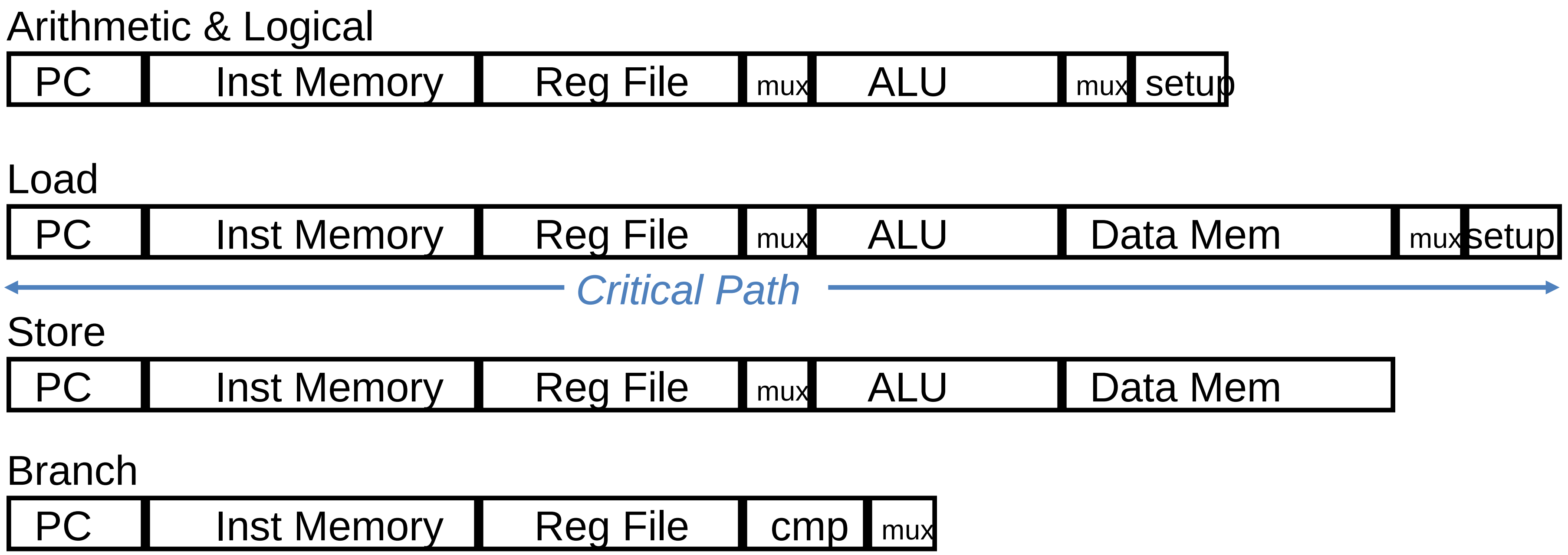
CO101
Principle of Computer Organization

Lecture 5: Multi-Cycle Processor

Liang Yanyan

澳門科技大學
Macau of University of Science and Technology

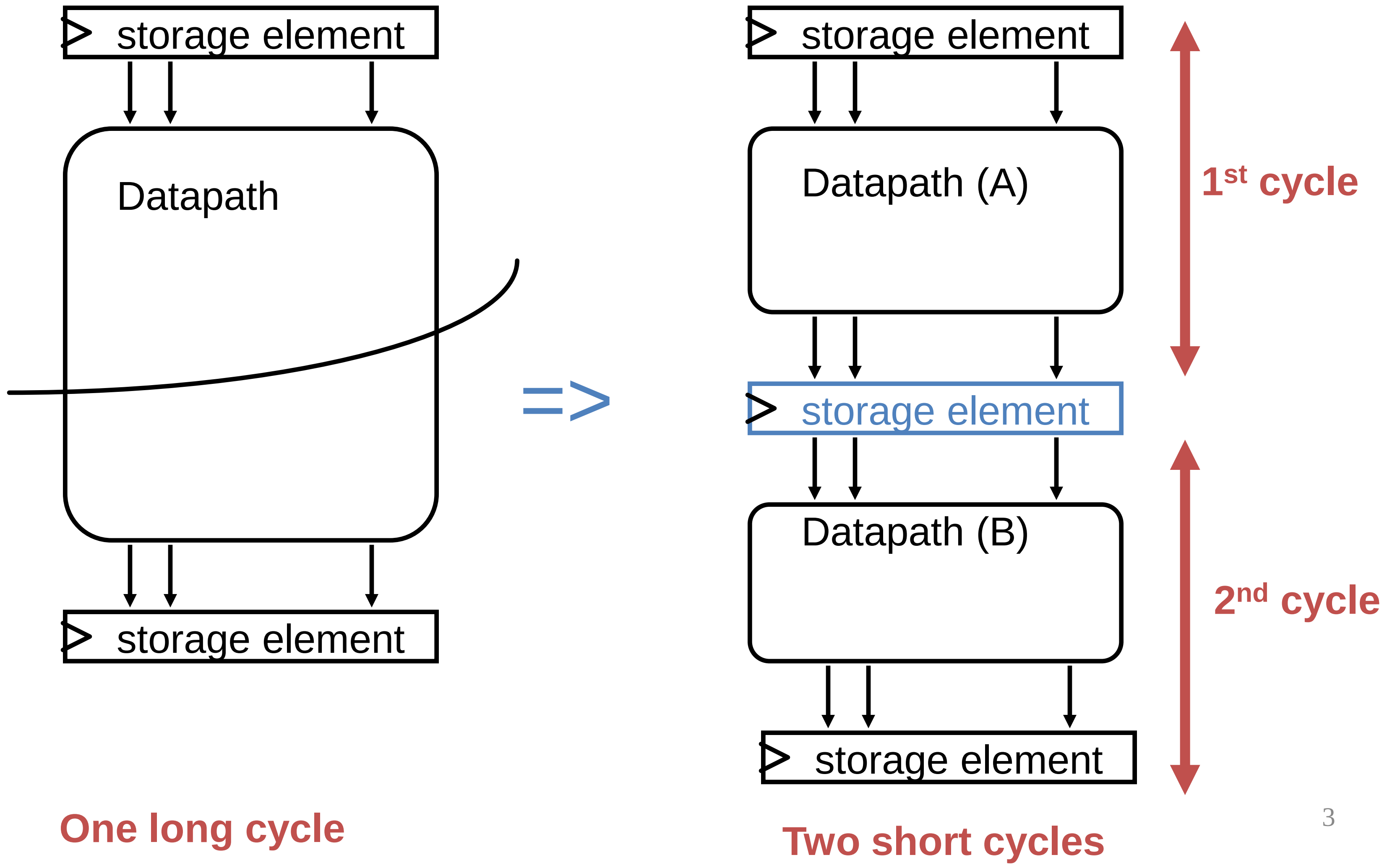
What's wrong with our CPI=1 processor?



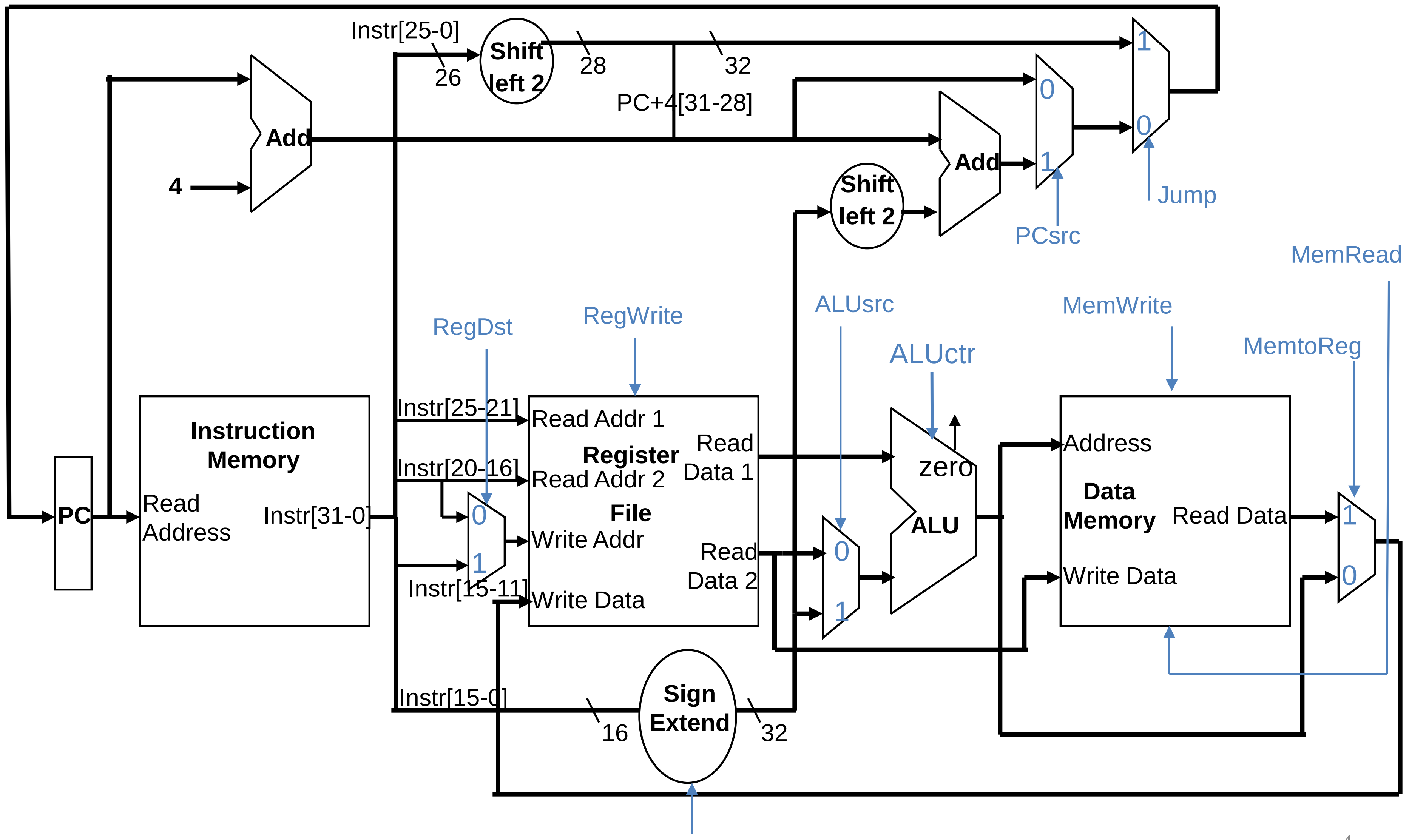
- Long Cycle Time.
- Real memory is not so nice as our idealized memory.
  - cannot always get the job done in one (short) cycle.

Reducing Cycle Time

- Cut combinational dependency graph and insert register.
- Do same work in two short cycles, rather than one long cycle.

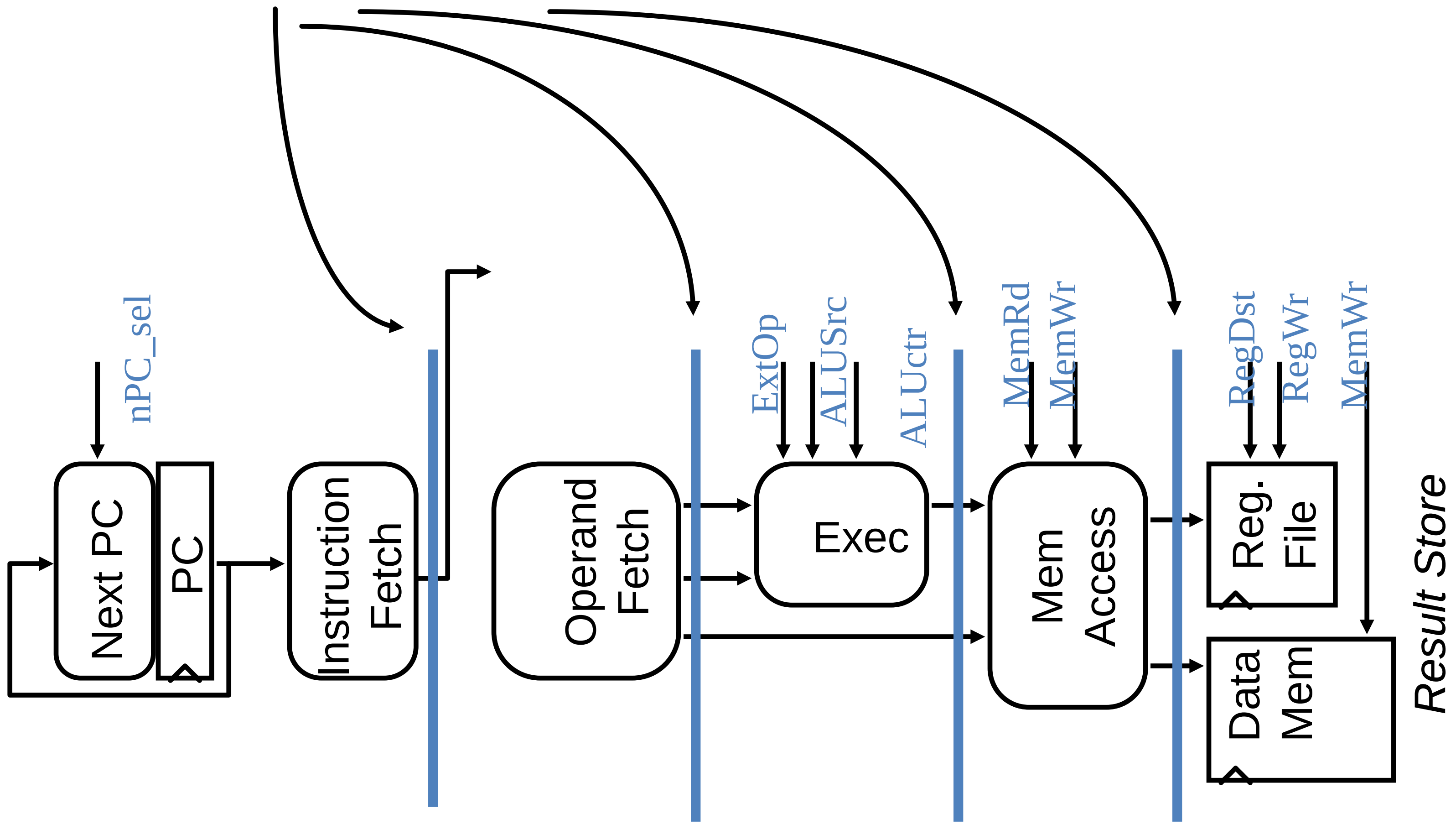


Review: A Single Cycle Datapath



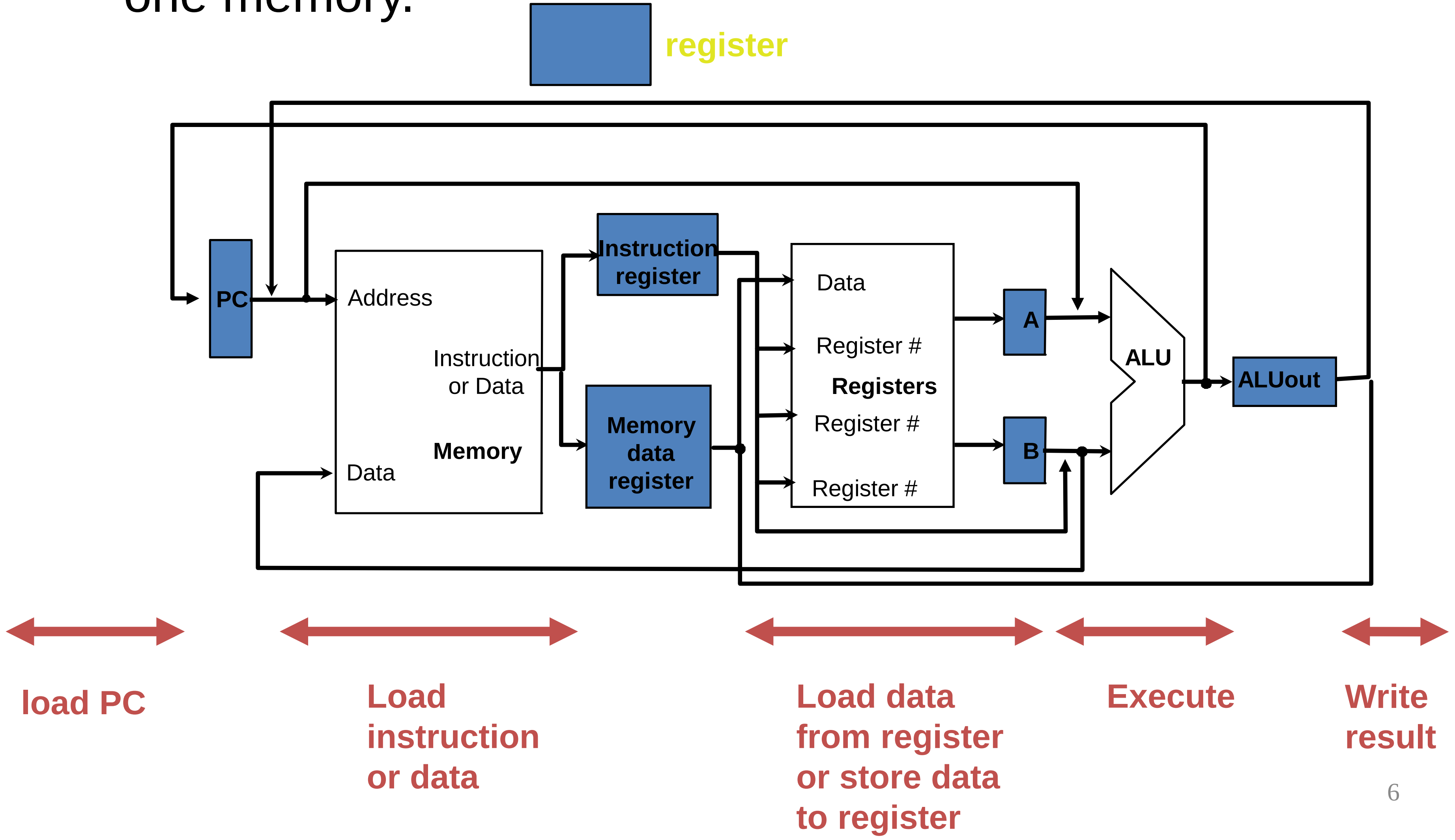
Partition the CPI=1 Datapath

- Add registers between partitions.



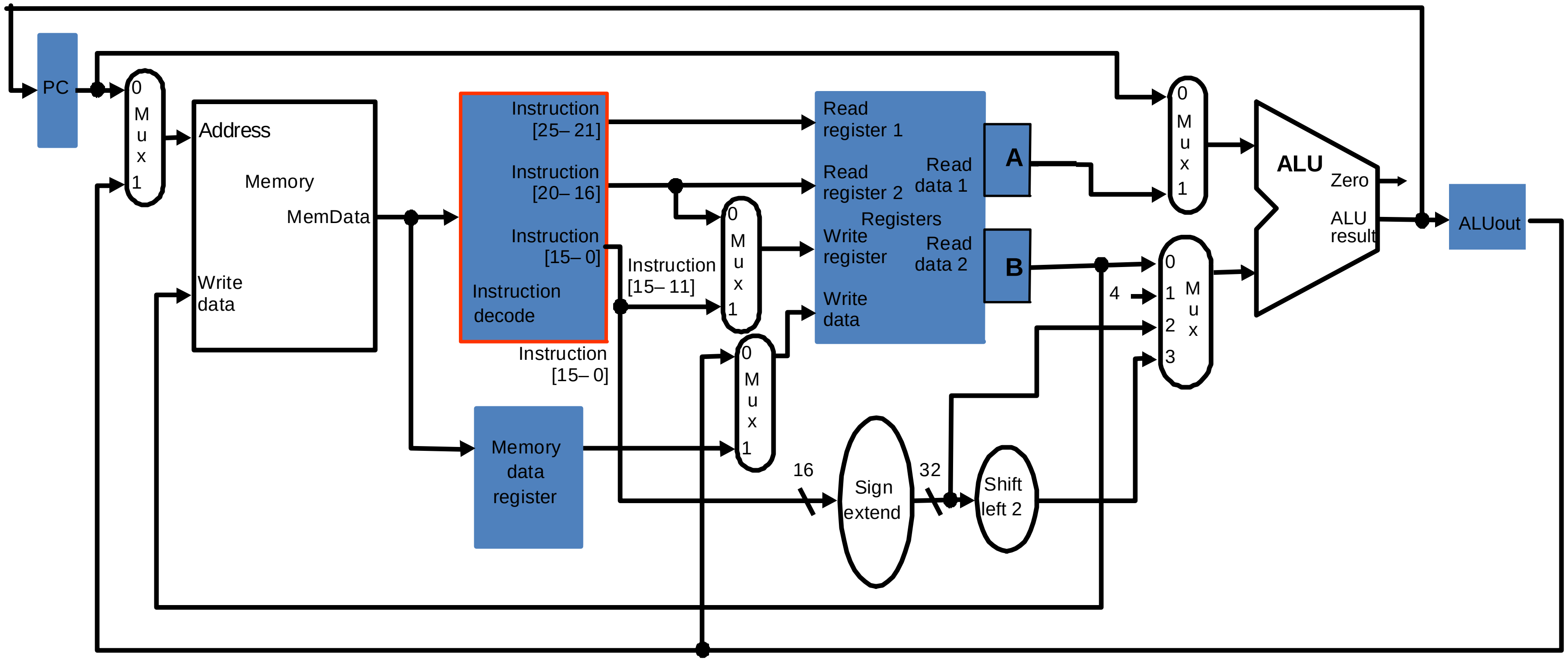
Partition the CPI=1 Datapath

- Instruction memory and data memory are combined into one memory.



A more detailed view

- Instruction register is combined with instruction decode.

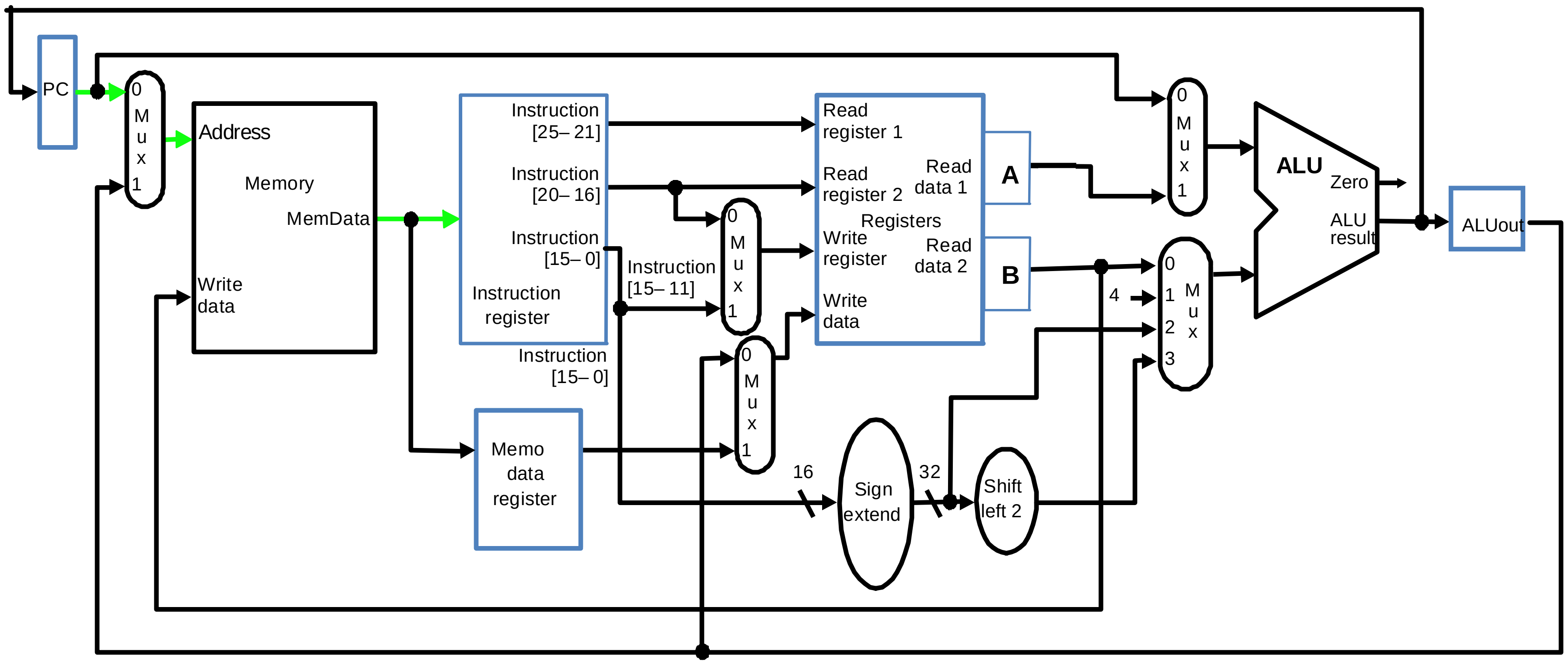


How does it work?

- R type instruction: R[rd] <= R[rs] op R[rt]
- Five cycles:
  - Instruction fetch
  - Instruction decode, read R[rs] and R[rt]
  - Execute instruction: R[rs] op R[rt]
  - Write result back to register R[rd]
  - Update PC: PC = PC + 4

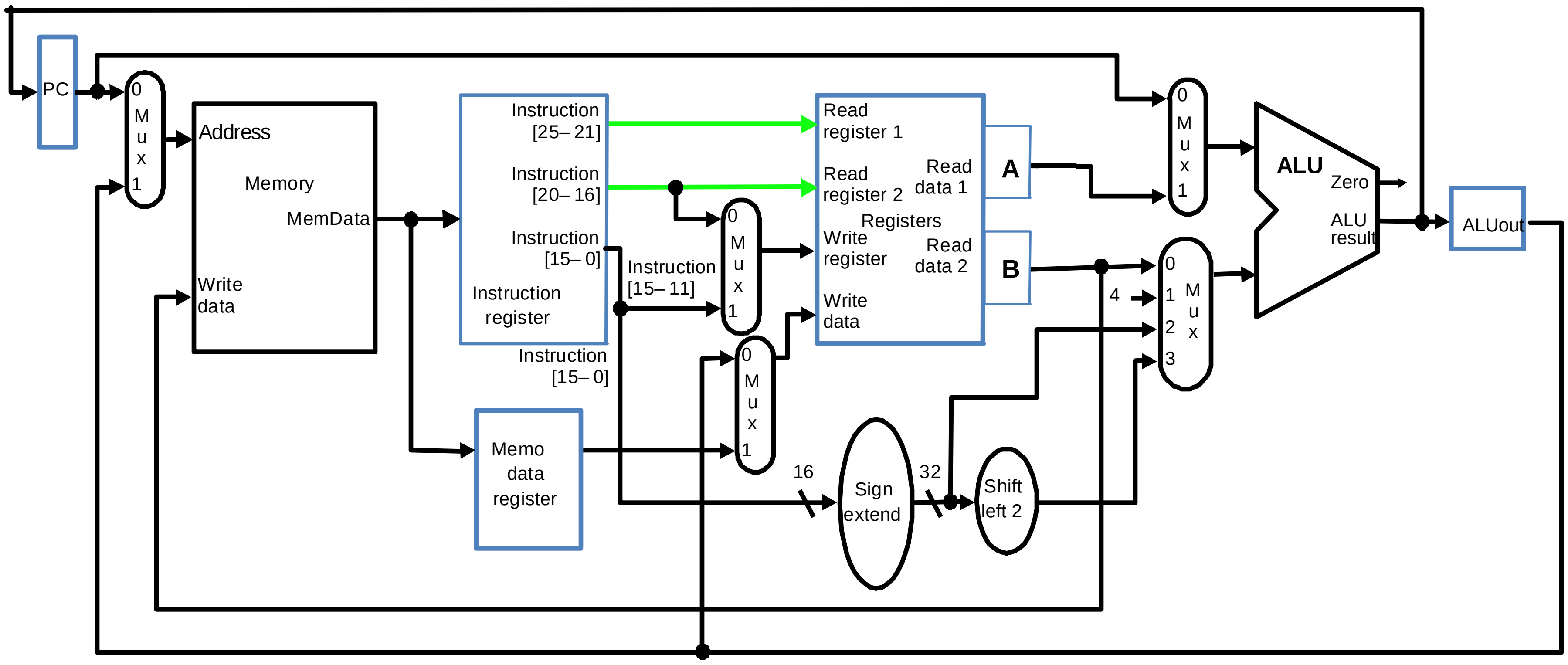
R type: 1st cycle

- Instruction fetch



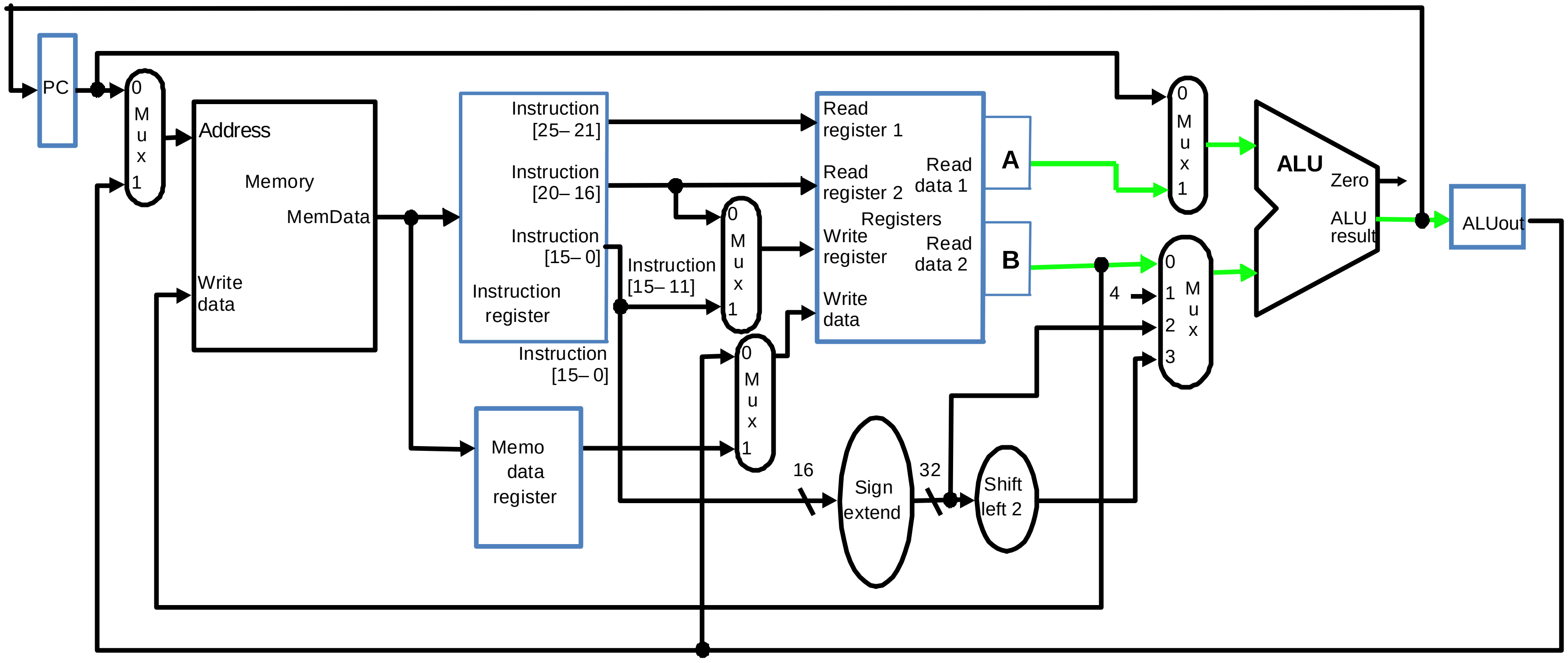
R type: 2nd cycle

- Instruction decode: read R[rs] and R[rt]



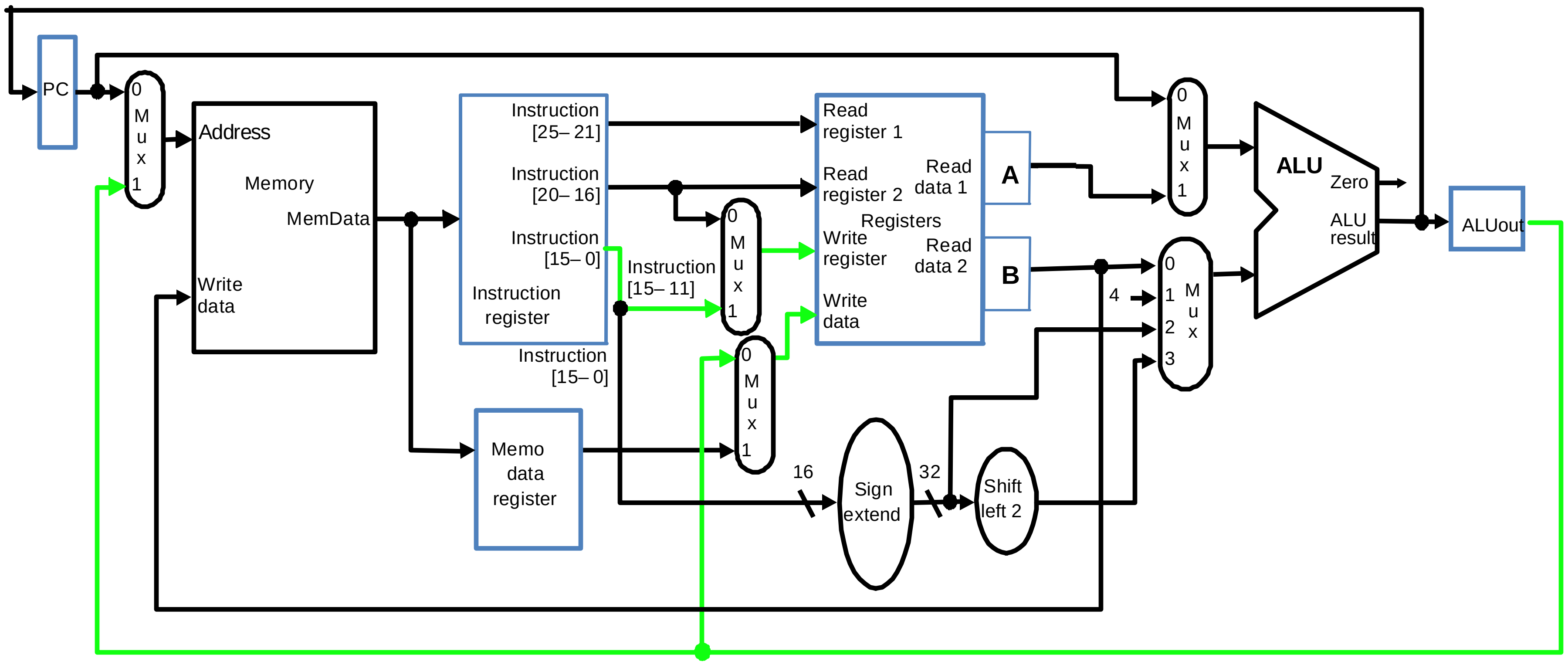
R type: 3rd cycle

- Execute instruction



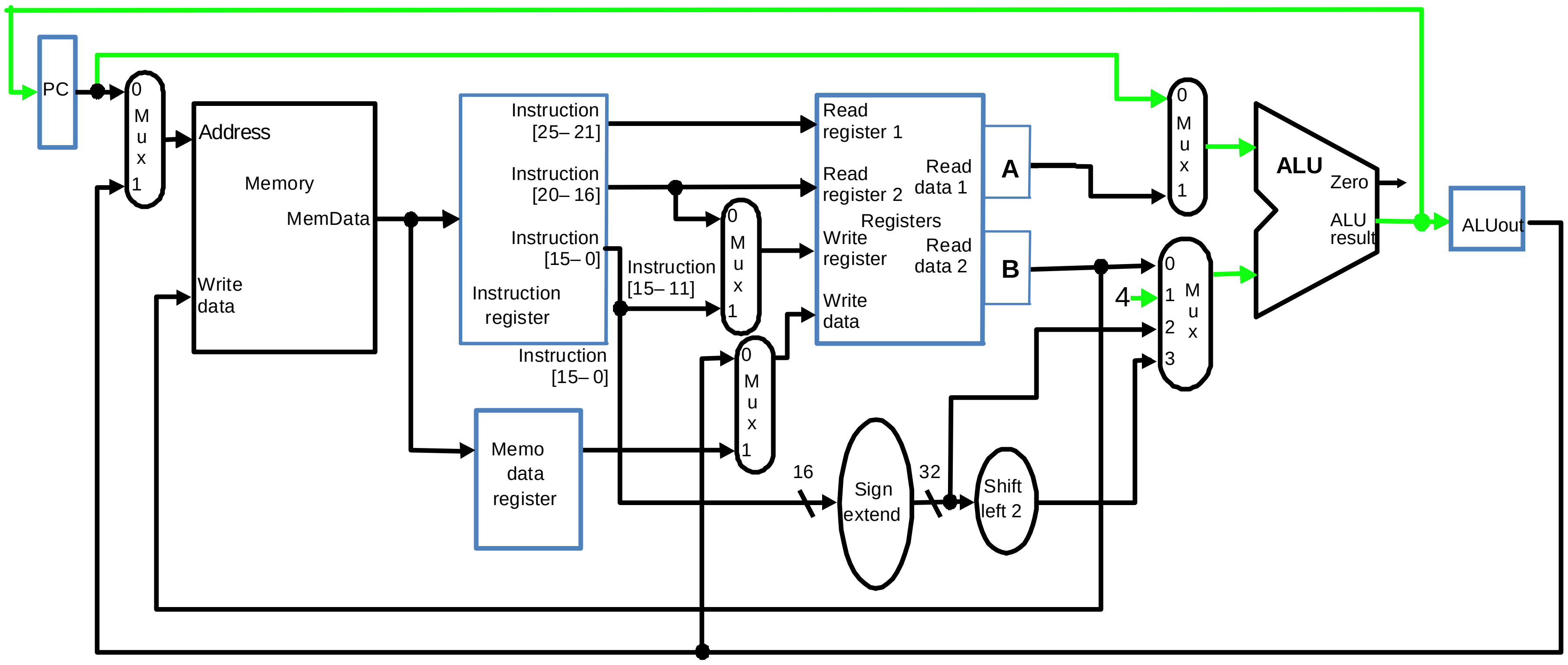
R type: 4th cycle

- Write result back to register R[rd]



R type: 5th cycle

- Update program counter: PC = PC + 4
- Instruction at (PC + 4) will be fetched next time

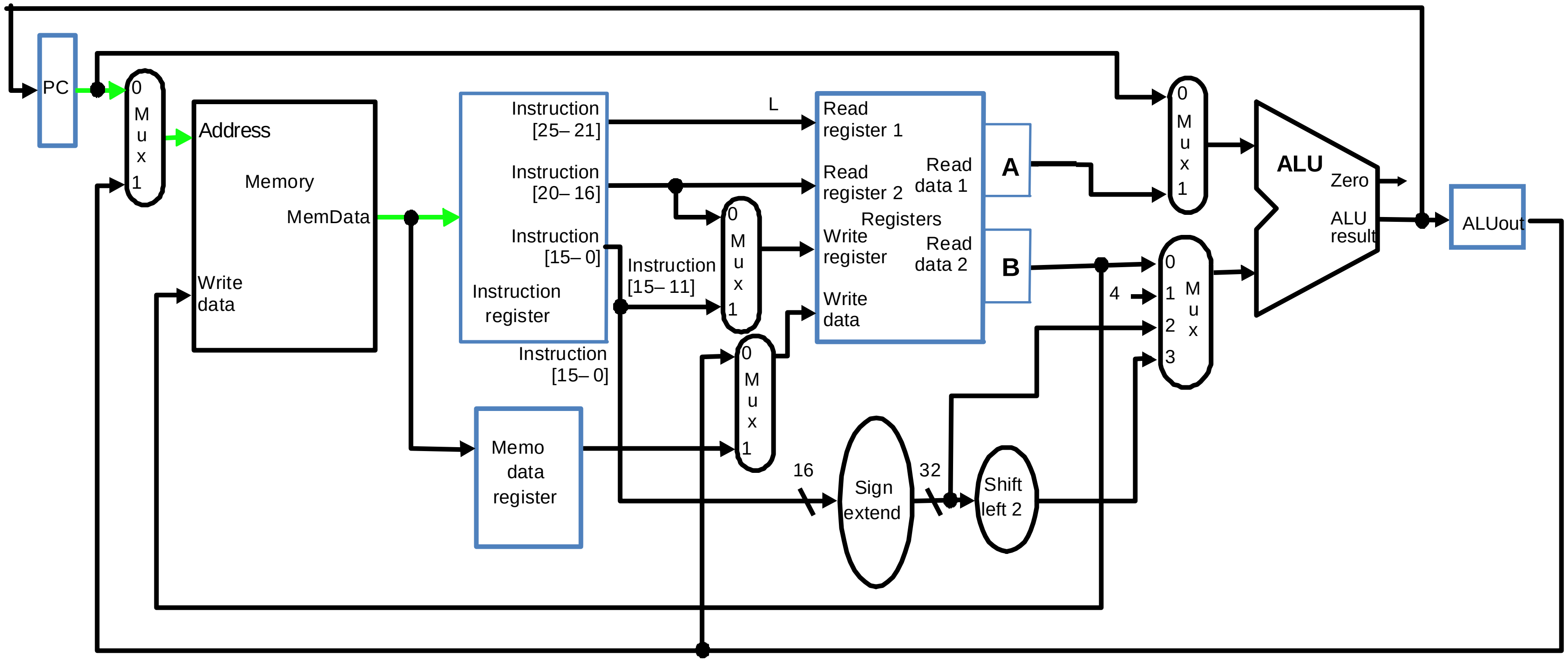


How does it work?

- I type instruction: R[rt] <= R[rs] op Imm
- Five cycles:
  - Instruction fetch
  - Instruction decode, read R[rs]
  - Execute instruction: R[rs] op Imm
  - Write result back to register R[rt]
  - Update PC: PC = PC + 4

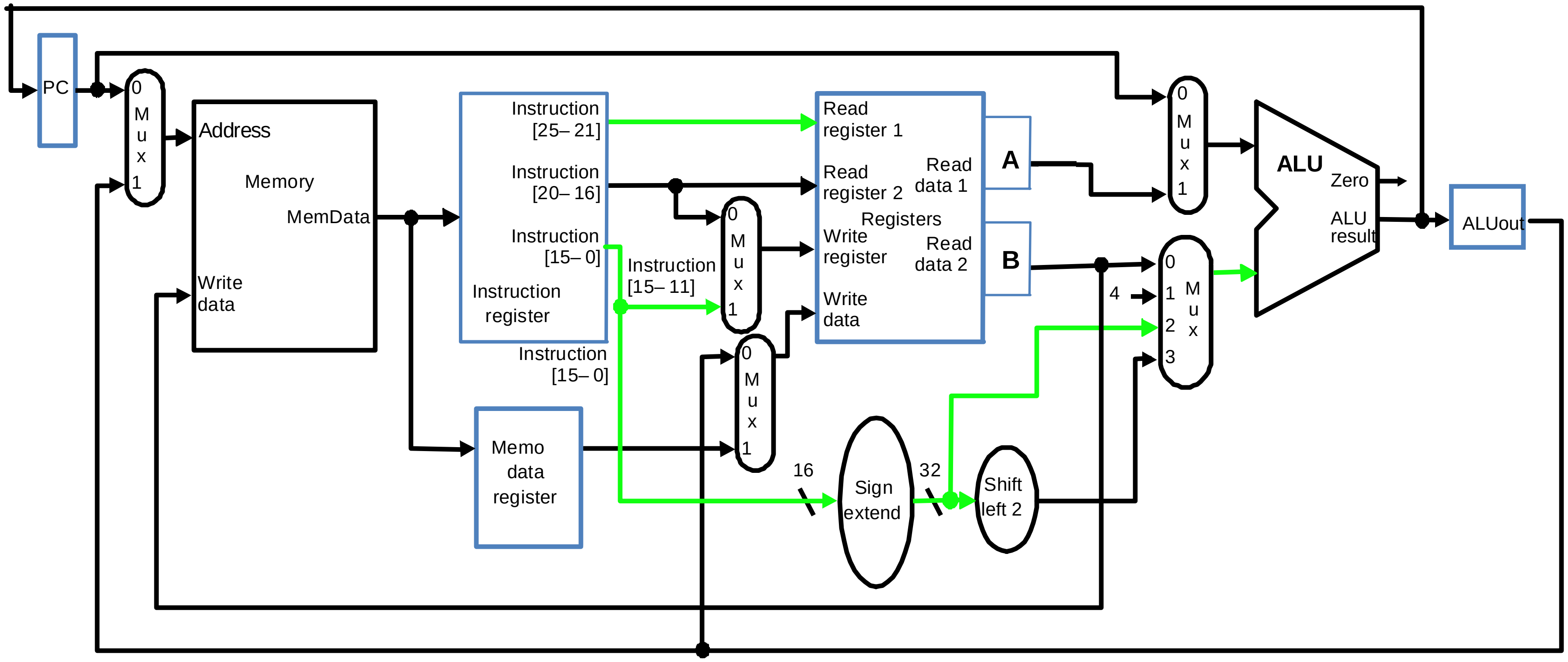
I type: 1st cycle

- Instruction fetch



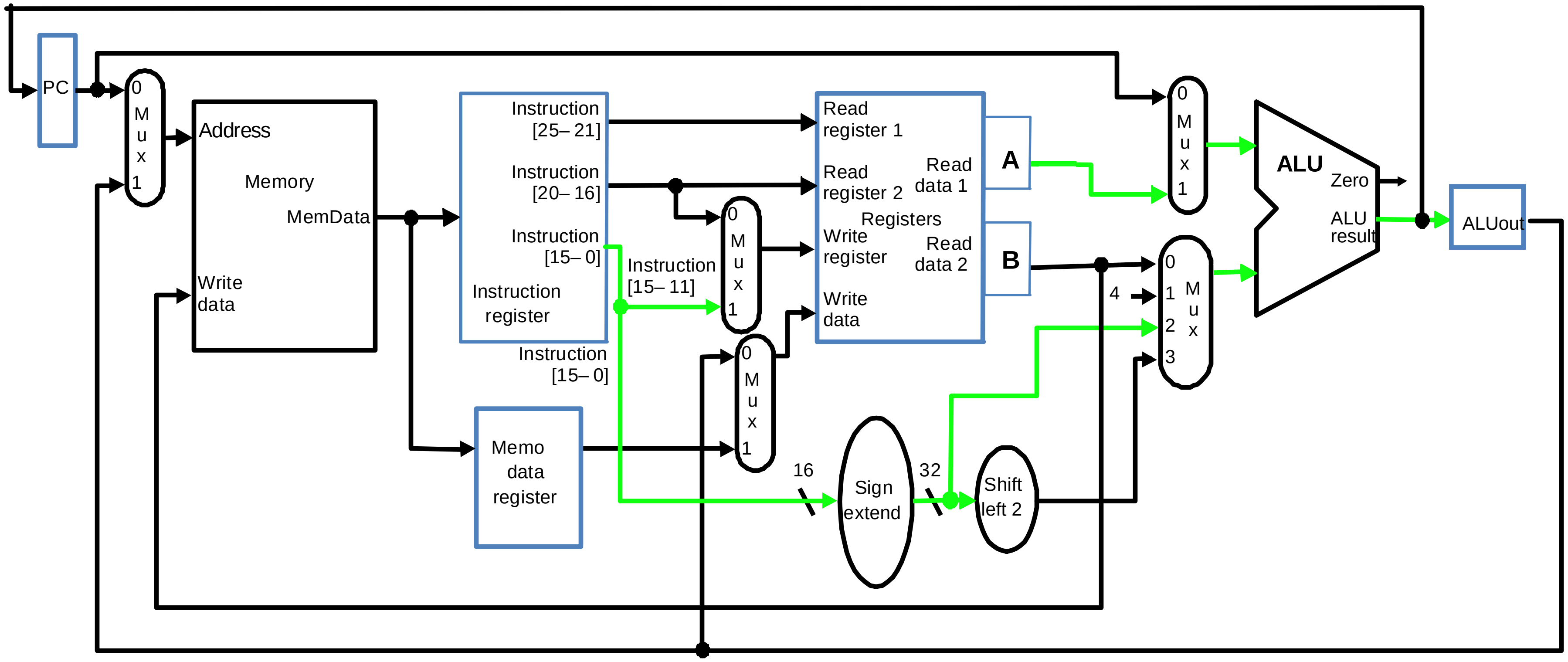
I type: 2nd cycle

- Instruction decode: read R[rs], decode Imm



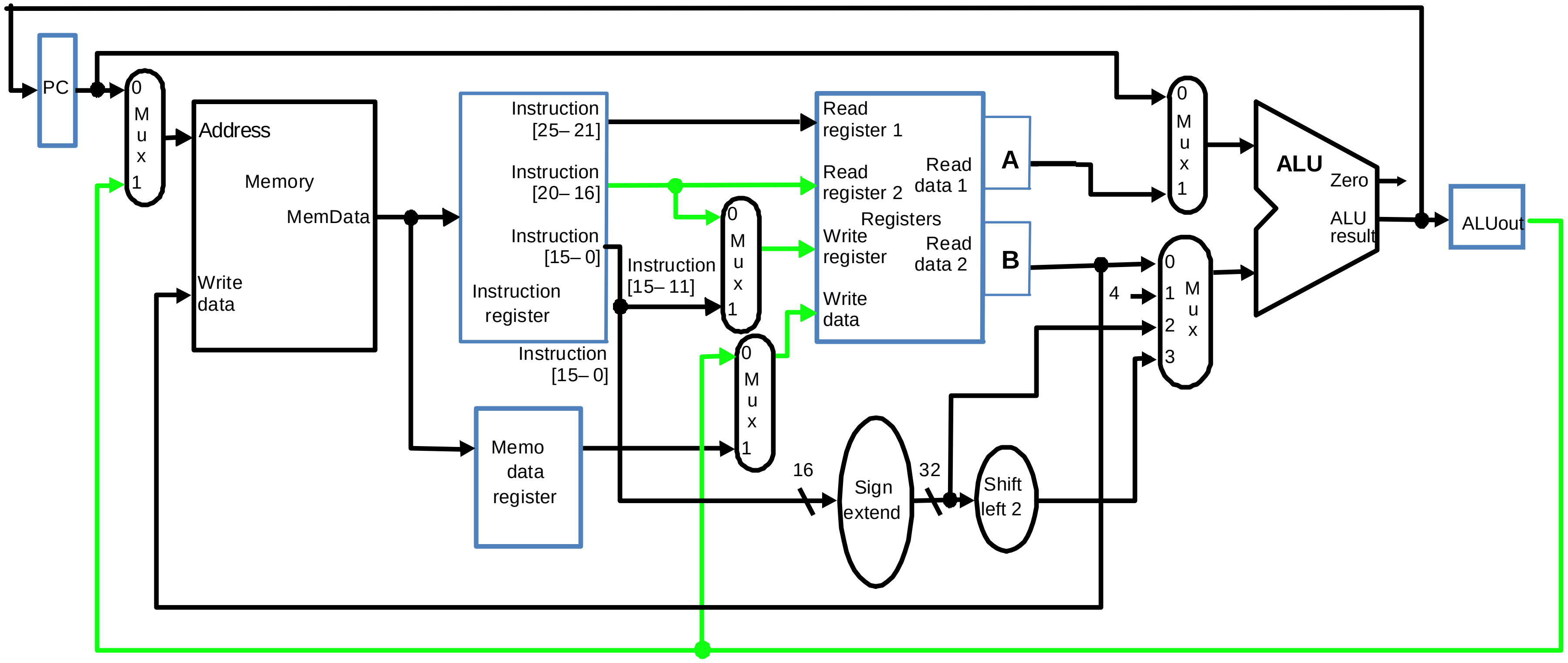
I type: 3rd cycle

- Execute instruction: R[rs] op Imm



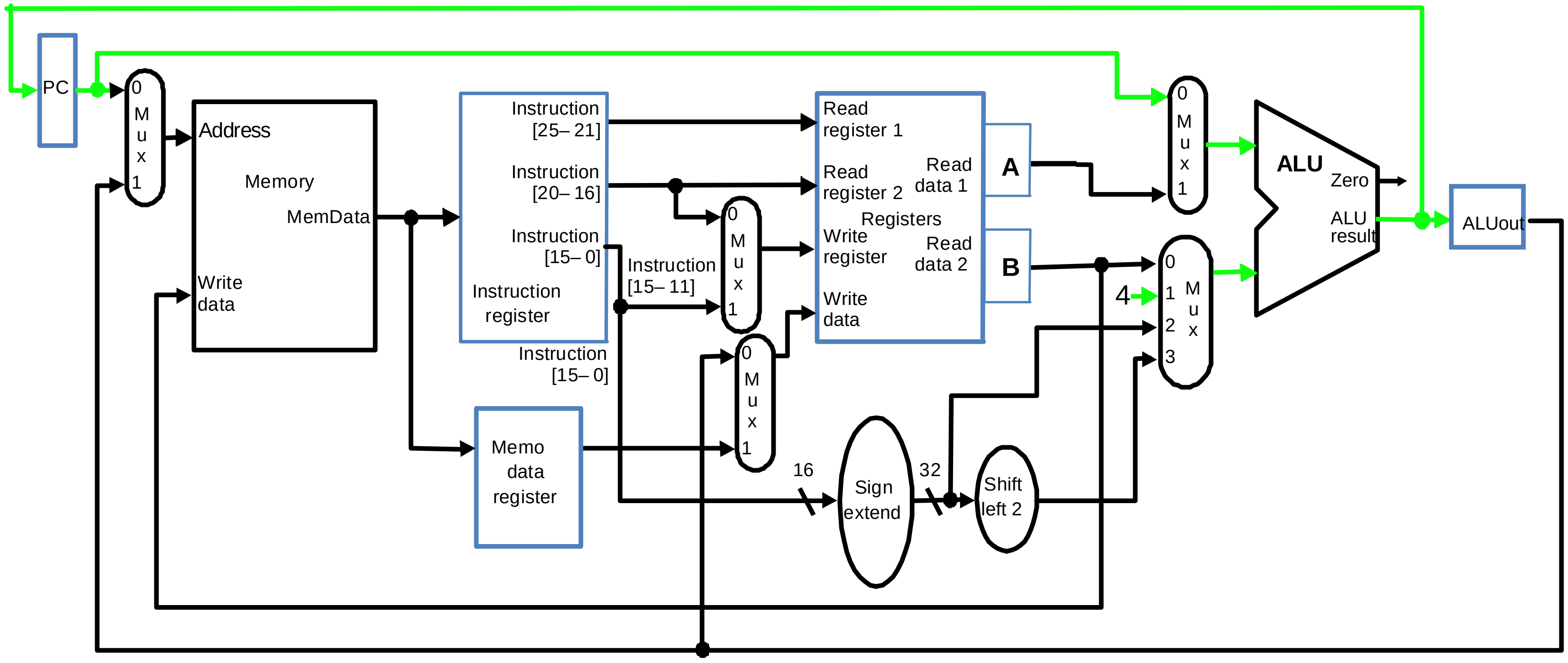
I type: 4th cycle

- Write result back to register R[rt]



I type: 5th cycle

- Update program counter: PC = PC + 4
- Instruction at (PC + 4) will be fetch next time

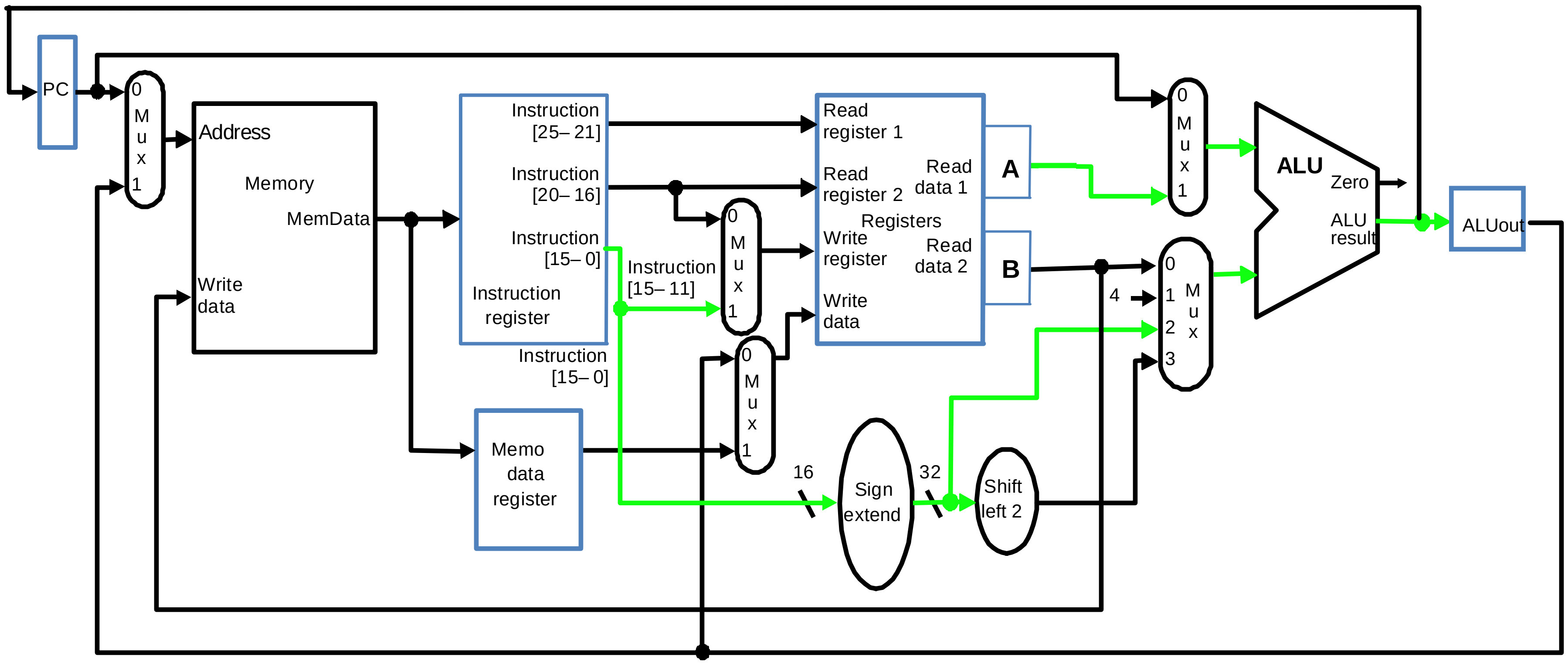


How does it work?

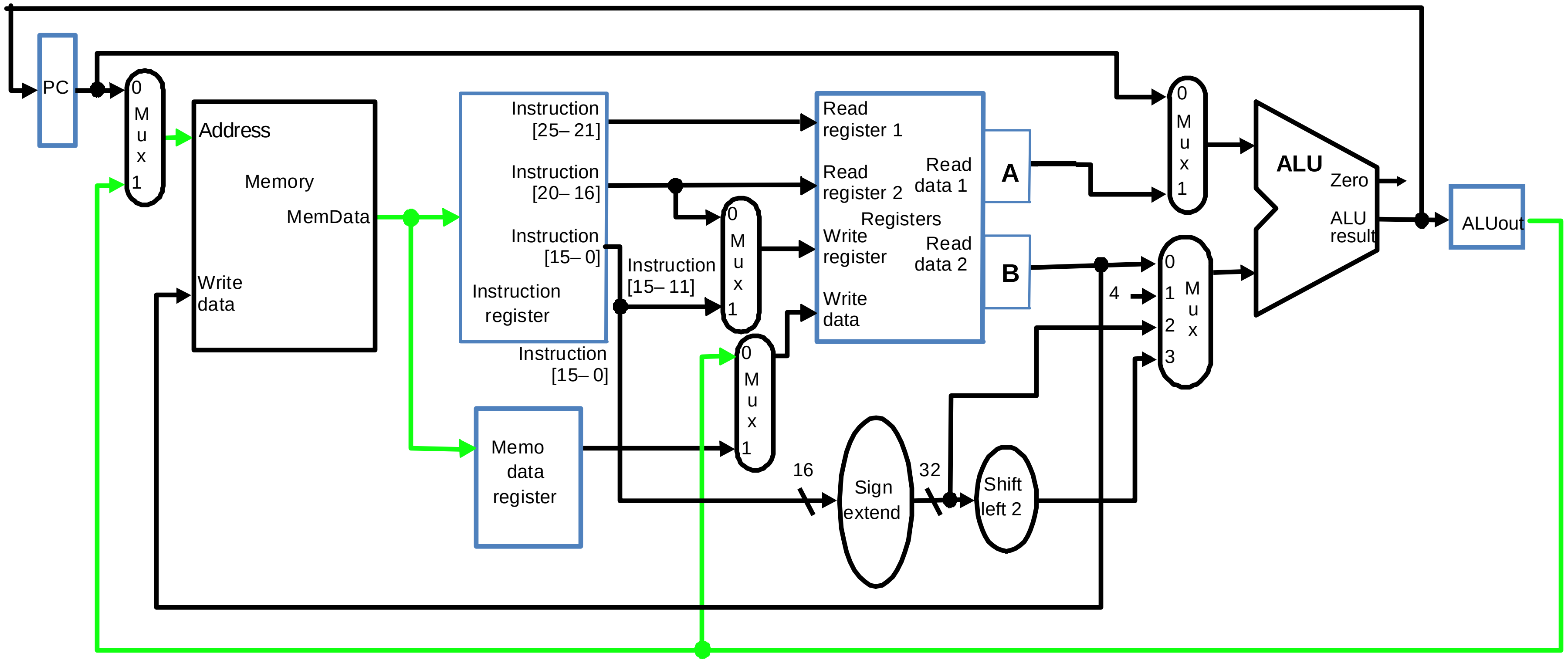
- Load word: lw R[rt], Imm[R[rs]]
- Major cycles:
  - Instruction fetch
  - Instruction decode: read R[rs]
  - Calculate memory address: R[rs] + Imm
  - Load data from memory address: R[rs] + Imm
  - Write memory data to register R[rt]
  - Update PC: PC = PC + 4

Calculate memory address

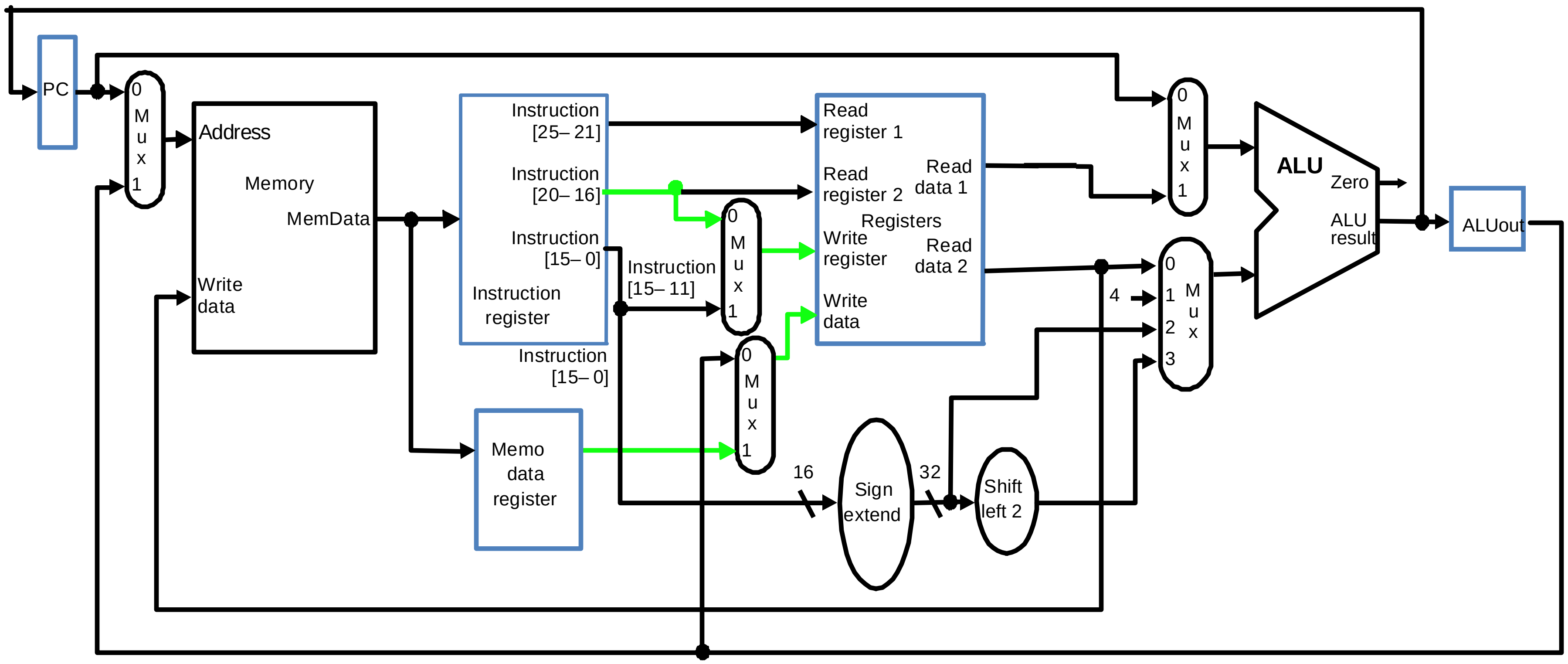
- address = R[rs] + Imm



Load data from memory



Write memory data to R[rt]

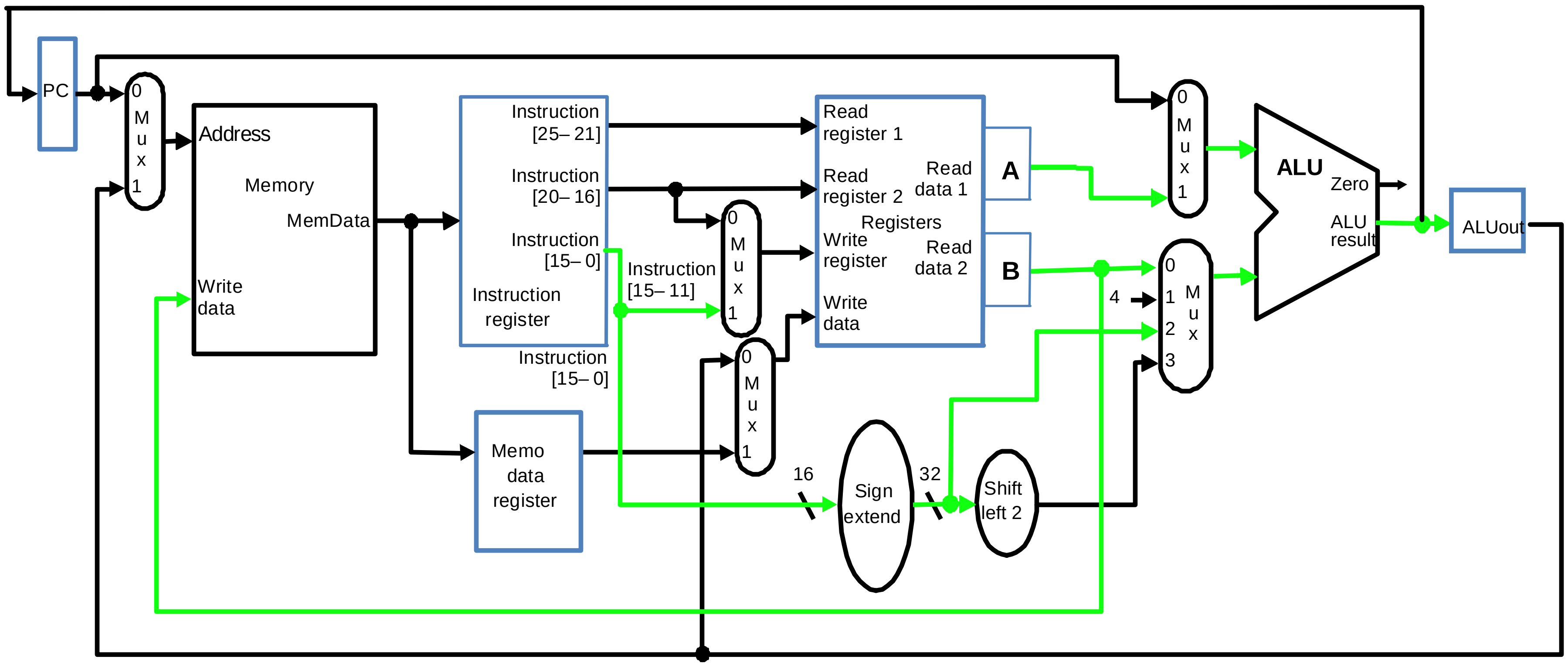


How does it work?

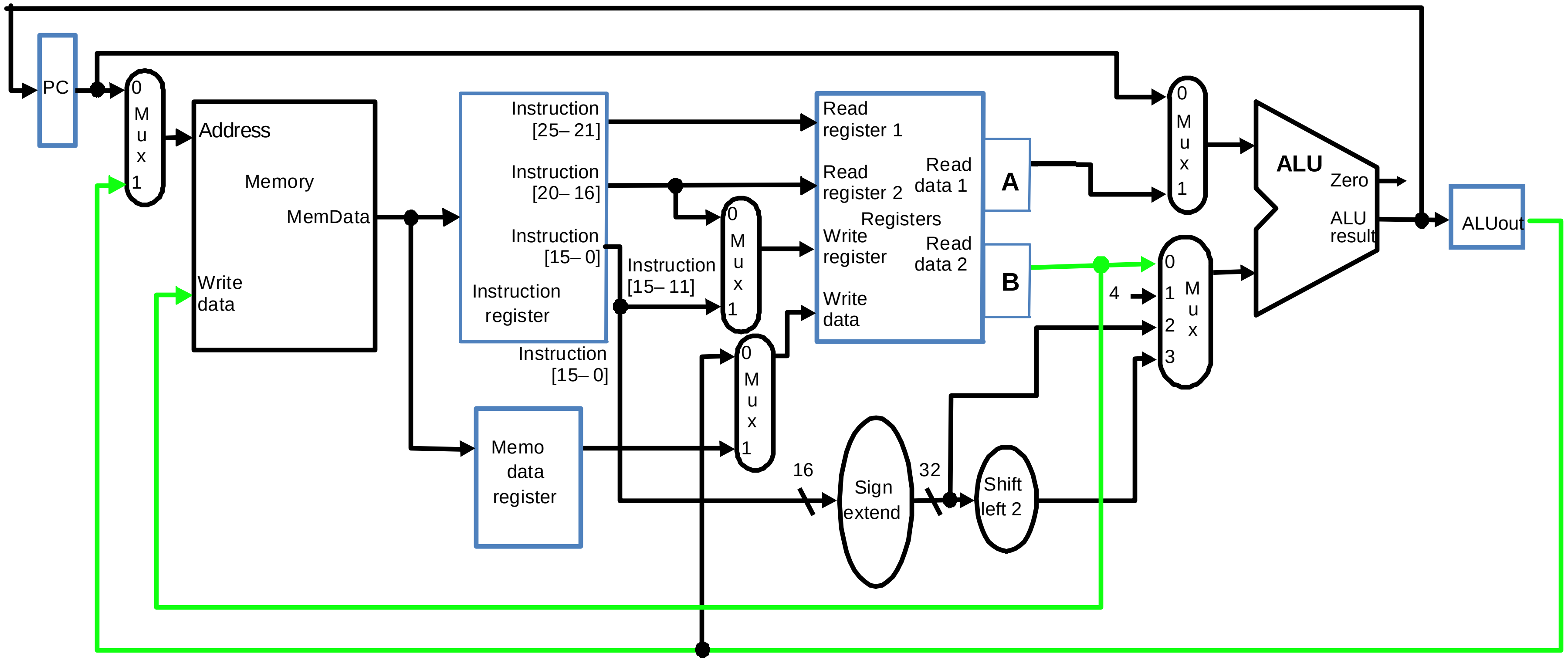
- Store word: sw R[rt], Imm[R[rs]]
- Major cycles:
  - Instruction fetch
  - Instruction decode: read R[rs]
  - Calculate memory address: R[rs] + Imm
  - Store data from R[rt] to memory
  - Update PC: PC = PC + 4

Calculate memory address

- address = R[rs] + Imm
- ALUout is used to store the calculated address which will be used in the store stage later.
- Read R[rt]



Store data from register to memory

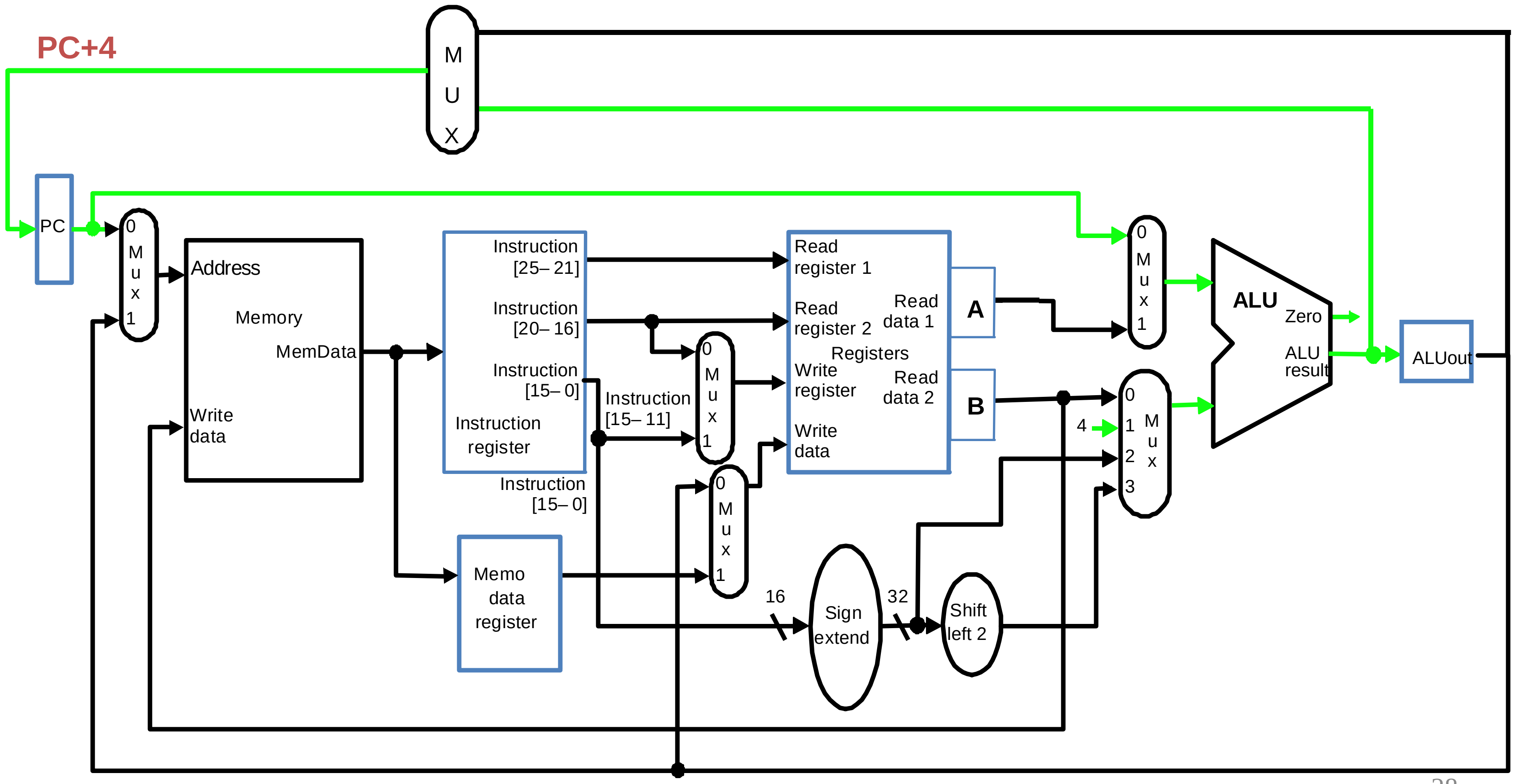


How does it work?

- Branch instruction: beq R[rt], R[rs], Imm
- Three major cycles
  - Instruction fetch
  - Instruction decode: read R[rs] and R[rt]
  - Calculate (PC + 4)
  - Calculate (PC + 4 + Imm $\times$ 4)
  - Calculate branch condition: e.g. R[rt] == R[rs] ??
  - Update program counter as (PC + 4 + Imm $\times$ 4) if branch condition is true

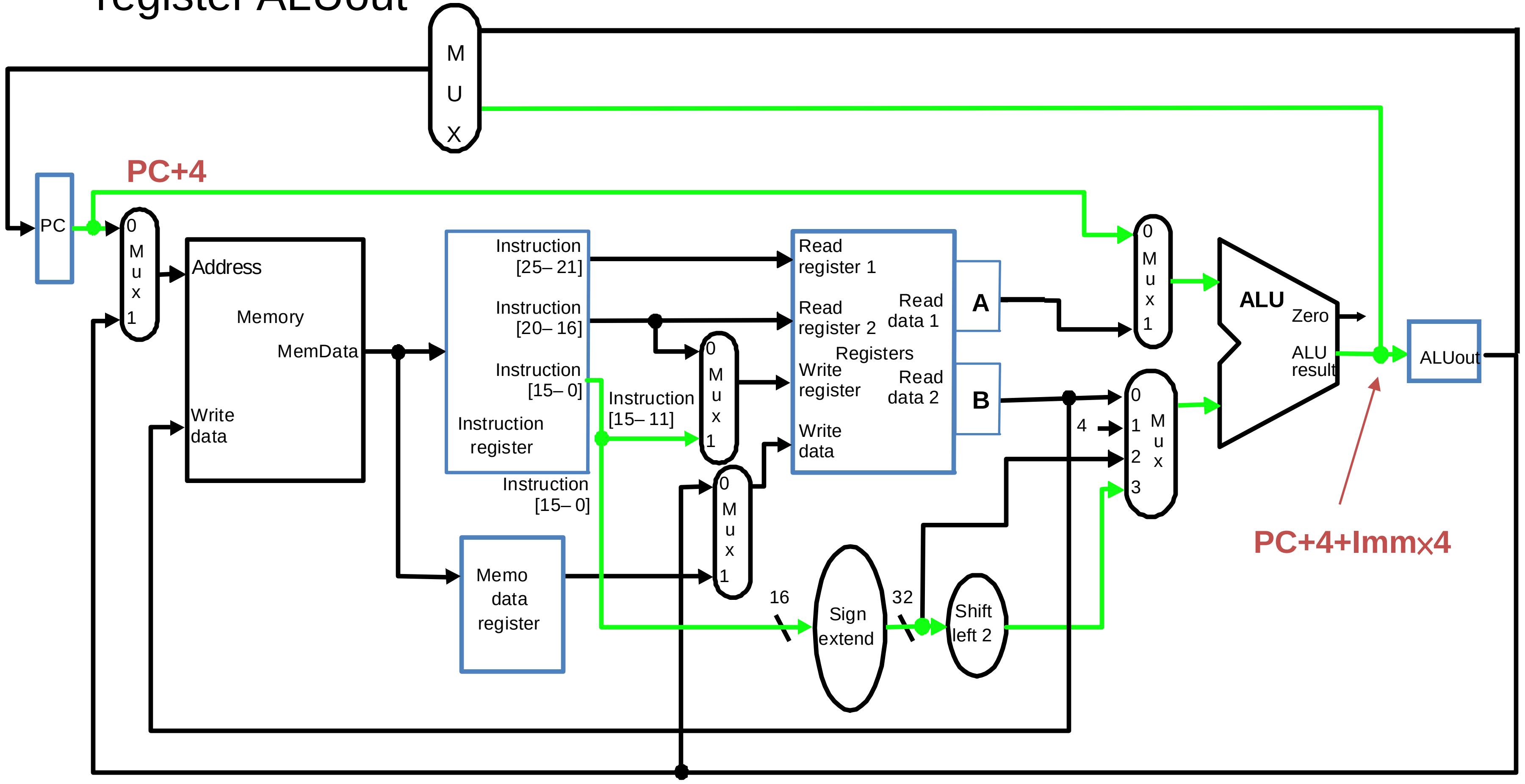
Calculate PC+4

- Add a multiplexer
- Calculate PC + 4, and write (PC+4) to register PC



Calculate branch address

- Branch address = PC + 4 + Imm $\times$ 4
- Don’t write branch address to register PC, write branch address to register ALUout

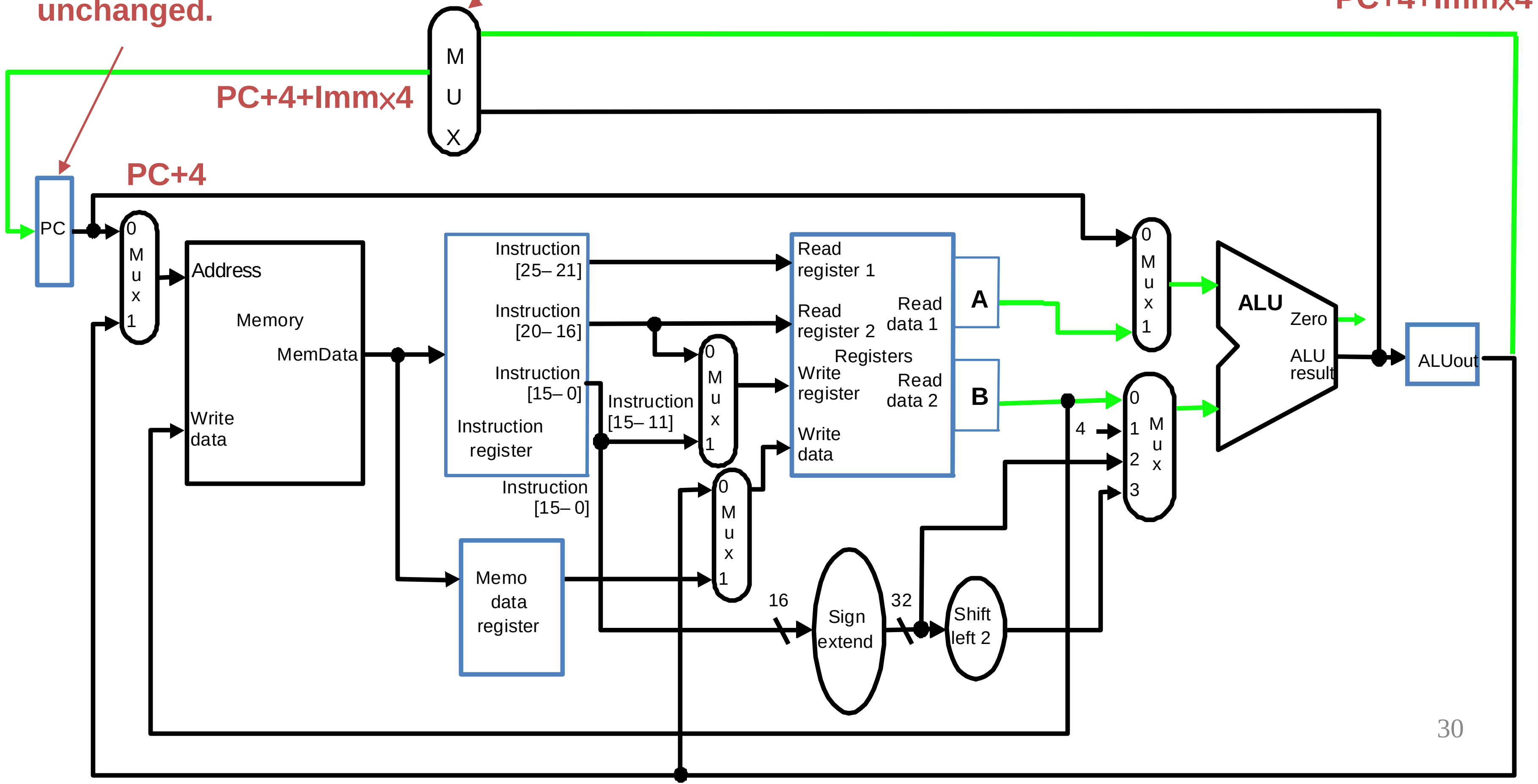


Calculate branch condition

- Calculate R[rs] – R[rt]
- Update register PC based on zero flag

Update PC if zero==1. Otherwise, PC remains unchanged.

Select (PC+4+Imm×4) if zero==1.

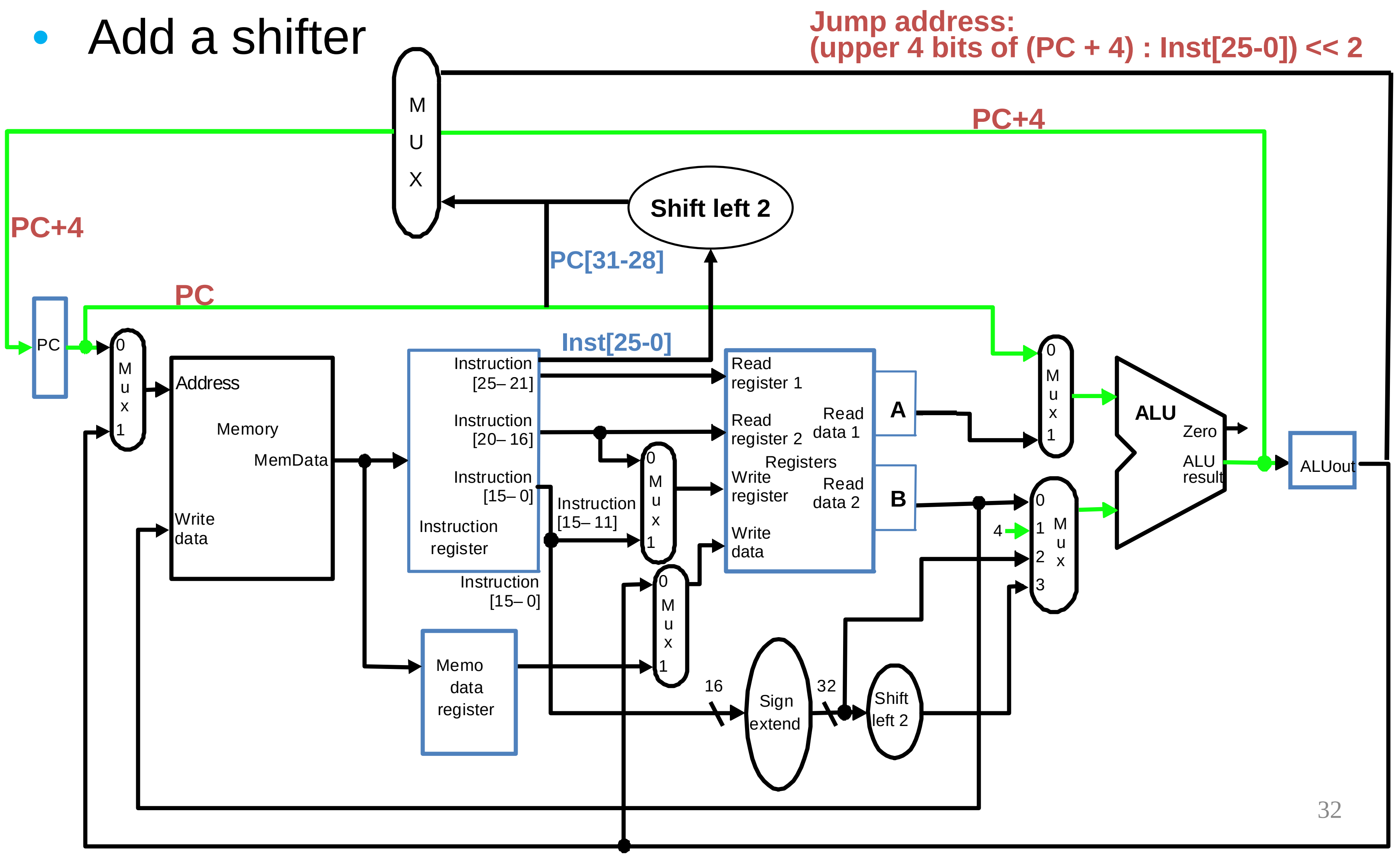


How does it work?

- Jump instruction: J Label
- One major cycle
  - Instruction fetch
  - Instruction decode
  - Calculate target address (2 steps)

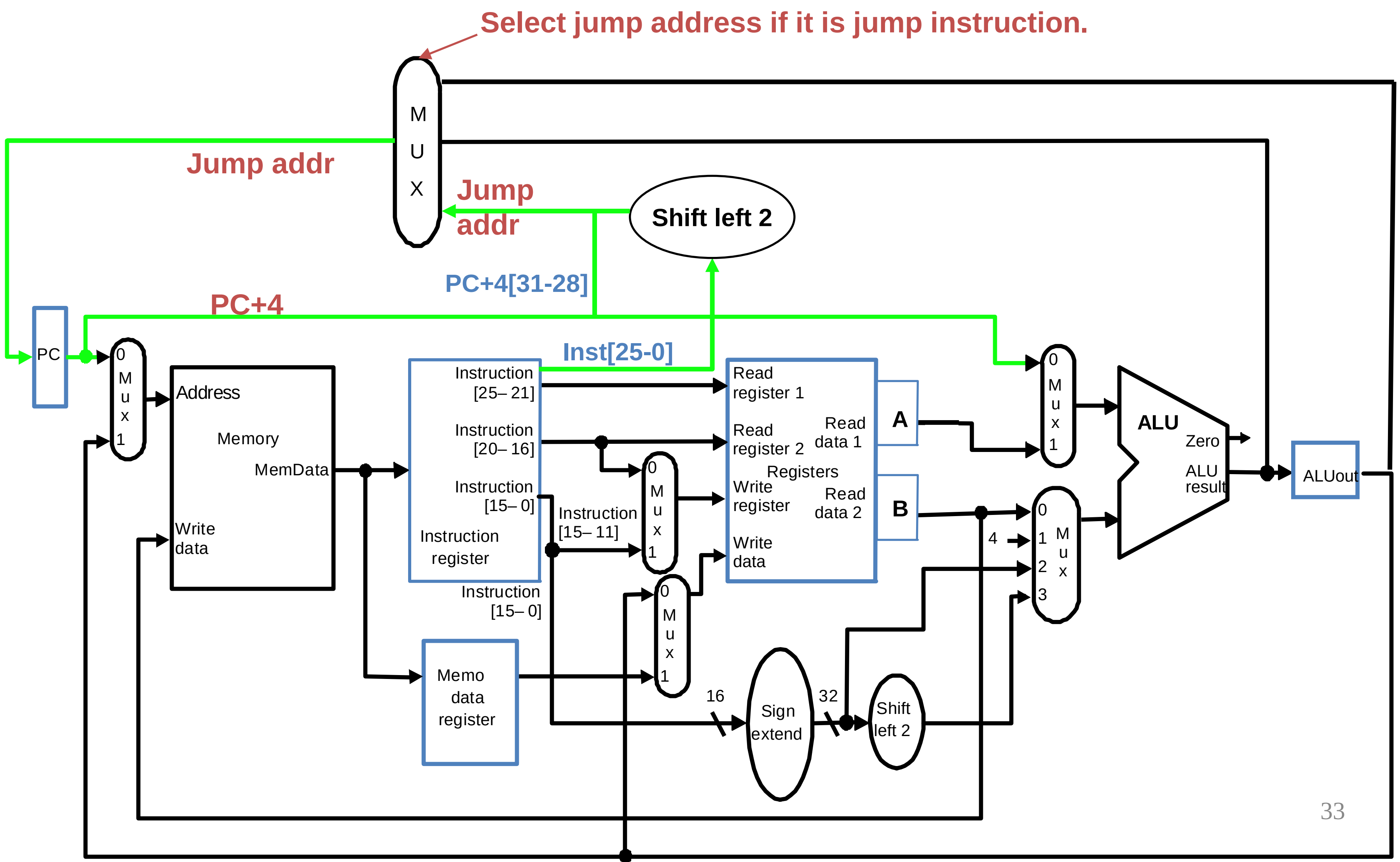
Calculate (PC + 4)

- Extend the multiplexer to 3 inputs
- Add a shifter



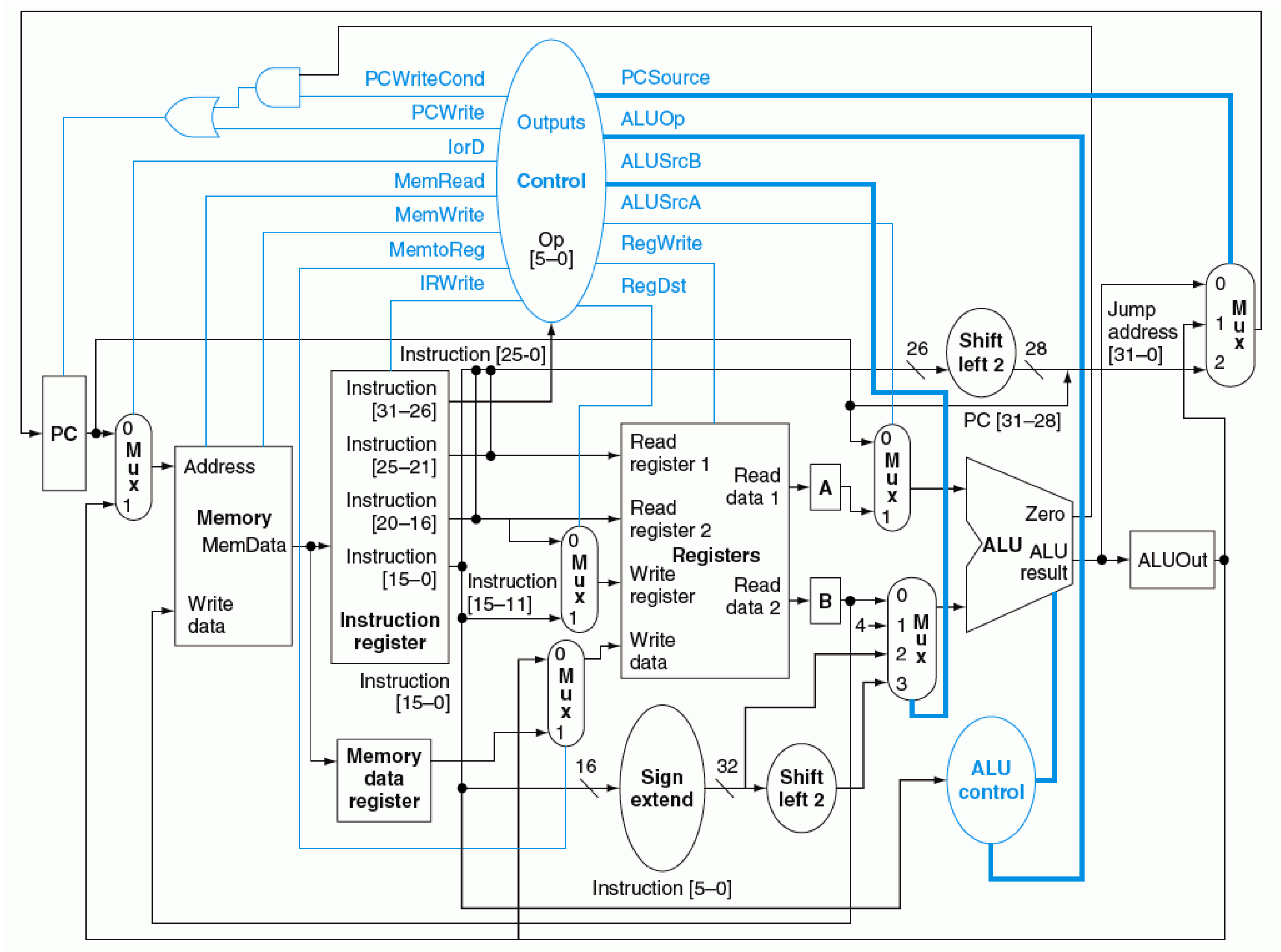
Calculate target address

- Calculate Jump address: (upper 4 bits of (PC + 4) : Inst[25-0]) << 2



Multi-Cycle datapath with control signals

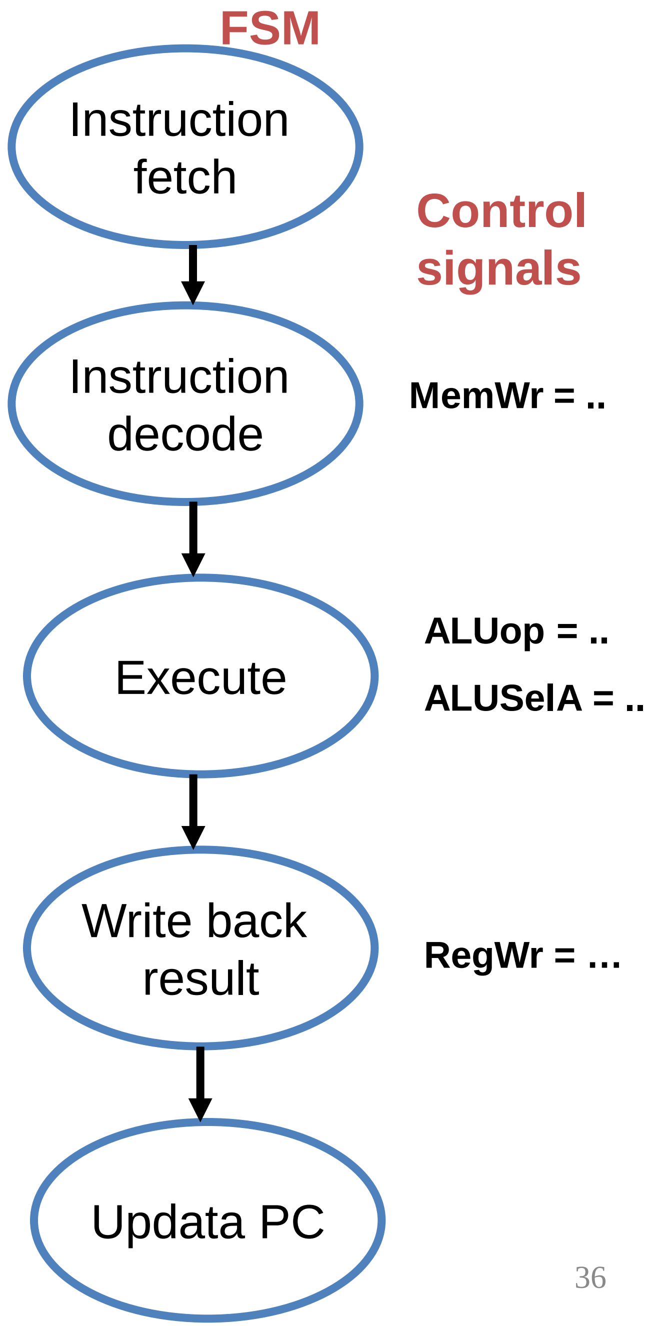
- Minimum Hardware: 1 memory, 1 ALU (instruction and data memories are combined, no adder to calculate branch address)



Designing control logic

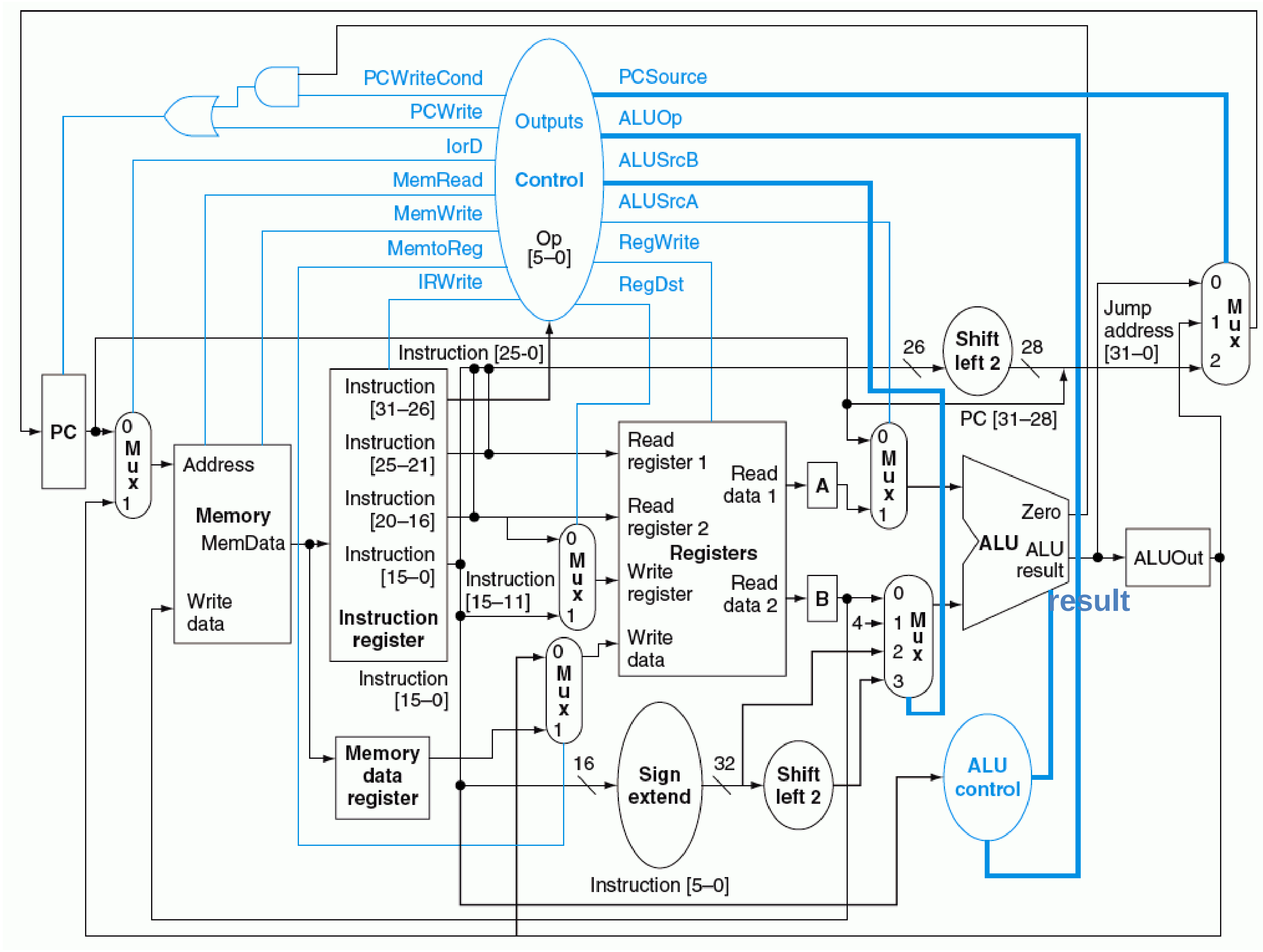
Steps to execute instructions: R-type

- R-type, e.g. add $t0, $t1, $t2
  - Instruction fetch
  - Instruction decode, read R[rs] and R[rt]
  - Execute instruction: R[rs] op R[rt]
  - Write result back to register R[rd]
  - Update PC: PC = PC + 4
- Can we design a finite state machine to represent these states? And generate different control signals at different states to control the operation of the datapath?
- E.g. design a FSM with 5 states: IF, ID, EX, WB, PC. Each state outputs appropriate control signals to control the datapath to finish adequate operation in a certain state.



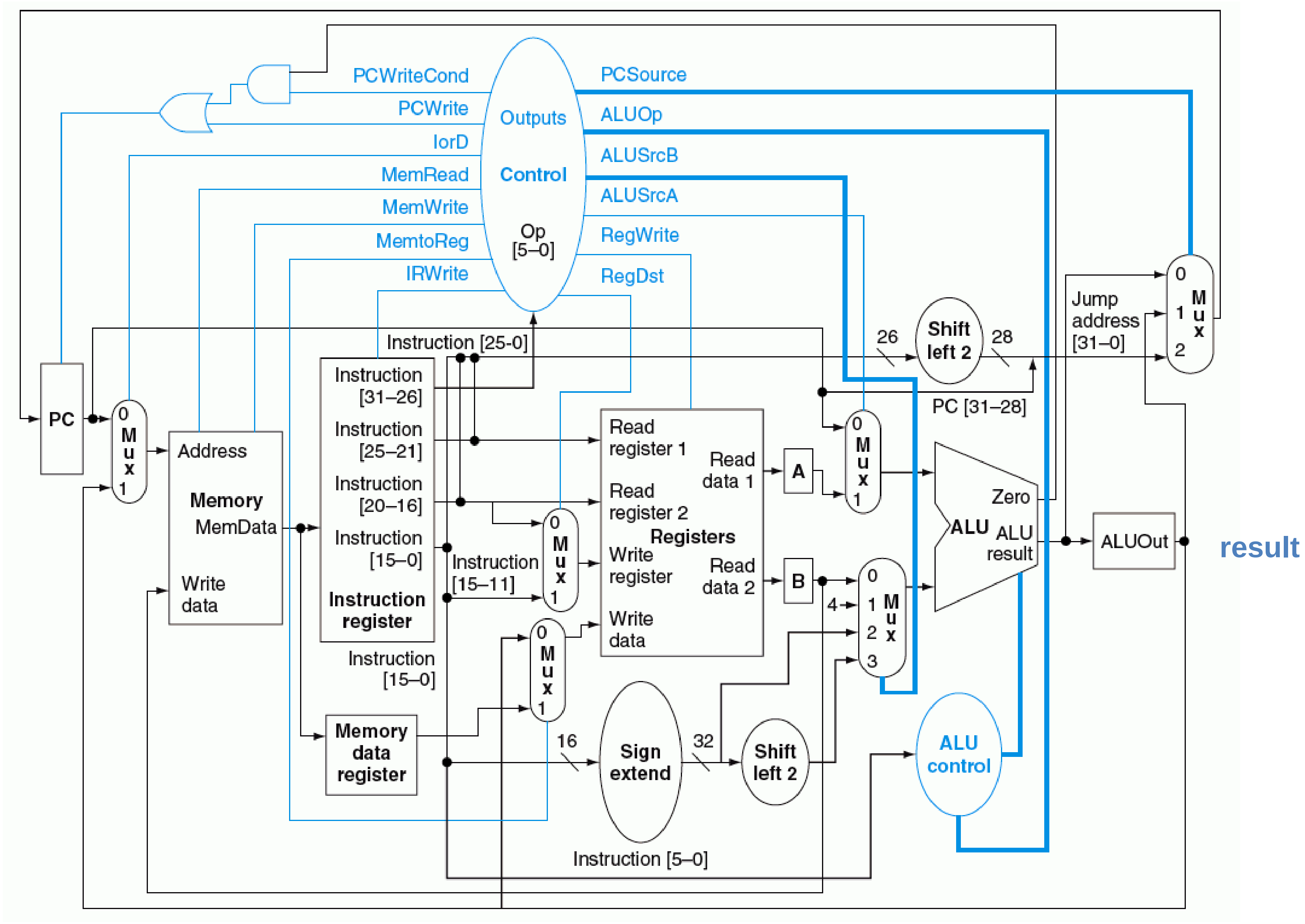
Example: control signals for ($t1 + $t2) at execute state

- ALUOp = add, ALUSrcA = 1, ALUSrcB = 0, MemtoReg = x, RegWrite = 0



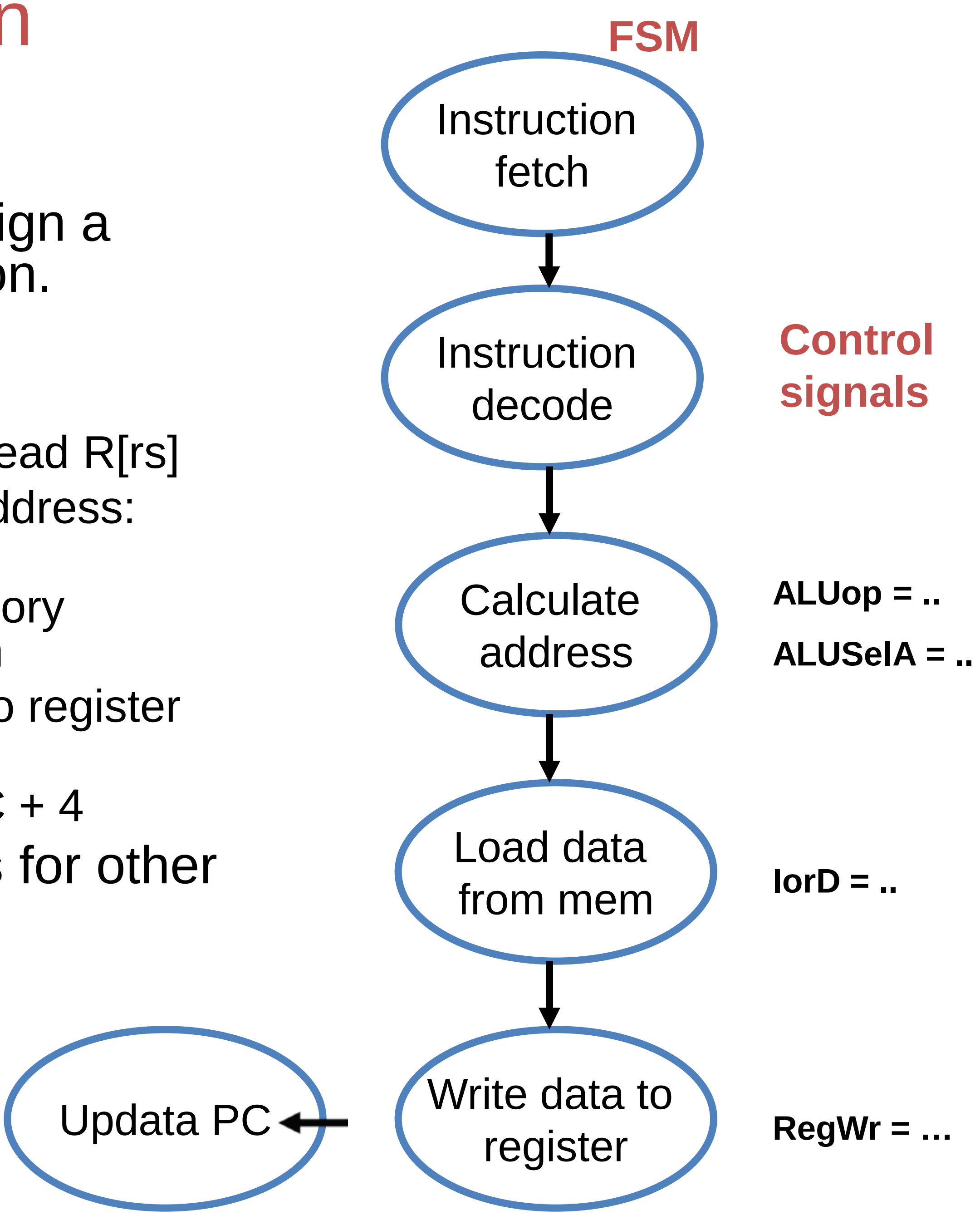
Example: control signals for ($t1 + $t2) at write back state

- ALUOp = x, ALUSrcA = x, ALUSrcB = x, MemtoReg = 0, RegWrite = 1



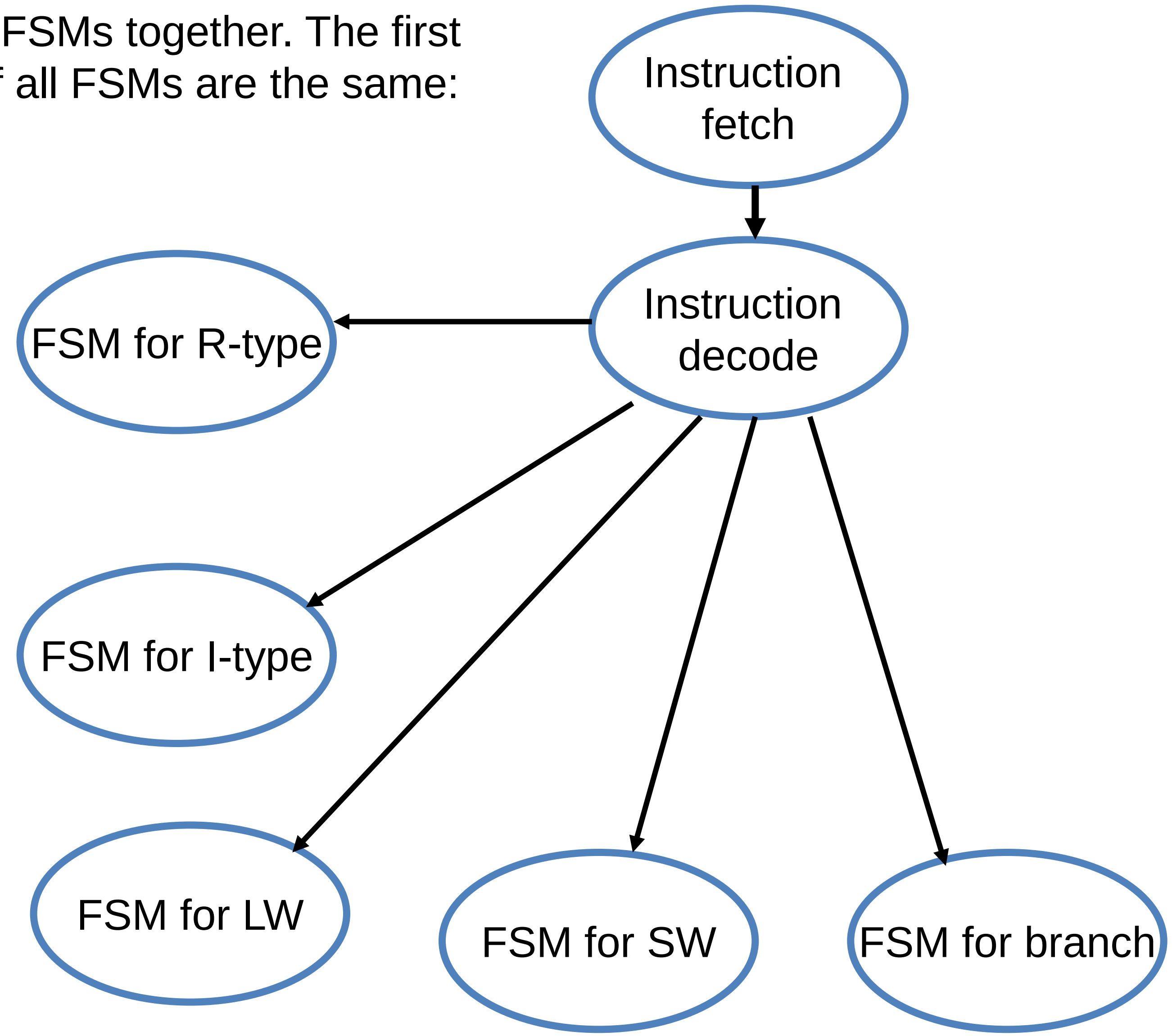
Steps involved in instructions: LW

- Similarly, we can design a FSM for LW instruction.
- Load word
  - Instruction fetch
  - Instruction decode: read R[rs]
  - Calculate memory address: R[rs] + Imm
  - Load data from memory address: R[rs] + Imm
  - Write memory data to register R[rt]
  - Update PC: PC = PC + 4
- We can design FSMs for other instructions.

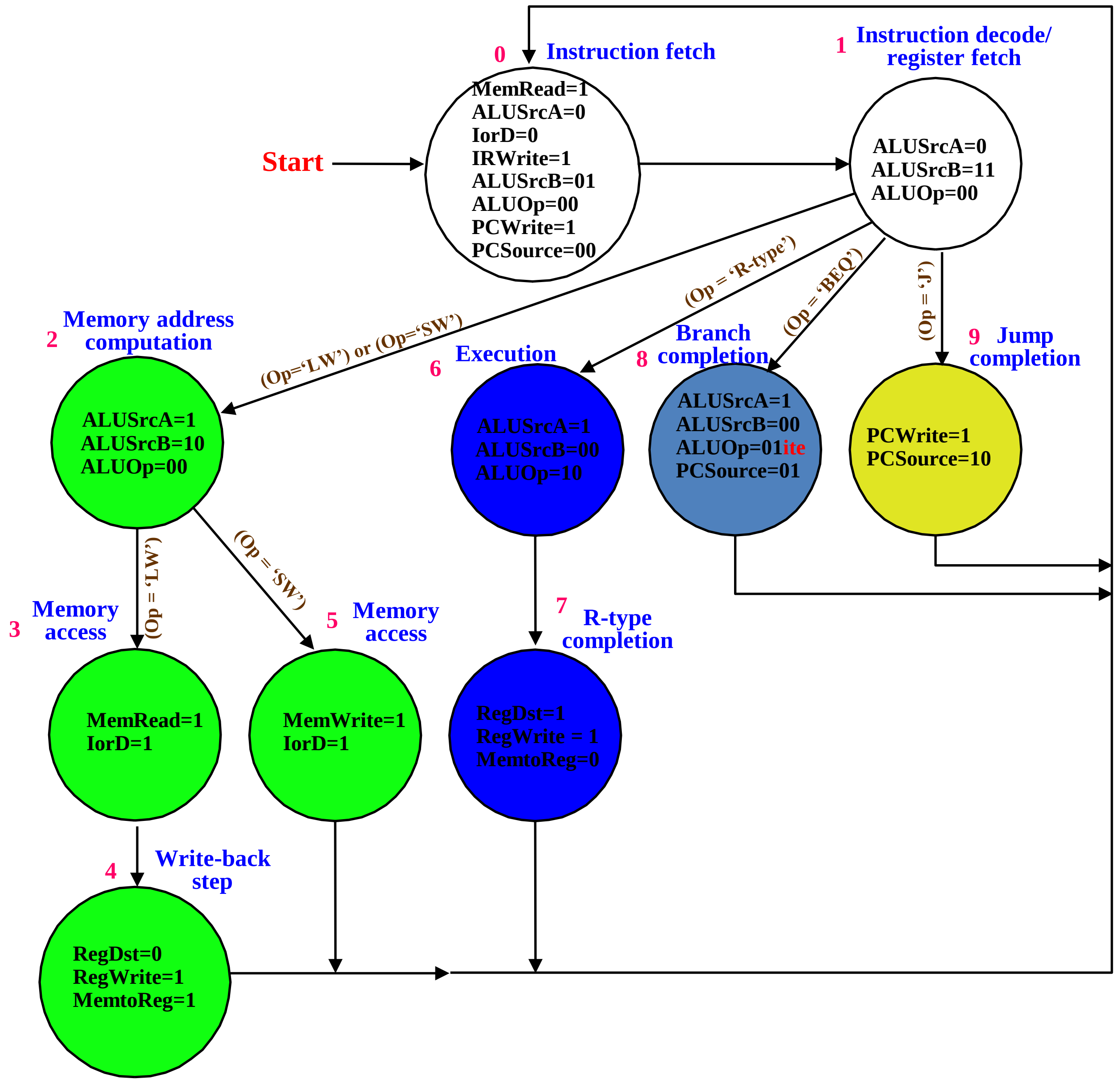


Control FSM for the multi-cycle processor

- Combine all FSMs together. The first two states of all FSMs are the same: IF, ID.

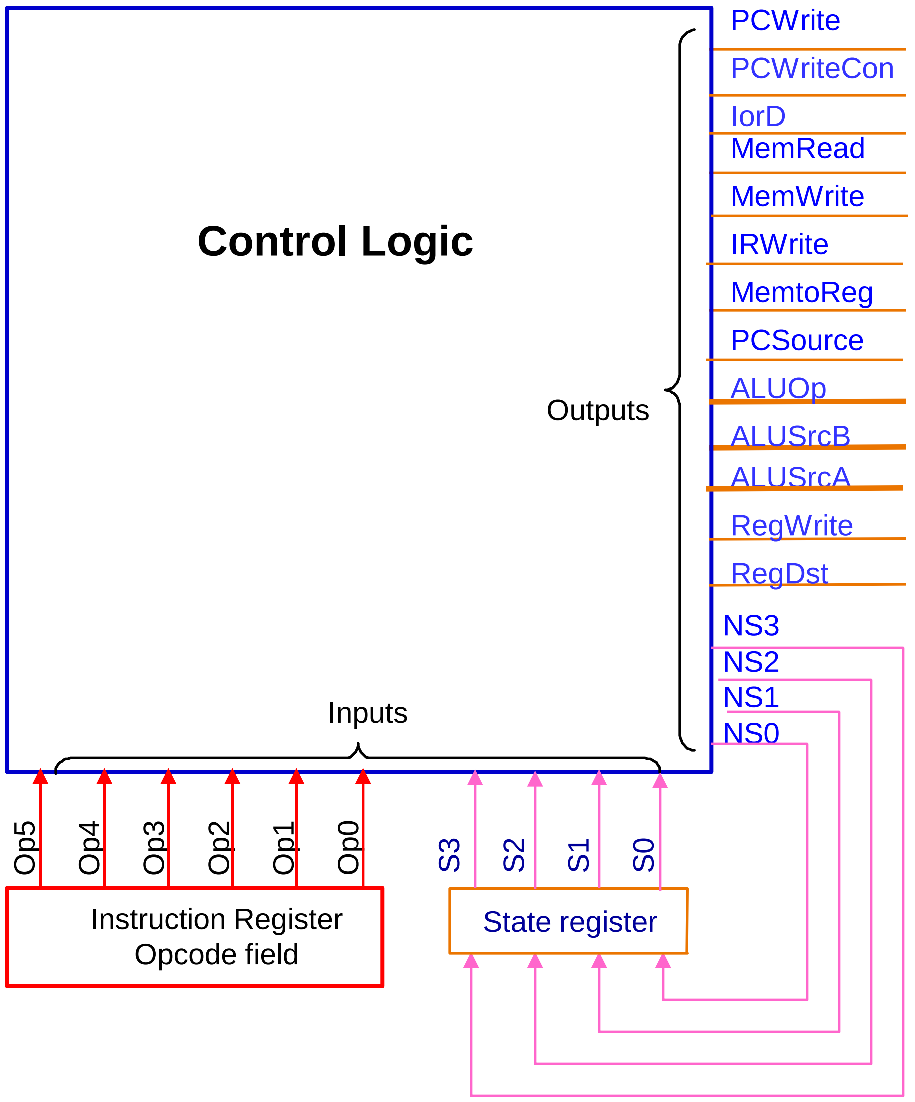


Control FSM for the multi-cycle processor: a detailed view



Implement the FSM in hardware

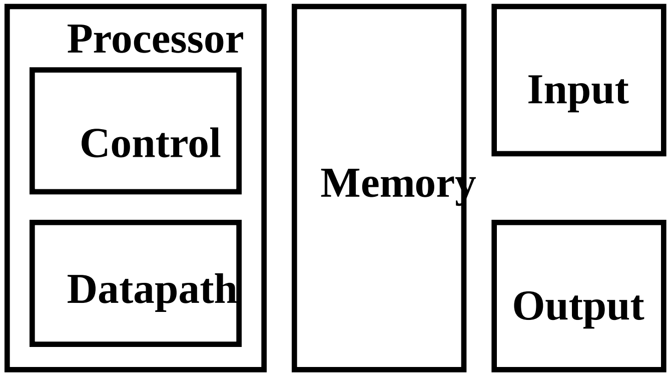
- State changes depends on opcode and current state.



Single cycle vs Multi-cycle processor

- Example
  - A single cycle processor has a cycle time of 800ps
  - A multi-cycle processor: $CPI_{R\text{-}type}=4$, $CPI_{lw}=5$, $CPI_{beq}=3$. Cycle time = 200ps
  - A program has 40% R-type, 20% lw, 40% beq. Totally 10000 inst.
  - Single processor time = $10000 \times 800 \times 1$ = 8M ps
  - Average CPI for multi-cycle processor = $4\times40\%+5\times20\%+3\times40\%=3.8$
  - Multi-cycle processor time = $10000 \times 200 \times 3.8$ = 7.6M ps
  - Multi-cycle processor is better in this example, how about 40% of lw, and 20% of beq?
    - $4\times40\%+5\times40\%+3\times20\%=4.2$
    - $10000 \times 200 \times 4.2$ = 8.4M ps
- Which processor is better?
  - Depends on the instruction mix of programs $\rightarrow$ program dependent.
  - Execution time = Instruction count $\times$ CPI $\times$ cycle time

Summary



- Two components of processors.
  - Datapath and control.
- Design multi-cycle processor.
  - Partition datapath of single cycle processor (break an instruction execution into multiple steps).
  - Add registers into the partitioned datapath.
  - Design FSM to control the datapath.
  - Execute each step in one cycle, require multiple cycles to execute one instruction.
- Advantages and disadvantages
  - Single cycle: CPI = 1, long cycle time.
  - Multi-cycle: shorter cycle time as each cycle just execute one step, different instructions have different CPI. As a result, simple instructions require less cycles (less time) to execute.
- Performance comparison
  - Program dependent.
  - Execution time = Instruction count $\times$ CPI $\times$ cycle time