National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lecture 6: Multi-Cycle CPU
CS10014 Computer Organization

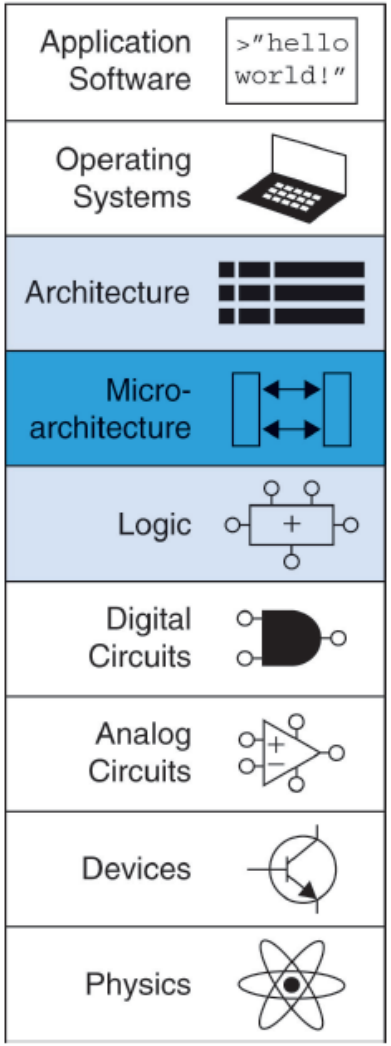
Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao Tung University

Acknowledgements and Disclaimer

- Slides were developed in the reference with
  - CS 61C at UC Berkeley
    - https://inst.eecs.berkeley.edu/~cs61c/sp23/
  - CS 252 at UC Berkeley
    - https://people.eecs.berkeley.edu/~culler/courses/cs252-s05/
  - E85 at HMC
    - https://pages.hmc.edu/harris/class/e85/old/fall21/

Outline

- Single-Cycle Processor
- Multi-Cycle Processor

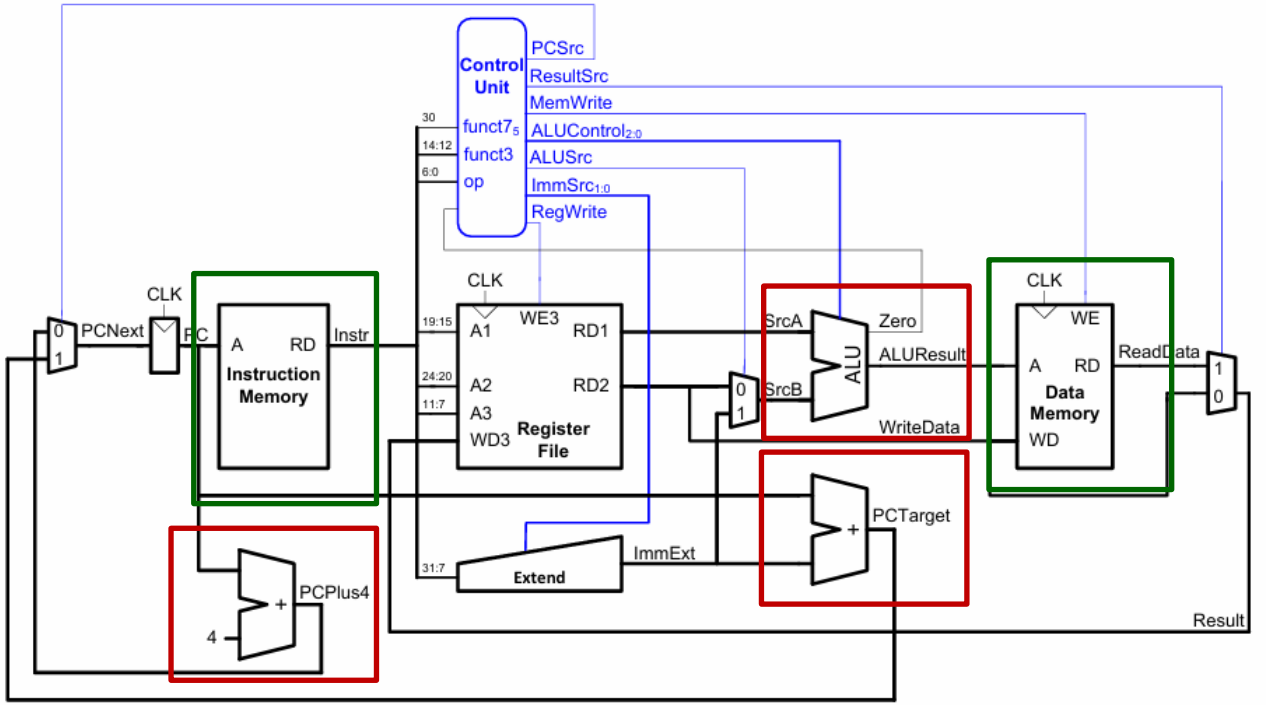


Single- vs. Multicycle Processor

- Single-Cycle Processor
  - Separate instruction and data memories
    - Read the instruction memory and read or write the data memory all in one cycle
  - Require a clock cycle long enough to support the slowest instruction (memory load), even though most instructions could be faster
  - 3 adders (one in the ALU and two for the PC logic)
    - Adders are relatively expensive circuits, especially if they must be fast

Single- vs. Multicycle Processor

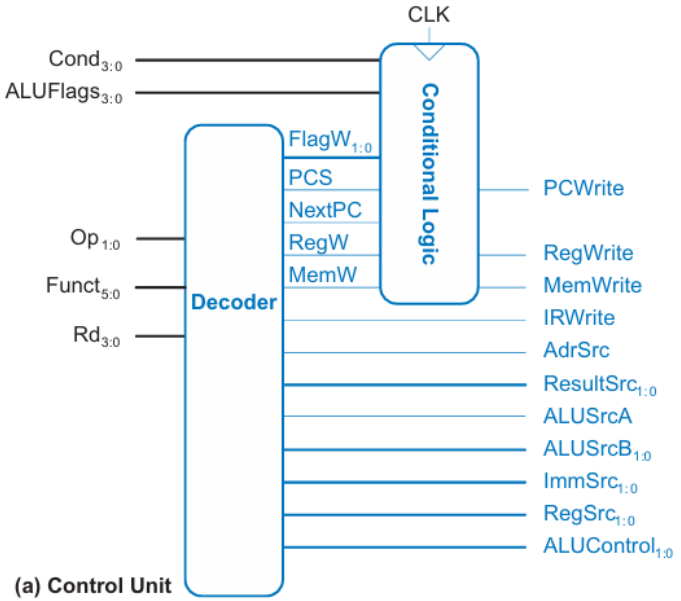
- Single-Cycle Processor



a. 3 adders
b. Separate inst/data memory

Single- vs. Multicycle Processor

- Single-Cycle Processor controller
  - Decoder + combinational logic



Single- vs. Multicycle Processor

- Multi-Cycle Processor
  - Combined memory for both instructions and data → feasible
    - Read the instruction in one cycle, then read or write the data in a separate cycle
  - Break an instruction into multiple shorter steps
    - The processor can read or write the memory or register file or use the ALU
    - Simpler instructions can complete faster than complex ones
  - One adder -> reused on different steps

Single- vs. Multicycle Processor

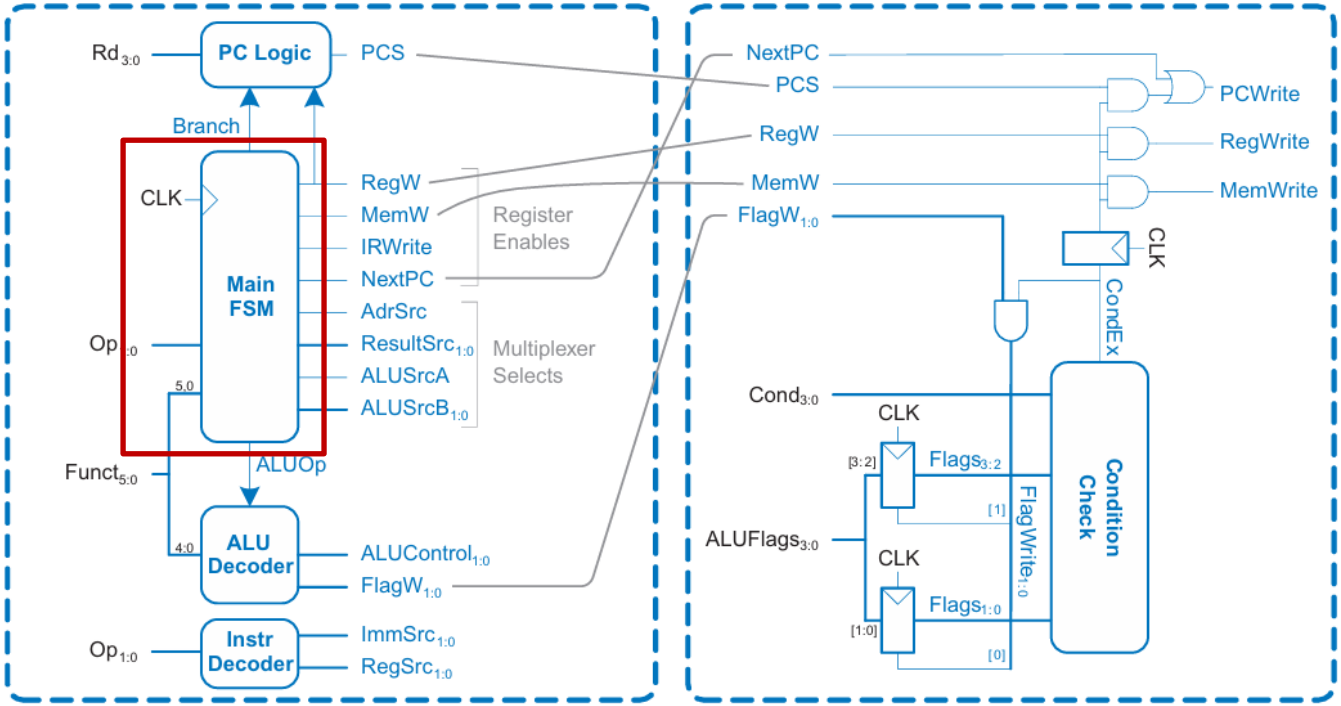
- Multi-Cycle Processor
  - One adder -> reused on different steps
    - Add a new nonarchitectural instruction register (IR) to hold intermediate results between steps

Single- vs. Multicycle Processor

- Multi-Cycle Processor
  - The controller produces different signals on different steps during execution of a single instruction
    - Use a finite state machine (FSM) rather than combinational logic

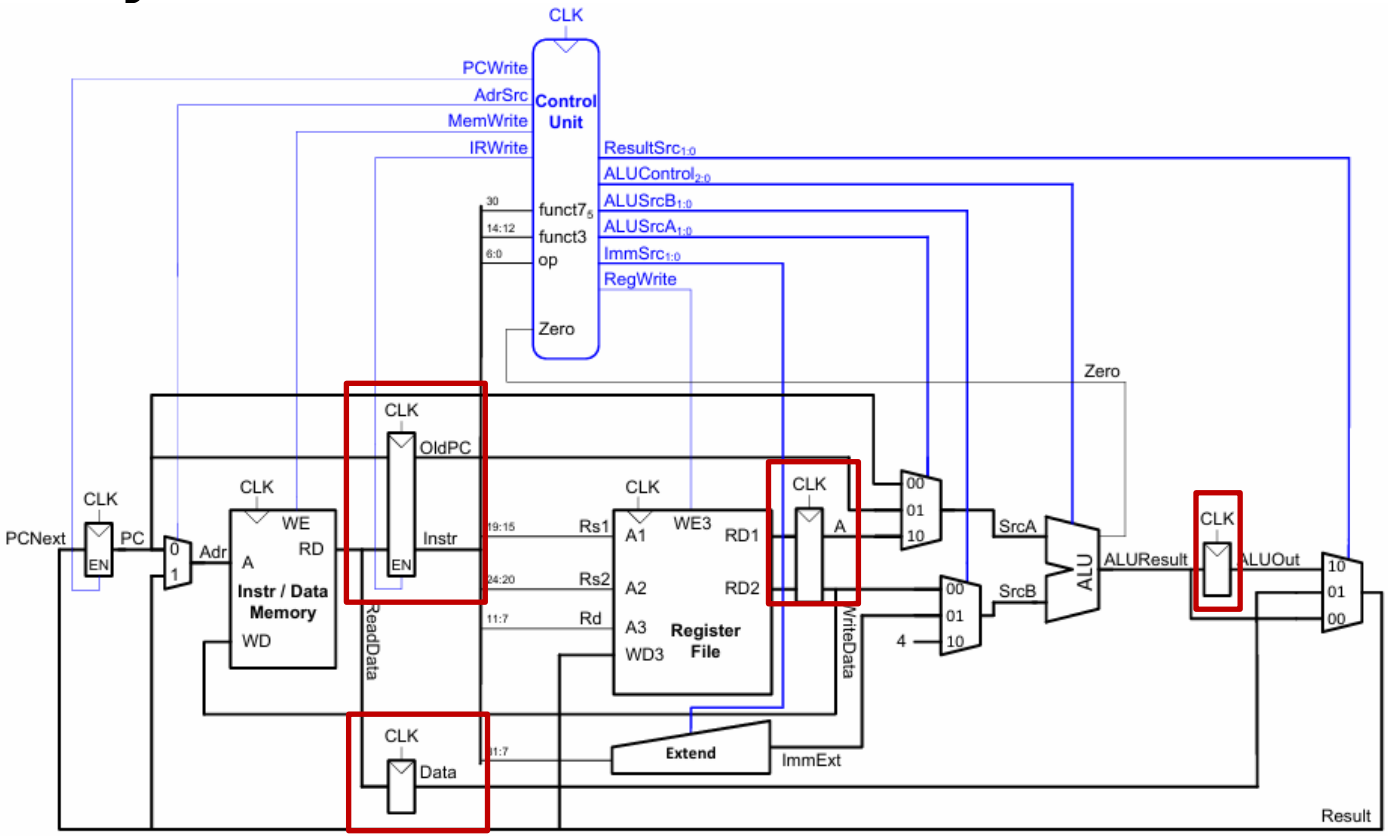
Single- vs. Multicycle Processor

- MultiCycle Processor controller



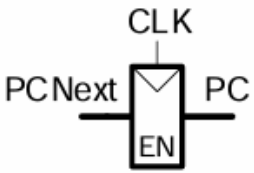
Multi-Cycle Processor

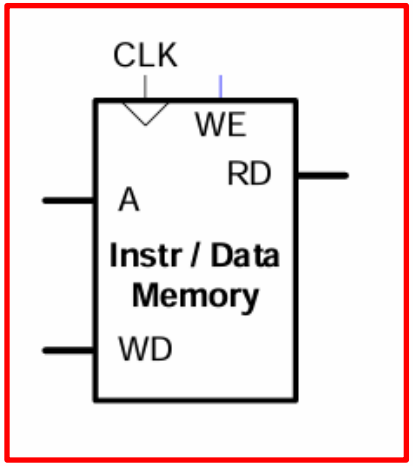
a. no

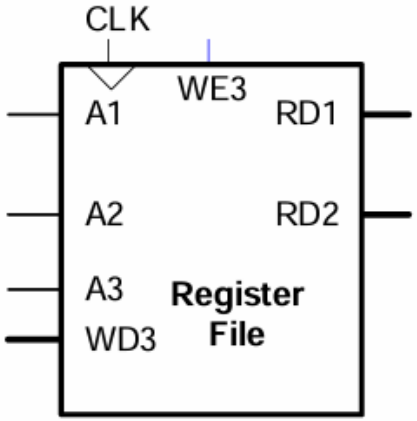


Multi-Cycle State Elements

Replace separate Instruction and Data memories with a **single unified memory** – more realistic

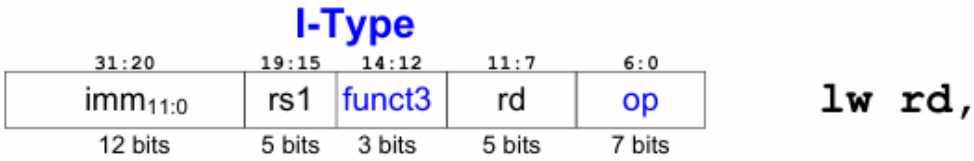




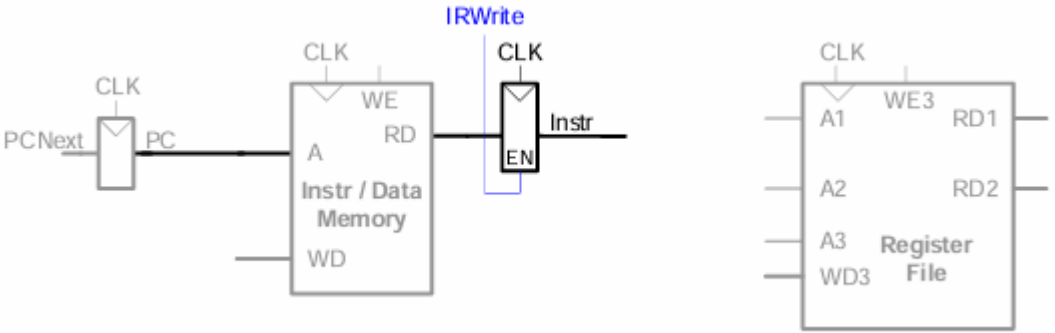


Multi-Cycle Datapath: Instruction Fetch

Step 1: Read `lw` instruction from Instr memory

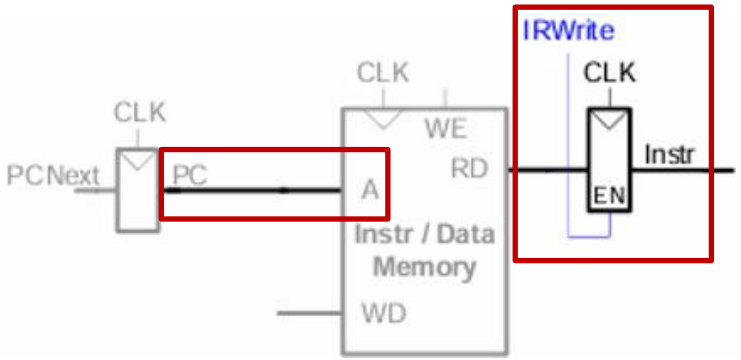



```
lw rd, imm(rs1)
```



Multi-Cycle Datapath: Instruction Fetch

- Step 1: Read lw instruction from Instr memory
  - The PC contains the address of the instruction to execute
  - The PC is connected to the address input of the memory
  - The instruction is read and stored in IR so that it is available for future cycles
  - The IR receives an enable signal, called IRWrite, which is asserted when the IR should be loaded with a new instruction



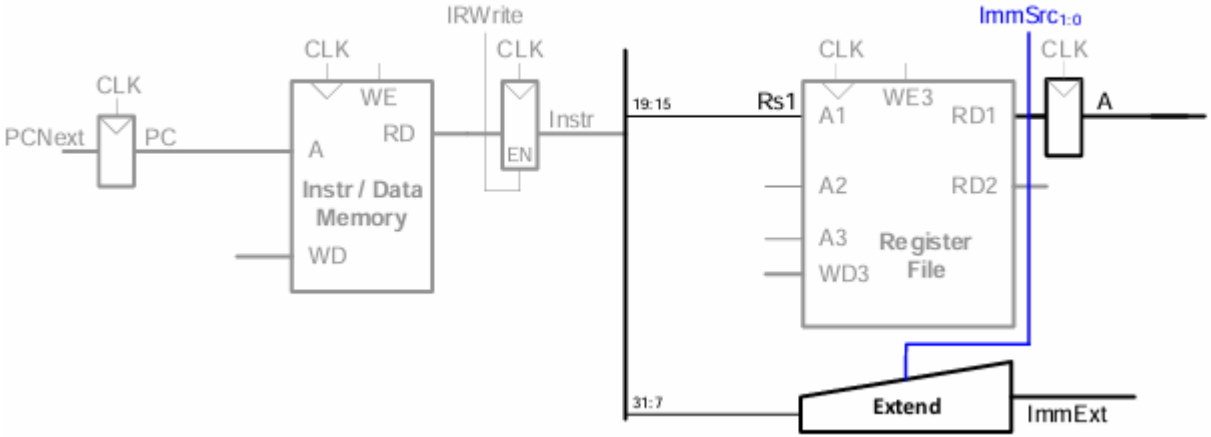
Multi-Cycle Datapath: `lw`

STEP 2: Read source operand from RF and extend immediate

I-Type

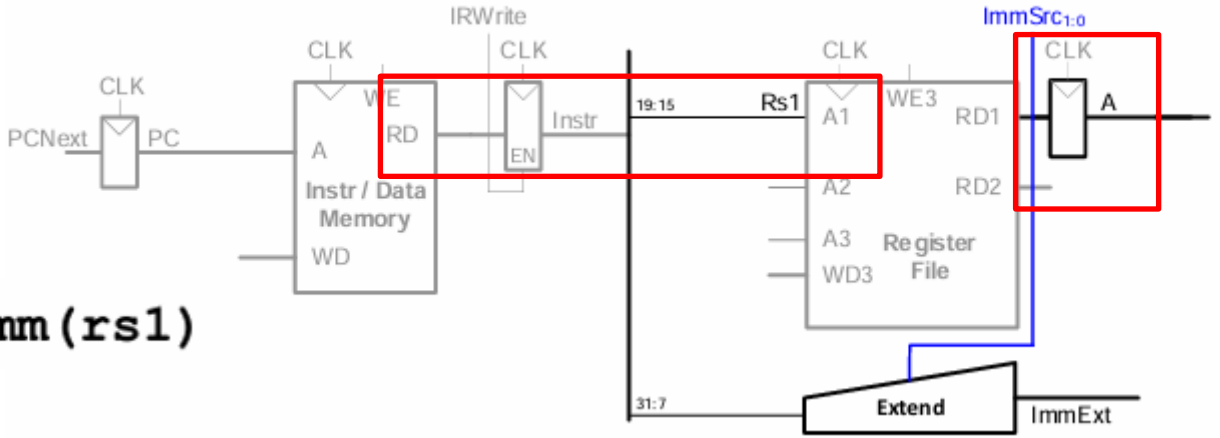
31:20	19:15	14:12	11:7	6:0
$imm_{11:0}$	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

```
lw rd, imm(rs1)
```



Multi-Cycle Datapath: Instruction Fetch

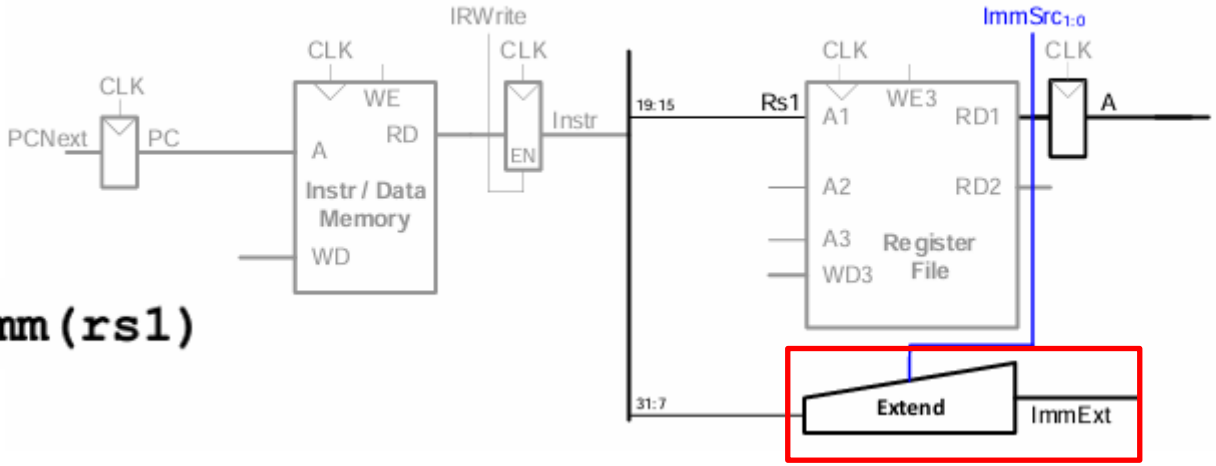
- Step 2: read the source register (`r1`) base address
  - The bits of `RD` are connected to address input `A1` of the register file
  - The register file reads the register into `RD1`
  - This value is stored in another nonarchitectural register, A




```
lw rd, imm(rs1)
```

Multi-Cycle Datapath: Instruction Fetch

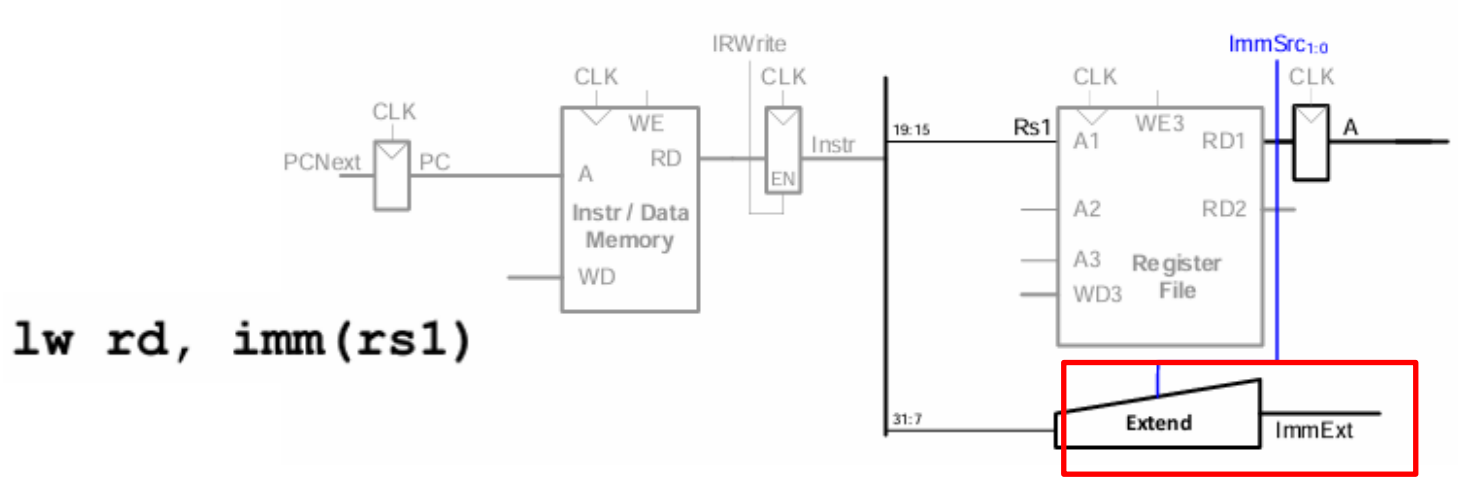
- Step 2: read the source register (`r1`) base address
  - The 12-bit immediate must be zero-extended to 32 bits
  - The 32-bit extended immediate is called ImmExt
  - Do we need to have a register to hold the this imm constant value?




```
lw rd, imm(rs1)
```

Multi-Cycle Datapath: Instruction Fetch

- Step 2: read the source register (`rs1`) base address
  - ImmExt is a combinational function of Instr and won't change while the current instruction is being processed
  - There is no need to dedicate a register to hold this constant value



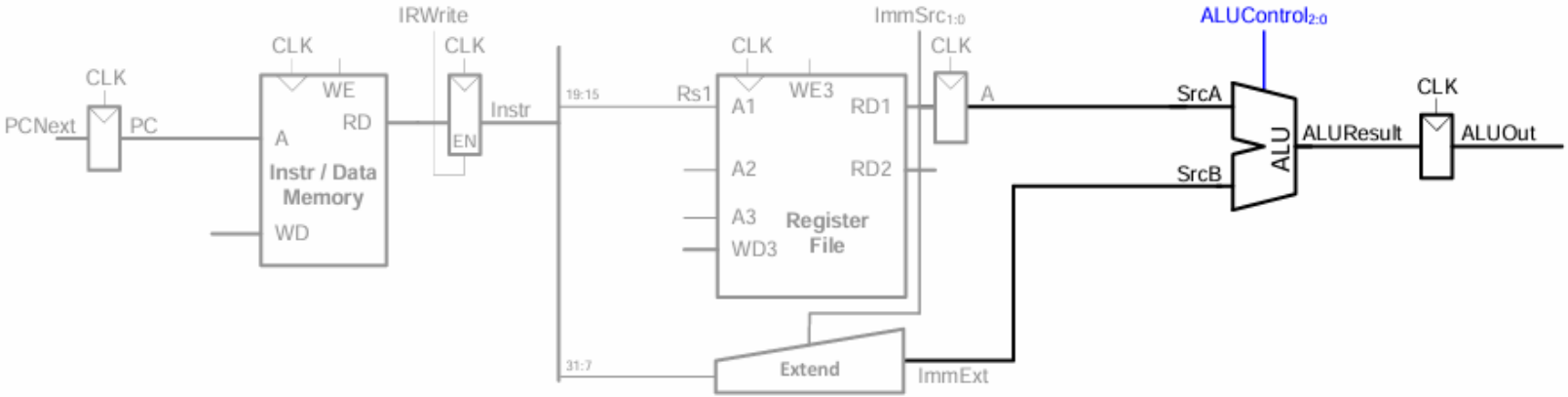
Multi-Cycle Datapath: `lw`

STEP 3: Compute the memory address

I-Type

31:20	19:15	14:12	11:7	6:0
$imm_{11:0}$	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

```
lw rd, imm(rs1)
```

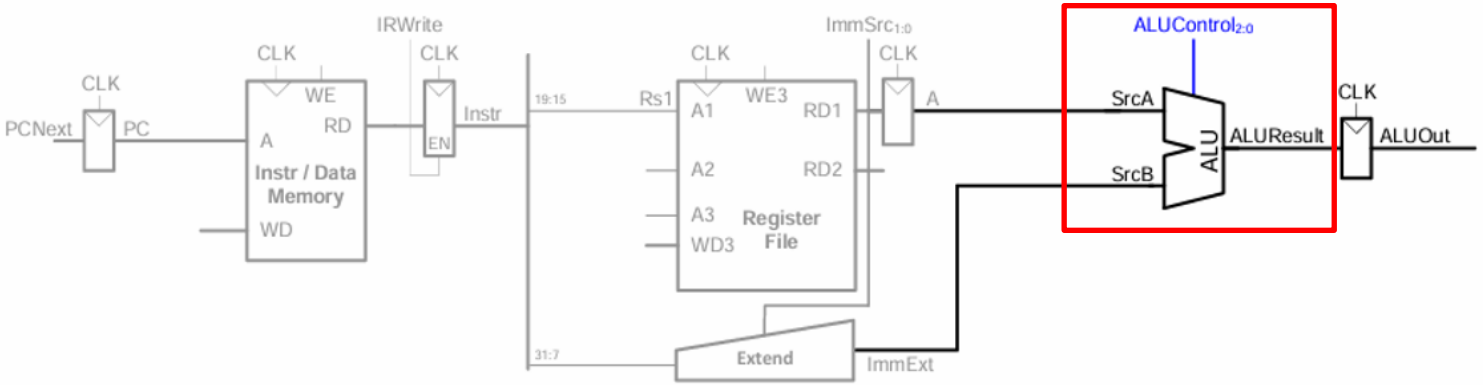


Multi-Cycle Datapath: Instruction Fetch

- Step 3: The address of the load is the sum of the based address and offset

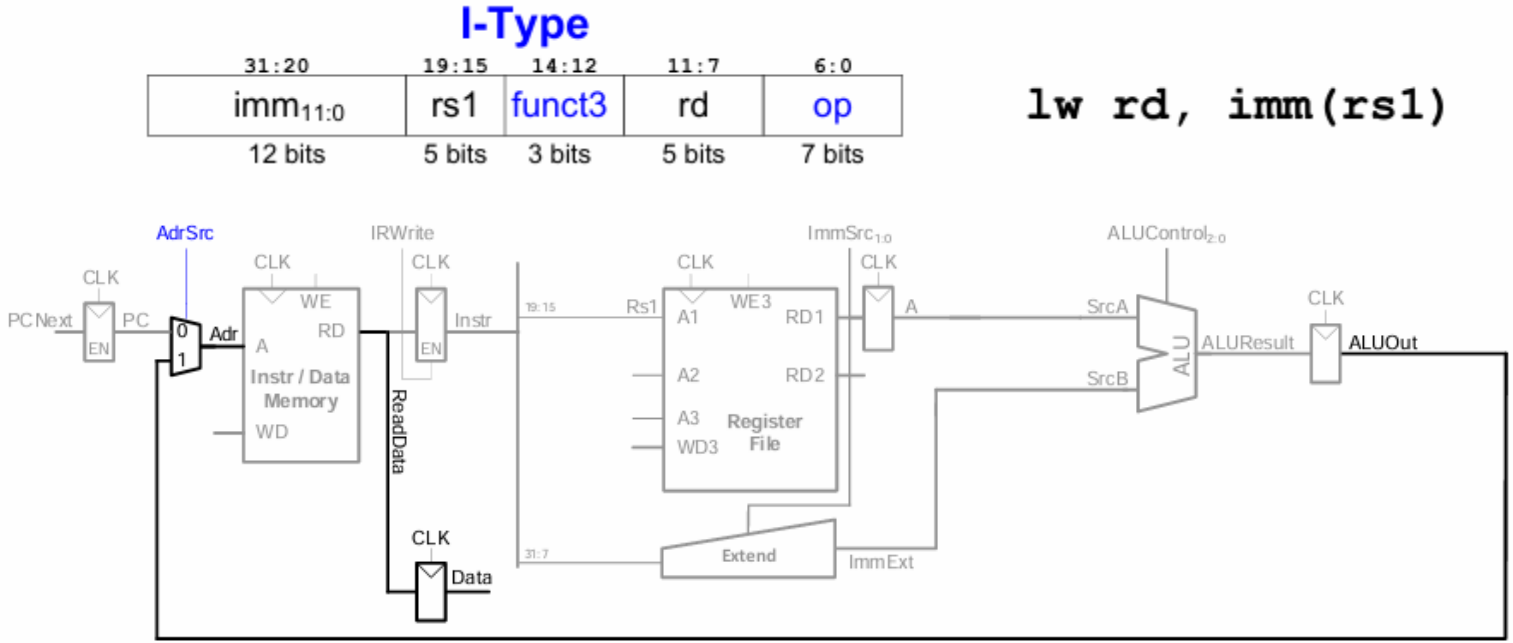
```
lw rd, imm(rs1)
```

  - Use an ALU to compute this sum
  - `ALUControl` is set to 00 to perform the addition
  - `ALUResult` is stored in an register called ALUOut



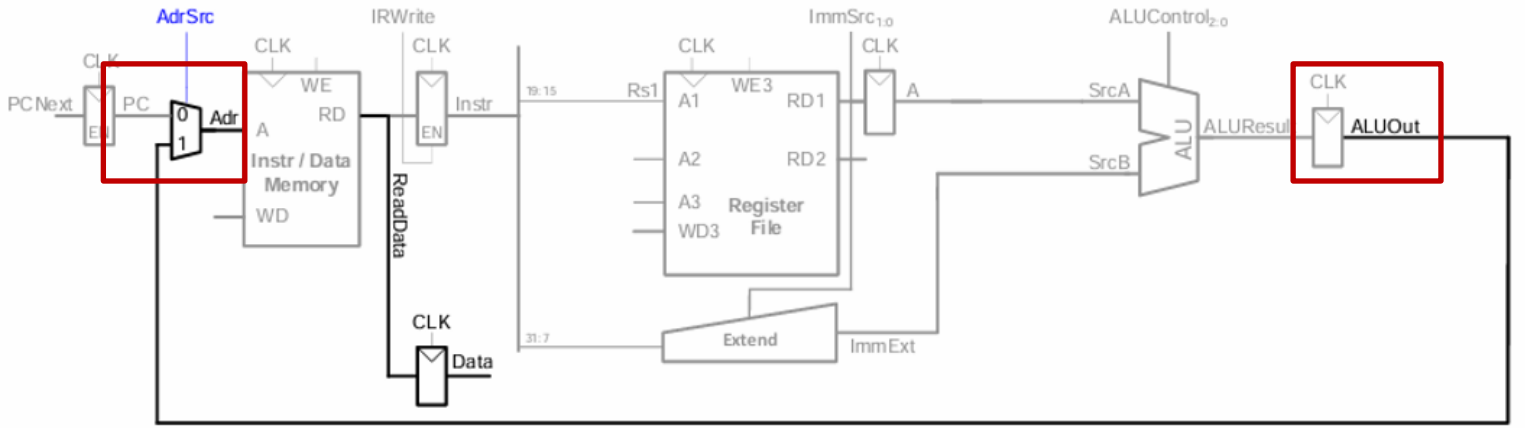
Multi-Cycle Datapath: `lw`

STEP 4: Read data from memory



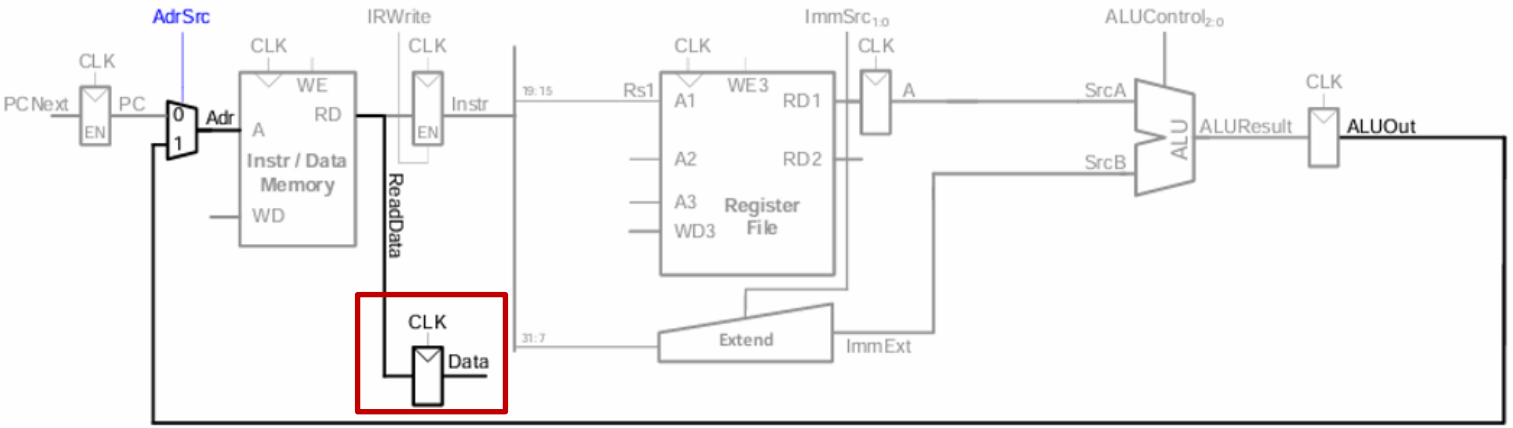
Multi-Cycle Datapath: Instruction Fetch

- Step 4: load data from the calculated address in memory
  - Add a multiplexer in front of the memory to choose the memory address, `Adr`, from either the PC or `ALUOut` based on the `AdrSrc` select



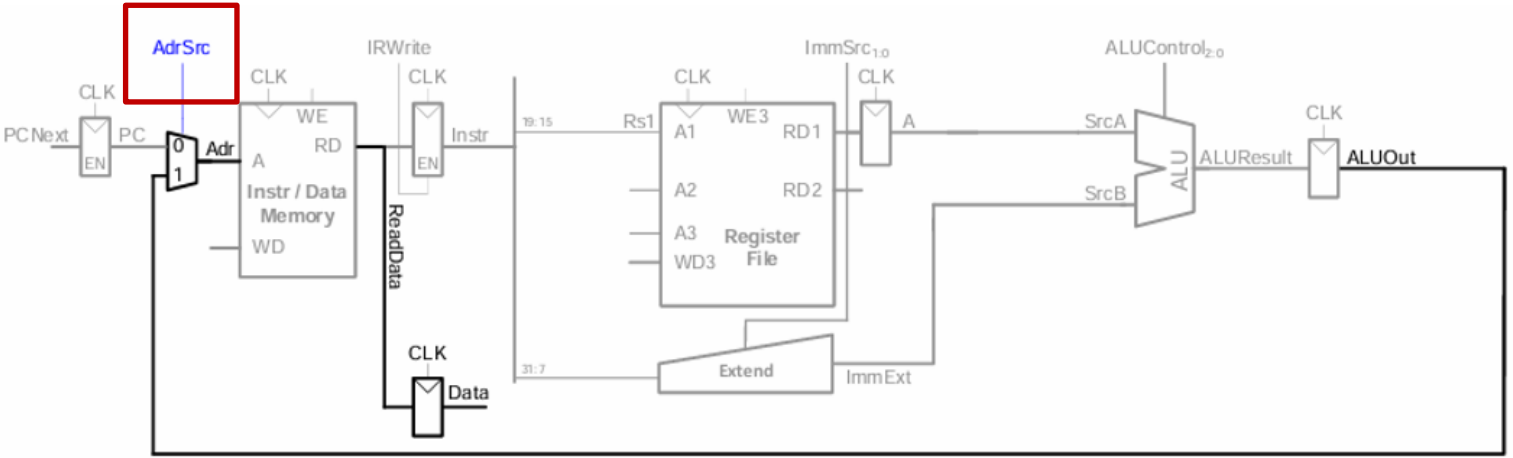
Multi-Cycle Datapath: Instruction Fetch

- Step 4: load data from the calculated address in memory
  - The data read from memory is stored in another nonarchitectural register, call `Data`
  - The address MUX permits us to reuse the memory during `lw` instr



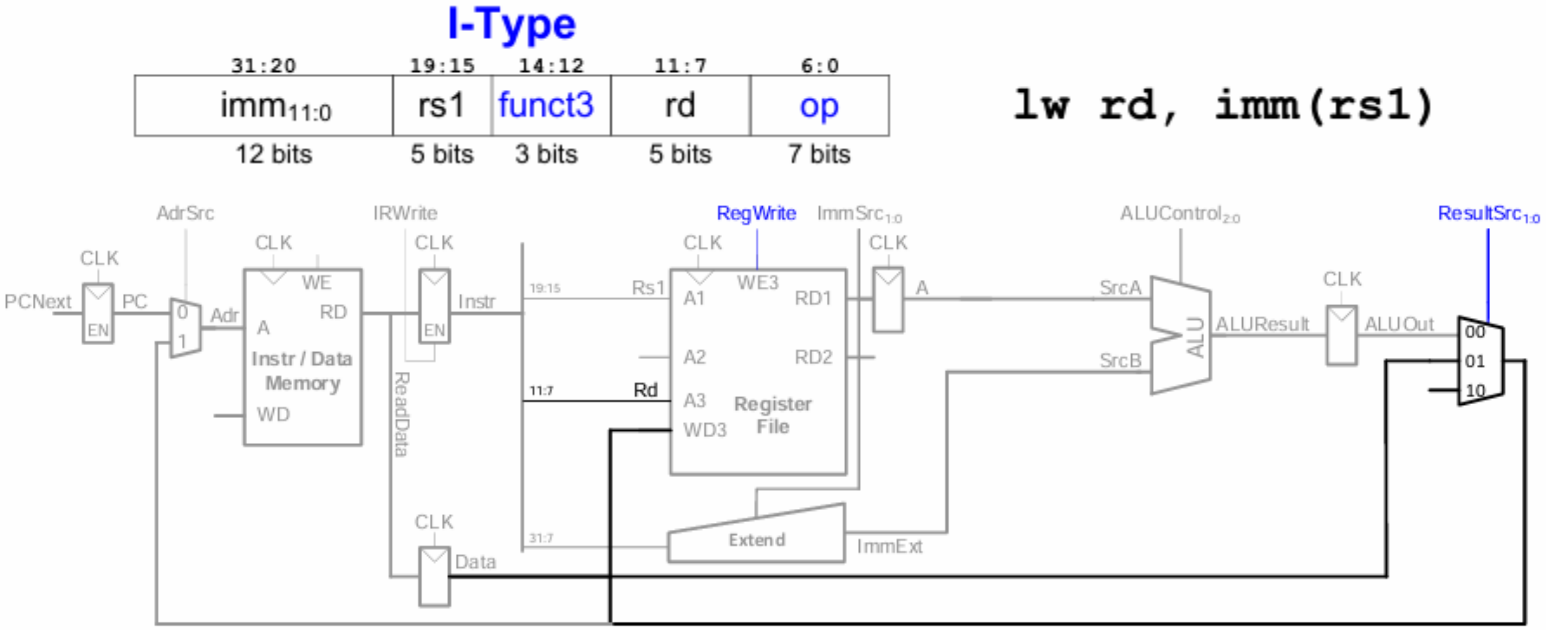
Multi-Cycle Datapath: Instruction Fetch

- Step 4: AdrSrc must have different values on different steps
  - On a first step, the address is taken from the PC to fetch a Instr
  - On a later step, the address is taken from `ALUOut` to load data
  - The FSM controller generates these sequences of control signals



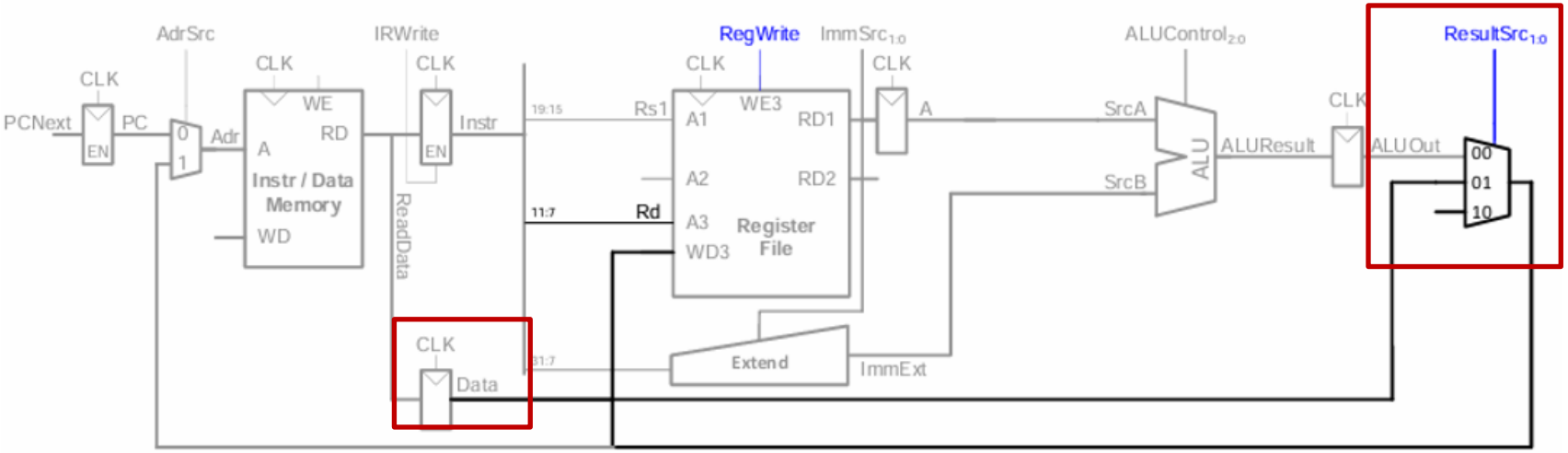
Multi-Cycle Datapath: `lw`

STEP 5: Write data back to register file



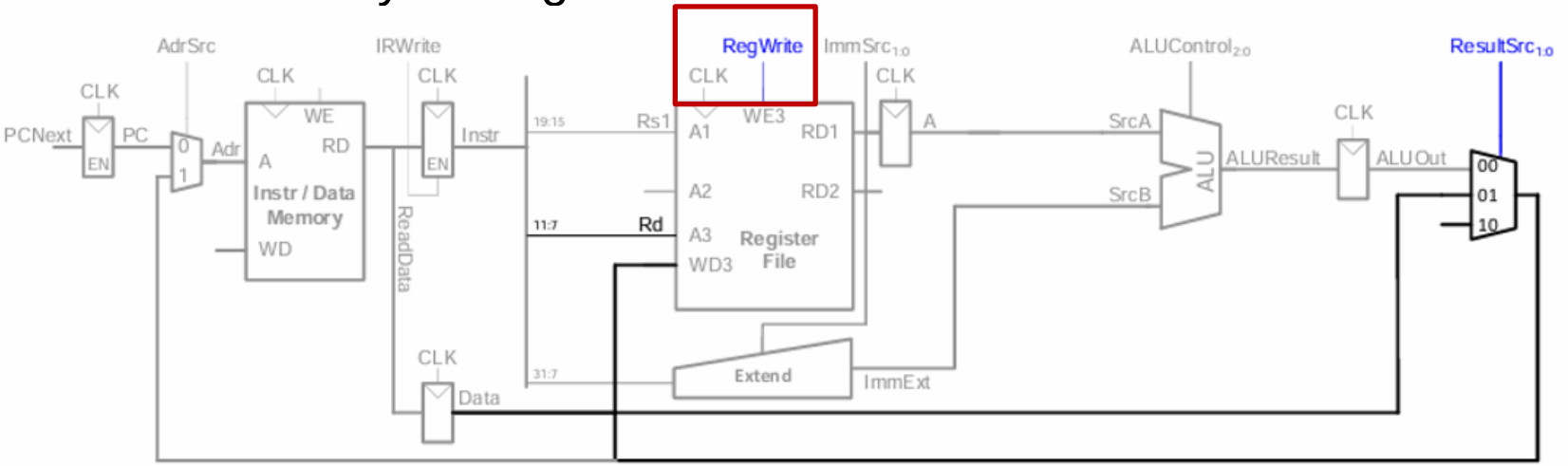
Multi-Cycle Datapath: Instruction Fetch

- Step 5: The data is written back to the register file
  - Add a MUX on the `Result` bus to choose either `ALUOut` or `Data` before feeding `Result` back to the register file WD3 write port

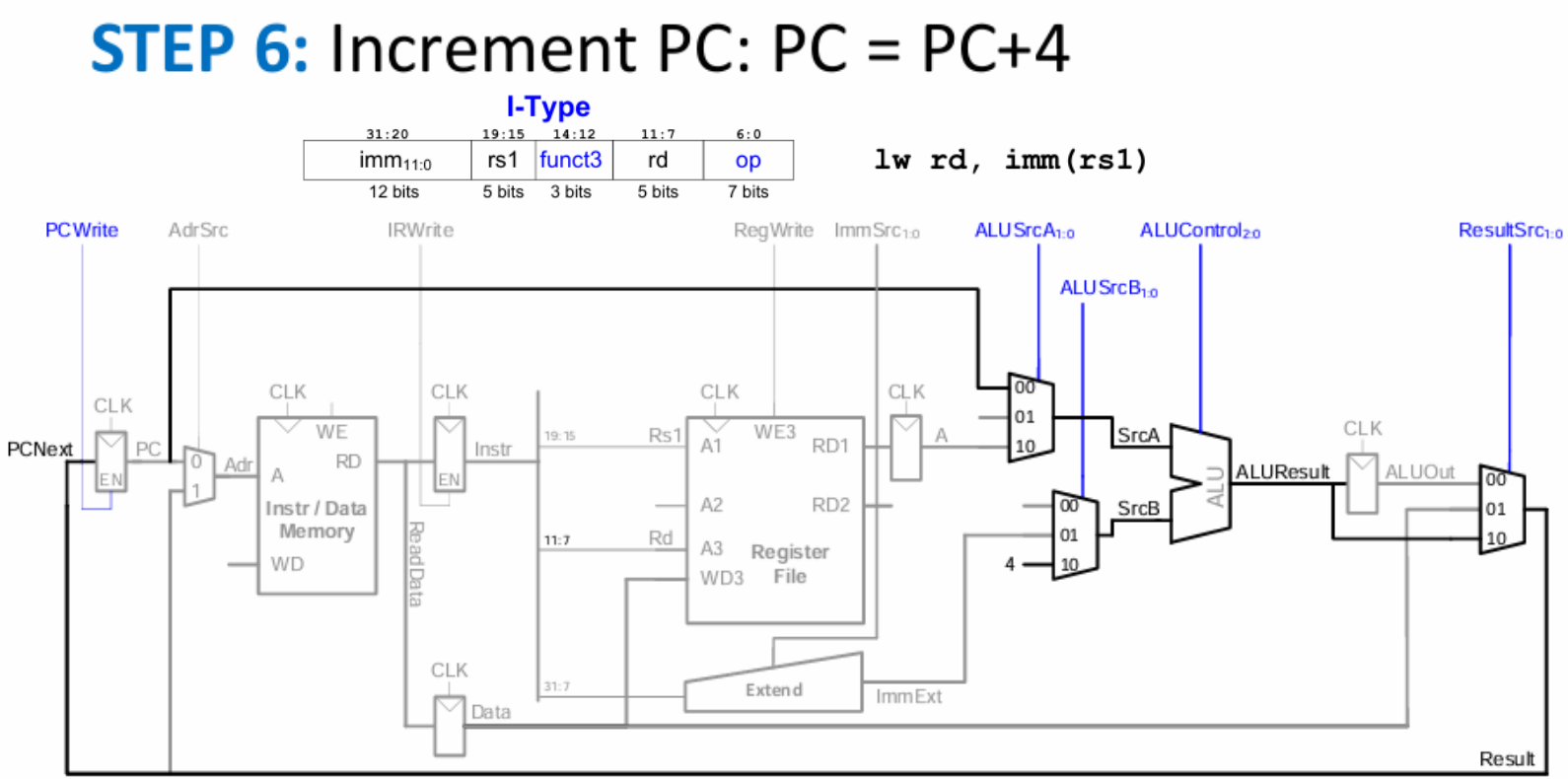


Multi-Cycle Datapath: Instruction Fetch

- Step 5: The data is written back to the register file
  - The `RegWrite` signal is 1 to indicate register should be updated
  - While all this is happening, the processor must update the program counter by adding 4 to the old PC

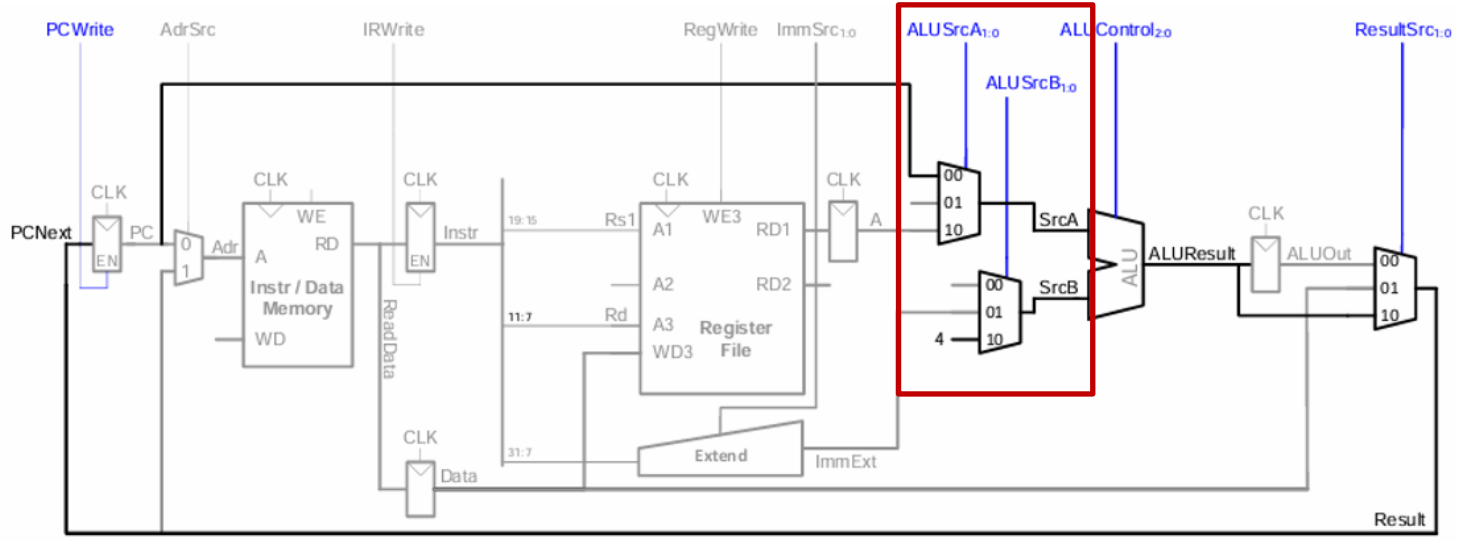


Multi-Cycle Datapath: `lw`



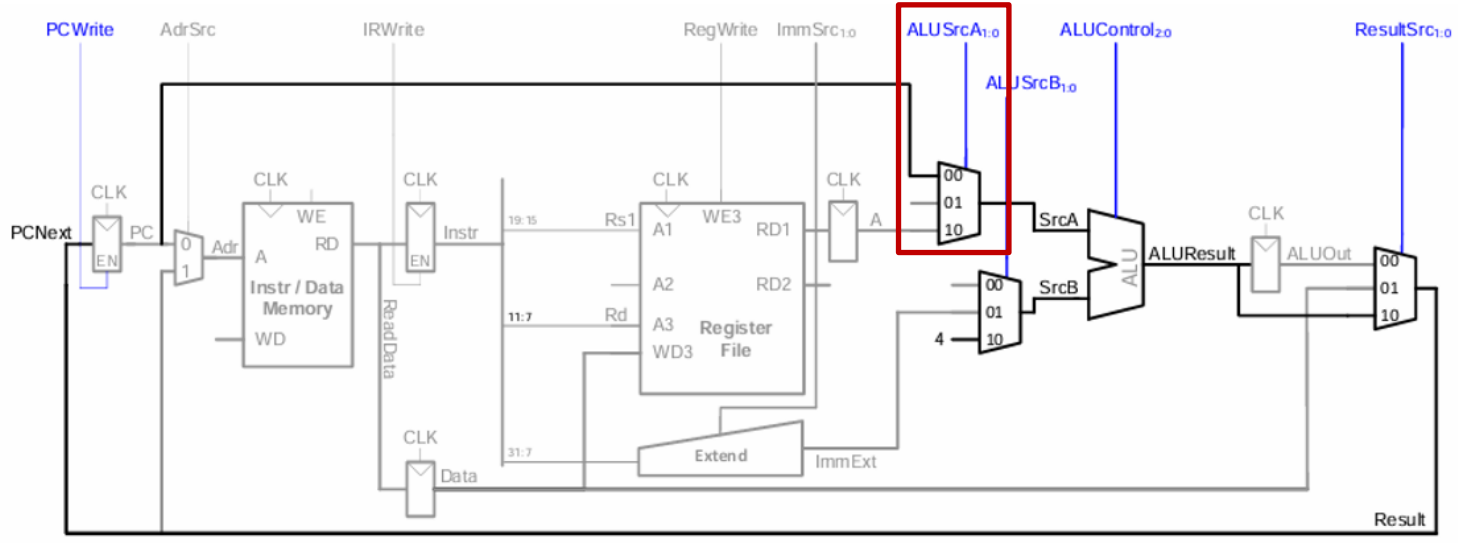
Multi-Cycle Datapath: Instruction Fetch

- Step 6: Increment PC: PC +=4
  - We can use existing ALU during fetch step because it is not busy
  - Add source MUX to choose PC and the constant 4 as ALU inputs



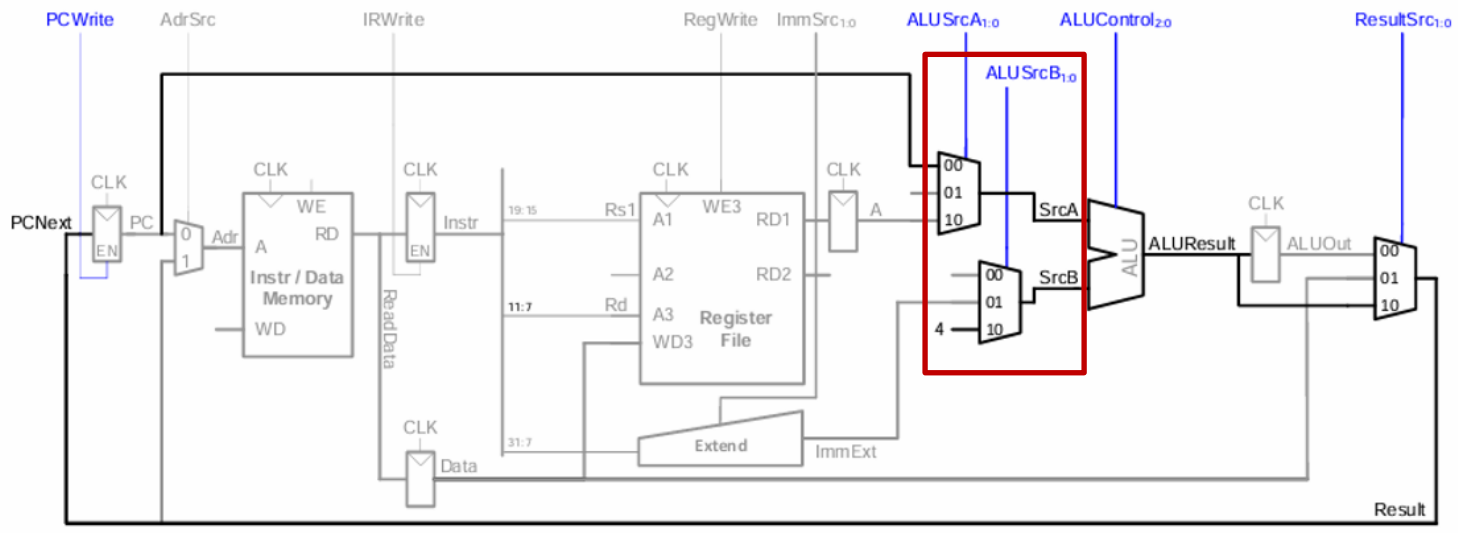
Multi-Cycle Datapath: Instruction Fetch

- Step 6: Increment PC: PC +=4
  - A MUX controlled by `ALUSrcA` chooses either `PC` or `register A` as `SrcA`



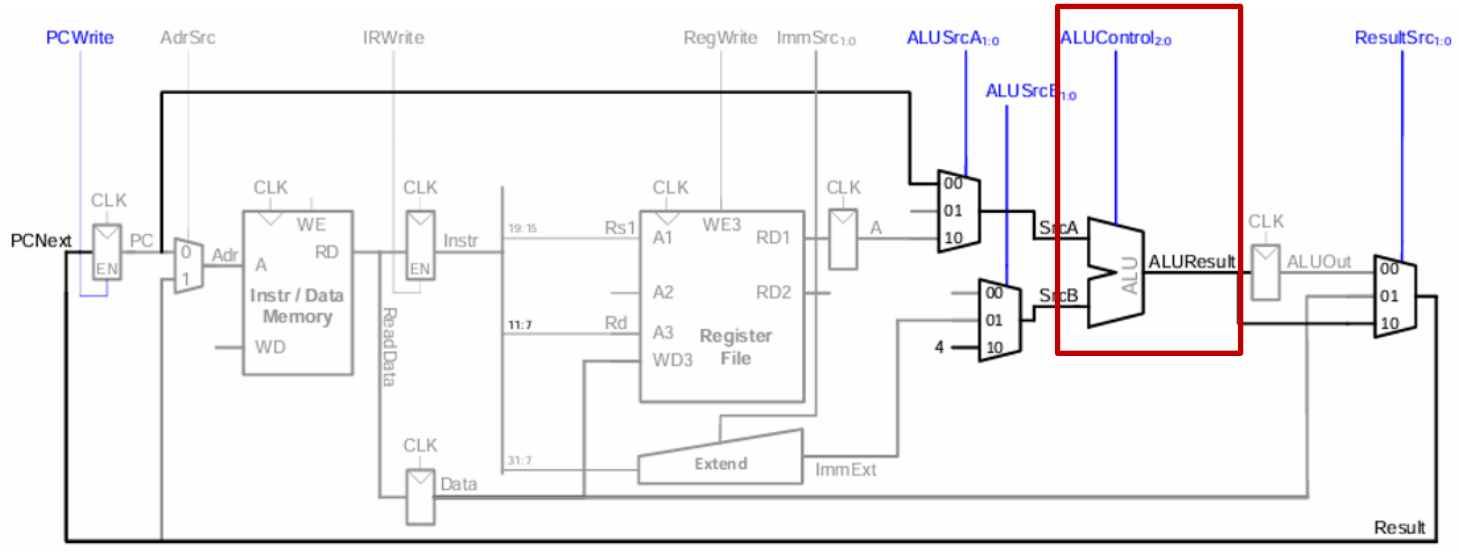
Multi-Cycle Datapath: Instruction Fetch

- Step 6: Increment PC: PC +=4
  - Another MUX controlled by `ALUSrcB` chooses either `4` or `ImmExt` as `SrcB`



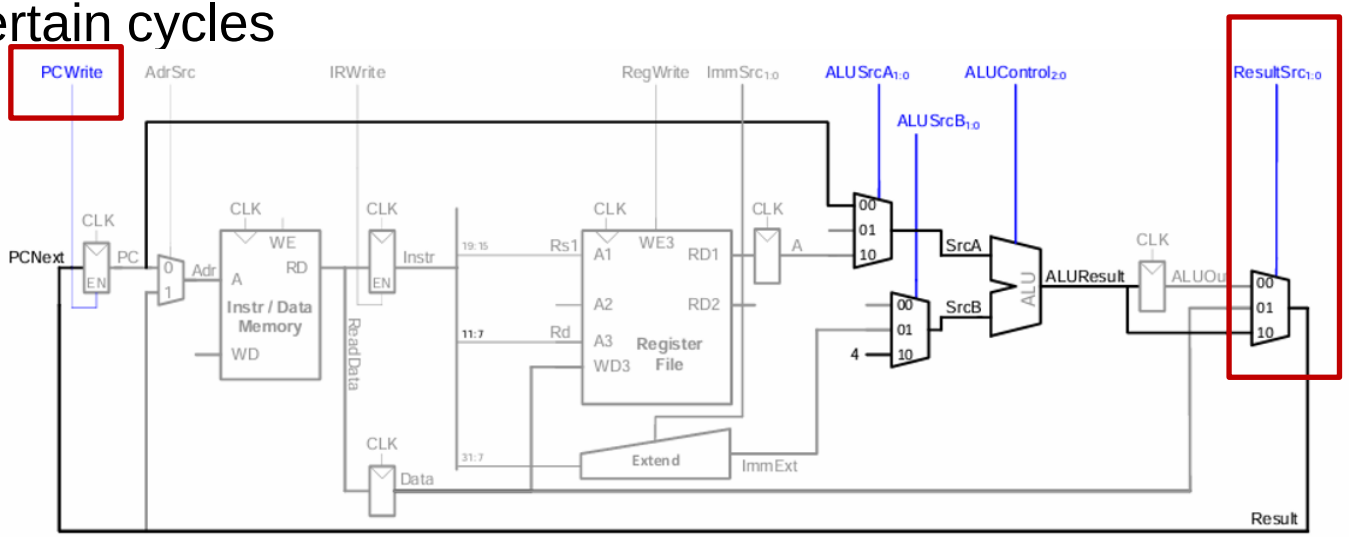
Multi-Cycle Datapath: Instruction Fetch

- Step 6: Increment PC: PC +=4
  - To update the `PC`, the ALU adds `SrcA (PC)` to `SrcB (4)`
  - The result is written into the program counter



Multi-Cycle Datapath: Instruction Fetch

- Step 6: Increment PC: PC +=4
  - `ResultSrc` MUX chooses the sum from `ALUResult` not `ALUOut`
  - The `PCWrite` control signal enables the PC to be written only on certain cycles



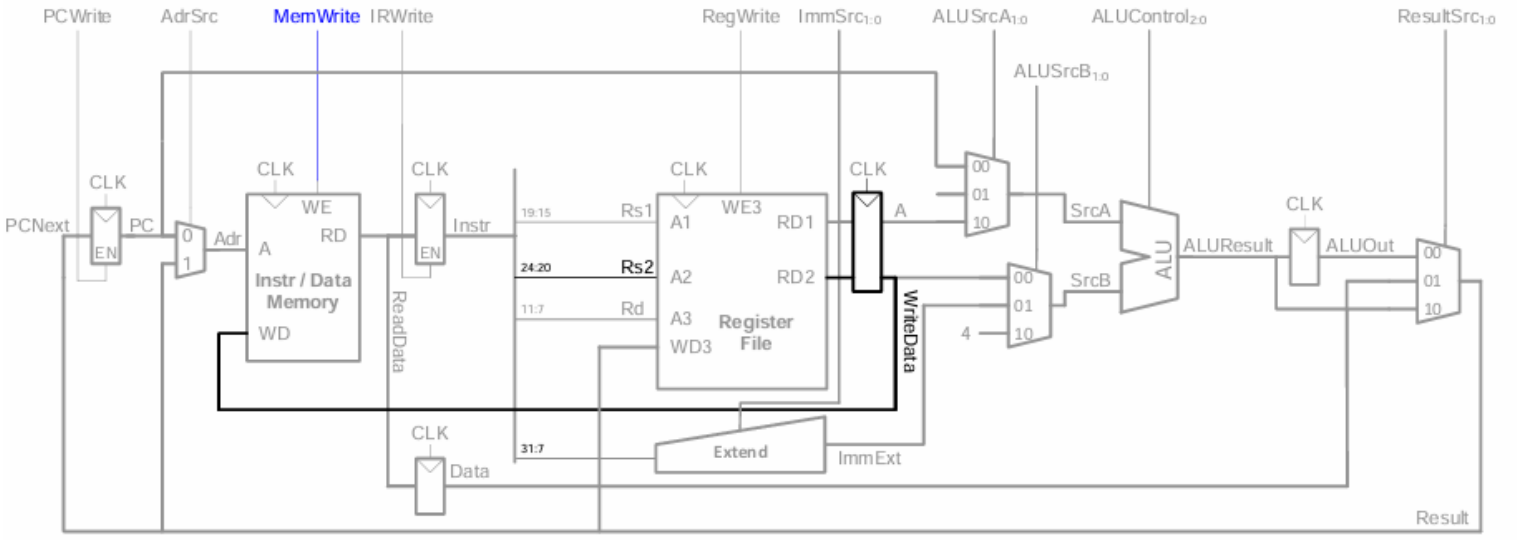
Multi-Cycle Datapath: `sw`

Write data in **rs2** to memory

S-Type

31:25	24:20	19:15	14:12	11:7	6:0
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

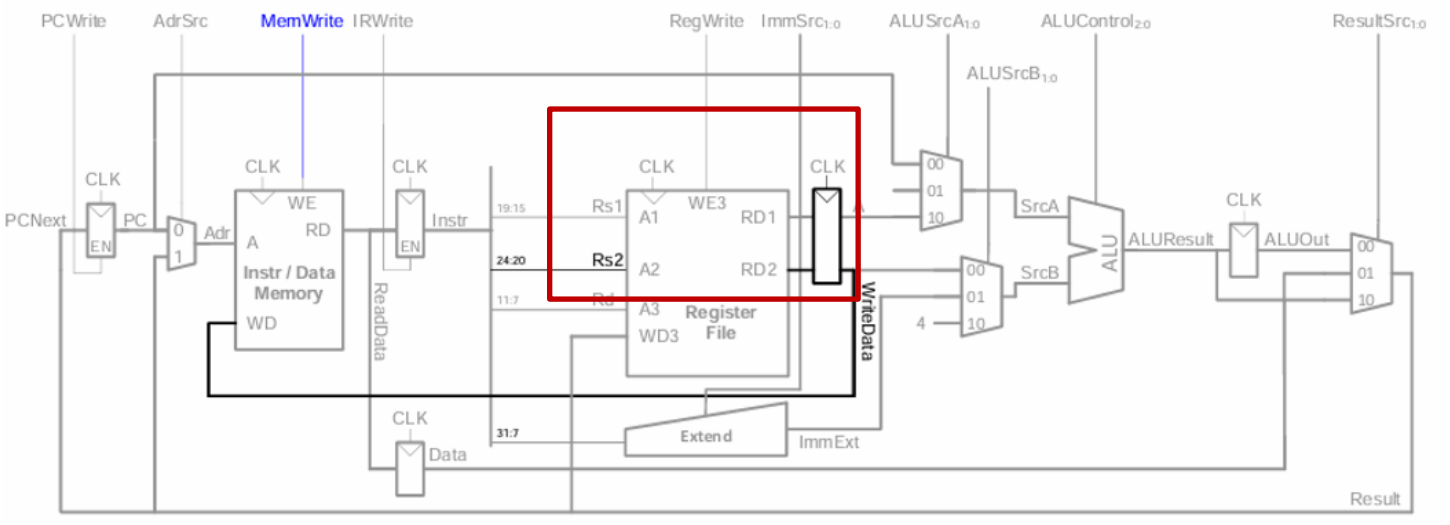
```
sw rs2, imm(rs1)
```



Multi-Cycle Datapath: Instruction Fetch (`sw`)

- Step 1: Read a base address
  - Read a base address from port 1 (`RD1`) of the register file and extends the immediate

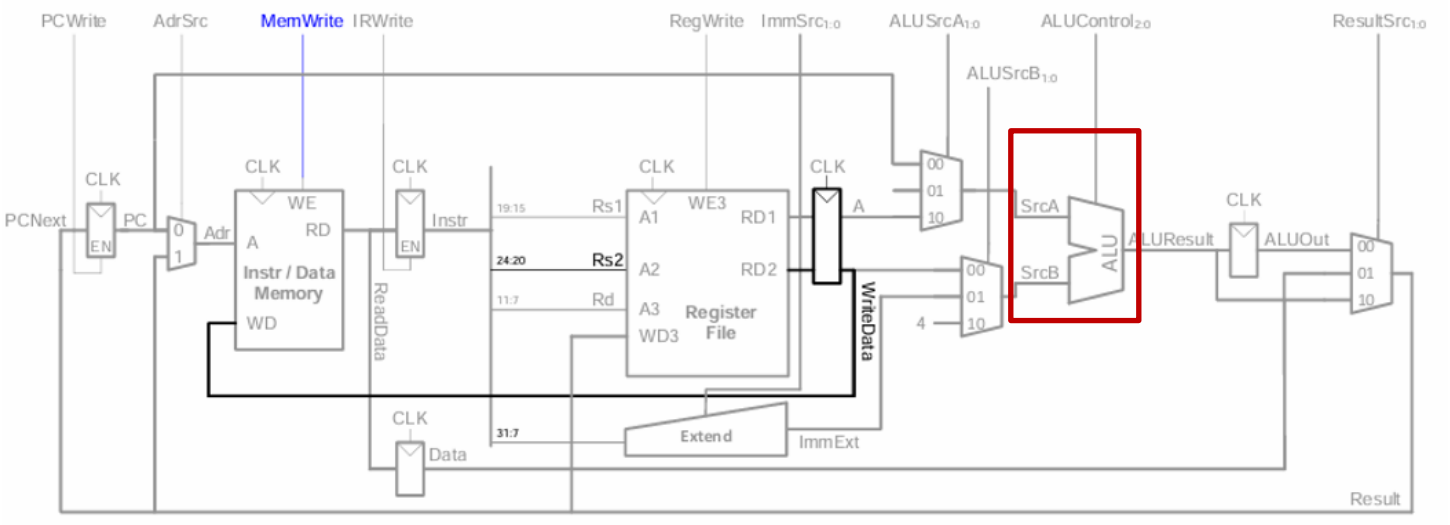
`sw rs2, imm(rs1)`



Multi-Cycle Datapath: Instruction Fetch (`sw`)

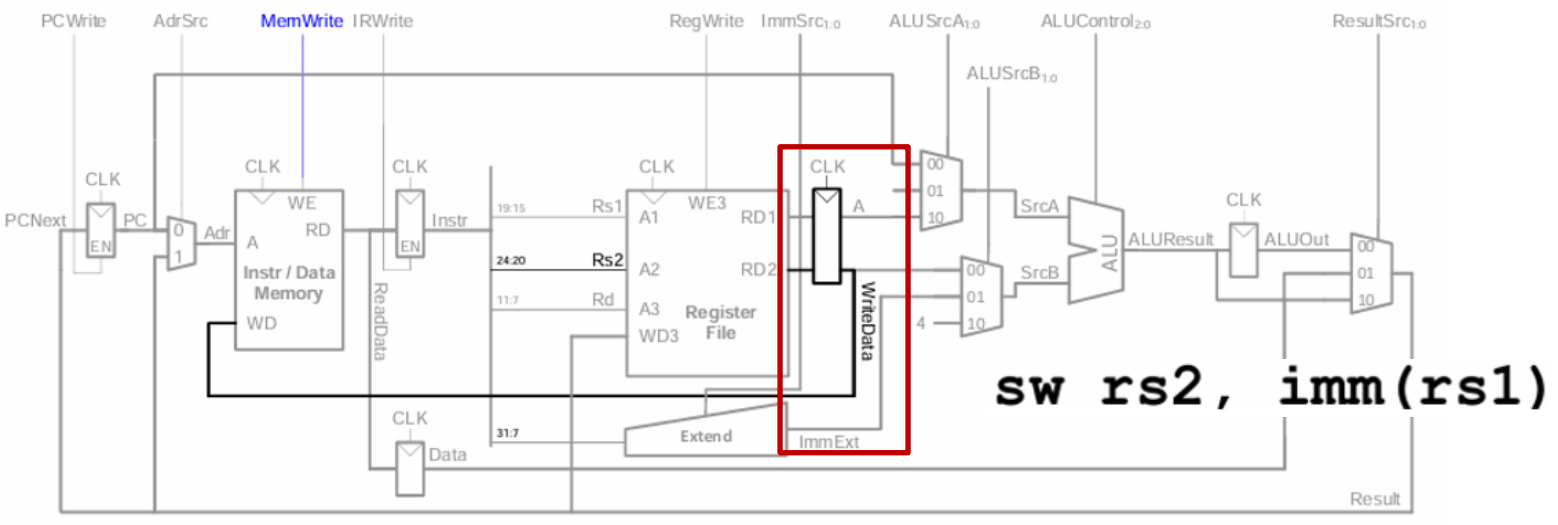
- Step 1: Read a base address
  - The ALU adds the base address to the immediate to find the memory address

```
sw rs2, imm(rs1)
```



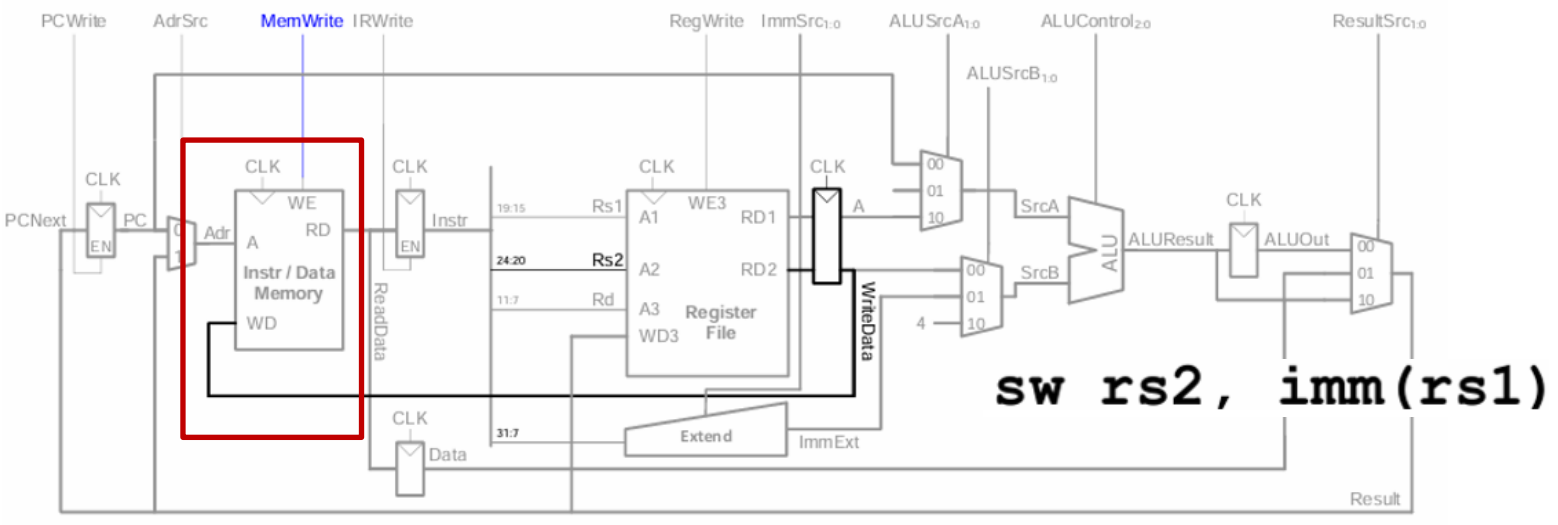
Multi-Cycle Datapath: Instruction Fetch (`sw`)

- Step 2: Read `RS2` from register file and write it into memory
  - The `Rd` is connected to the second port of the register file
  - When the register is read, it is stored in a register (`WriteData`)



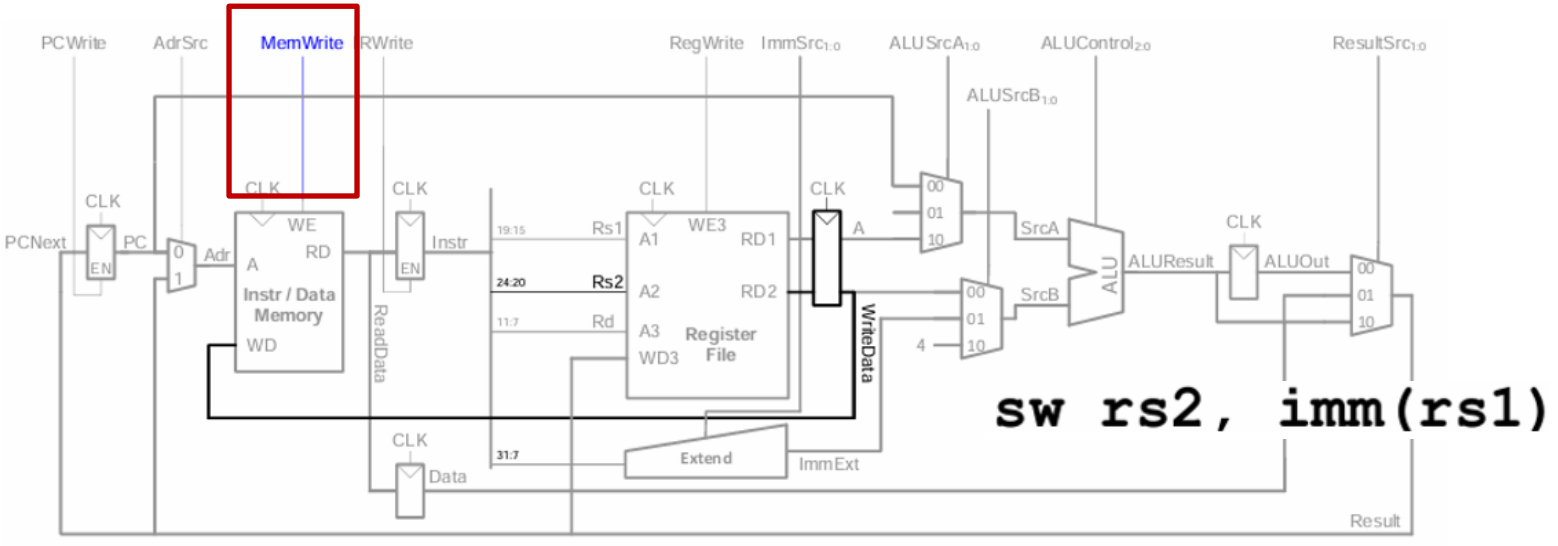
Multi-Cycle Datapath: Instruction Fetch (`sw`)

- Step 2: Read `RS2` from register file and write it into memory
  - The data is send to the write data port (`WD`) of the data memory to be written

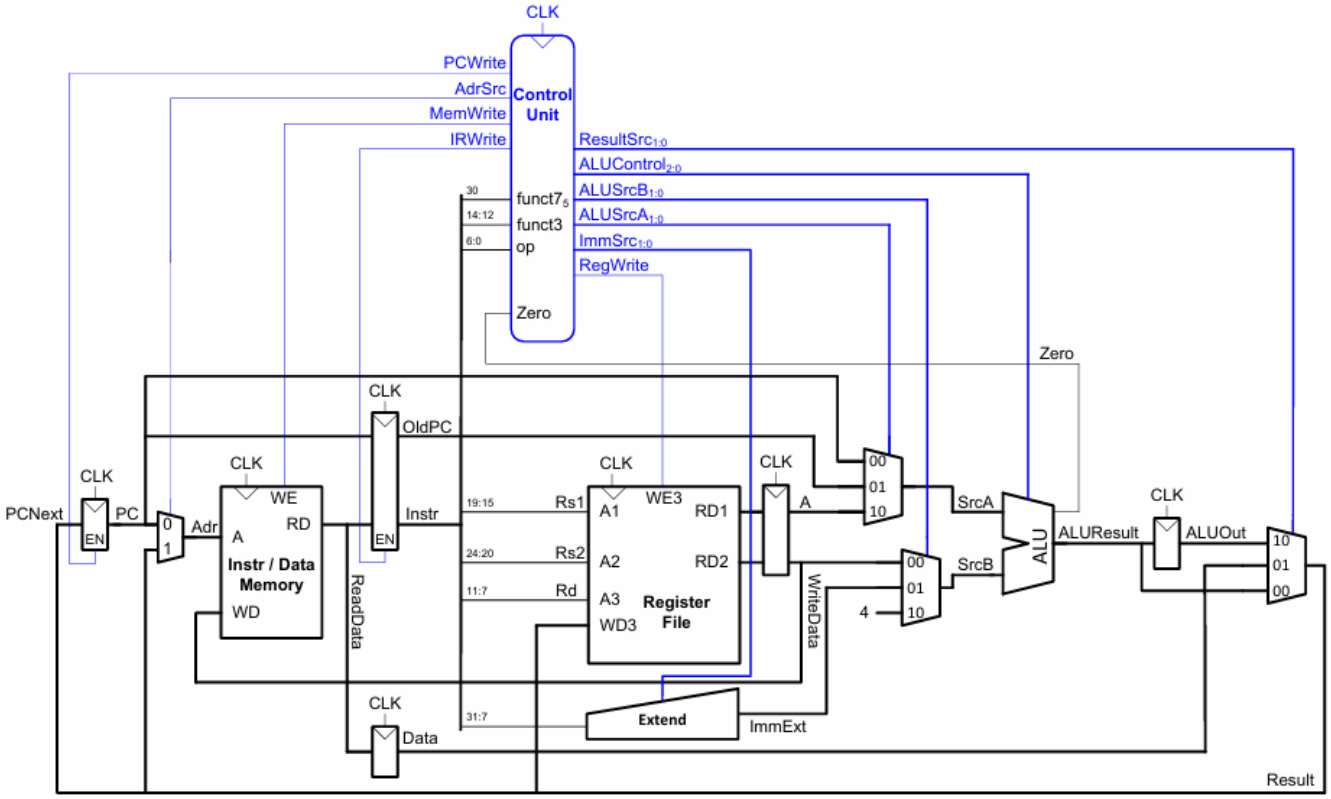


Multi-Cycle Datapath: Instruction Fetch (`sw`)

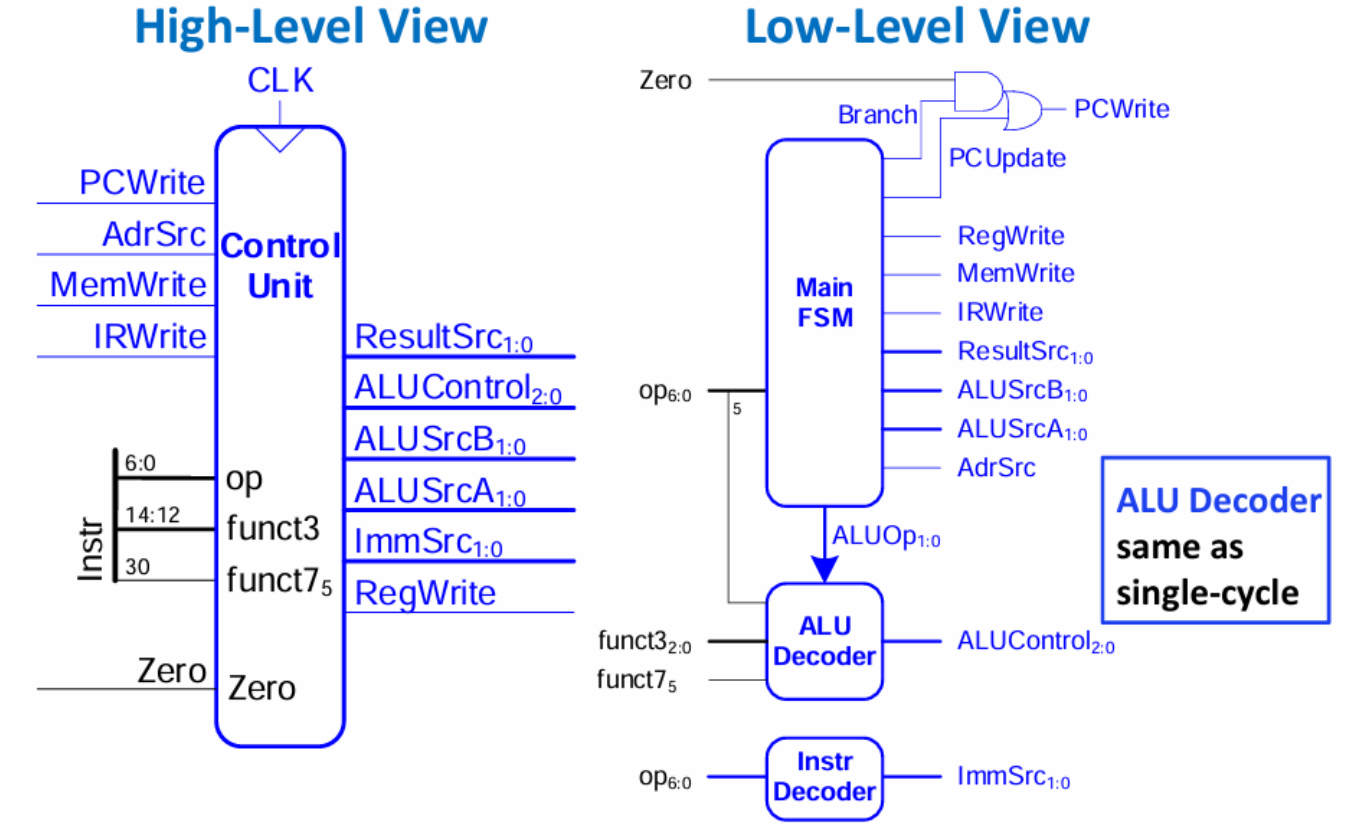
- Step 2: Read `RS2` from register file and write it into memory
  - The memory receives the `MemWrite` control signal to indicate that the write should occur



Multi-Cycle RISC-V Processor

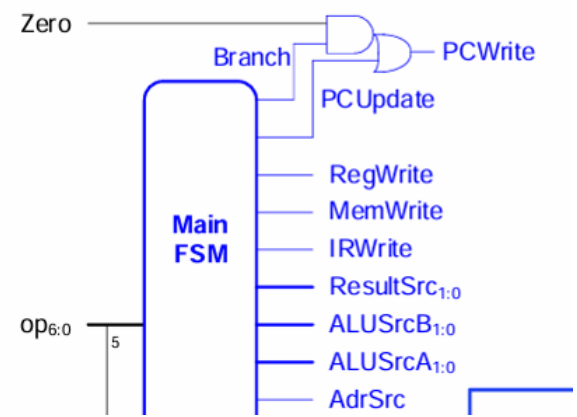


Multi-Cycle Control



Multi-Cycle Control

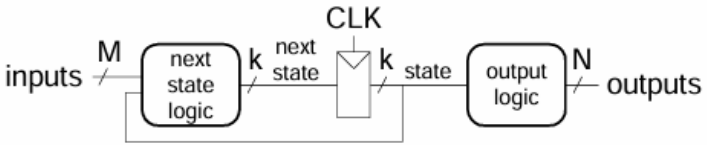
- The multicycle controller
  - The controller produces a sequence of control signals
  - The combinational main decoder of the single-cycle processor is replaced with a Main FSM in the multicycle processor
  - The Main FSM is a Moore machine
    - The outputs are only a function
    - of the current state



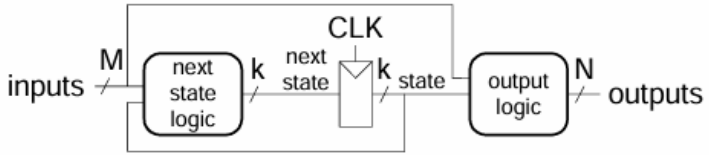
Finite State Machine (FSMs)

- **Next state** determined by **current state** and **inputs**
- Two types of finite state machines differ in **output** logic:
  - **Moore FSM: outputs** depend **only** on **current state**
  - **Mealy FSM: outputs** depend on **current state *and* inputs**

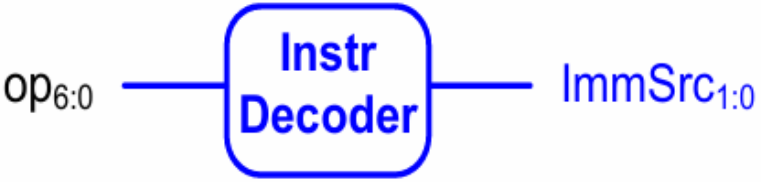
Moore FSM



Mealy FSM



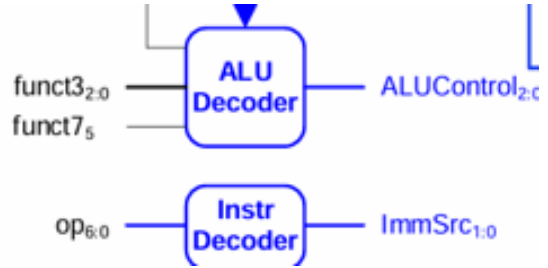
Multi-Cycle Control: Instruction Decoder



op	Instruction	ImmSrc
3	`lw`	00
35	`sw`	01
51	**R-type**	XX
99	`beq`	10

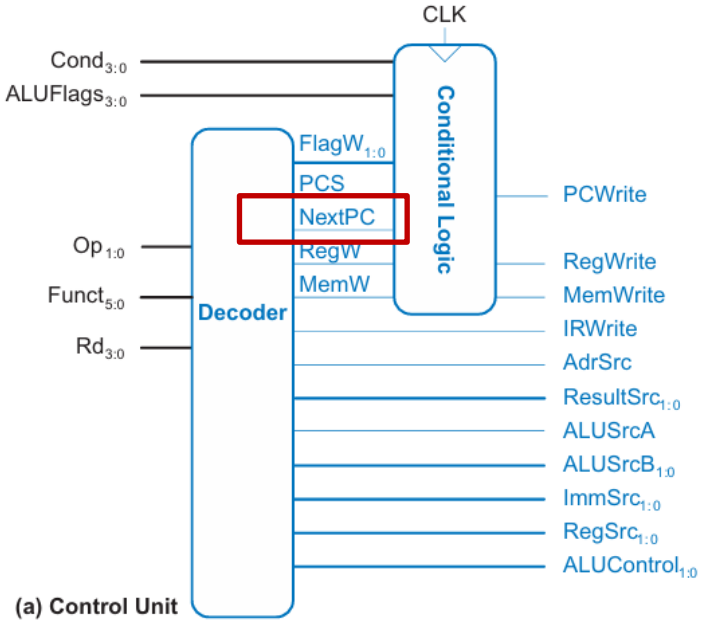
Multi-Cycle Control: Instruction Decoder

- The multicycle controller
  - The instruction decoder computes these signals
    - The `ImmSrc` is a function of `Op` rather the current state
    - The ALU Decoder and PC Logic are identical to those in the single-cycle processor
    - The Conditional Logic is almost identical to that of the single-cycle processor



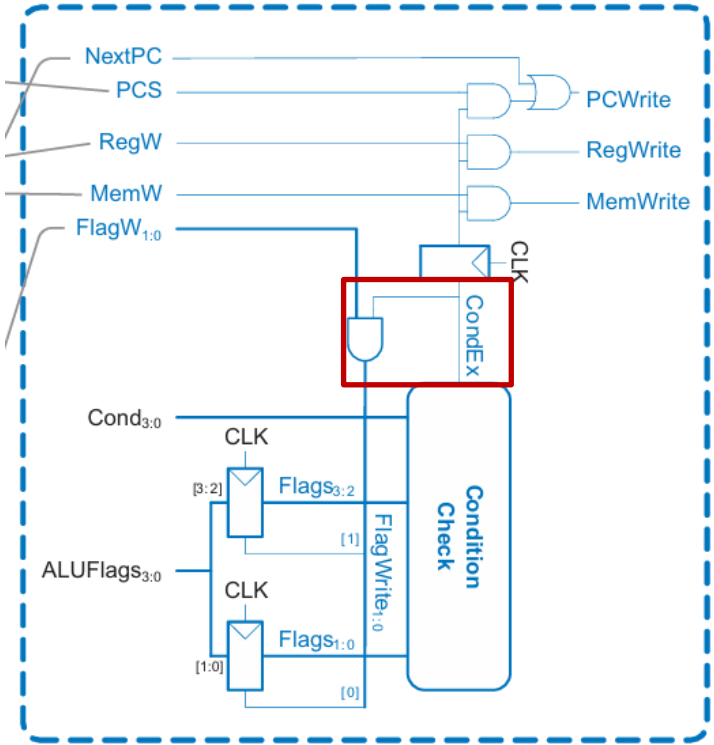
Multi-Cycle Control: Instruction Decoder

- The multicycle controller
  - The NextPC signal forces a write to the PC when we compute PC+4



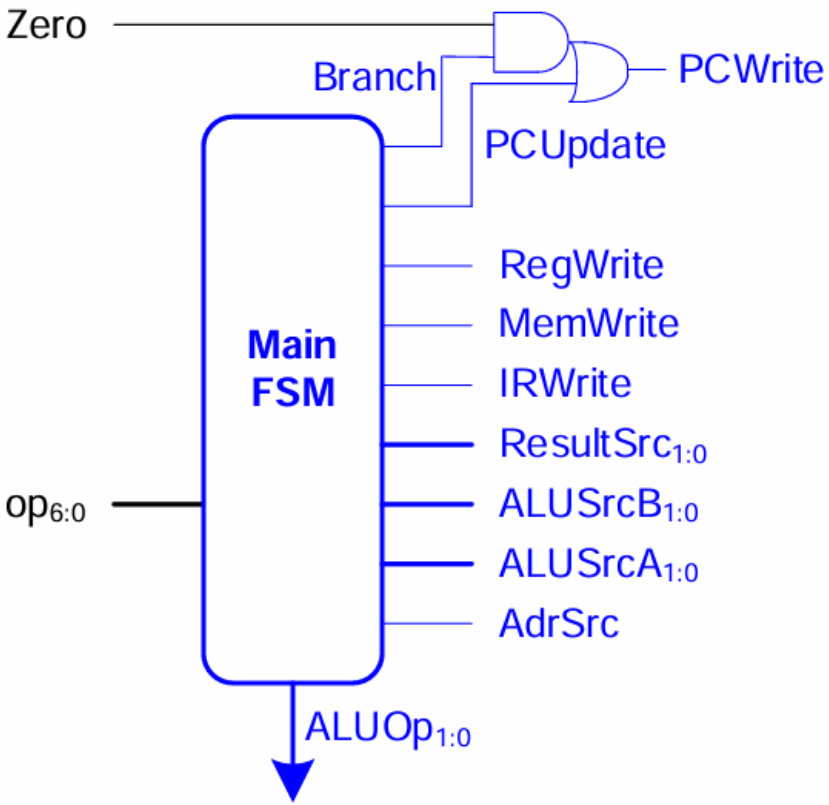
Multi-Cycle Control: Conditional Logic

- The multicycle controller
  - Delay `CondEx` by one cycle before sending it to `PCWrite`, `RegWrite` and `MemWrite`
  - Update conditional flags are not seen until the end of an instruction



(c) Conditional Logic

Multi-Cycle Control: Main FSM



To declutter FSM:

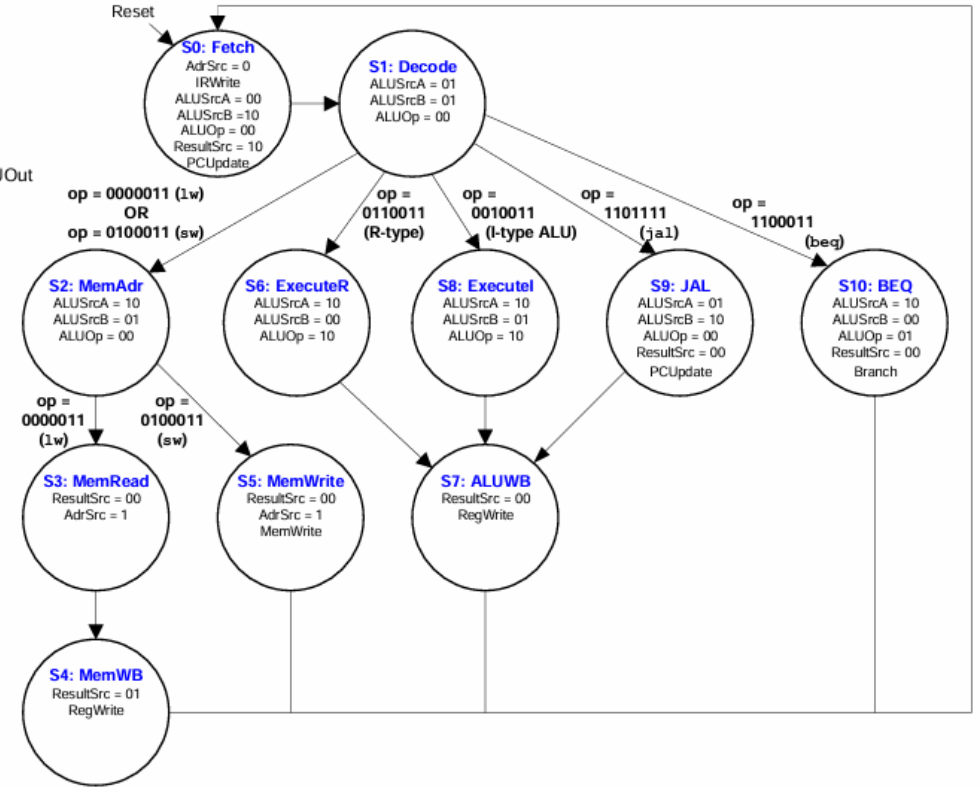
- **Write enable signals** (RegWrite, MemWrite, IRWrite, PCUpdate, and Branch) are **0** if not listed in a state.
- **Other signals are don't care** if not listed in a state

Multi-Cycle Control: Main FSM

- The Main FSM
  - Produces multiplexer select, register enable, and memory write enable signals for the datapath
  - Enable signals (`RegW, MemW, IRWrite, and NextPC`) are list only when they are asserted; otherwise, they are 0

Multicycle Processor Main FSM

State	Datapath μOp
Fetch	Instr ←Mem[PC]; PC ← PC+4
Decode	ALUOut ← PCTarget
MemAdr	ALUOut ← rs1 + imm
MemRead	Data ← Mem[ALUOut]
MemWB	rd ← Data
MemWrite	Mem[ALUOut] ← rd
ExecuteR	ALUOut ← rs1 op rs2
ExecuteI	ALUOut ← rs1 op imm
ALUWB	rd ← ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC ← ALUOut
JAL	PC ← ALUOut; ALUOut ← PC+4



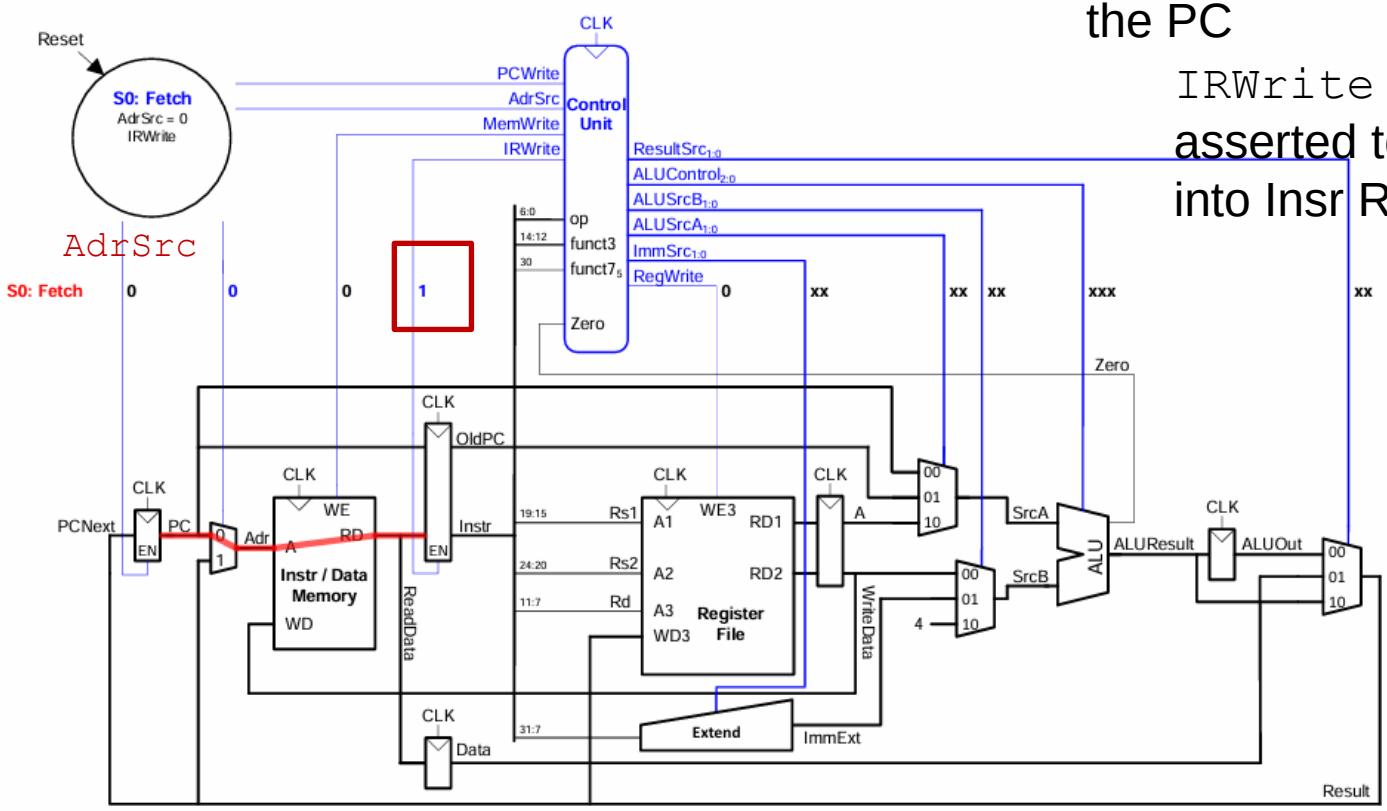
Main FSM: Fetch

- The first step for any instruction
  - Fetch the instruction from memory at the address held in the PC
  - To increment the PC to the next instruction
  - The FSM enters this Fetch state on reset

Main FSM: Fetch

To read memory, `AdrSrc` = 0 -> the address is taken from the PC

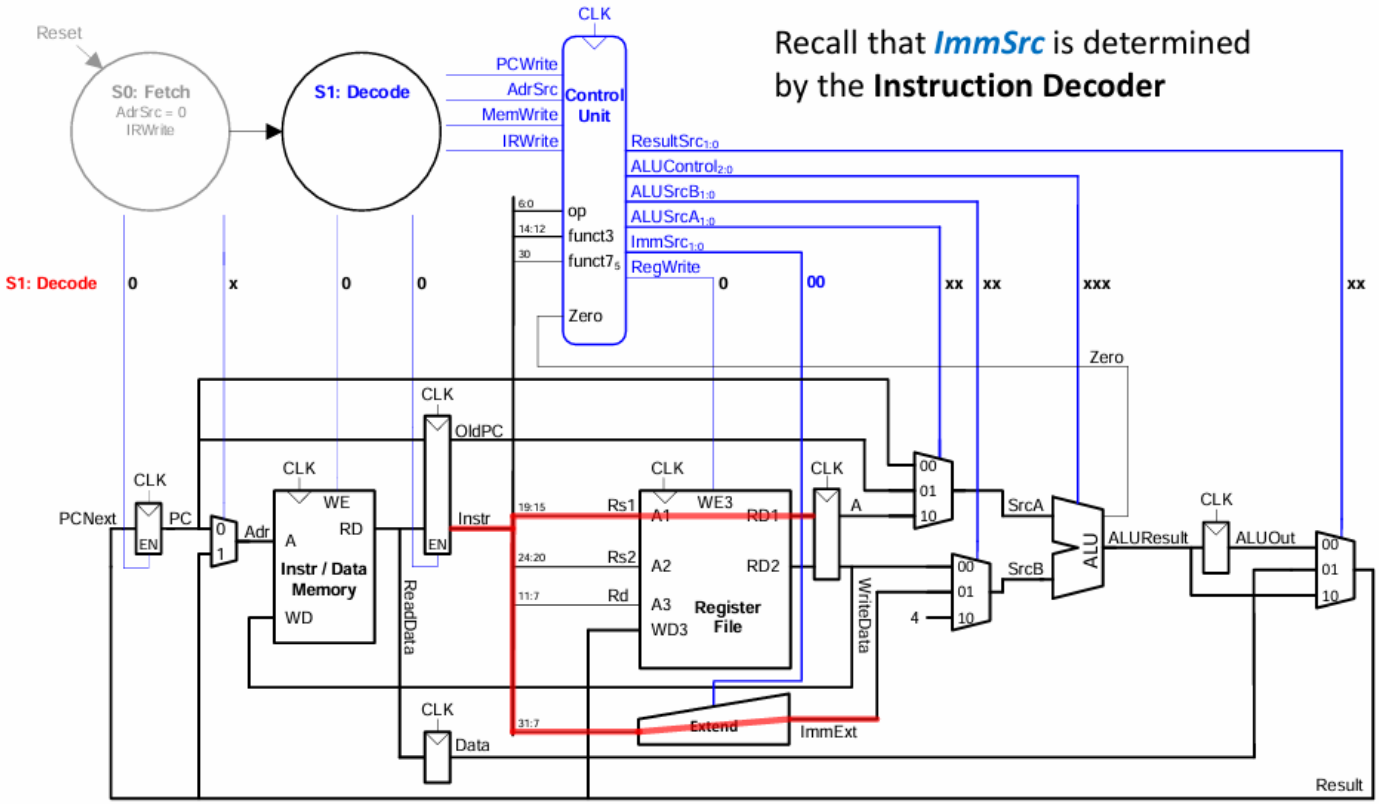
`IRWrite` is asserted to write Inst into Insr Reg



Main FSM: Fetch

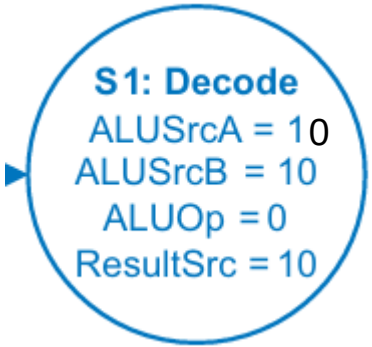
- The first step for any instruction
  - `ALUSrcA` = 10, so `SrcA` comes from the PC
  - `ALUSrcB` = 10, so `SrcB` is the constant 4
  - `ALUOp` = 0, so the ALU produces `ALUControl` = 00 to make the ALU add
  - To update the PC with PC+4, `ResultSrc` = 1 to choose the `ALUResult`
  - `NextPC = 1` to enable `PCWrite`

Main FSM: Decode



Main FSM: Decode

- The second step is to decode the instruction
  - Read the register file and/or immediate
  - The registers and immediate are select based on `RegSrc` and ImmSrc, which are computed by the Instr Decoder based on Inst

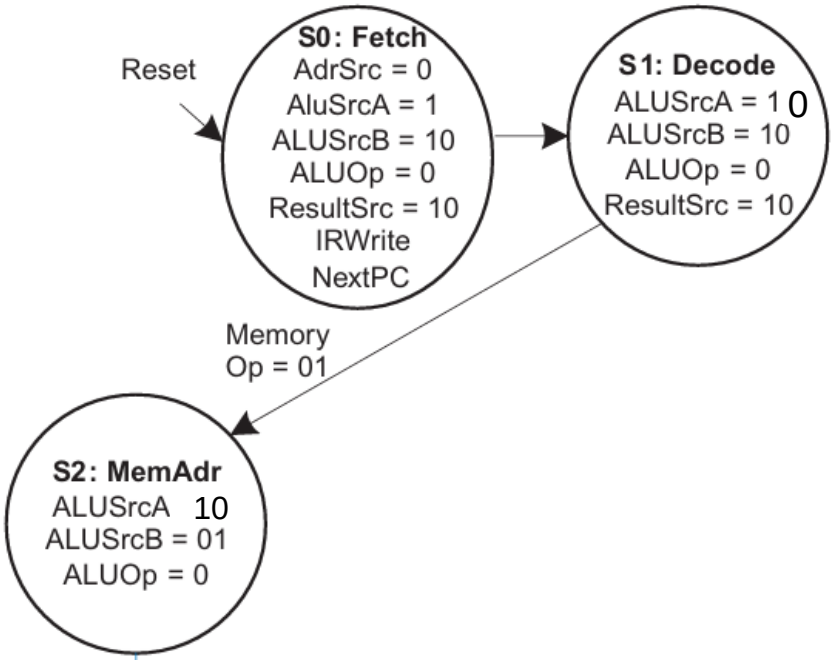


Main FSM: Decode

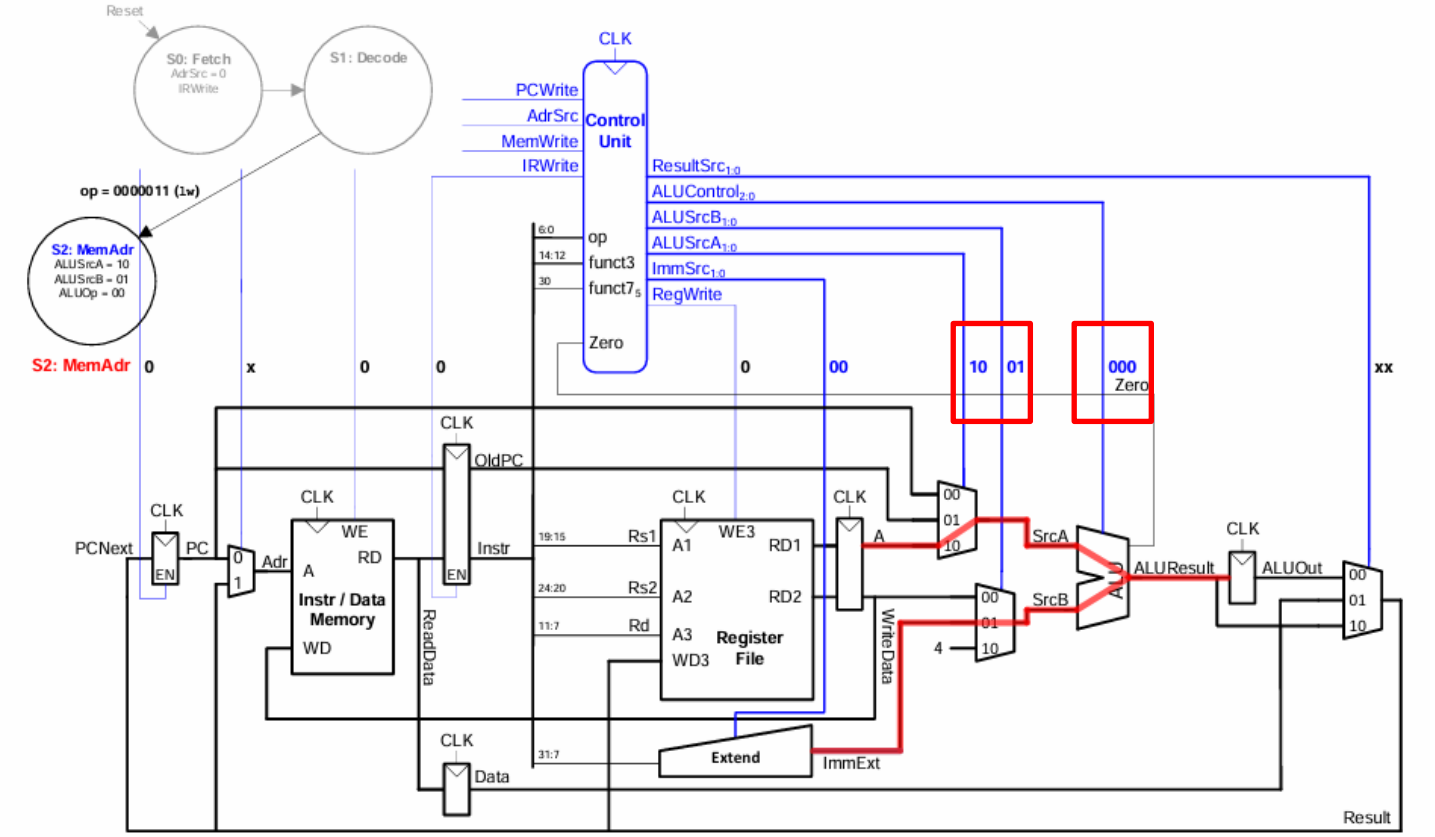
- The FSM proceeds to one of several possible states
  - Depending on Op and Funct that are examined during the Decode
  - If the instruction is a memory load or store
    - Compute the address by adding these base address to the zero-extended offset
    - `ALUSrcA` = 10 to select the base address from the register file
    - `ALUSrcB` = 01 to select ImmExt
    - `ALUOp` = 0 so the ALU adds
    - The effective address is stored in the `ALUOut` register for use on the next step

Main FSM: MemAdr

- The FSM for memory read computation
  - If the instruction is a memory load or store
  - S2: computes the memory address

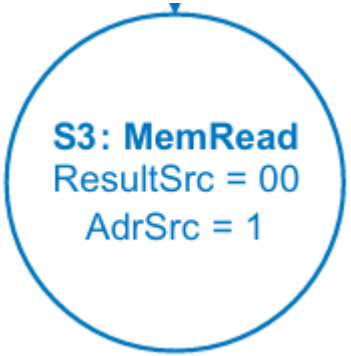


Main FSM: Address

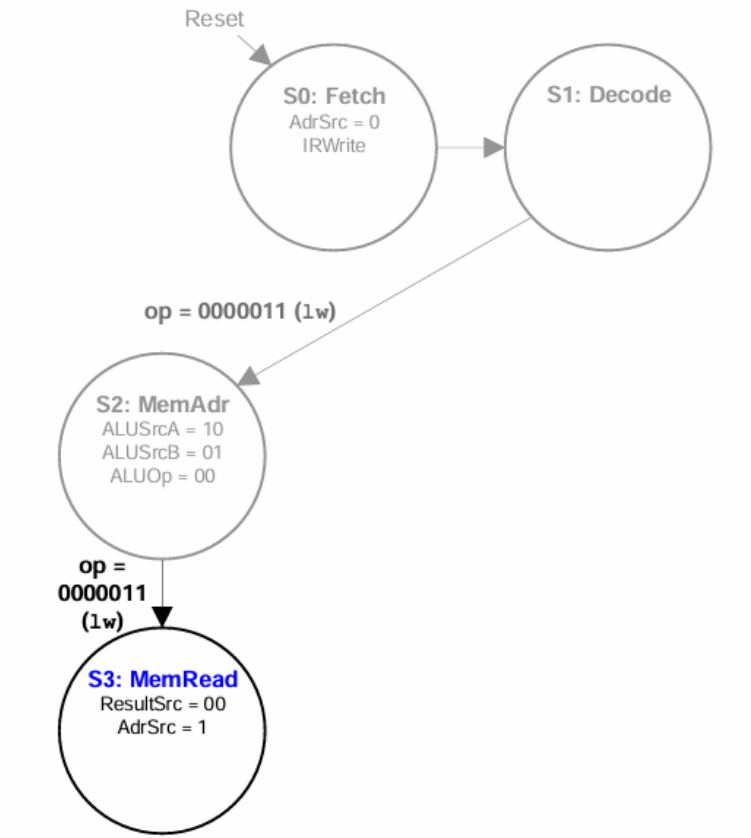


Main FSM: MemRead

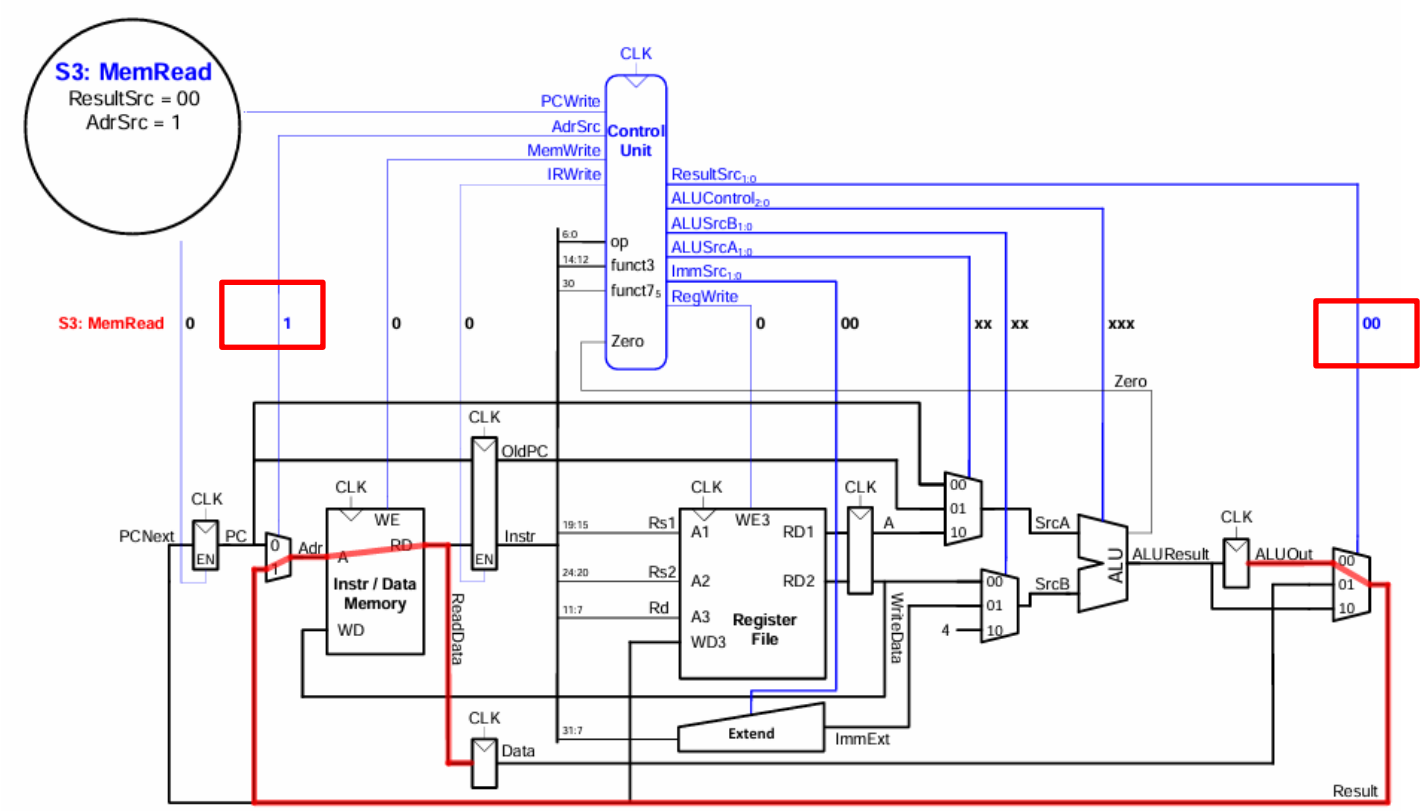
- Read data from the memory and write it to the Reg
  - To read from the memory
    - `ResultSrc` = 00, `AdrSrc` = 1 to select the memory address that was just computed and saved in `ALUOut`
    - The address in memory is read and saved in the Data register during the MemRead step



Main FSM: Read Memory



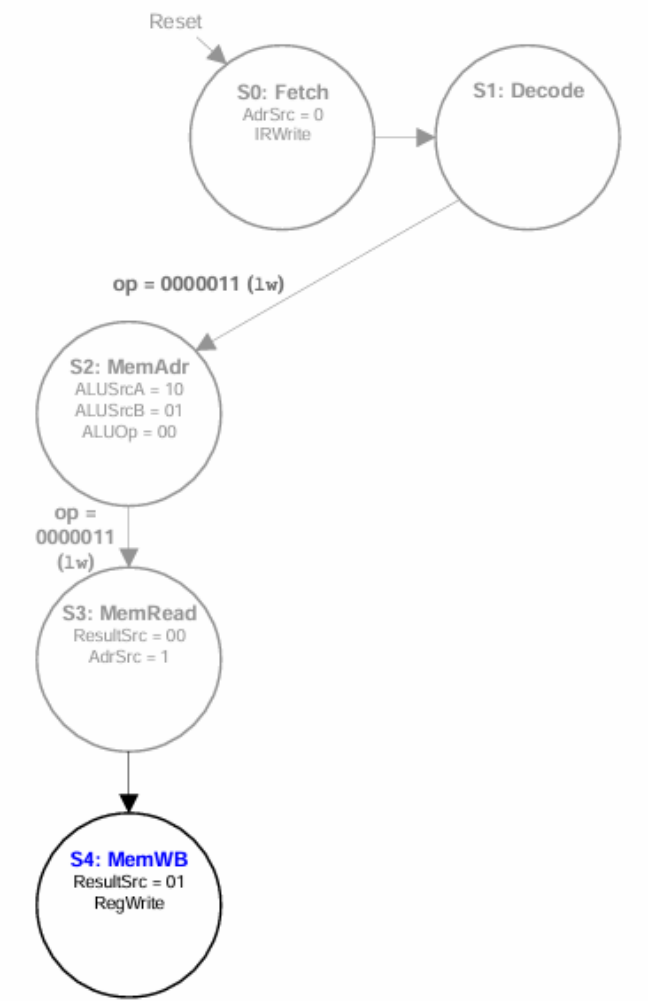
Main FSM: Read Memory Datapath



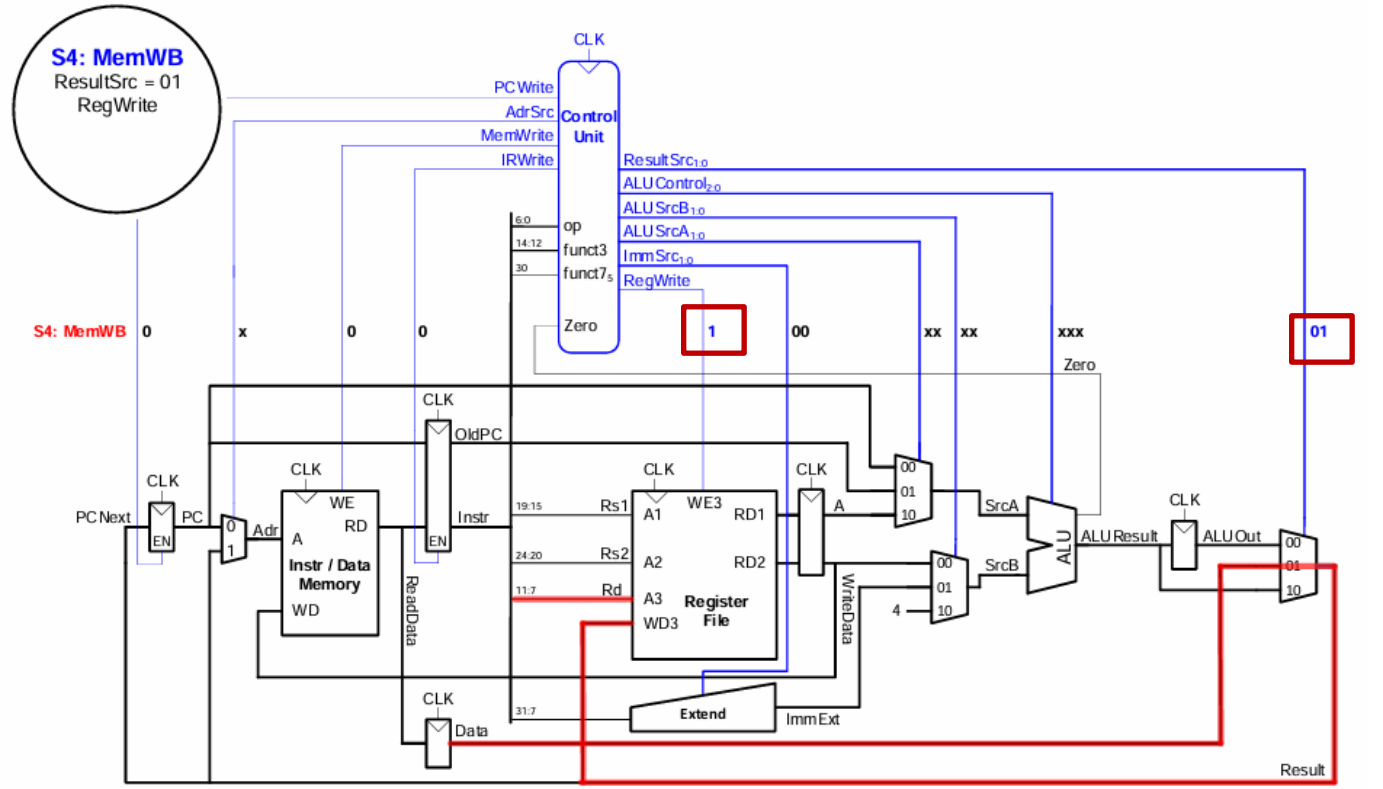
Main FSM: MemWB

- Data is written to the register file
  - `ResultSrc` = 01 to choose Result from Data and RegW is asserted to write the register file – completing `lw` instruction
  - Finally, the FSM returns to the Fetch state to start the next instruction

Main FSM: Write RF

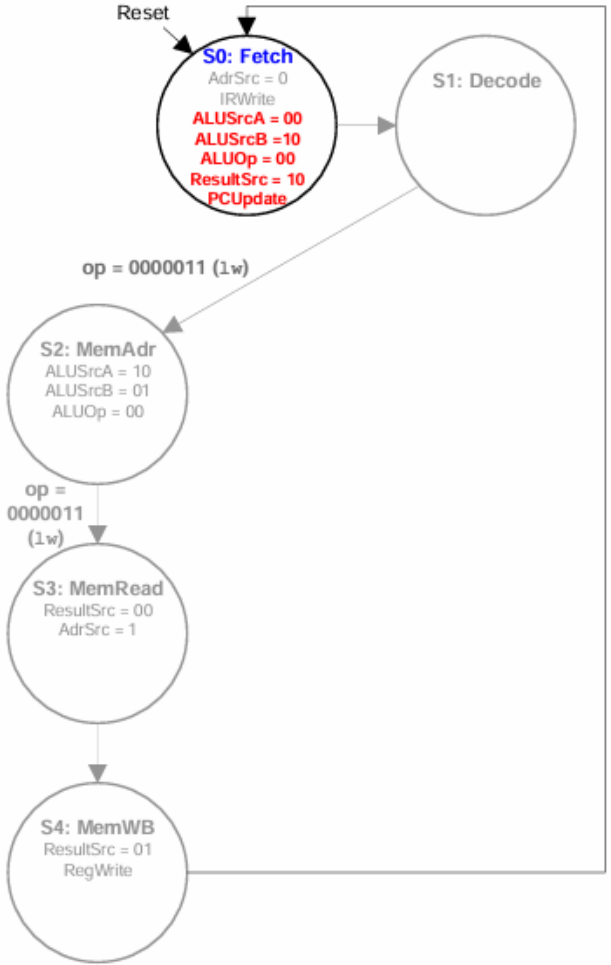


Main FSM: Write RF Datapath

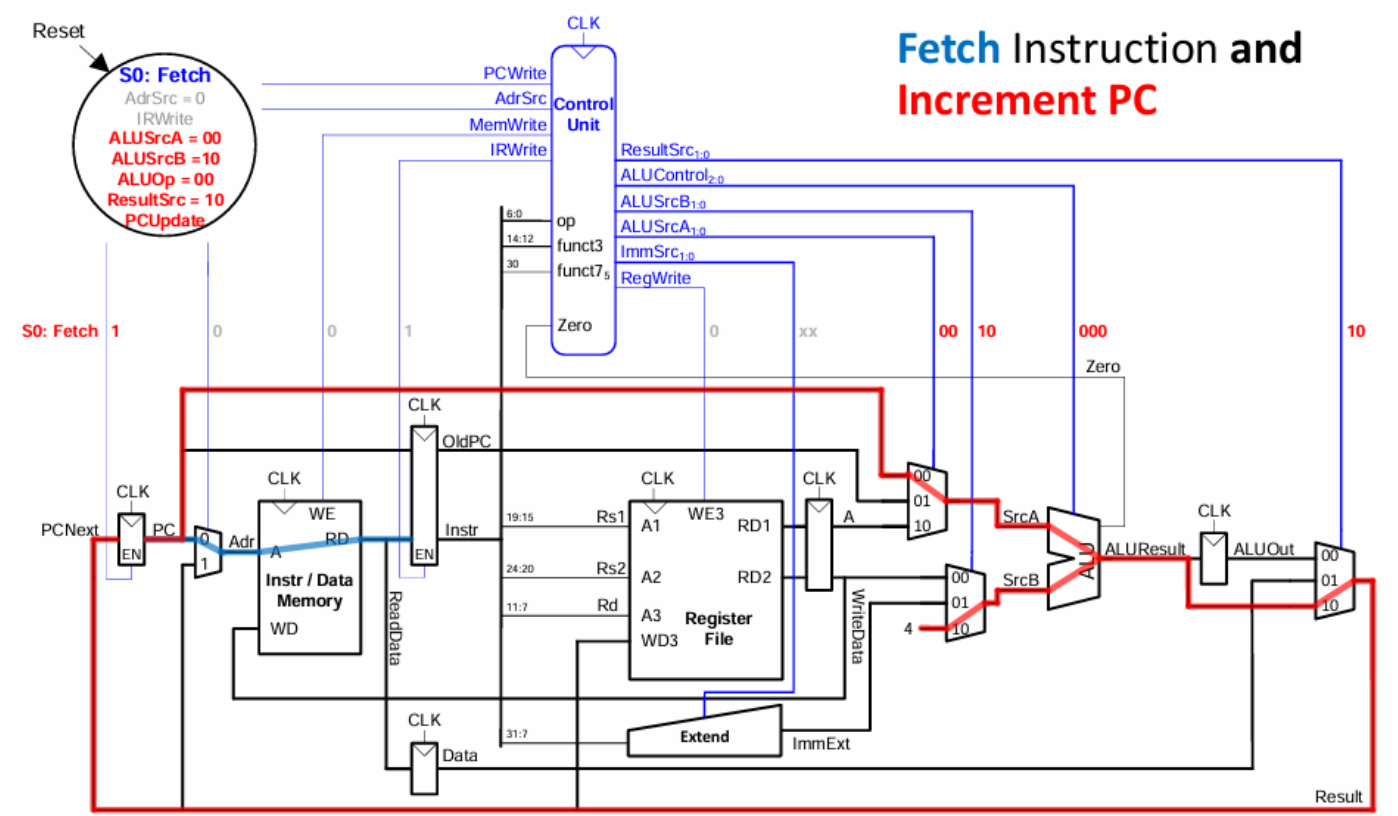


Main FSM: Fetch Revisited

Calculate **PC+4** during Fetch stage (ALU isn't being used)



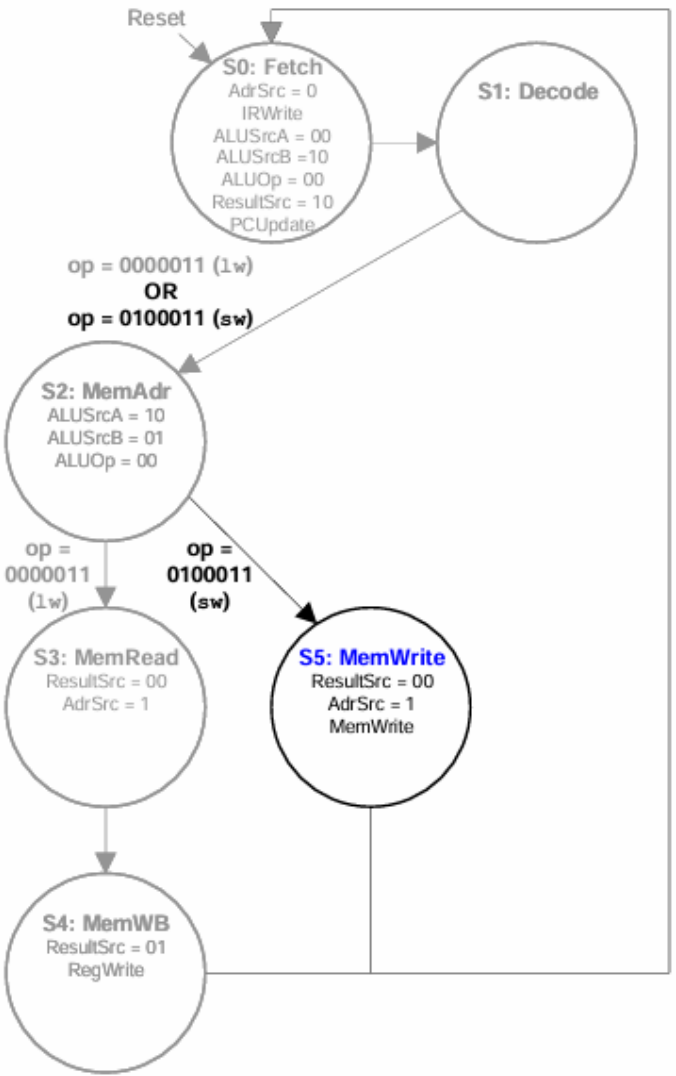
Main FSM: Fetch (PC+4) Datapath



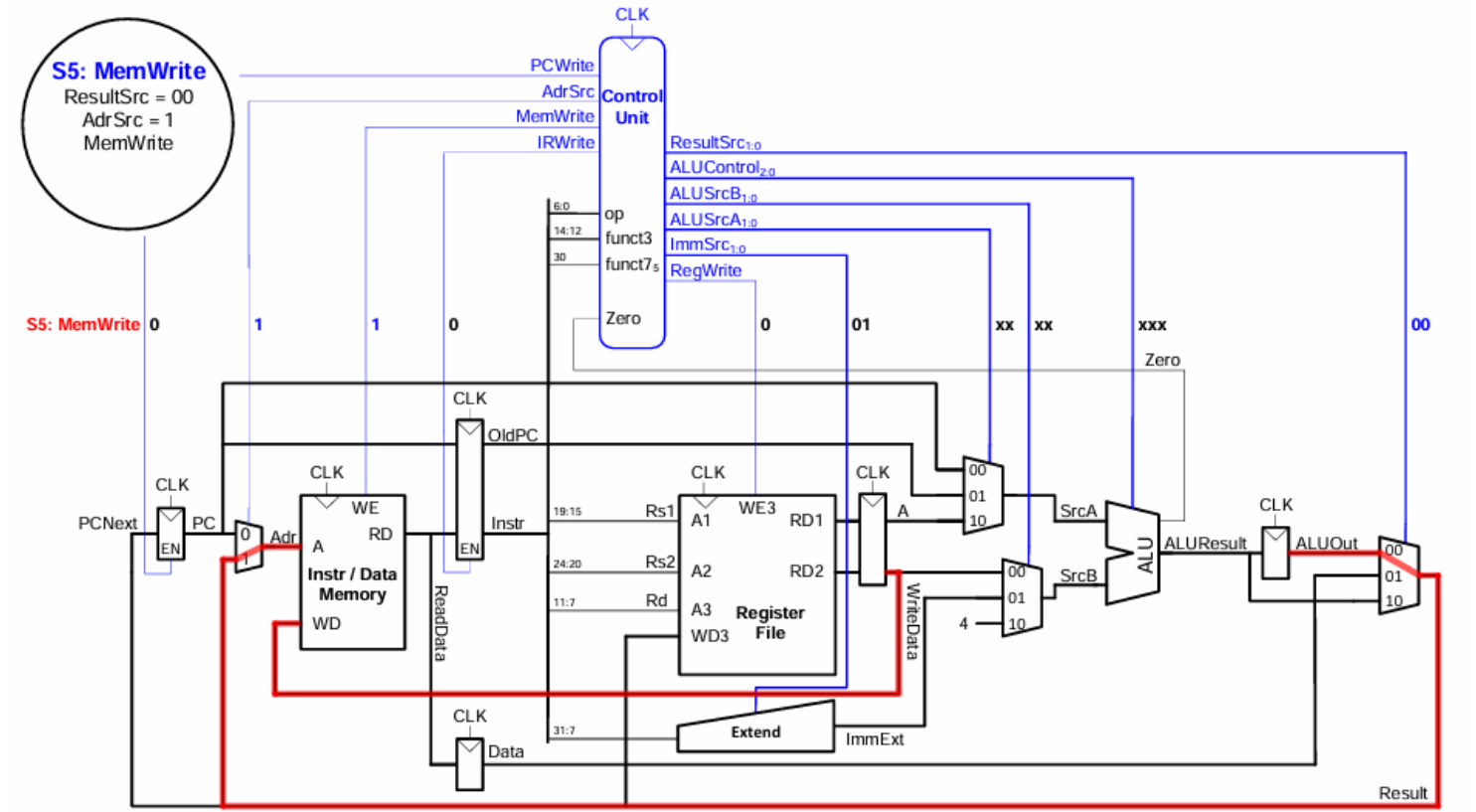
Main FSM: MemWrite

- In sw instruction
  - The data read from the second port of the register file is simply written to memory
  - In this MemWrite state
    - `ResultSrc` = 00 and `AdrSrc` = 1 to select the address computed in the MemSrc state and save in ALUOut
    - MemW is asserted to write the memory

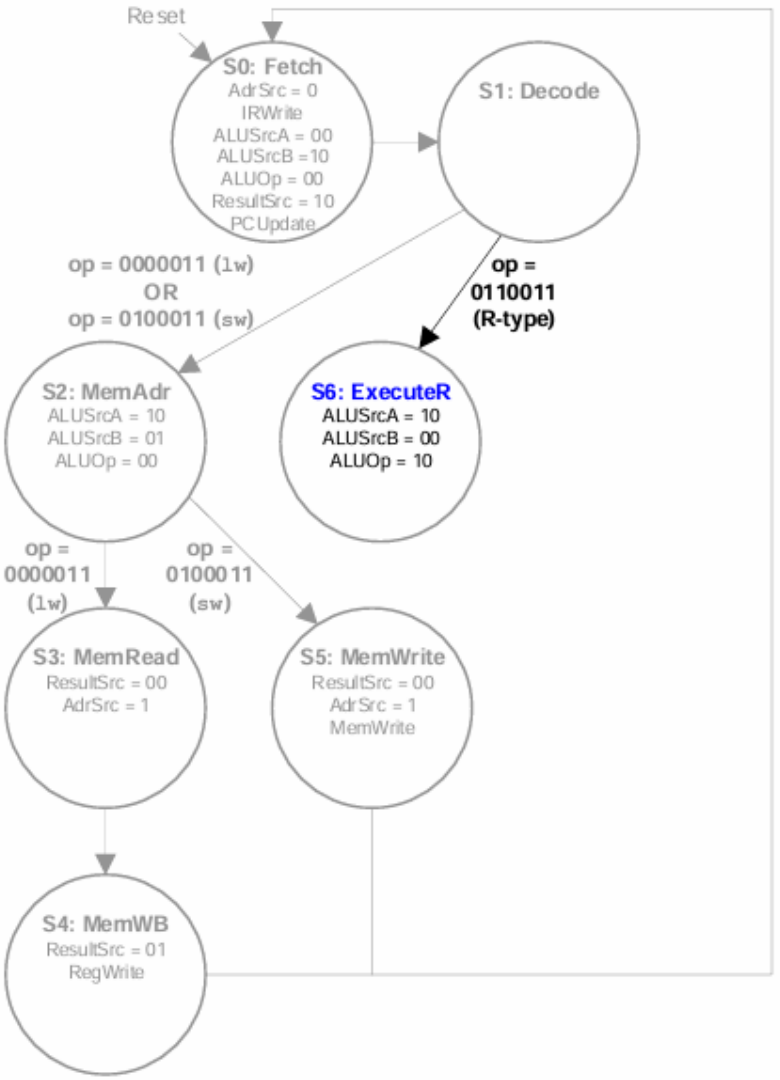
Main FSM: `sw`


Reset
S0: Fetch
AdrSrc = 0
IRWrite
ALUSrcA = 00
ALUSrcB =10
ALUOp = 00
ResultSrc = 10
PCUpdate
S1: Decode
op = 0000011 (lw)
OR
op = 0100011 (sw)
S2: MemAdr
ALUSrcA = 10
ALUSrcB = 01
ALUOp = 00
op = 0000011 (lw)
op = 0100011 (sw)
S3: MemRead
ResultSrc = 00
AdrSrc = 1
S5: MemWrite
ResultSrc = 00
AdrSrc = 1
MemWrite
S4: MemWB
ResultSrc = 01
RegWrite

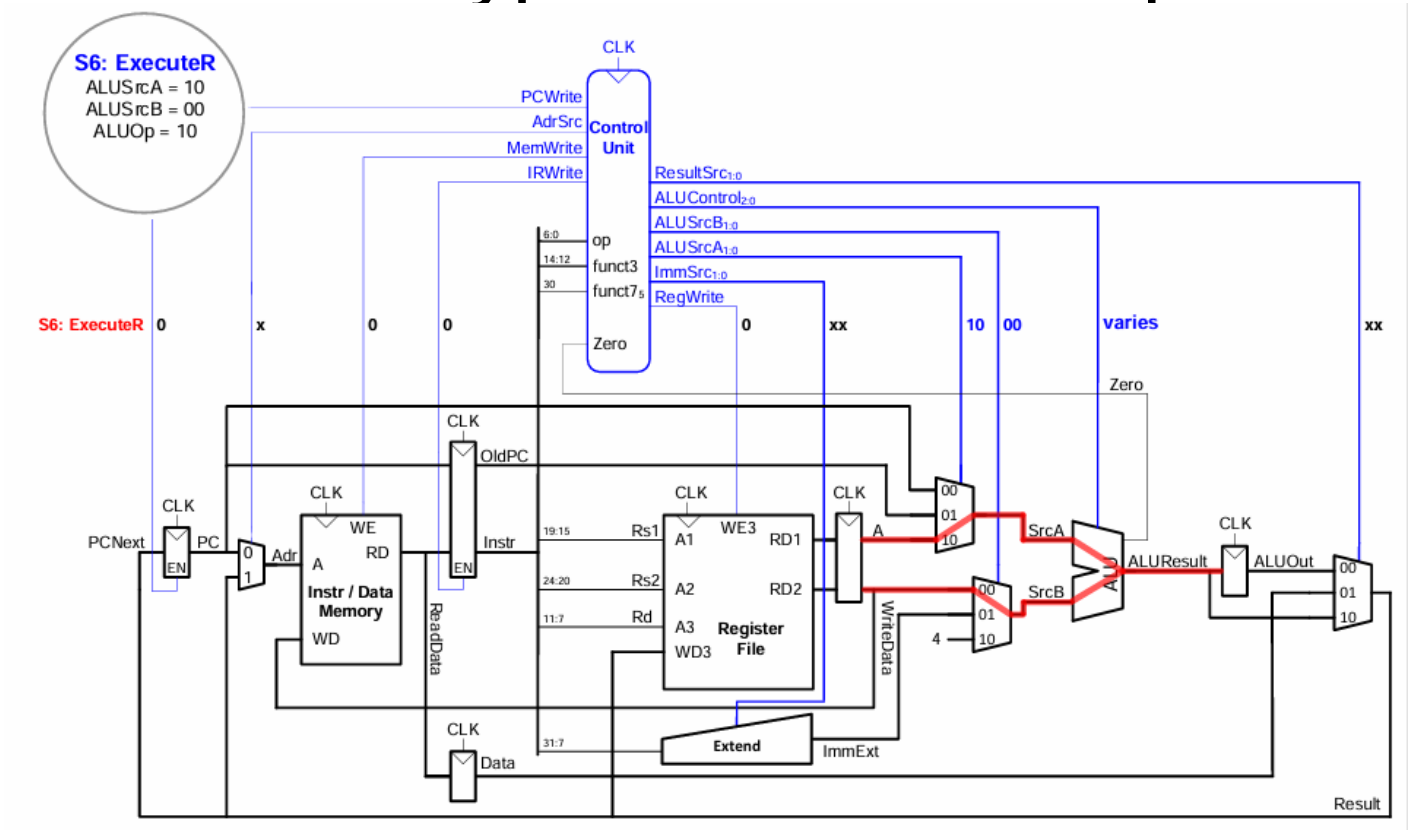
Main FSM: `sw` Datapath



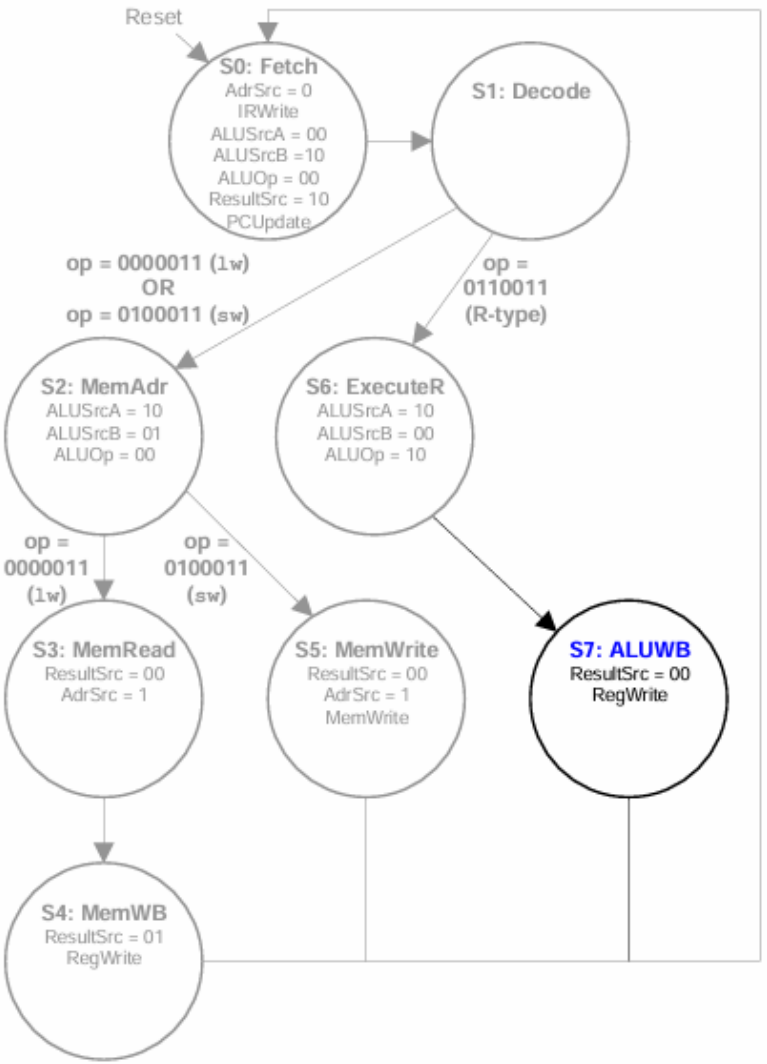
Main FSM: R-Type Execute



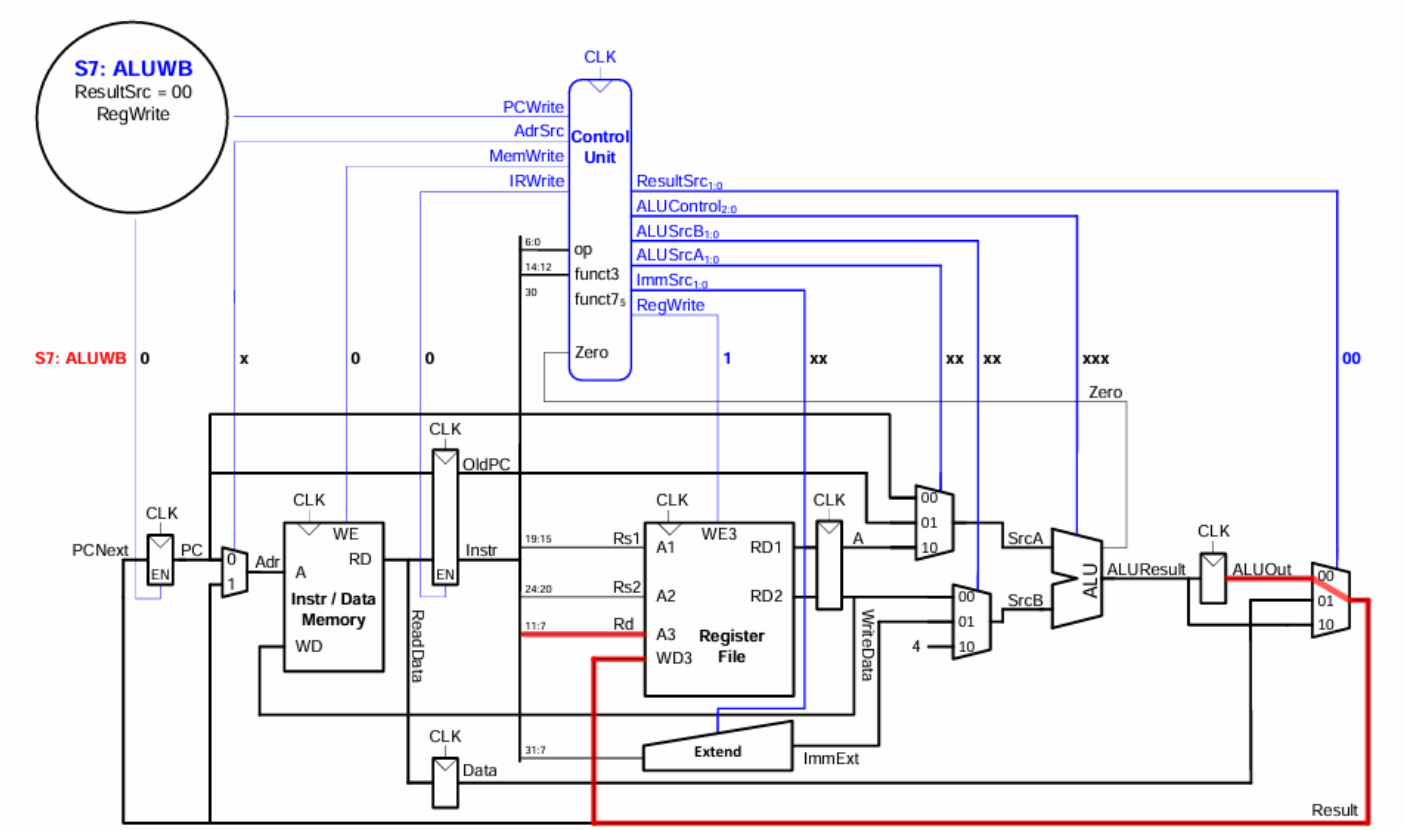
Main FSM: R-Type Execute Datapath



Main FSM: R-Type ALU Write Back



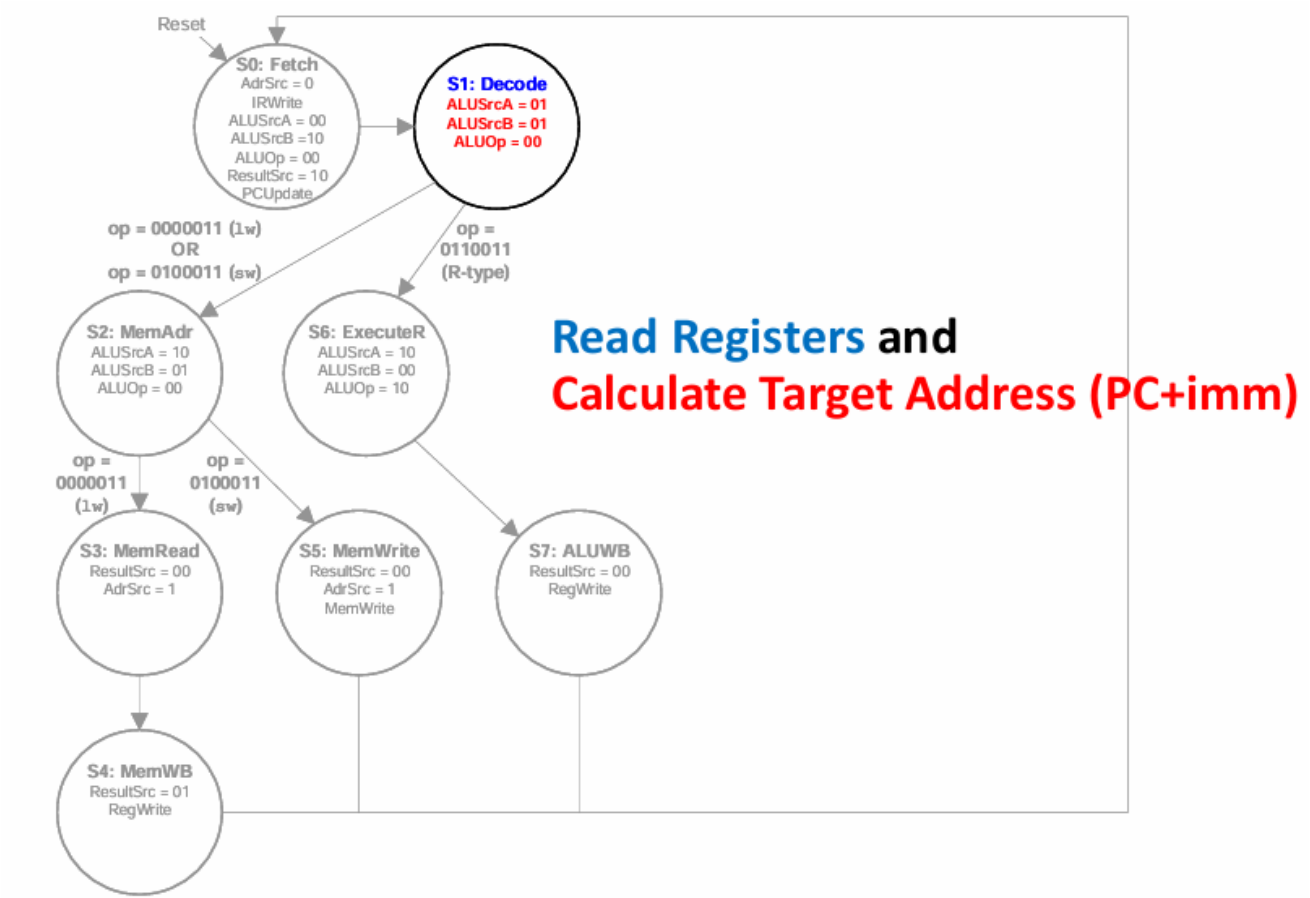
Main FSM: R-Type Writeback Datapath



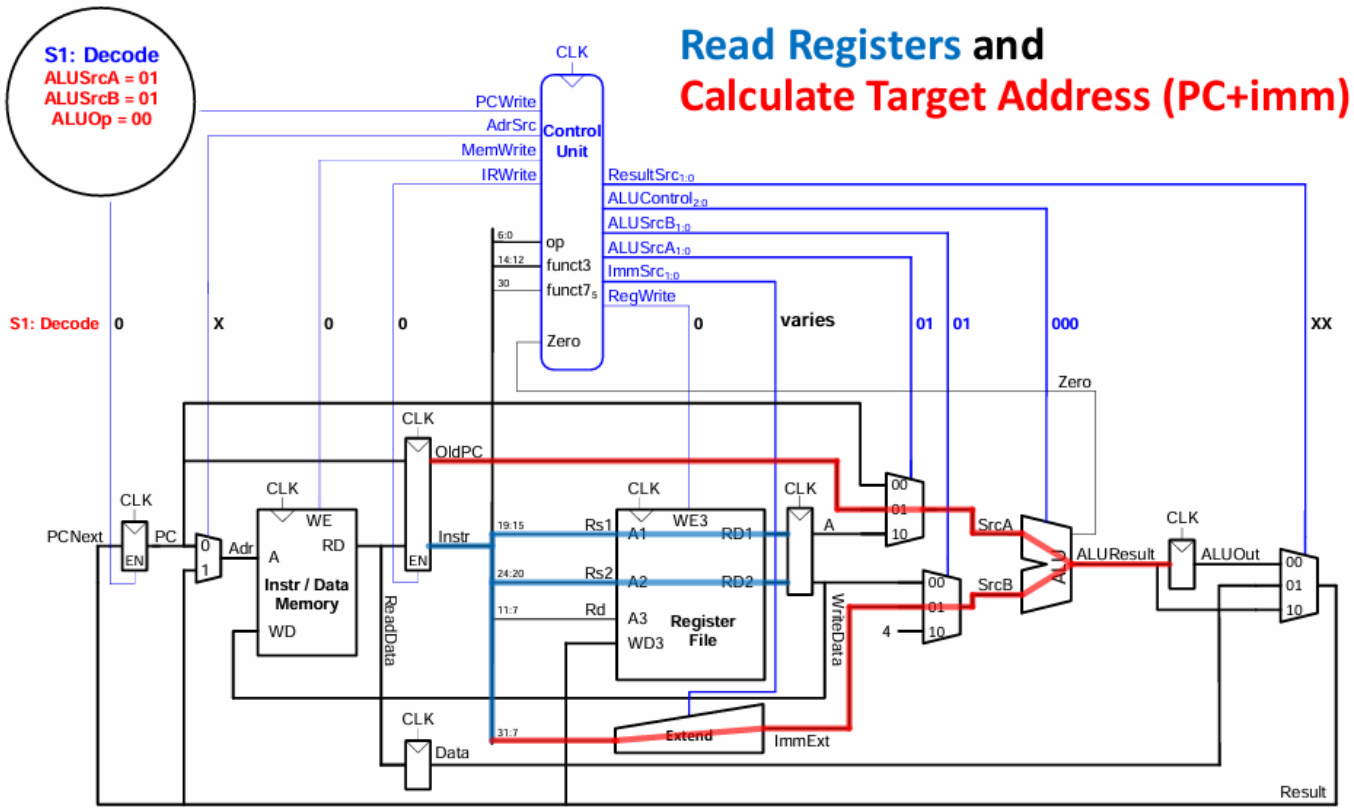
Main FSM: `beq`

- **Need to calculate:**
  - Branch Target Address
  - **rs1** - **rs2** (to see if equal)
- **ALU** isn't being used in Decode stage
  - Use it to calculate Target Address (PC + imm)

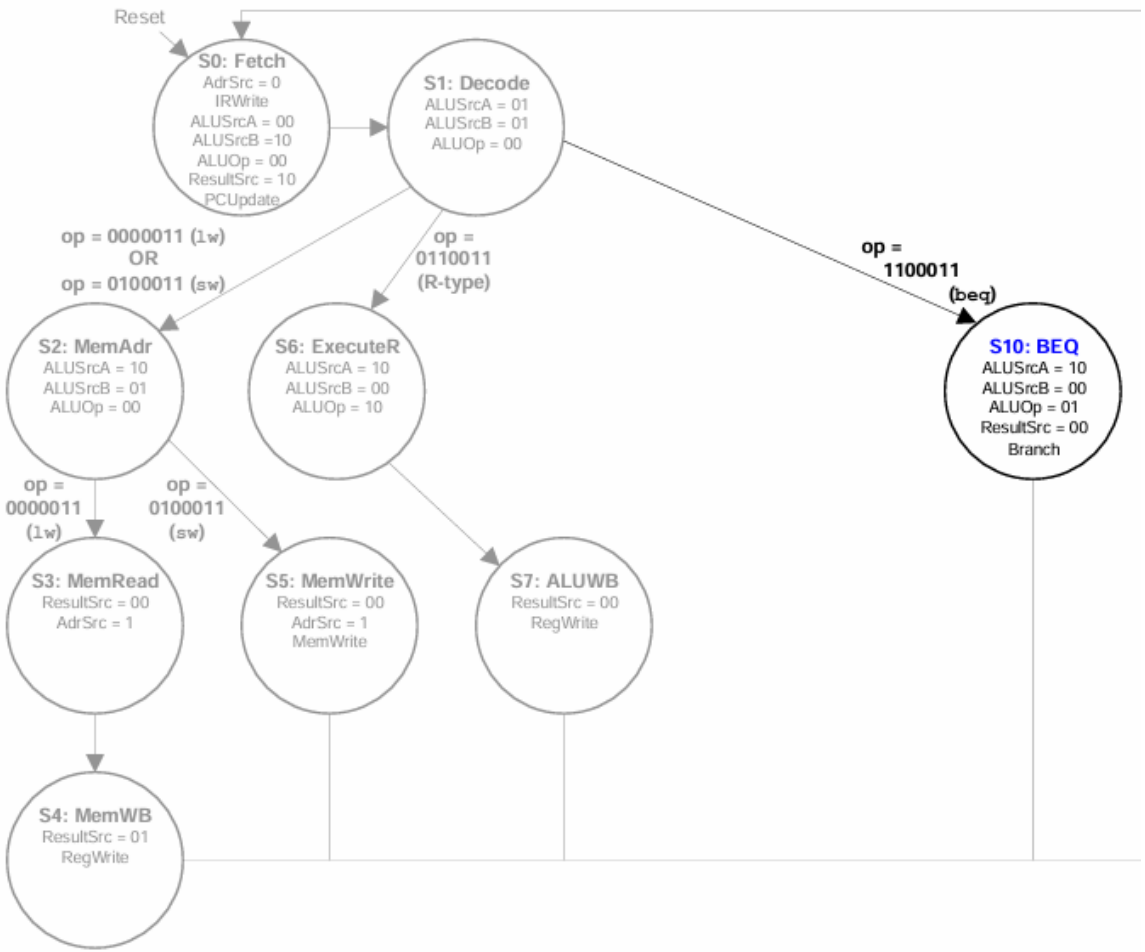
Main FSM: Decode Revisited



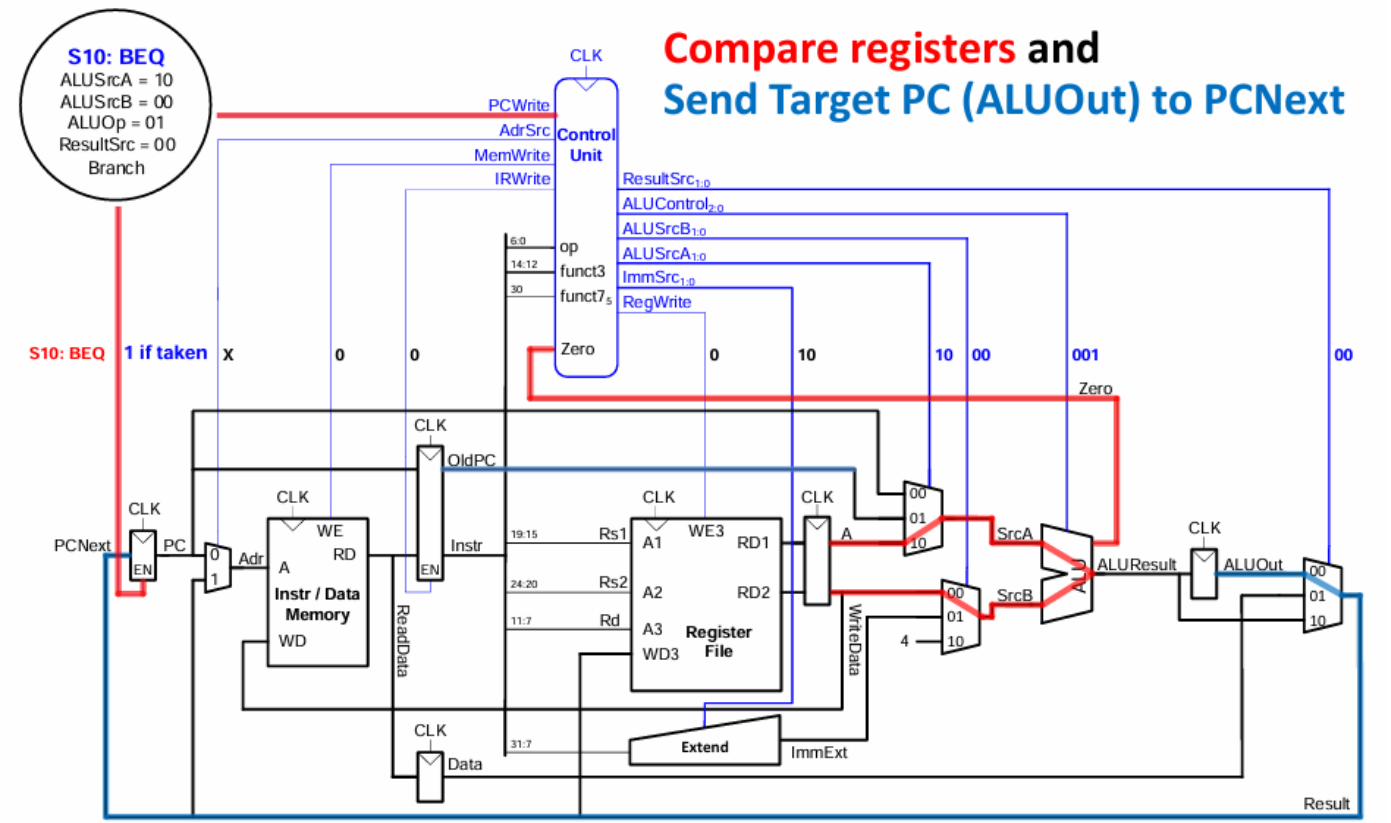
Main FSM: Decode (Target Address)



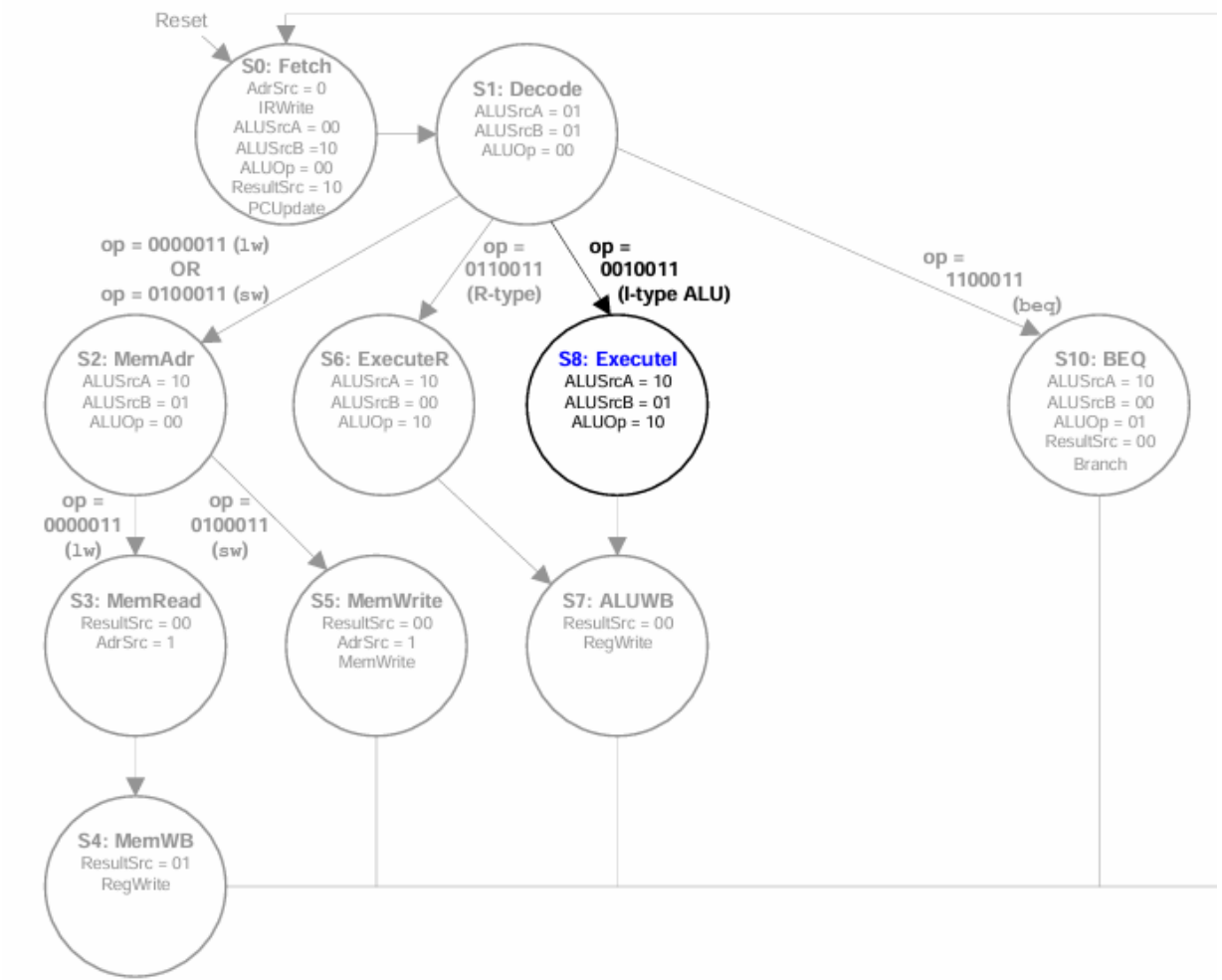
Main FSM: beq


Reset
S0: Fetch
AdrSrc = 0
IRWrite
ALUSrcA = 00
ALUSrcB =10
ALUOp = 00
ResultSrc = 10
PCUpdate
S1: Decode
ALUSrcA = 01
ALUSrcB = 01
ALUOp = 00
op = 0000011 (lw)
OR
op = 0100011 (sw)
op = 0110011 (R-type)
op = 1100011 (beq)
S2: MemAdr
ALUSrcA = 10
ALUSrcB = 01
ALUOp = 00
S6: ExecuteR
ALUSrcA = 10
ALUSrcB = 00
ALUOp = 10
S10: BEQ
ALUSrcA = 10
ALUSrcB = 00
ALUOp = 01
ResultSrc = 00
Branch
op = 0000011 (lw)
op = 0100011 (sw)
S3: MemRead
ResultSrc = 00
AdrSrc = 1
S5: MemWrite
ResultSrc = 00
AdrSrc = 1
MemWrite
S7: ALUWB
ResultSrc = 00
RegWrite
S4: MemWB
ResultSrc = 01
RegWrite

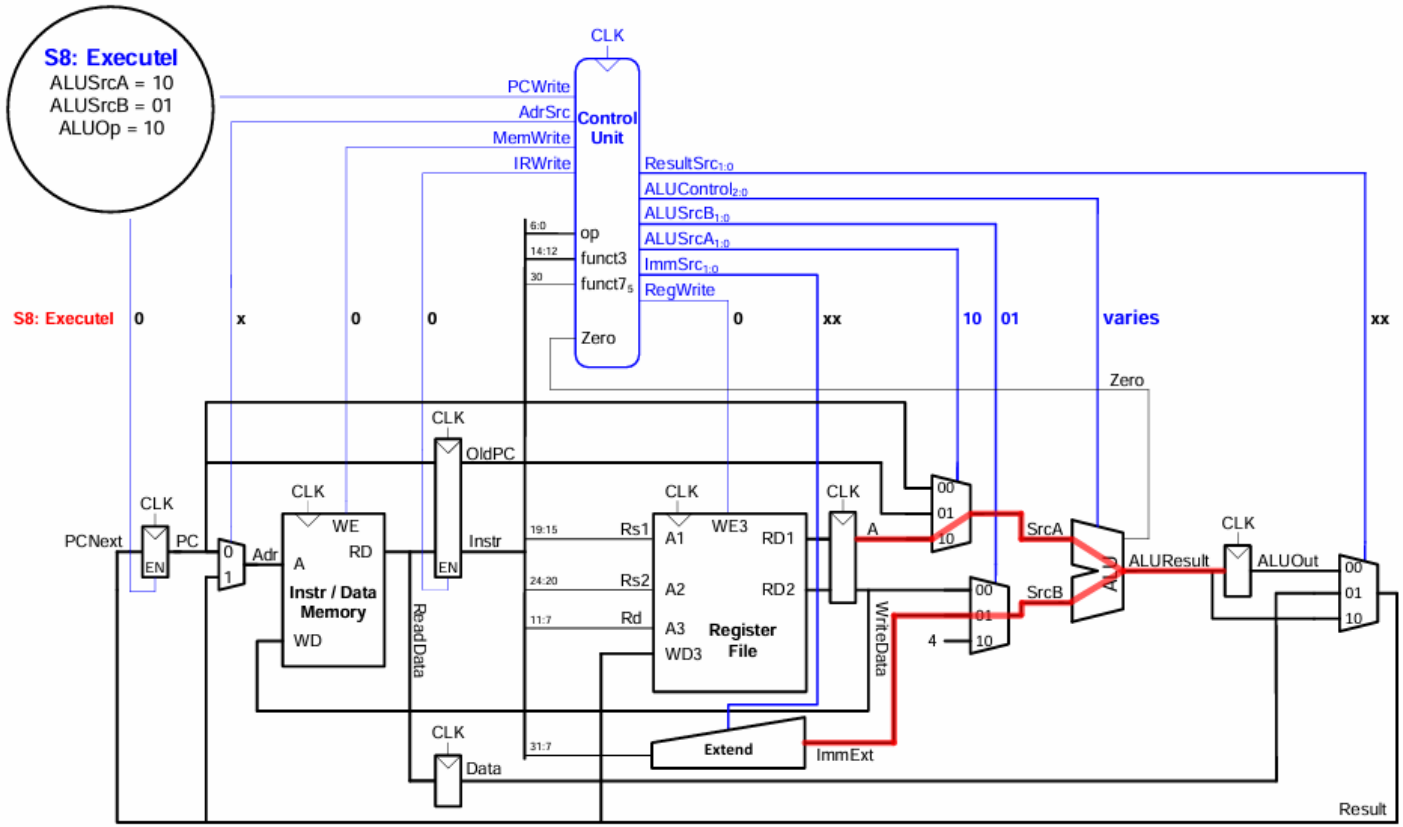
Main FSM: `beq` Datapath



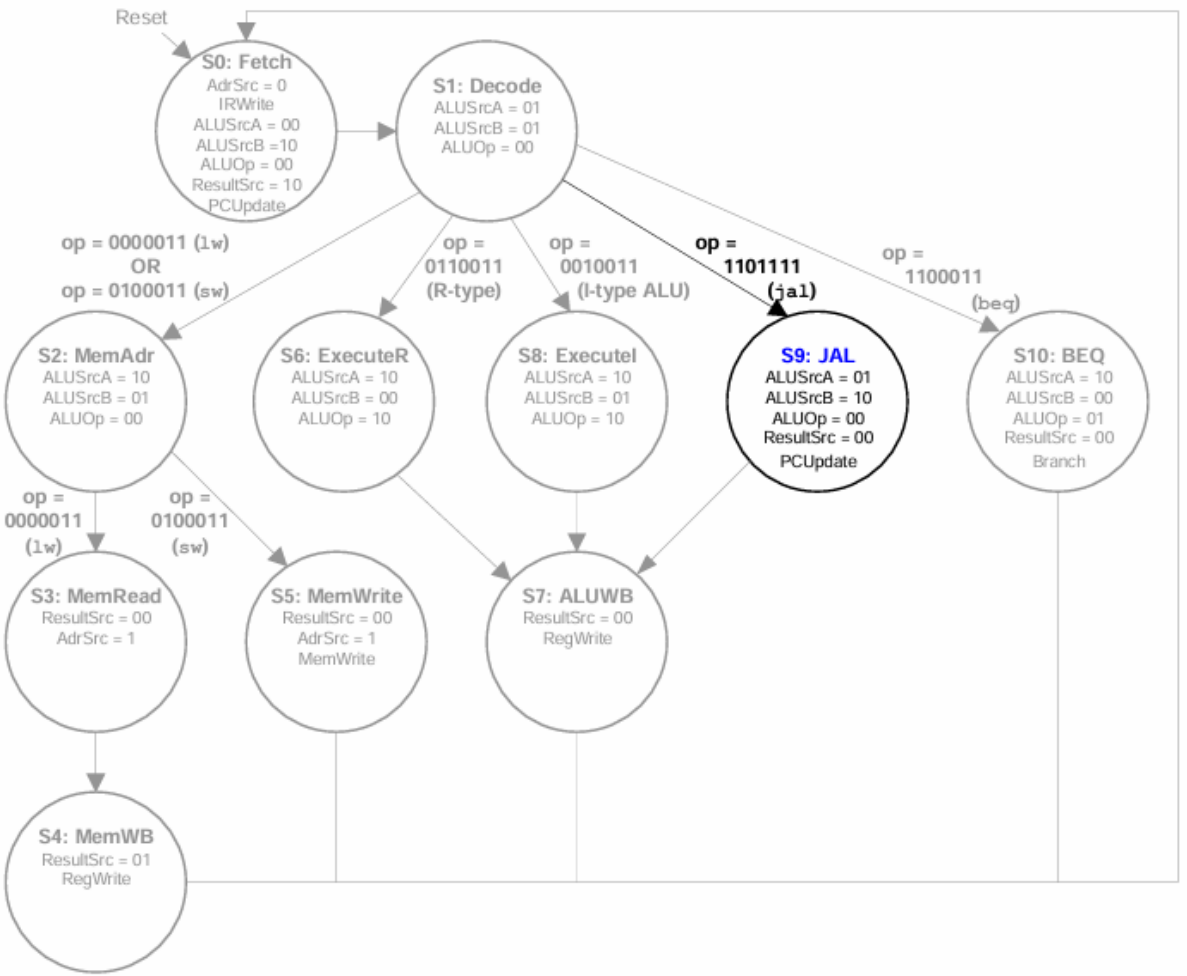
Main FSM: I-Type ALU Execute



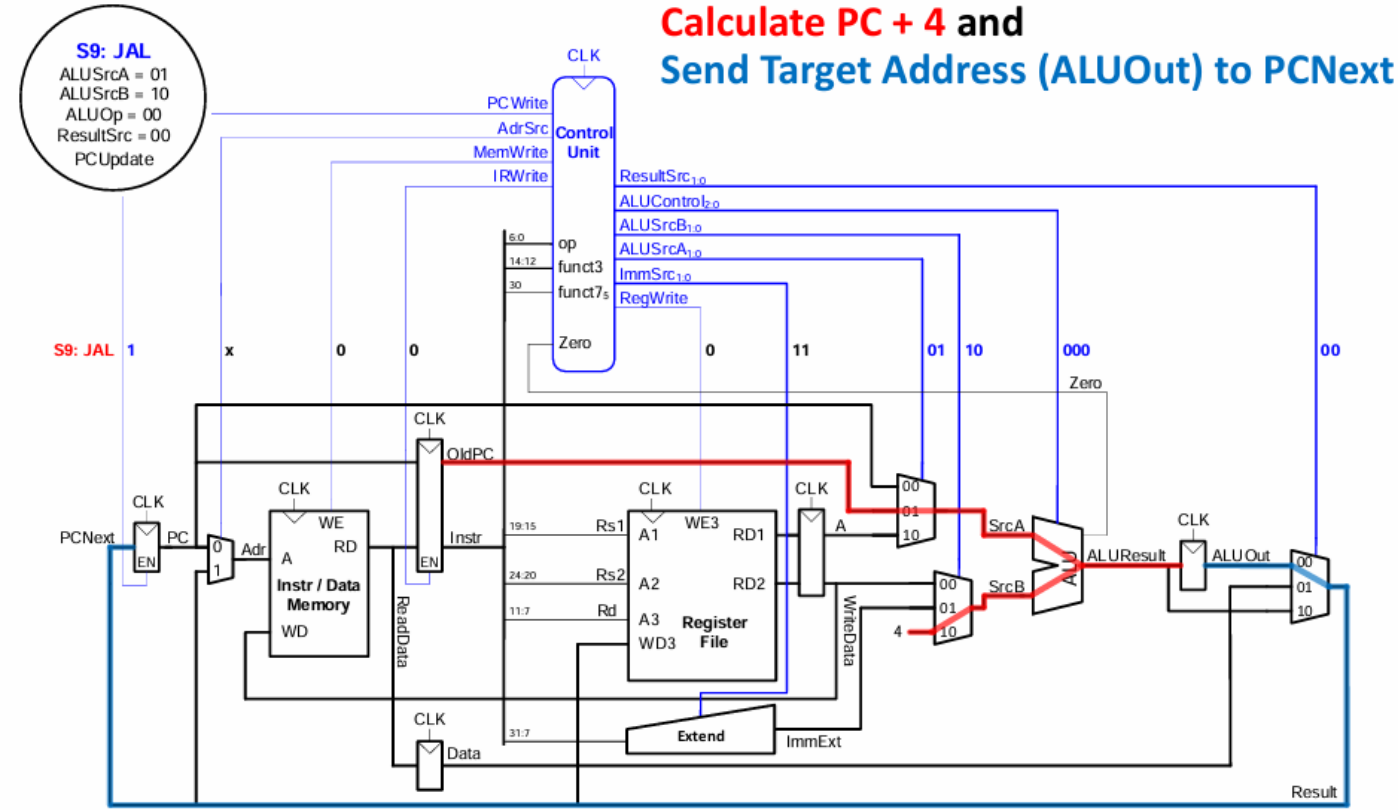
Main FSM: I-Type ALU Exec. Datapath



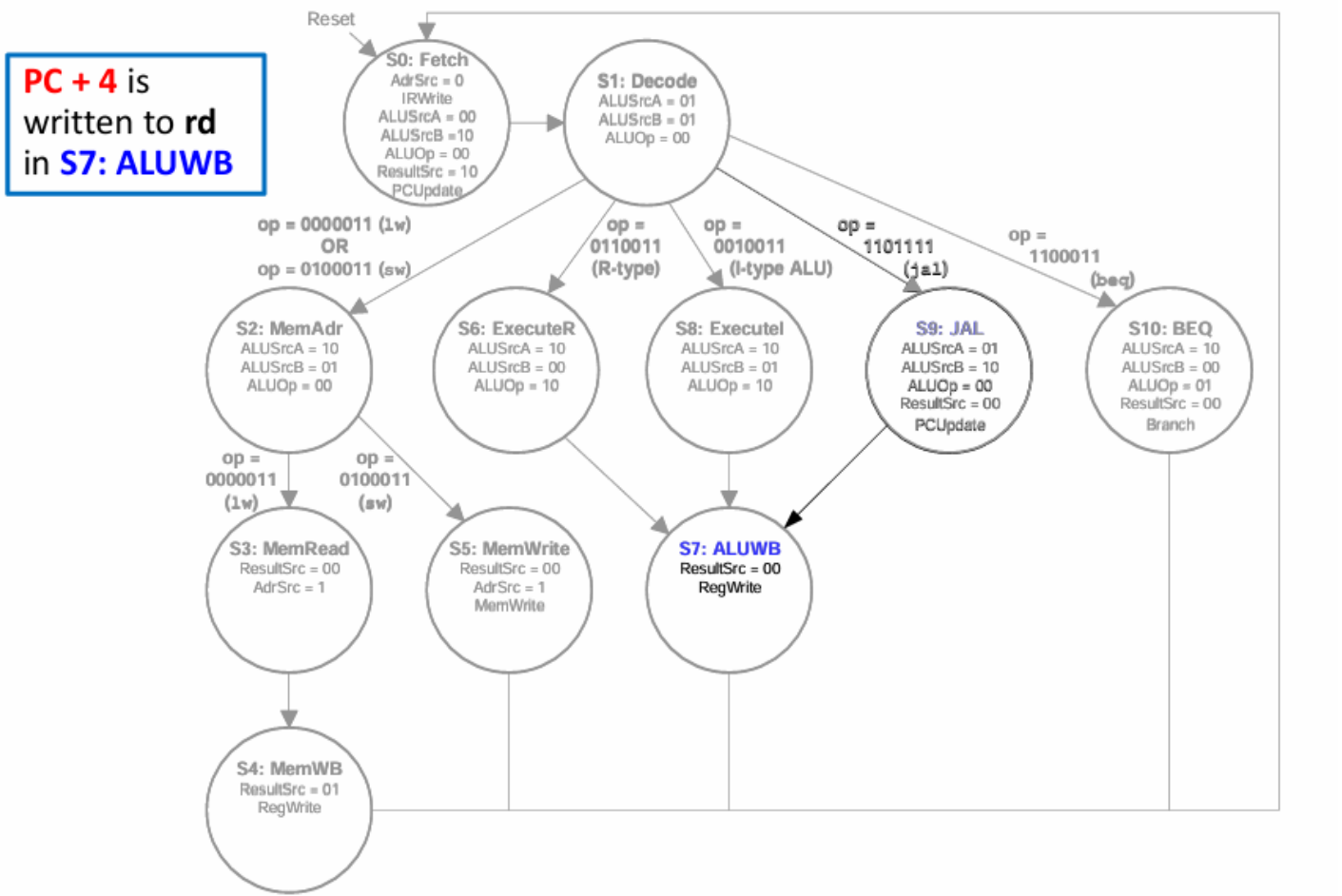
Main FSM: `jal`


Reset
S0: Fetch
AdrSrc = 0
IRWrite
ALUSrcA = 00
ALUSrcB =10
ALUOp = 00
ResultSrc = 10
PCUpdate
S1: Decode
ALUSrcA = 01
ALUSrcB = 01
ALUOp = 00
op = 0000011 (lw)
OR
op = 0100011 (sw)
op = 0110011 (R-type)
op = 0010011 (I-type ALU)
op = 1101111 (jal)
op = 1100011 (beq)
S2: MemAdr
ALUSrcA = 10
ALUSrcB = 01
ALUOp = 00
S6: ExecuteR
ALUSrcA = 10
ALUSrcB = 00
ALUOp = 10
S8: ExecuteI
ALUSrcA = 10
ALUSrcB = 01
ALUOp = 10
S9: JAL
ALUSrcA = 01
ALUSrcB = 10
ALUOp = 00
ResultSrc = 00
PCUpdate
S10: BEQ
ALUSrcA = 10
ALUSrcB = 00
ALUOp = 01
ResultSrc = 00
Branch
op = 0000011 (lw)
op = 0100011 (sw)
S3: MemRead
ResultSrc = 00
AdrSrc = 1
S5: MemWrite
ResultSrc = 00
AdrSrc = 1
MemWrite
S7: ALUWB
ResultSrc = 00
RegWrite
S4: MemWB
ResultSrc = 01
RegWrite

Main FSM: `jal` Datapath

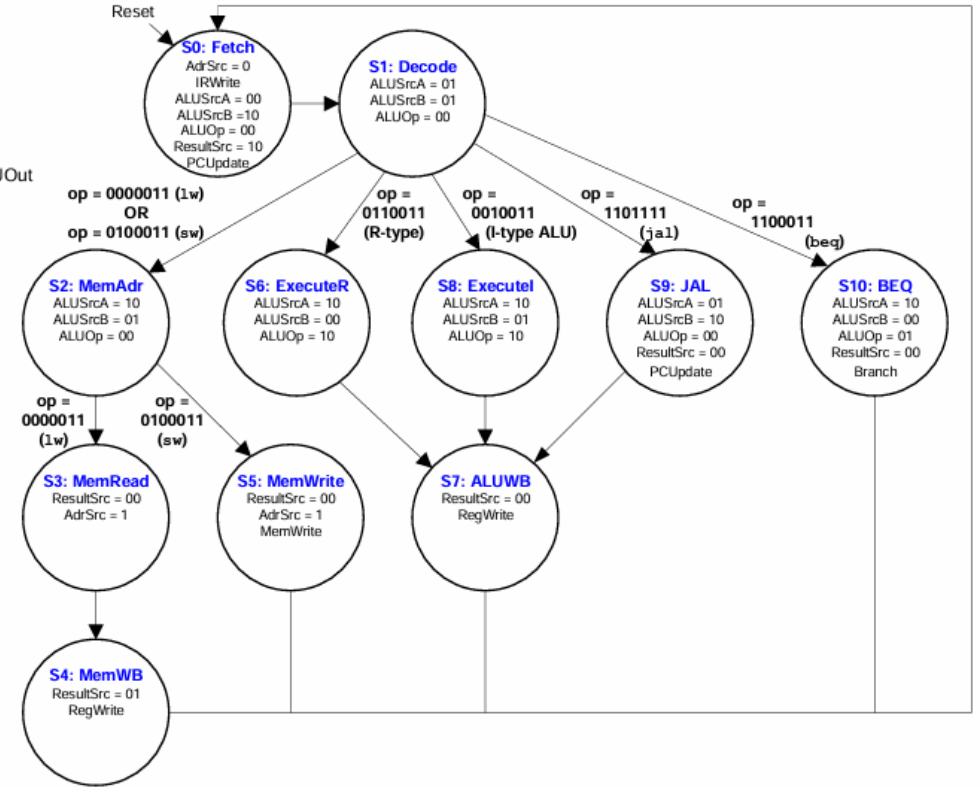


Main FSM: `jal`


PC + 4 is written to rd in S7: ALUWB
Reset
S0: Fetch
AdrSrc = 0
IRWrite
ALUSrcA = 00
ALUSrcB =10
ALUOp = 00
ResultSrc = 10
PCUpdate
S1: Decode
ALUSrcA = 01
ALUSrcB = 01
ALUOp = 00
op = 0000011 (lw) OR op = 0100011 (sw)
op = 0110011 (R-type)
op = 0010011 (I-type ALU)
op = 1101111 (jal)
op = 1100011 (beq)
S2: MemAdr
ALUSrcA = 10
ALUSrcB = 01
ALUOp = 00
S6: ExecuteR
ALUSrcA = 10
ALUSrcB = 00
ALUOp = 10
S8: ExecuteI
ALUSrcA = 10
ALUSrcB = 01
ALUOp = 10
S9: JAL
ALUSrcA = 01
ALUSrcB = 10
ALUOp = 00
ResultSrc = 00
PCUpdate
S10: BEQ
ALUSrcA = 10
ALUSrcB = 00
ALUOp = 01
ResultSrc = 00
Branch
op = 0000011 (lw)
op = 0100011 (sw)
S3: MemRead
ResultSrc = 00
AdrSrc = 1
S5: MemWrite
ResultSrc = 00
AdrSrc = 1
MemWrite
S7: ALUWB
ResultSrc = 00
RegWrite
S4: MemWB
ResultSrc = 01
RegWrite

Multicycle Processor Main FSM

State	Datapath μOp
Fetch	Instr ←Mem[PC]; PC ← PC+4
Decode	ALUOut ← PCTarget
MemAdr	ALUOut ← rs1 + imm
MemRead	Data ← Mem[ALUOut]
MemWB	rd ← Data
MemWrite	Mem[ALUOut] ← rd
ExecuteR	ALUOut ← rs1 op rs2
ExecuteI	ALUOut ← rs1 op imm
ALUWB	rd ← ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC ← ALUOut
JAL	PC ← ALUOut; ALUOut ← PC+4



Processor Performance

Program Execution Time

= (#instructions)(cycles/instruction)(seconds/cycle)

= # instructions x CPI x T_C

Single-Cycle Performance

- **Single-cycle critical path:**

$$T_{c1} = t_{pcq\_PC} + t_{\text{mem}} + \max[t_{RF\text{read}}, t_{dec} + t_{ext} + t_{\text{mux}}] + t_{\text{ALU}} + t_{\text{mem}} + t_{\text{mux}} + t_{RF\text{setup}}$$

- **Typically, limiting paths are:**
  - memory, ALU, register file
  - $T_{c1} = t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{ALU}} + t_{\text{mux}} + t_{RF\text{setup}}$

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
ALU	t_{ALU}	120
Decoder (Control Unit)	t_{dec}	35
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

$T_{c1} = ?$

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
ALU	t_{ALU}	120
Decoder (Control Unit)	t_{dec}	35
Memory read	t_{mem}	200
Register file read	$t_{RF\text{read}}$	100
Register file setup	$t_{RF\text{setup}}$	60

$$T_{c1} = t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{ALU}} + t_{\text{mux}} + t_{RF\text{setup}}$$

$$= [40 + 2(200) + 100 + 120 + 30 + 60]\ \text{ps}$$

$$= \mathbf{750\ ps}$$

Single-Cycle Performance Example

Program with 100 billion instructions:

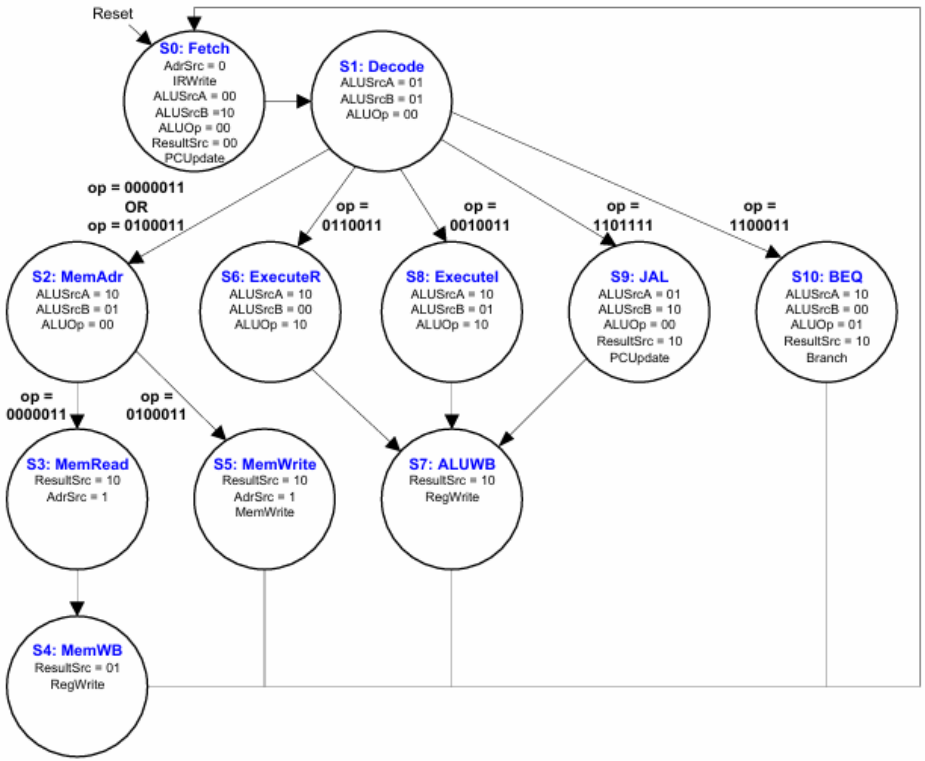
Execution Time = # instructions x CPI x T_C

$= (100 \times 10^9)(1)(750 \times 10^{-12}\ s)$

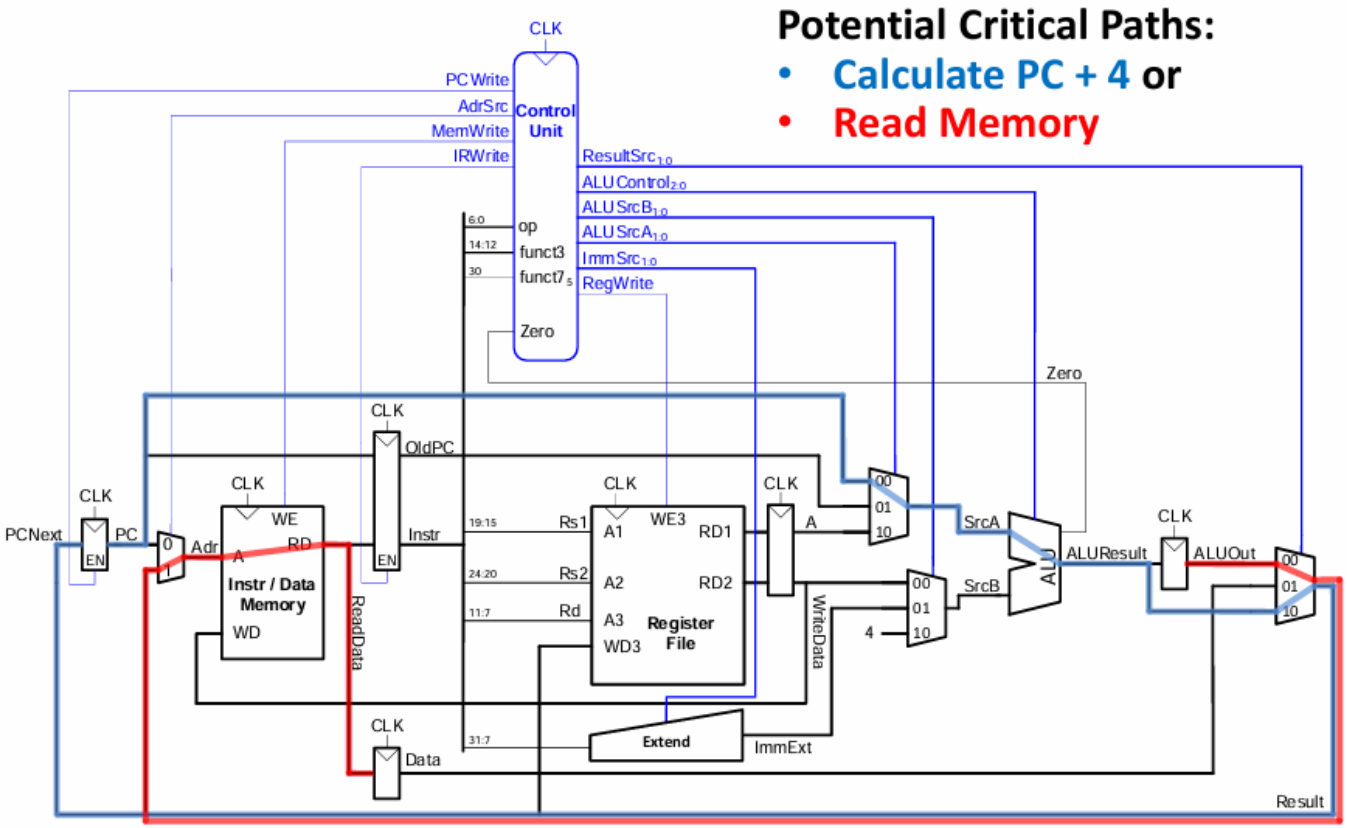
$=$ **75 seconds**

Multicycle Processor Performance

- Instructions take different number of cycles:



Multicycle Processor Critical Path

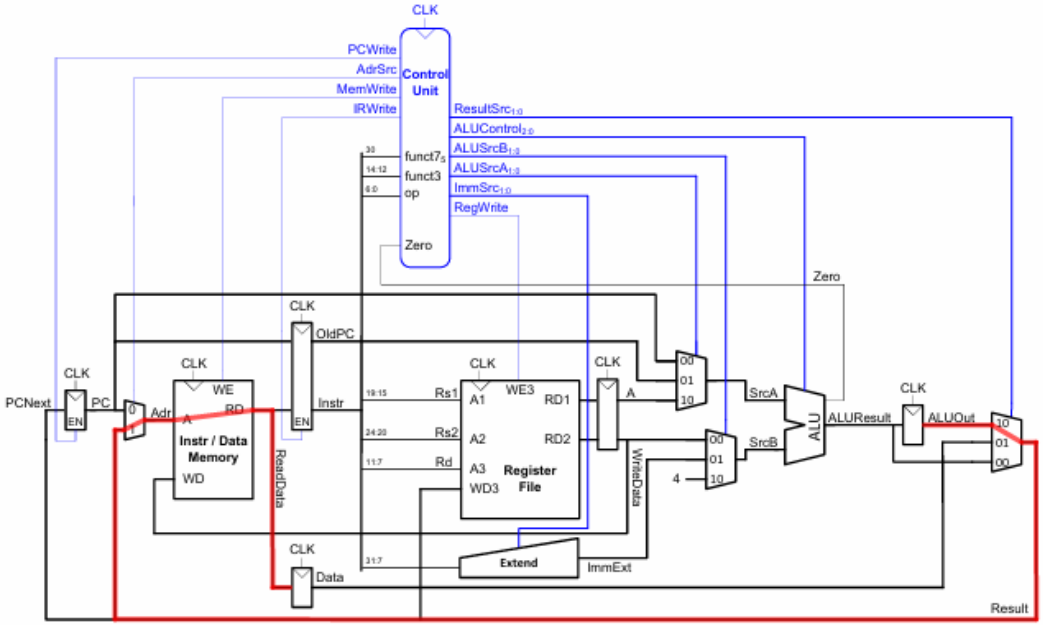


Multicycle Processor Performance

- Assumptions:
  - RF is faster than memory
  - writing memory is faster than reading memory
- Two possibilities:
  - Read memory (MemRead state)
  - PC = PC + 4 path (Fetch state)

Multicycle Processor Performance

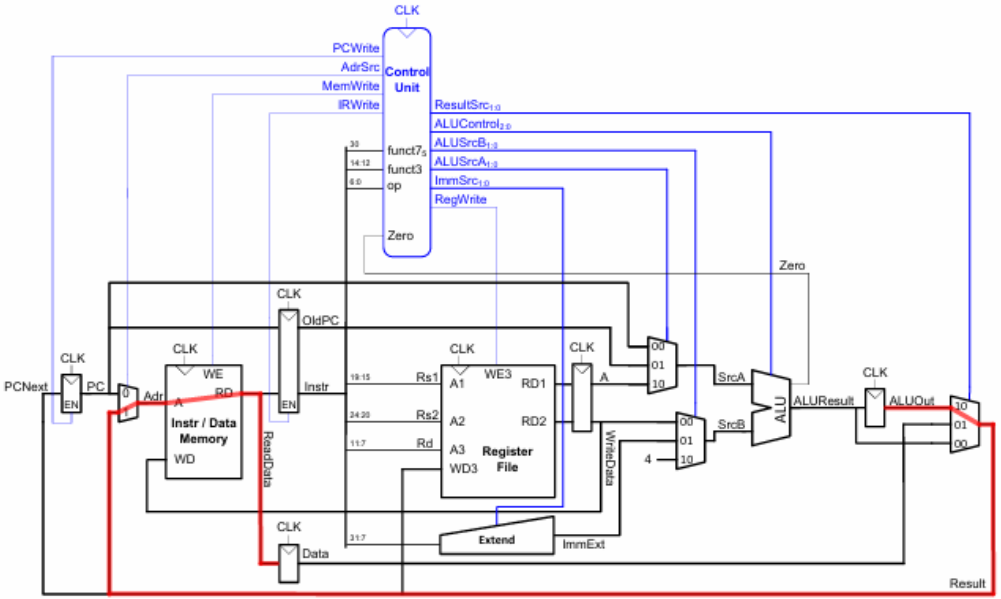
Option 1: Read memory (MemRead state)



$$T_{c2} = t_{pcq} + t_{\text{mux}} + t_{\text{mux}} + t_{\text{mem}} + t_{\text{setup}}$$

Multicycle Processor Performance

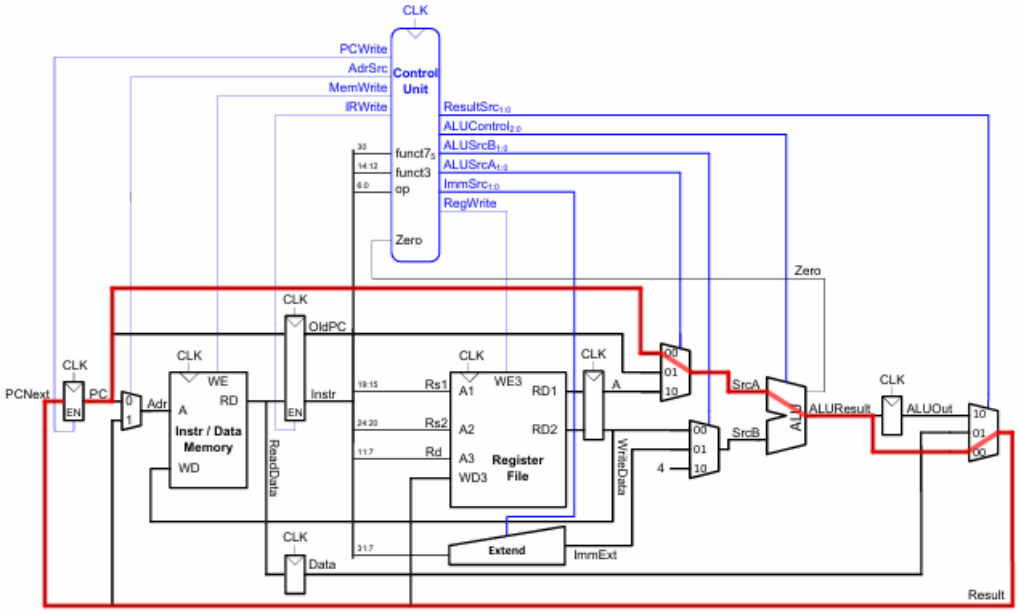
Option 1: Read memory (MemRead state)



$$T_{c2} = t_{pcq} + t_{\text{mux}} + t_{\text{mux}} + t_{\text{mem}} + t_{\text{setup}}$$
$$= t_{pcq} + 2t_{\text{mux}} + t_{\text{mem}} + t_{\text{setup}}$$

Multicycle Processor Performance

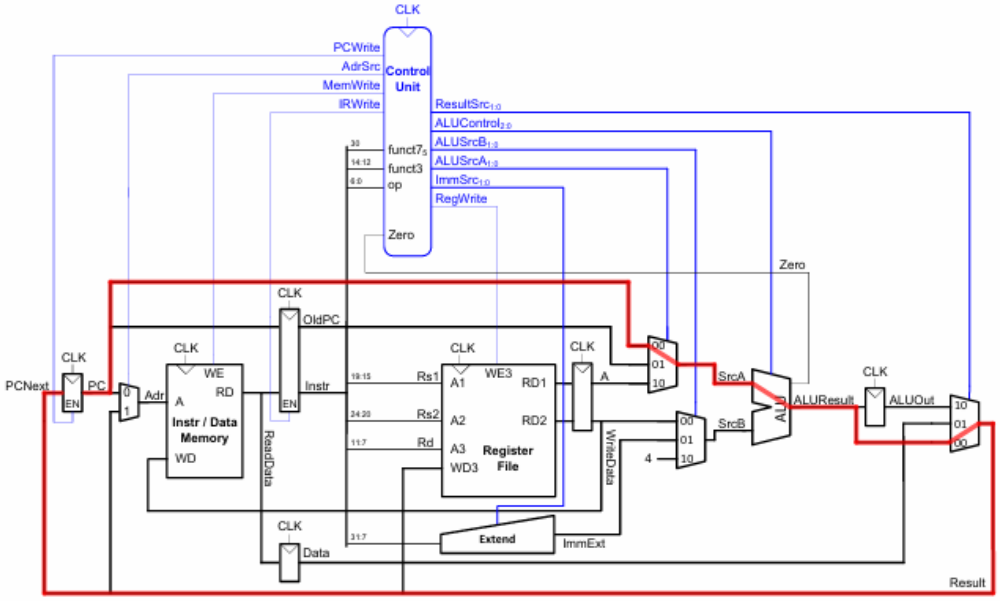
Option 2: PC = PC + 4 path (Fetch state)



$$T_{c2} = t_{pcq} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{mux}} + t_{\text{setup}}$$

Multicycle Processor Performance

Option 2: PC = PC + 4 path (Fetch state)



$$T_{c2} = t_{pcq} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{mux}} + t_{\text{setup}}$$
$$= t_{pcq} + 2t_{\text{mux}} + t_{\text{ALU}} + t_{\text{setup}}$$

Multicycle Processor Critical Path

- Two possibilities:
  - Read memory (MemRead state)
  - PC = PC + 4 path (Fetch state)

$$T_{c2} = t_{pcq} + 2t_{\text{mux}} + \max[t_{\text{ALU}}, t_{\text{mem}}] + t_{\text{setup}}$$

Multicycle Performance Example

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

$$T_{c2} = t_{pcq} + 2t_{\mathrm{mux}} + \max[t_{\mathrm{ALU}}, t_{\mathrm{mem}}] + t_{\mathrm{setup}}$$
$$= (40 + 2(30) + 200 + 50)\ \mathrm{ps} = \mathbf{350\ ps}$$

Multicycle Processor Performance

- Instructions take different number of cycles:
  - 3 cycles: `beq`
  - 4 cycles: R-type, `addi`, `sw`, `jal`
  - 5 cycles: `lw`
- CPI is weighted average
- SPECINT2000 benchmark:
  - **25%** loads
  - **10%** stores
  - **13%** branches
  - **52%** R-type

Average CPI = (**0.13**)(3) + (**0.52 + 0.10**)(4) + (**0.25**)(5) = **4.12**

Multicycle Performance Example

For a program with **100 billion** instructions executing on a **multicycle** RISC-V processor

- **CPI** = 4.12 cycles/instruction
- **Clock cycle time:** T_{c2} = 350 ps

Execution Time = (# instructions) × CPI × T_c

$= (100 \times 10^9)(4.12)(350 \times 10^{-12})$

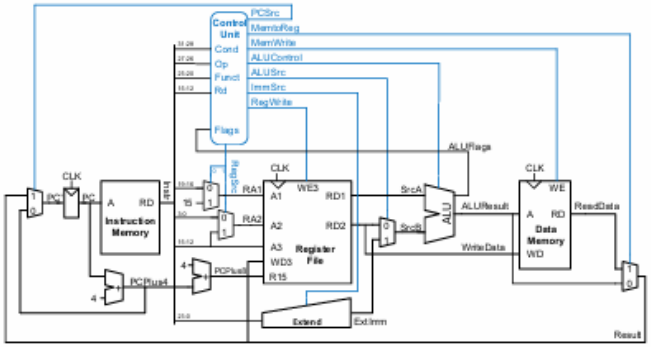
$=$ **144 seconds**

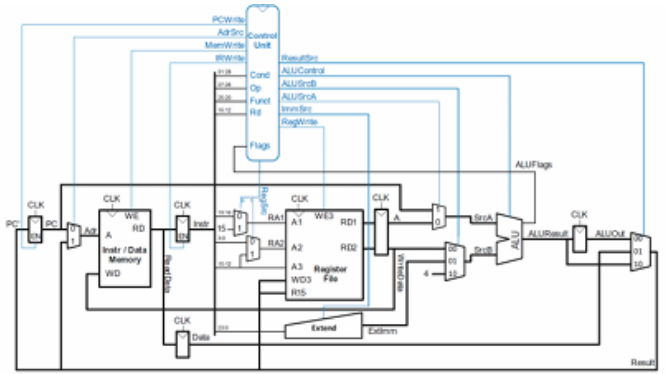
This is slower than the single-cycle processor (75 sec.) Why?

Multicycle Performance Example

- In this case, the multicycle processor is slower than the single-cycle processor, why?
  - Not all steps the same length
  - Sequencing overhead for each step (t_{pcq}+t_{setup}) = (40+50 = 90)

Processor Comparison





Single Cycle	**Multicycle**
One cycle/instruction	3-5 cycles/instruction
Long clock period	Shorter clock period
Separate I and D Mem	Unified Memory
Combinational controller	FSM controller
Architectural State Only	Extra state