CSE 220: System Fundamentals I Unit 15c: MIPS Architecture: Multicycle Processors III

Topics

- Multicycle Processors
  - Approach
  - State Elements
  - Fetch Datapath
  - Datapath [lw, sw, R-Type, beq]
- Five CPU Stages
  - Instruction Fetch
  - Instruction Decode
  - Execute ALU Operation
  - Access Memory/Write Register File
  - Memory Writeback
- Multicycle Timing Example
- Finite State Machines
- Adding Instructions
- Muticycle CPU Performance
- Adding Instructions

Adding Instructions

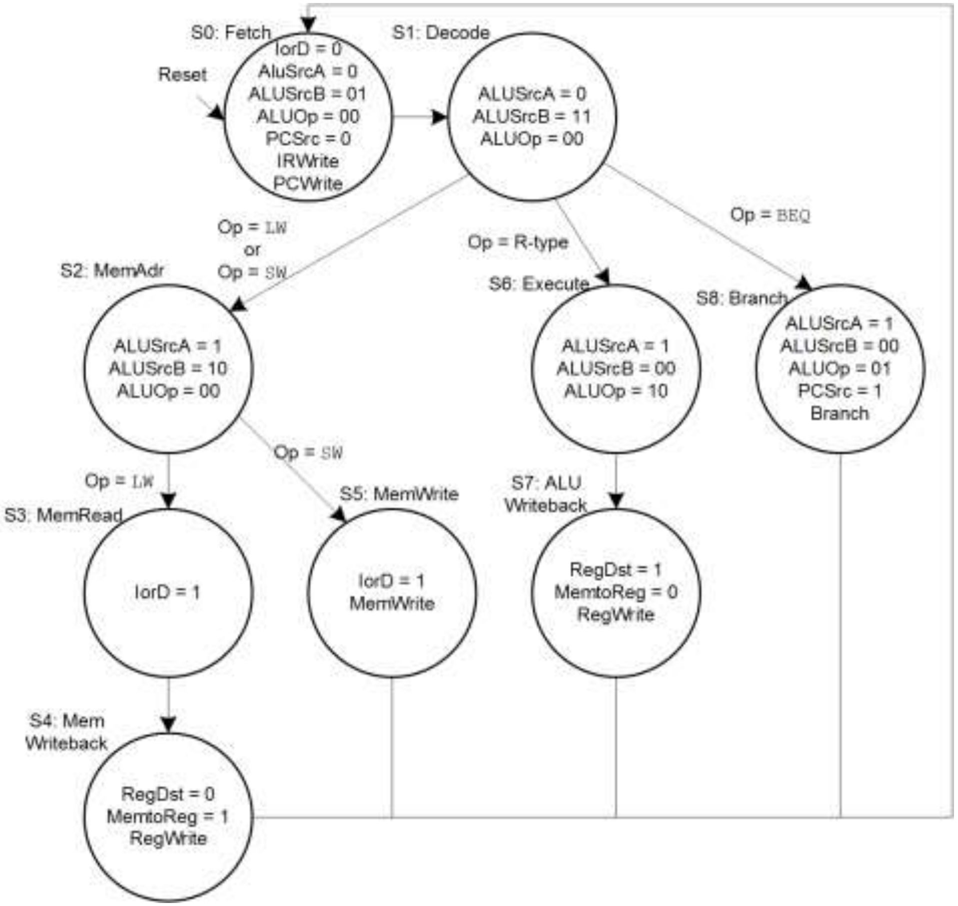
- Modifications to multicycle datapath must be considered in stages
  - New instructions should be broken into sub-tasks/sub-stages which are similar to the existing stages
  - When possible, reuse existing datapath stages and controller states
  - Modification to datapath (new data lines, multiplexers, etc.) should be added only when necessary
  - Large components, such as ALUs, memories, registers, etc. should be added only as a last resort

Adding Instructions

- Care must be made not to increase critical path of any stage
- Adding an extra clock cycle is preferred over increasing critical path of a stage
- Why?
  - Extra stages only affect that instruction type
  - Increasing the critical path in a stage would affect all instructions

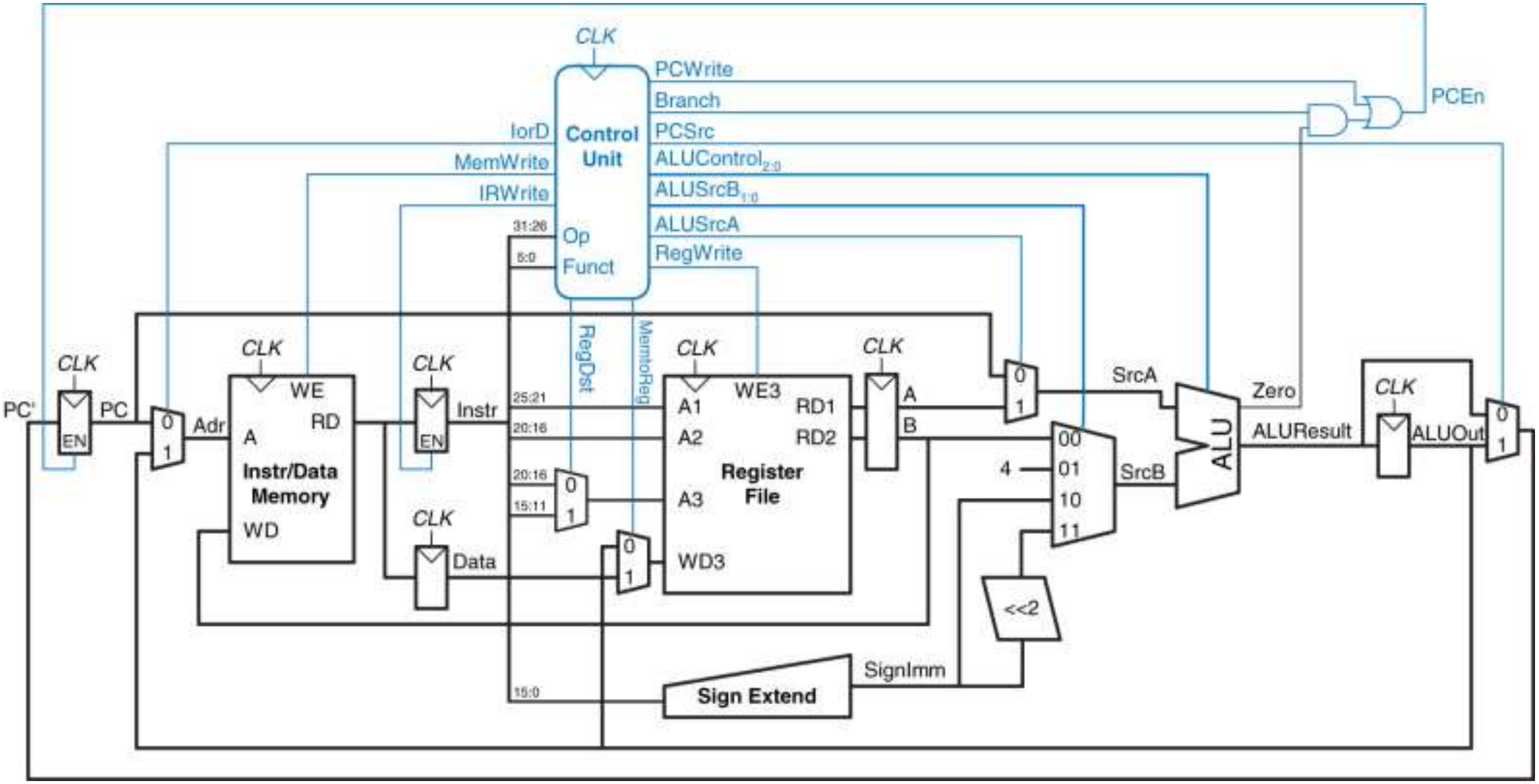
Adding Instructions: `addi`

- The datapath includes all the hardware to execute operations like **`addi`**
- What remains is to modify FSM accordingly



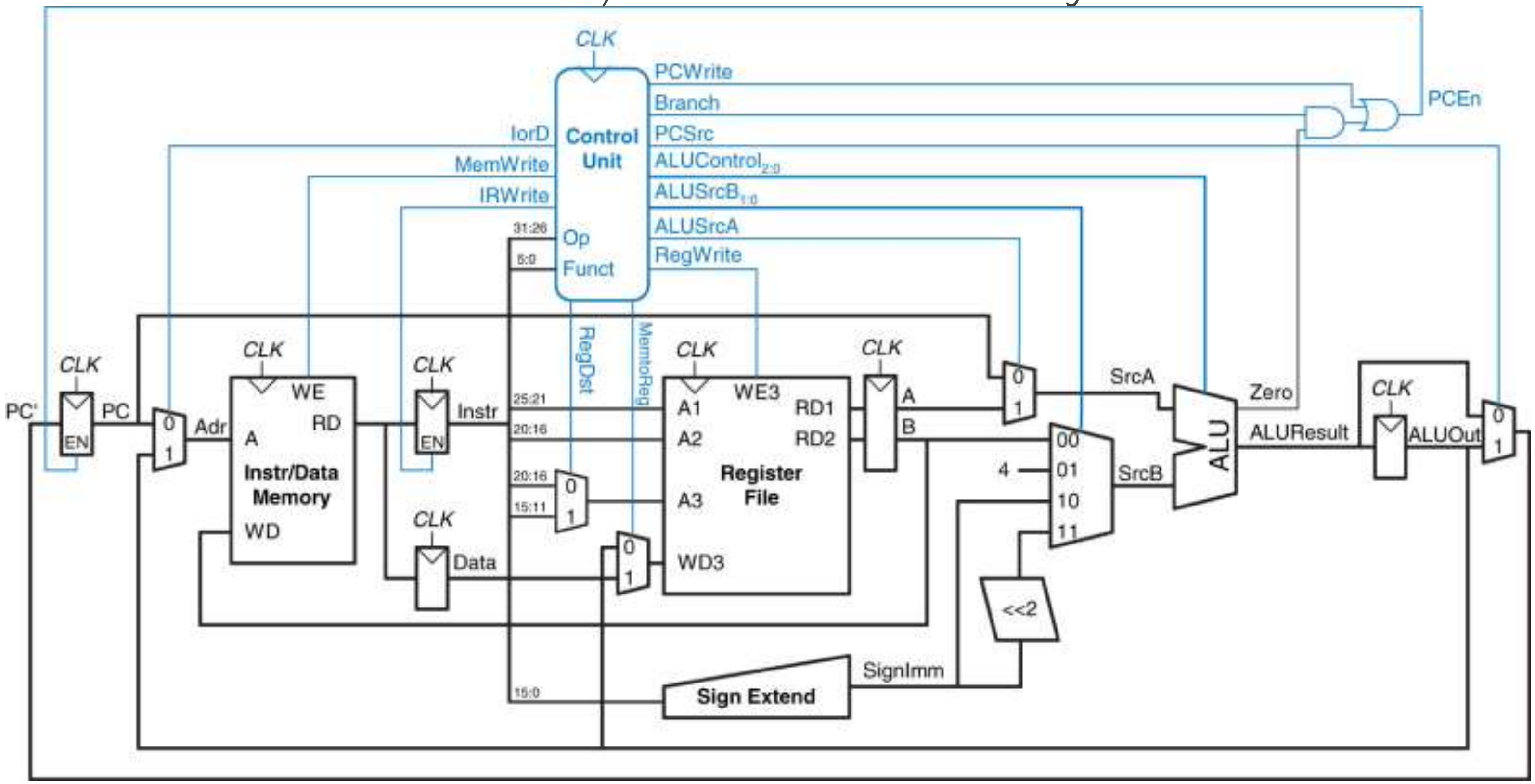
Adding Instructions: `addi`

- Look at the datapath. This is the version that does not have hardware and control signals for jumps



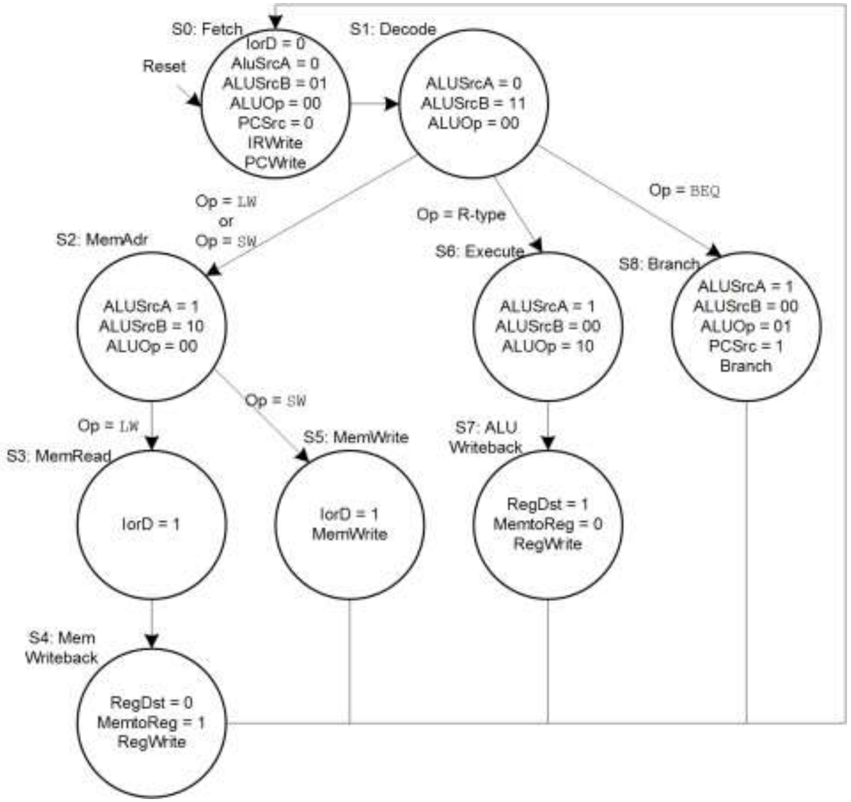
Adding Instructions: `addi`

- **`addi`** is an I-type instruction that writes to the register file
  - Similar to **`lw`** instruction, but has no memory access

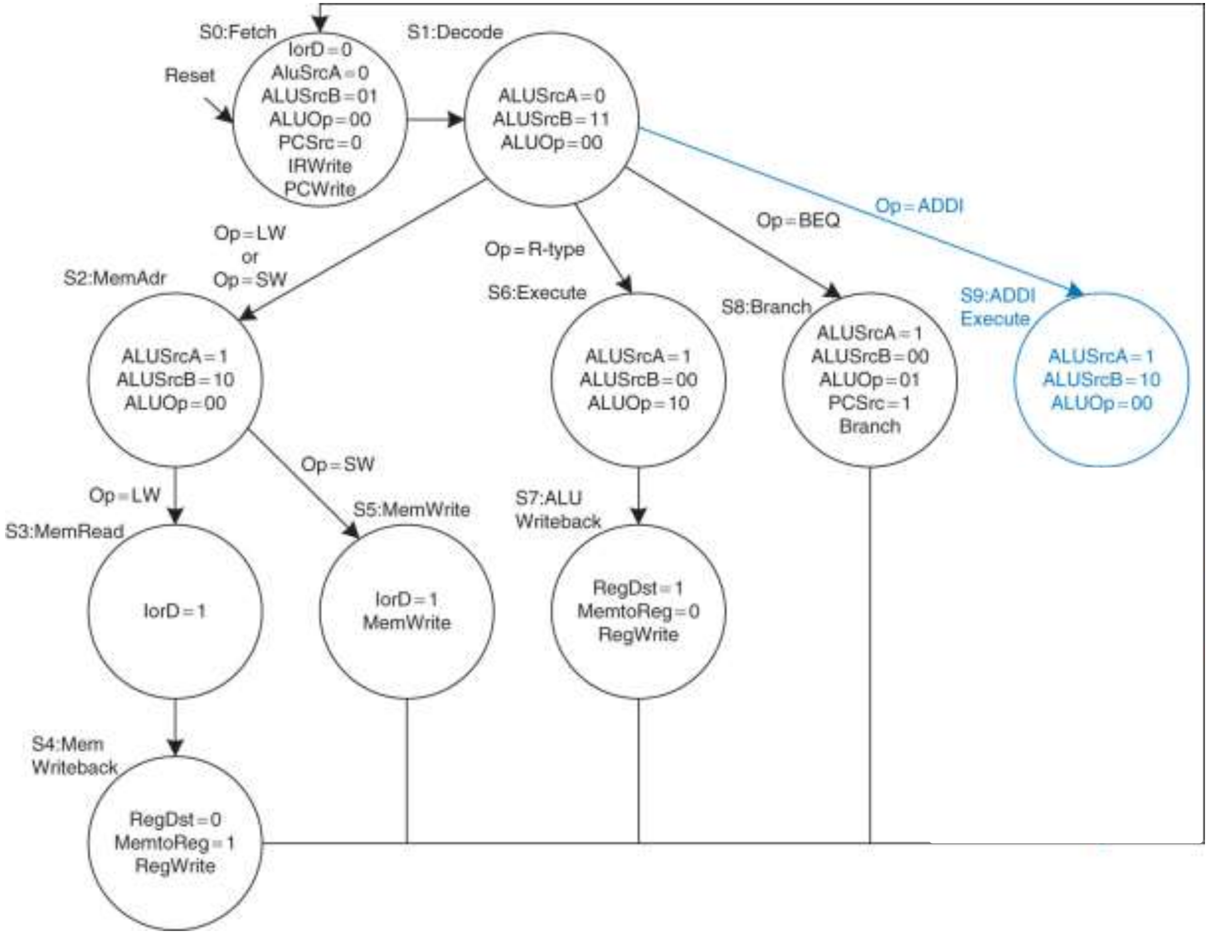


Adding Instructions: `addi`

- How should we modify FSM?
  - States S0 and S1 are unchanged
  - In Cycle 3, should be able to add **`rs`** and sign-extended immediate
    - $ALUSrcA\ =\ 1$
    - $ALUSrcB\ =\ 10$
    - $ALUOp = 00$
    - $ALUCtrl = 010$



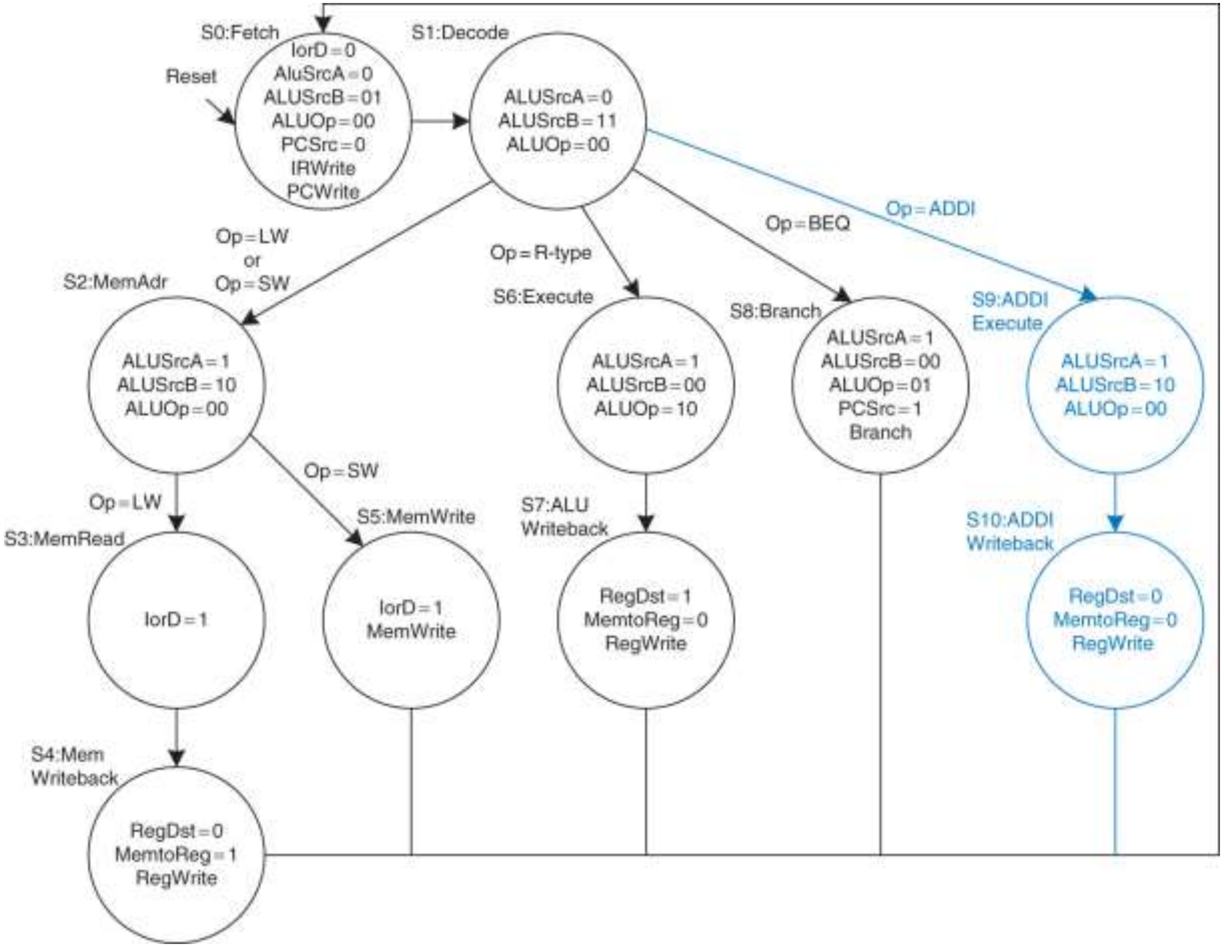
Adding Instructions: `addi`



Adding Instructions: `addi`

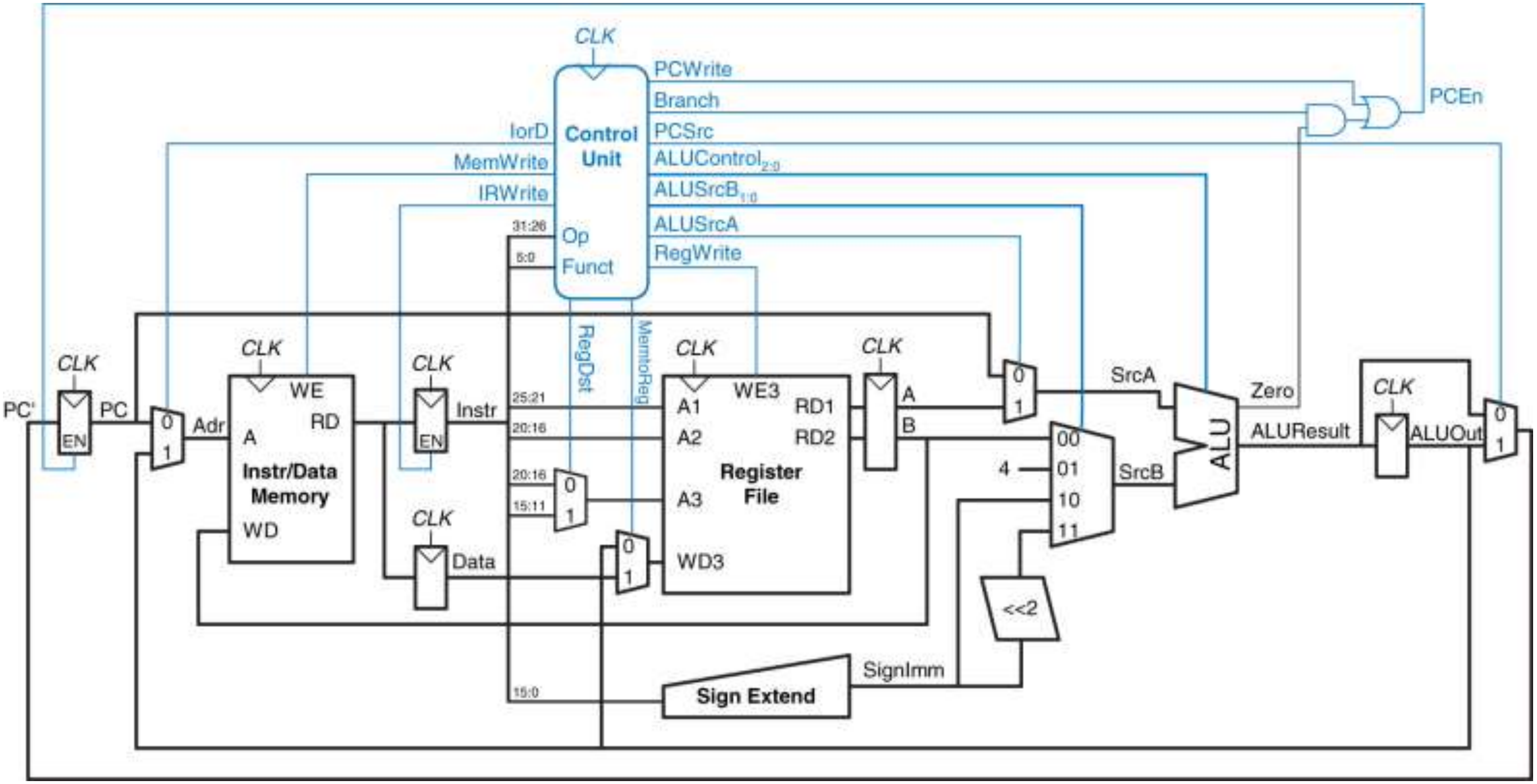
- Now, need to write sum to register file
- Sum is sitting in *ALUOut* register, so need to select that value to be saved in register **`rt`**
- Can look to State S7 (ALU Writeback) for ideas
  - $RegDst = 0$
  - $MemtoReg = 0$
  - $RegWrite = 1$

Adding Instructions: `addi`



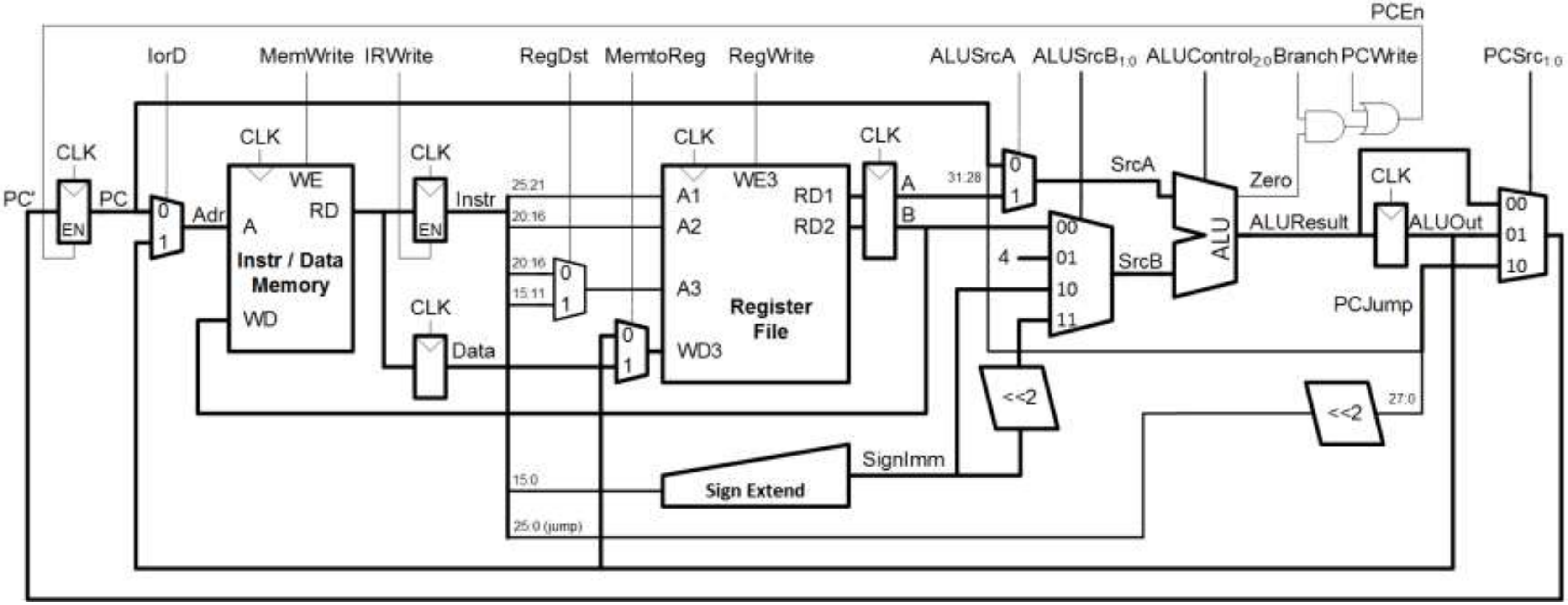
Adding Instructions: `j`

- Hardware datapath without hardware components for **`j`**.



Adding Instructions: `j`

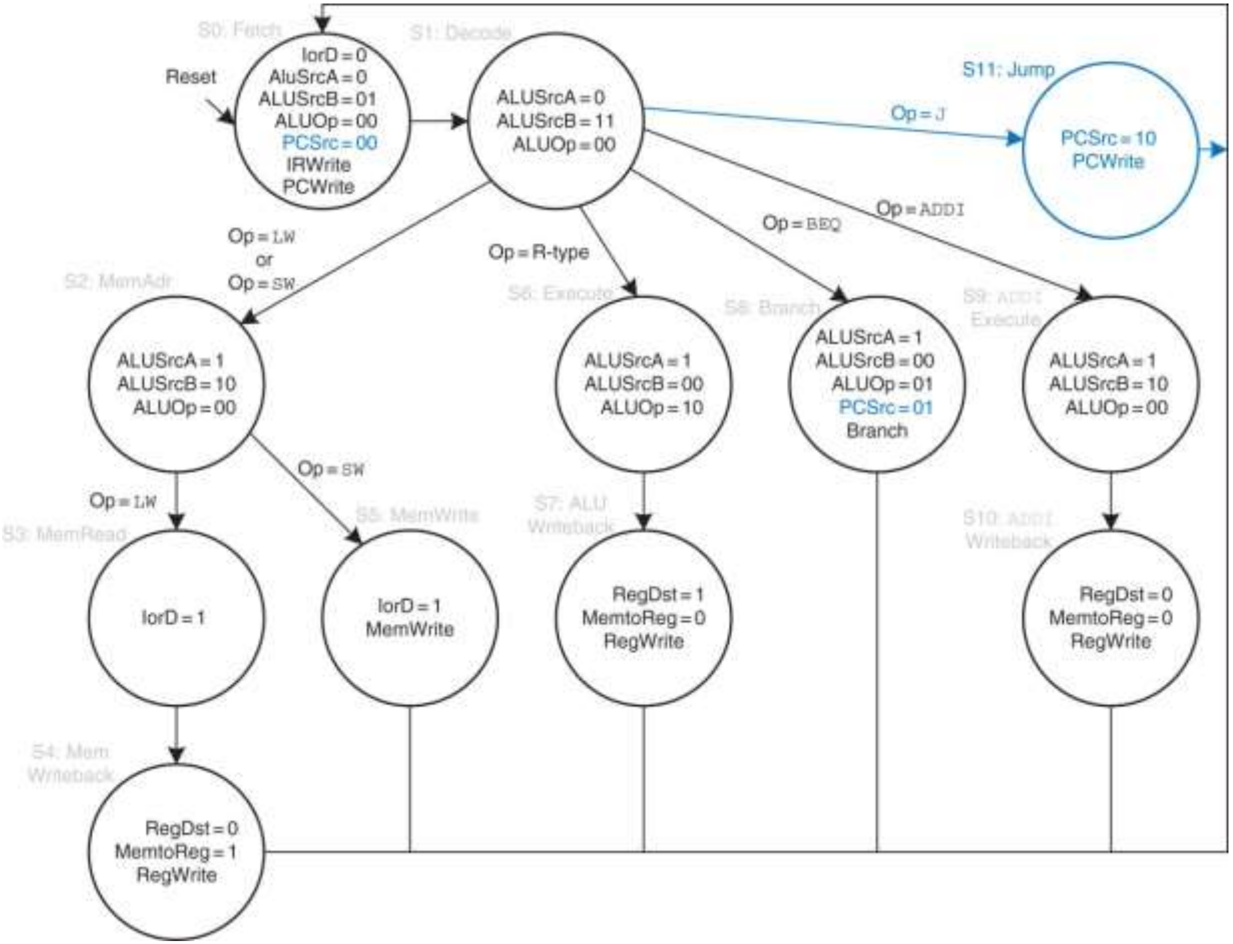
- `j` prepends 4 most significant bits of PC to 28 bits formed by left-shifting instruction's immediate value by 2 bits
- Additional input and control bit added to *PCSrc* to support sending jump address to PC



Adding Instructions: `j`

- In State S0 (Cycle 1) instruction is fetched
- In S1 (Cycle 2) bits $IR_{25:21}$ and $IR_{20:16}$ are read into register file
  - These actions not applicable to jumps, but it happens anyway and there's nothing to be done about it
  - Also, CPU is calculating a branch address (and thus, *reading* the PC) Again, it doesn't help with a jump, but that's OK because results will not be used
- In Cycle 2 (State S1) jump target will be assembled from $PC_{31:28}$ and right-shifted immediate
- Add a new state, State S11 (Cycle 3) where PC is updated with jump target
  - $PCWrite = 1$
  - $PCSrc = 10$

Adding Instructions: j



Multicycle CPU Performance

- Main motivation for developing multicycle CPU to replace the single-cycle CPU
  - ➔avoid making all instructions take as long as slowest one (usually **`lw`**)
- Multicycle CPU does less work in one
  - each instruction has been broken into multiple, shorter steps
  - Worst case, for multicycle CPU an instruction will take 5 cycles
- If each of those cycles is at most one-fifth as long as a cycle for the single-cycle CPU, then we are ahead
- It turns out we probably cannot achieve higher performance with multicycle CPU

Multicycle CPU Performance

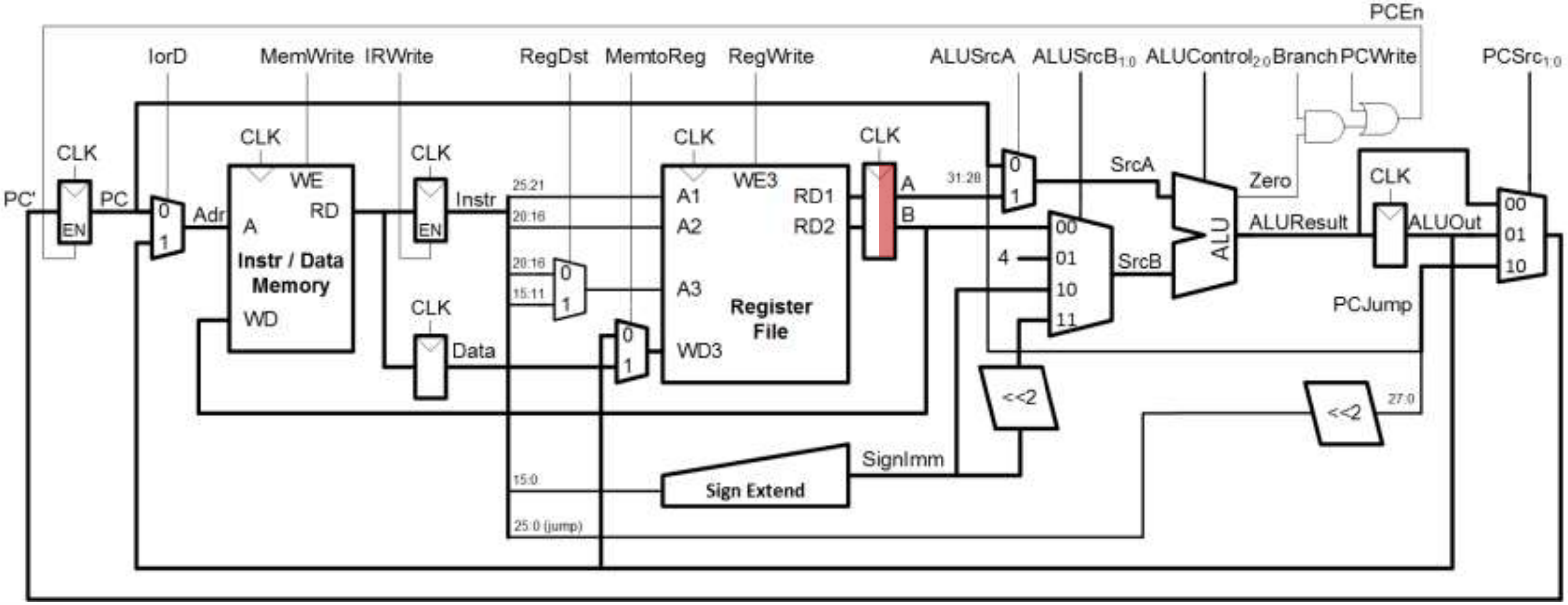
- There are a couple of reasons
  - For single-cycle CPU paid a register “setup” delay once per instruction when updating value in the PC
  - 1. With multicycle CPU the penalty is paid every single stage (for PC and non-architectural registers)
  - 2. CPU’s clock cycle for multicycle CPU limited by slowest stage
    - ➔ typically a stage involving memory access
- Gone through a great deal of work to make a faster CPU
  - ➔ but multicycle CPU is actually going to be slower in many cases
  - May be cheaper though
    - ➔ has fewer components than single-cycle CPU

Adding Instructions: `jr`

- Jump to the address specified by a register:
  - **`PC = Reg[rs]`**
- Instruction could use either the R or I format
  - Only requires one register to be specified
  - Multiple ways to add this instruction to the datapath and control
    - Depending on goals (e.g., minimize hardware changes, minimize instruction time) choose one option over others
- Examine three approaches:
  1. Modify datapath (add hardware) and FSM
  2. Don't modify datapath, but change FSM
  3. Don't modify datapath, but change FSM in a better way than #2

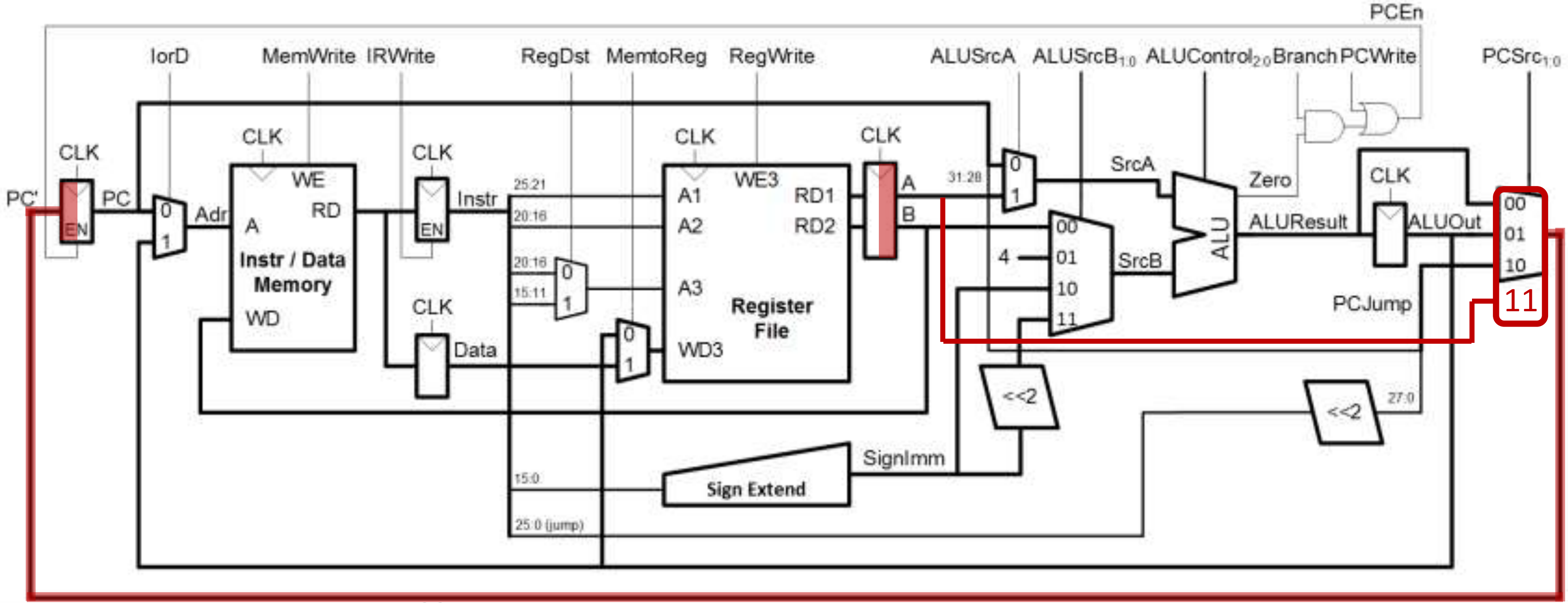
Adding Instructions: `jr` (Ver. 1)

- Approach #1: modify datapath
- Fetch and decode are performed as normal
- At this point, **`Reg[rs]`** is in *A* register of datapath
- We need to send *A*'s contents to PC
  - This would be in third clock cvcle of instruction

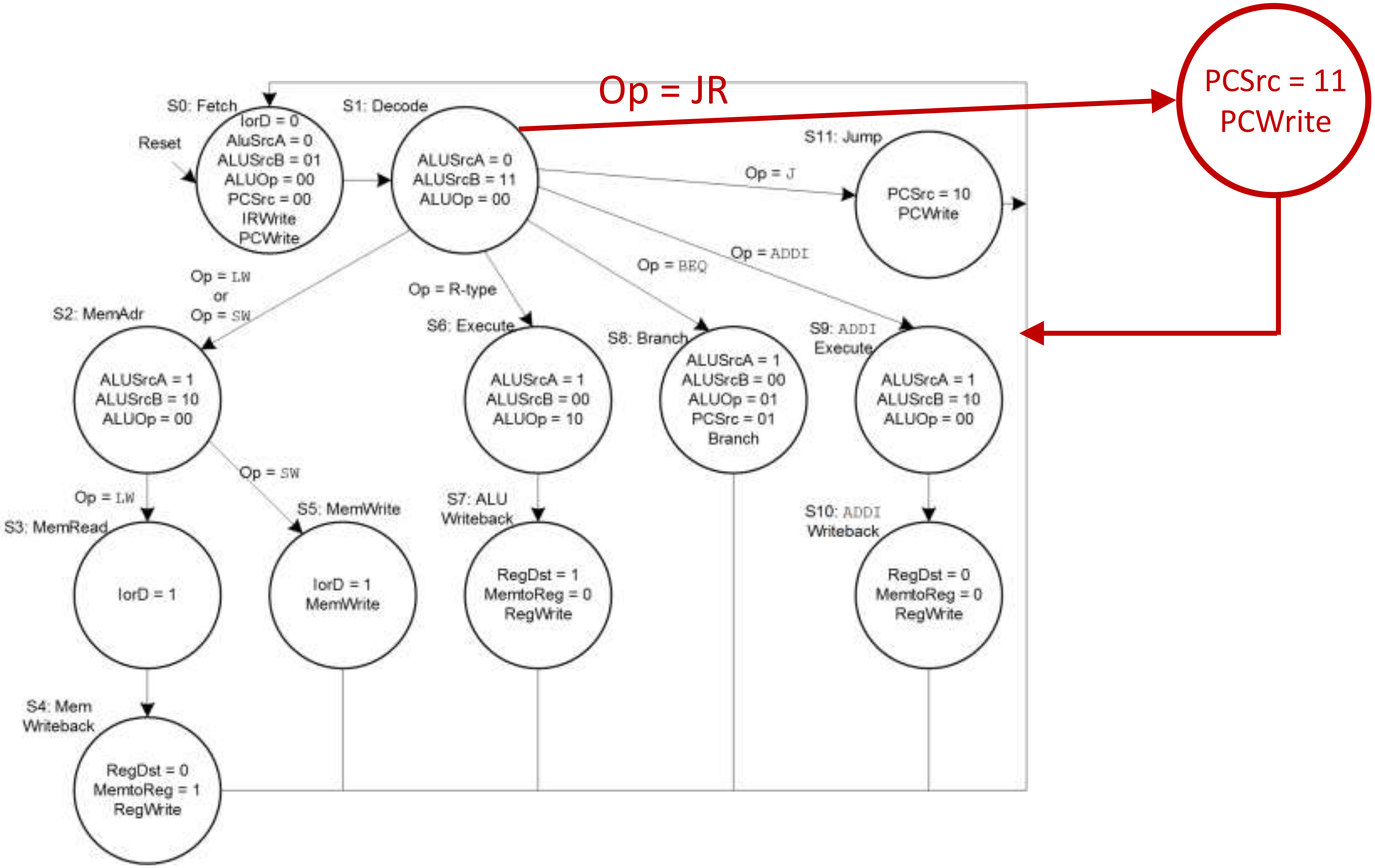


Adding Instructions: `jr` (Ver. 1)

- *PCSrc* mux controls what is sent to the PC
  - none of the inputs is connected to *A*, so modify it by adding fourth input
- Finally, need to add new state to the FSM that will direct control unit to output the right control signals

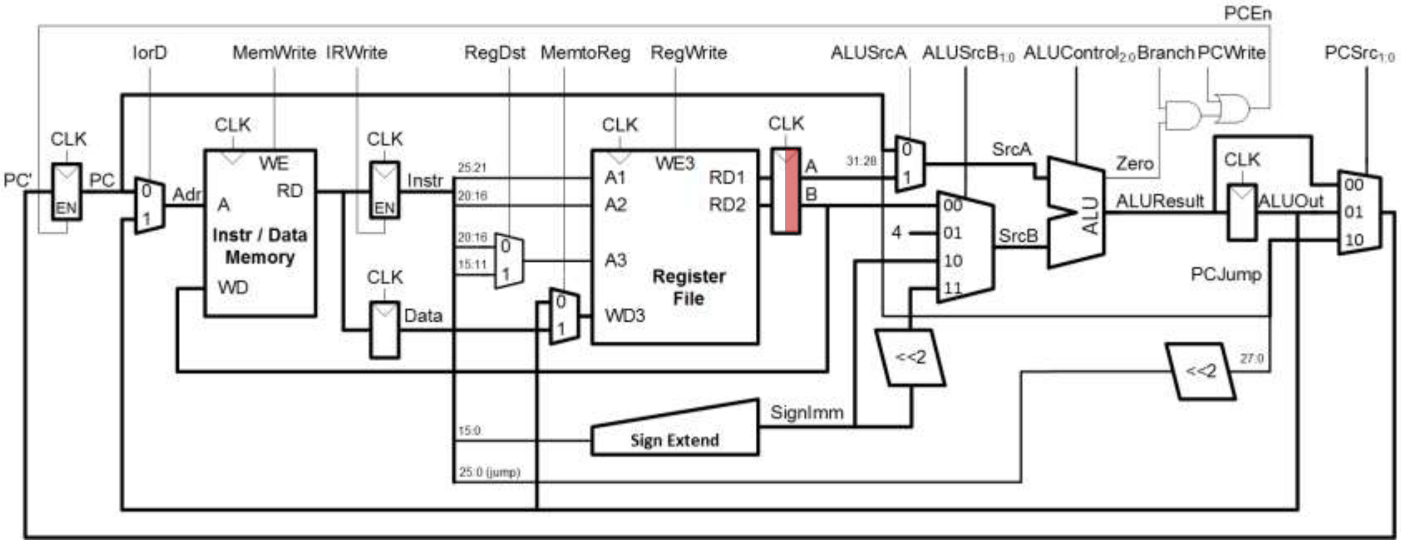


Adding Instructions: `jr` (Ver. 1)



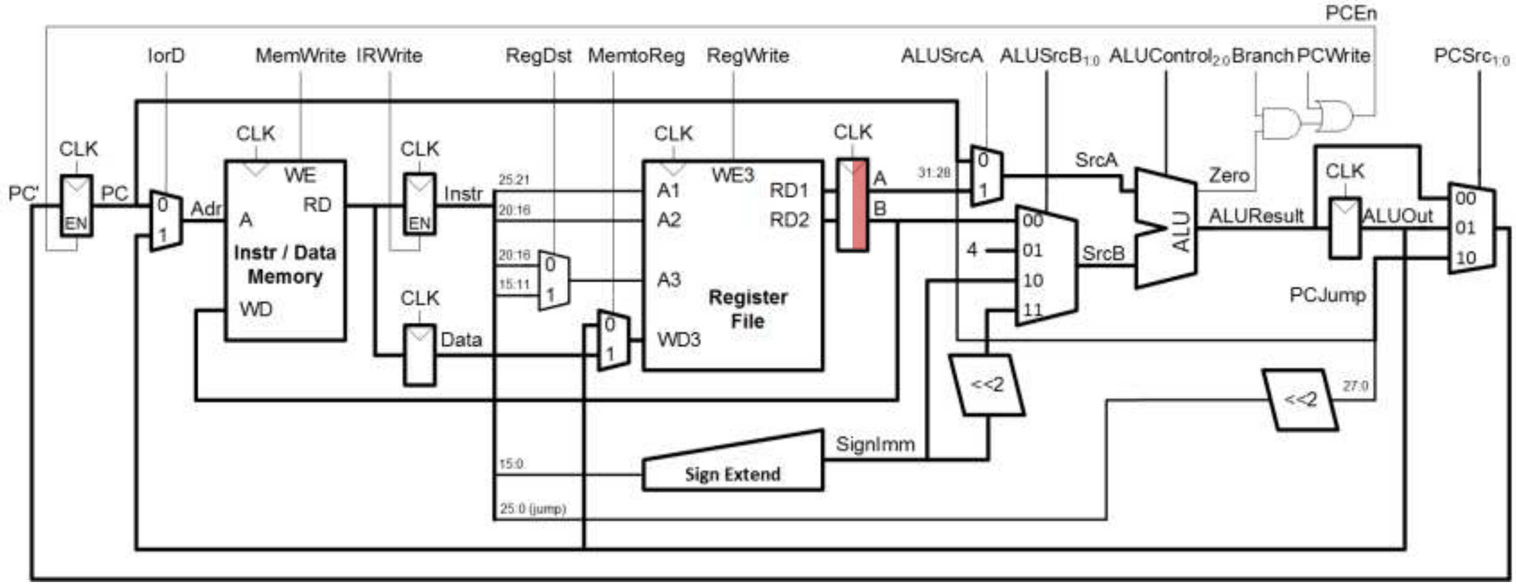
Adding Instructions: `jr` (Ver. 2)

- Approach #2: no datapath modification (the long way)
  - Fetch and decode are performed as normal
  - Now, **`Reg[rs]`** is in *A* register of datapath
- Of the three inputs to *PCSrc*, which one should be used?
- Only 00 or 01 would make sense



Adding Instructions: `jr` (Ver. 2)

- In either case, need to send **`Reg[rs]`** through ALU
- Then the question becomes whether
  - to read *ALUResult* directly
  - send it the *ALUOut* register, then read it out on next clock cycle
- First, try sending it to *ALUOut*

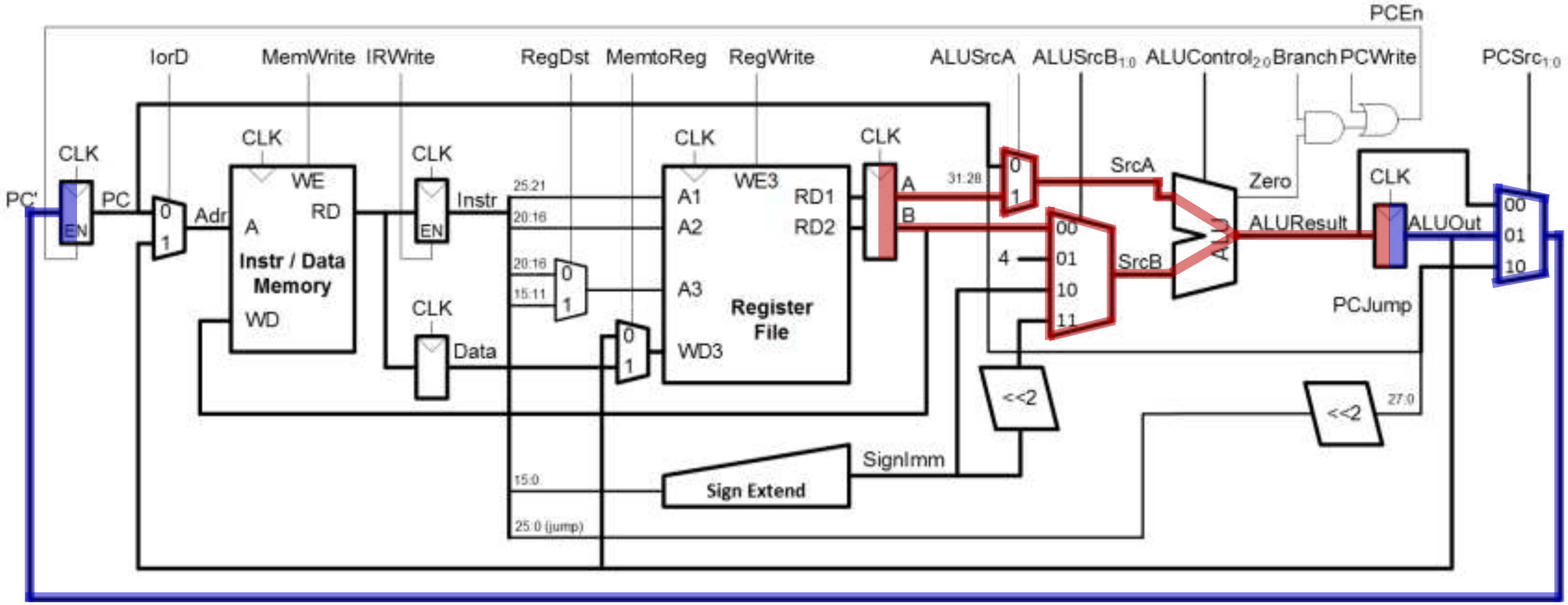


Adding Instructions: `jr` (Ver. 2)

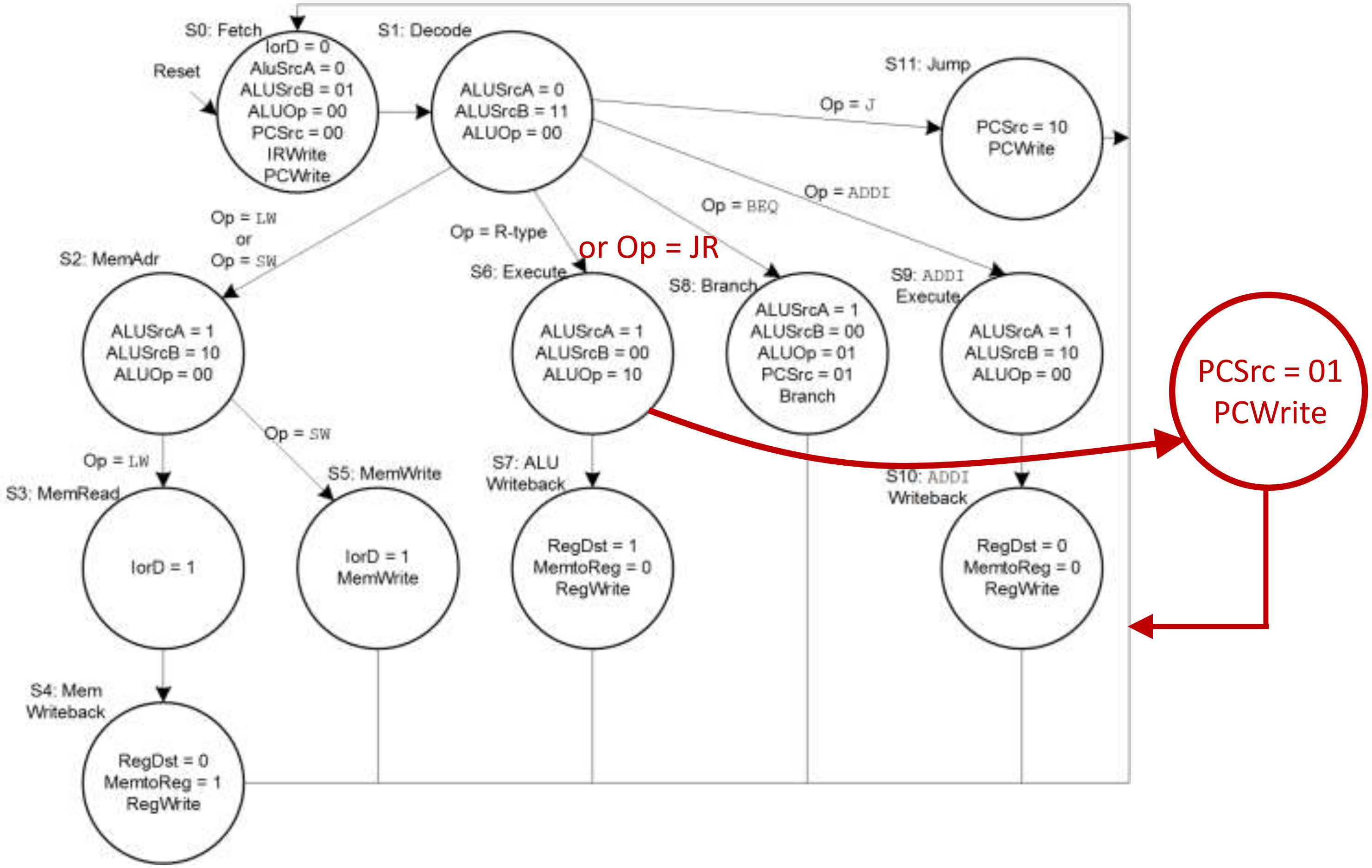
- In Stage 3 of instruction, can pass *A* through ALU by adding/subtracting 0
  - Q: Where does the 0 come from?
  - A: In both R-type and I-type instruction formats a second register can be specified
- Assume the assembler places the **`$0`** register (00000) in instruction's **`rt`** field
  - Now in decode stage, value zero is placed in register *B*
- Using R-type instruction format, set the **`funct`** field to specify addition or subtraction for *ALUControl*
- ➔ So can use existing Stage 3 for R-type instructions

Adding Instructions: `jr` (Ver. 2)

- The result of the calculation is stored in *ALUOut* (red)
- In next cycle (Stage 4), *PC=ALUOut* (blue)
  - Requires a new state for the FSM!

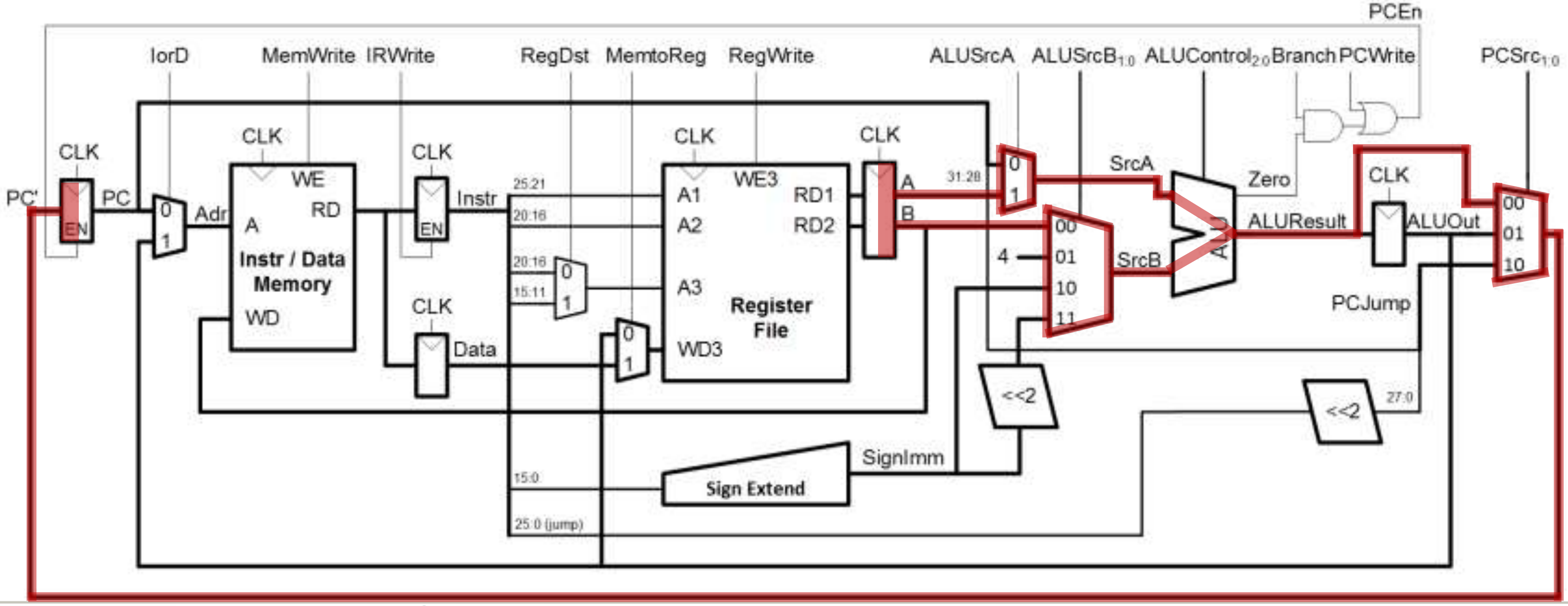


Adding Instructions: `jr` (Ver. 2)

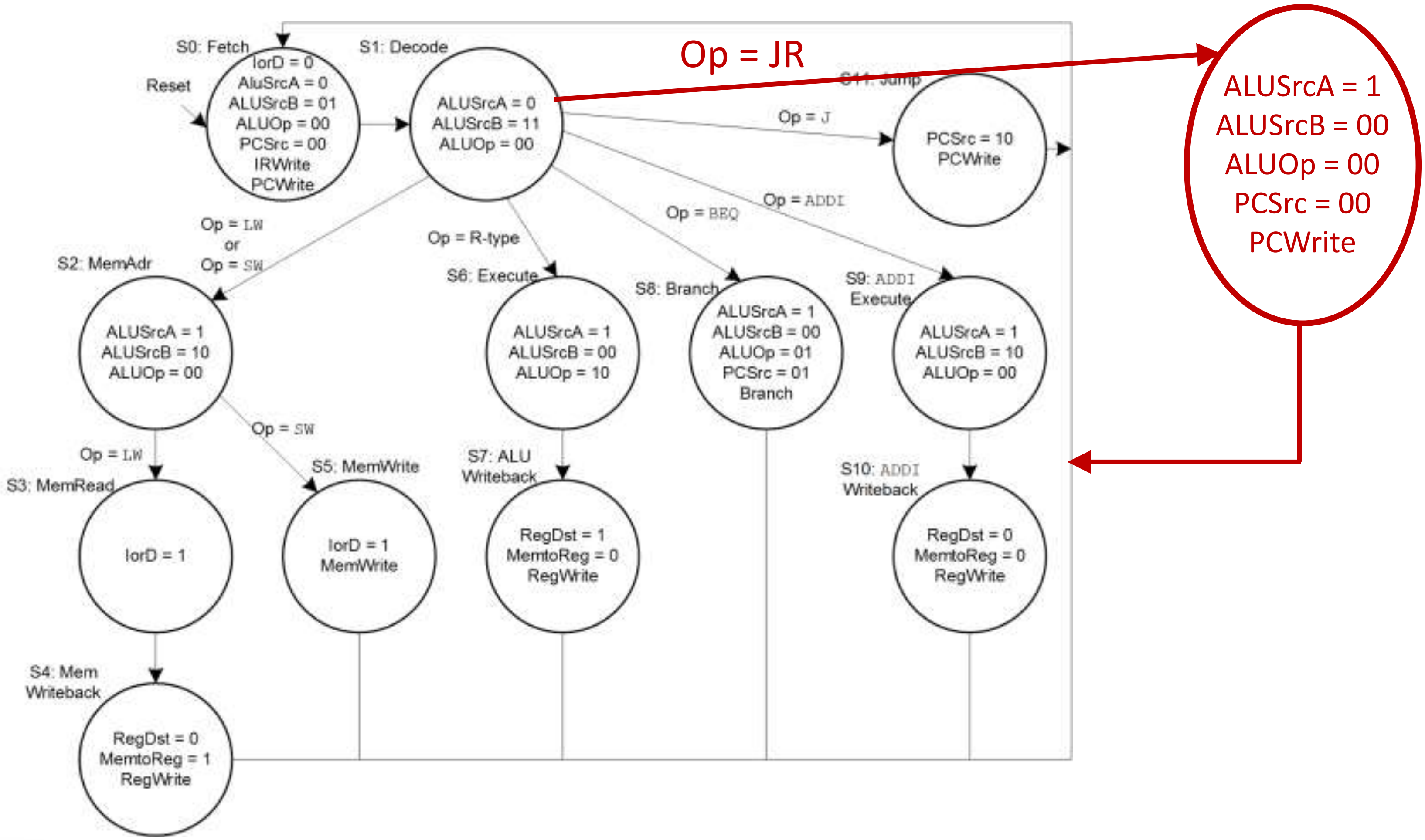


Adding Instructions: `jr` (Ver. 3)

- Approach #3: no datapath modification (the short way)
  - Fetch and decode are performed as normal
  - Now, **`Reg[rs]`** is in *A* register of datapath
- On third clock cycle (Stage 3) read *ALUResult* directly
  - send it to PC



Adding Instructions: `jr` (Ver. 3)



Adding Instructions: `swinc`

- This fictional I-type instruction
  - stores contents of **`Reg[rt]`** at memory address **`Reg[rs]+SignImm`**
  - Also adds 4 to **`rs`** and writes result back to **`rs`**
  - **`Mem[Reg[rs]+SignImm] = Reg[rt]`**
    `Reg[rs] = Reg[rs] + 4`
- Format: **`swinc rt, immediate(rs)`**
- This is a typical **`sw`** instruction with extra increment by 4 of a register
- Based on this, consider stages and control flow for **`sw`** instruction
  - Fetch and decode are performed as normal
  - Stage 3 takes care of computing **`Reg[rs]+SignImm`**

Adding Instructions: `swinc`

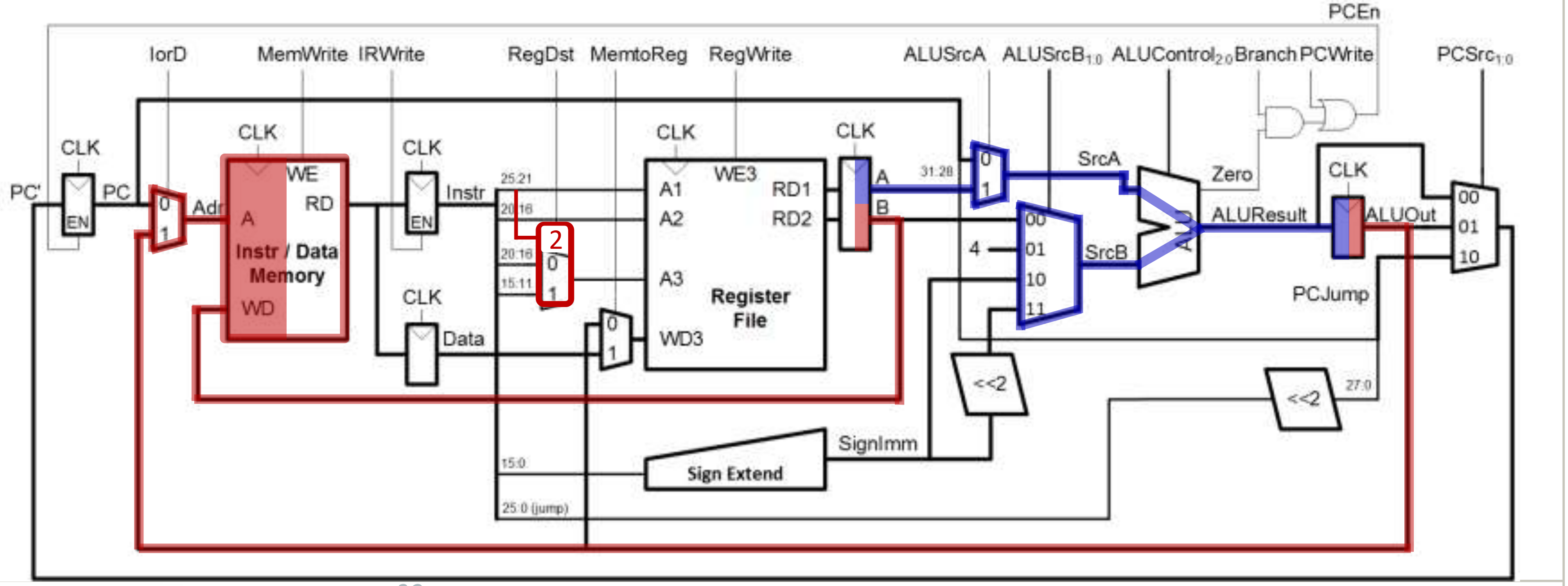
- Stage 4 of **`sw`** (State S5) does not use the ALU
  - So use it during that state to compute **`Reg[rs] = Reg[rs] + 4`**
- Potential issue:
  - In State S5, looking at FSM and match it up with the datapath figure, what is happening?
  - **`Mem[ALUOut] = Reg[rt]`**
- Hmm… calculating **`Reg[rs]+4`** will require a write to *ALUOut* register, so need to read and write *ALUOut* in the same cycle
  - ***Fortunately, this is not a problem because register reads take place at the start of a cycle and register writes take place at the end of a cycle***

Adding Instructions: `swinc`

- What happens is:
  - at the start of Stage 4, effective memory address computed during Stage 3 will be read out of *ALUOut* and sent to the *Adr* input of Memory
  - Meanwhile, ALU will compute **`Reg[rs]+4`**
- At the end of Stage 4, sum is written to *ALUOut*
- New stage must be added to perform **`Reg[rs] = ALUOut`**
  - *RegDst* mux must be modified to take **`rs`** register as an input
- So, to summarize
  - Need to modify State S5 to instruct the ALU to perform addition
  - Need to add a new state to cause sum to be written to register file

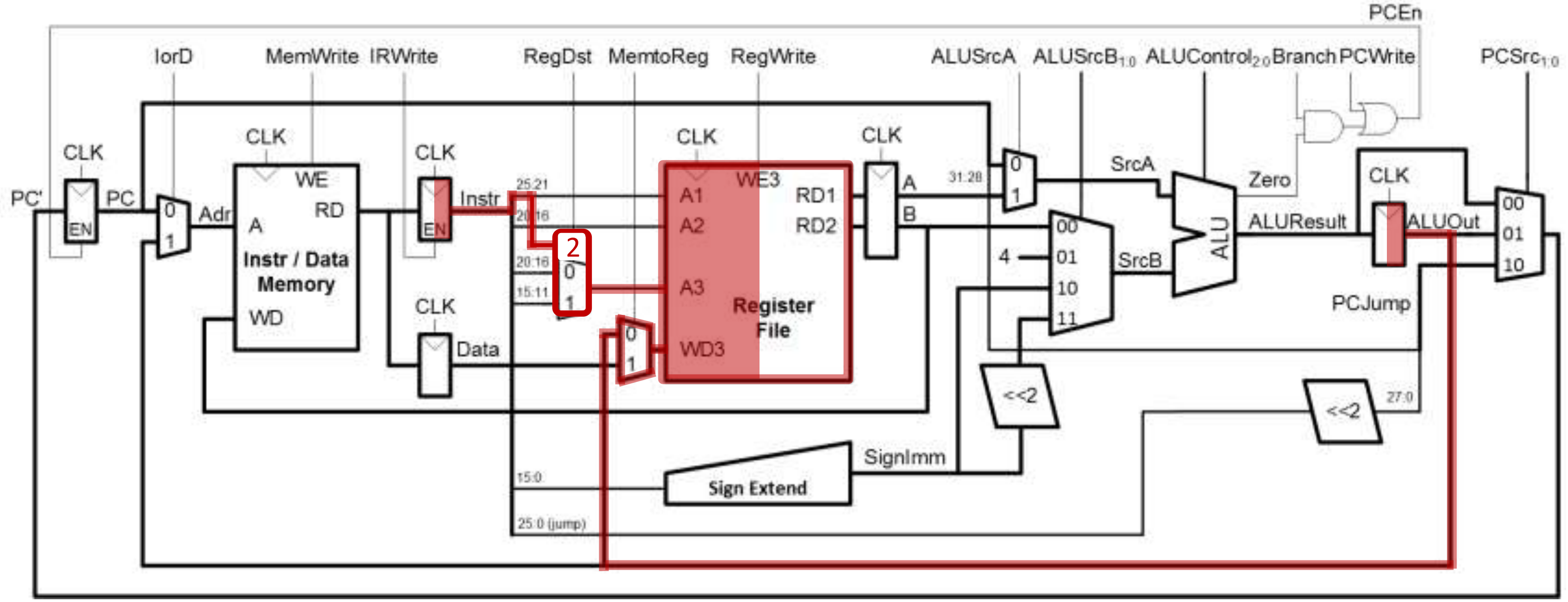
Adding Instructions: `swinc`

- Modified Stage 4 of **`sw`**
- Red indicates the regular activity of **`sw`**
- Blue indicates new activity to compute **`ALUOut = Reg[rs] + 4`**

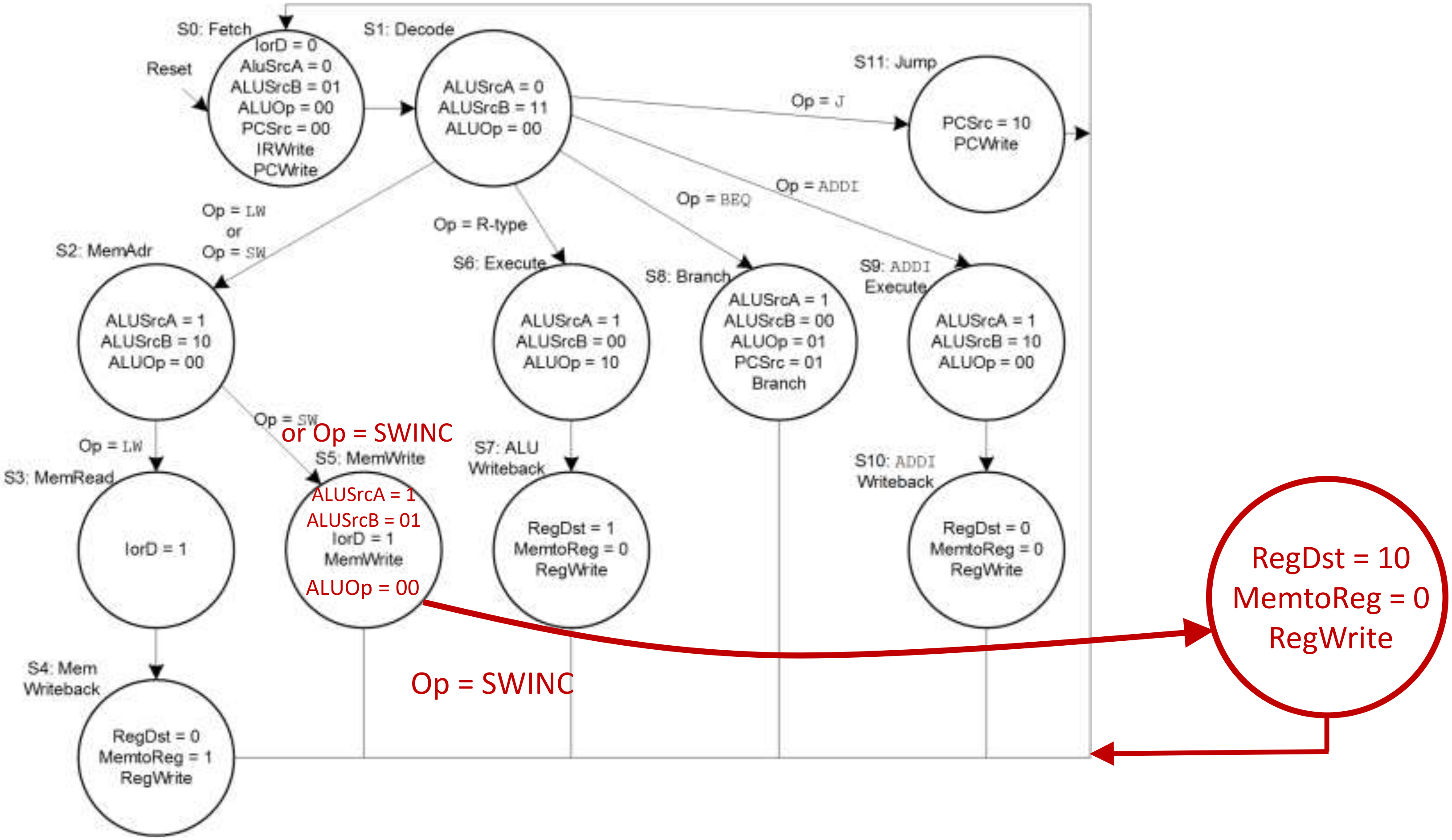


Adding Instructions: `swinc`

- New Stage 5 to finish executing **`swinc`**



Adding Instructions: `swinc`



Questions?