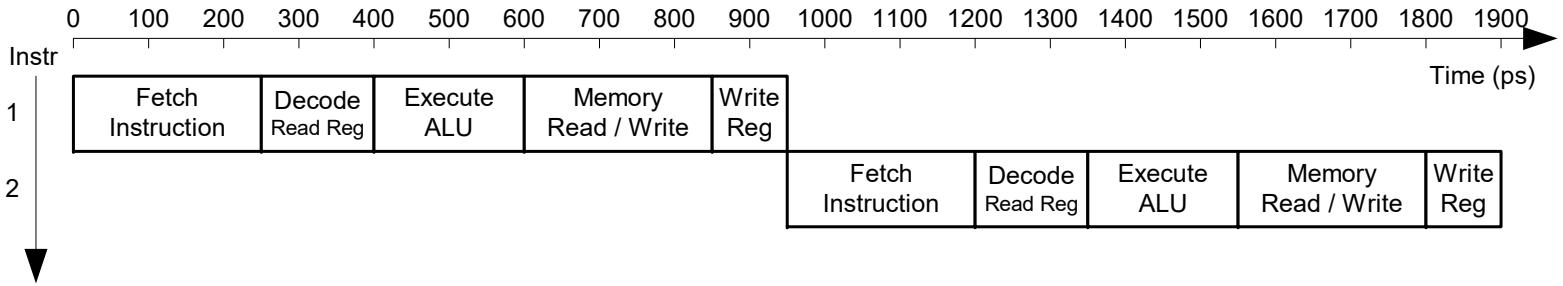
$$T_c = t_{pcq\_PC} + t_{mux} + t_{mem} + t_{setup} =$$
$$= [30 + 25 + 250 + 20] \text{ ps} = 325 \text{ ps}$$

Analysis: single-cycle vs multicycle

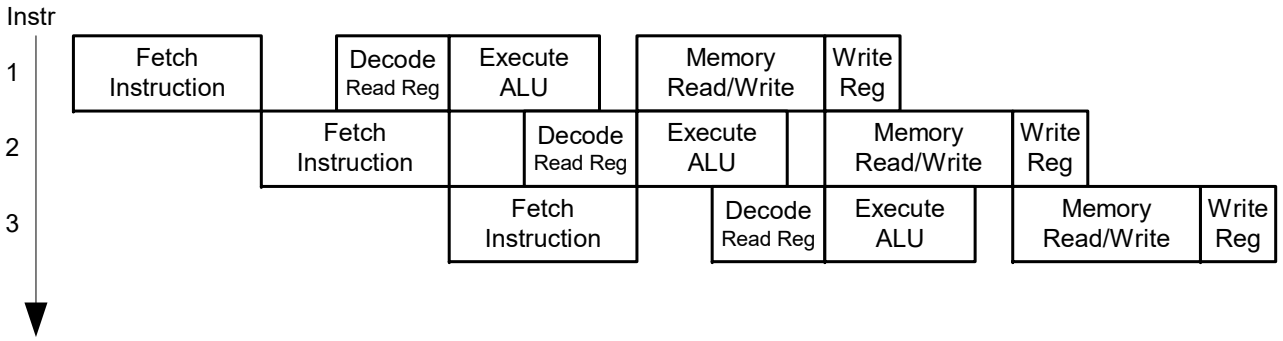
- In single-cycle, all instructions use the same time as the slowest instructions (lw), 925 ps.
- In multicycle, all clock cycles have the period of the slowest cycle (cycle 1), 325 ps.
- In muticycle, each instruction needs a different number of cycles, between 3 and 5. The execution time is between 3·325 and 5·325, that is, 975 and 1625 ps.
- Multicycle is slower than single-cycle (in this example): the reason is that the clock cycles have not been equally divided (equal clock cycles would have been 925 ps/5 = 185 ps).
- However, multicycle is the base of *pipeline*, where every "short" clock cycle a new instruction starts, executing several instructions in parallel.

Single-cycle vs Pipelined

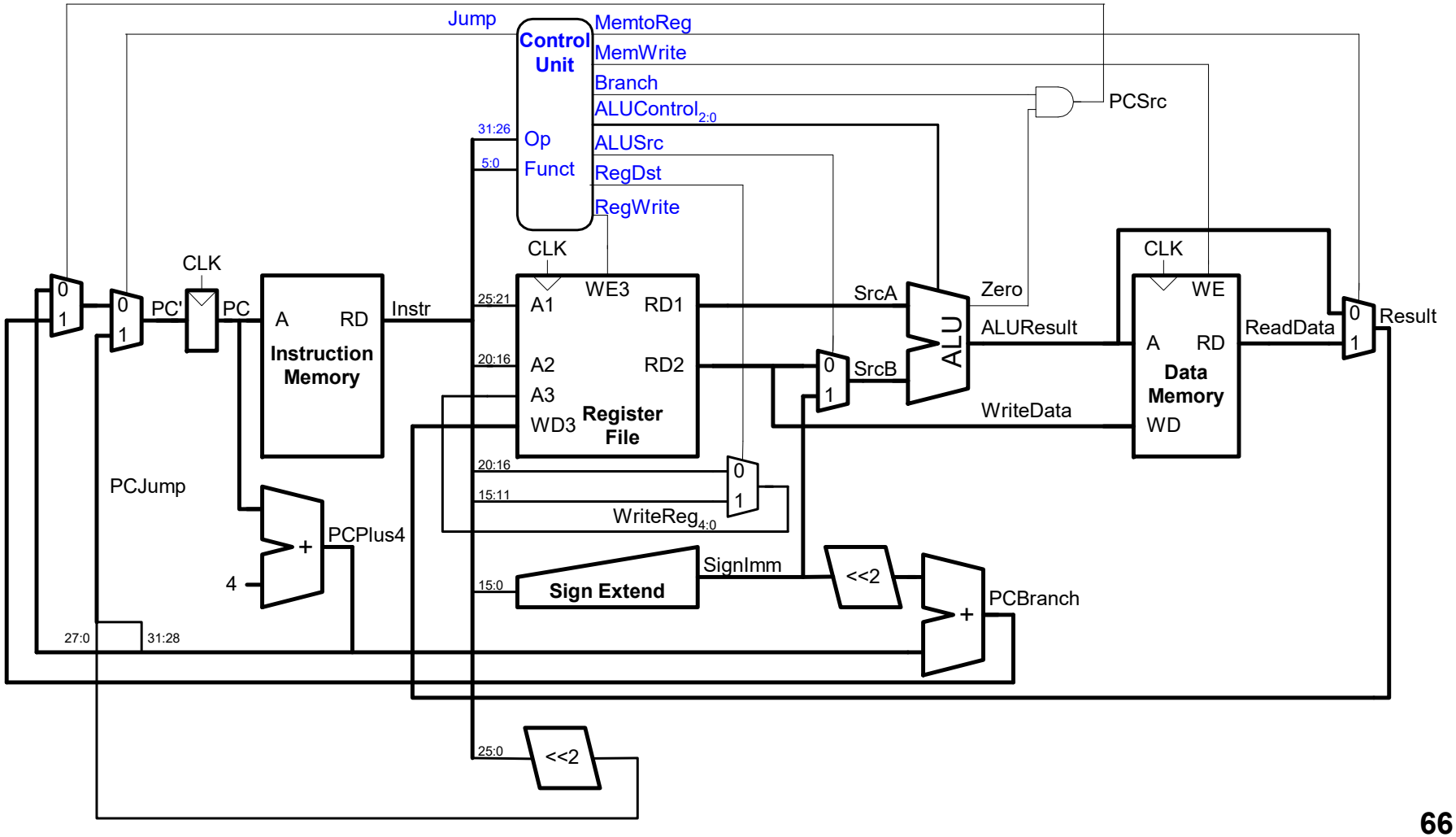
Single-Cycle



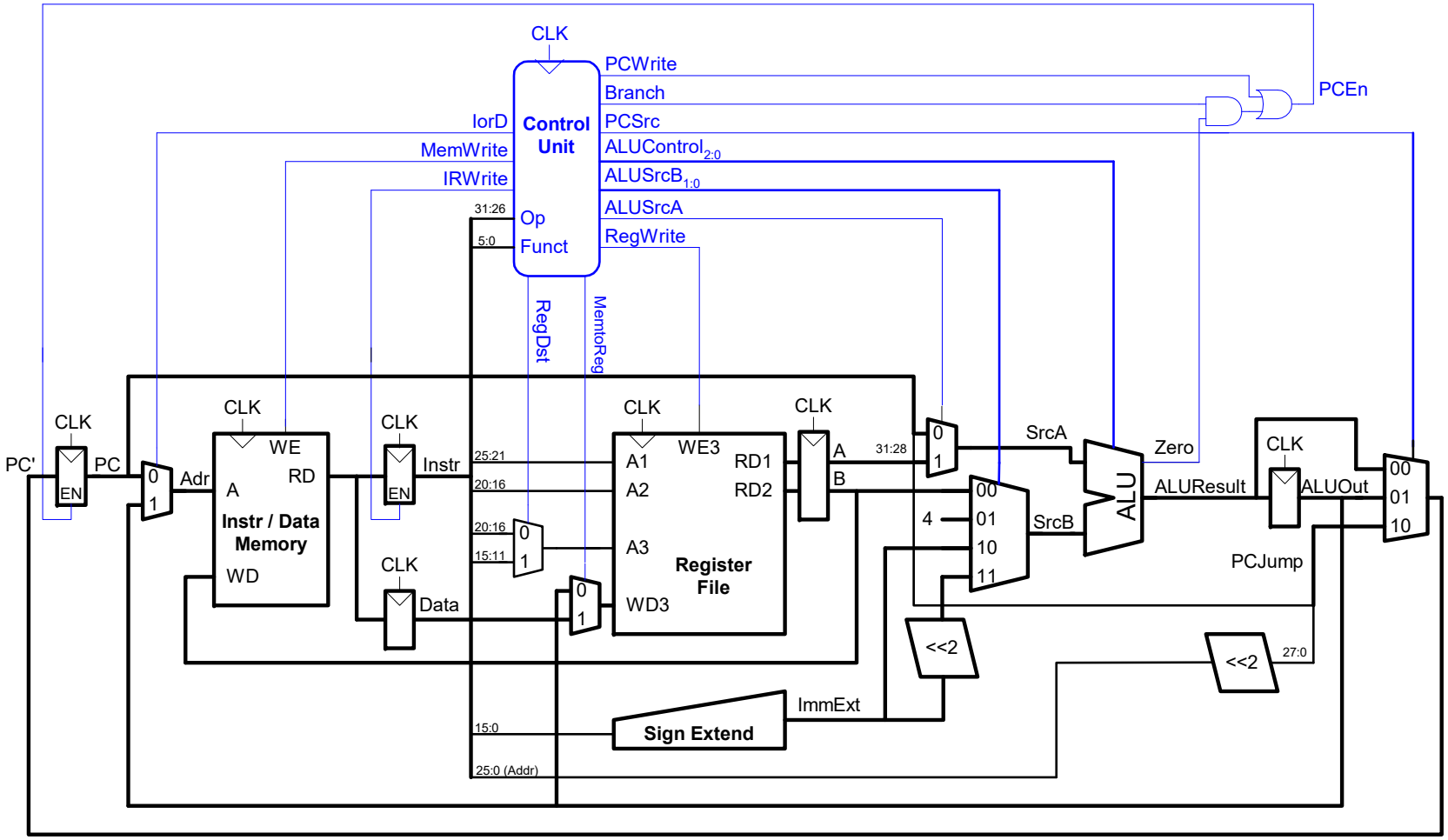
Pipelined



Summary: single-cycle MIPS



Summary: muticycle MIPS



Unit 5. Processor III: Design and control of the datapath. Multicycle architecture

Escuela Politécnica Superior - UAM

Exercises U5

T

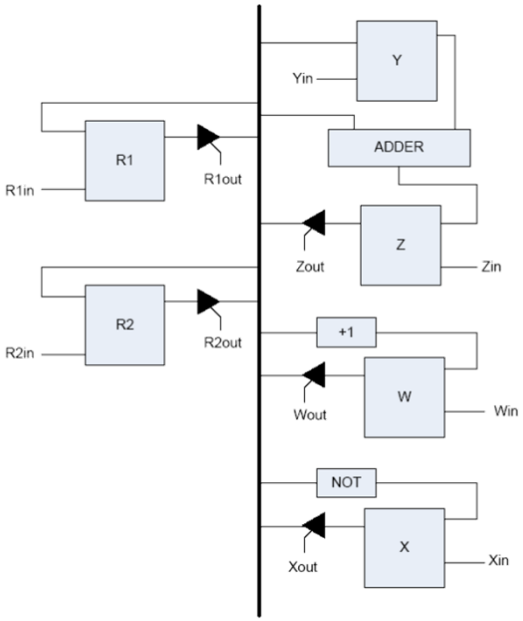
5.1. Using the architecture of the figure, which are the clock by clock operations and control signals to execute the following high-level instructions:

a) R1 ← R1 - R2 + 1

b) R2 ← 2*R2 + 2*R1 + 2

Describe the operation performed each clock cycle, indicating the data flow between registers (RTL description, Register Transfer Level).

Note: for each register there is a write enable (active high): Yin, Zin, Win, Xin, R1in and R2in. There are also read enables, signals XXout, which enable the three-state buffer when at '1'.



Exercises U5

T

5.3. The following code is executed in a muticycle MIPS processor. The memory addresses, as known, are 32-bits wide.

a) Using the opcode/funct tables, translate the instruction "bne $s1, $0, lab" into machine code.

b) Translate the machine code 0xAC112010 into assembly code. Also indicate which is the functionality of this instruction, taking into account its operands.

c) Using the schematic of the muticycle processor, fill in the following table with the control signals values in each clock cycle. Fill the table with all the cycles necessary to execute the first two instructions of the code.

Also write the value of the following registers after executing the first three instructions of the code ($pc, $t1, $t2, $t3).

RAM CÓDIGO / DATOS	
.text 0 lab: lw $t2, X($0) sllv $t3, $t2, $t2 addi $t1, $t3, 0x200C and $t4, $t1, $t2 xor $s1, $t4, $t4 bne $s1,$0, lab 0xAC112010 fin: j fin	.data 0x2000 .space 8 X: 0x0002 Y: 0x00FF Z:

Control signals										
IorD	IRWrite	RegDst	MemWrite	MemtoReg	RegWrite	PC_Write	Branch	ALUScrA	$ALUScrB_{(1:0)}$	$PCSrc_{(1:0)}$

Exercises U5

5.20. A MIPS muticycle processor executes the following code. Please, fill in the following table for the nine first clock cycles, which includes some registers and signals. In cycle 1 the first clock cycle of the first instruction is being executed.

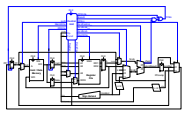
RAM CÓDIGO/DATOS	

.text 0x0000
lw $s2, 0x2000($s1)
beq $s2, $s3, fin
sw $s2, 0x2000($0)
fin: j fin
	.data 0x2000
	X: 0x0A
	Y: 0x0C

Initial registers values: $s1 = 0x04; $s2 = 0x08; $s3 = 0x0C

Registers						
$pc:	$s1:	$s2:	$s3:	A:	B:	ALUOut:

Control signals		
RegWrite	ALUScrA	$ALUScrB_{(1:0)}$



Exercises U5

5.22. The following schematic shows the datapath of a muticycle processor called ARC (A Risc Computer), including the access to the only memory (code and data memory). The size of both instructions and data is 32-bits. The registers of the schematic, from left to right, are the instruction register (ir), the register file or general purpose registers (GPR), the temporal registers (rtA and rtB), and the accumulator register (rtAcc) at the ALU output. The control signals are also included in the schematic.

The following ARC instruction adds a register and an immediate storing the result in other register:

addcc %r10, - 100, %r15 (equivalent in MIPS to addi $15, $10, - 100).

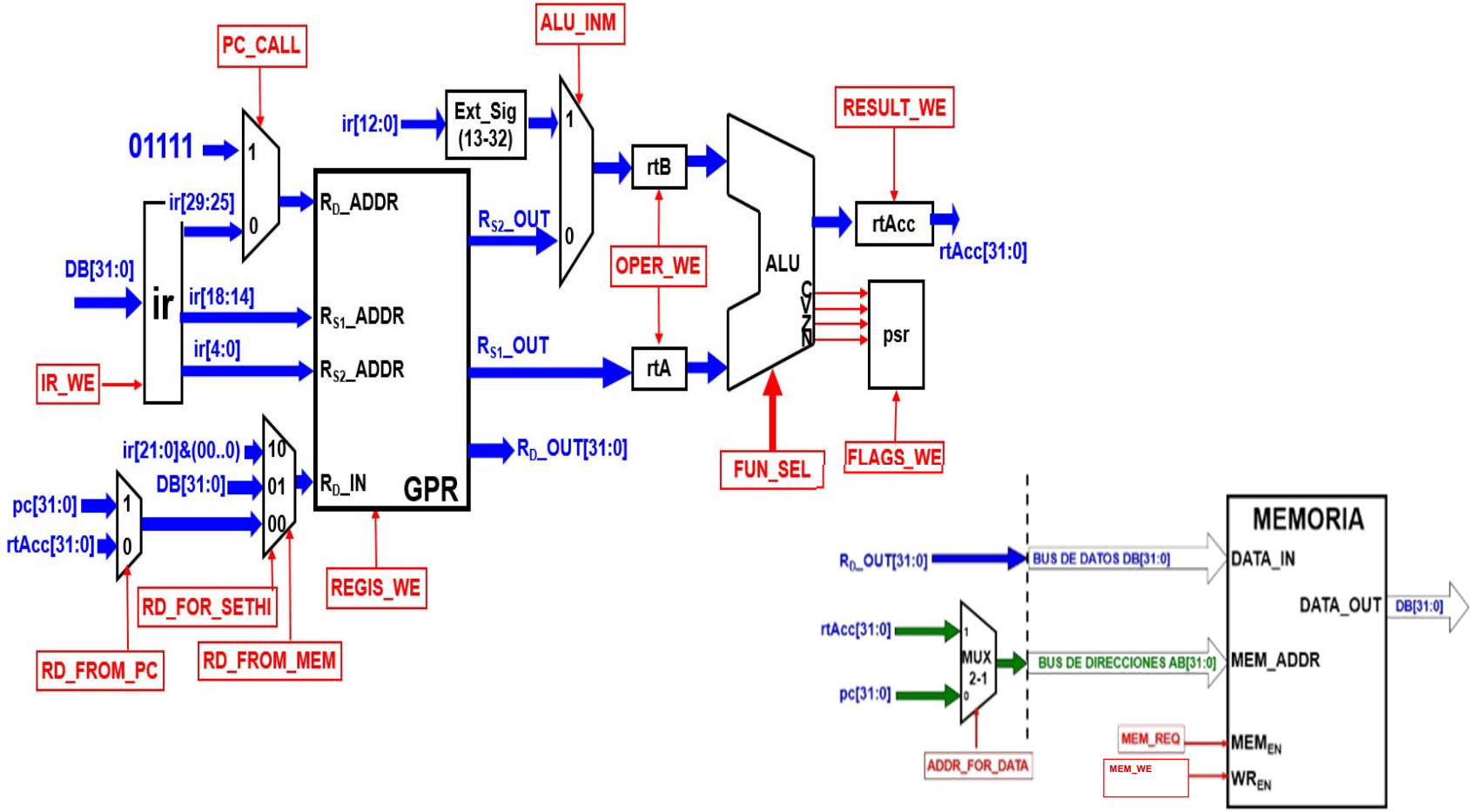
a) The machine code of the previous instruction. Group the bits in the known fields and indicate with the value 'X' those bits of which there is no information (which correspond to opcode information).

b) The multicycle ARC processor uses 4 clock cycles for this instruction. Please, fill in the following table with the values of the control signals each clock cycle. Signals ended in _WE are write enables of the corresponding circuit. The rest of the signals meaning can be deduced using the schematic. Fill in using '0', '1' or 'X' as needed each cycle. Note: the PC datapath has been omitted, so it does not need to be updated.

Signals	MEM_WE	ADDR_FOR_DATA	IR_WE	PC_CALL	ALU_INM	OPER_WE	RESULT_WE
	RD_FROM_PC	RD_FOR_SETHI	RD_FROM_MEM	REGIS_WE			

Exercises U5

5.22.



Exercises U5

5.5. We want to modify the datapath of the multicycle MIPS in order to allow executing the instruction to swap the content of two general purpose registers. The assembly code of the instruction would be: **swap $0, $X, $Y** and also **swap $X, $Y**; where $X and $Y are the registers to be swapped. This new instruction is an R-Type one with funct = “110000”. It must be executed in 4 cycles including the fetch cycle. The RTL (register transfer level) description of what is donde in each cycle is included next.

①	**[Instr] <= MEM [pc] ; [pc] <= [pc] + 4**
②	**[A] <= [rs]; [B] <= [rt]**
③	**[rt] <= [A]**
④	**[rs] <= [B]**

a) Write the machine code of this instruction (use X and Y with any two values).
b) Indicate the necessary changes in the schematic of the multicycle processor in order to be able to execute the instruction (new elements and/or paths).
c) Indicate the new control signals for this instruction, including how and when to use them.

Exercises U5

5.17. In a multicycle MIPS processor, during the clock cycle T the processor is executing the first cycle (fetch) of one of the instructions of the following code. In this code, the parameters are exchanged between caller and callee using the specific registers ($ax y $vx).
Using the given information in the control signals table, identify the current instruction and the two previously executed instructions.

a. Give the hexadecimal values of **pc** in the cycles T-1, T and T+1.

b. Fill in the control signals table when executing the current instruction (use as many columns as necessary).

c. Fill in the hexadecimal values of the following register and memory position after completely executing the current instruction. Register $ra and MEM[R].

d. Registers A and B are used to store the ALU operands. Knowing that the machine code of the instruction **jr $ra**, **at the end of the procedure mulx4**, is 0x03E00008, please, fill in the hexadecimal values of register A and B after the clock edge of the two last cycles of the instruction being executed (jr $ra in mulx4).

Exercises U5

5.17.

Cycle	T-7	T-6	T-5	T-4	T-3	T-2	T-1	T
PCWrite	1	0	0	0	1	0	1	1
IorD	0	X	X	1	0	X	X	0
IRWrite	1	0	0	0	1	0	0	1
MemWrite	0	0	0	1	0	0	0	0
RegWrite	0	0	0	0	0	0	0	0
MemtoReg	X	X	X	X	X	X	X	X
RegDst	X	X	X	X	X	X	X	X
ALUSrcA	0	0	1	X	0	0	1	0
ALUSrcB$_{(1:0)}$	01	11	10	XX	01	11	XX	01

RAM CODE / DATA

```
.text 0x0000
main:   lw $s1,
   Num($0)
        add $a0, $s1,
   $0
        jal sub1
        lw $s1, R($0)
fin:    j fin
----------------------------
----------------------------
.data 0x2000
Num:    0x00000064
R:      0x00000000
```

```
.text 0x0100
sub1:   addi $sp, $sp, -4
        sw $ra, 0($sp)
        addi $t1, $0, 100
        beq $a0, $t1, etiq
        jal mulx8
        j ret
etiq:   jal mulx4
ret:    lw $ra, 0($sp)
        addi $sp, $sp, 4
        sw $v0, R($0)
        jr $ra

.text 0x0200
mulx4:      sll $v0, $a0, 2
            jr $ra

.text 0x0300
mulx8:      sll $v0, $a0, 3
            jr $ra
```

Exercises U5

5.12. The following code is executed in a muticycle MIPS processor:
Fill in the following table with the indicated registers and signals values from the beginning (first cycle) of the instruction labelled “salto” until completing the program. Also fill in the final content of the memory positions A, B and C.

RAM CODE/DATA	

.text 0x0000
lw $s1, 5($s2)
and $s2, $s1, $s3
beq $s1,$s2,salto
sw $s2, -4($s3)
j fin
.text 0x001C
salto: jal fin
or $s3,$s1,$s2
fin: and $s2,$s1,$s3
sw $s3,B($s2)
	.data 0x2200
	A: 0x00000002
	B: 0x00002200
	C: 0X0000EA17

Initial values: $s1 = 0xA4; $s2 = 0x02; $s3 = 0x4C y $ra = 0x00

Registers					**Memory**		
$pc:	$s1:	$s2:	$s3:	$ra:	A:	B:	C:

Control signals										
IorD	IRWrite	RegDst	MemWrite	MentoReg	RegWrite	PC_Write	Branch	ALUScrA	$ALUScrB_{(1:0)}$	$PCSrc_{(1:0)}$

Summary: muticycle MIPS