Computer Architecture

Processor: Part 2

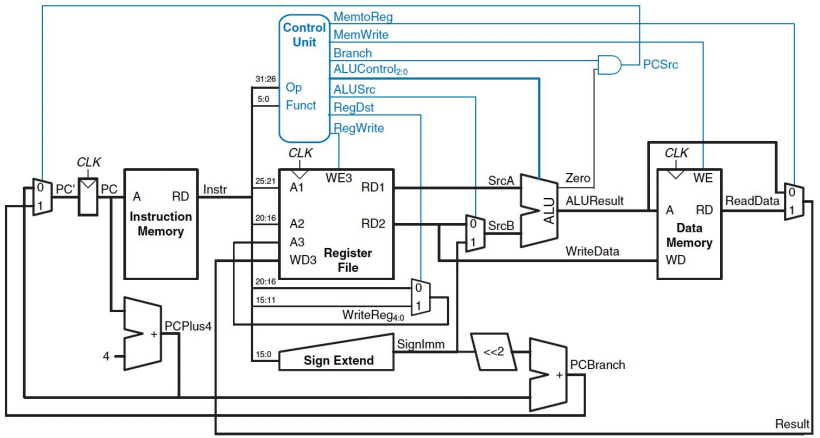
Multicycle microarchitectures

Agenda

- The weakness of a single-cycle processor
- Multiple cycle implementation
  - Datapath
  - Control
- Performance analysis
  - Multicycle processor CPI
  - Performance comparison

The weakness of a single-cycle processor

- It requires a clock cycle long enough to support the slowest instruction (`lw`).
  - Most instructions are faster than `lw`.
- It requires three adders (one in the ALU and two for the PC logic)
  - Adders are relatively expensive circuits, especially if they must be fast.
- It has separate instructions and data memories, which may not be realistic.
  - Most computers have a single large memory that holds both instructions and data and that can be read and written.



Multiple-cycle processor

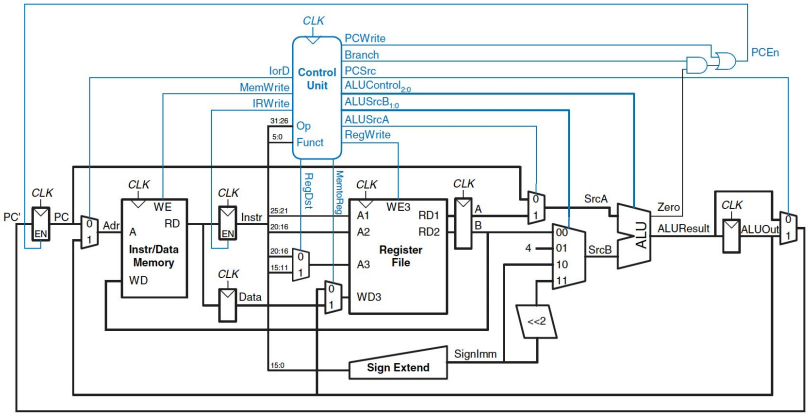
To address these weakness, we have to break an instruction into multiple shorter steps.

In each step, the processor can only do one of the followings

- read or write the memory or register file, or
- use the ALU

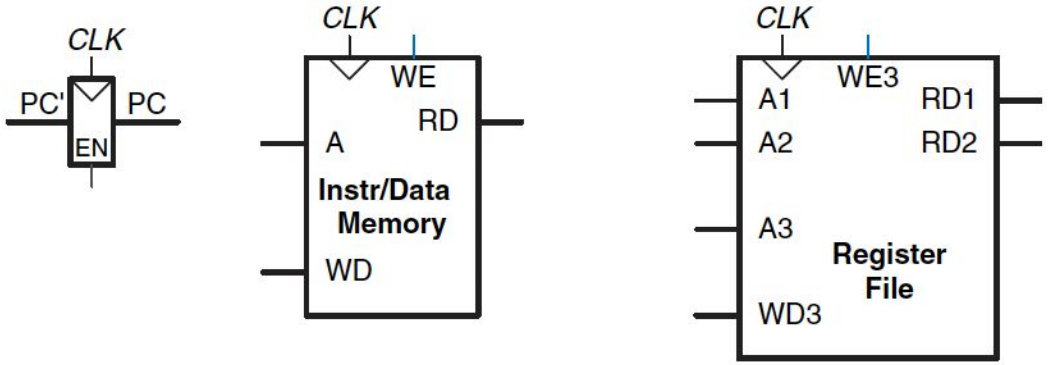
So, the memory and ALU can be **reusable**.

As a result, we should **add state elements** to **hold intermediate results** between steps, and thus, the control for each instruction should be a FSM rather than combinational logic.



State elements

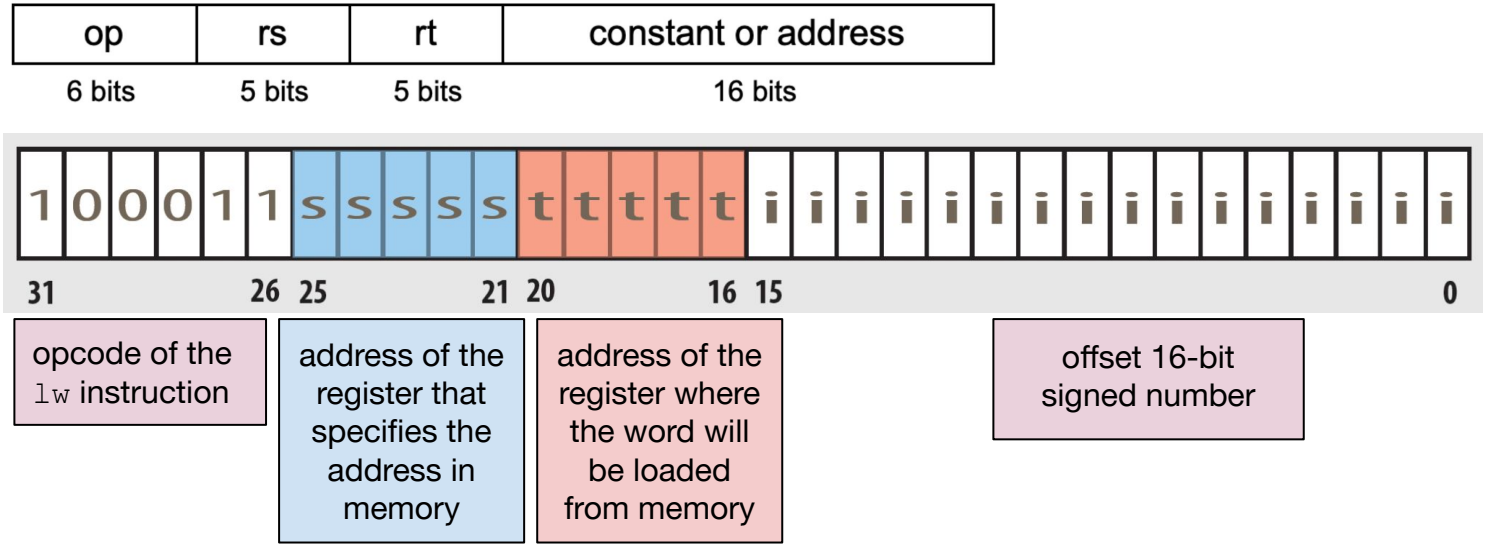
- The PC and register file remain unchanged.
- The memory is changed to a combined memory for both instructions and data.
  - We will read the instruction in one cycle, then read or write the data in a separate cycle.



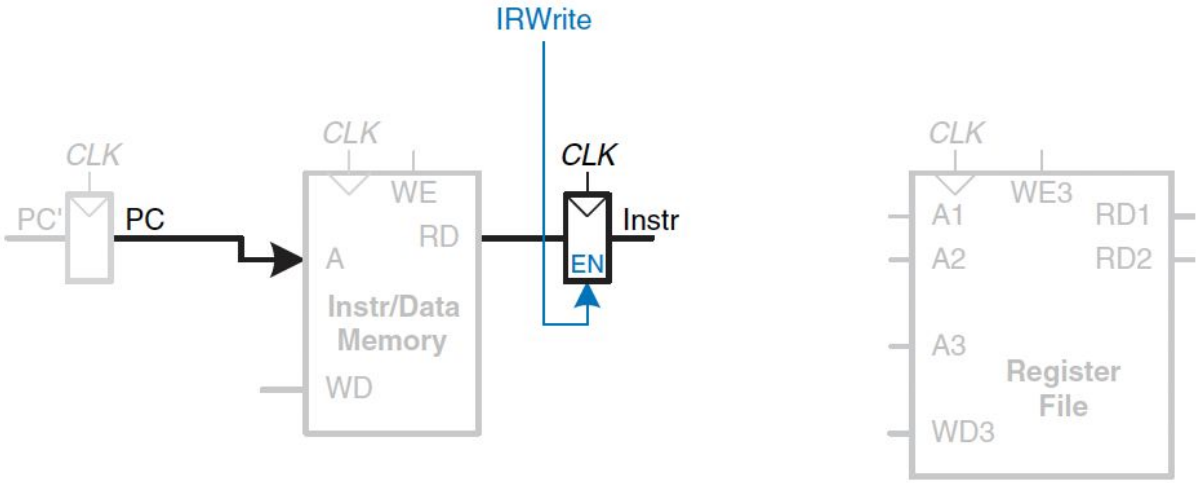
The multiple-cycle microarchitecture

First, we will work out the datapath connections for the `lw` instruction.

- `lw` is an I-format instruction: `lw $t, offset($s)`

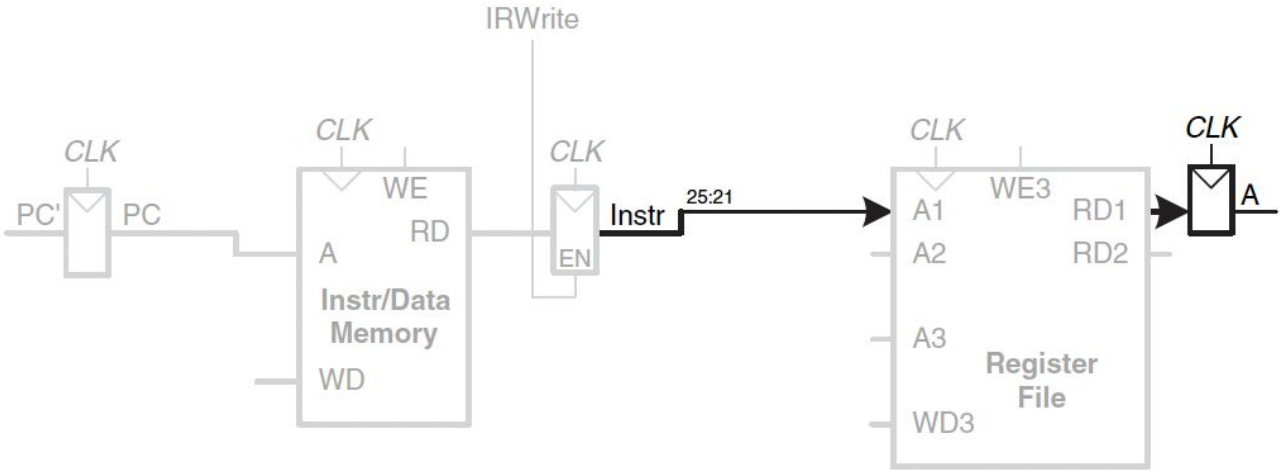


Step 1: fetch instruction from memory



The instruction is read and stored in a new nonarchitectural *Instruction Register* so that it is available for future cycles. The *Instruction Register* receives an enable signal called ***IRWrite*** which is asserted when the *Instruction Register* should be updated with a new instruction.

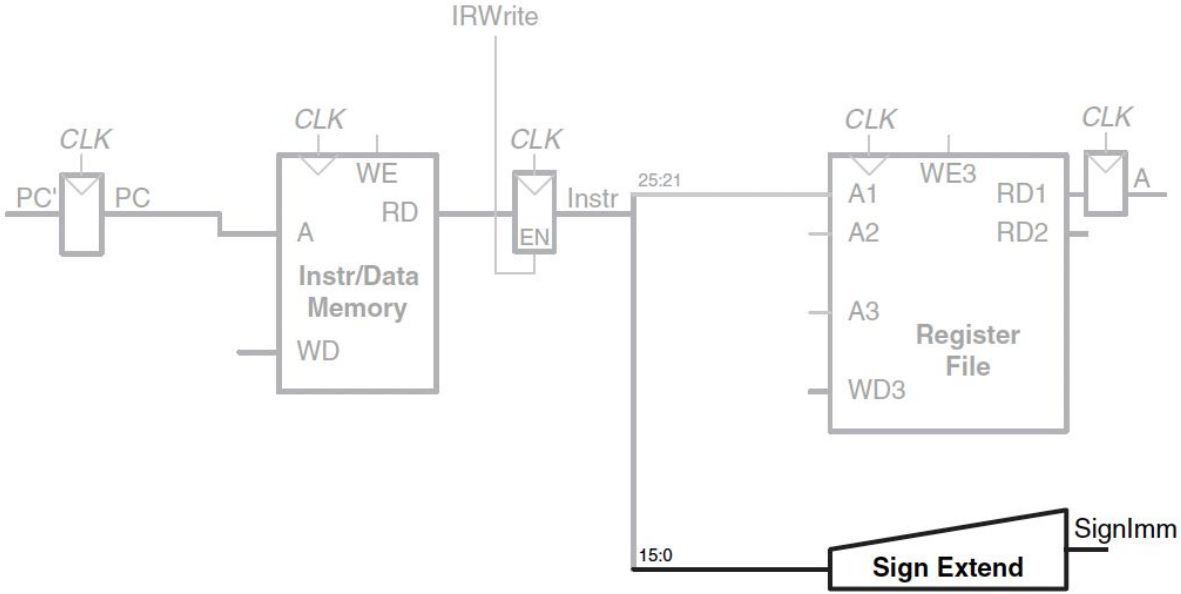
Step 2: read the base address from register file



The 2nd step is to read the source register containing the base address.

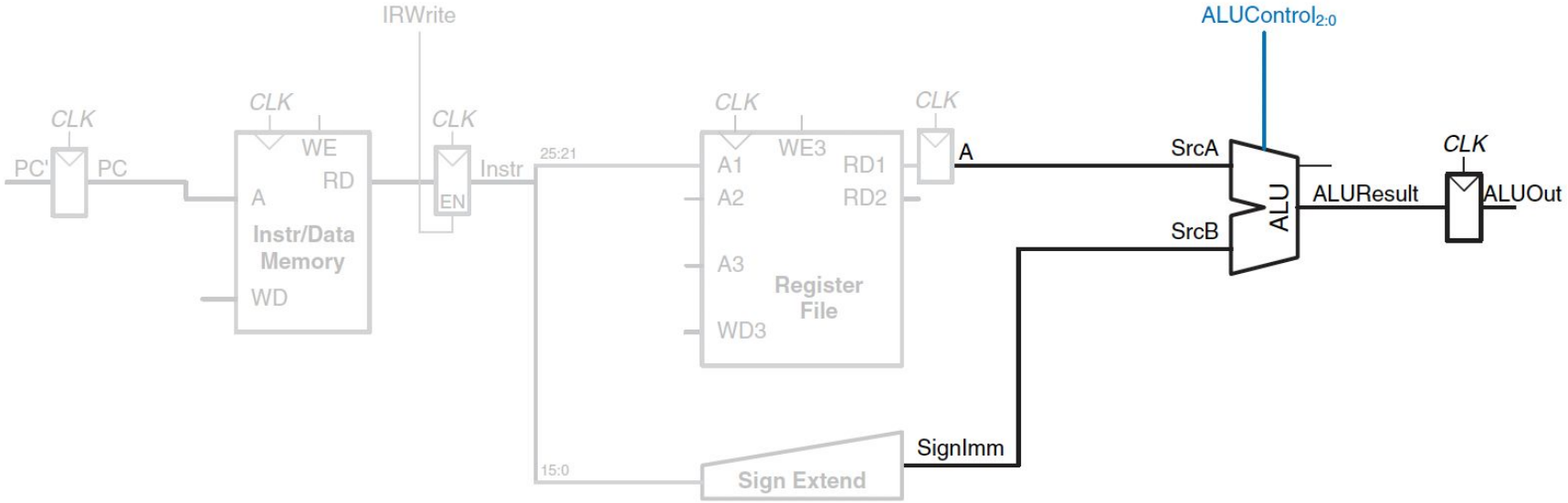
- The source register is specified in the ***rs*** field, ***Instr[25 : 21]***.
- So, just as in the single-cycle case, the 5 bits of the instruction are connected to the address input **A1** of the register file.
- It reads the register onto **RD1** and stores the value in a nonarchiterctural register ***A***.

Step 2: sign-extend the immediate



The `lw` instruction also requires an offset. The offset is stored in the ***immediate*** field, ***Instr[15:0]***. Because the 16-bit immediate might be either positive or negative, it must be sign-extended to 32 bits. The 32-bit sign-extended value is called ***SignImm***.

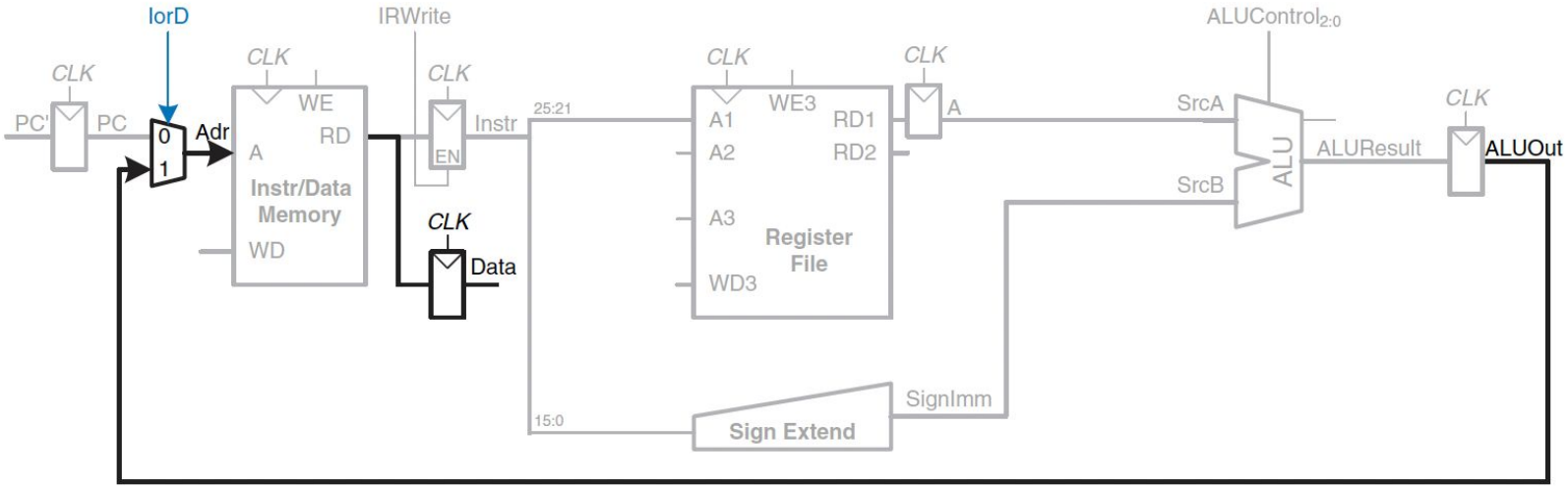
Step 3: add base address to offset



We use an **ALU** to compute the sum of base address and offset. ***ALUResult*** is stored in a nonarchitectural register called ***ALUOut.***

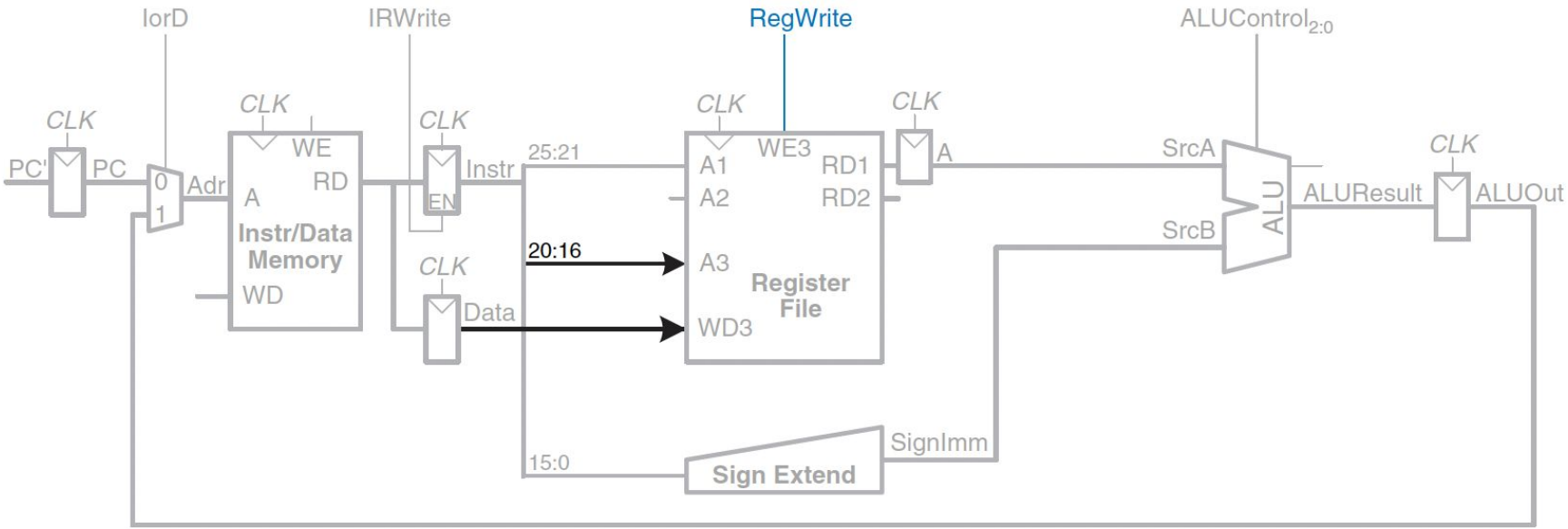
- *SignImm* won't change while the current instruction is being processed because it is given by the current instruction. So we don't have to add register to store it.

Step 4: load data from memory



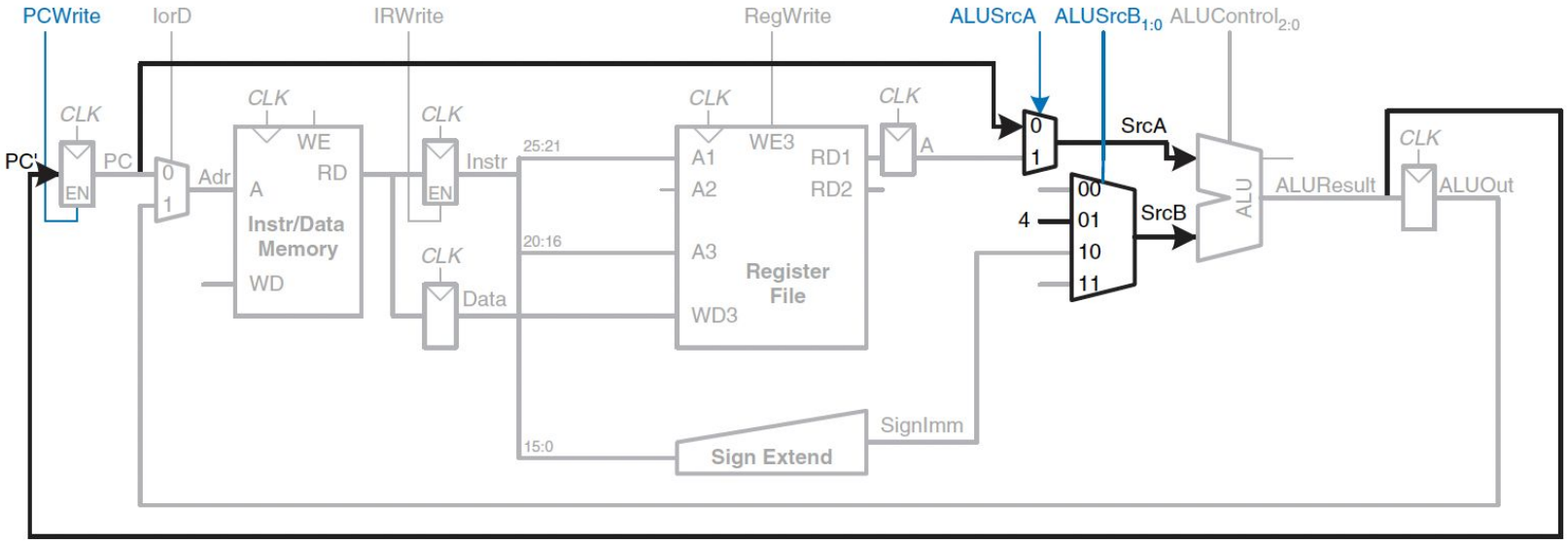
We add a multiplexer in front of the memory to choose the memory, ***Adr***, from either ***PC*** or ***ALUOut***. The select signal is called ***IorD***. (**Note**: to reuse an statement element, we often need to add a multiplexer.)

Step 5: write data back to register file



The destination register is specified in the `rt` field, ***Instr[20:16]***.

Step 1: increment *PC* by 4

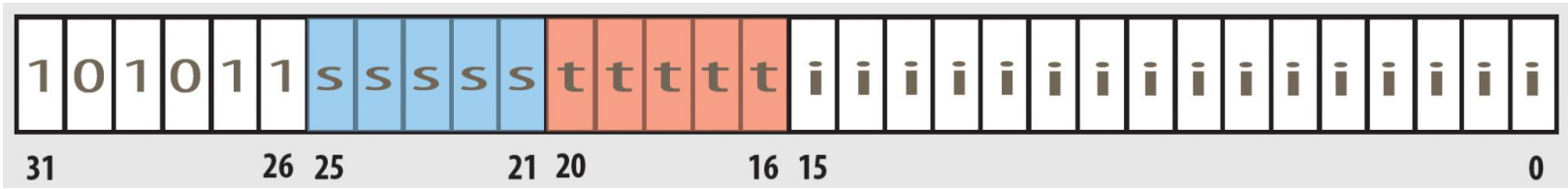


The processor must update the program counter by adding **4** to the old PC. So, we insert multiplexers to choose the ***PC*** and the constant ***4*** as **ALU** inputs. (**Note**: the ***PC + 4*** will be saved into *PC register* in the **same cycle** when fetching an instruction)

- ***ALUSrcA*** for ***PC*** or register ***A***; ***ALUSrcB*** for **4** or ***SignImm***.
- ***PcWrite*** enables the PC register to be written only on certain cycles.

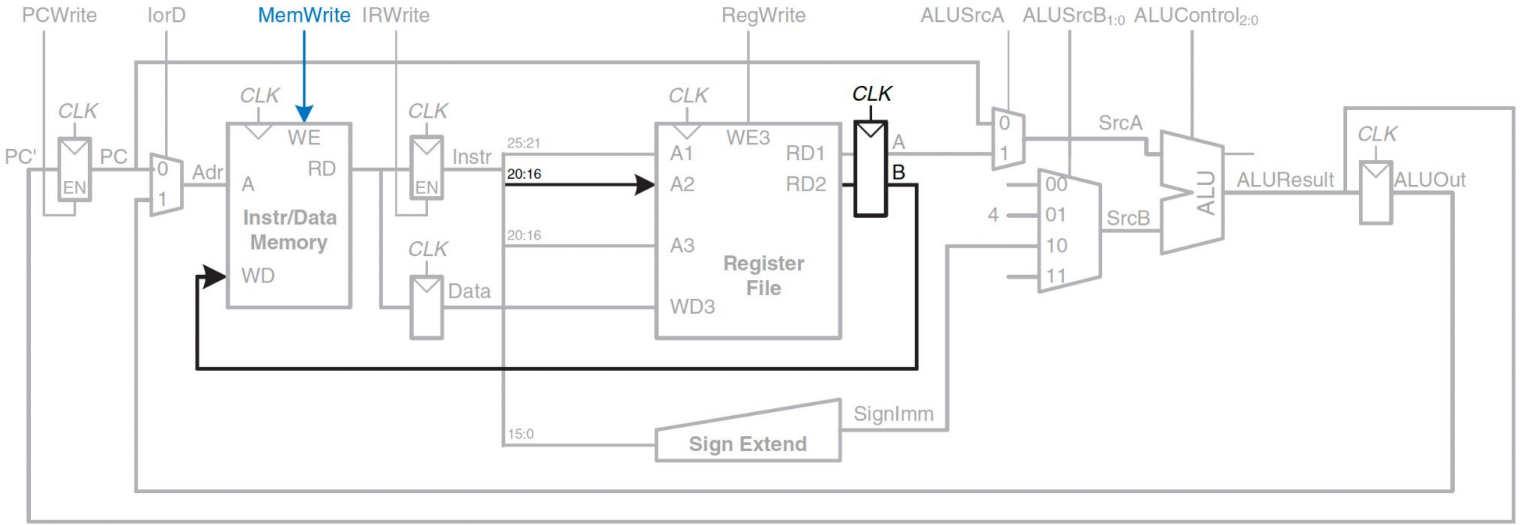
The store word instruction: sw

`sw $t, offset($s)` #stores the value of `$t` in the memory address `Mem[$s+offset]`



- It reads the source operand (i.e., `$s`), then, add the immediate number to the base address to obtain the address. Then, it reads the word from `$t`. Finally, it stores the value to the address.

Enhanced datapath for sw instruction

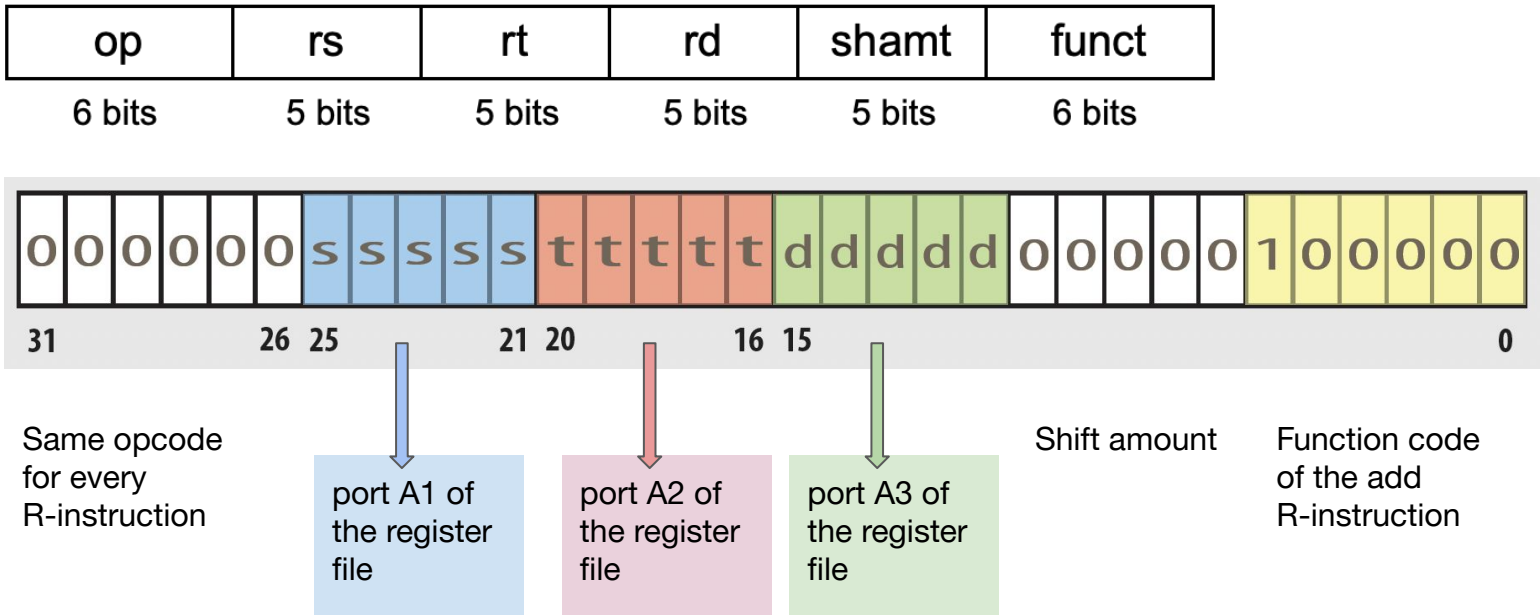


`sw` requires reading a second register from the register file and write it into the memory.

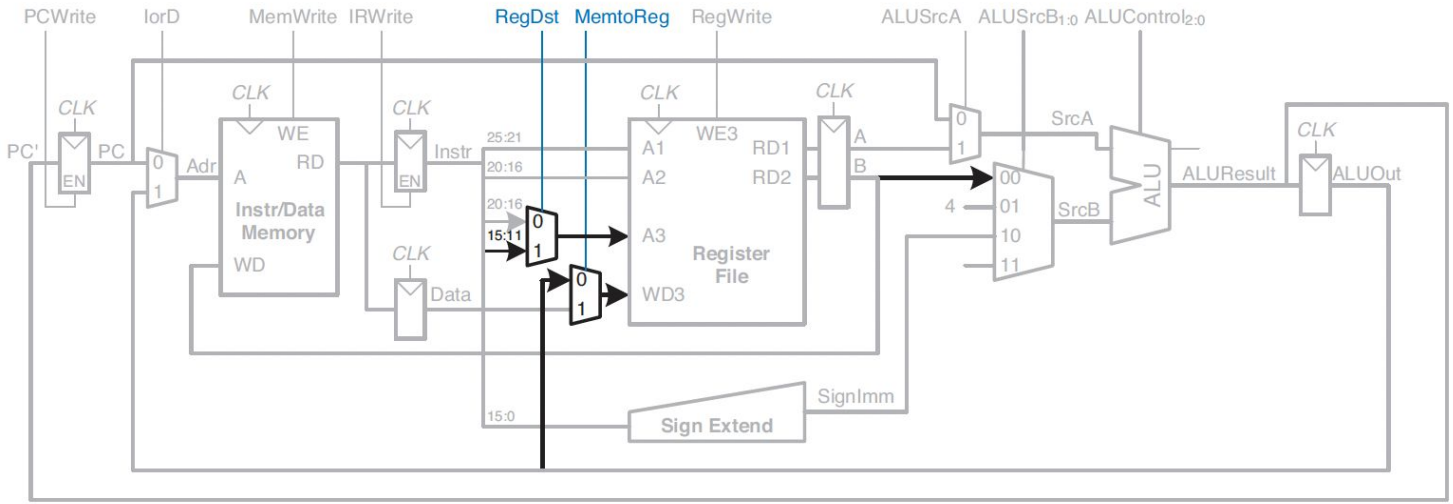
- `rt` field, ***Instr[20:16]*** is connected to the second port of the register file, and value is stored in a register, ***B***.
- In the next step, the value is sent to the write data port (**WD**) of the data memory to be written. Also, the memory receives an additional **MemWrite** control signal to enable the write.

R-type instructions

`add $d, $s, $t` #adds the values of the `$s` and `$t` and stores the result in `$d`

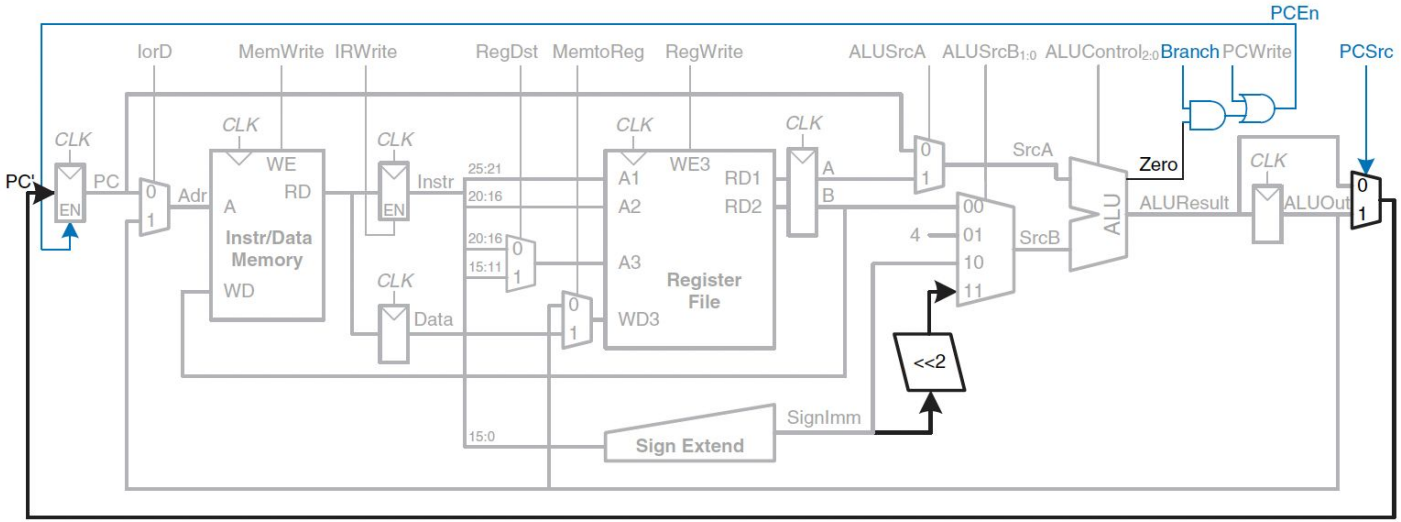


Enhanced data path for R-type instructions



For R-type instructions, it reads two source registers ***ALUSrcA*** and ***ALUSrcB*** control the inputs. On the next step, **ALUOut** is written back to the register specified by the rd field, ***Instr[15:11]***. It adds two multiplexers, and ***MemtoReg*** and ***RegDst*** control the input for **WD3** and **A3**, respectively.

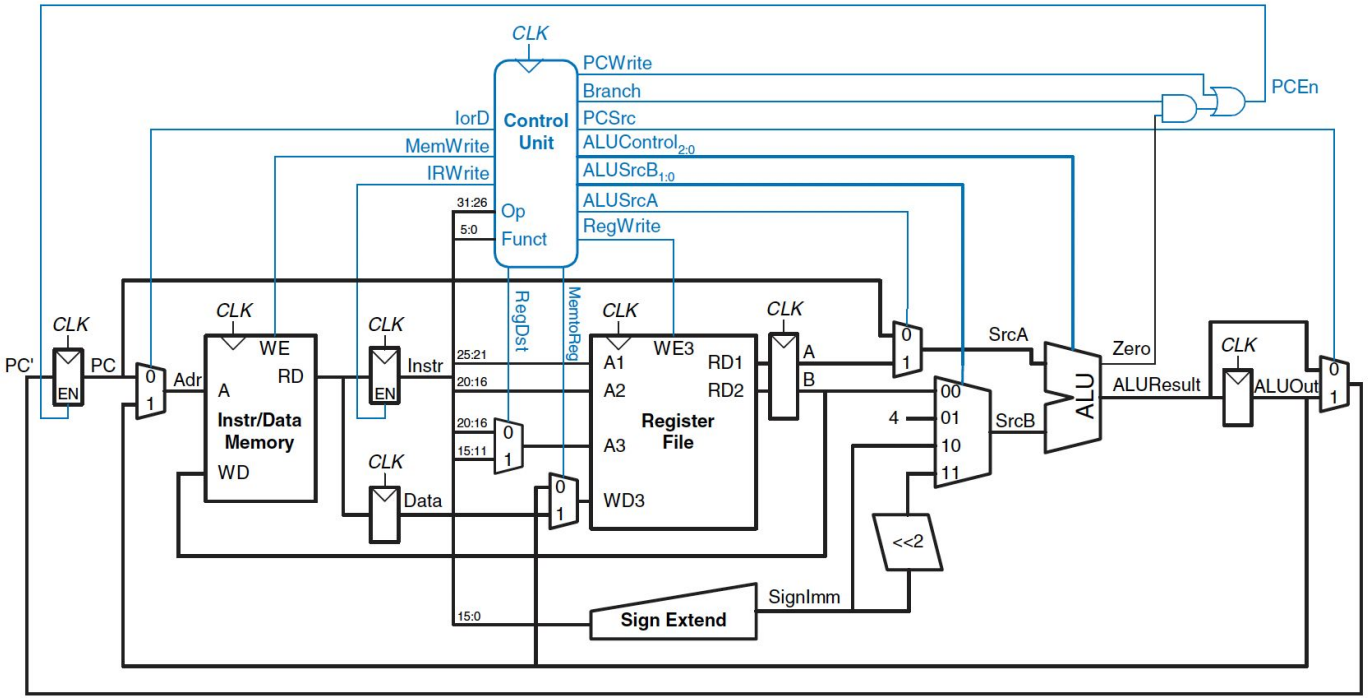
Enhanced datapath for beq instruction



The `beq` instruction compares two registers and branches if they are equal. We should

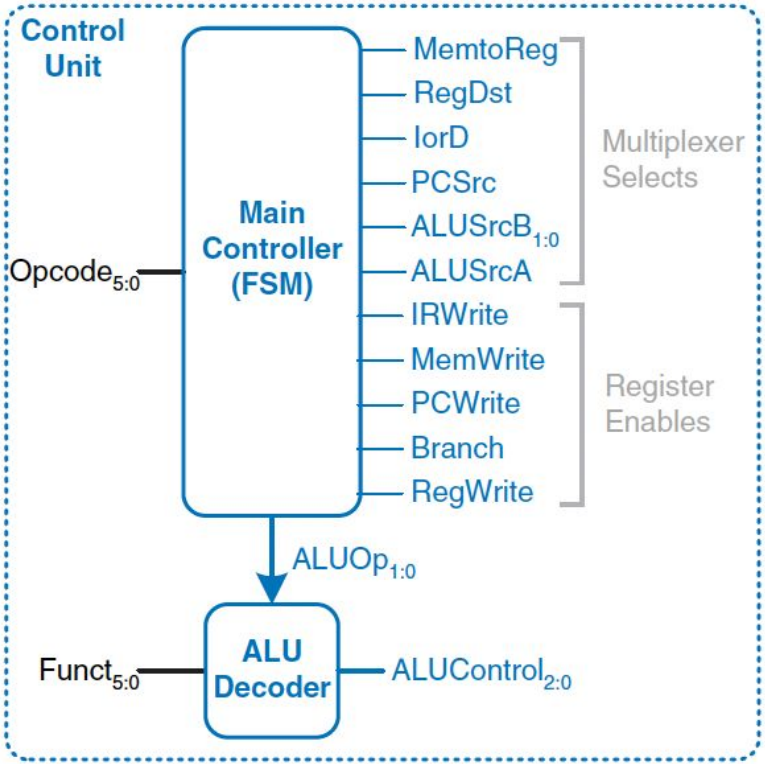
- add the control signal ***Branch*** to the *PC* register, which is set to 1 during the `beq` instruction
- add a shifter for calculating ***PC' = PC + 4 + SignImm ✕ 4***
- add a multiplexer to select between ***PC + 4*** and ***PC + 4 + SignImm ✕ 4***

Complete multicycle MIPS processor



The control unit computes the control signals based on the `opcode` and `funct` fields: ***Instr[31:26]*** and ***Instr[5:0]***.

Control unit internal structure

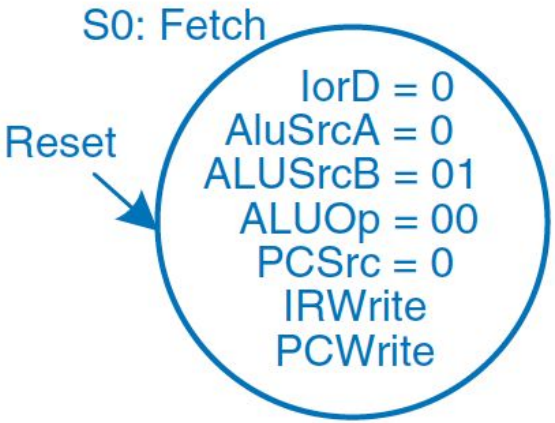


The ALU decoder is the same as that of the single-cycle processor.

But the main controller is an FSM that applies the proper control signals on the proper cycles or steps.

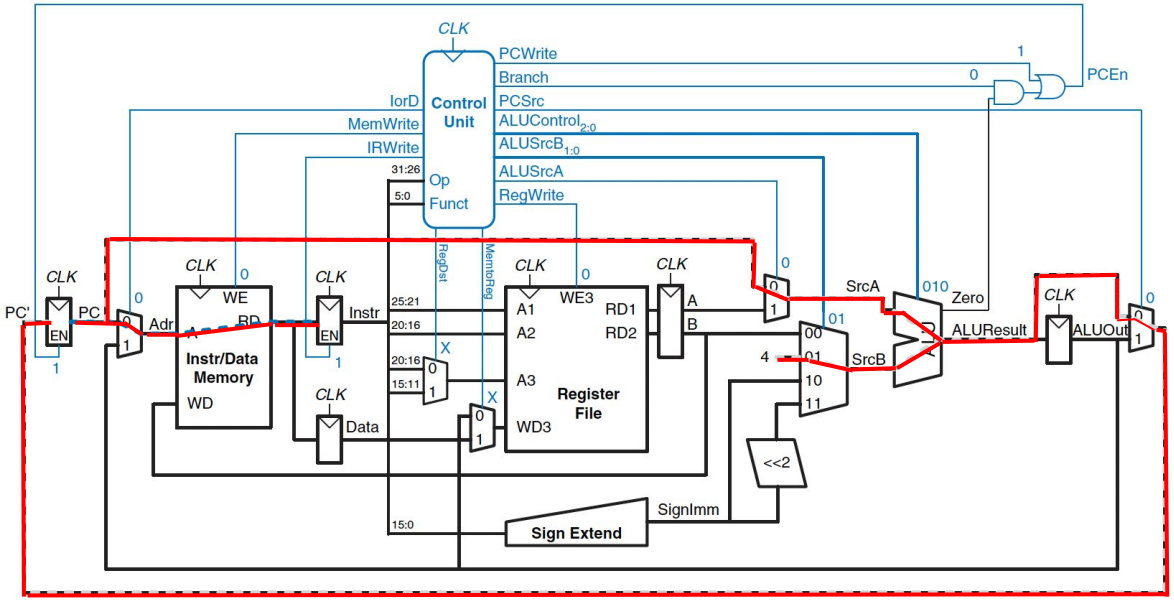
The sequence of control signal depends on the instruction being executed.

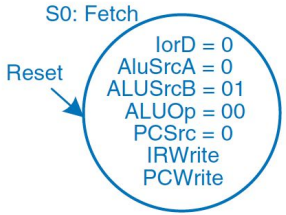
S0: Fetch



The FSM enters S0 on reset.

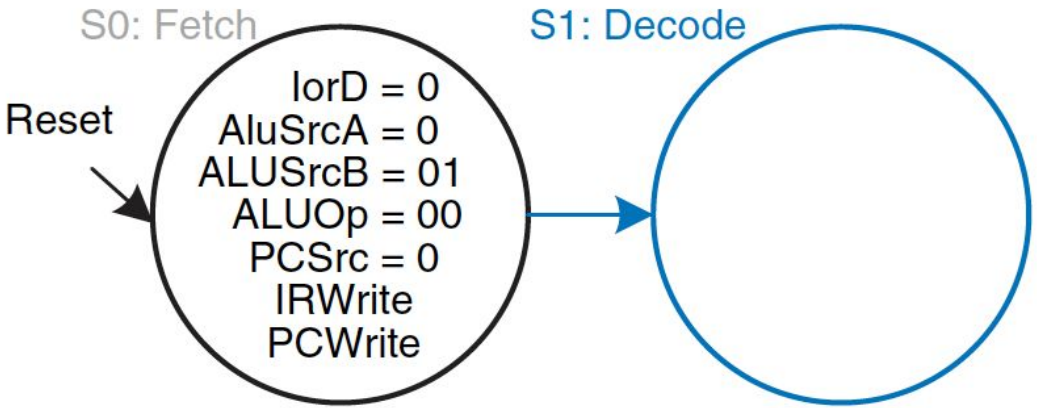
Data flow during the fetch step





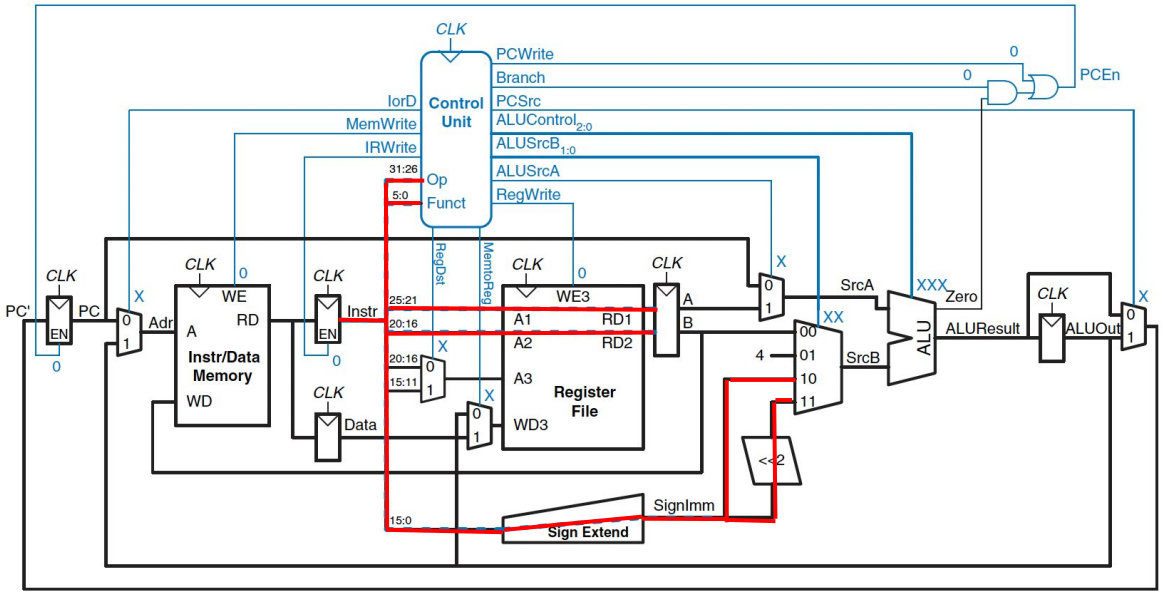
IorD=0: the address is taken from the PC.
IRWrite→1: the instruction is written into the IR.
ALUSrcA=0: SrcA = PC
ALUSrcB=01: SrcB = 4
ALUControl = 010: add the two inputs
PCSrc=0 and PCWrite→1: write PC+4 to the PC register.

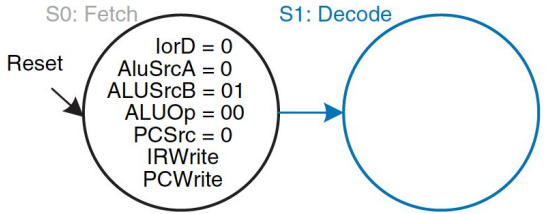
S1: Decoder



S0 ⇒ S1: it transfers after one clock cycle; no control signals are needed for the transition.

Data flow during the decode step

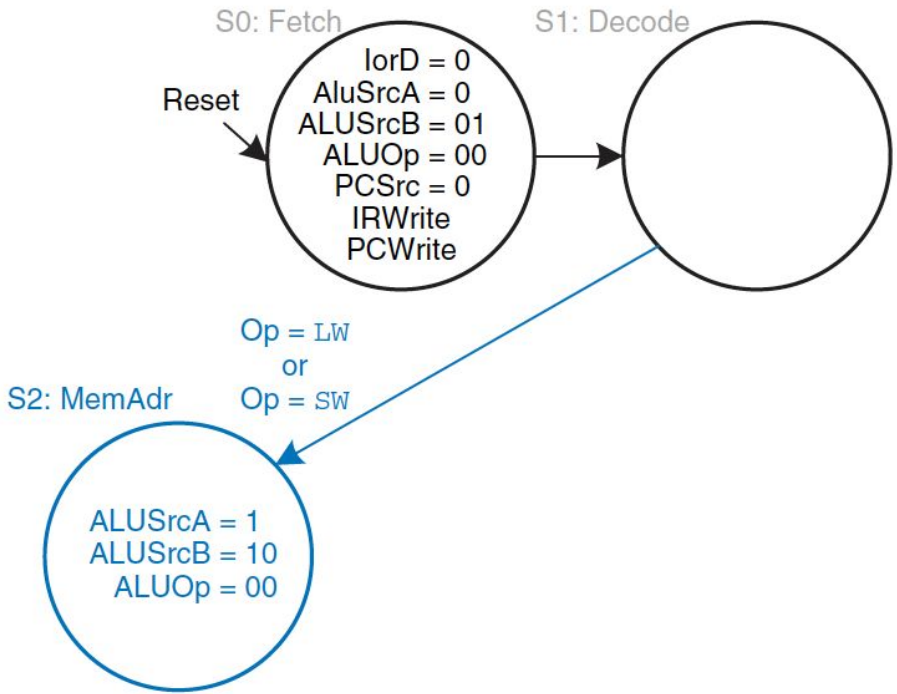




The next step is to read the register file and decode the instruction.

- The register file always reads the two source specified by rs and rt fields. At the same time, the immediate is sign-extended.
- The control unit examines the opcode and determines what to do in the next step.

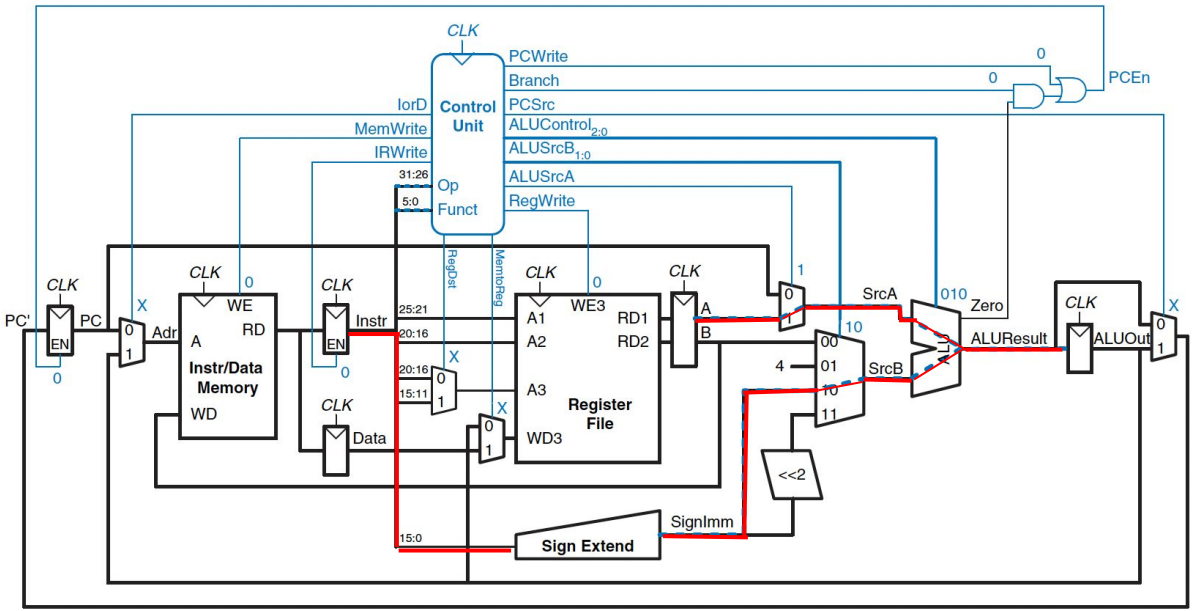
Memory Address Computation

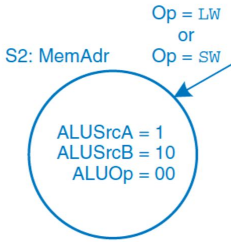


Now, there are several possible states, depending on the opcode.

S2 is memory address computation: If the instruction is a memory load or store (`lw` or `sw`), the processor computers the address by adding the base address to the sign-extended immediate.

Data flow during memory address computation



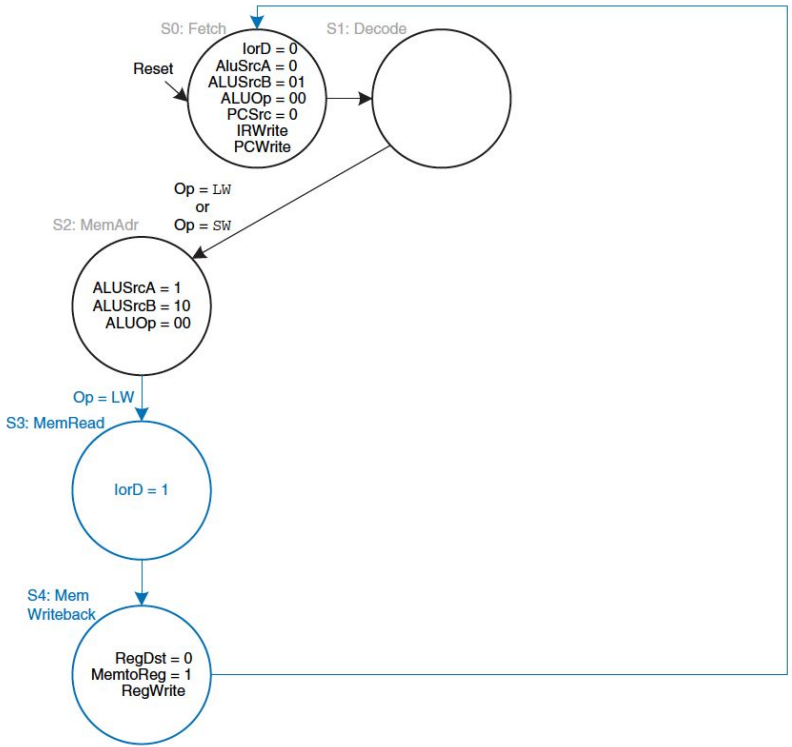


Adding the base address to the sign-extended immediate.

- ALUSrcA=1: select register A
- ALUSrcB=10: select SignImm
- ALUOp=00: ALU adds inputs

The effective address is stored in the ALUOut register for use on the next step.

Memory Read

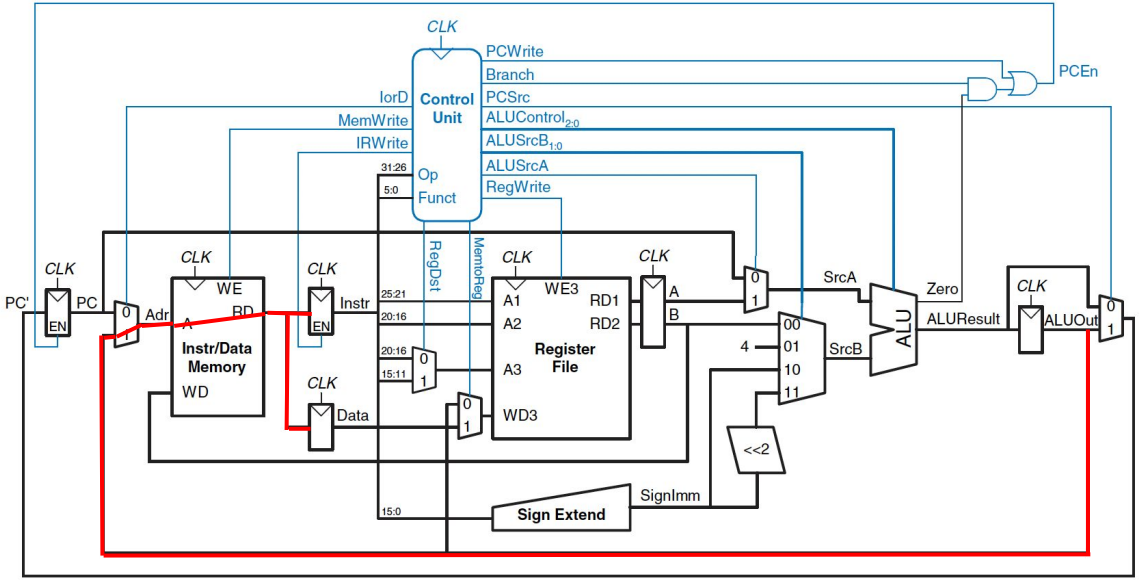


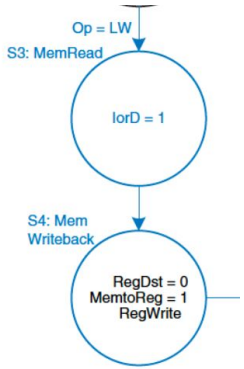
If the instruction is `lw`, the processor must next read data from memory and write it to the register file.

S3: the address in memory is read and saved in the Data register.

S4: Data is written to the register file.

Data flow during the Memory Read

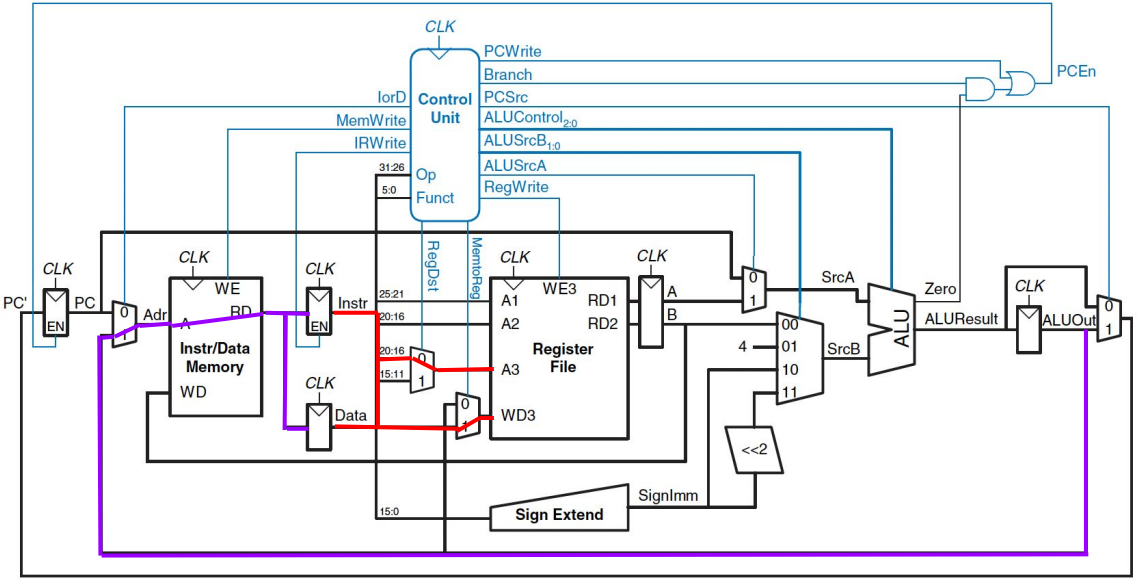


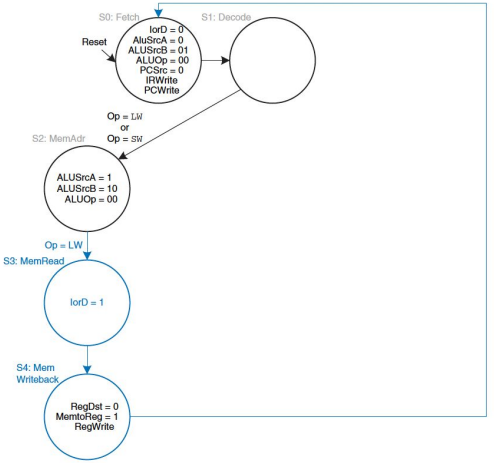


IorD = 1: select the memory address that was just computed and saved in ALUOut.

The address is read and saved in the Data register.

Data flow during the Writeback

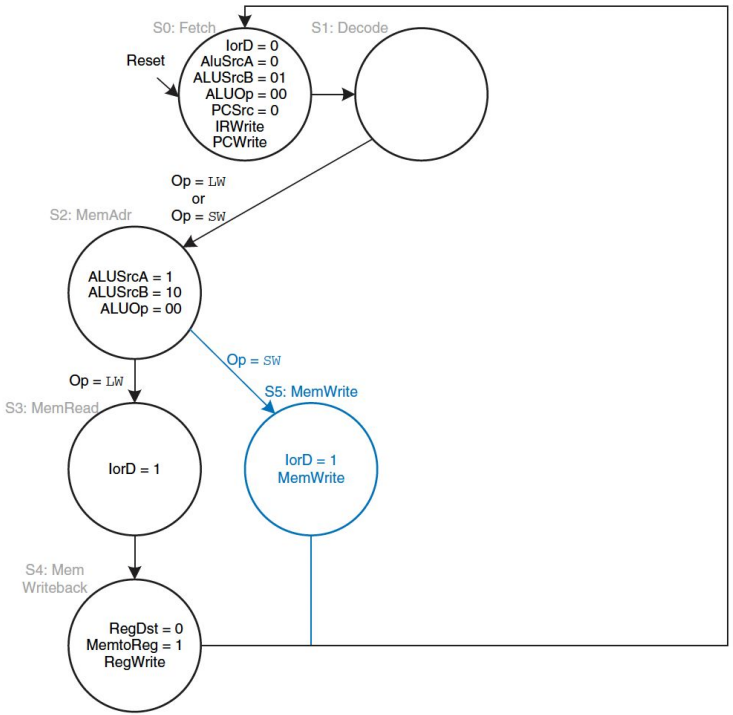




MemtoReg = 1: select *Data*
RegDst = 0: pull the destination register from the `rt` field

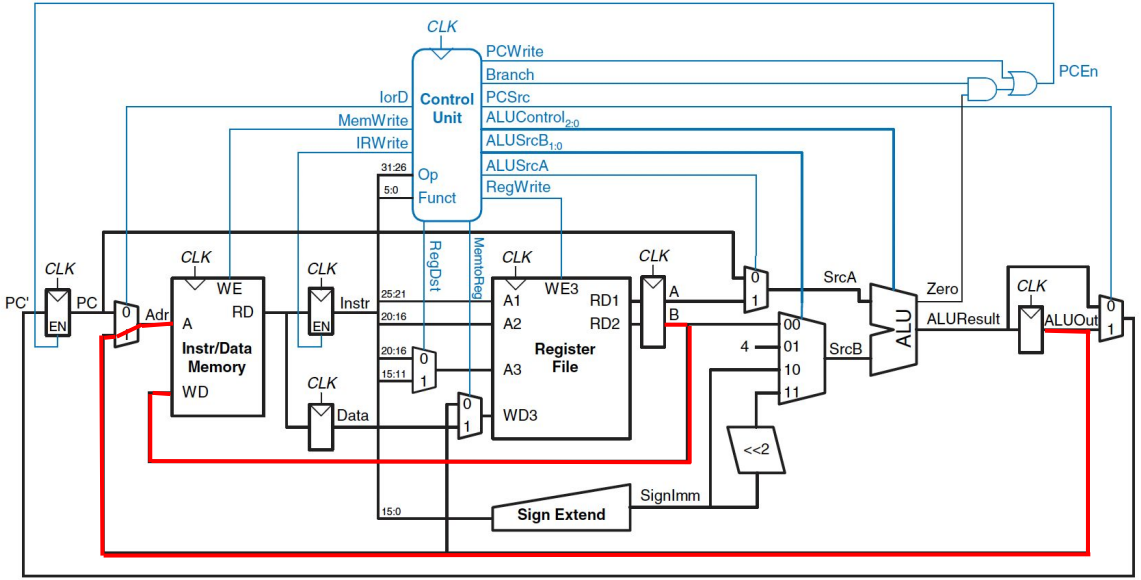
RegWrite→1: to perform the write, completing the `lw` instruction. So, the FSM returns to the initial state, S0.

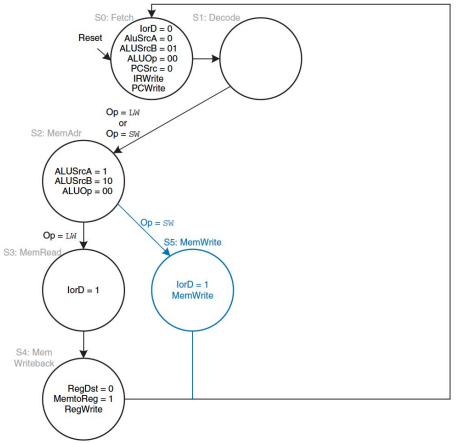
Memory Write



S5: If the instruction is `sw`, the data read from the second port of the register file is simply written to memory.

Data flow during the Memory Write

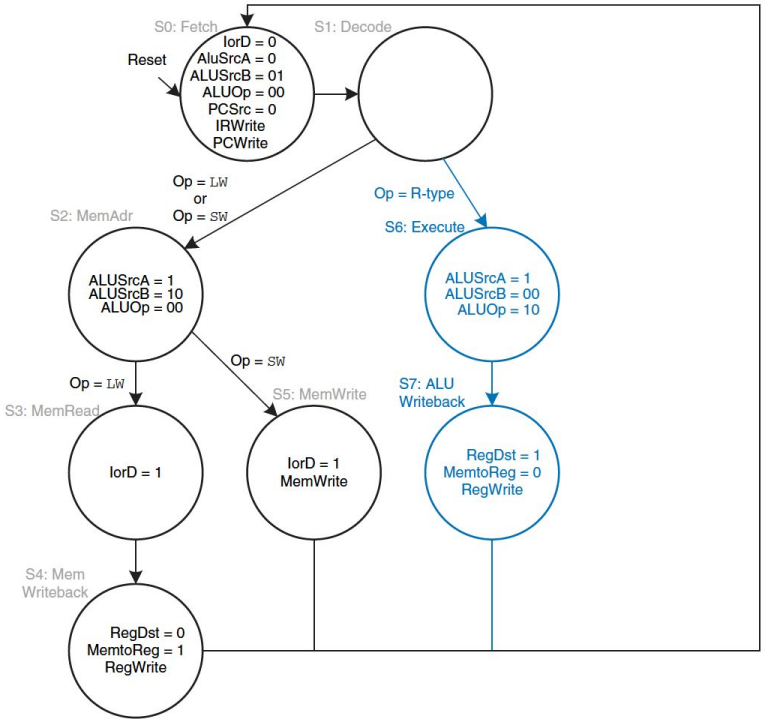




IorD = 1: select the address computed in S2 and saved in ALUOut.

MemWrite→1: write to the memory

Execute R-type operation

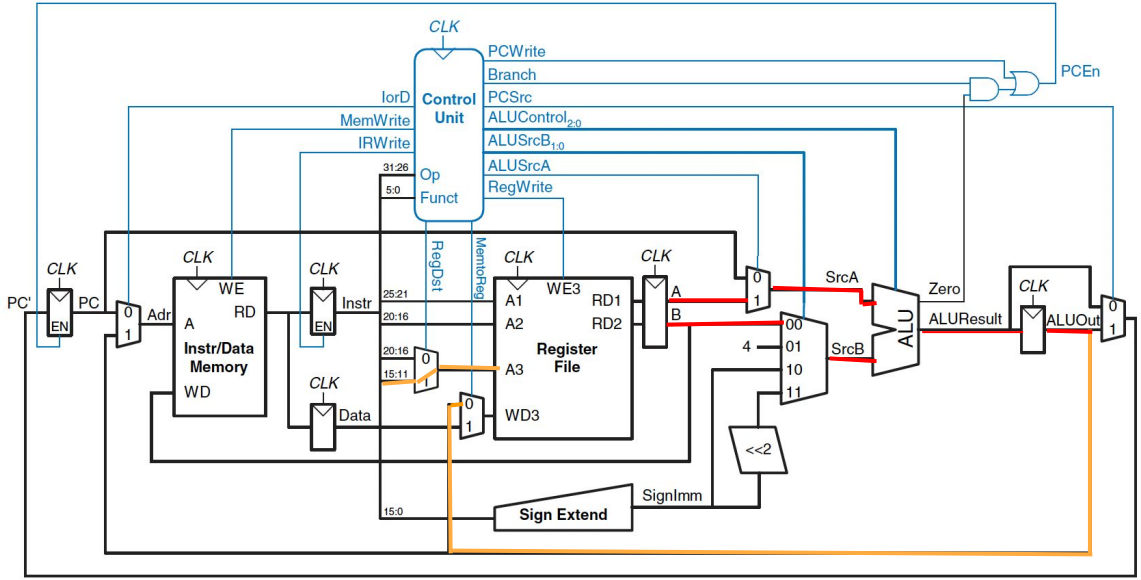


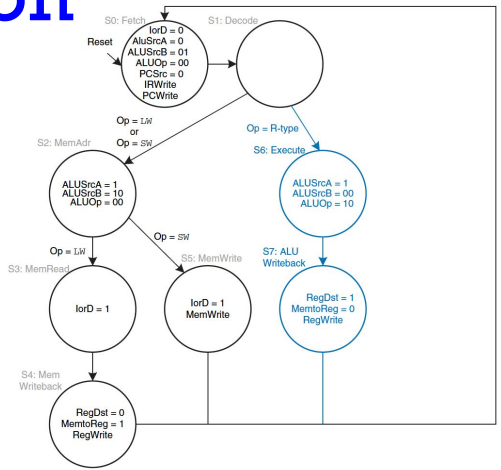
For an R-type instruction, the processor must calculate the result using the ALU and write that result to the register file.

S6: select the A and B registers and perform the ALU operation indicated by the `funct` field.

S7: write ALUOut to the register file.

Data flow during the R-type operation





ALUSrcA=1

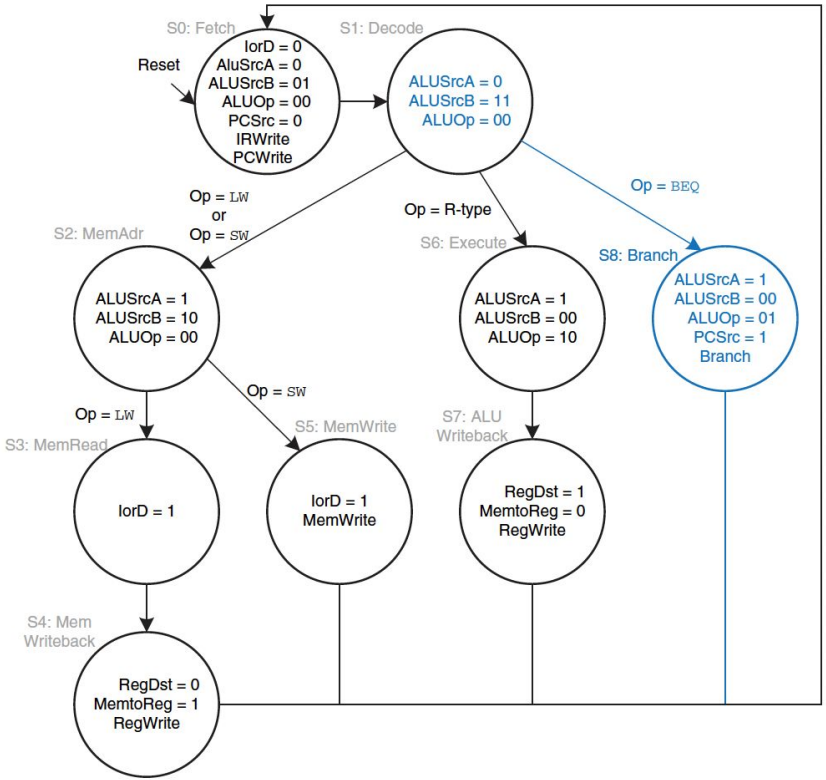
ALUSrcB=00

ALUOp=10: `funct` field determines the ALUControl (e.g., add, subtract, etc.)

RegDest=1: So, A3=Instr[15:11]

MemtoReg=0

Branch



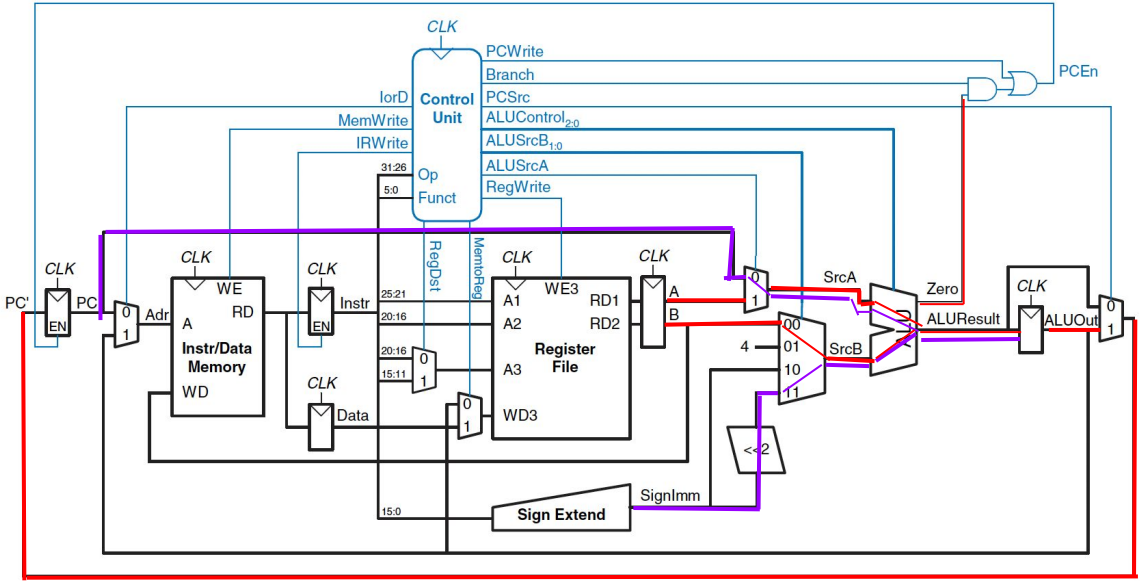
For a `beq` instruction, the processor must calculate the destination address and compare the two source registers to determine whether the branch should be taken.

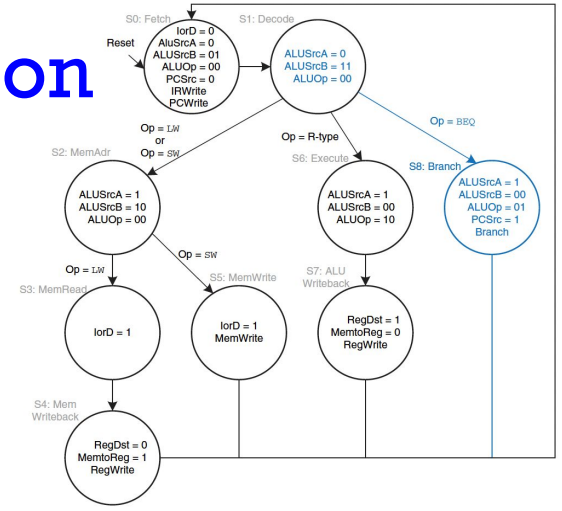
Since the PC register was updated to PC+4 during S0, the ALU was idle at S1. So, the processor can use ALU at S1 to add the incremented *PC* (i.e., *PC+4*) to *SignImm x 4*. Therefore, at S1, we set

- ALUSrcA=0
- ALUSrcB=11
- ALUOp=00

The result is stored in ALUOut. The computation result is harmless because it will not be used in subsequent cycles if the instruction is not `beq`.

Data flow during the beq instruction





ALUSrcA=1

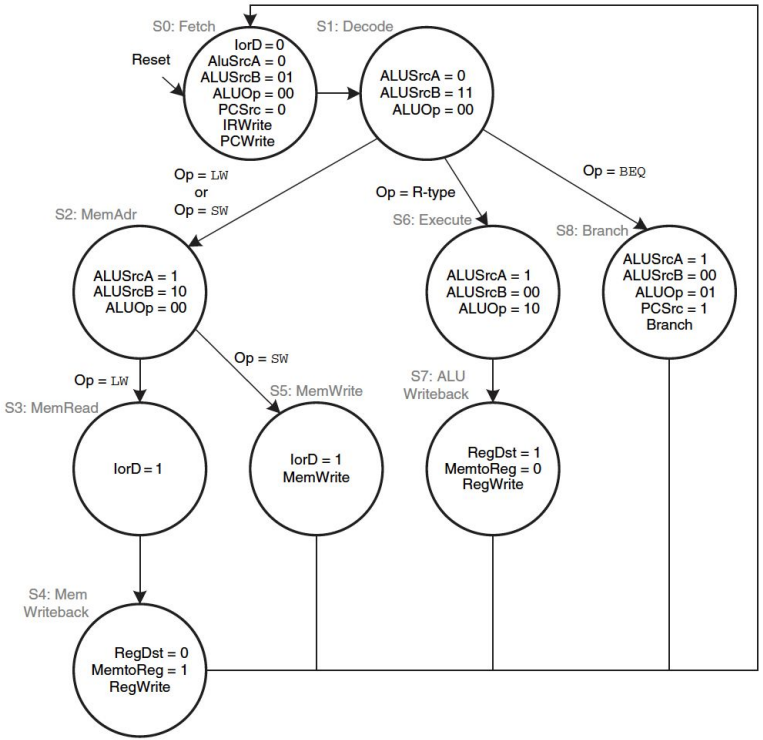
ALUSrcB=00

ALUOp=01: `subtraction`

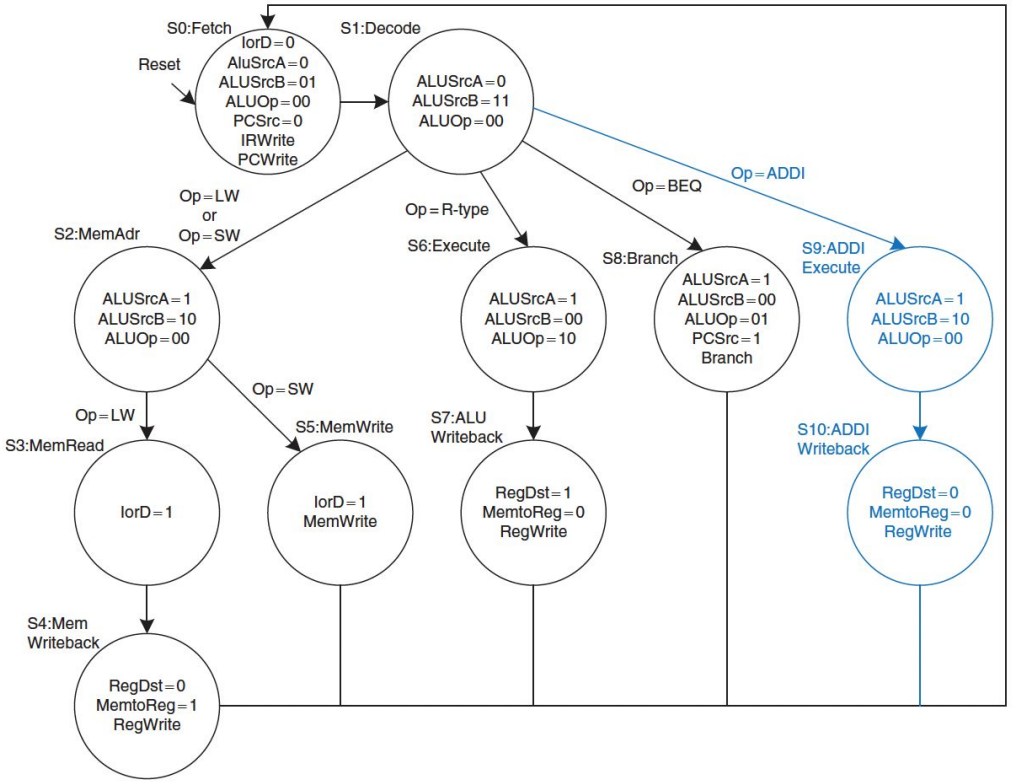
PCSrc=1: ALUOut is select for PC'

Branch→1: if Zero=1, PCEn→1;

Complete multicycle control FSM

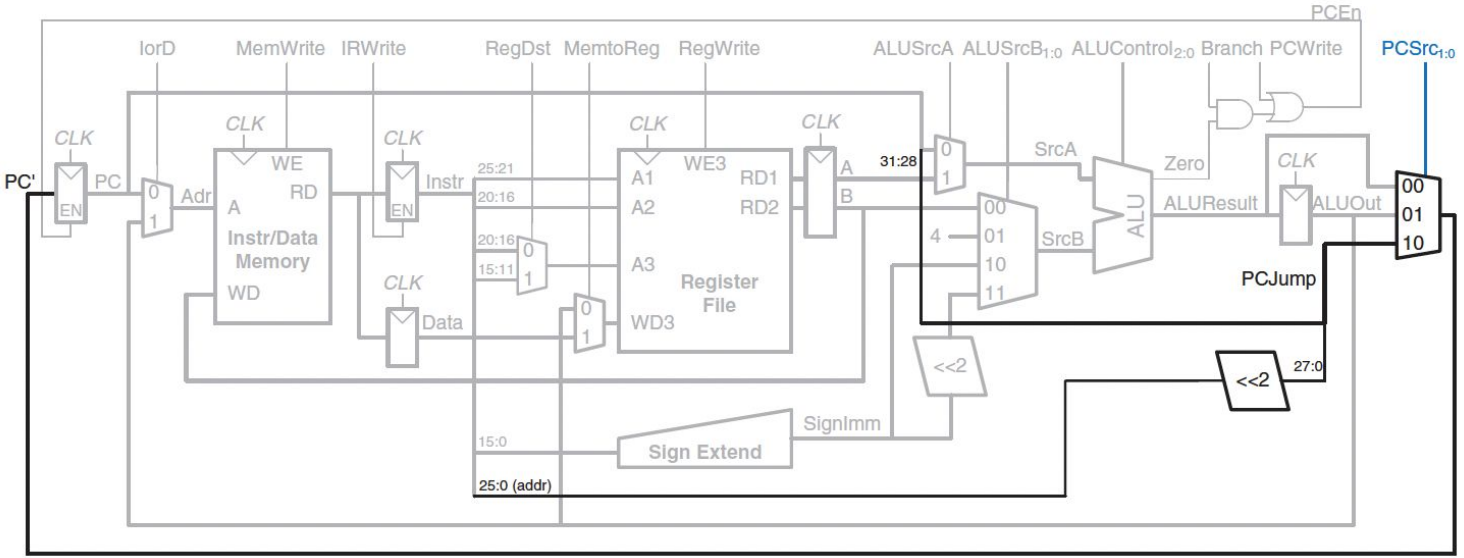


Main controller states for addi



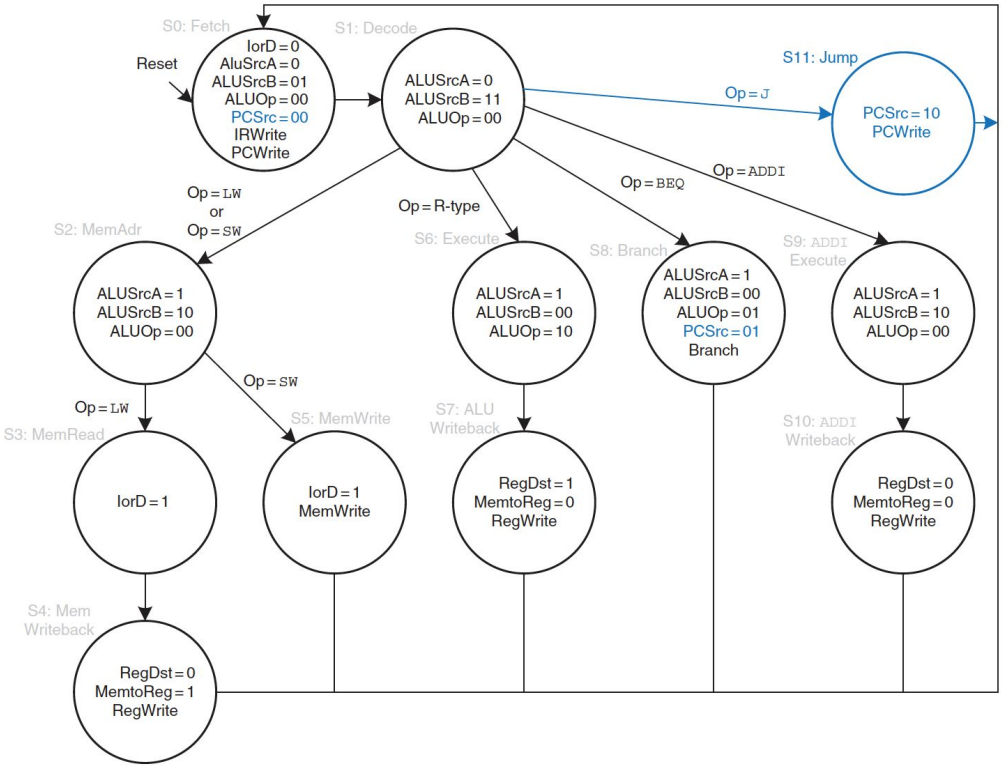
S9 and S2 are identical; they can be merged into one state.

Datapath to support j instruction



Incremented PC[31:28] merges the offset immediate x 4, then, go to PC'.

Main controller state for j



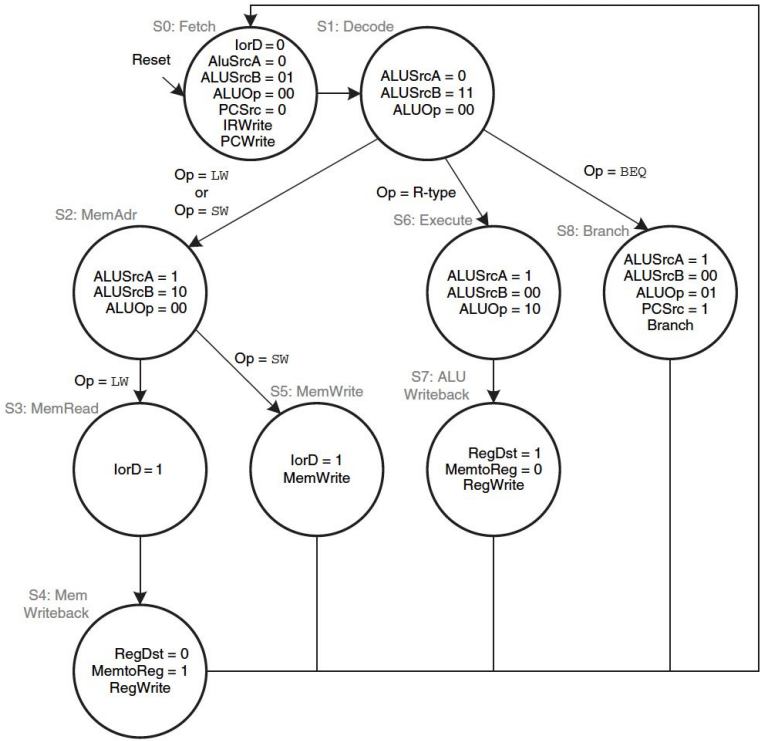
Performance analysis

The execution time of an instruction depends on both the number of cycles it uses and the cycle time.

- Multicycle processor uses varying number of cycles for the various instructions.
- Multicycle processor does less work in a single cycle and has a shorter cycle time.

Recall the single-cycle processor, all instructions are performed in one cycle, so the cycle time is determined by the instruction that has the most workload.

Performance analysis



Number of cycles required: (counting the states)

- `beq` and `j` instructions: 3 cycles;
- `sw`, `addi`, and `R-type` instructions: 4 cycles;
- `lw` instructions: 5 cycles;

Multicycle processor CPI (H&H example 7.7)

The SPECINT2000 benchmark consists of approximately 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R-type instructions. Determine the average CPI for this benchmark.

The cycles of different instructions are

- loads: 5, stores: 4, branches: 3, jumps: 3, R-type: 4

So, the averages CPI = (0.11+0.02)*3 + (0.52+0.10)*4+(0.25)*5 = 4.12.

Note: the worse case CPI is 5 if the benchmark contains only loads instructions.

Multicycle processor CPI

In a multicycle processor, each cycle involved **one** ALU operation, memory access, or register file access. Also, we assume that register file is faster than the memory and that writing memory is faster than reading memory.

So, there are two possible critical paths that would limit the cycle time:

$$T_c = t_{pcq} + t_{mux} + max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup}$$

The values of the times depend on the specific implementation technology.

(Note: t_{setup} is the time to set up (i.e., store) the register)

Processor performance comparison (H&H, example 7.8)

Compare the execution time of the multicycle processor and single-cycle processor for 100 billion instructions from the SPECINT2000 benchmark (see Example 7.7). The delays given in Table 7.6.

Table 7.6 Delays of circuit elements

Element	Parameter	Delay (ps)
register clk-to-Q	t_{pcq}	30
register setup	t_{setup}	20
multiplexer	t_{mux}	25
ALU	t_{ALU}	200
memory read	t_{mem}	250
register file read	t_{RFread}	150
register file setup	$t_{RFsetup}$	20

The cycle time of multicycle processor is

$T_{c2} = 30+25+250+20 = 325$ ps.

We know the CPI is 4.12 from Example 7.7, so, the total execution time is, $T_2 = (100 \times 10^9 \text{ instructions})(4.12 \text{ cycles/instruction})(325 \times 10^{-12} \text{ s/cycle}) = 133.9$ seconds.

According to the example in last class, we know the cycle time of a single-cycle processor is 925 ps, and the total execution time is 92.5 seconds.

Processor performance comparison

The multicycle processor can be slower than the single-cycle processor in the example.

- `lw` is broken into five steps, but the cycle time is not improved five-fold. It is partly because not all the steps are exactly the same length, and partly because the 50-ps sequencing overhead of the register clk-to-Q and setup time must be paid on every step, not just once for the entire instruction.
- The multicycle processor is like to be less expensive than the single-cycle processor because it eliminates two adders and combines the instruction and data memory into a single unit. (But it requires five nonarchitectural registers and additional multiplexers.)