Microarchitecture

University of South Carolina

Introduction to Computer Architecture
Fall, 2024
Mehdi Yaghouti

Molinaroli College of Engineering and Computing
UNIVERSITY OF SOUTH CAROLINA

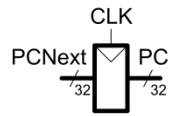
Microarchitecture

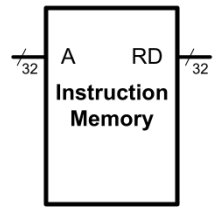
- A computer architecture is defined by its instruction set and architectural state
- RISC-V architectural state: the program counter and the 32 X 32-bit registers
- Architectural state and memory hold all the needed information to resume
- Microarchitecture is the connection between logic and architecture
- Microarchitecture is the arrangement of:
  - Registers
  - Arithmetic Logic Units (ALUs)
  - Finite State Machines (FSMs)
  - Memories
  - Logic building blocks

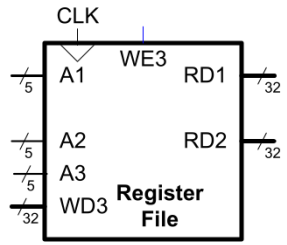
  needed to implement a specific architecture
- Microarchitecture can be considered to be composed of *datapath* and *control unit*
  - *datapath* is the path in which the data flow between different components
  - *control unit* controls the datapath based on the current instruction at hand

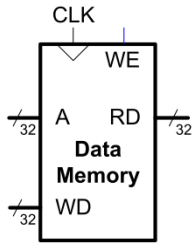
Main Components

- Program counter (PC) points to the current instruction
- PCNext, indicates the address of the next instruction
- Instruction memory takes a 32-bit address A and reads the data
- Register file has two read ports and one write port
- Read ports take 5-bit address inputs, A1 and A2
- Register file places the 32-bit register values onto RD1 and RD2
- It takes a 5-bit address input A3 and a 32-bit write data WD3
- If WE3 is asserted, the data WD3 will be written into address A3
- Writing happens at the clock edge
- Data memory has a single read/write port
- It reads data at address A and places it into RD
- If WE is asserted it writes WD into address A at the clock edge



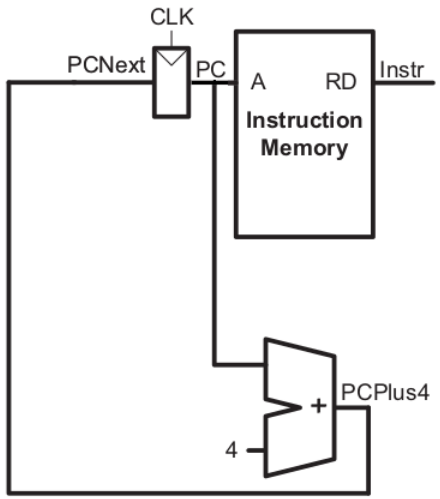


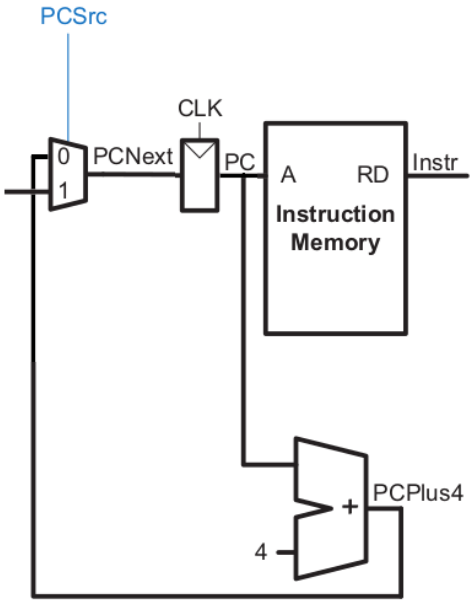




Fetch

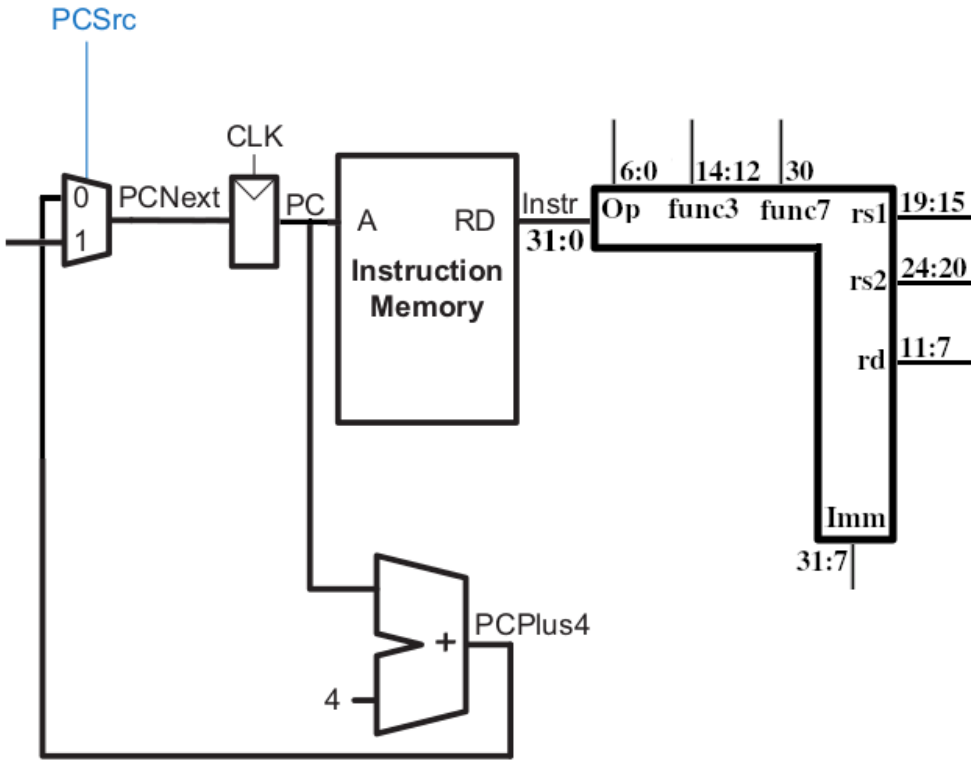
- PC is the current instruction address
- Each word is 32 bits
- $PCNext \leftarrow PC + 4$
- PC only changes at the clock transition
- Branch instructions need to set the $PCNext$





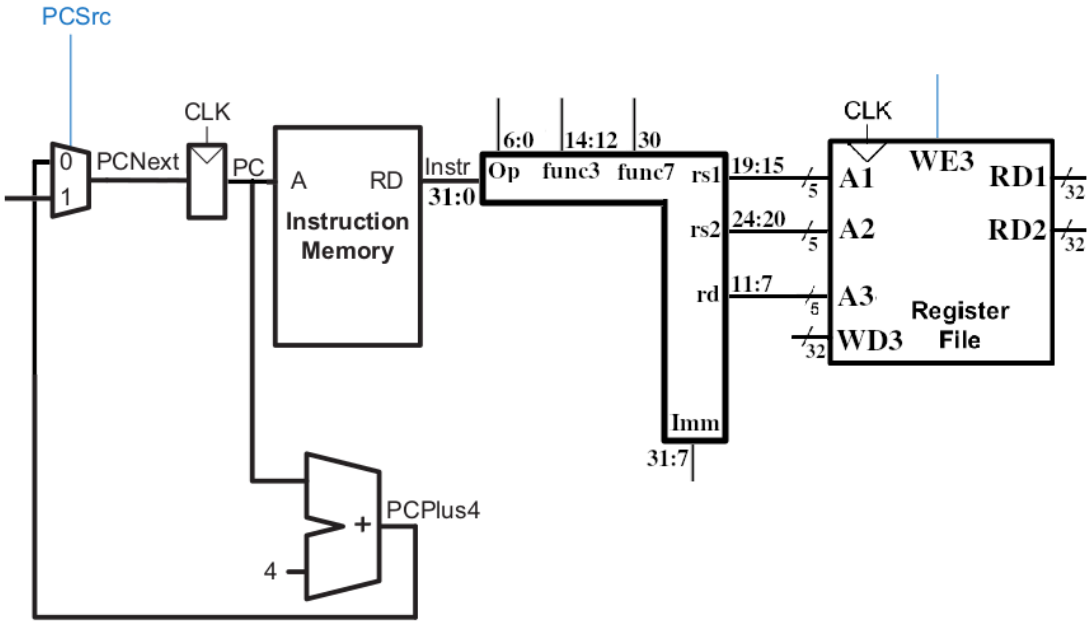
Bitfield Extraction

- PC is the current instruction address



Registers

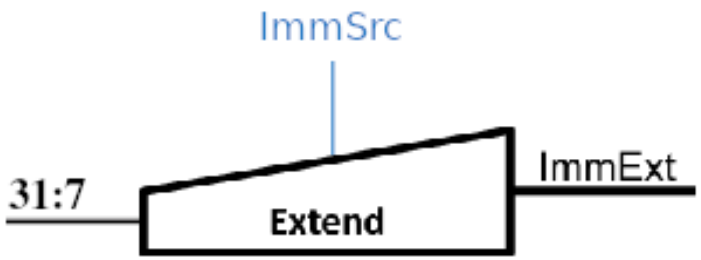
- PC is the current instruction address



Extension Unit

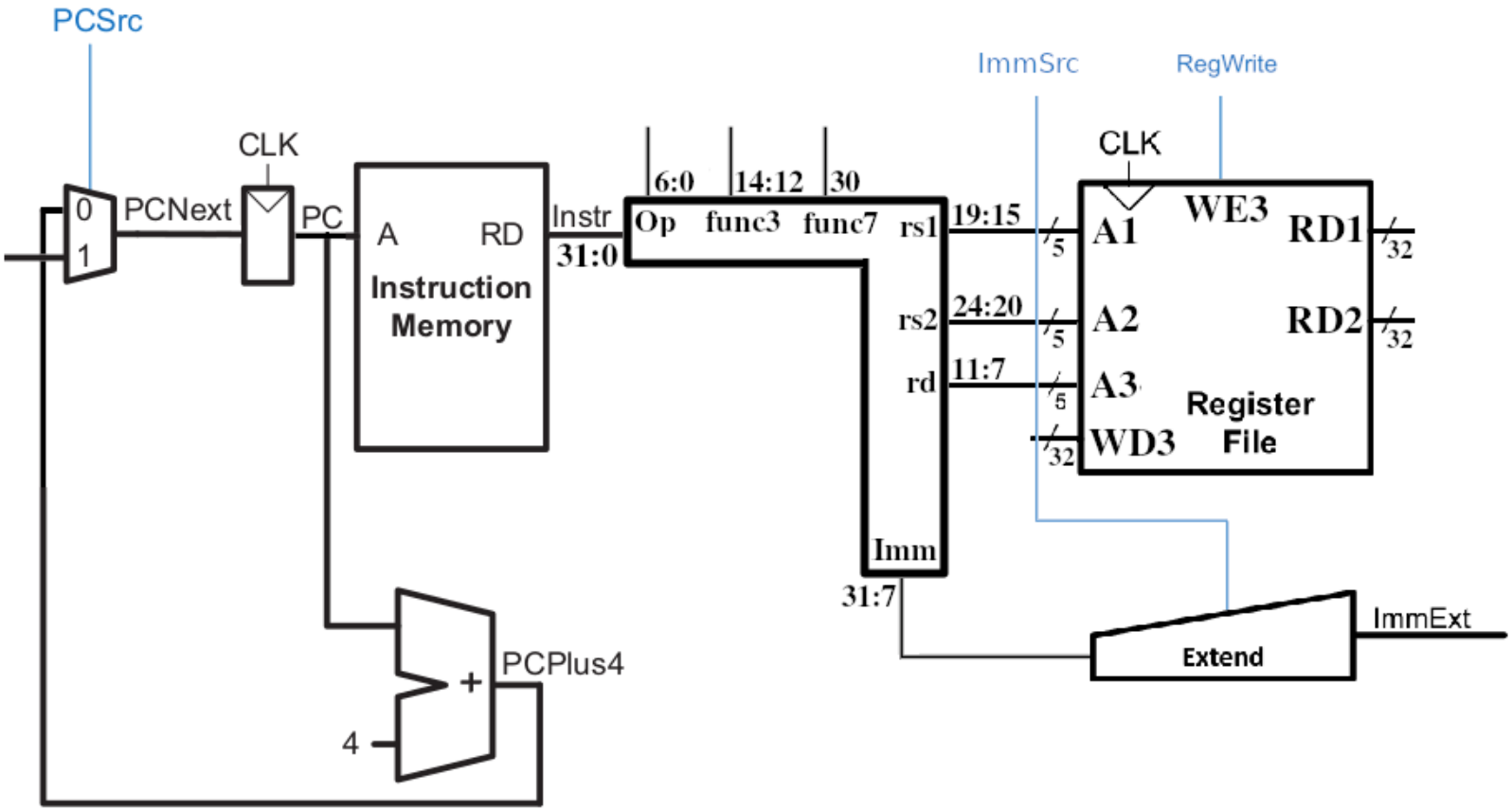
- PC is the current instruction address

ImmSrc	ImmExt	Type	Description
00	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-bit signed immediate
01	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-bit signed immediate
10	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1'b0}	B	13-bit signed immediate
11	{{12{*Instr*[31]}}, *Instr*[19:12], *Instr*[20], *Instr*[30:21], 1'b0}	J	21-bit signed immediate



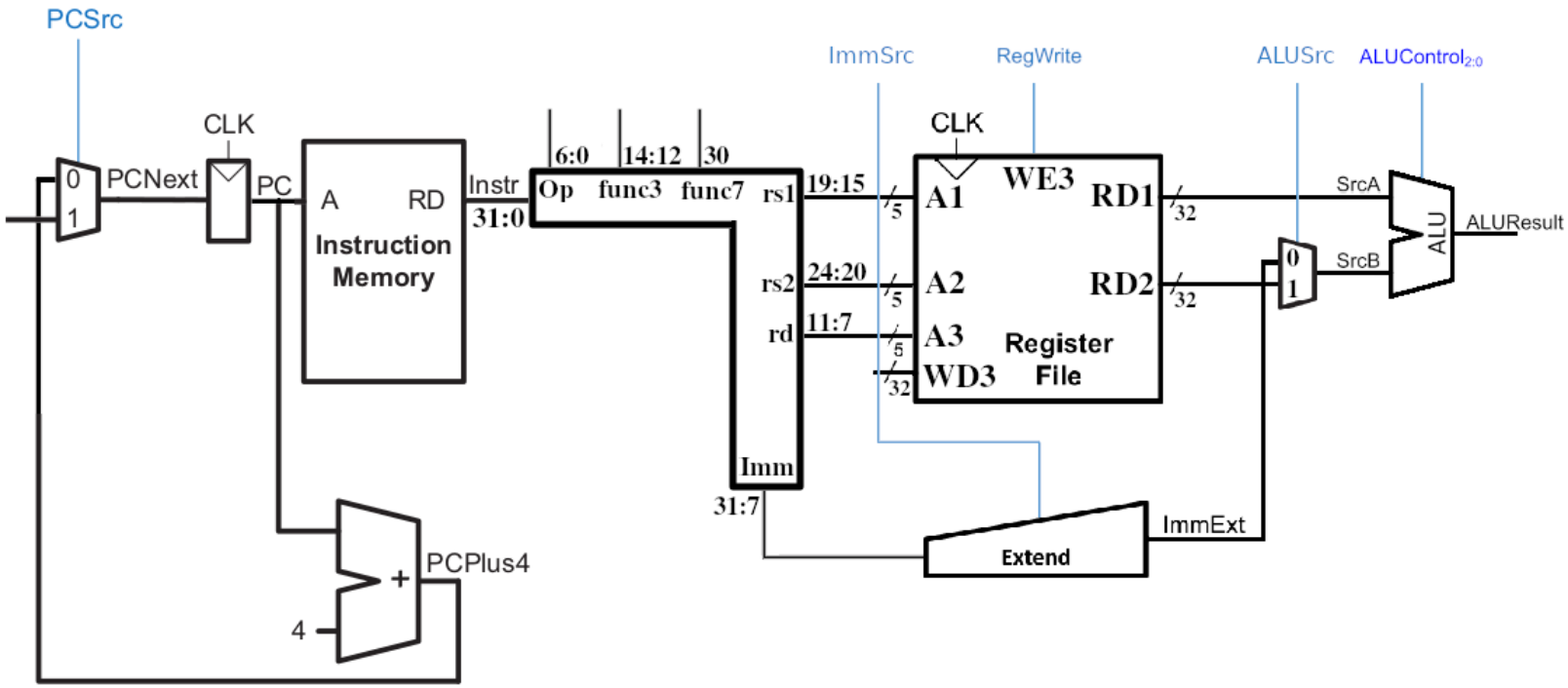
Sign Extension

- PC is the current instruction address



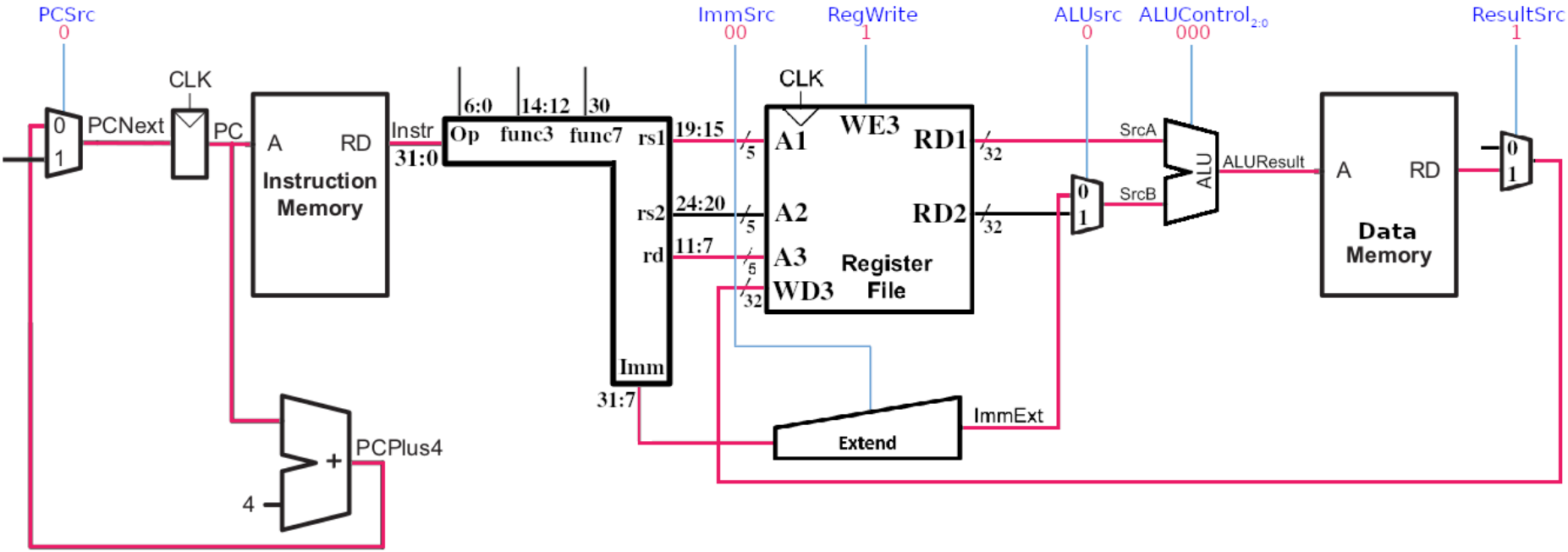
ALU

- PC is the current instruction address



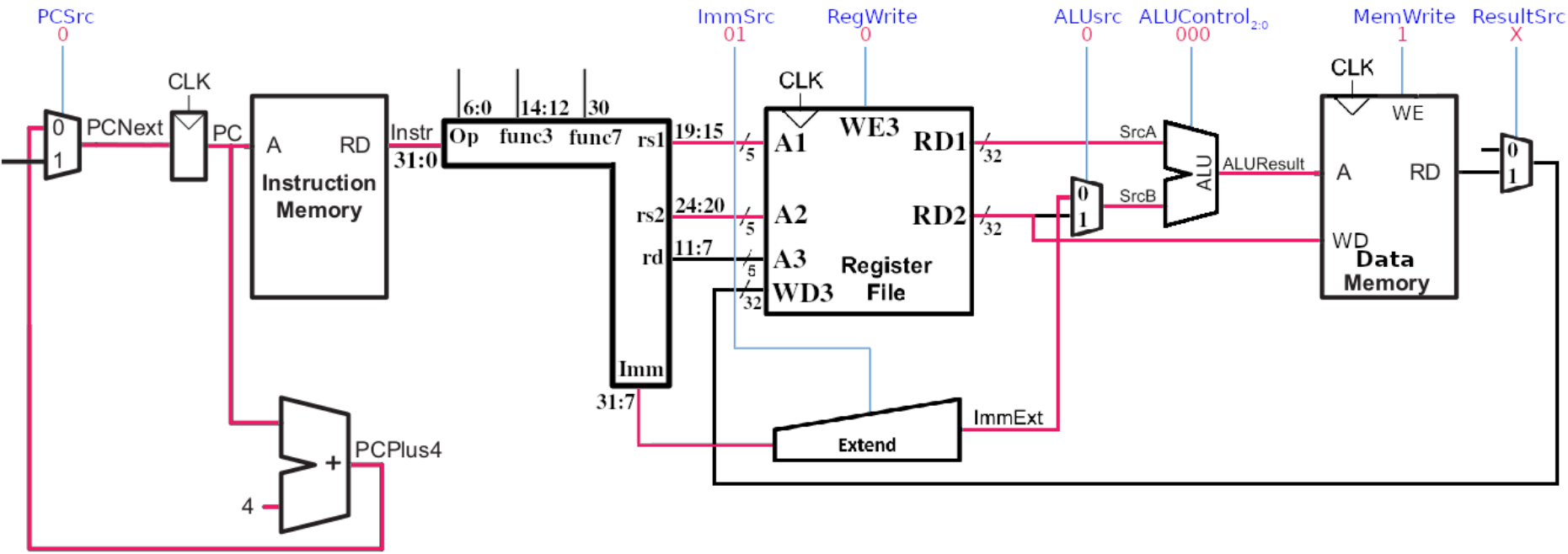
Load Instruction

- `lw rd, imm(rs1)`
  1. Read one source register
  2. Perform addition with immediate
  3. Write the second source register into data memory



Store Instruction

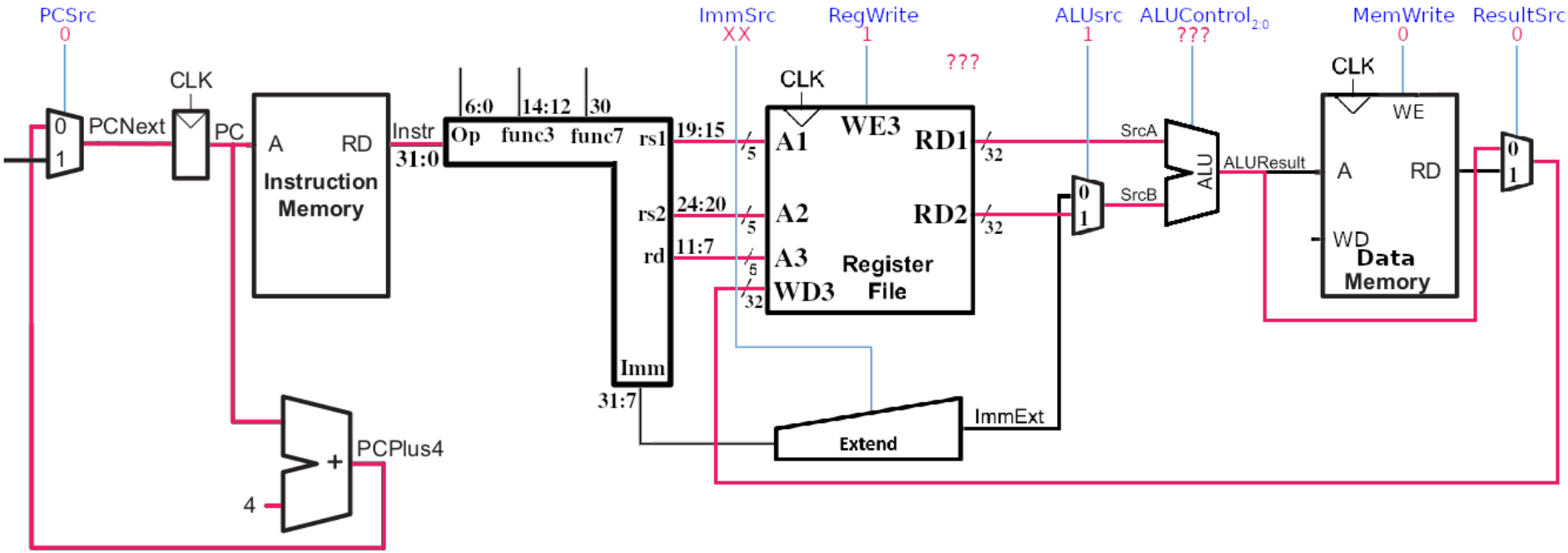
- `sw rs2, imm(rs1)`
  1. Read two source registers
  2. Perform addition between a source register and immediate
  3. Write the second source register into the memory



R-Type Instruction

- `add`, `sub`, `and`, `or`, `slt`
- They only differ in `ALUControl`
- `add rd, rs1, rs2`,
  1. Read two source registers
  2. Perform ALU operation
  3. Write back the ALU result into destination register

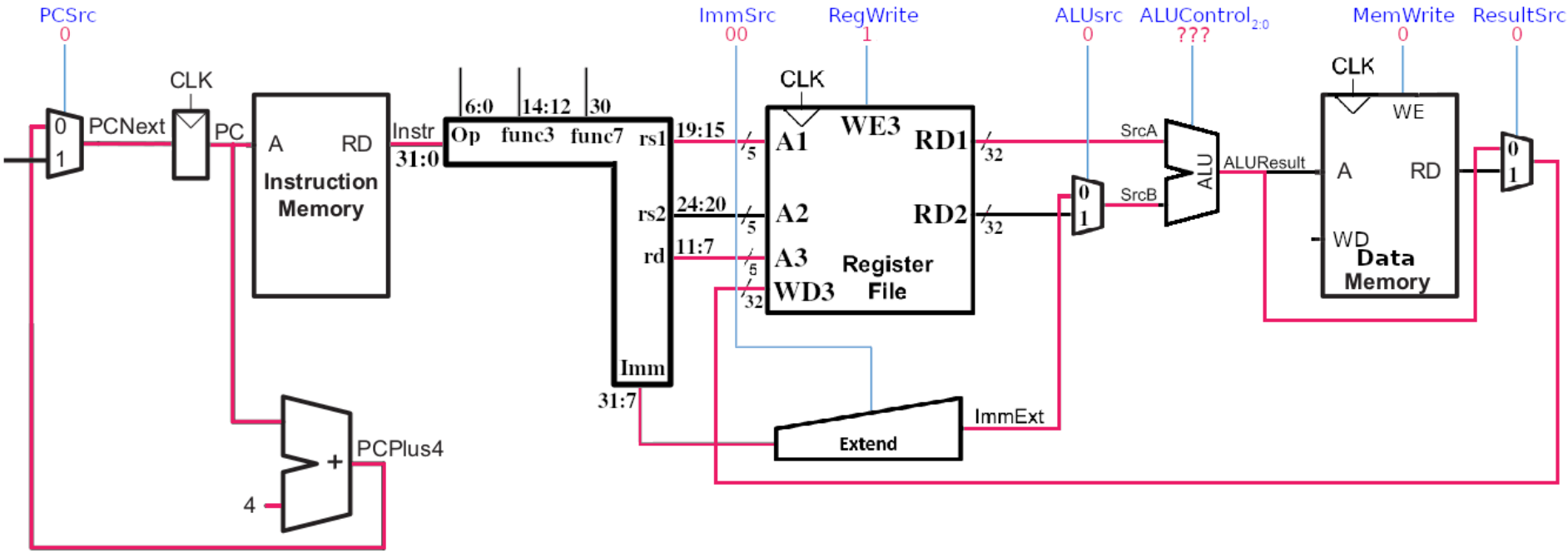
$ALUControl_{2:0}$	Function
000	Add
001	Subtract
010	AND
011	OR
101	SLT



I-Type Instruction

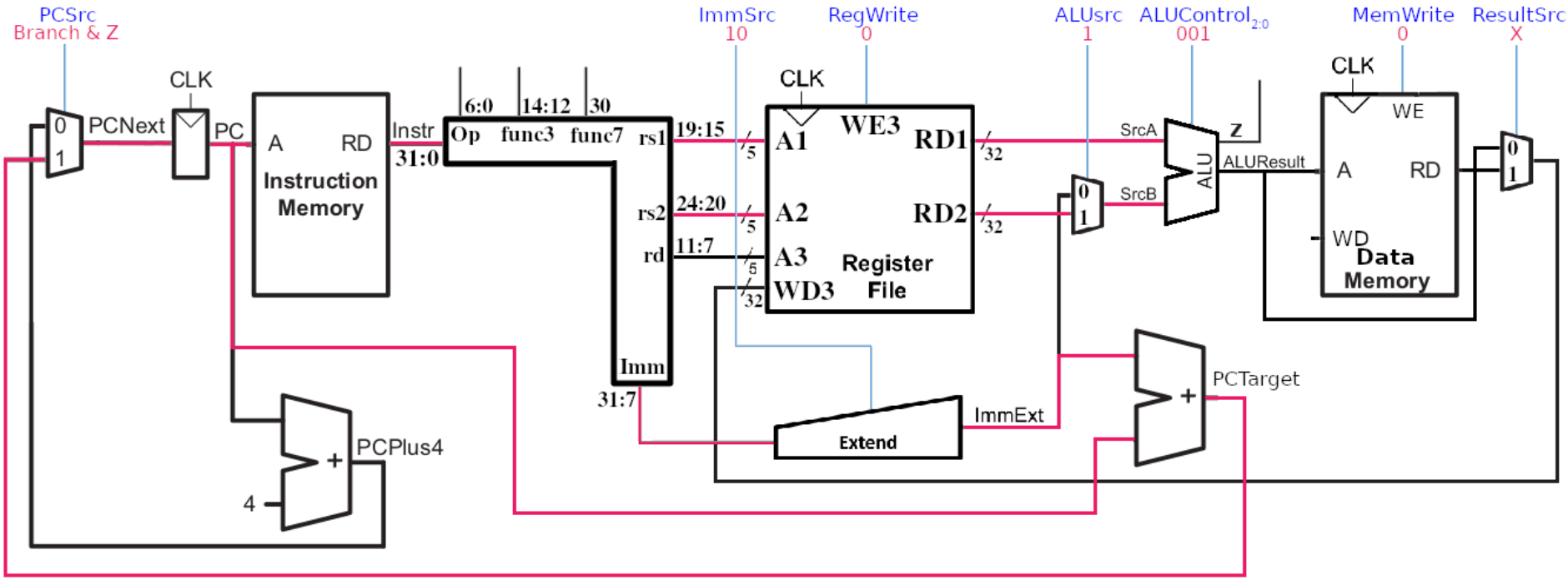
- `addi`, `andi`, `ori`, `slti`
- They only differ in `ALUControl`
- `addi rd, rs1, imm`,
  1. Read one source register
  2. Perform ALU operation between the source and Imm
  3. Write back the ALU result into destination register

$ALUControl_{2:0}$	Function
000	Add
001	Subtract
010	AND
011	OR
101	SLT



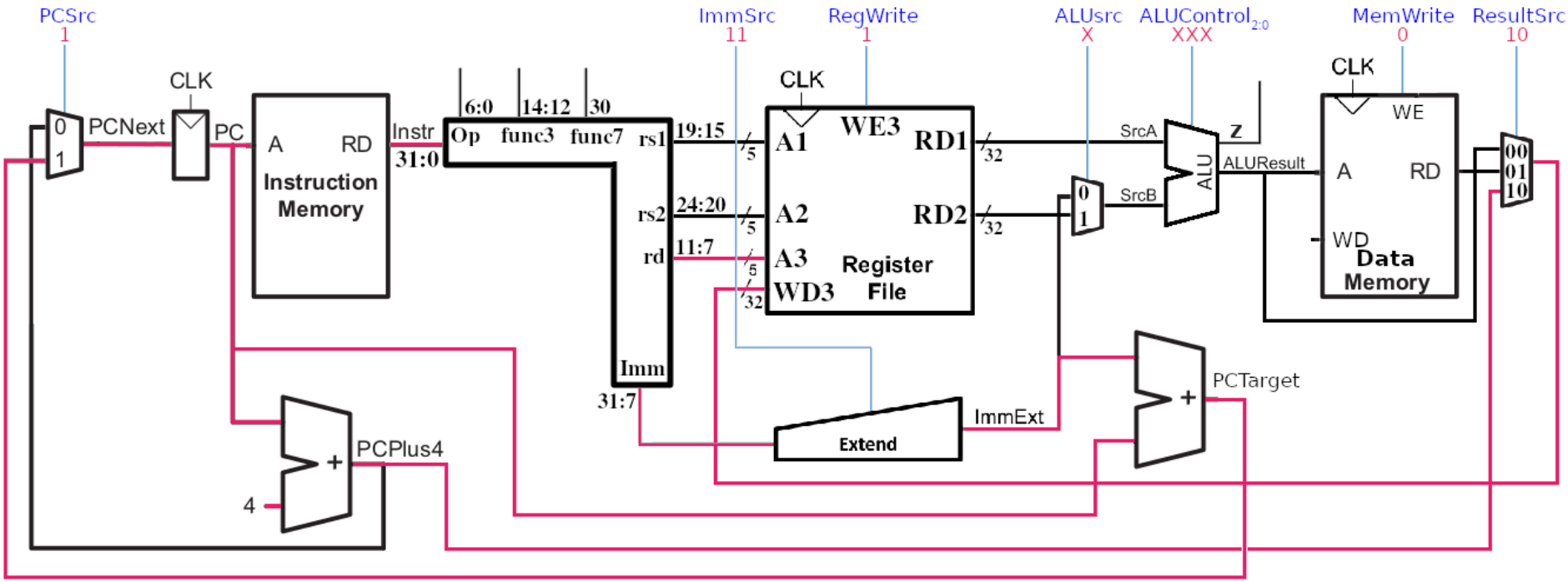
Conditional Branch

- `beq rs1, rs2, addr`,
  1. Read two source registers
  2. Perform the subtraction and issue `Z`
  3. Calculate `PC+Imm`
  4. Choose the select signal for `PCSrc`



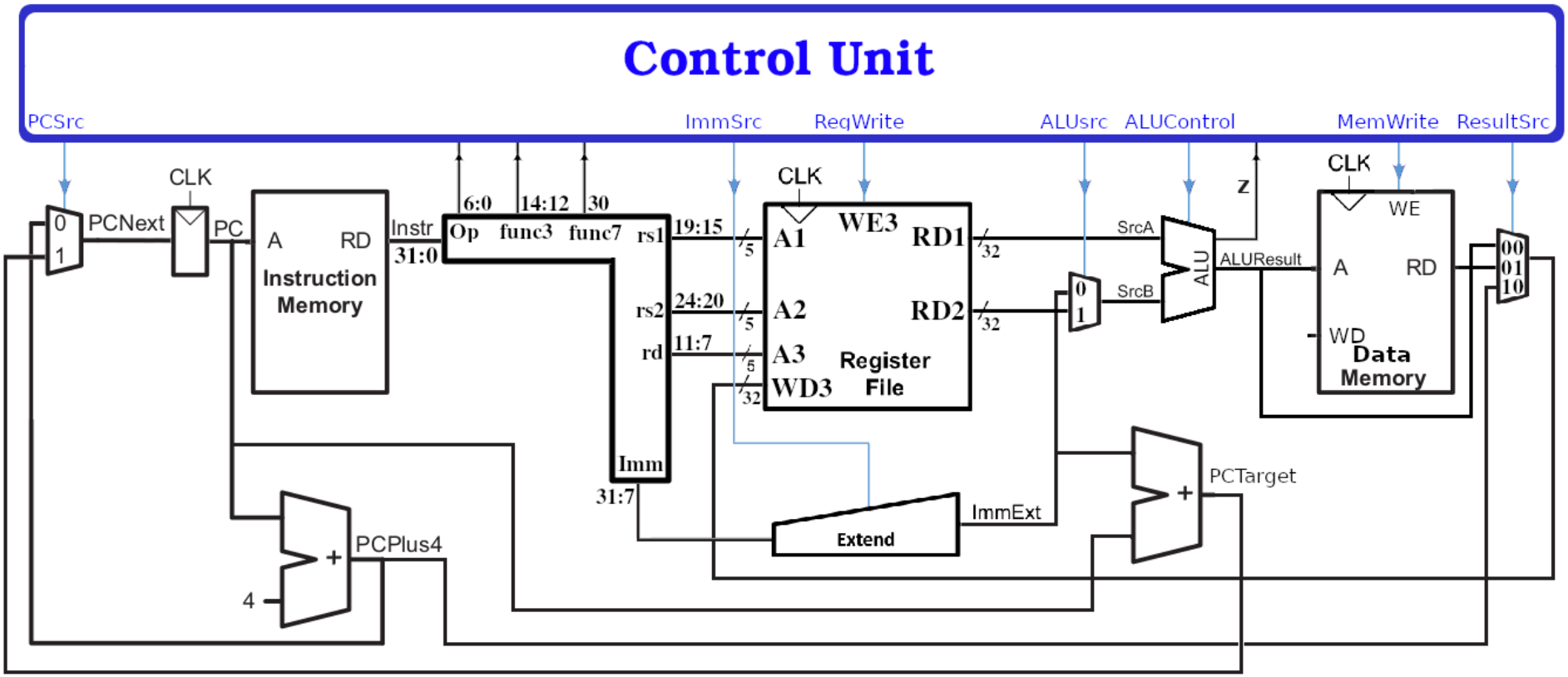
Jump Instruction

- `jal rd, label`,
  1. Jump to `Pc+imm`
  2. Write `Pc+4` into rd



Control Unit

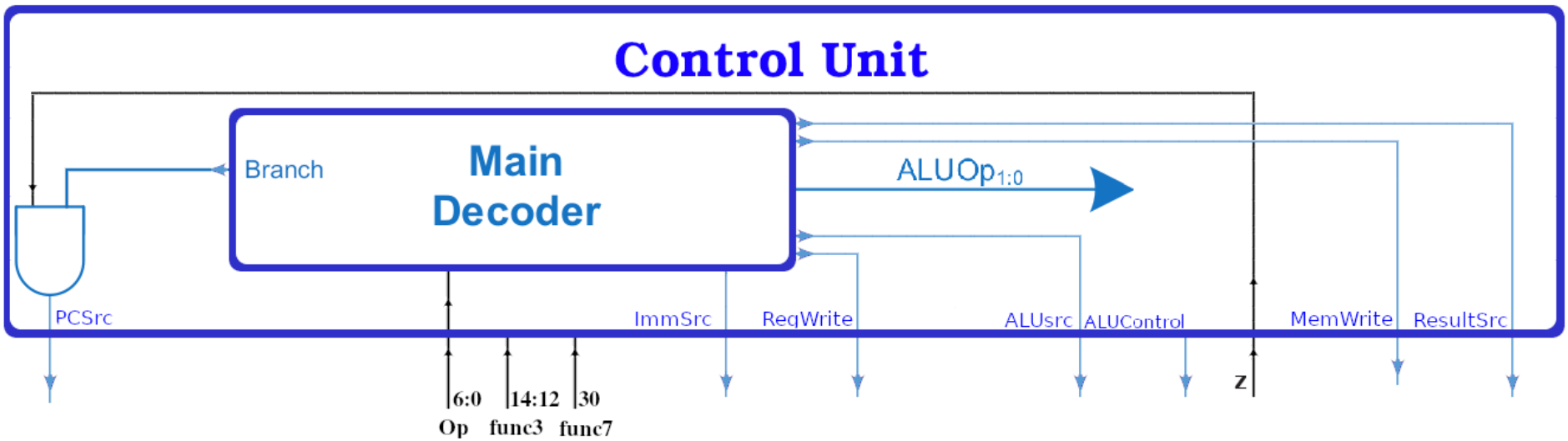
- `Op (6:0), func3 (14:12) and func7 (30)` are the inputs to the control unit
- Outputs: `PCSrc,ImmSrc,RegWrite,ALUSrc,ALUControl,MemWrite,ResultSrc`



Control Unit

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
lw	0000011	1	00	1	0	01	0	00	0
sw	0100011	0	01	1	1	xx	0	00	0
R-type	0110011	1	xx	0	0	00	0	10	0
beq	1100011	0	10	0	0	xx	1	01	0
I-type ALU	0010011	1	00	1	0	00	0	10	0
jal	1101111	1	11	x	0	10	0	xx	1

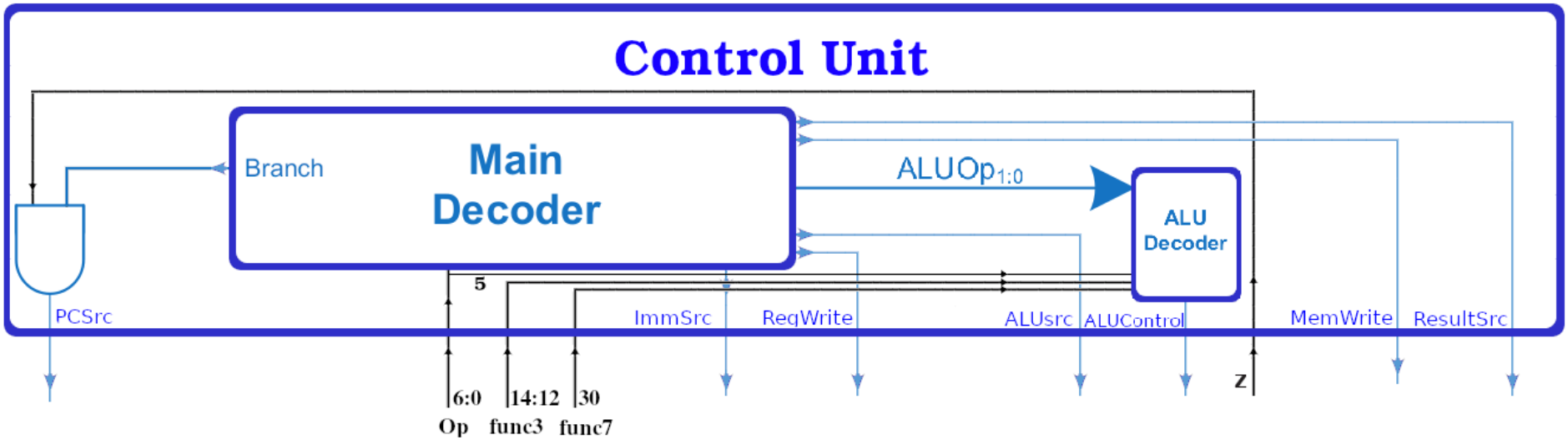
- Main Decoder truth tabel



Control Unit

ALUOp	funct3	$\{op_5, funct7_5\}$	ALUControl	Instruction
00	x	x	000 (add)	`lw, sw`
01	x	x	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add`
	000	11	001 (subtract)	`sub`
	010	x	101 (set less than)	`slt`
	110	x	011 (or)	`or`
	111	x	010 (and)	`and`

- `ALU truth tabel`



Performance Analysis

- The CPU performance

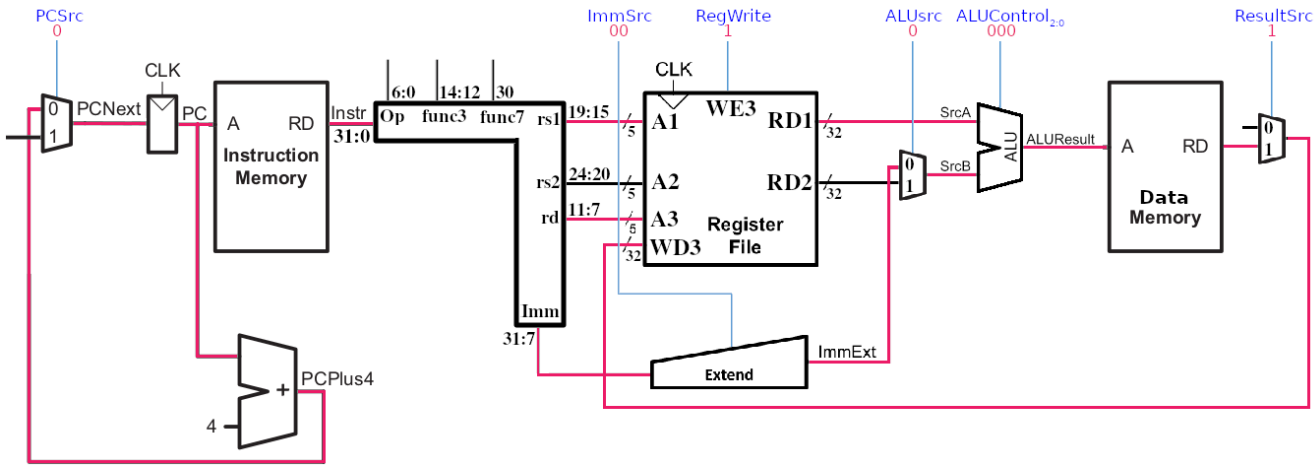
$$ExecutionTime = (\#Inst.) \times (CPI) \times \frac{1}{F_{clock}}$$

- In single cycle CPU, `CPI=1`
- The cycle time is set by the critical path
- `lw` has the longest path

$$T_{single} = t_{pcq} + t_{mem} + \max\{t_{RFread}, t_{dec} + t_{ext} + t_{mux}\} + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$$

- Assuming that $t_{RFread} > t_{dec} + t_{ext} + t_{mux}$,

$$T_{single} = t_{pcq} + 2\,t_{mem} + t_{RFread} + t_{ALU} + t_{mux} + t_{RFsetup}$$



Performance Analysis

- Using the following table, the clock cycle can be estimated as

$$T_{single} = t_{pcq} + 2\,t_{mem} + t_{RFread} + t_{ALU} + t_{mux} + t_{RFsetup}$$
$$= 40 + 2 \times 200 + 100 + 120 + 30 + 60 = 750ps$$

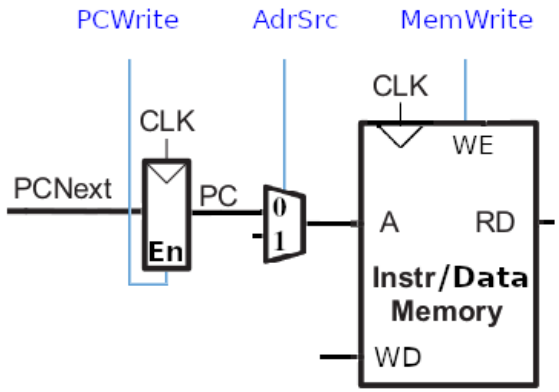
Element	Parameter	Delay (ps)
Register clk-to-Q	t_{pcq}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	$t_{AND\text{-}OR}$	20
ALU	t_{ALU}	120
Decoder (control unit)	t_{dec}	25
Extend unit	t_{ext}	35
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

Disadvantages

- Requires a clock cycle long enough to support the slowest instruction
- Requires separate memories for instructions and data
- Hardware are not being used efficiently, e.g. one adder for each task

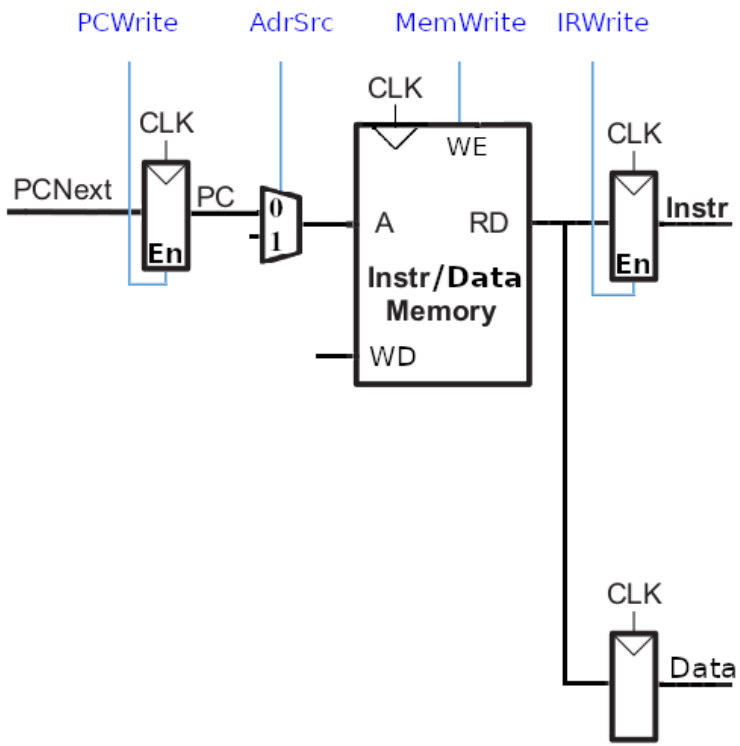
Combining the Memories

- Instruction and Data memories can be unified using a multiplexer for the address



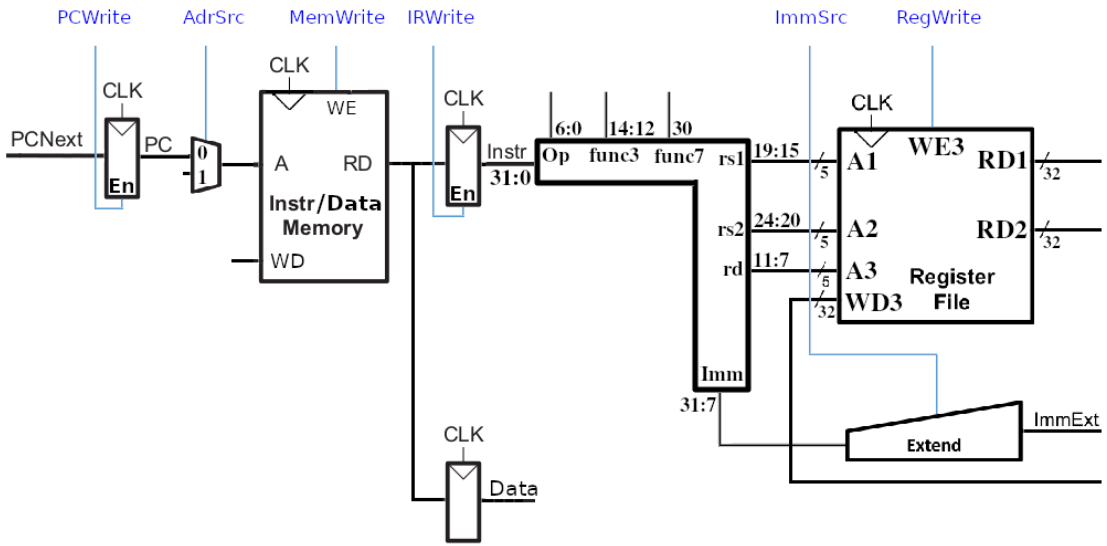
Combining the Memories

- Instruction and Data memories can be unified using a multiplexer for the address
- The Instruction and data need to be stored in registers



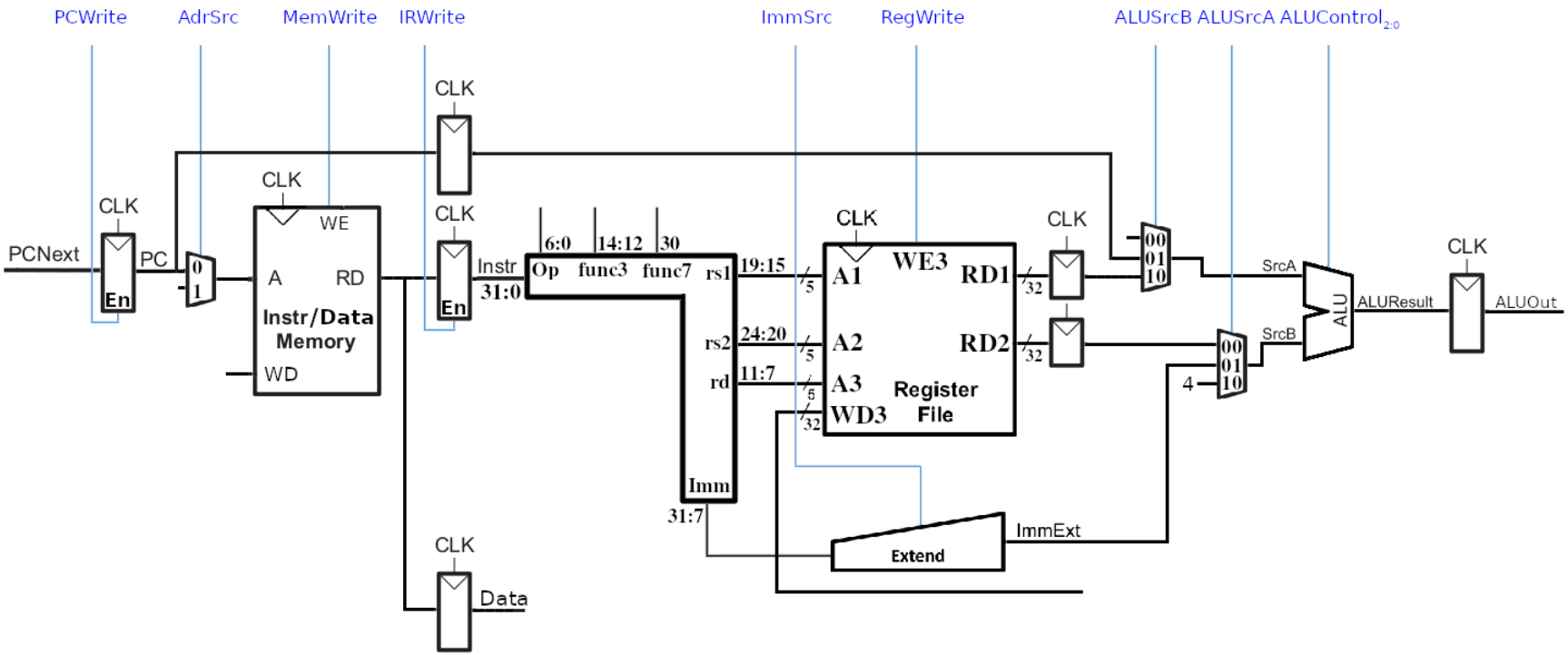
Instruction Decoding

- Instruction and Data memories can be unified using a multiplexer for the address
- The Instruction and data need to be stored in registers
- Instructions are the same as before, so is the instruction decoder!



ALU

- Instruction and Data memories can be unified using a multiplexer for the address
- The Instruction and data need to be stored in registers
- Instructions are the same as before, so is the instruction decoder!
- We can save adders by reusing ALU for all the additions



Big Problem

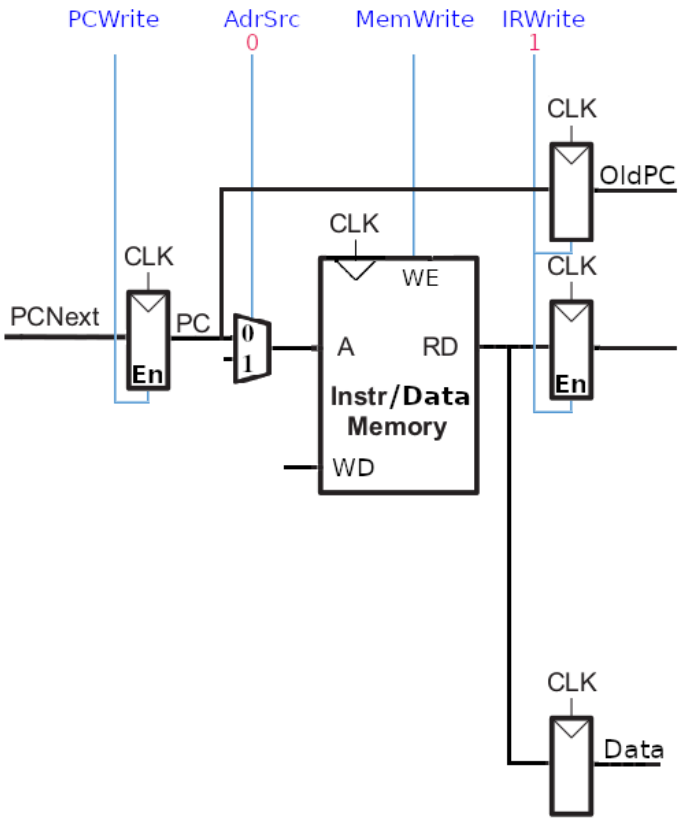
- Instruction and Data memories can be unified using a multiplexer for the address
- The Instruction and data need to be stored in registers
- Since the instructions are the same, so is the instruction decoder
- We can save adders by reusing ALU for all the additions
- We have a problem!
  We can not perform multiple operation with one component in one cycle.

Big Problem

- Instruction and Data memories can be unified using a multiplexer for the address
- The Instruction and data need to be stored in registers
- Since the instructions are the same, so is the instruction decoder
- We can save adders by reusing ALU for all the additions
- We have a problem!
  We can not perform multiple operation with one component in one cycle.
- Therefore we break each instruction into multiple cycles
- The great advantage is that different instructions can have different run of cycles

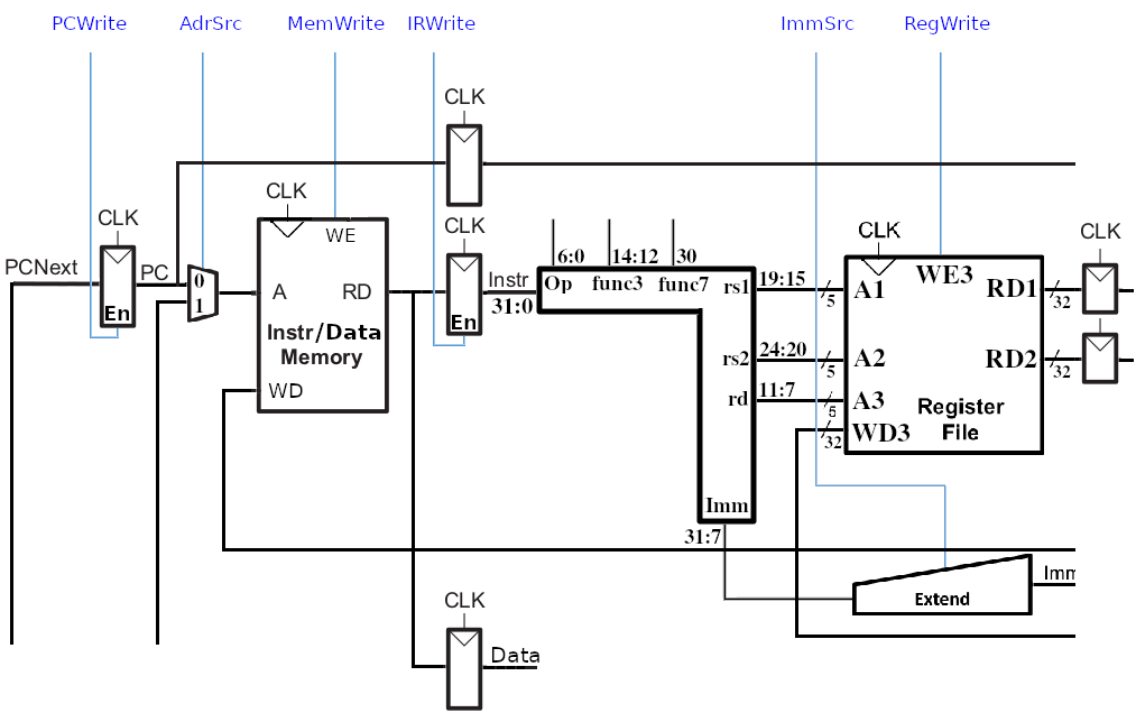
Fetch

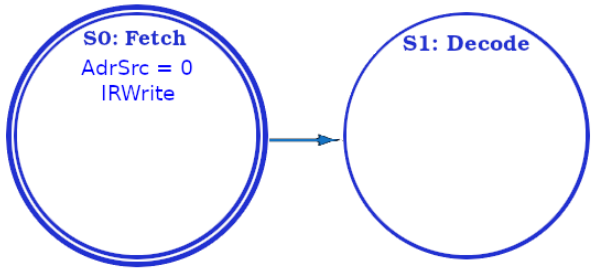
- Fetch Cycle:
  - **The instruction will be fetched from the current PC**
  - **The fetched instruction will be stored into Instruction Register**
  - **The current PC will be saved into old PC**



Decode

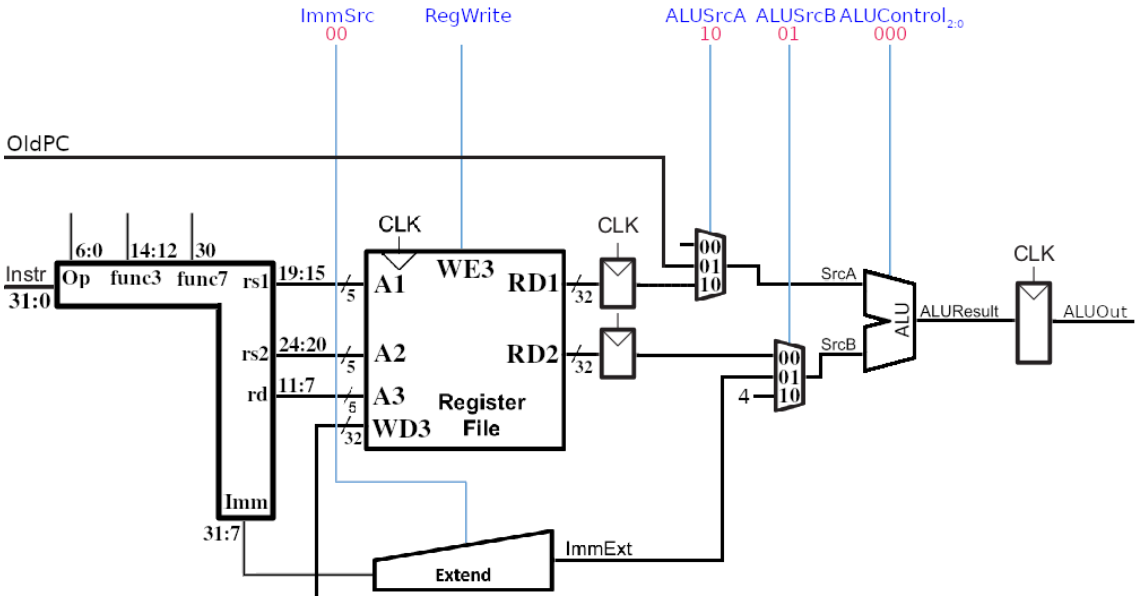
- Decode Cycle:
  - **Signals flow combinationally, no need to issue any control signal**

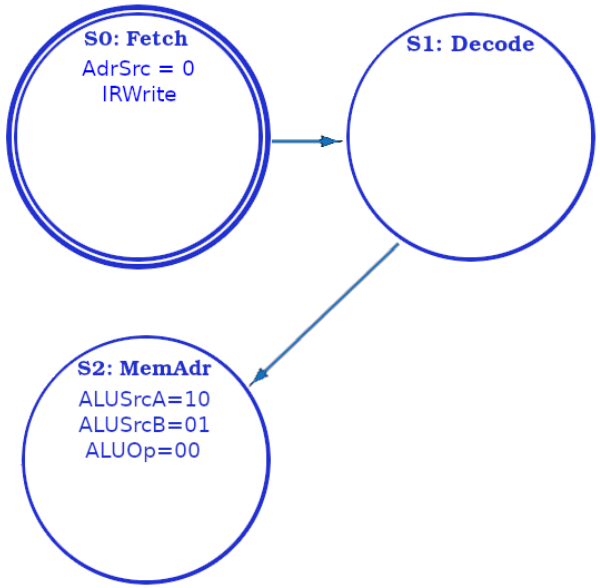




Load Instruction

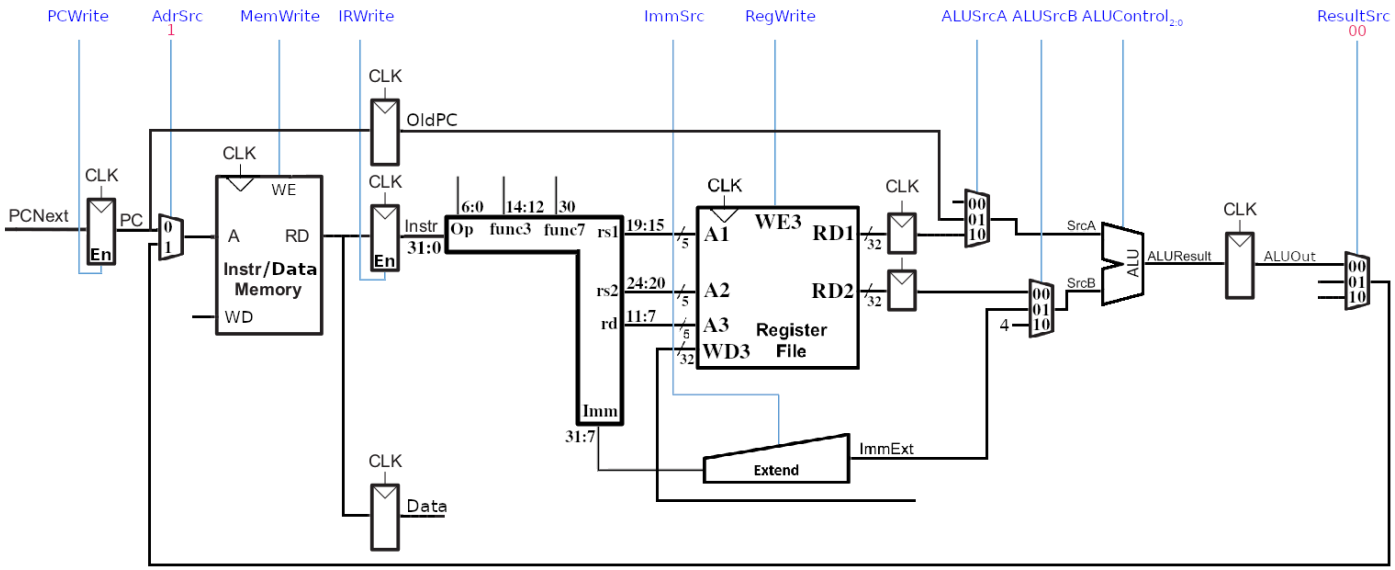
- MemAdr Cycle:
  - **ALU must add the base address and the offset**
  - **The address will be stored in `ALUOut` register**

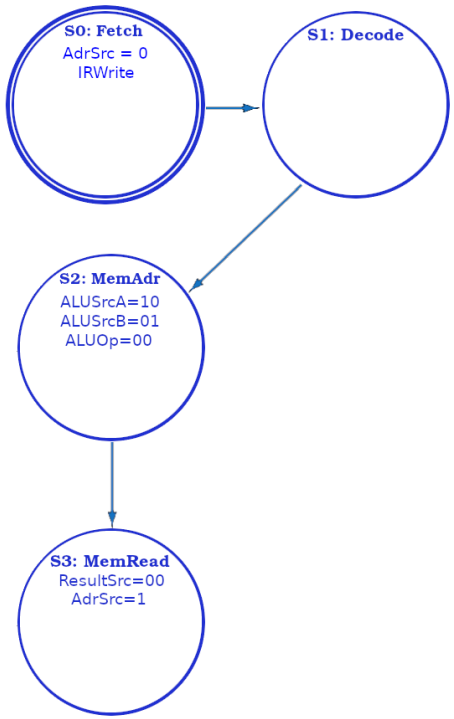




Load Instruction

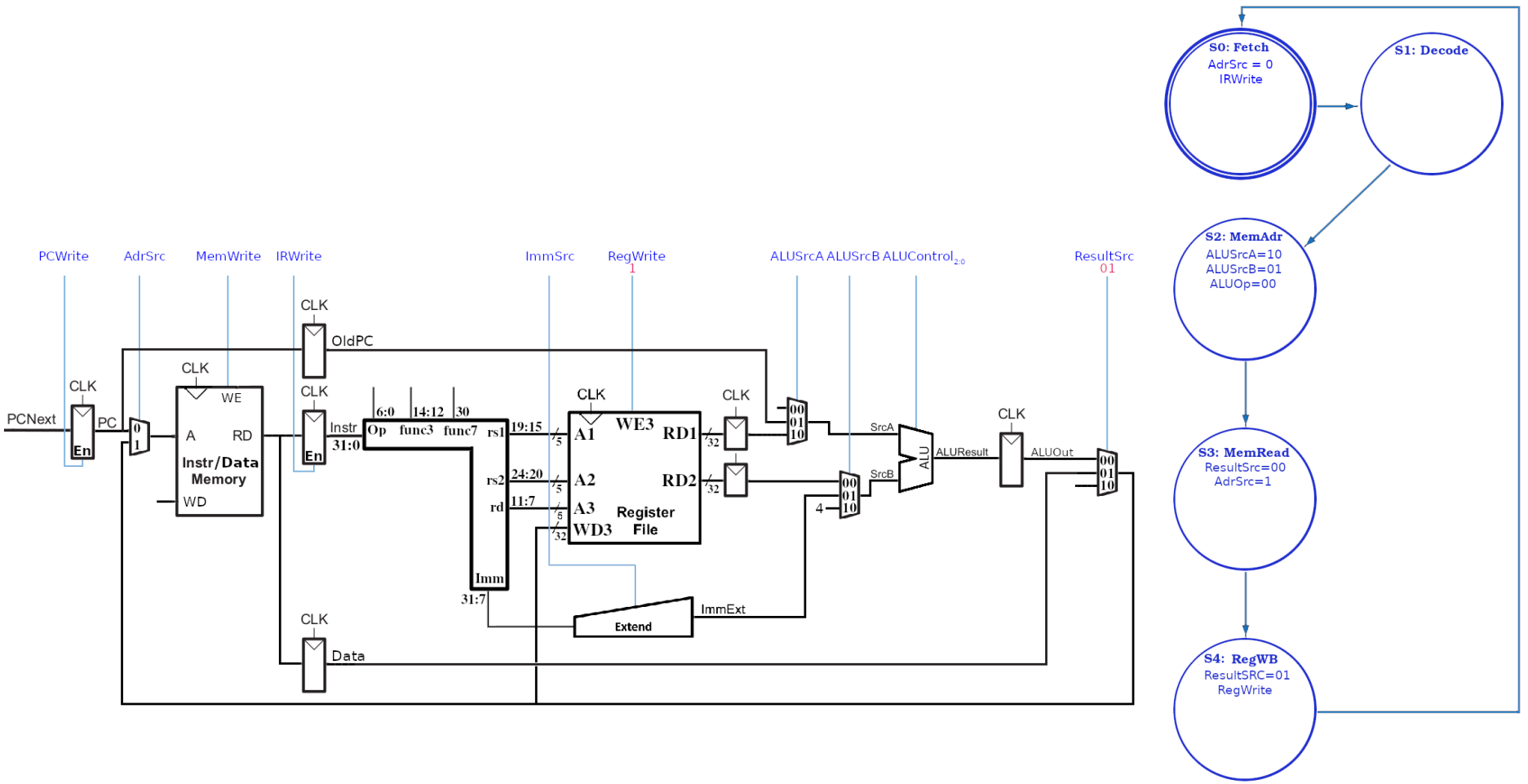
- MemRead Cycle:
  - **Memory must be read from the address stored in `ALUOut`**
  - **The data will be stored in `Data` register**





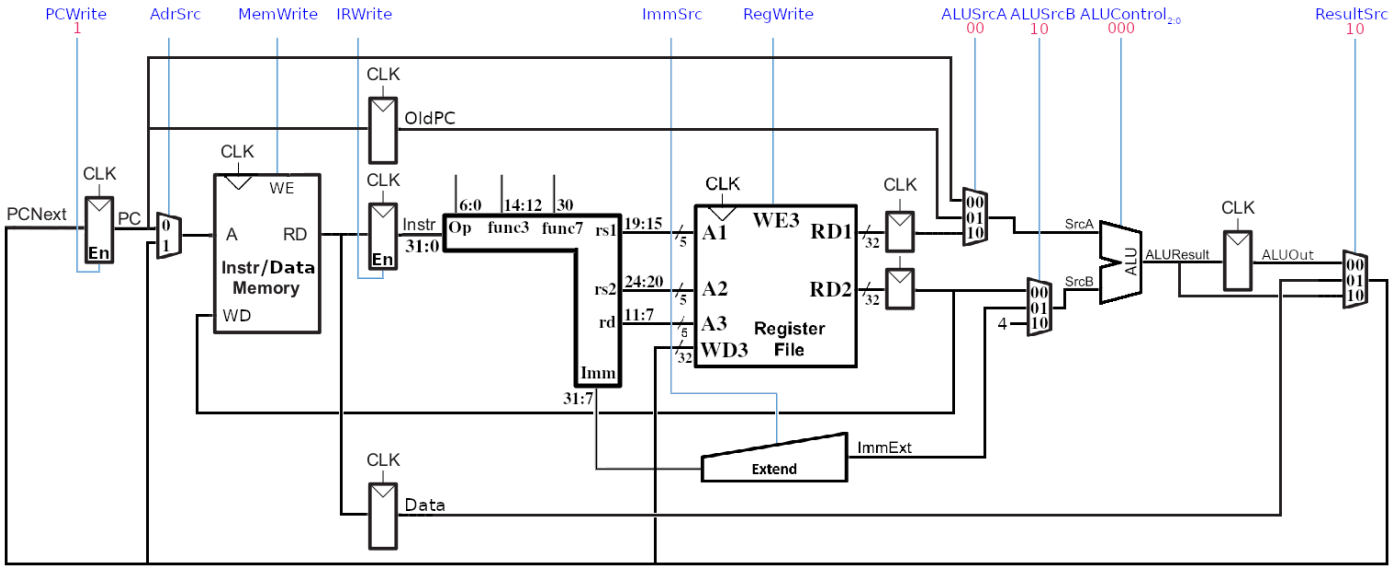
Load Instruction

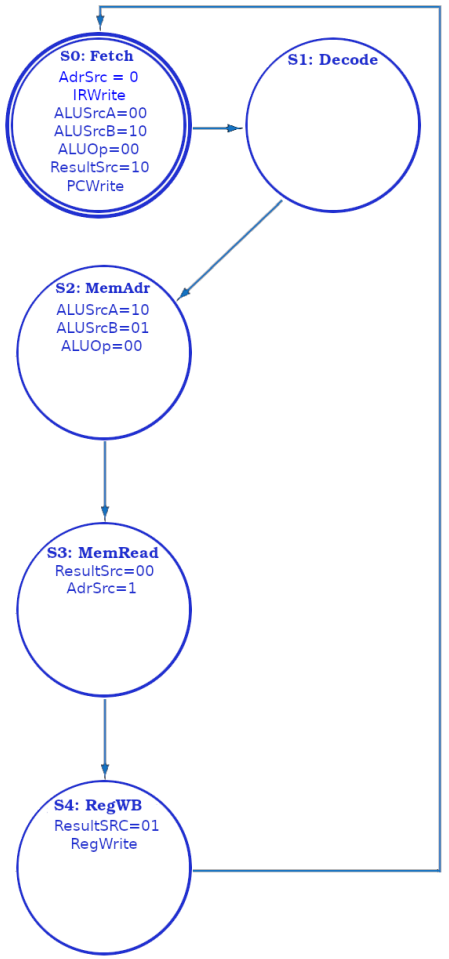
- MemWB Cycle:
  - **The data stored in `Data` reg must be written back into register file**



Updating PC

- We increment PC by 4 before fetching the next instruction
- ALU is not being used during Fetch cycle
- We can add necessary control signals to update PC during fetch cycle

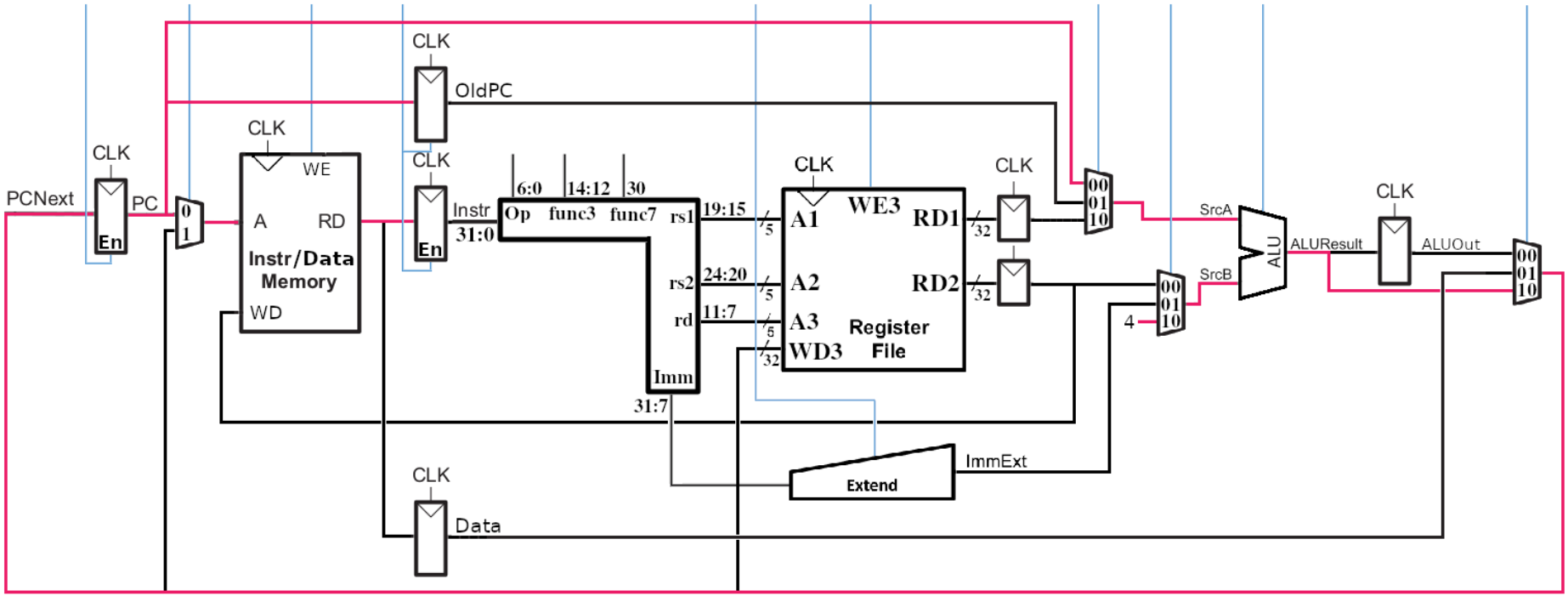




Load Instruction

- `lw rd, imm(rs1)`

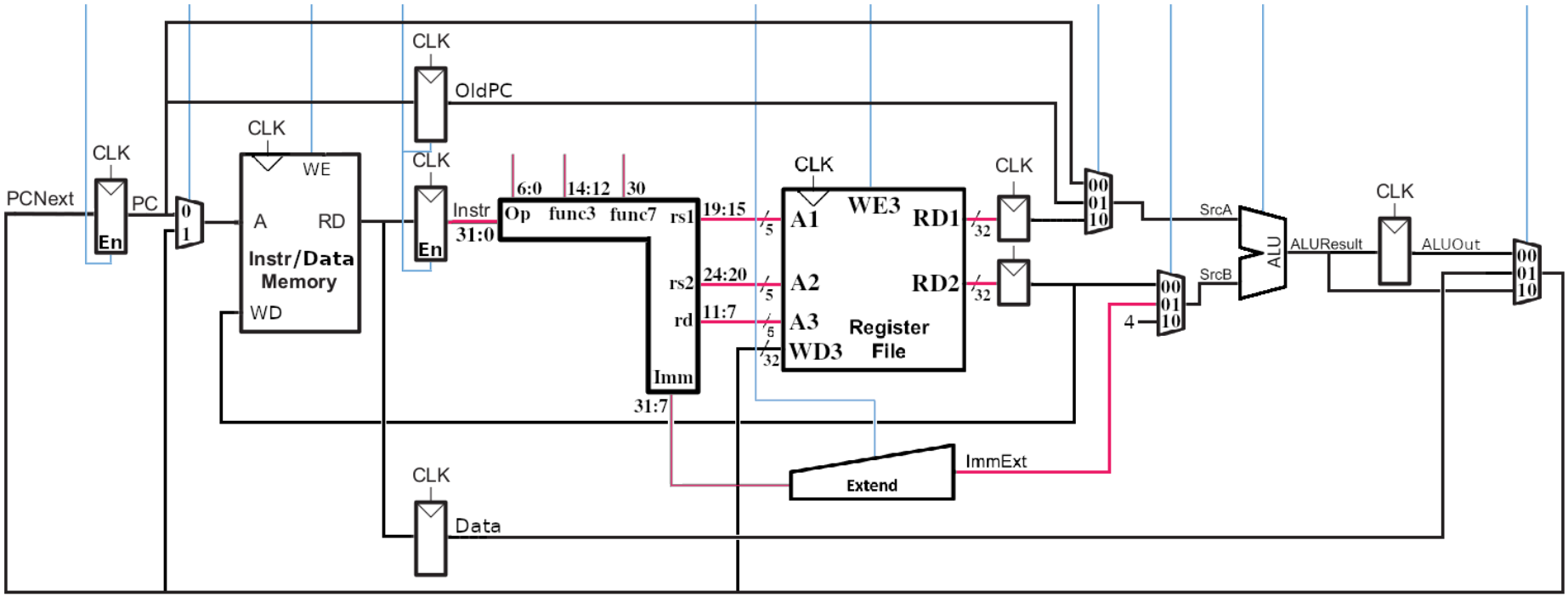
	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	$\text{ALUControl}_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10



Load Instruction

- `lw rd, imm(rs1)`

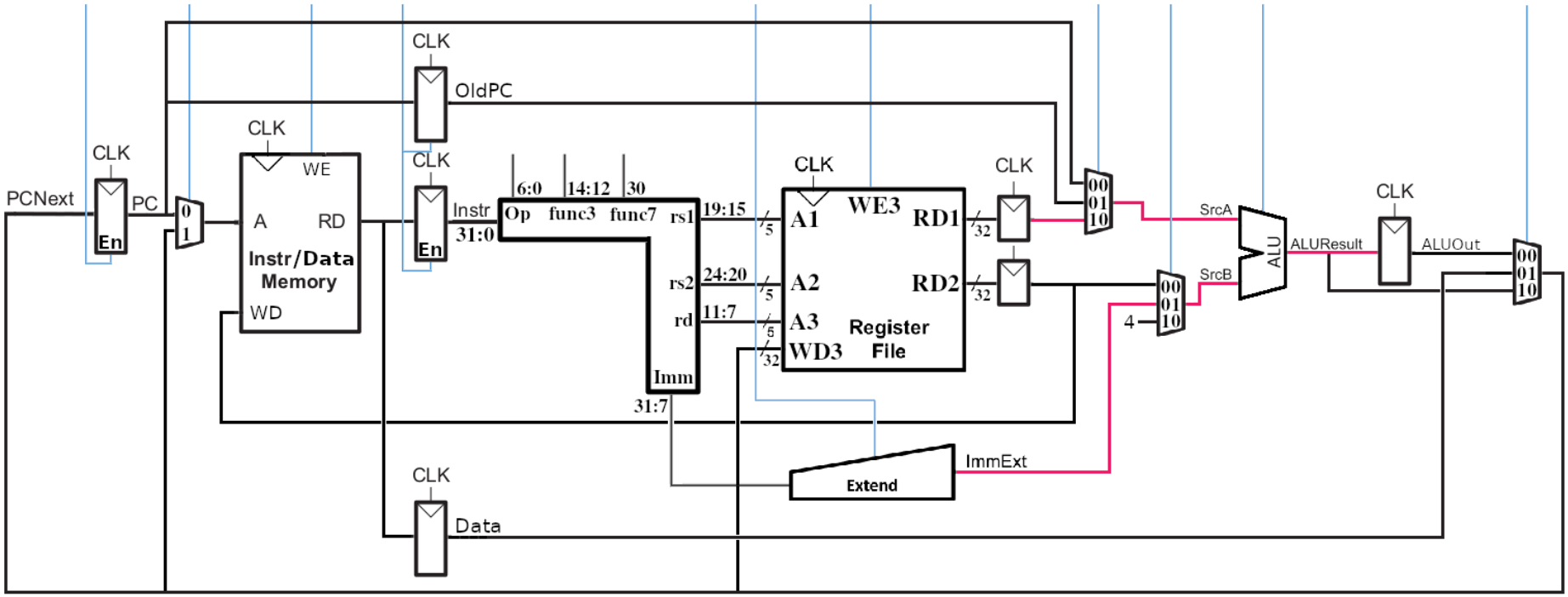
	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	$\text{ALUControl}_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10
Decode:					00					



Load Instruction

- `lw rd, imm(rs1)`

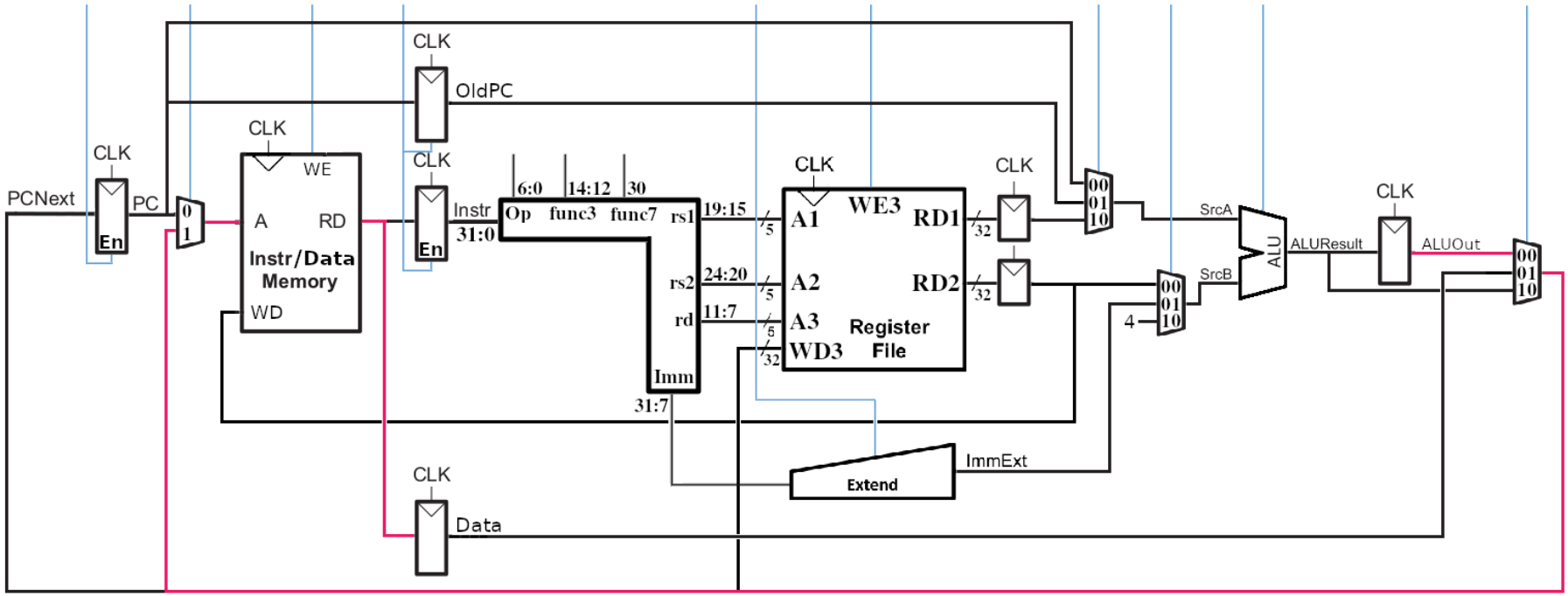
	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	ALUControl$_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10
Decode:					00					
MemAdr:					00		10	01	000	



Load Instruction

- `lw rd, imm(rs1)`

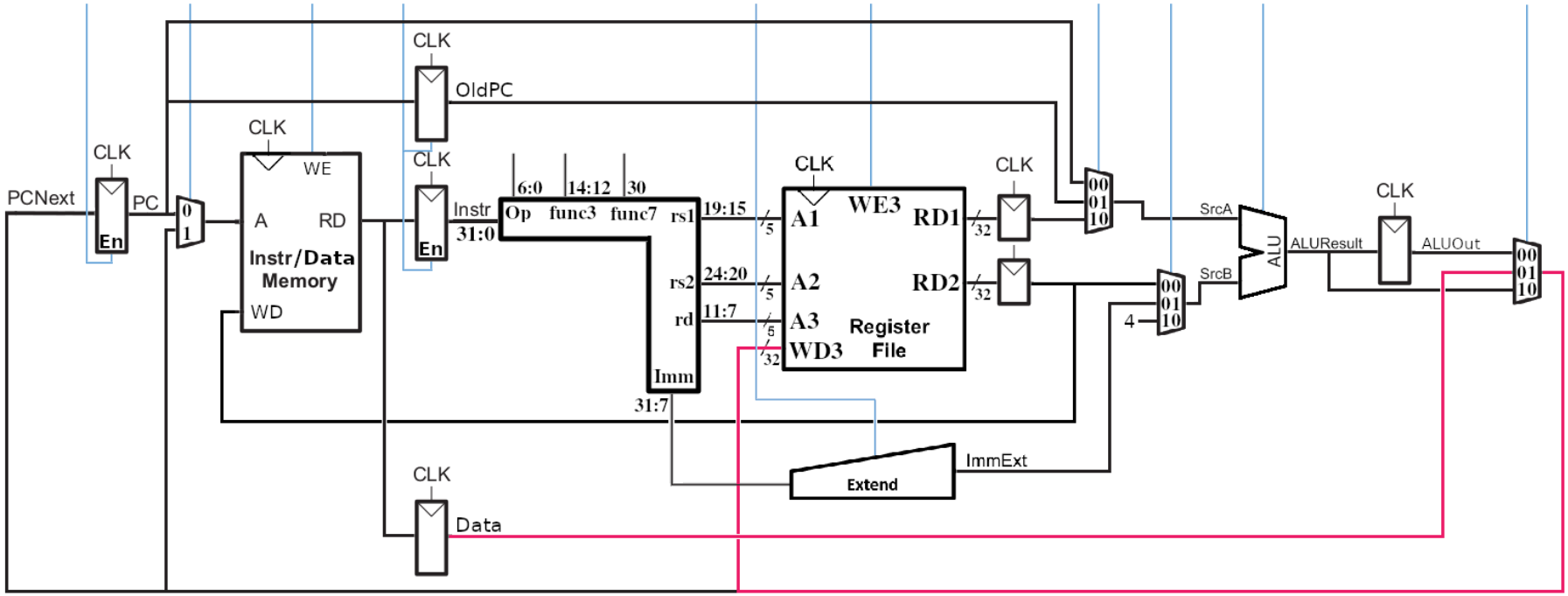
	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	$\text{ALUControl}_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10
Decode:					00					
MemAdr:					00		10	01	000	
MemRead:		1			00					00



Load Instruction

- `lw rd, imm(rs1)`

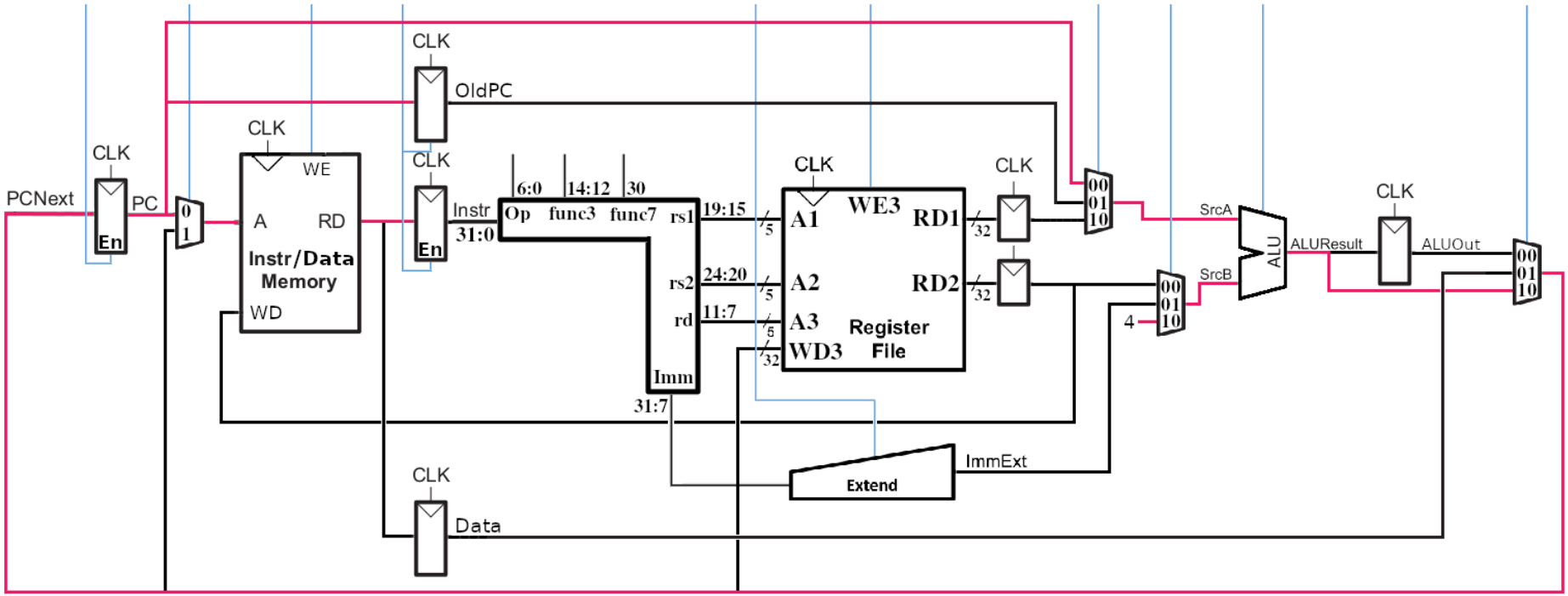
	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	$ALUControl_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10
Decode:					00					
MemAdr:					00		10	01	000	
MemRead:		1			00					00
RegWB:					00	1				01



Store Instruction

- `sw rs2, imm(rs1)`

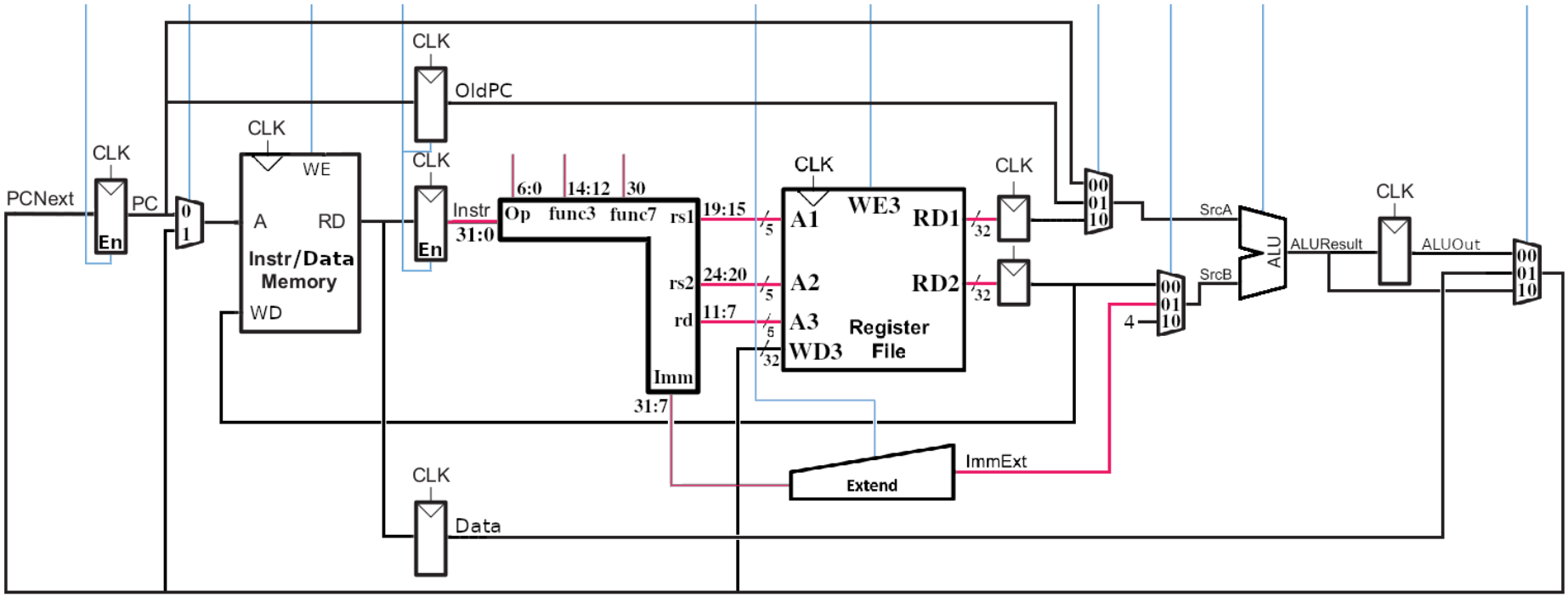
	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	$ALUControl_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10



Store Instruction

- `sw rs2, imm(rs1)`

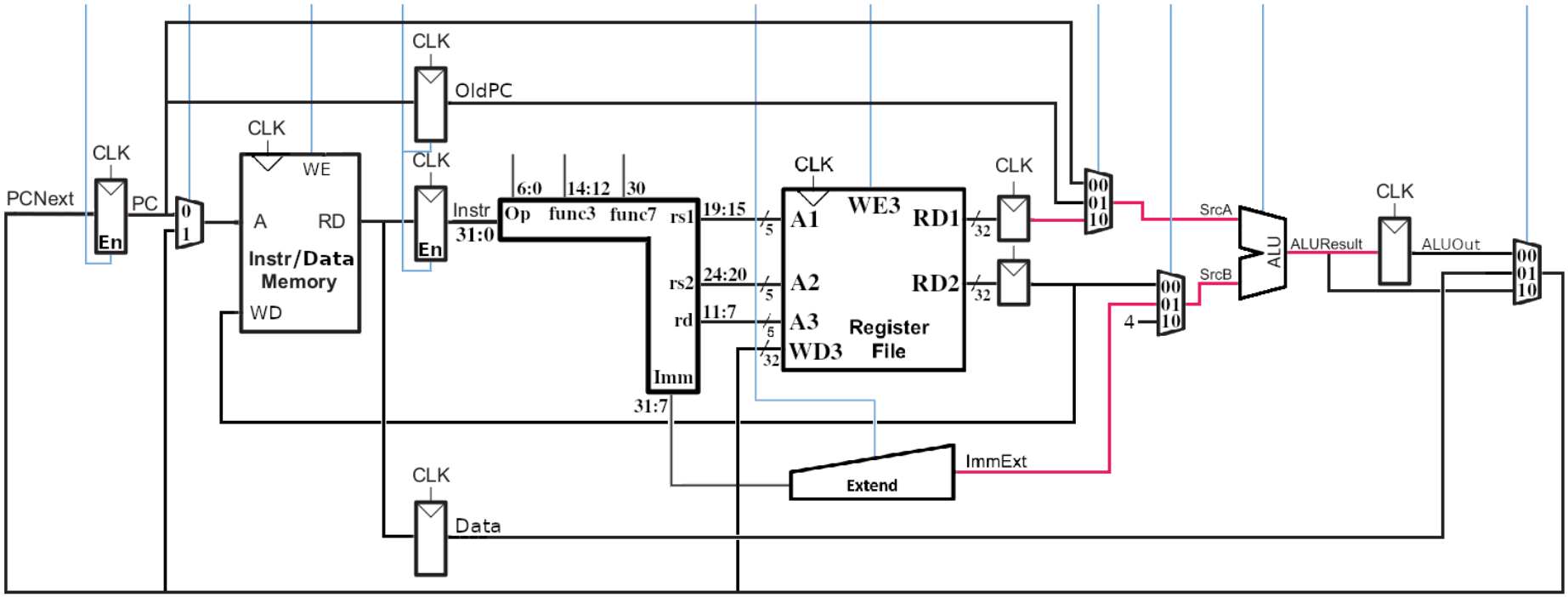
	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	$\text{ALUControl}_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10
Decode:					00					



Store Instruction

- `sw rs2, imm(rs1)`

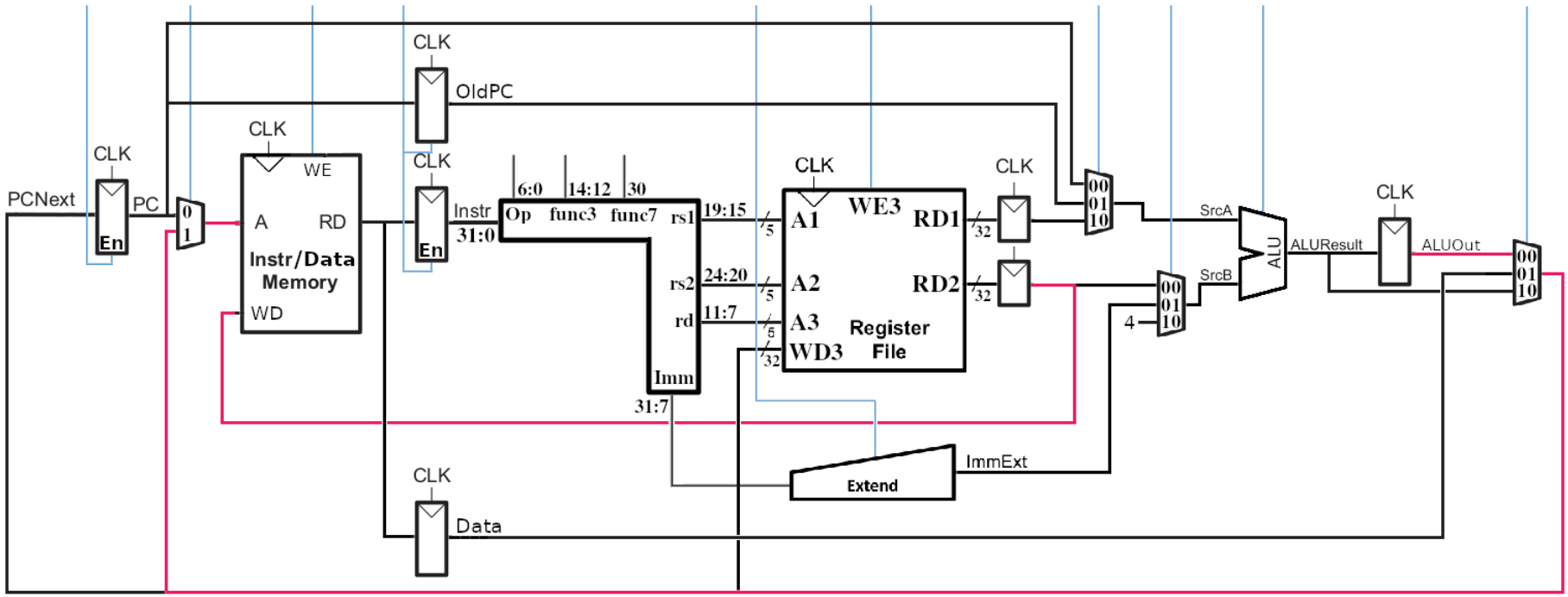
	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	$ALUControl_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10
Decode:					01					
MemAdr:					01		10	01	000	



Store Instruction

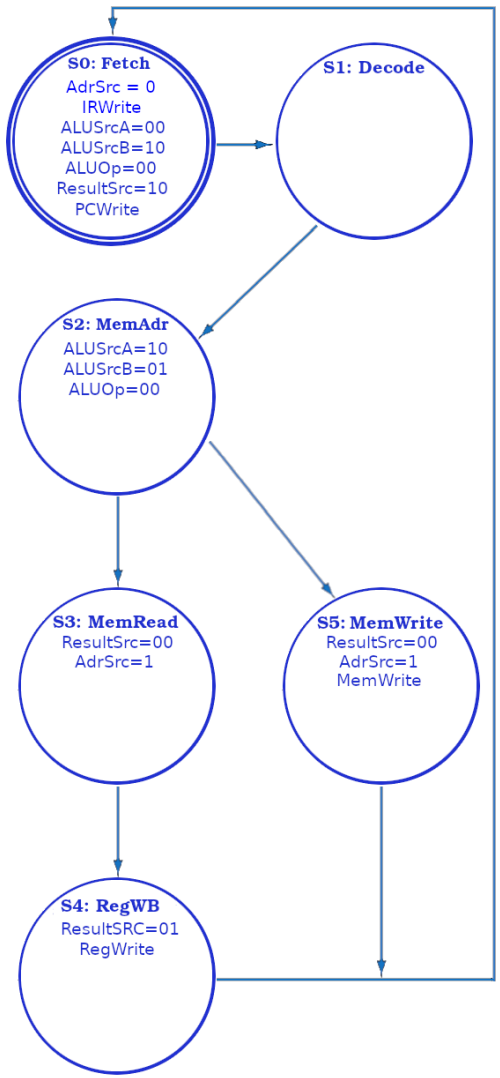
- `sw rs2, imm(rs1)`

	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	$\text{ALUControl}_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10
Decode:					01					
MemAdr:					01		10	01	000	
MemWrite:		1	1							00



Store Instruction

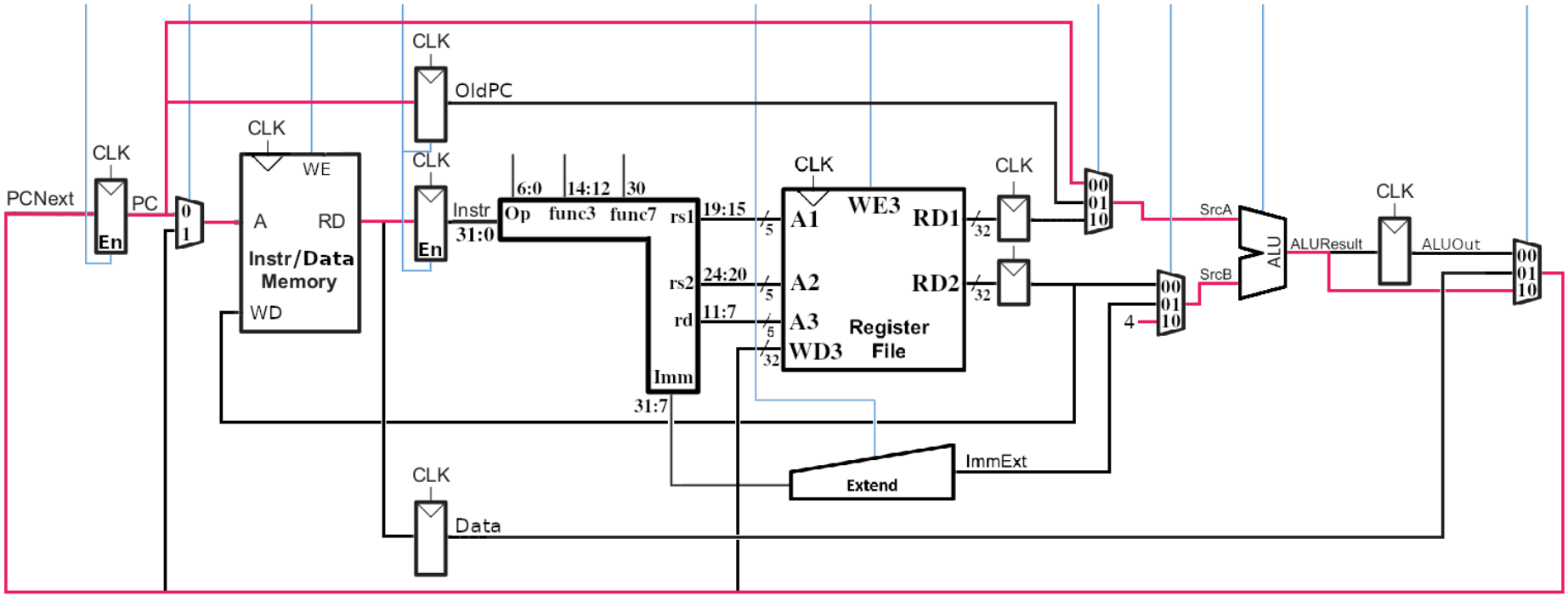
- `sw rs2, imm(rs1)`



R-Type Instruction

- `add rd, rs1, rs2`

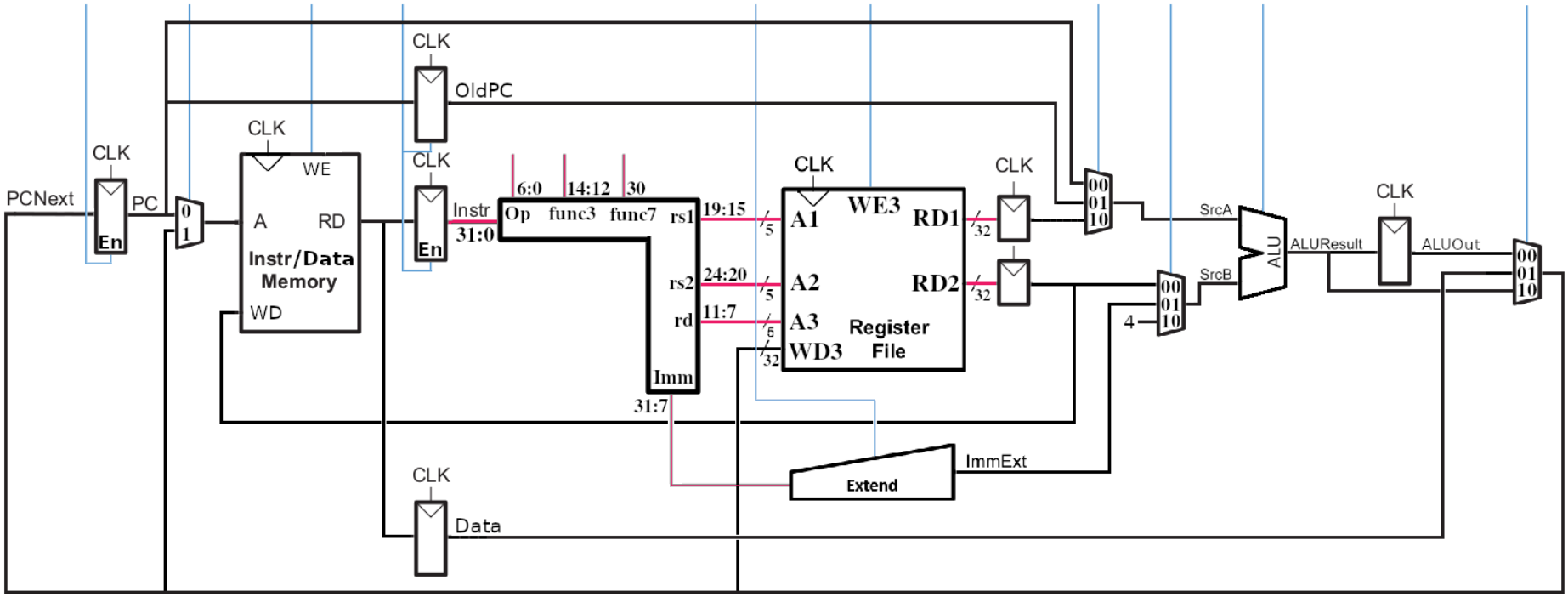
	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	$\text{ALUControl}_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10



R-Type Instruction

- `add rd, rs1, rs2`

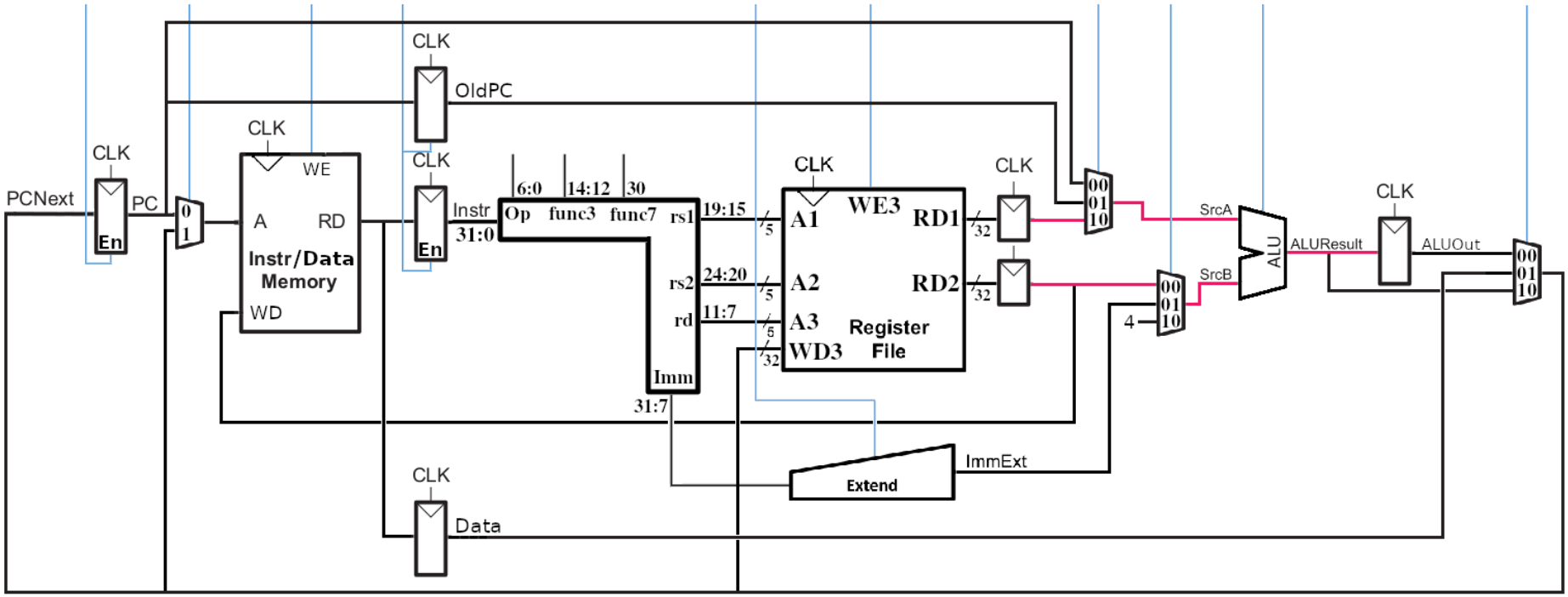
	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	$ALUControl_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10
Decode:										



R-Type Instruction

- `add rd, rs1, rs2`

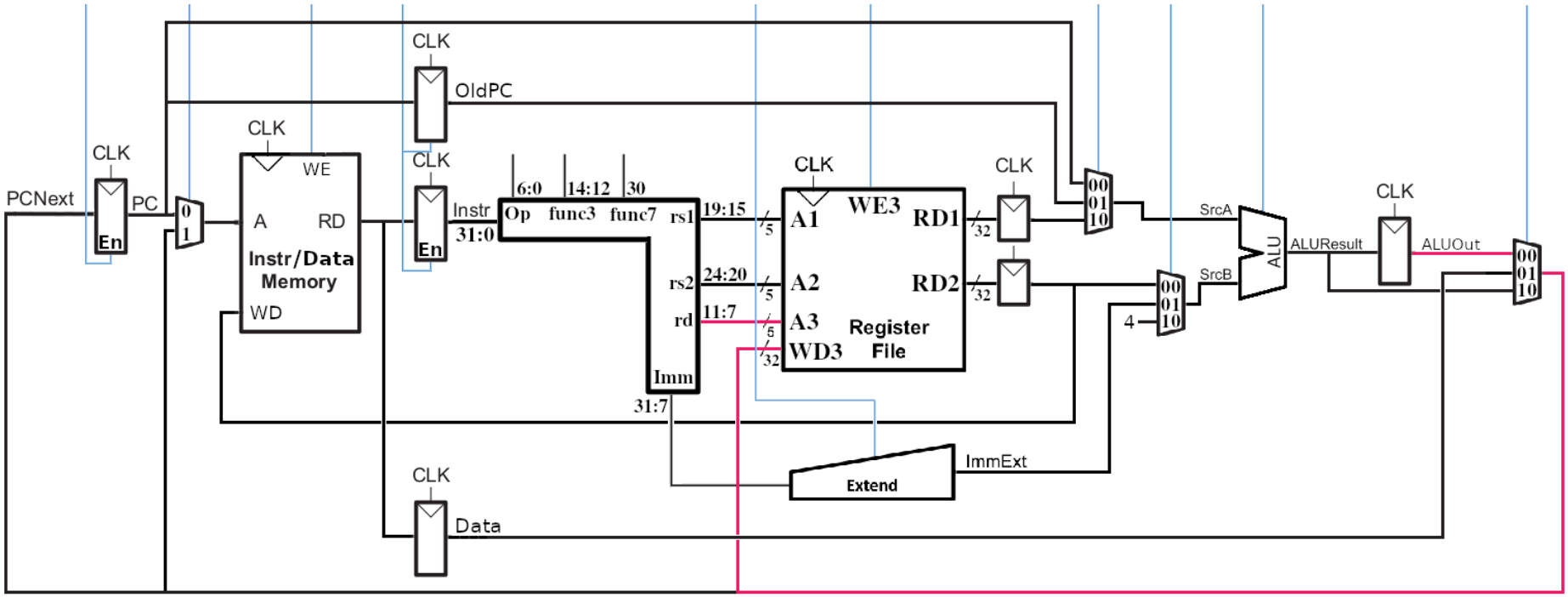
	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	$ALUControl_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10
Decode:										
ExecuteR:							10	00	???	



R-Type Instruction

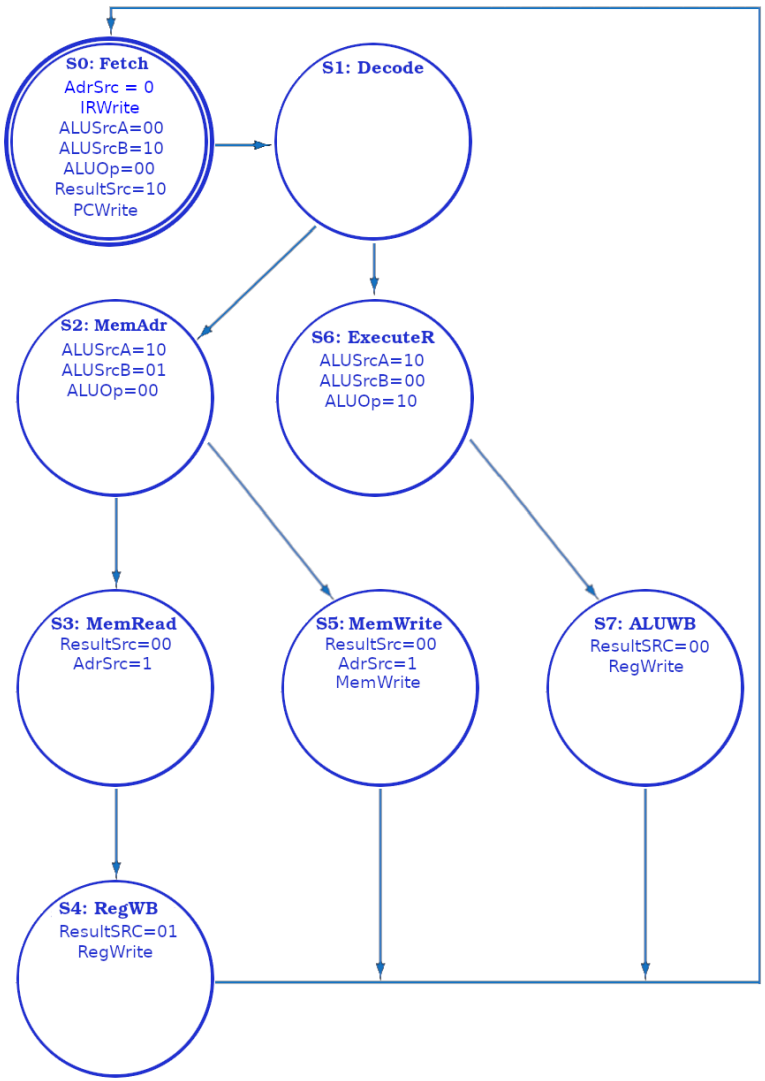
- `add rd, rs1, rs2`

	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	$ALUControl_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10
Decode:										
ExecuteR:							10	00	???	
ALUWB:						1				00



R-Type Instruction

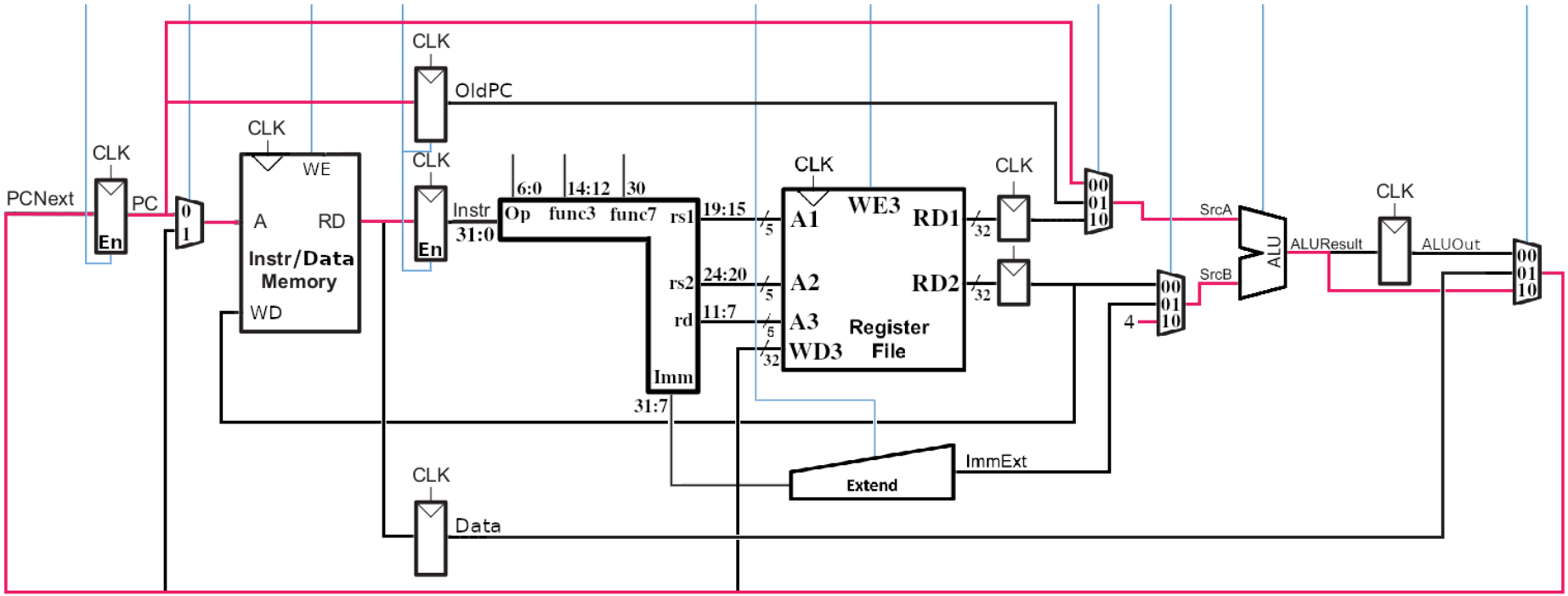
- `add rd, rs1, rs2`



I-Type Instruction

- `addi rd, rs1, imm`

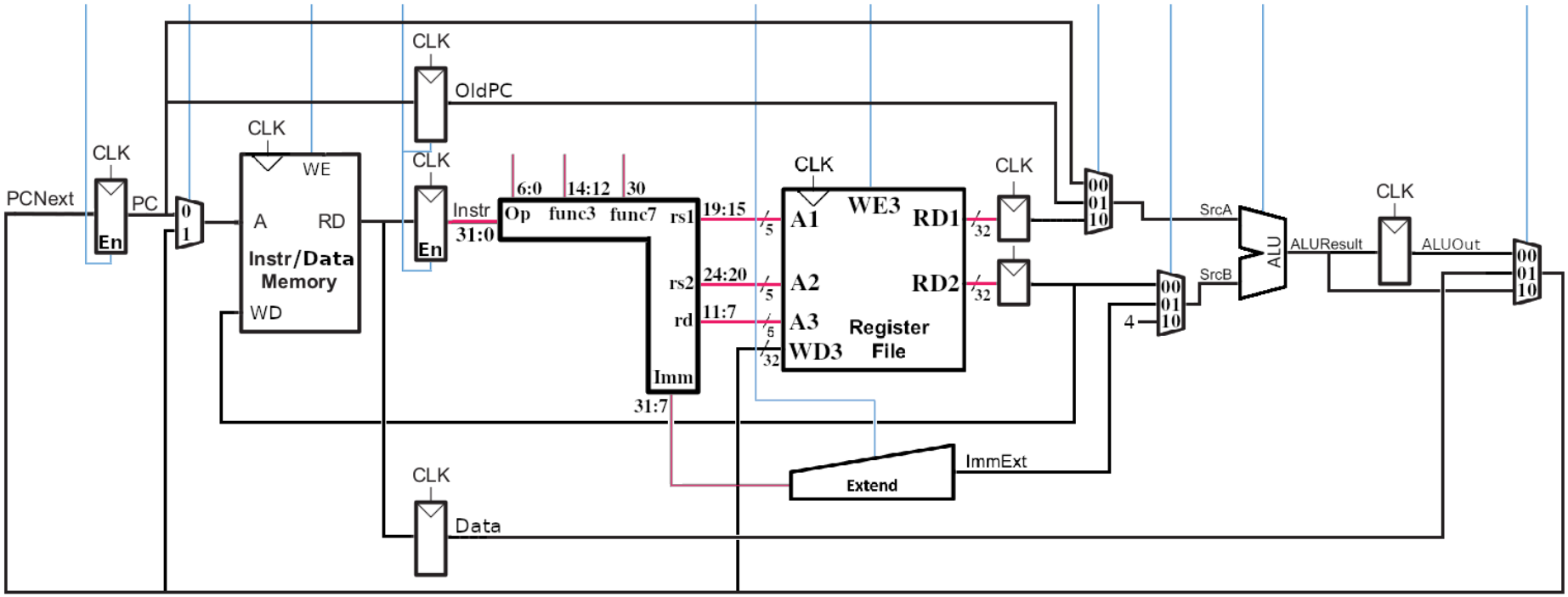
	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	$\text{ALUControl}_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10



I-Type Instruction

- `addi rd, rs1, imm`

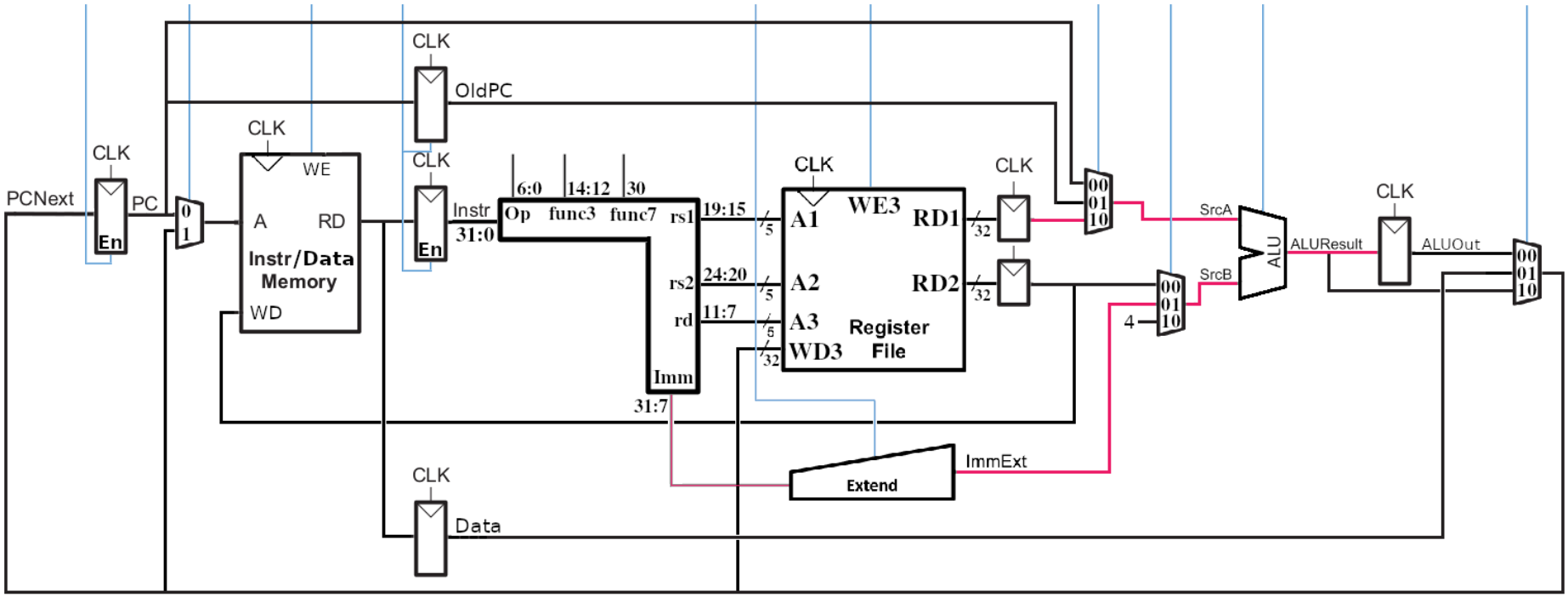
	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	$ALUControl_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10
Decode:										



I-Type Instruction

- `addi rd, rs1, imm`

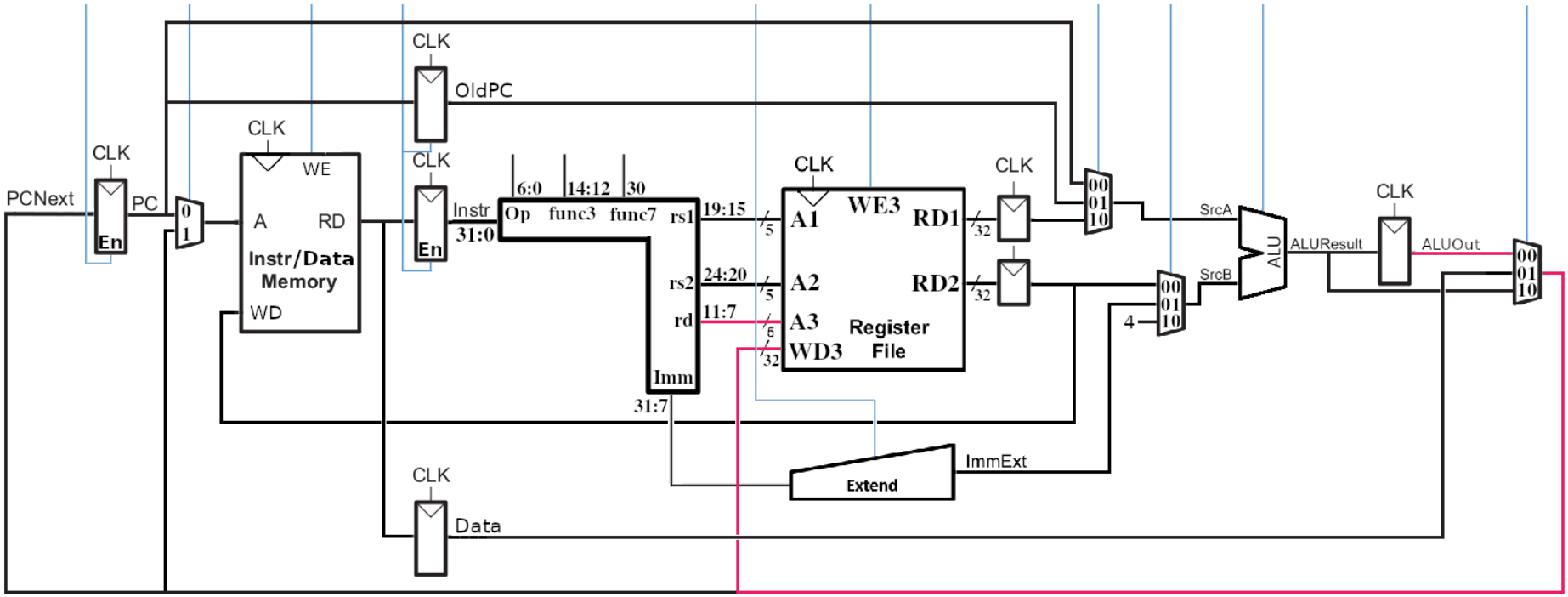
	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	$ALUControl_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10
Decode:										
ExecuteI:							10	01	???	



I-Type Instruction

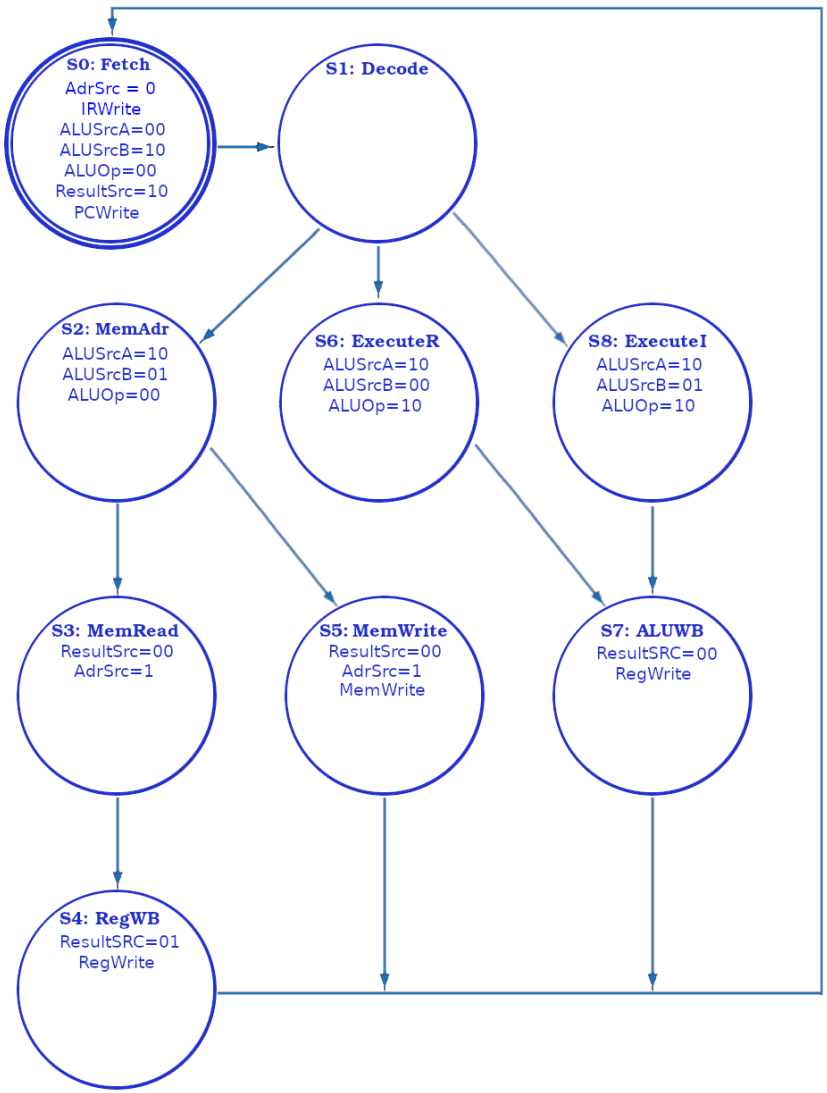
- `addi rd, rs1, imm`

	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	$ALUControl_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10
Decode:										
ExecuteI:							10	01	???	
ALUWB:						1				00



I-Type Instruction

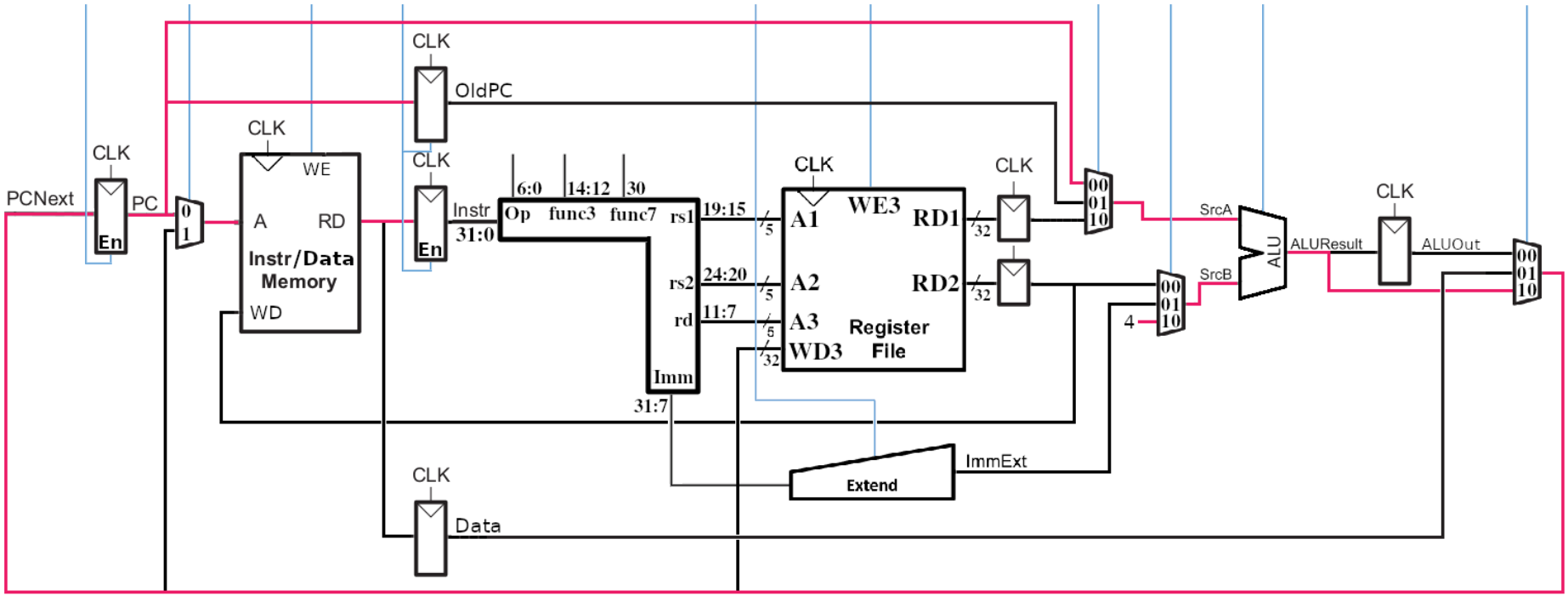
- `addi rd, rs1, imm`



beq Instruction

- `beq rs1, rs2, imm`

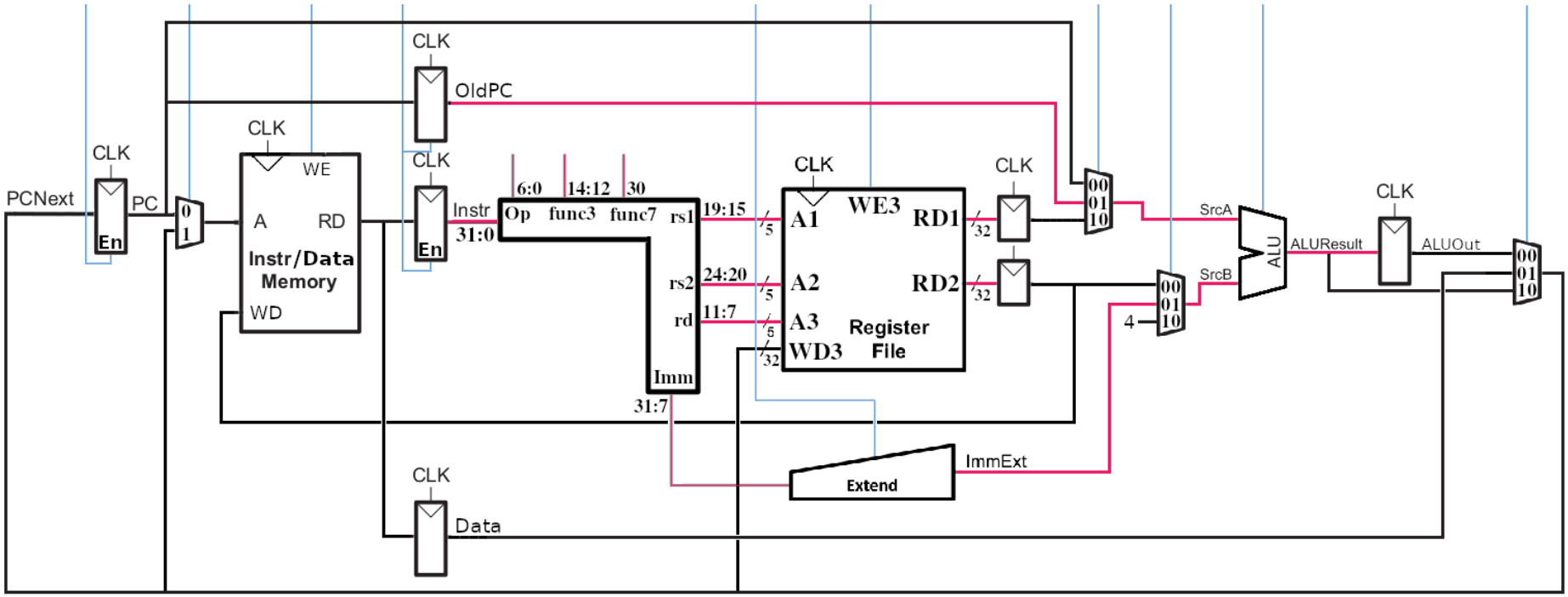
	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	$ALUControl_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10



beq Instruction

- `beq rs1, rs2, imm`

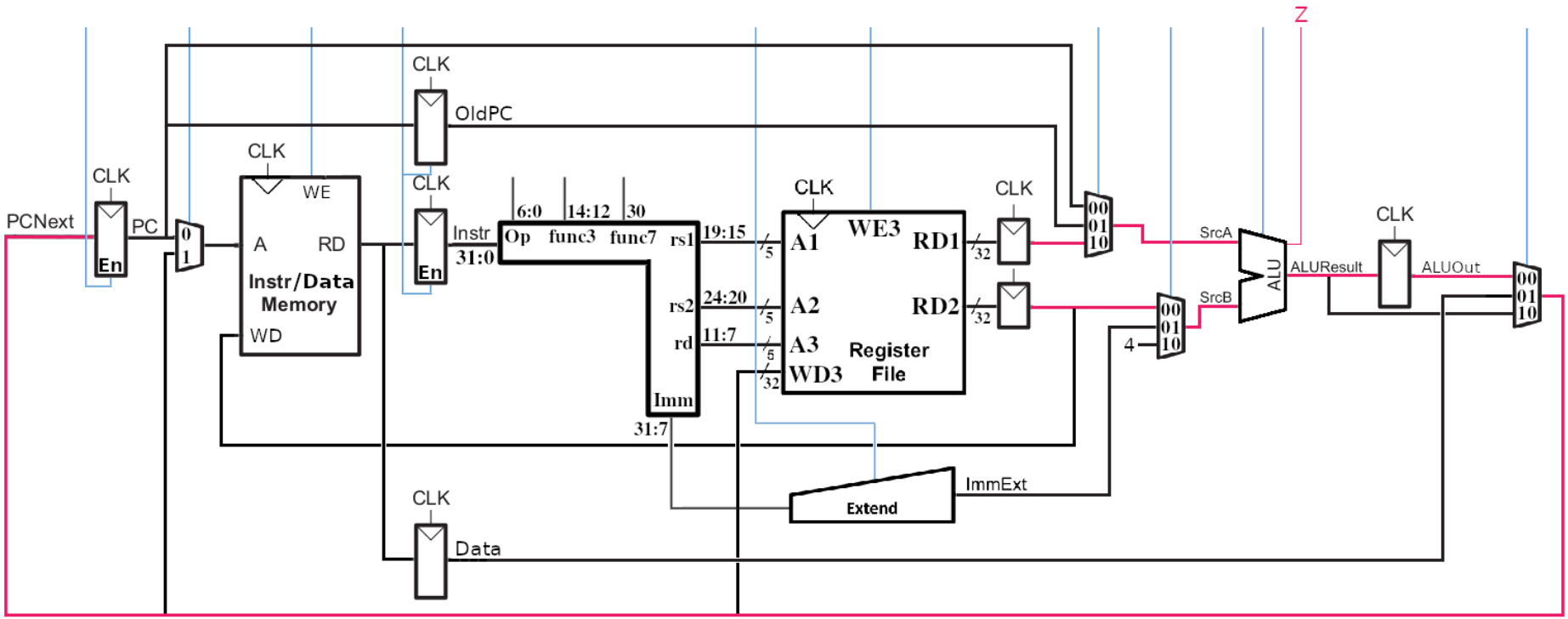
	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	ALUControl$_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10
Decode:					10		01	01	000	



beq Instruction

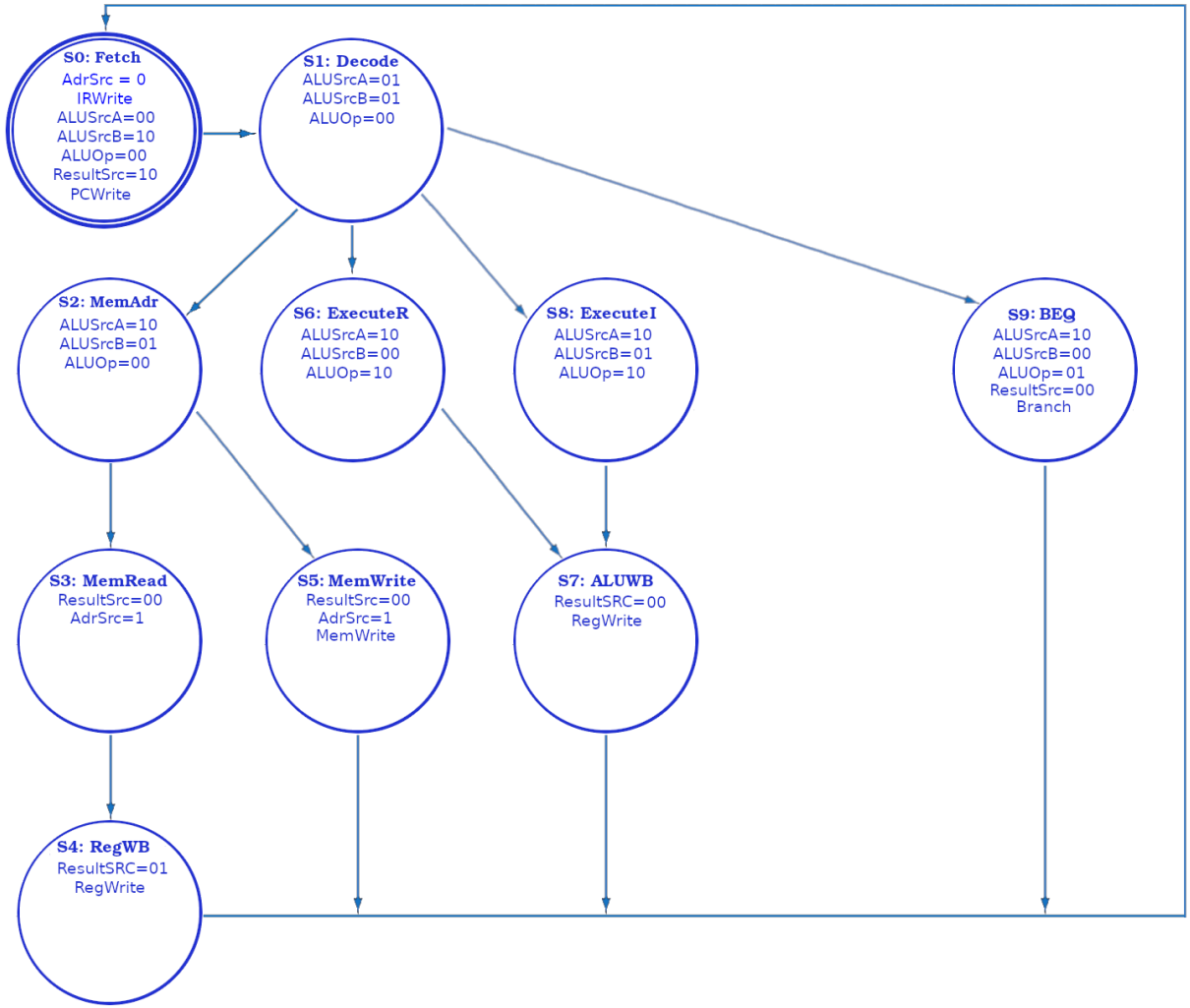
- `beq rs1, rs2, imm`

	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	ALUControl$_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10
Decode:					10		01	01	000	
BEQ: Branch & Z							10	00	001	00



beq Instruction

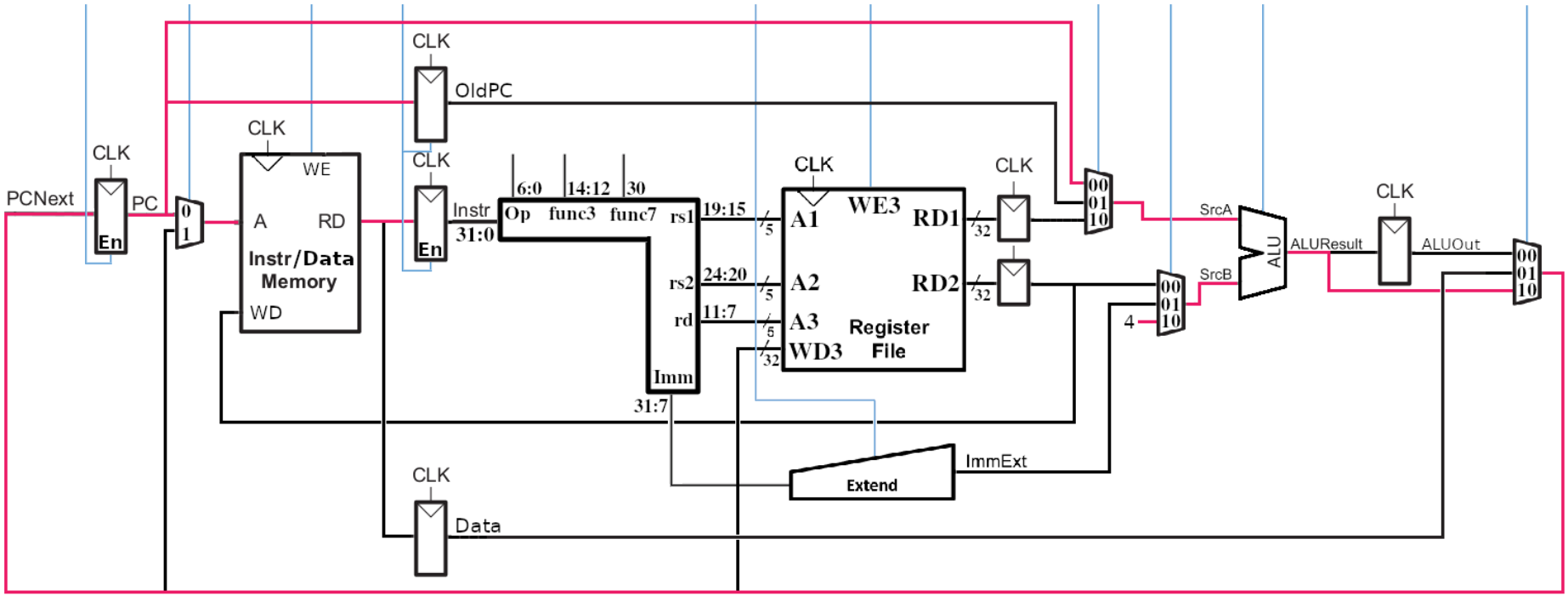
- `beq rs1, rs2, imm`



jal Instruction

- `jal rd, imm`

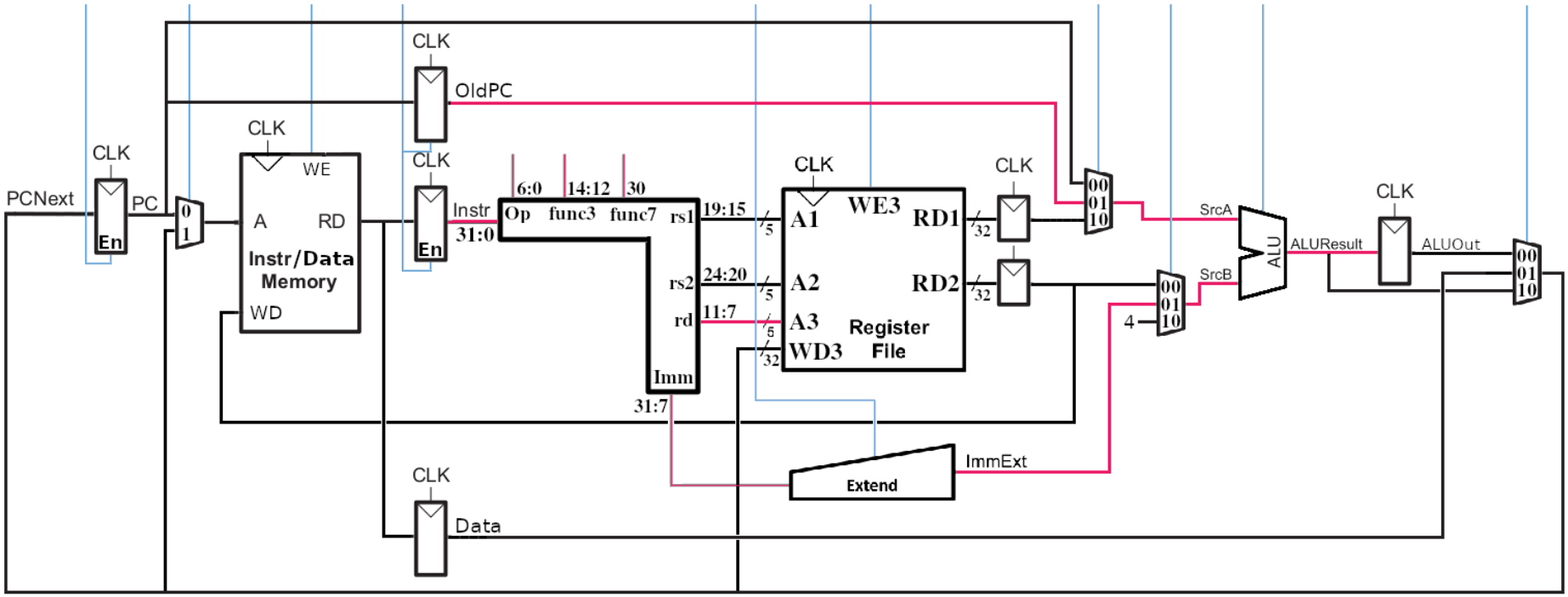
	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	$ALUControl_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10



jal Instruction

- `jal rd, label`

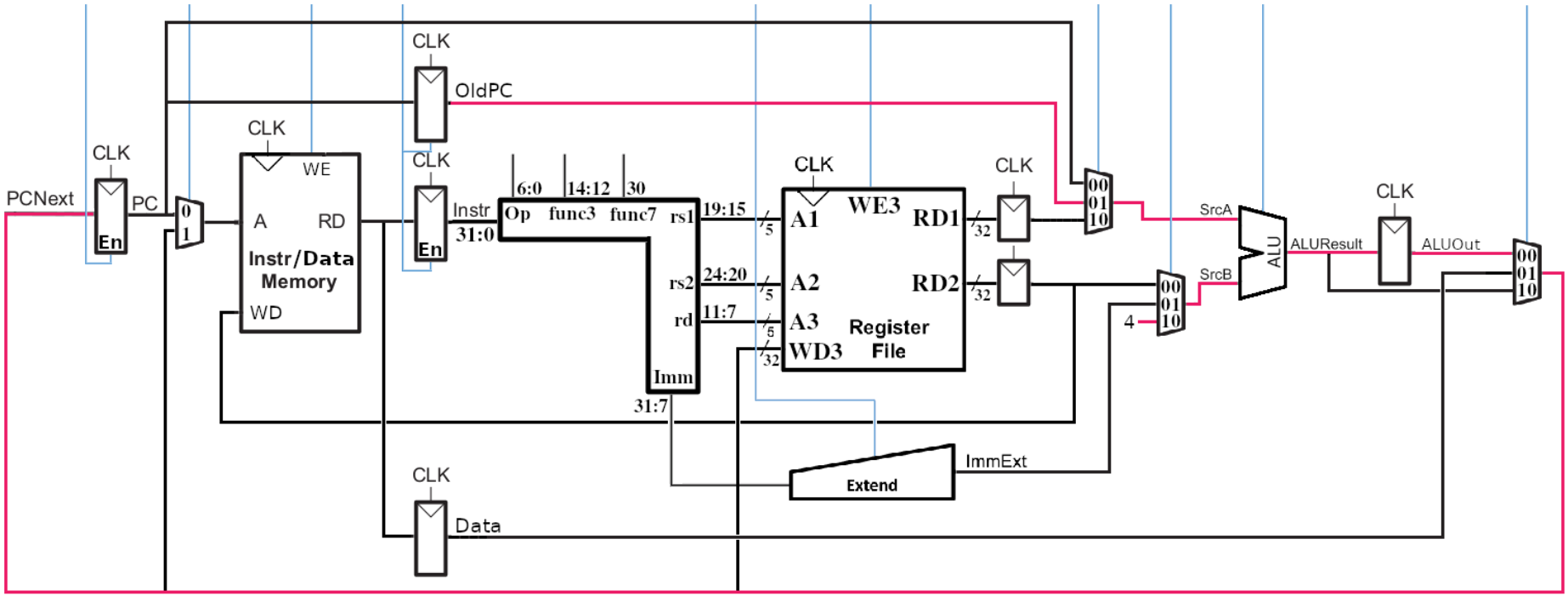
	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	$\text{ALUControl}_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10
Decode:					11		01	01	000	



jal Instruction

- `jal rd, label`

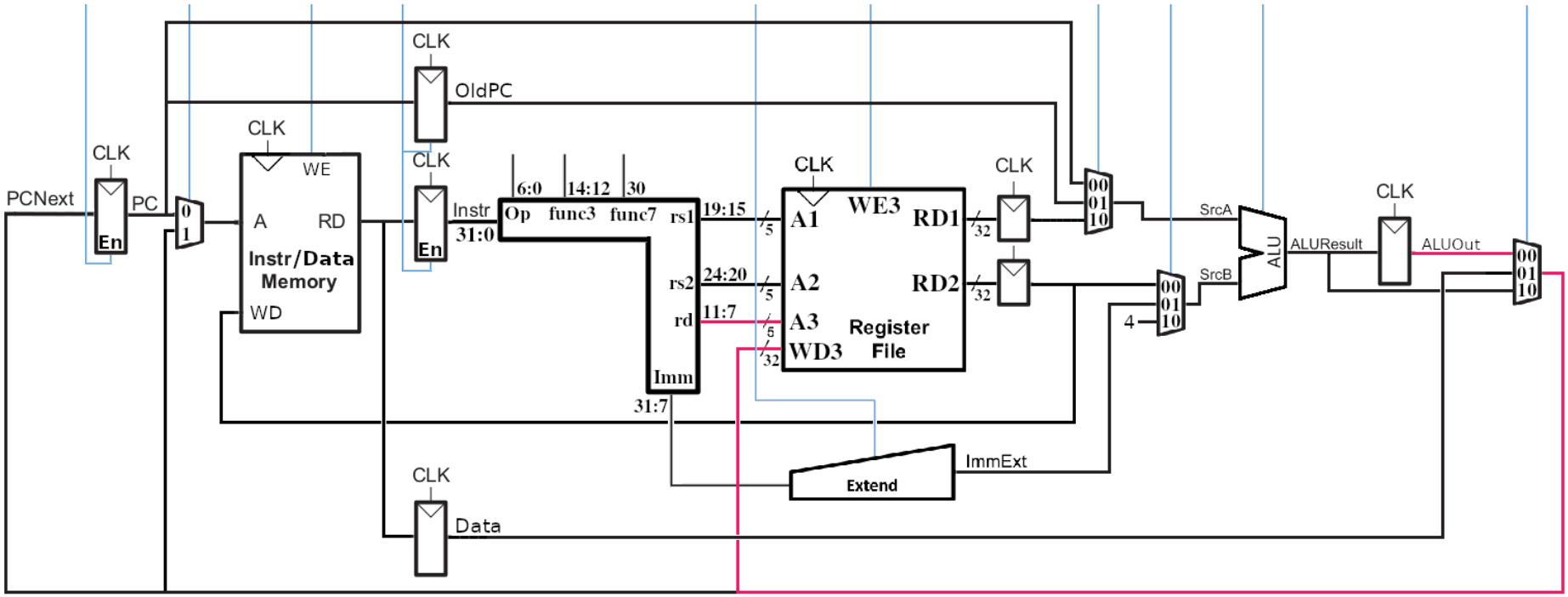
	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	$ALUControl_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10
Decode:					11		01	01	000	
JAL:	1						01	10	000	00



jal Instruction

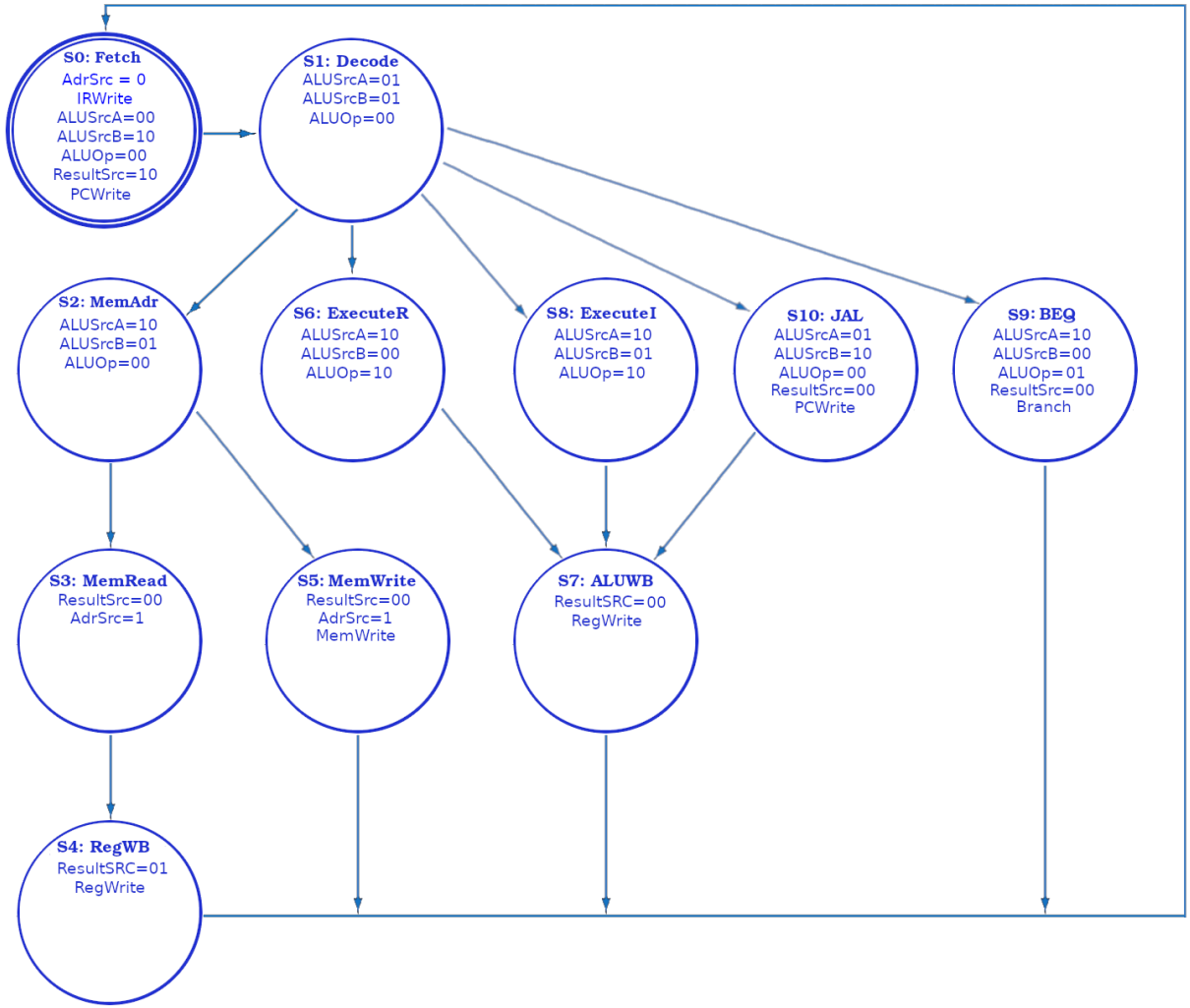
- `jal rd, label`

	PCWrite	AdrSrc	MemWrite	IRWrite	ImmSrc	RegWrite	ALUSrcA	ALUSrcB	$ALUControl_{2:0}$	ResultSrc
Fetch:	1	0		1			00	10	000	10
Decode:					11		01	01	000	
JAL:	1						01	10	000	00
ALUWB:						1				00



jal Instruction

- `jal rd, label`



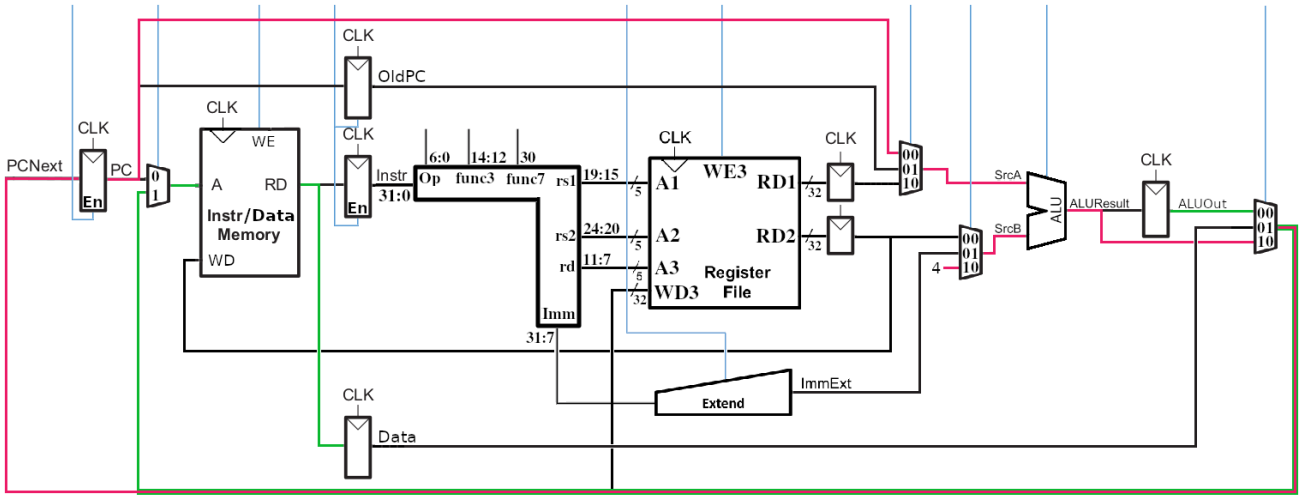
Performance Analysis

- The CPU performance

$$ExecutionTime = (\#Inst.) \times (CPI) \times \frac{1}{F_{clock}}$$

- The cycle time is set by the critical cycle path
- The longest cycle path

$$T_{multi} = t_{pcq} + t_{dec} + 2\,t_{mux} + \max\{t_{mem}, t_{ALU}\} + t_{setup}$$



Performance Analysis

- Given the following delay table,

$$T_{multi} = t_{pcq} + t_{dec} + 2\,t_{mux} + \max\{t_{mem}, t_{ALU}\} + t_{setup}$$
$$= 40 + 25 + 2 \times 30 + 200 + 60 = 385ps$$

Element	Parameter	Delay (ps)
Register clk-to-Q	t_{pcq}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	$t_{AND\text{-}OR}$	20
ALU	t_{ALU}	120
Decoder (control unit)	t_{dec}	25
Extend unit	t_{ext}	35
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

Performance Analysis

- How long does it take to execution a 100 billion instructions code, if it is composed of 25% Loads, 10% Stores, 11% Branches, 2% Jumps, 52% R- or I-Type instructions?
- According to the state diagram,
  - **Load**: 5 cycles
  - **Store**: 4 cycles
  - **R-Type**: 4 cycles
  - **I-Type**: 4 cycles
  - **Jump**: 4 cycles
  - **Branch**: 3 cycles
- Therefore we can calculate,

$$CPI = 0.25 \times 5 + 0.1 \times 4 + 0.11 \times 3 + 0.02 \times 4 + 0.52 \times 4 = 4.14$$

- Execution time can be calculated as,

$$\begin{aligned} ExecutionTime &= (\#Inst.) \times (CPI) \times \frac{1}{F_{clock}} \\ &= 10^{11} \times 4.14 \times 385 \times 10^{-12} = 159.39\, sec \end{aligned}$$