Chapter 4

CPU Design

Reading: The corresponding chapter in the 2nd edition is Chapter 5, in the 3rd edition it is Chapter 5 and in the 4th edition it is Chapter 4.

4.1 Design goal

In Chapter 1, we learned that execution time (ET), one of the standard performance measures for a machine, depends on several parameters such as the number of instructions, the number of clock cycles per instruction (CPI), and the clock period (T). To increase the performance, we want the smallest possible ET, which can be achieved by optimizing the previous parameters. The number of instructions is mainly determined by the instruction set an architecture provides, as well as the compiler. In Chapter 2, we studied the MIPS R2000 instruction set and how it is designed to optimize performance. The CPI as well as the minimal $T = 1/f$ are determined by the specific implementation, in *hardware*, of the CPU and how the instructions (in the instruction set) are executed by the hardware.
In this chapter, we will present a basic implementation of the MIPS R2000 processor, focusing on the hardware that is required to implement the following subset of MIPS R2000 instructions:

- Memory access instructions: `lw`, `sw`
- Arithmetic and logical instructions: `add`, `addi`, `sub`, `and`, `or`
- Branch instructions: `beq`, `j`

We will approach the CPU design in two stages. First, we design the datapath, i.e., the set of building blocks (like, e.g., ALU, registers, multiplexers, etc.), as well as their connectivity, that is required to enable execution of every single instruction listed above, in hardware. In a second step, we will design the control logic, i.e., the "brains" of the CPU, which will ensure that the datapath components processes every instruction correctly. We want to optimize the design such that the overall performance of the CPU is high and the hardware complexity low.

4.2 An Abstract Implementation of a Datapath

We start with a high-level, abstract implementation of the *datapath*, showing the major functional units and their connectivity, depicted in Figure 4.1. Details on the memory structures (and their hierarchical design) will be covered in the next chapter. For now, memory is considered as a set of individually addressable memory cells, without providing any further detail. Also, remember that memory is NOT part of the CPU.

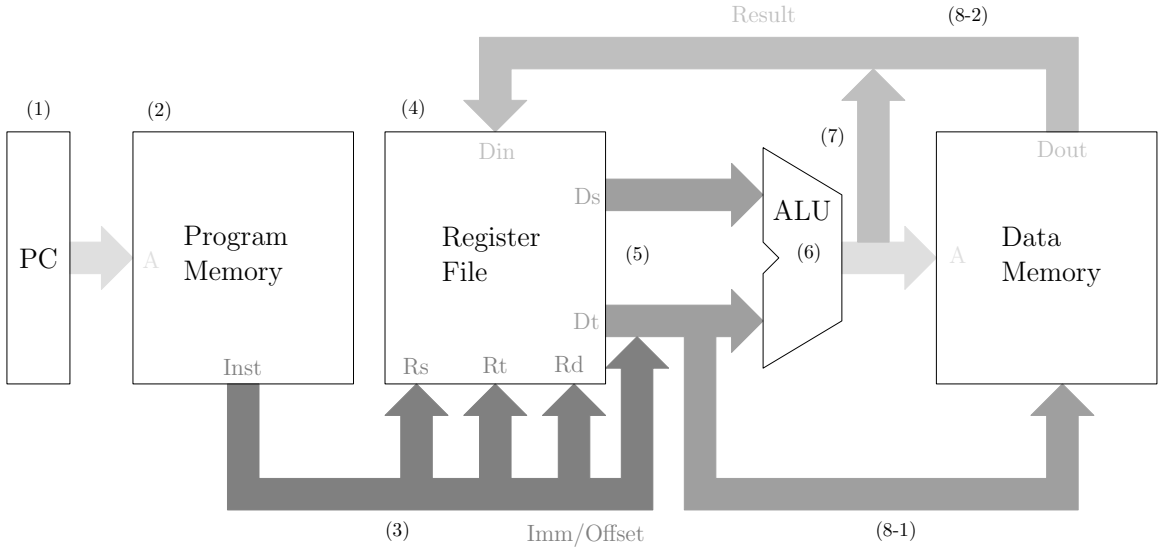

Figure 4.1: A first, abstract implementation of the data path.

The execution of every instruction starts with the PC register (1) supplying the address of the next instruction to the **Program Memory** (2), i.e., the memory that stores all of a program's instructions (specified in the `.text` segment of a program). As a result, the corresponding 32-bit instruction is provided at the output (3) of the program memory. This memory access is called the **instruction fetch**. Once an instruction has been fetched from program memory, we need to know on which registers the instruction will operate. This is determined by the Rs, Rt, and Rd fields in the instruction (3), each of which corresponds to a very specific 5-bit subset of the 32 bits that represent the instruction in (3). Each of the 5-bit fields Rs, Rt and Rd specifies one of the 32 general purpose CPU registers, all of which are grouped into one hardware component called the **Register File** (4), which is part of the CPU. The register file has the following functionality:

- the output Ds is set to the contents of the register specified by input Rs (5 bits);
- the output Dt is set to the contents of the register specified by input Rt (5 bits);
- the input Din is stored in the register specified by input Rd (5 bits), at the next rising clock edge.

Depending on the type of instruction, the contents of one or two source registers will be fetched and made available at the Ds and/or Dt output(s) (5) of the register file. Next, the ALU operates on its inputs (6), depending on the instruction that is being executed:

- For `lw`, `sw`: compute the memory address, from Ds and the 16-bit offset specified in the instruction's immediate field (3)
- For `add`, `addi`, `sub`, `and`, `or`: compute the arithmetic or logic result from Ds and Dt or the 16-bit constant specified in the instruction's immediate field (3)
- For `beq`: compare Ds and Dt
- For `j`: no ALU operation required

For R-type instructions, both inputs of the ALU are provided by the register file. For I-type instructions (e.g., `addi`), the value specified in the immediate field of the instruction (3) is provided as the second input to the ALU. For arithmetic and logic operations, the result computed by the ALU is stored in one of the registers in the register file (7), as specified by Rt or Rd in (3). For a `lw` or `sw` instruction, the value computed by the ALU is used as a memory address, to access the **Data Memory**, i.e., the memory that stores all of a program's data (specified in the `.data` segment of a program). It specifies the memory location in data memory where the `lw` instruction will load data from (8-2), into the register in the register file specified by Rt, or where the `sw` instruction will store data to (8-1), from the register in the register file specified by Rt, the contents of which is provided at Dt (5).

We will now examine in more detail how to build the datapath, by examining the components and connections that each type of instruction requires, one at the time.

4.2.1 Instruction Fetch

Since every instruction is stored in the program memory, an **instruction fetch** step is required for every instruction. The PC register directly addresses the program memory through its address input A. Updating PC, so it contains the address of the next instruction as the program executes, requires incrementing PC by 4 in most cases. A simple adder can be used to implement this. Figure 4.2 illustrates the architecture.

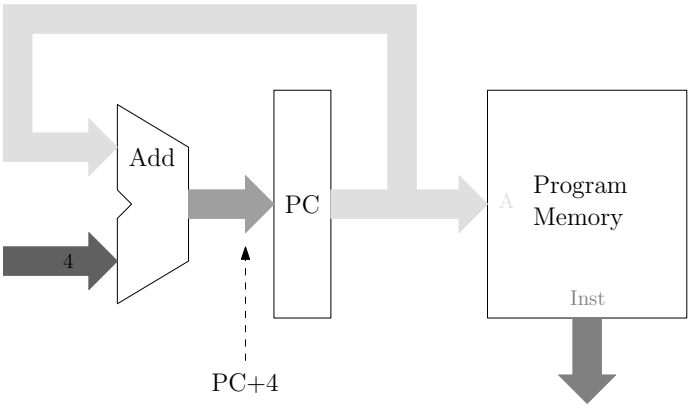

Figure 4.2: Datapath for instruction fetch.

4.2.2 Arithmetic and logic instructions

The execution of an arithmetic or logic instruction will depend on the *instruction format*, i.e., whether we have an R-type or I-type instruction.

R-type instruction

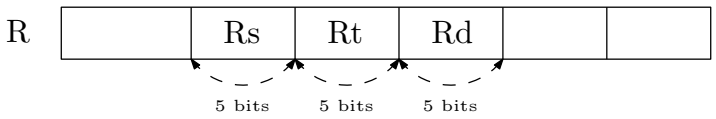

Figure 4.3: Format of an R-type instruction.

For an R-type instruction, the Rs and Rt fields specify the source operands, while Rd specifies the destination register. The 5 bits corresponding to Rs specify one of the 32 registers in the register file; the contents of the register encoded by Rs are made available at Ds. Similarly, the contents of the register specified by Rt will be made available at the output Dt of the register file. Next, Ds and Dt are fed into the ALU, which executes the arithmetic or logic operation specified in the instruction. Finally, the result computed by the ALU is applied at Din, for storage in the register file. The 5 bits of the Rd instruction field determine which register will store the data appearing at Din. Figure 4.4 implements the datapath for an R-type arithmetic or logic instruction.

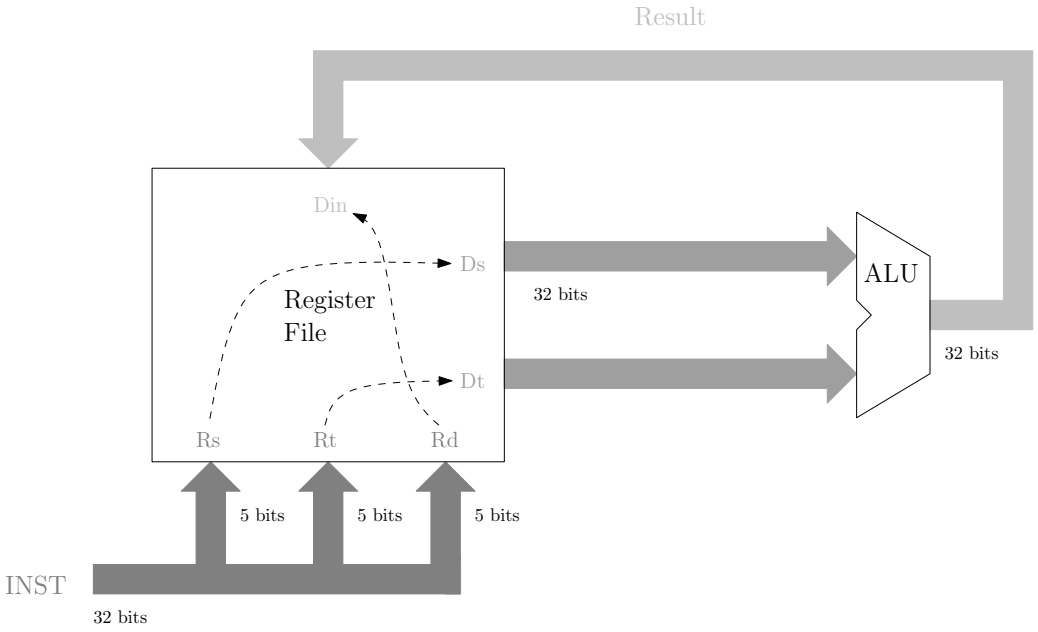

Figure 4.4: Datapath for R-type arithmetic or logic instruction.

I-type instruction

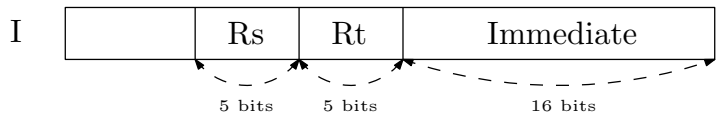

Figure 4.5: Format of an I-type instruction.

For an I-type instruction, Rs and the immediate field specify the source operands, while Rt specifies the destination register. The 5 Rs bits are processed as for an R-type instruction, and the corresponding register value will be made available at the Ds output of the register file. The value specified in the immediate field of the instruction makes the second input of the ALU. Since the immediate field of an I-type instruction consists of 16 bits, while the ALU input should be 32 bits, the immediate field needs to be *sign-extended* (SE) or *zero-padded* (ZP) to create the equivalent 32-bit two's complement respectively natural number representation, before being applied as an input to the ALU. After the ALU executes the arithmetic or logic operation specified in the instruction, the result from the ALU is stored in the register file, by applying it to Din, in the register specified by the Rt field (not Rd, in this case). Figure 4.6 implements the datapath for an I-type arithmetic or logic instruction.

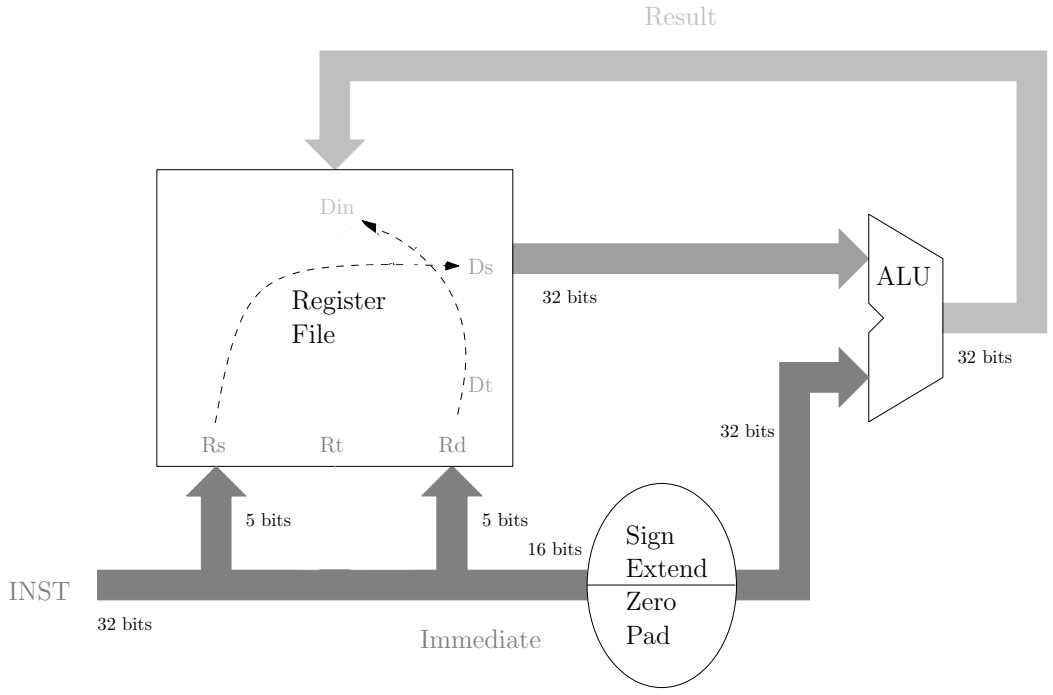

Figure 4.6: Datapath for I-type arithmetic or logic instruction.

Putting Figures 4.4 and 4.6 together, the datapath for arithmetic and logic instructions is shown in Figure 4.7.

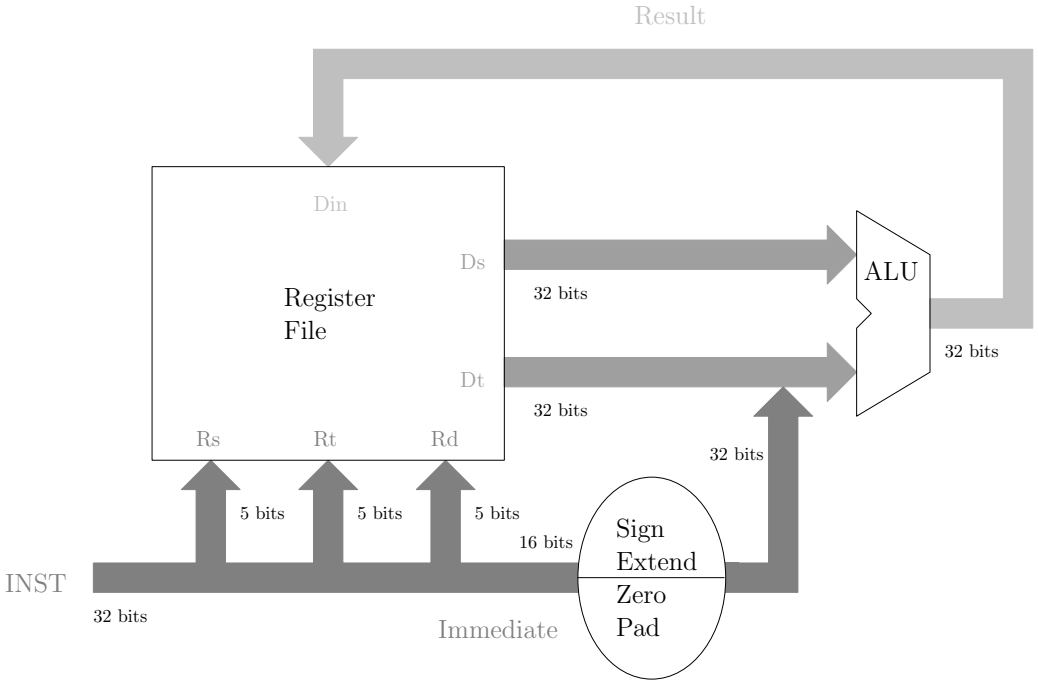

Figure 4.7: Datapath for arithmetic and logic instructions.

From the previous discussion, it is clear that Ds and Dt will always provide the contents of the registers specified by Rs and Rt. For an I-type instruction, Dt will simply get ignored and, instead, the instruction's immediate field will provide the second ALU input (this will require a multiplexer).

4.2.3 Load and store instructions

The instructions `lw` and `sw` use the instruction's Rs and immediate fields to compute the data memory address that will be accessed by the instruction. The instruction format is shown in Figure 4.8.

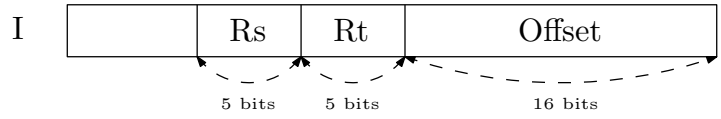

Figure 4.8: Format of a load or store instruction.

The memory address is computed by adding the contents of the register specified by Rs (provided at Ds) and the offset specified by the 16-bit immediate field, which needs to be *sign-extended* (SE) to create a 32-bit two's complement number and provide the correct

input for the ALU. Given these inputs, the ALU computes the memory address and applies it to the address input A of the data memory. The next steps depend on whether a `lw` or `sw` instruction is being executed:

`lw` instruction

The data at memory address A is provided by the data memory at Dout and applied as input Din to the register file, to be stored in one of the registers. The register that will store the data applied at Din is specified by the instruction's Rt field: Rt represents the destination register for a `lw` instruction. Figure 4.9 shows the datapath for the `lw` instruction.

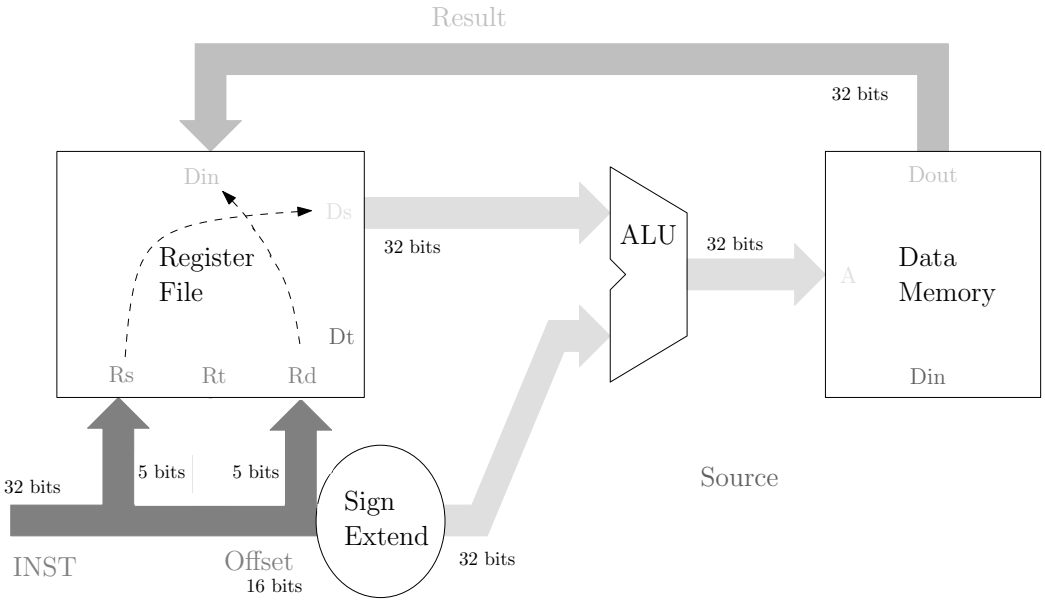

Figure 4.9: Datapath for `lw`.

`sw` instruction

For a `sw` instruction, the instruction's Rt field represents the source register, the contents of which are to be stored at memory address A (computed by the ALU) in the data memory. Indeed, Dt provides the contents of register Rt which are then stored in the data memory, at address A, by applying Dt as input Din to the data memory. This is shown in Figure 4.10.

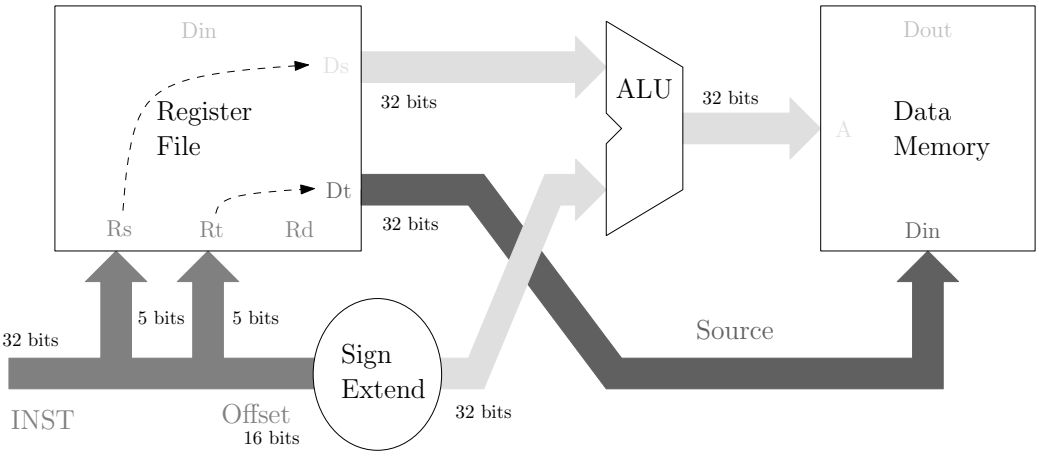

Figure 4.10: Datapath for `sw`.

Putting Figures 4.9 and 4.10 together, the datapath for load and store instructions is shown in Figure 4.11.

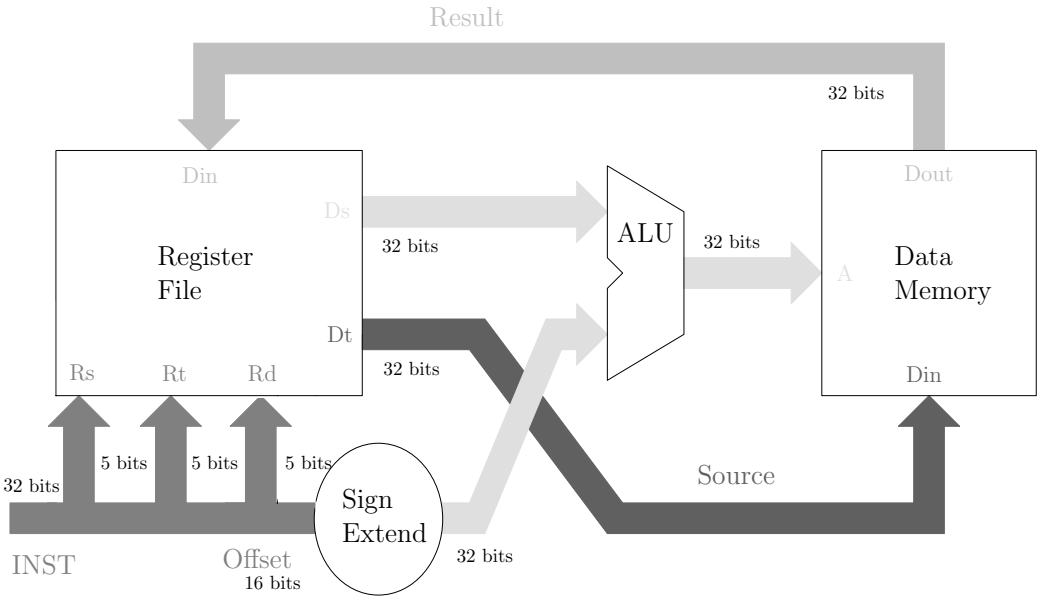

Figure 4.11: Datapath for load and store instructions.

Note, again, that Ds and Dt will always provide the contents of the registers specified by Rs and Rt. For a `lw` instruction, Dt will simply get ignored, since we will be reading from and not writing into the data memory.

4.2.4 The conditional branching instruction `beq`

The conditional branching instruction `beq` is an I-type instruction, formatted as shown in Figure 4.12.

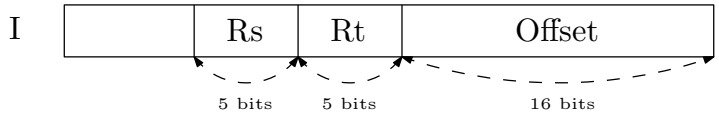

Figure 4.12: Format of the `beq` instruction.

The 5-bit Rs and Rt fields specify the registers involved in the branching condition. The contents of the registers Rs and Rt are provided at Ds and Dt respectively and the ALU compares them by computing a subtraction, checking whether the result is zero and generating a *zero flag*: if the ALU output is 0 (i.e., if Ds is equal to Dt), then the flag is set, otherwise (i.e., Ds not equal to Dt) the flag is 0. If the zero flag is set, the CPU should continue with the instruction at the branch target address, determined by the offset value in the instruction's immediate field. As we learned before, the branch target address is computed as $(\text{PC} + 4) + (\text{offset} \times 4)$, where the offset is specified by the last 16 bits of the instruction. This requires to sign extend the 16-bit offset to a 32-bit offset, a 2-bit left shift to multiply by 4 and adding PC+4, as depicted in Figure 4.13. The result of the branch address computation is fed into the PC register if and only if the zero flag is equal to 1. If the zero flag is 0, the CPU continues with the next instruction and PC gets updated as discussed before, i.e., to PC+4. Figure 4.13 shows the datapath for the `beq` instruction.

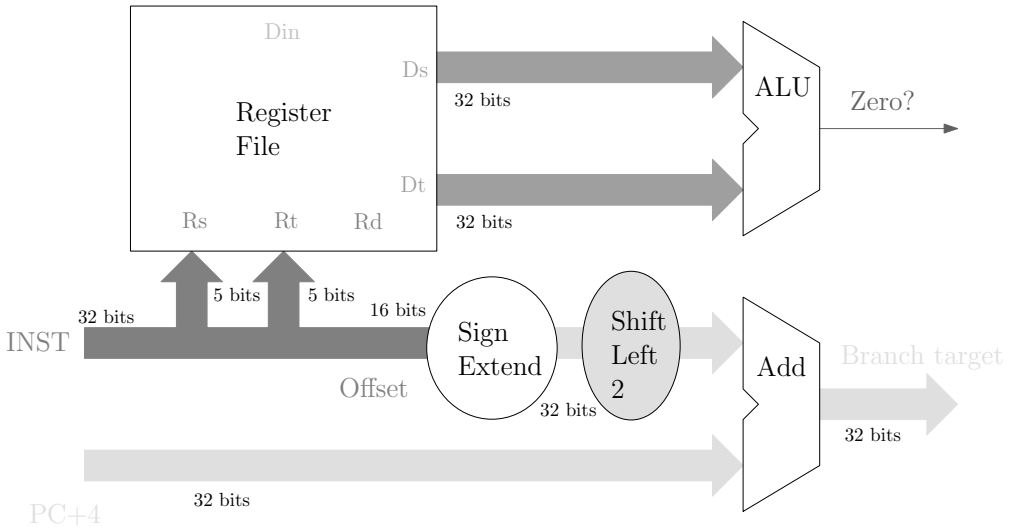

Figure 4.13: Datapath for `beq` instruction.

It is the control logic of the CPU (which we haven't discussed yet) that processes the value of the zero flag and then determines whether the branch is taken or not, i.e., whether

the next value of PC should be $(\text{PC} + 4) + (\text{offset} \times 4)$ or PC + 4 (using a multiplexer, to determine what value is clocked into PC). Note that all computations happen in parallel, so, even if it turns out branching is not required, $(\text{PC}+4)+(\text{offset}\times 4)$ will still be computed.

4.2.5 Putting it all together

We have examined the implementation of the datapath instruction by instruction. Putting together all the pieces, presented in Figures 4.2, 4.7, 4.11 and 4.13, we obtain the datapath shown in Figure 4.14.

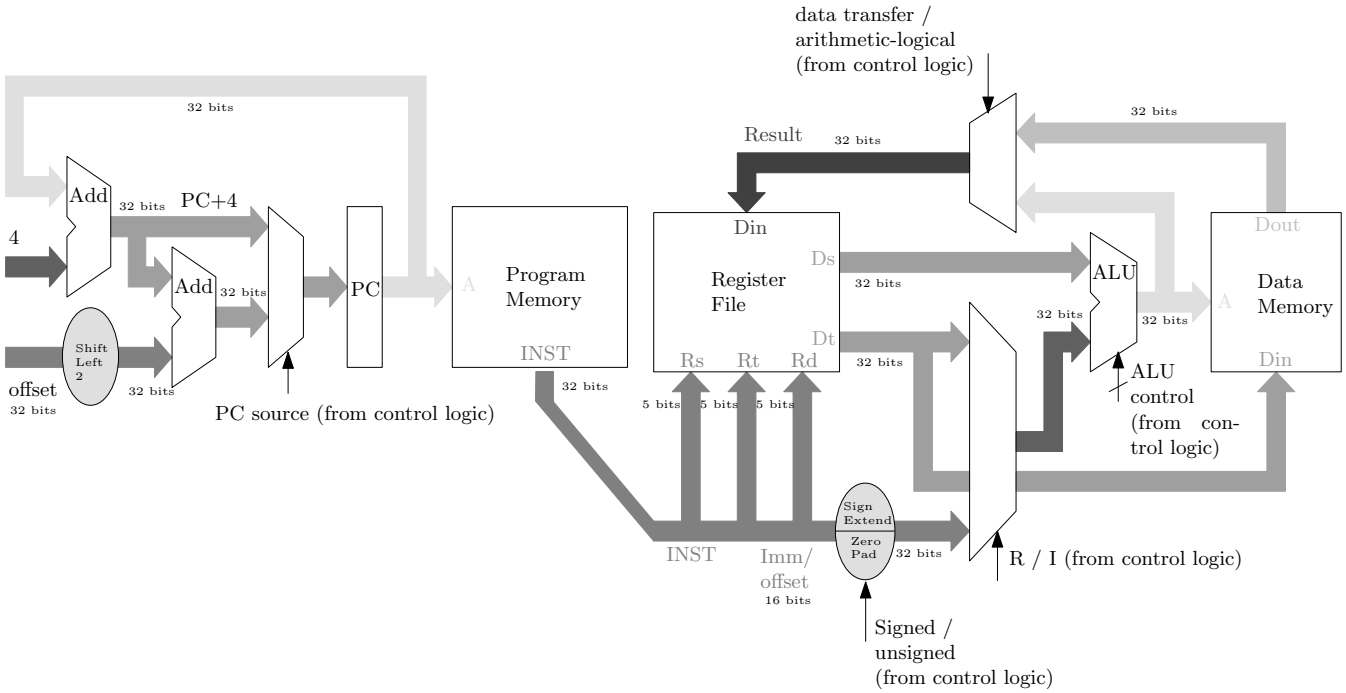

Figure 4.14: Putting all pieces together: the datapath.

This datapath has the necessary components and connections to execute the instructions `add`, `sub`, `or`, `and`, `addi`, `beq`, `j`, `lw`, and `sw` (we did not cover the jump instruction `j` yet but will see later that it is easy to add). Looking at the big picture, it is clear that:

- The next value of PC can be PC + 4 or $(\text{PC} + 4) + (\text{offset} \times 4)$ (for taken branch). Later, when we cover the `j` instruction, we will add a third possibility, that replaces the lower 28 bits of PC with the lower 26 bits of the instruction, shifted 2 bits to the left.

- The ALU takes two 32-bit source operands. The first operand is always from Ds and therefore specified by the instruction's Rs field. The second operand is from Dt, specified by Rt, for an R-type instruction and it is the sign extended or zero-padded immediate/offset value for an I-type instruction.

- The result (either from the ALU or from data memory) is stored in the register specified by Rd (derived from the Rd field for R-type instructions and the Rt field for I-type instructions).
- The data memory input for a `sw` instruction comes from the register specified by Rt (provided by Dt).

4.3 Single-cycle versus Multi-cycle

4.3.1 Single-cycle implementation

In a single-cycle implementation, every instruction takes one clock cycle to execute. The resulting CPI is 1 and the clock period is determined by the longest possible path, i.e., the worst-case delay, in the CPU's datapath. Let's examine the different instructions to assess this and determine the clock period T.

Instruction	Functional units - I(nstruction), R(egister file), M(emory)				
R-type	I Fetch	R Access	ALU	R Access	
`lw`	I Fetch	R Access	ALU	M Access	R Access
`sw`	I Fetch	R Access	ALU	M Access	
`beq`	I Fetch	R Access	ALU		
`j`	I Fetch				

Suppose that memory access takes 2ns, an ALU computation takes 2ns, and register file access takes 1ns. The time required to execute the different functional units for each instruction is then given in the following table.

Instruction	Time (ns)					Total
R-format	2	1	2	1		6ns
`lw`	2	1	2	2	1	8ns
`sw`	2	1	2	2		7ns
`beq`	2	1	2			5ns
`j`	2					2ns

So, the worst-case delay is $2 + 1 + 2 + 2 + 1 = 8$ns, for the `lw` instruction. Therefore, the clock period needs to be at least 8ns, plus the register set up time and register delay. Thus, when executing a `j` instruction, the CPU is not doing anything useful for about 6ns, because the clock period is determined by the instruction with the *longest delay*. Moreover, since every instruction is executed in one single clock cycle, every datapath component is needed simultaneously and no resource sharing is possible.

4.3.2 Multi-cycle implementation

Instead of allowing just one single clock cycle for the execution of each and any instruction (resulting in a significant amount of time loss for short, efficient instructions), a multi-cycle implementation allows the execution of a single instruction to take multiple clock cycles, as follows:

- Break each instruction into different functional *steps* (as illustrated, e.g., in the previous table)

- Allow one clock cycle to execute one step

Therefore, each instruction will now take multiple clock cycles (i.e., steps) to execute. Efficient instructions will take fewer clock cycles and more complex instructions will take more clock cycles. However, once a short, efficient instruction has executed its few steps in a few clock cycles, the first step of the next instruction can start execution immediately after, in the next clock cycle. This limits the amount of time the CPU is doing "nothing" and enhances the overall CPU performance.

Increasing performance

Clearly, the resulting CPI will be larger than 1 (and different for each type of instruction). However, the clock period T is much shorter than for a single-cycle implementation (T must allow enough time for the worst-case step to be executed, i.e., the step that takes the longest time to execute, which is much shorter than the worst-case delay for an entire instruction). This improves the overall execution time (and, thus, the CPU performance) if some instructions take fewer number of clock cycles, to execute, than the maximum.

Sharing resources

As mentioned before, a single-cycle implementation requires simultaneous action of every datapath component, which prevents any form of resource sharing. For a multi-cycle implementation, however, not every datapath component is needed in every step of the instruction, i.e., in every clock cycle. Therefore, some of the datapath resources can be shared amongst the different functional steps. For example:

- Since load and store instructions require simultaneous program and data access in a single-cycle implementation, two distinct memory components, one for the program and one for data, were required. A multi-cycle implementation, on the other hand, allows one, shared memory for program and data. Indeed, in a multi-cycle implementation of load and store instructions, the CPU fetches an instruction from program memory during one clock cycle (i.e., step), while it interacts with the data memory in another clock cycle (i.e., step). Since program and data memory are accessed in different clock cycles, we can merge them into one memory component, that allows to access both the program and data.

- A single ALU can be used for computing the new PC value, as well as for data computation, by scheduling these operations in different steps, i.e., clock cycles. This avoids having to provide two separate adders for PC computation (which, again, was required for a single-cycle implementation, since all adders were used simultaneously, i.e., in the same single clock cycle that executed an instruction).

Breaking up the datapath with registers

Remember that electric signals propagate through combinational logic until they reach the input of a register (i.e., flip-flop), where they "wait" for the next sampling edge of the clock. When a sampling edge of the clock arrives, signals are "latched" or "clocked" into the registers and the output of the registers gets updated. Once updated, the outputs of the registers do not change till the next sampling edge of the clock. Meanwhile, again, the new output signals propagate through any combinational logic following the register output, until they reach the next register. This is depicted in Figure 4.15.

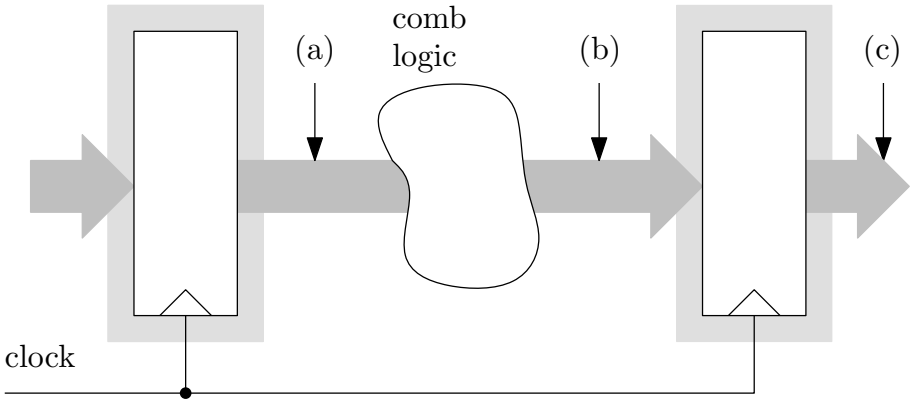

Figure 4.15: Starting at (a), signals propagate through combinational logic. They "wait" for the rising clock edge when they reach a register input, in (b), and, once latched into the register, they continue their journey in (c).

The only way to break each instruction into different functional steps, for a multi-cycle implementation, is by a) breaking up the datapath into "sectors" corresponding to each of the steps and b) ensuring that computation (i.e., electric signal propagation) is restricted to a sector during each clock cycle (i.e., electric signals cannot propagate into the next step's sector until the next clock cycle). From the discussion above, it is clear that this can be achieved by inserting additional registers between these "sectors", as "barriers" for electric signal propagation within one clock cycle and to store intermediate results. This is shown in Figure 4.16.

Furthermore, taking into account the resource sharing mentioned above, we can reduce the hardware complexity significantly. The resulting hardware implementation of the datapath is shown in more detail in Figure 4.17.

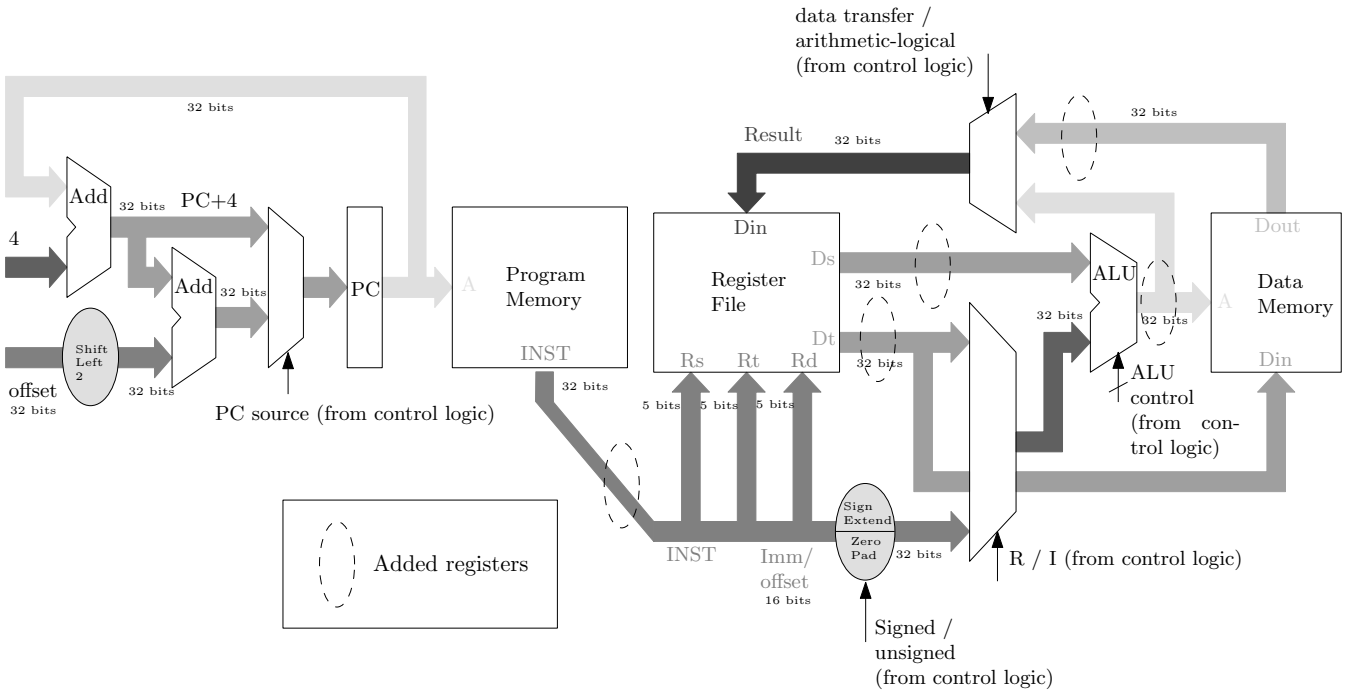

Figure 4.16: Adding registers to enable a multi-cycle implementation.

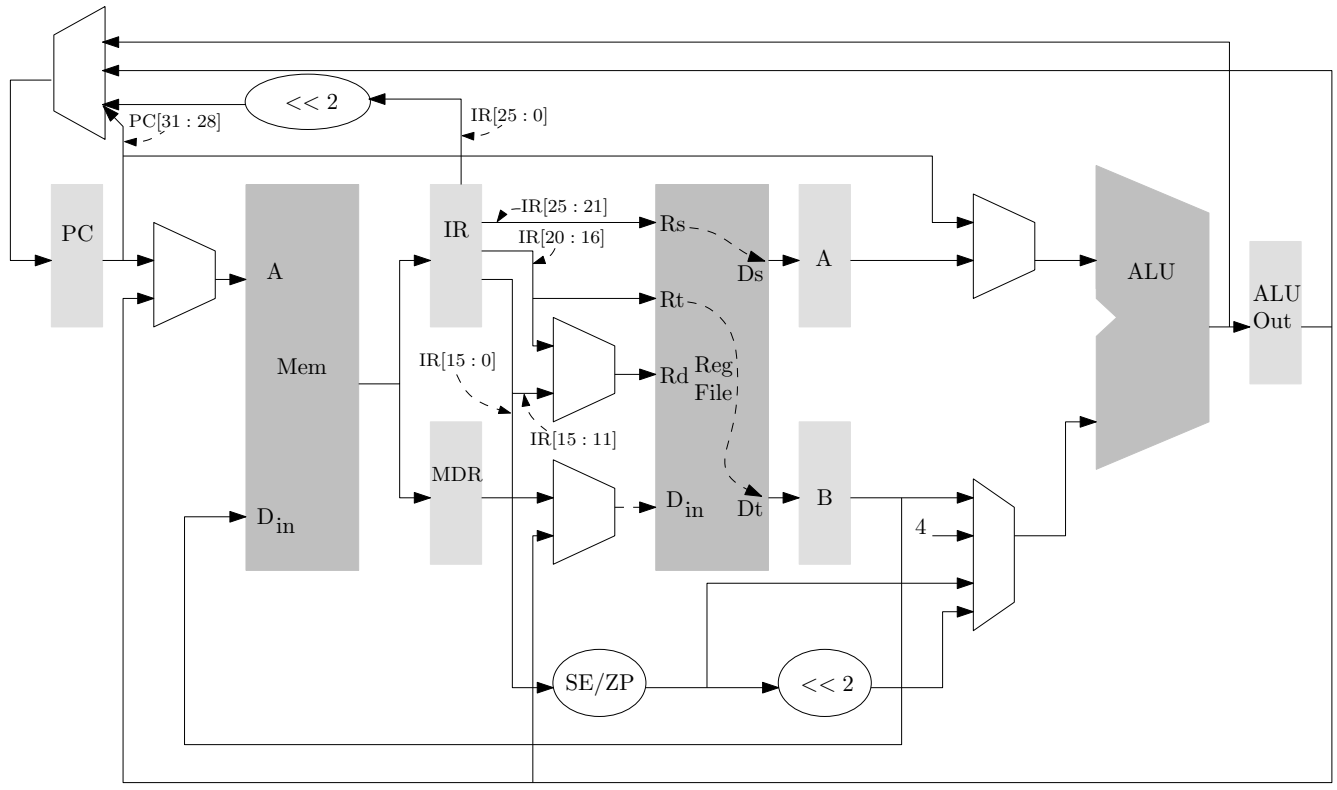

Figure 4.17: Multi-cycle implementation of the datapath, in detail.

In Figure 4.17, five additional registers enable a multi-cycle implementation.

- The instruction register (IR) stores the instruction we're currently executing (provided by the memory, when fetching the instruction at the address provided by PC). This allows reading data from or writing data to memory in any later clock cycle, if needed, without losing access to the current instruction.

- Registers A and B store the values that were retrieved from the register file's registers (specified by Rs and Rt and provided through Ds and Dt), so they are available in the next clock cycle, to execute the next step of an instruction (e.g., ALU computation).

- The register ALUout stores the output of the ALU, so it is available in the next clock cycle, for further processing (in the next step of the instruction).

- The register MDR stores data output from memory. This register will be used for the `lw` instruction: the data loaded from memory is stored in MDC and therefore available in the next clock cycle (next step of the lw instruction) to be transferred into the register file.

Note that the output of the IR register feeds into several connections, each of which corresponds to a specific subset of the 32 bits of the instruction, stored in IR.

4.4 Detailed Discussion of the Multi-cycle Datapath

As explained before, a multi-cycle implementation breaks every instruction into multiple steps, where every step takes one clock cycle to get executed. The goal is to balance the amount of work that is done in each clock cycle so as to minimize the clock cycle time. Instructions requiring less clock cycles will complete execution faster, while instructions requiring more clock cycles will take more time to complete execution. In this section, we discuss the different steps and how they are implemented using the datapath in Figure 4.17, to execute the instructions we mentioned before: `add`, `sub`, `and`, `or`, `lw`, `sw`, `beq` and `j`.

4.4.1 Instruction fetch

For every instruction, the first step in its execution is the **Instruction Fetch** (IF): feeding the value of PC into the address input, A, of the memory module, to retrieve from memory the instruction stored at the corresponding address. Once retrieved from memory, the instruction is stored in the register IR. Since the ALU would otherwise remain unused during the IF step, the ALU is utilized to simultaneously (i.e., in parallel to retrieving the instruction from memory) compute PC + 4 (this is an example of the resource sharing a multi-cycle implementation enables). That way, the value of PC will be ready to go, to fetch the next instruction, if and when that time comes. So, the IF step can be summarized as follows:

$$\text{IR} = \text{Memory}[\text{PC}];$$
$$\text{PC} = \text{PC} + 4;$$

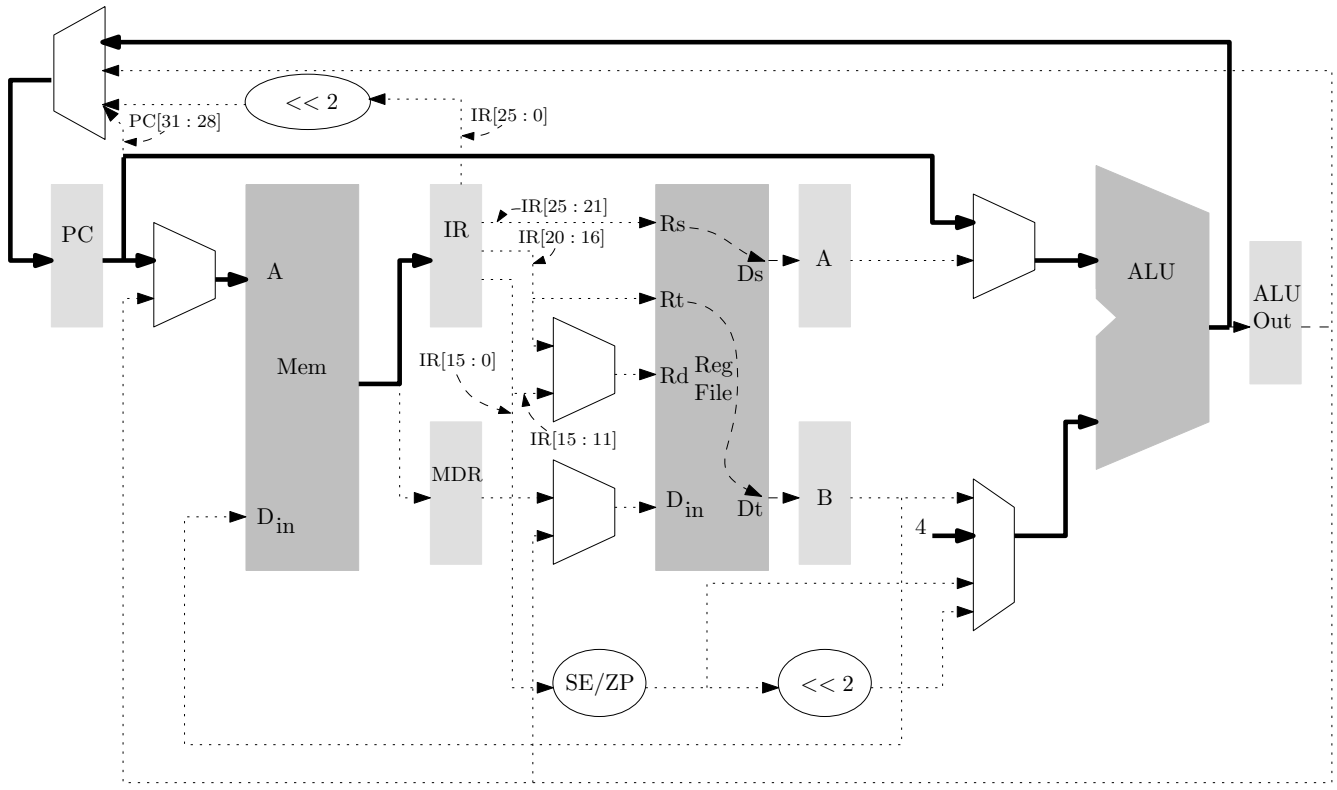

Figure 4.18: Multi-cycle datapath executing the IF step.

Note that the content of any register only gets updated at the next rising clock edge. So, at the next rising clock edge, IR will get updated to contain Memory[PC] and PC will get updated to contain PC + 4. Therefore, computing PC + 4 will not affect the correct instruction properly loading into IR. Moreover, by "freezing" the content of IR (i.e., prevent any undesired, random input to overwrite IR, at any later rising clock edge), we can keep the current instruction in IR as long as needed, without it getting overwritten. In the next section, on CPU control, we will add a "write control" signal to IR, to enable this.

The datapath usage is represented in Figure 4.18. The solid connections are actively used to implement the IF step. The dotted connections carry some random signal, not relevant to the IF step. The multiplexers ensure that the value of PC and the constant 4 are applied as inputs to the ALU. The output of the ALU is directly stored into the register PC.

4.4.2 Instruction decode and register fetch

This step is also common to every instruction. First, by reading the opcode bits IR[31:26] (bits 26 to 31 of the instruction, in the instruction register IR), the CPU control (discussed in the next section) will determine which exact instruction is to be executed (whether it is `add`, `lw`, etc.). Therefore, this is called the **instruction decode** (ID) step.

Simultaneously to decoding the opcode field IR[31:26], the Rs and Rt field of the instruction, i.e., IR[25:21] and IR[20:16] respectively, are applied as inputs Rs and Rt to the

register file, to fetch the contents of the registers Rs and Rt and provide them at Ds and Dt. The fetching of registers Rs and Rt is done regardless of the actual instruction and its format: if Ds and Dt are needed, they can be accessed in the next clock cycle (through A and B), otherwise, they're ignored. More in detail, the Rs field IR[25:21] specifies the first source operand for every instruction (except for a J-type instruction, which doesn't have an Rs field), which is loaded into register A, at the next rising clock edge. The instruction's Rt field IR[20:16] can specify a second source operand or a destination register, depending on the instruction (except for a J-type instruction, which doesn't have an Rt field). However, at this stage of an instruction's execution, it is *harmless* to assume IR[20:16] is specifying a second source register and, therefore, apply IR[20:16] to the Rt input of the register file to store the contents of the corresponding register in register B at the next rising clock edge. The CPU control will determine whether to use B or ignore it, in the next clock cycle.

Third, we note that the ALU is, thus far, unused during the ID step. To put it to good use, we can let it compute the target branch address, as if the current instruction were a `beq` instruction. Even if the current instruction is not a `beq` instruction, it is *harmless* again to use the ALU to calculate PC + (SE(IR[15:0])<<2), where << 2 stands for "shift 2 bits to the left" (note that we write PC and not PC + 4, since PC already contains the address of the next instruction, after the IF step). Moreover, to compute PC + (SE(IR[15:0])<< 2), we don't need any of the registers. The resulting target branch address, computed by the ALU, is stored in the ALUout register and can be used or ignored, during the next clock cycle (used to overwrite and update PC if the current instruction is indeed a `beq` instruction and the branch condition satisfied; ignored otherwise). In summary, the ID step requires:

$$\begin{aligned}&\text{A} = \text{Register[IR[25:21]]};\\ &\text{B} = \text{Register[IR[20:16]]};\\ &\text{ALUout} = \text{PC} + (\text{SE(IR[15} : \text{0])} << 2);\end{aligned}$$

Figure 4.19 illustrates the datapath usage for the ID step. So far, we discussed the IF and ID step, which are identical for every instruction. From the next step on, this will change: the datapath's behavior will depend on the specific instruction.

4.4.3 Execute arithmetic or logic operation, compute memory address or test branch condition

As discussed before, the third step depends on the specific instruction that is being executed.

R-type arithmetic-logic instruction

For an R-type arithmetic or logic instruction, the ALU needs to perform the required operation on the two source operands, available in registers A and B, and store the result in ALUout:

$$\text{ALUout} = \text{A op B};$$

The actual operation "op" can be +/−/AND/OR and is specified by the instruction's opcode and func field. The third step for an R-type arithmetic-logic instruction is shown in Figure 4.20.

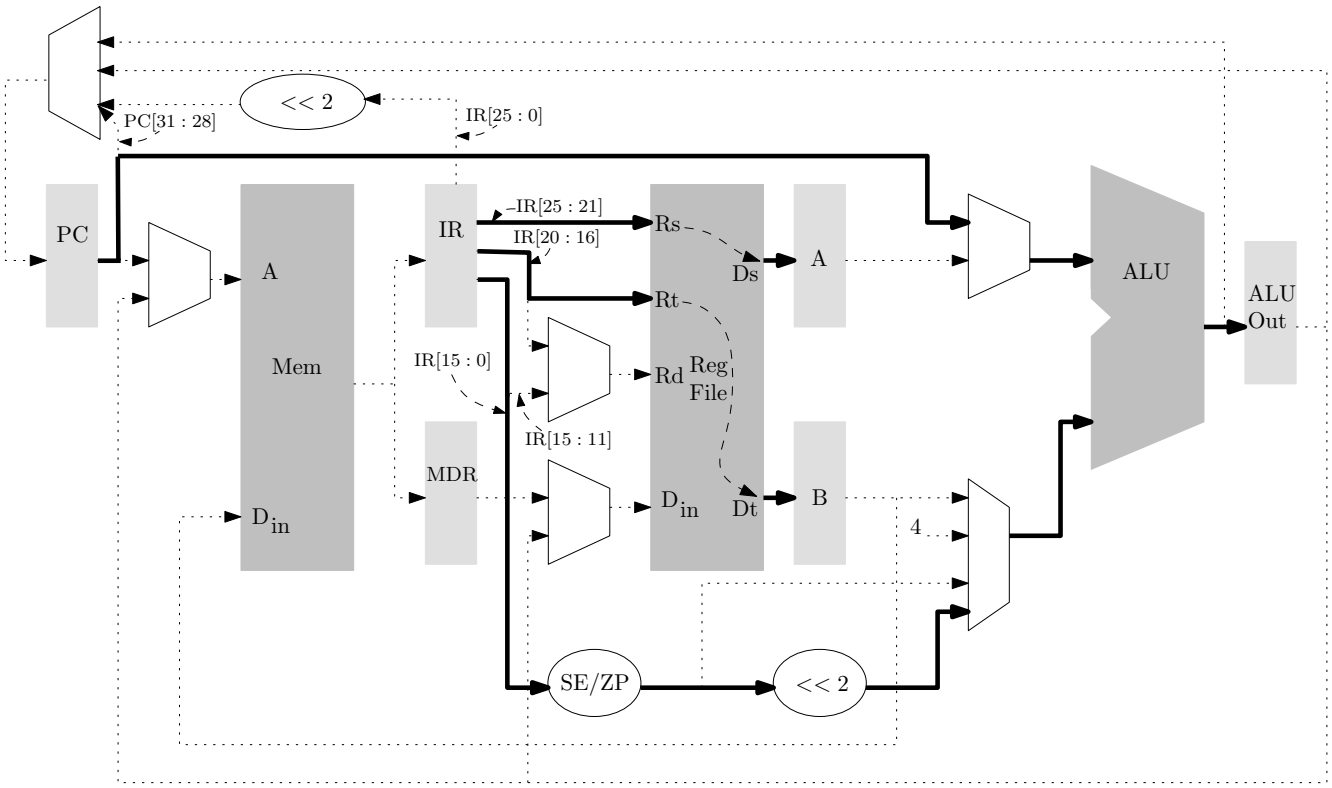

Figure 4.19: Multi-cycle datapath executing the ID step.

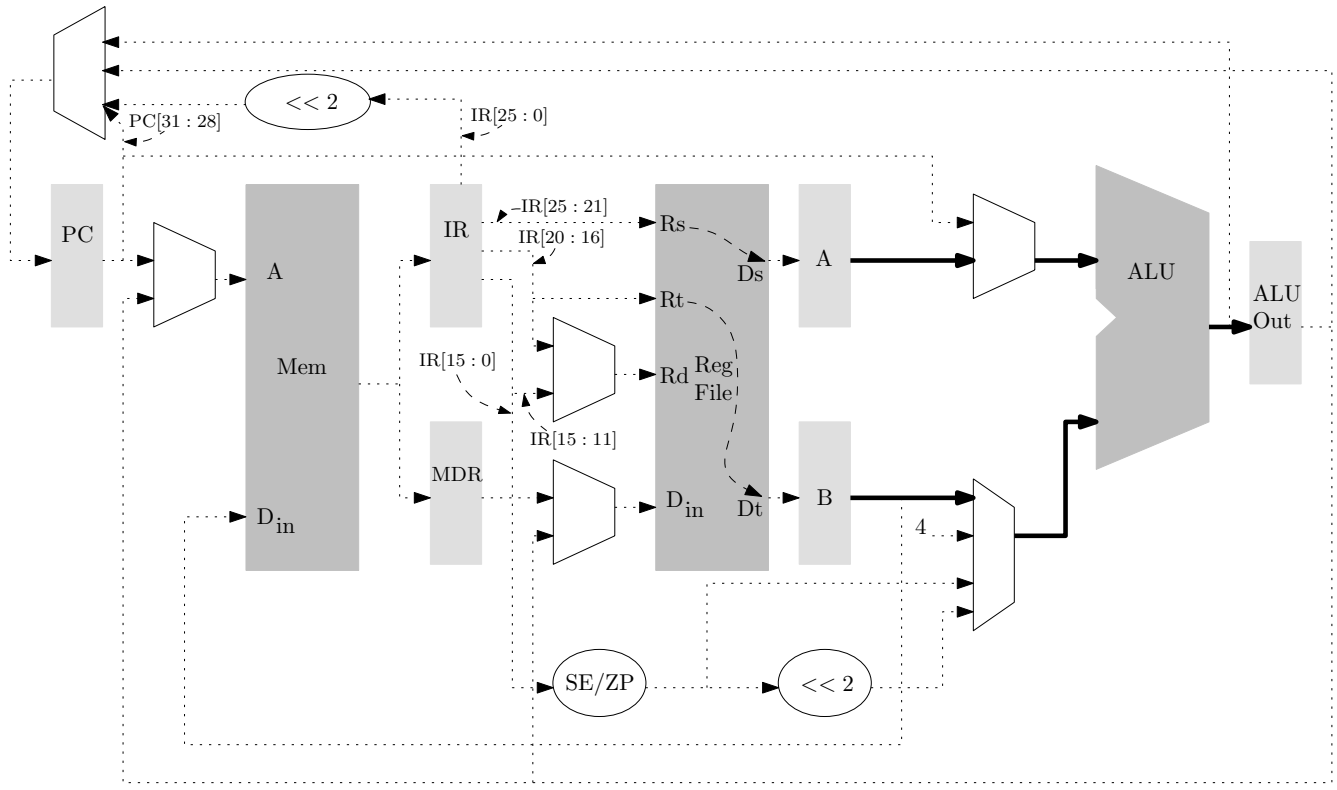

Figure 4.20: Multi-cycle datapath executing step 3 for an R-type arithmetic-logic instruction.

I-type arithmetic-logic instruction

For an I-type arithmetic or logic instruction, the ALU needs to perform the required operation on two operands. The first operand is stored in register A, as for an R-type instruction. The second operand is provided by the instruction's immediate field, i.e., IR[15:0], which needs to be sign-extended or zero-padded (depending on whether it represents an integer, respectively a natural number), to provide a 32-bit input for the ALU. This can be written as:

$$\text{ALUout} = \text{A op (sign-extend / zero-pad (IR[15:0]));}$$

The actual operation "op" can be +/−/AND/OR and is specified by the instruction's opcode and func field. The datapath's usage for the third step of an I-type arithmetic-logic instruction is shown in Figure 4.21.

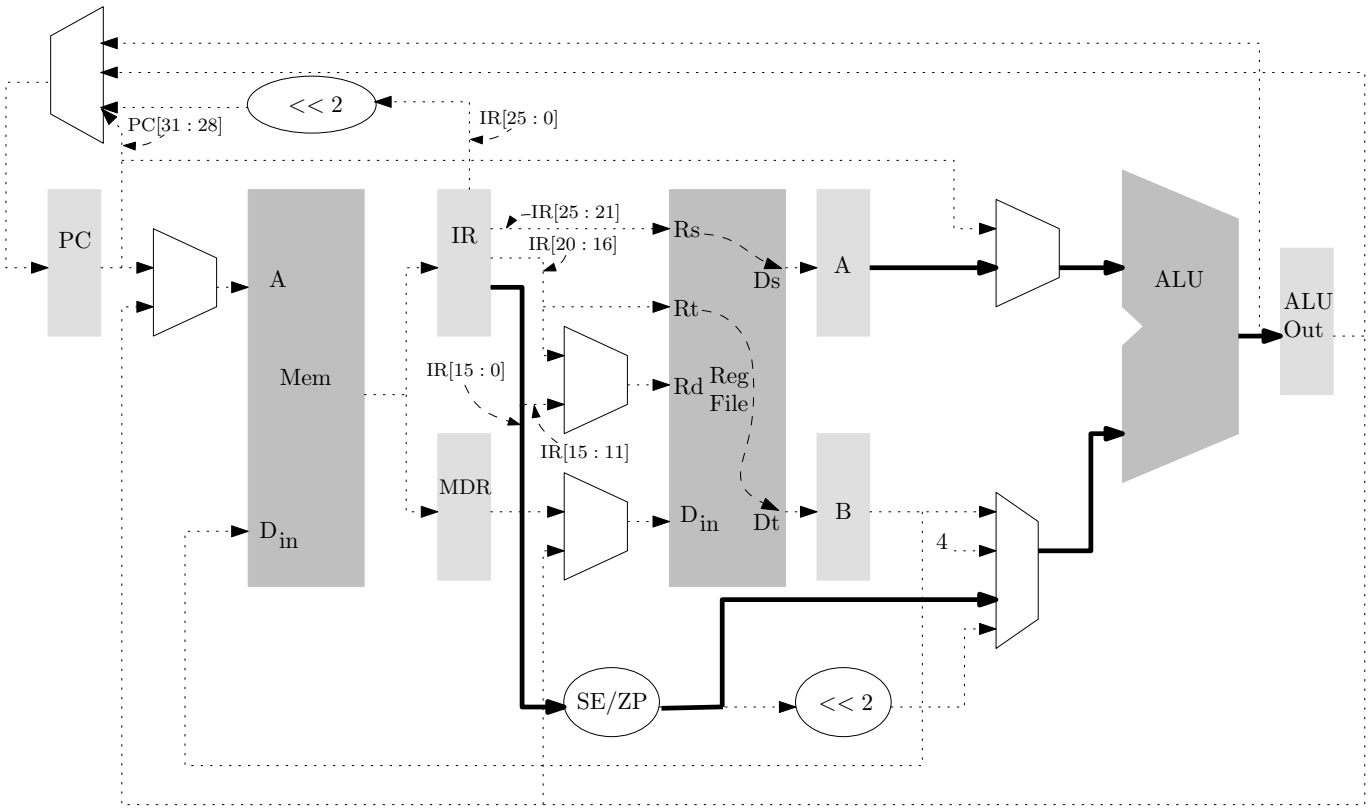

Figure 4.21: Multi-cycle datapath executing step 3 for an I-type arithmetic-logic instruction or computing the memory address for a data transfer instruction.

Data transfer instructions `lw` and `sw`

To transfer data between the memory and the register file, the CPU first needs to obtain the address of the memory location to be accessed. Register A contains the base address, from the register specified by the instruction's Rs field. To add the offset, specified by IR[15 : 0], to the base address, we use the ALU, after sign-extending the offset, to provide the necessary 32-bit input for the ALU:

$$\text{ALUout} = \text{A} + \text{sign-extend(IR[15:0])};$$

The resulting address is stored in the register ALUout. The datapath usage is similar to that for an I-type arithmetic-logic instruction, as depicted in Figure 4.21.

Conditional branching instruction `beq`

At this point, a `beq` instruction has to compare the first source operand, stored in register A, and the second source operand, stored in register B. Therefore, registers A and B are fed into the ALU. By subtracting both operands and checking whether the result is zero, the ALU can perform the comparison. The zero flag output of the ALU is provided to the CPU control logic, which decides whether to branch or not. Note that the target branch address

is available in the ALUout register, from the ID step in the previous clock cycle:

if (A==B) PC = ALUout;
else do not update PC;

If the zero flag is set, the previously calculated target branch address, in ALUout, will be loaded into the register PC. If the zero flag is not set, the PC register will remain unchanged. This concludes the `beq` instruction. Next, the CPU will continue with an IF step, for the next instruction. Figure 4.22 shows the datapath usage for the third and last step of a `beq` instruction's execution, for the case where the branch condition is satisfied.

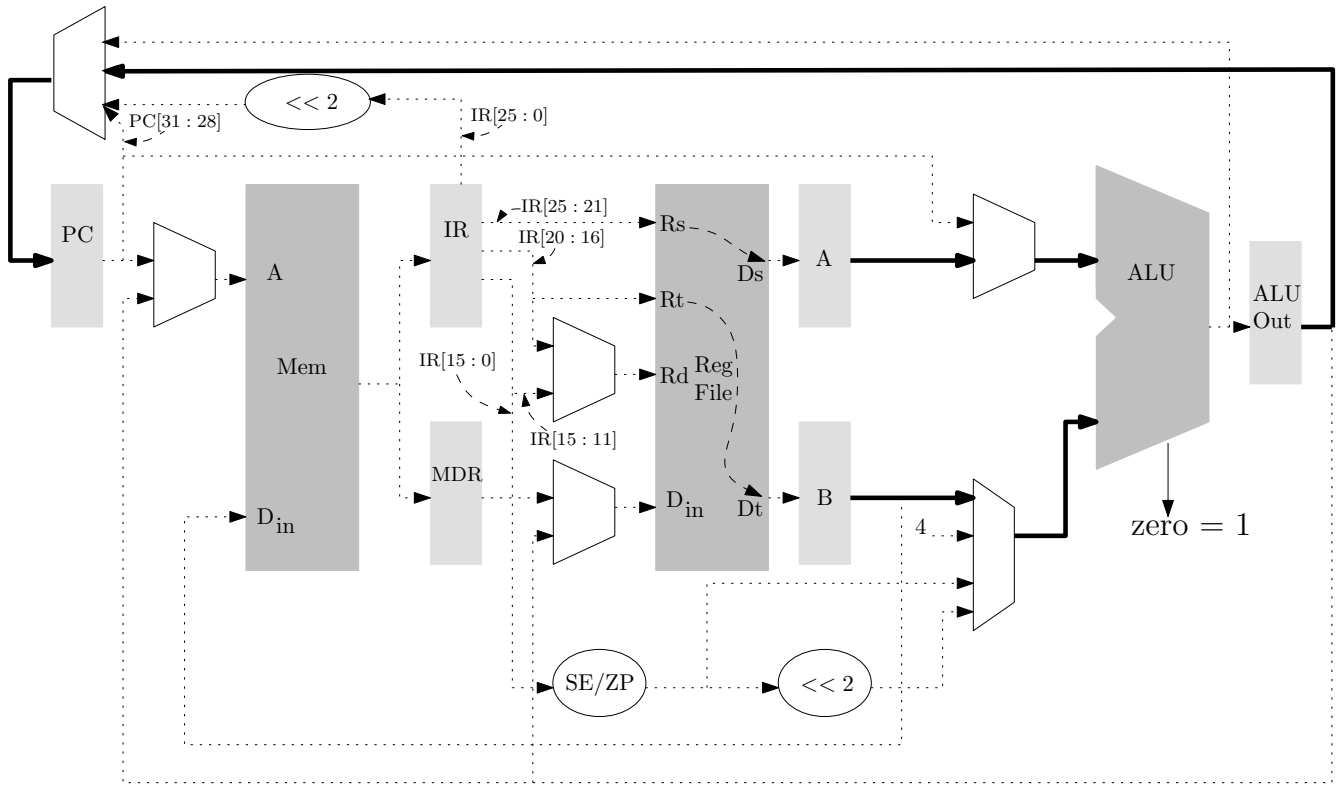

Figure 4.22: Multi-cycle datapath testing the branch condition for a `beq` instruction.

Note that, at the rising edge of the next clock cycle, the ALUout register is overwritten with the result of the ALU's subtraction, which overwrites the target branch address in ALUout. However, at that same rising clock edge, the target branch address is latched into the PC register, as depicted in Figure 4.22. So, PC will contain the correct address of the next instruction, which is all that is needed for the upcoming IF step. ALUout can be ignored at that point.

Jump instruction

For a jump instruction, IR[25:0] specifies the absolute word address of the next instruction. Shifting IR[25:0] by 2 bits to the left provides the absolute byte address. Since instructions

are stored in the text segment in memory, PC[31:28] doesn't change and can be concatenated (||) with IR[25 : 0] << 2 to provide the full 32-bit memory address of the next instruction:

$$\text{PC} = \text{PC}[31 : 28] \;||\; (\text{IR}[25 : 0] << 2);$$

The resulting datapath usage is given in Figure 4.23. As for the `beq` instruction, this is the third and final step of a `j` instruction's execution. Notice that, although the first table in Subsection 4.3.1 could suggest that `j` requires just one clock cycle, it actually requires 3 cycles: that is because a) the first 2 cycles are shared amongst all instructions and b) we need an extra cycle to update PC, which is not mentioned in the table (as it does not require R Access, M Access or ALU).

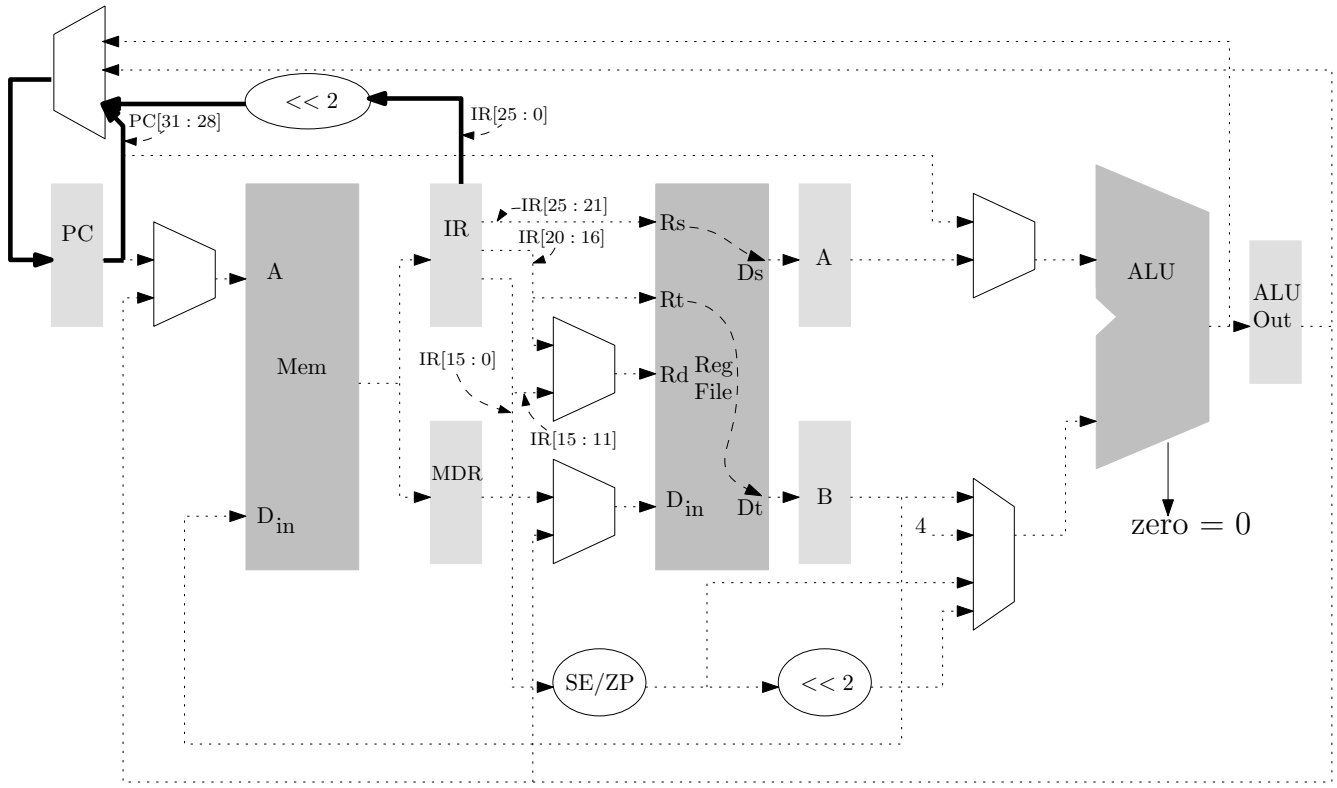

Figure 4.23: Multi-cycle datapath executing the third and final step of a `j` instruction.

4.4.4 Complete arithmetic-logic instruction or access memory

While `beq` and `j` only require 3 steps, the following instructions require at least one additional step.

R-type arithmetic-logic instruction

To complete an R-type arithmetic-logic instruction, the result of the operation, stored in ALUout, needs to be transferred to the correct destination register in the register file. Since

the Rd field, i.e., IR[15:11], of an R-type instruction specifies the destination register, we feed these 5 bits into the Rd input of the register file. To store the contents of ALUout in the register specified by Rd, the content of ALUout is applied to the Din input of the register file:

$$\text{Register[IR[15:11]]} = \text{ALUout;}$$

This is shown in Figure 4.24.

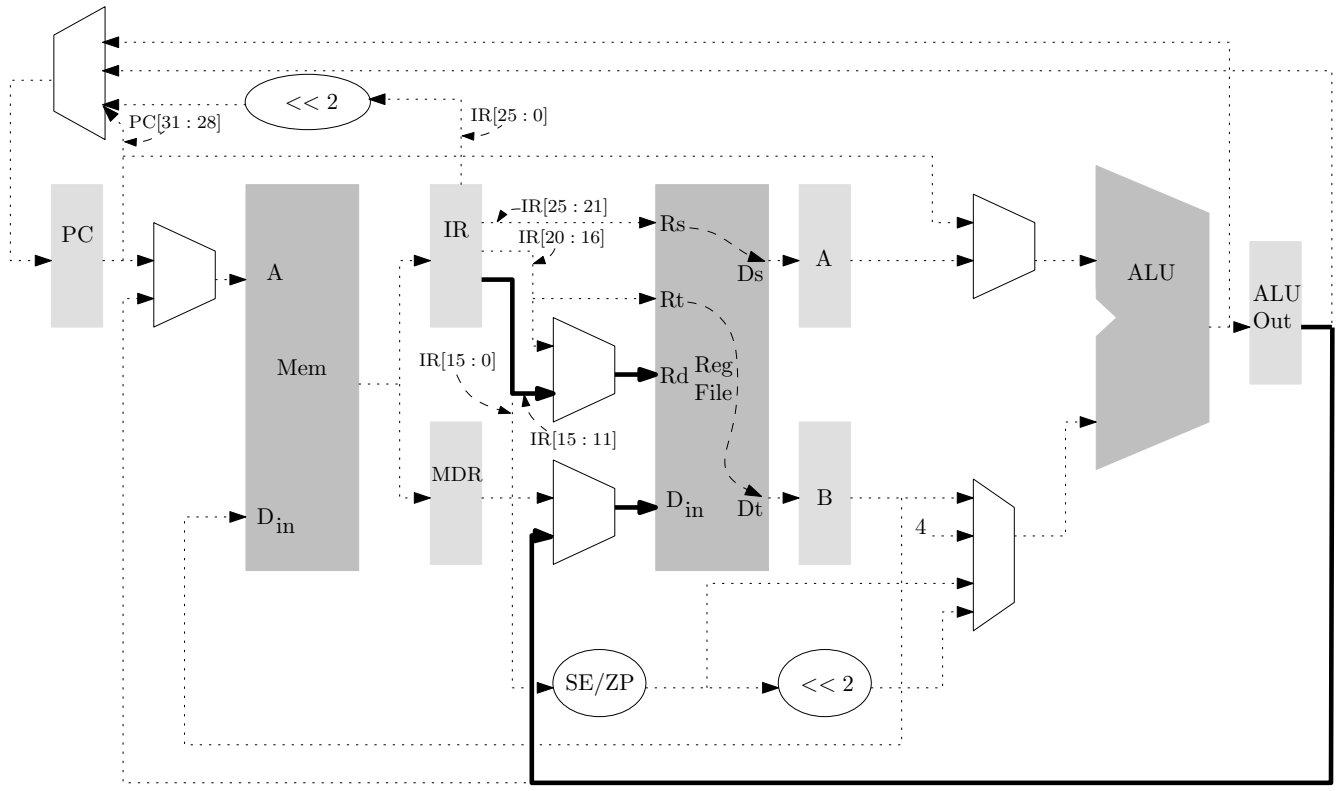

Figure 4.24: Multi-cycle datapath executing the 4th and final step for an R-type arithmetic-logic instruction.

I-type arithmetic-logic instruction

To complete an I-type arithmetic-logic instruction, the result of the operation, stored in ALUout, needs to be transferred to the correct destination register in the register file. For an I-type instruction, the instruction's Rt field, i.e., IR[20:16], specifies the destination register (and not IR[15:11], as for an R-type instruction):

$$\text{Register[IR[20:16]]} = \text{ALUout;}$$

To implement this, the following connections are made: apply the content of ALUout to the Din input of the register file and feed IR[20:16] into the Rd input of the register file. So, in this particular case, **the Rt field of the instruction, IR [20:16], is applied as the**

Rd input of the register file. Although this might seem confusing, keep in mind that the register file is a hardware component with fixed functionality: its output Ds is determined by input Rs, its output Dt by Rt, and Din is stored in the register determined by input Rd. For some instructions, e.g., an I-type arithmetic-logic instruction, the destination register is specified by the instruction's Rt field, IR[20:16] (which then needs to be applied as the Rd input of the register file); for other instructions, e.g., an R-type arithmetic-logic instruction, the destination register is specified by the instruction's Rd field, IR[15:11] (which then needs to be applied as input Rd of the register file). The resulting datapath usage is shown in Figure 4.25.

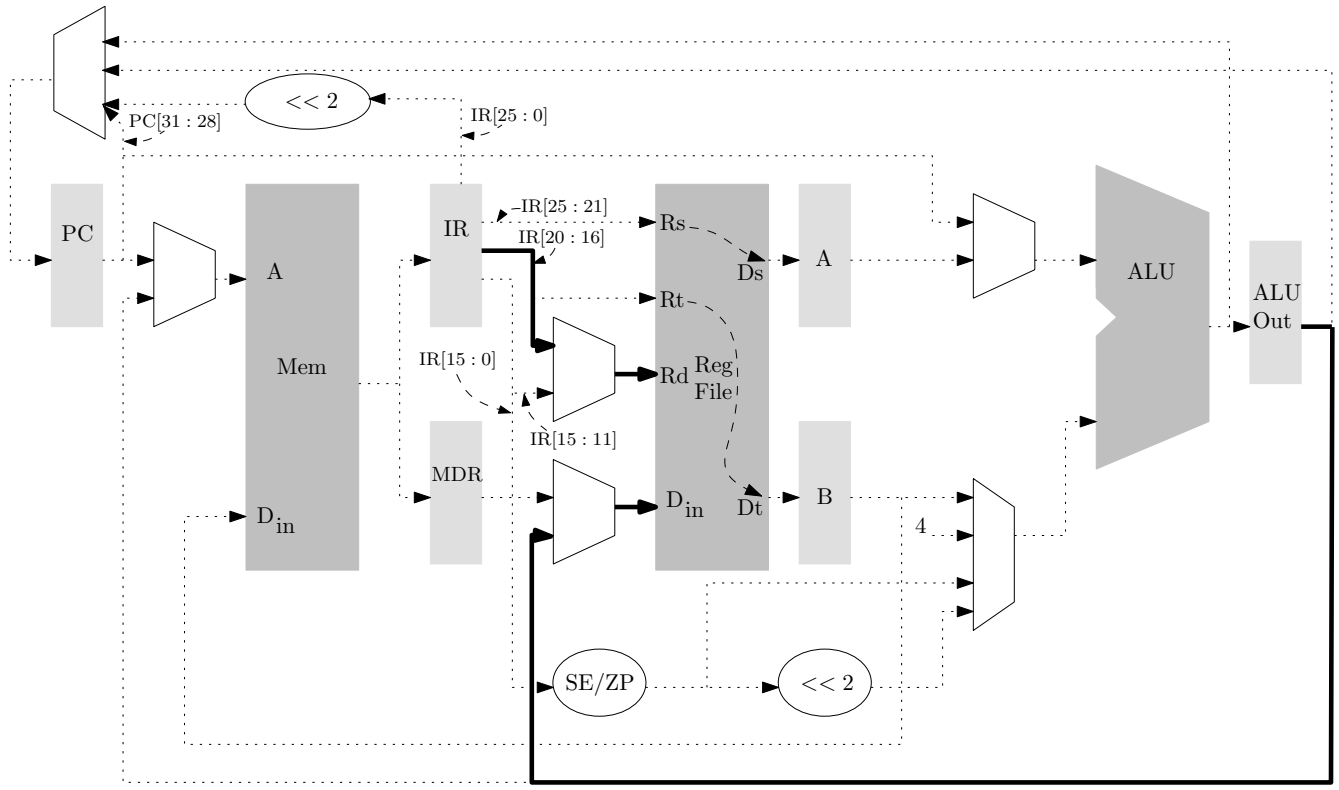

Figure 4.25: Multi-cycle datapath executing the 4th and final step for an I-type arithmetic-logic instruction.

Store word

The previous step computed the memory address that needs to be accessed and stored it in the ALUout register. The data that needs to be stored in memory is specified by the Rt field, IR[20:16], of the `sw` instruction and is available in register B. Therefore, to complete the `sw` instruction, the datapath needs to provide the memory with the address from register ALUout and the data from register B:

$$\text{Memory[ALUout]} = \text{B};$$

The datapath usage that completes the `sw` instruction is shown in Figure 4.26.

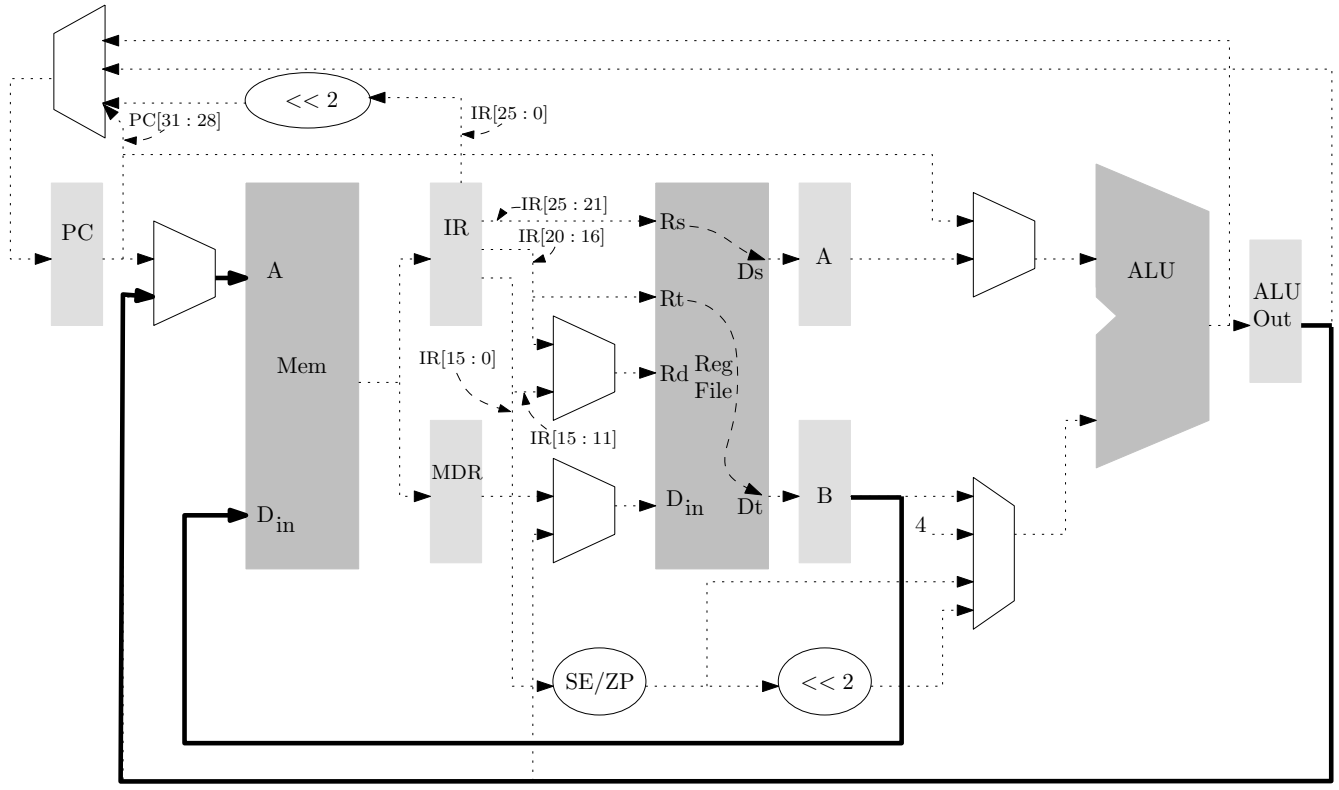

Figure 4.26: Multi-cycle datapath executing the memory access step for an `sw` instruction.

Load word

The previous step computed the memory address that needs to be accessed and stored it in the ALUout register. Next, the datapath needs to access the memory at that address and store the contents of that memory location in the register MDR:

$$\text{MDR} = \text{Memory[ALUout]};$$

The datapath usage for this step is shown in Figure 4.27.

4.4.5 Complete the `lw` instruction

Only the `lw` instruction requires a fifth step, to complete its execution. The data, stored in the MDR register, needs to be transferred into the appropriate register in the register file. The latter is specified by the Rt field, IR[20:16] of the `lw` instruction (which is an I-type instruction):

$$\text{Register[IR[20:16]]} = \text{MDR};$$

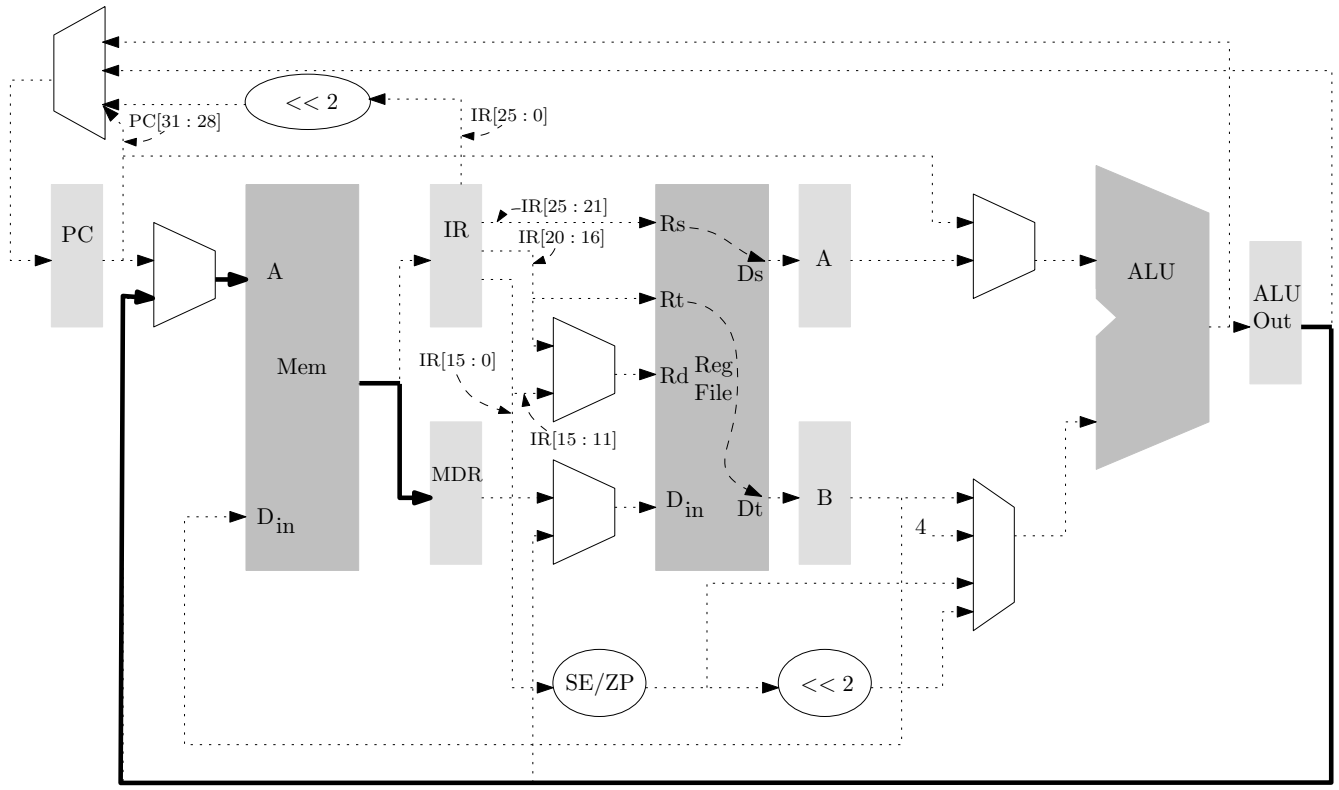

Figure 4.27: Multi-cycle datapath executing the memory access step for an `lw` instruction.

Applying MDR as the Din input and IR[20:16] as the Rd input of the register file achieves the correct outcome. The resulting datapath usage is given in Figure 4.28.

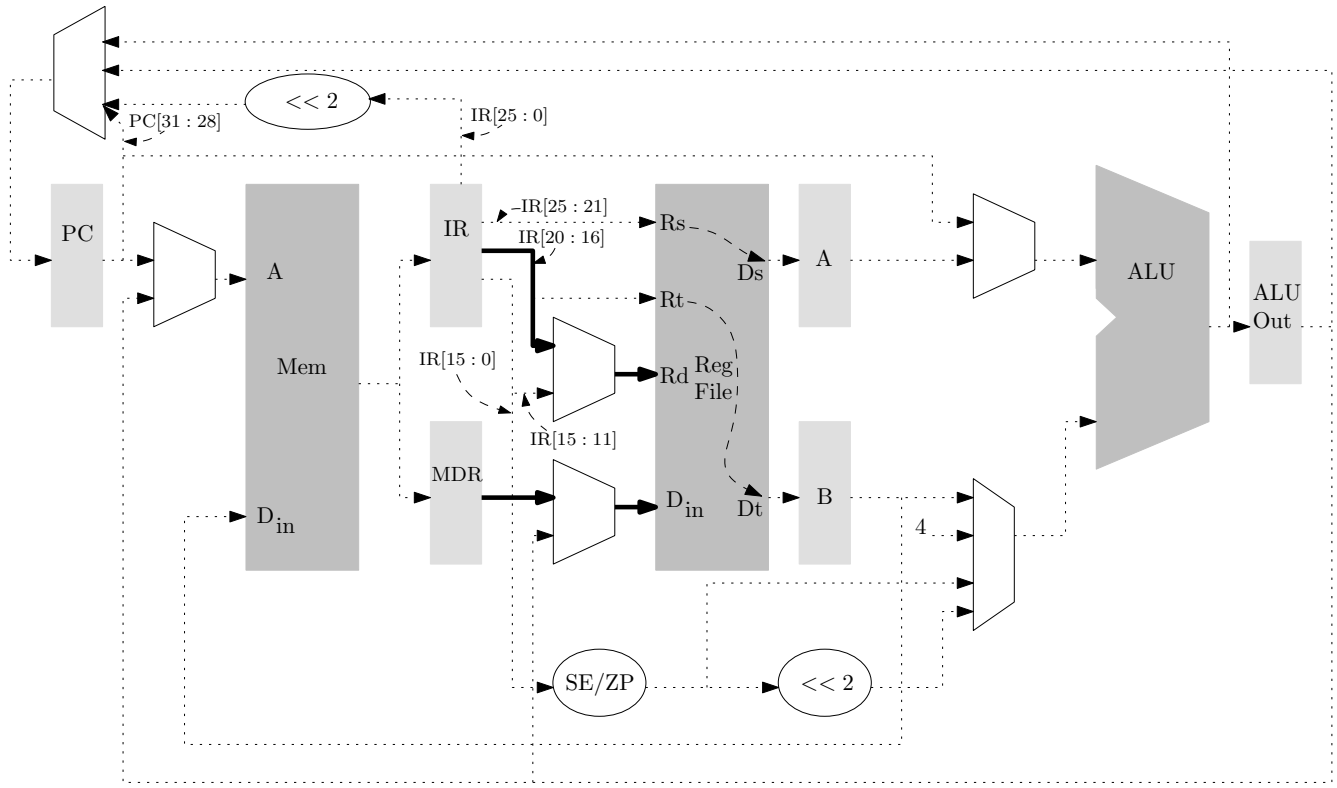

Figure 4.28: Multi-cycle datapath executing the 5th and final step for an `lw` instruction.

4.5 Control Logic

In this section we will analyze the CPU's control logic, i.e., the *brains of the computer*. The previous section showed that specific connections (i.e., multiplexer settings) and register updates in the datapath of the CPU are required to guarantee the correct execution of each step of an instruction. In this section, we will provide the datapath with several control signals and describe the control logic steering these signals, to ensure that each step is executed correctly at any clock cycle and that the steps required to execute an instruction are performed in the right order.

4.5.1 The ALU control unit

The **ALU control** unit is a subset of the control logic, implemented as a separate unit. The different purposes the ALU could be used for can be summarized as follows:

- Compute PC + 4
- Compute PC + 4 + offset ×4

- Evaluate the branch condition (and compute the zero flag)
- Compute the memory address for an `lw` or `sw` instruction
- Perform normal arithmetic or logic operations

The specific operation executed by an ALU depends on the 3-bit ALU control signal: 2 bits determine whether the ALU's internal AND gate (00), the ALU's internal OR gate (01) or the ALU's internal adder (10) will be used; the third bit is only relevant if the previous 2 bits are 10, to determine whether to perform addition (0) or subtraction (1).

The 3-bit ALU control signal is generated by the ALU control unit, which has two inputs: the FUNC field bits from the instruction register, IR[5:0], and a 2-bit **ALUop** control signal provided by the CPU's main control unit. If ALUop is 00, the ALU will perform an addition, no matter what the FUNC field specifies (e.g., for `lw` and `sw`). For ALUop equal to 01, the ALU will perform a subtraction, no matter what the FUNC field specifies (e.g., for `beq`), while for ALUop equal to 10, the operation specified by the FUNC field will be followed (e.g., for an R-type instruction). The following truth table shows the inputs to the ALU control unit and the corresponding output and can be implemented using simple combinational logic.

ALUop	Function Code	Action	ALU ctrl
00	–	add	010
01	–	sub	110
10	100000	add	010
	100010	sub	110
	100100	and	000
	100101	or	001

4.5.2 The control signals

Figure 4.29 shows the multi-cycle datapath, provided with control signals (driven by the control logic explained in the next section), indicated as gray-colored signals. We now describe them one by one.

1. **PCSrc** : PCSrc determines what input is applied to the PC register. The value of PC can be updated to PC+4, the target branch address or the jump address.

2. **IorD** : Since instructions and data are stored in the same memory, the CPU will address the memory with an instruction address (from PC) or a data address (from ALUout), depending on what step of what instruction is being executed. The IorD signal will decide whether to apply the value of PC or the value of ALUout as memory address.

3. **IRWr** : This is the write enable signal for the IR register. If IRWr = 1, the register IR will be updated with the value applied at its input, at the next rising clock edge. If IRWr = 0, IR will not be updated, even if its input has changed. This allows to save the current instruction in the IR register, throughout several clock cycles (from instruction fetch to completion of the instruction), even if the input of IR changes (e.g.,

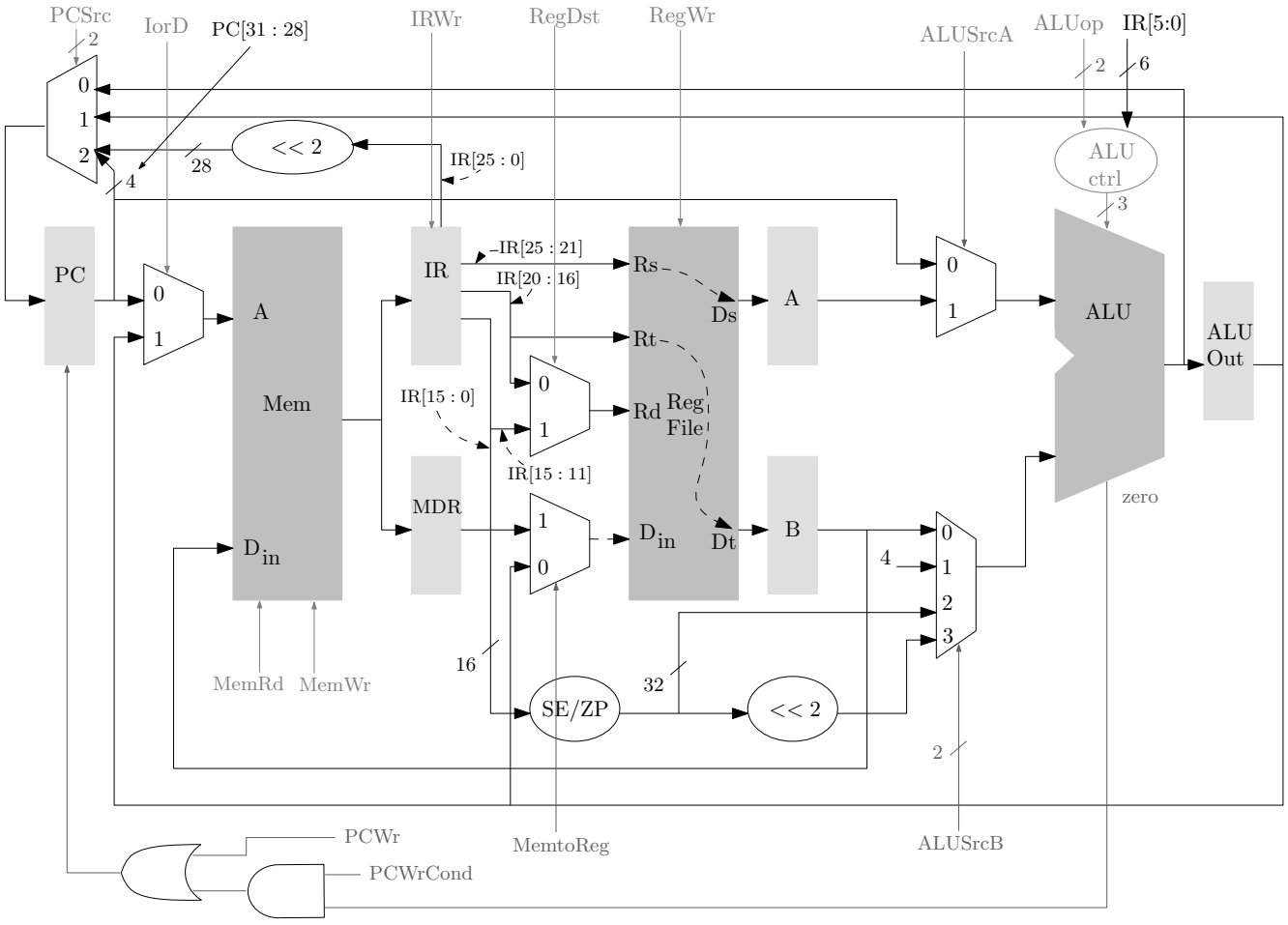

Figure 4.29: Multi-cycle datapath.

by reading data from memory, etc.). Amongst all special registers in the datapath (PC, IR, MDR, A, B, and ALUout), only PC and IR have a write enable signal. So, the values of MDR, A, B, and ALUout will be updated automatically, at every rising clock edge, to whatever their input value is (if their inputs have changed, their values get overwritten). For IR and PC, the write enable control signals prevent them from being accidentally overwritten.

4. **RegDst** : As discussed in the previous section, the Rd input of the register file specifies the register that is written. According to the type of instruction, the destination register is specified by IR[20 : 16] or by IR[15 : 11]. The RegDst control signal allows the control unit to route the right value to the input Rd of the register file.

5. **ALUSrcA** : Determines whether register A or PC provides the first input of the ALU.

6. **ALUop & IR**[5 : 0] : ALUop and IR[5 : 0] are the inputs of the ALU control unit, as discussed earlier.

7. **ALUSrcB** : Similar to ALUSrcA, this control signal determines which is the second input of the ALU. Because there are four possible inputs, this signal consists of 2 bits.

8. **MemtoReg** : The Din input of the register file takes data from either the ALUout or the MDR register. MemtoReg determines which input is applied.

9. **MemRd** : Needs to be enabled to read data from the memory.

10. **MemWr** : Similar to other write enable signals, MemWr = 1 allows the data, applied at the Din input of the memory, to be written into the memory location with address A. If MemWr is 0, no writing occurs (to prevent accidental overwriting).

11. **RegWr** : This control signal being 1 allows the data, applied at the Din input of the register file, to be written into the register specified by Rd. If RegWr is 0, no writing occurs (to prevent accidental overwriting).

12. **PCWr** : The write enable signal for the PC register. Similar to IRWr. PC gets updated if PCWr = 1 and remains unchanged if PCWr = 0 (notice how the OR-gate ensures that PCWr = 1 gets passed on to the PC register, to enable writing).

13. **PCWrCond** : The conditional write enable signal for the PC register. This control signal being 1 allows the zero output of the ALU to determine whether the register PC gets updated with the target branch address, or not, for a `beq` instruction. If PCWRCond is 0, the zero output of the ALU has no influence on writing the PC register (because of the AND gate).

4.5.3 FSM describing the CPU control unit

The CPU control unit determines all the control signals introduced in the previous subsection, to ensure correct execution of every step of an instruction. It also makes sure the different steps, to execute an instruction, are executed in the right order. Therefore, the

control unit requires sequential logic and is described using an FSM (Finite State Machine). The table below gives an overview of the steps required for each instruction.

`j`	IF⇒ID⇒Complete Jump
`beq`	IF⇒ID⇒Complete Branch
`add`, `sub`, `and`, `or`	IF⇒ID⇒Execute⇒Write Back
`lw`	IF⇒ID⇒Mem Address Calculation⇒Mem Access⇒Write Back
`sw`	IF⇒ID⇒Mem Address Calculation⇒Mem Access

Based on the previous table, we can lay out the basic structure of the state diagram of the FSM describing the CPU's control unit. Such diagram describes the different states, how to transition between them and the specific value of the outputs (the control signals) for each state. This is depicted in Figure 4.30. Next, we will complete this diagram. Please note that write enable signals will only be mentioned when set. If they are not mentioned, they are assumed to be zero.

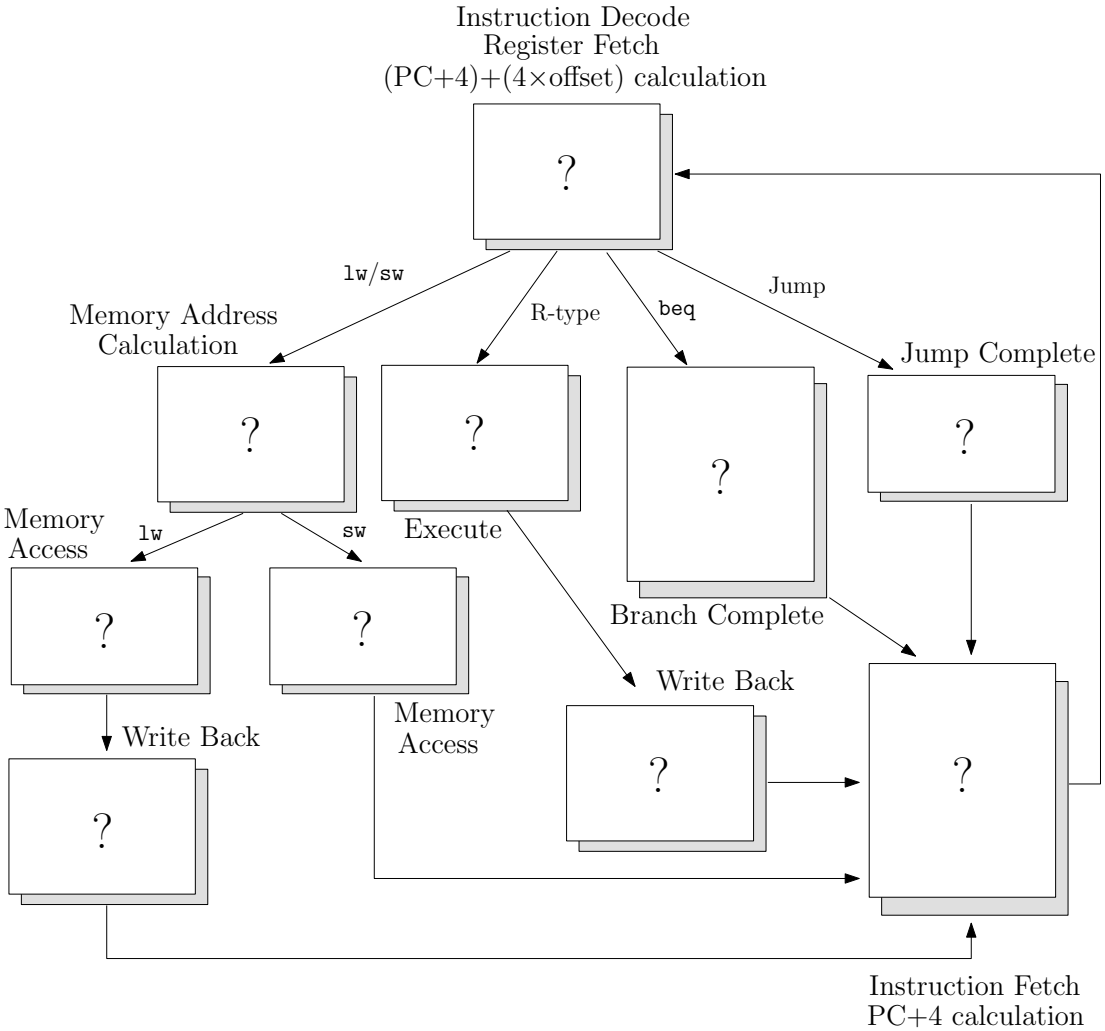

Figure 4.30: Basic layout of the state diagram describing the control FSM.

Instruction fetch, PC+4 calculation

Figure 4.31 shows the datapath and the relevant control signals (emphasized) for the IF step, including the PC + 4 calculation. The required control signals are:

1. IorD = 0 : feed the value of PC into input A of the memory;
2. MemRd = 1 : enable reading the next instruction from memory;
3. IRWr = 1 : store the instruction, retrieved from memory, into the IR register;
4. ALUSrcA = 0 : feed PC into the ALU;
5. ALUSrcB = 01 : feed constant 4 into the ALU;
6. ALUop = 00 : add the two inputs of the ALU, to compute PC + 4;
7. PCSrc = 00 : apply PC + 4 at the input of PC;
8. PCWr = 1 : enable updating PC.

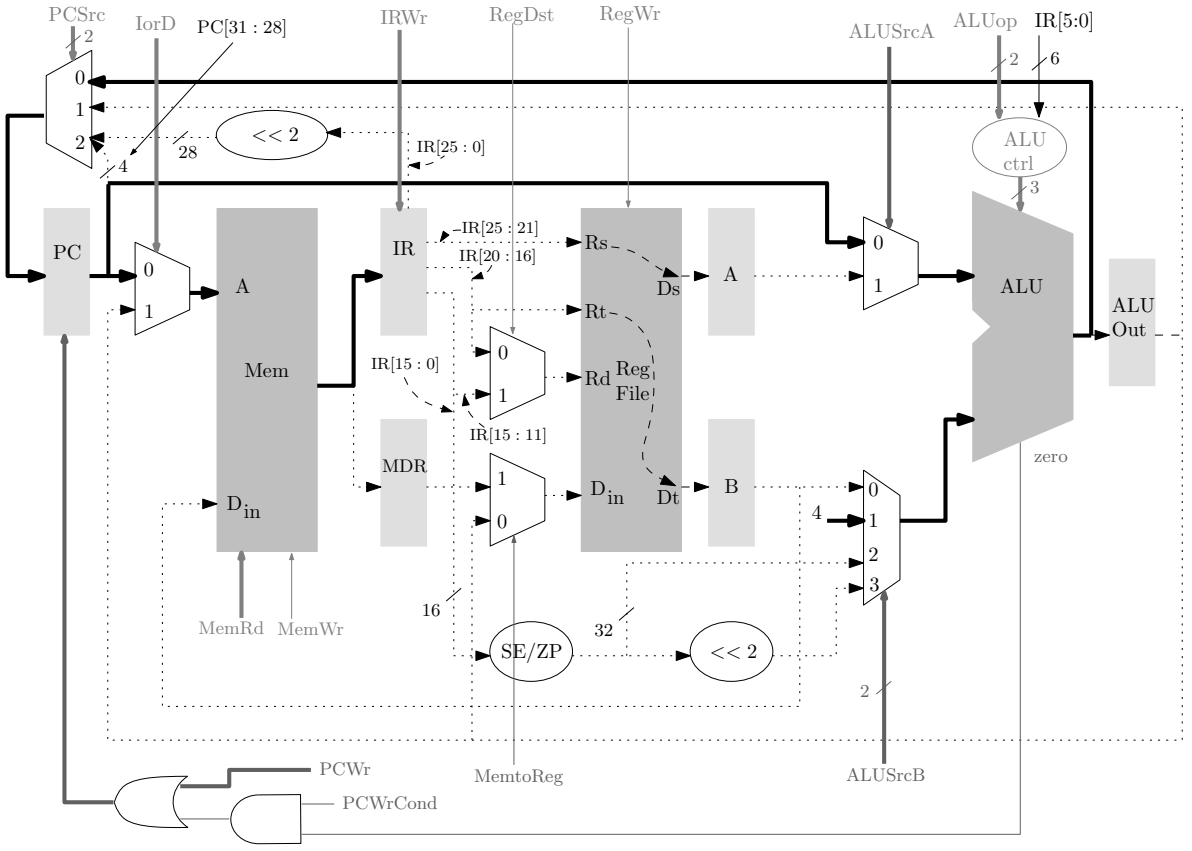

Figure 4.31: Datapath and relevant control signals (emphasized) for instruction fetch, PC+4 calculation.

The corresponding control FSM is updated in Figure 4.32.

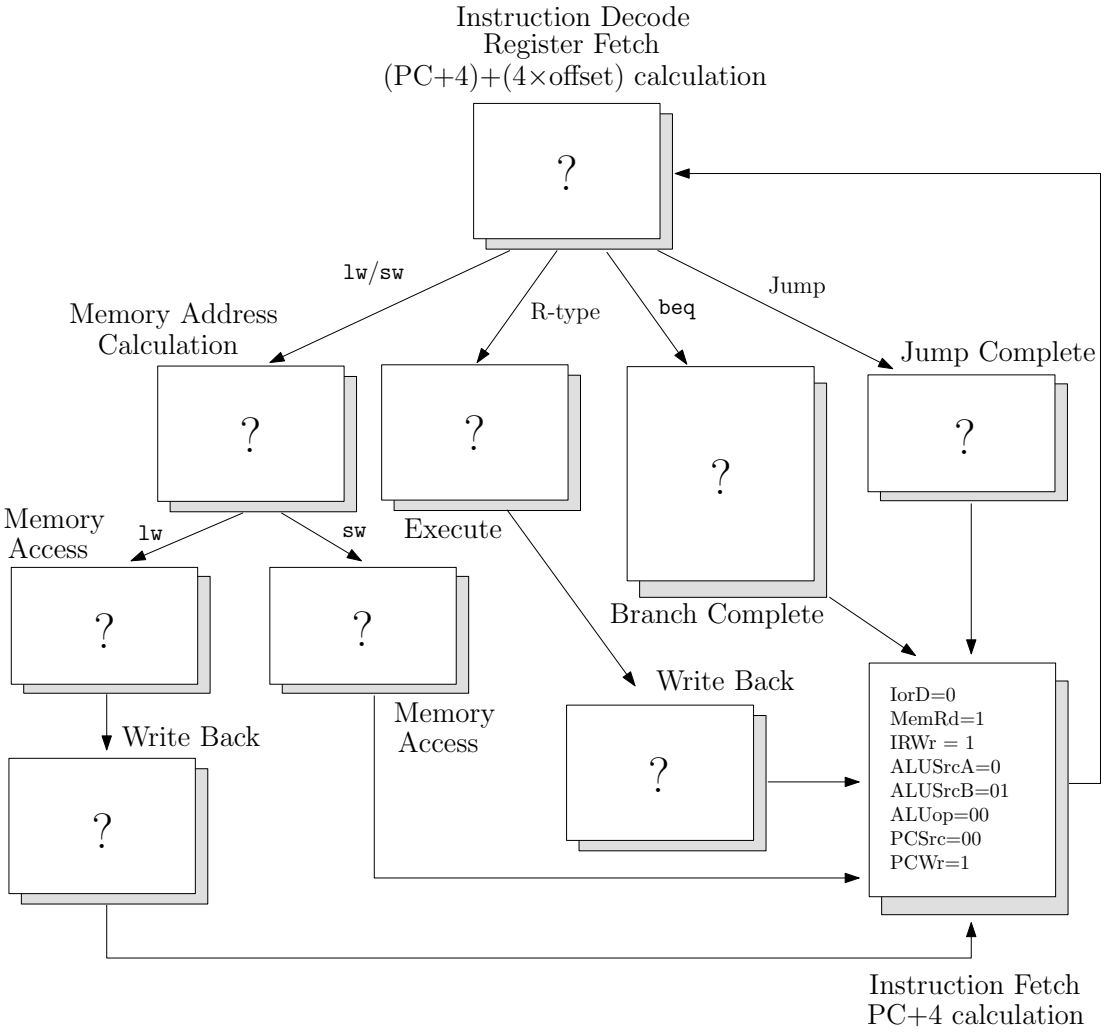

Figure 4.32: Control FSM, updated for instruction fetch, PC+4 calculation.

Instruction decode, register fetch and branch target address calculation

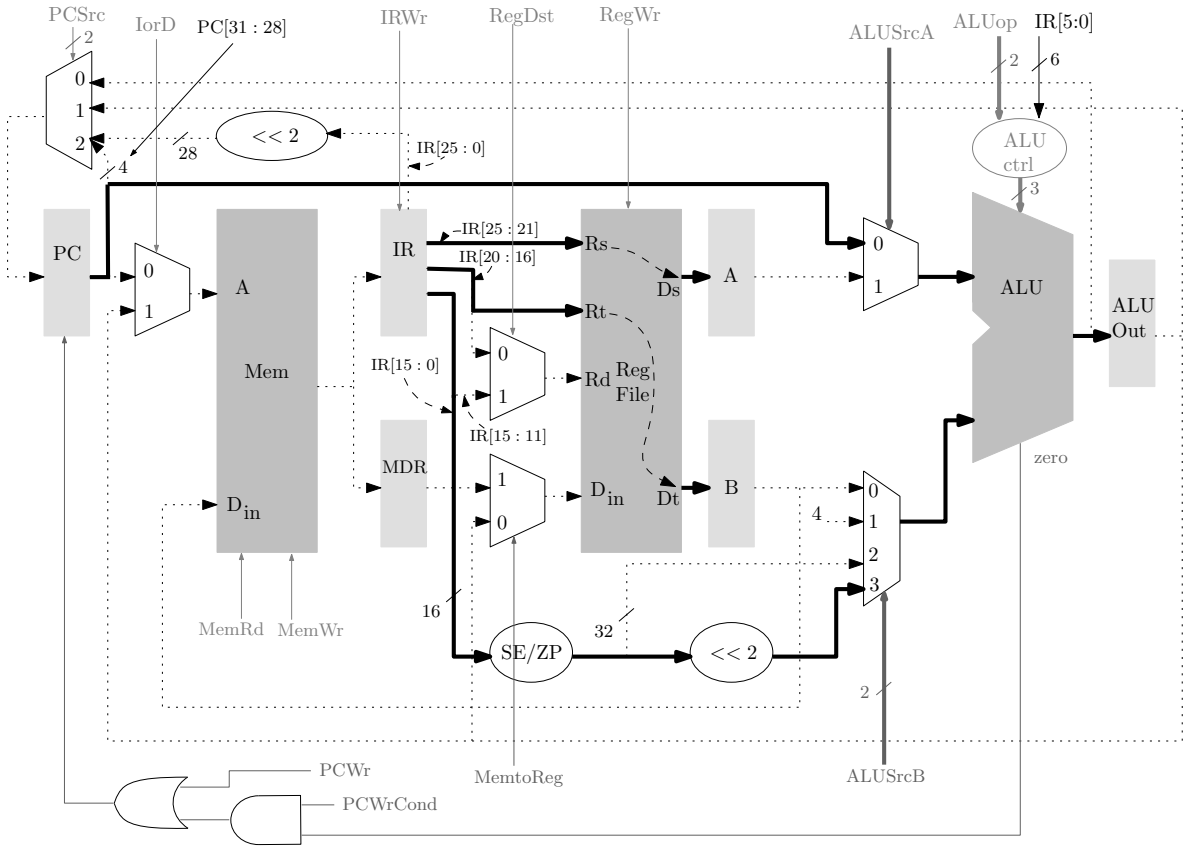

Figure 4.33: Datapath and relevant control signals for instruction decode, register fetch and branch target address computation.

Figure 4.33 represents the datapath for the ID step, common to all instructions. The contents of the registers specified by IR[25 : 21] and IR[20 : 16] are stored in A and B, respectively, and the branch target address is computed and stored in the ALUout register.

1. ALUSrcA = 0 : feed PC (increased with 4 in the previous step) into the ALU;

2. ALUSrcB = 11 : feed the sign-extended and shifted offset value specified in IR[15 : 0] into the ALU;

3. ALUop = 00 : add the two ALU inputs.

The FSM state diagram is updated in Figure 4.34.

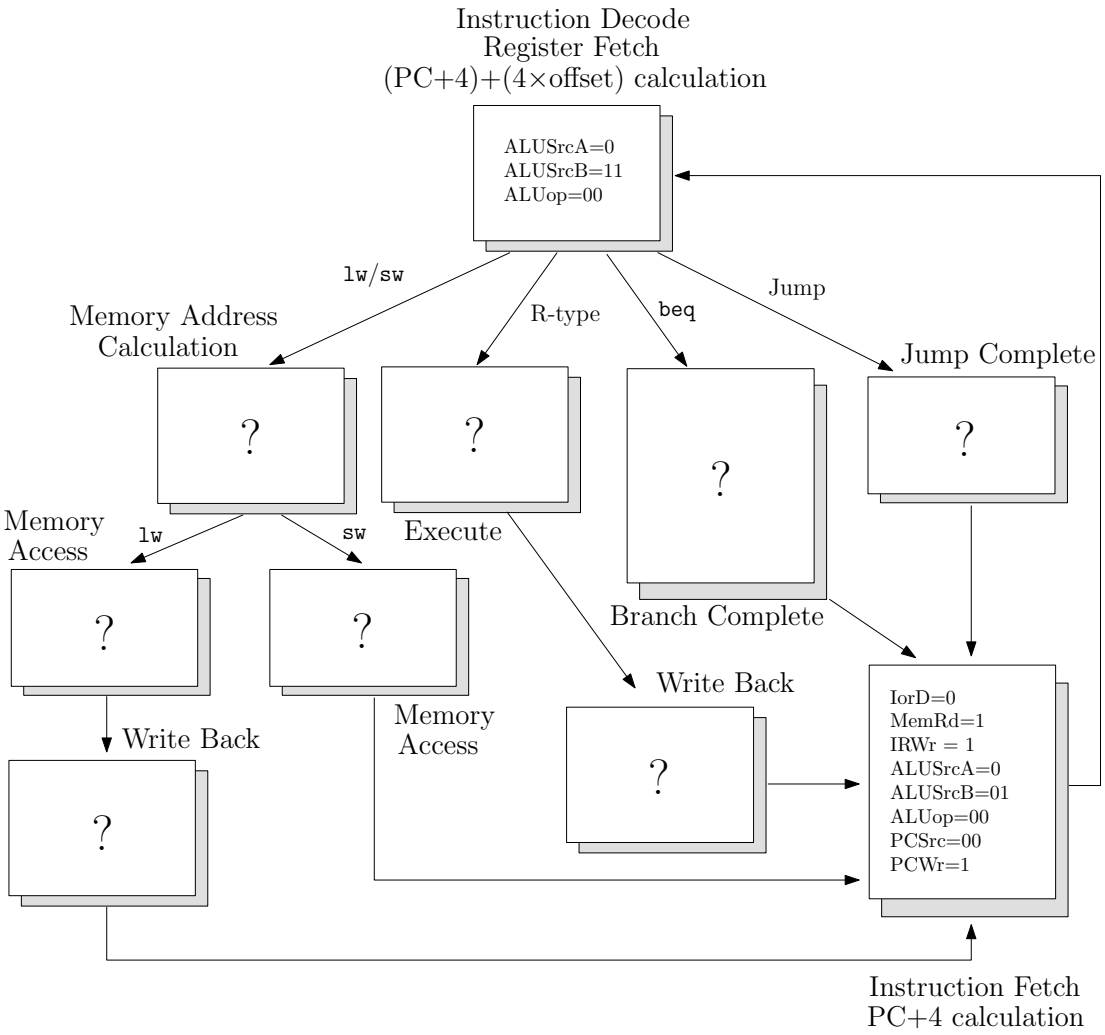

Figure 4.34: Control FSM, updated for instruction decode, register fetch and branch target address calculation.

Jump instruction complete

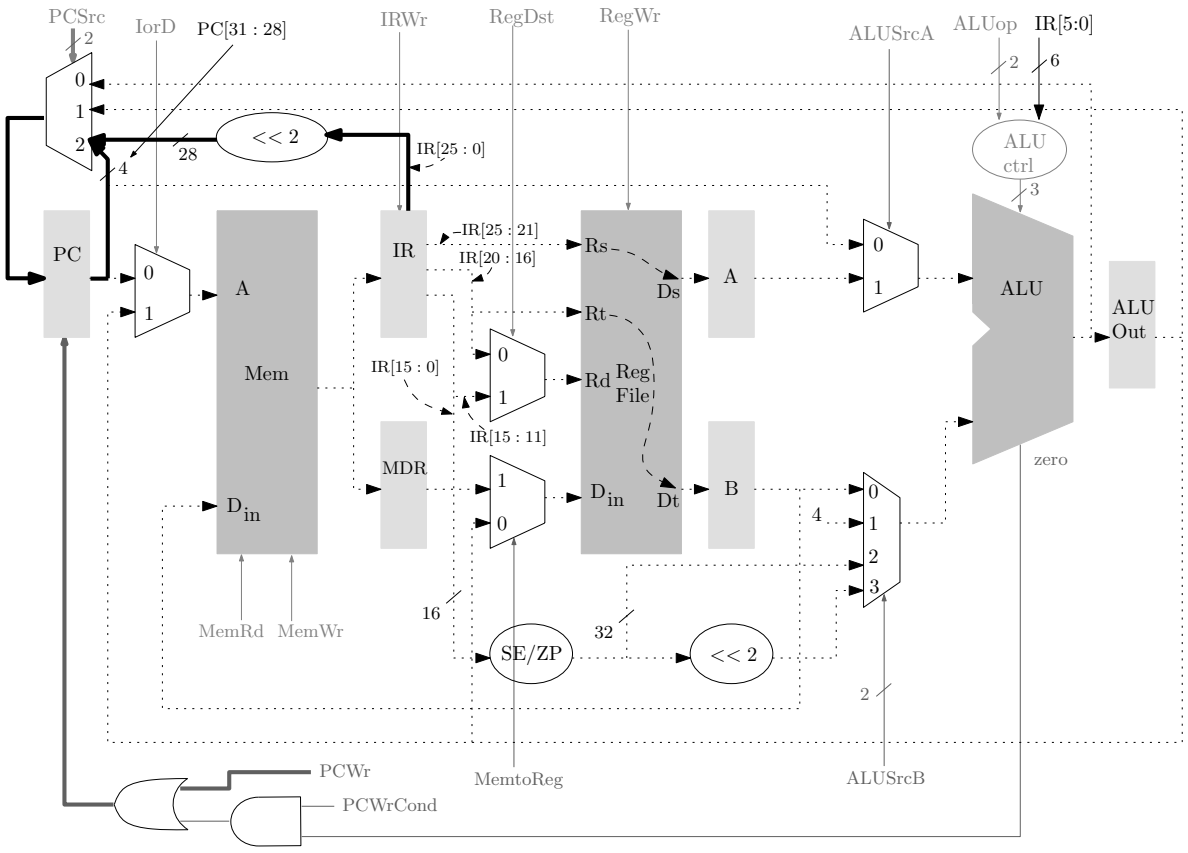

Figure 4.35: Datapath and relevant control signals for jump complete.

To complete the jump instruction, PC needs to be updated using IR[25 : 0]<<2 and PC[31:28], depicted in Figure 4.35, which requires the following control signals:

1. PCSrc = 10 : apply PC[31:28] || IR[25 : 0]<<2 to PC;

2. PCWr = 1 : update PC.

The updated FSM is given in Figure 4.36

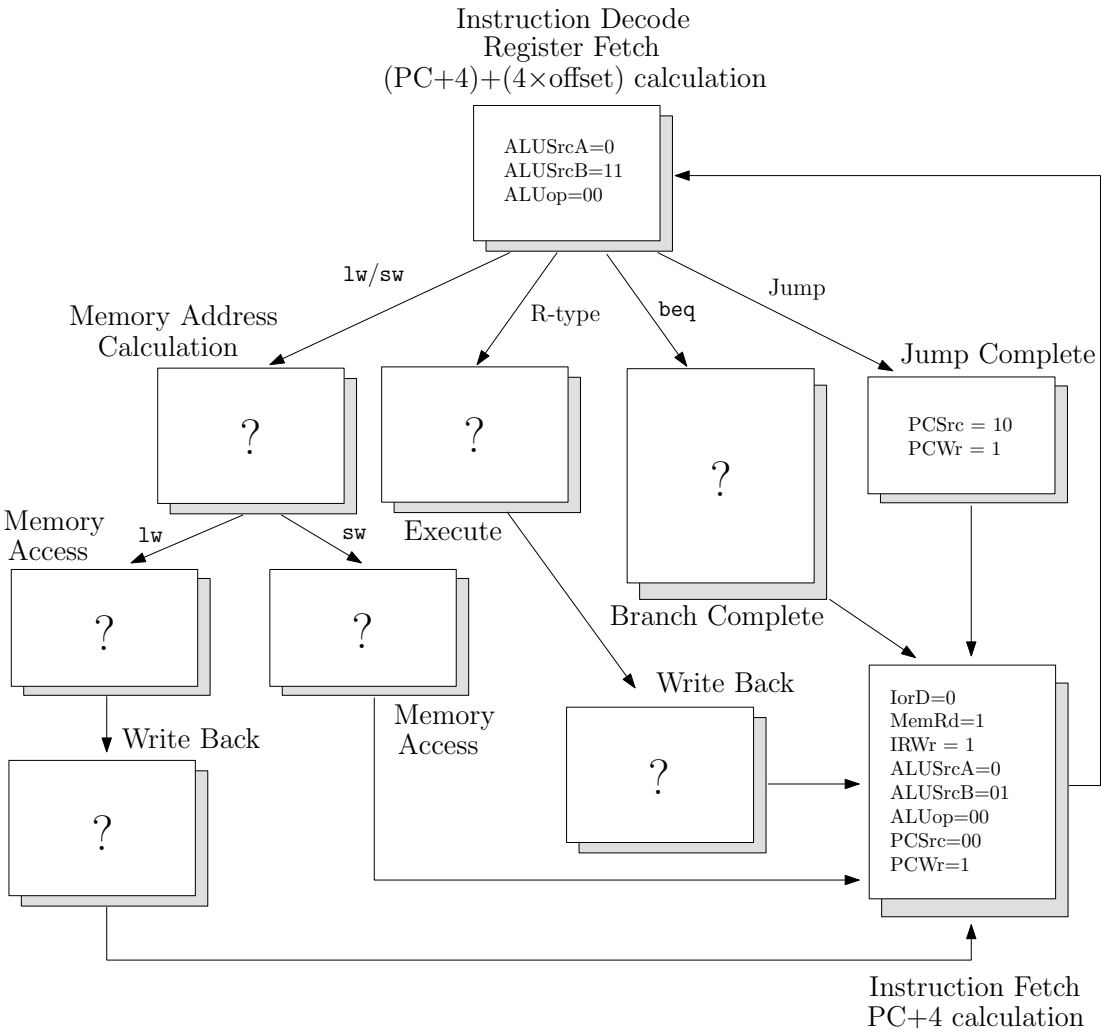

Figure 4.36: Control FSM, updated for jump complete.

Branch instruction complete

To complete the branch instruction `beq`, the control signals should be set up as follows:

1. ALUSrcA = 1 : feed A into the ALU;
2. ALUSrcB = 00 : feed B into the ALU;
3. ALUop = 01 : subtract A and B in the ALU;
4. PCWrCond = 1 : let zero output from the ALU enable PC update (or not);
5. PCSrc = 01 : apply branch target address to PC;

Depending on the outcome of testing the branch condition, there are two possible scenarios for a `beq` instruction:

- If A == B : update PC with the value in ALUout. Figure 4.37 emphasizes how the zero flag being 1 enables PC being updated (regardless of PCWr), since PCWrCond is set to 1.

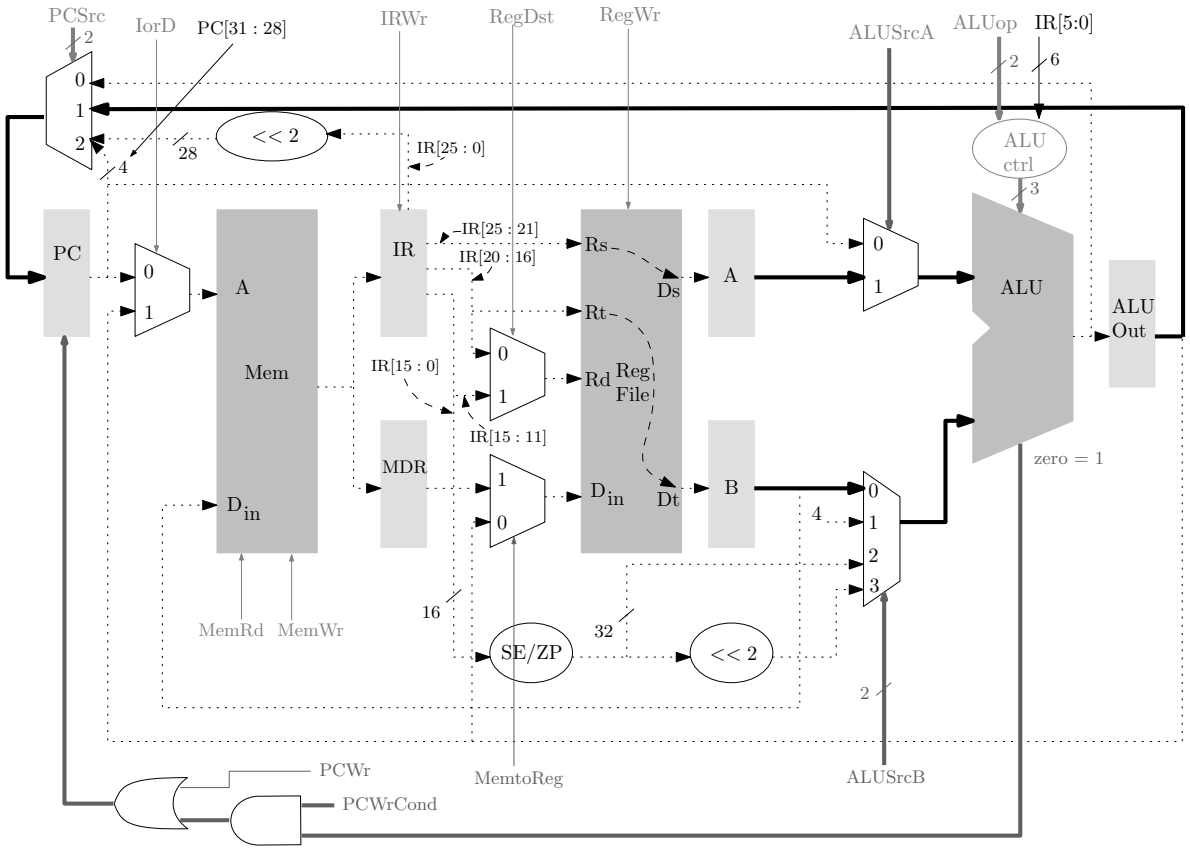

Figure 4.37: Datapath and relevant control signals for branch complete, if A==B.

- If A != B : do not update PC (to continue with the next instruction). Figure 4.38 shows how the zero flag being 0 prevents PC from being overwritten with the branch target address.

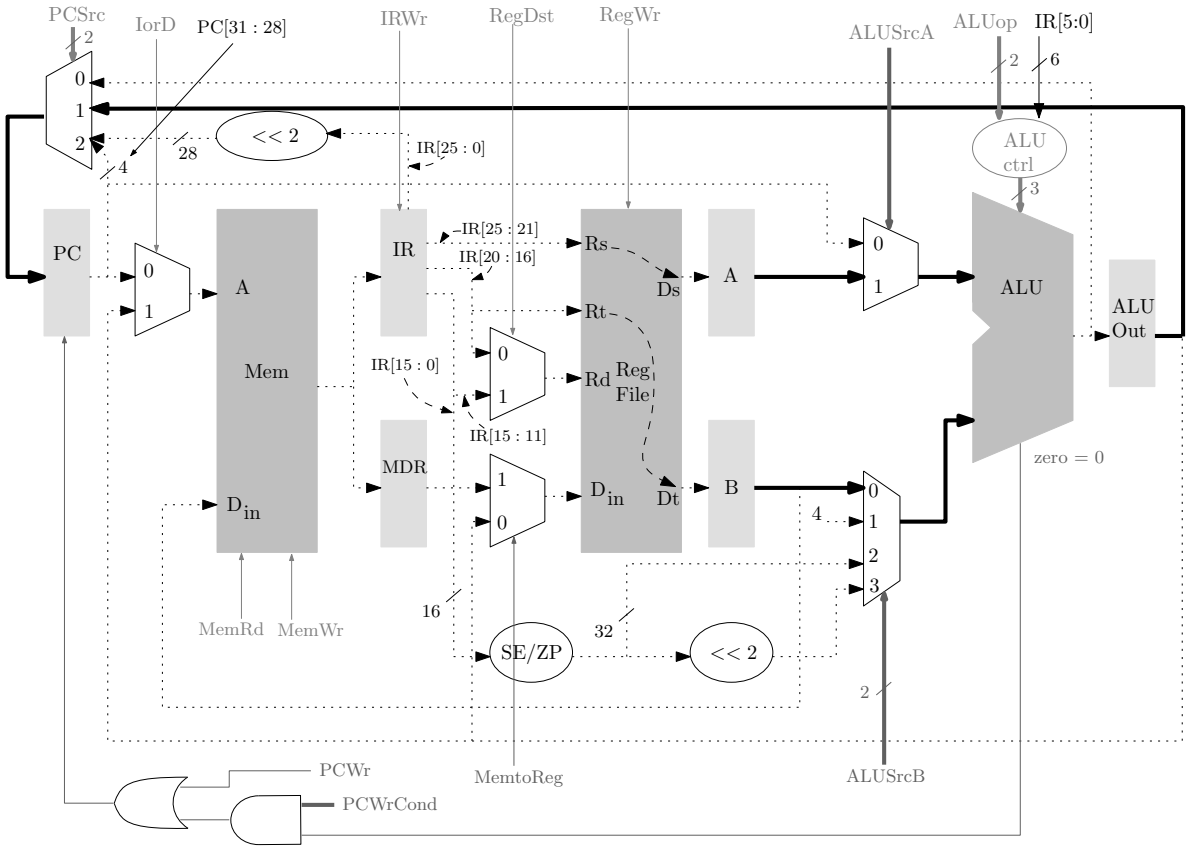

Figure 4.38: Datapath and relevant control signals for branch complete, if A!=B.

The updated FSM is given in Figure 4.39

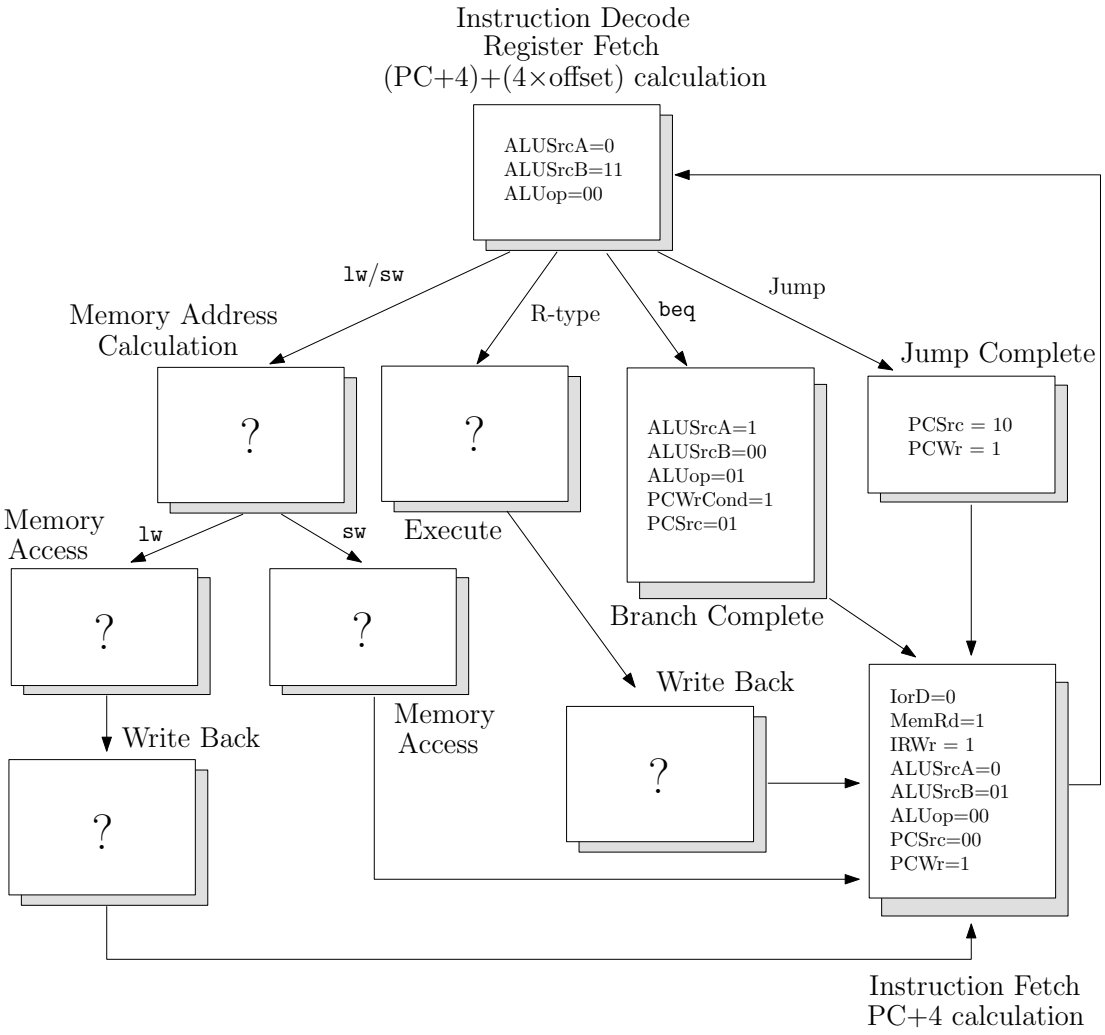

Figure 4.39: Control FSM, updated for branch complete.

R-type arithmetic-logic execute

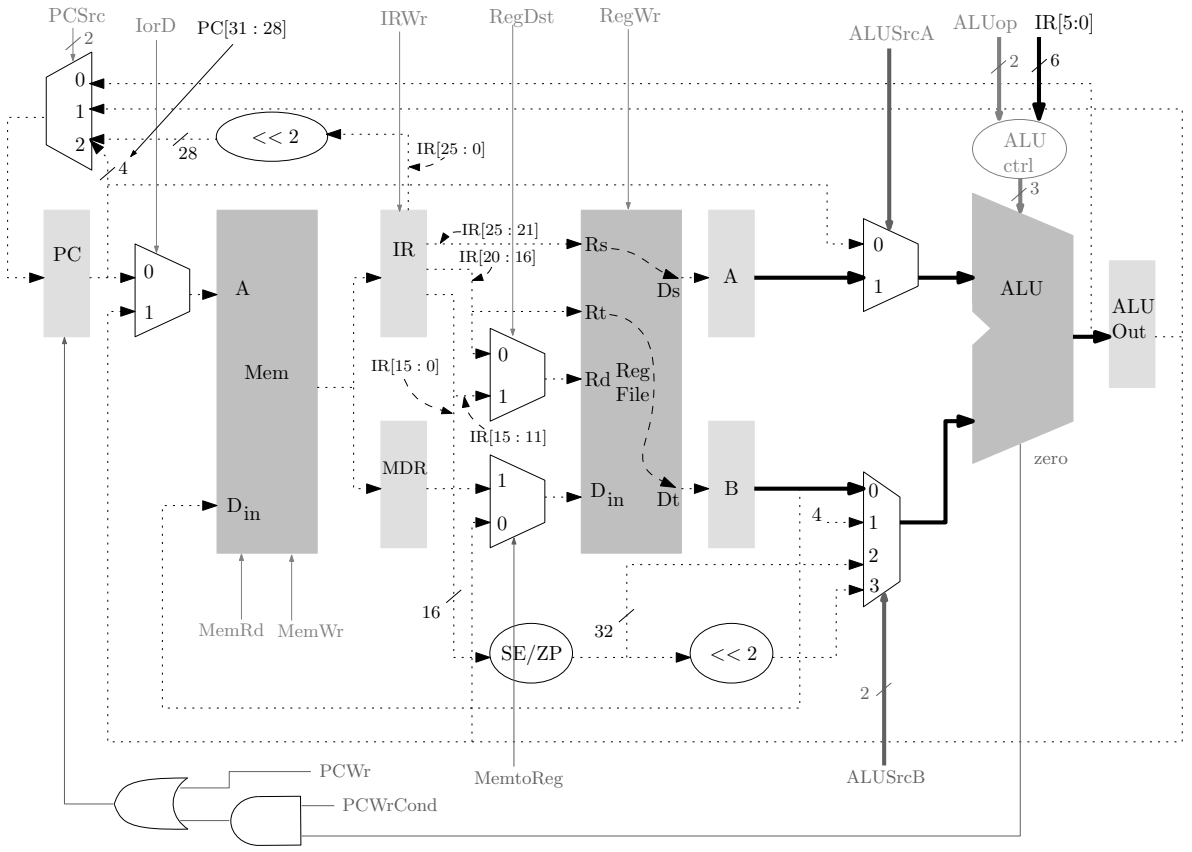

Figure 4.40: Datapath and relevant control signals for R-type arithmetic-logic execute.

Executing an R-type arithmetic-logic instruction requires:

1. ALUSrcA = 1 : feed A into the ALU;

2. ALUSrcB = 00 : feed B into the ALU;

3. ALUop = 10 : execute the arithmetic-logic operation with the ALU.

The state diagram for the control FSM has been updated in Figure 4.41.

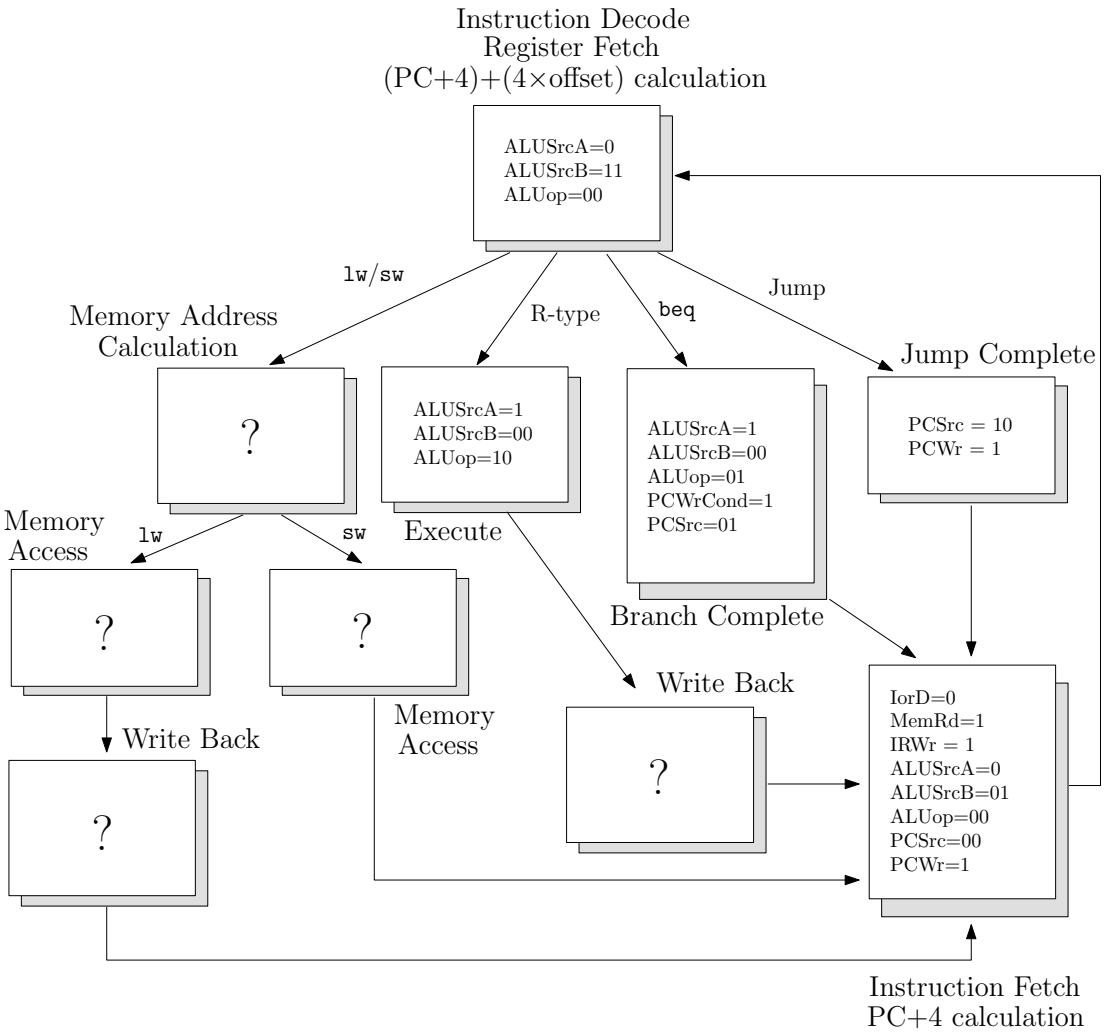

Figure 4.41: Control FSM, updated for R-type arithmetic-logic execute.

R-type arithmetic-logic write back

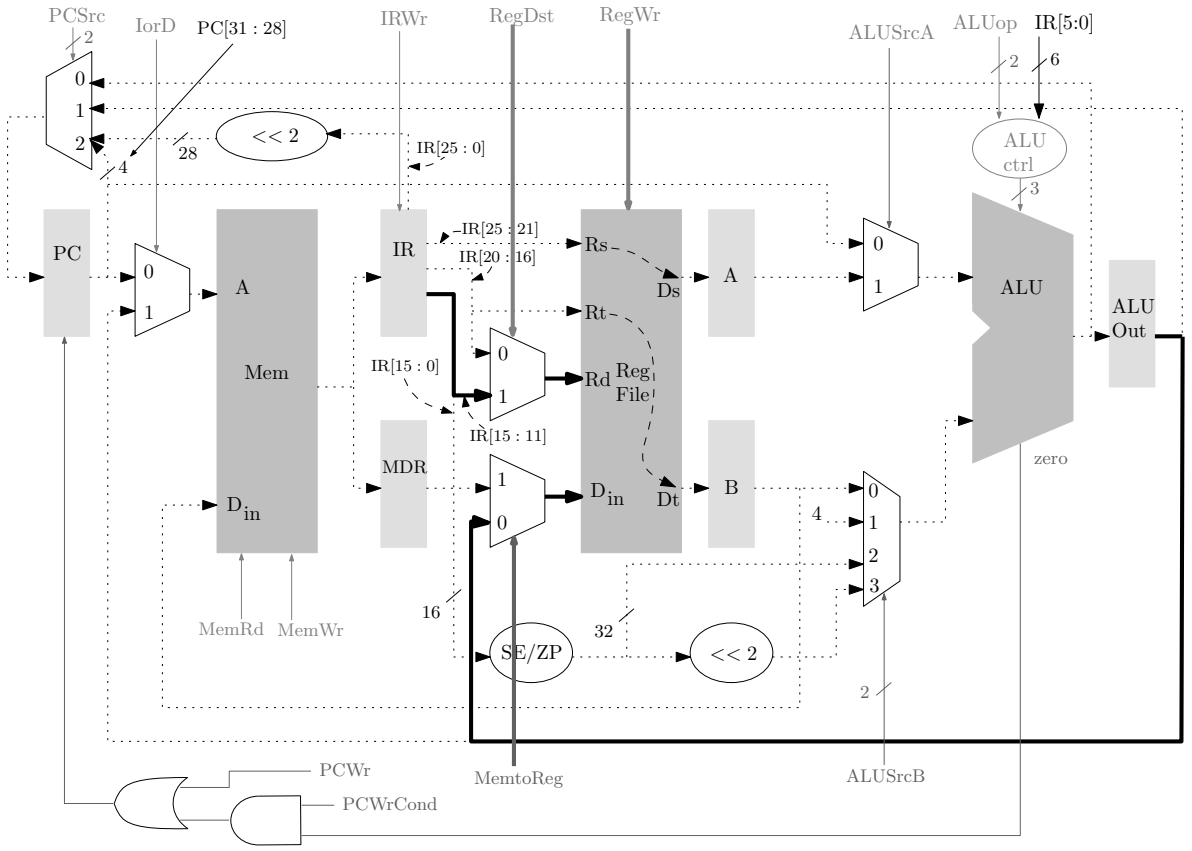

Figure 4.42: Datapath and relevant control signals for R-type arithmetic-logic write back.

The desired result, available in the ALUout register, is now stored in the register specified by the instruction's Rd field, IR[15 : 11]. The control signals required are:

1. RegDst = 1 : apply IR[15 : 11] to the register file's Rd input;

2. MemtoReg = 0 : apply ALUout to the register file's Din input;

3. RegWr = 1 : enable updating the appropriate register.

The updated FSM is shown in Figure 4.43.

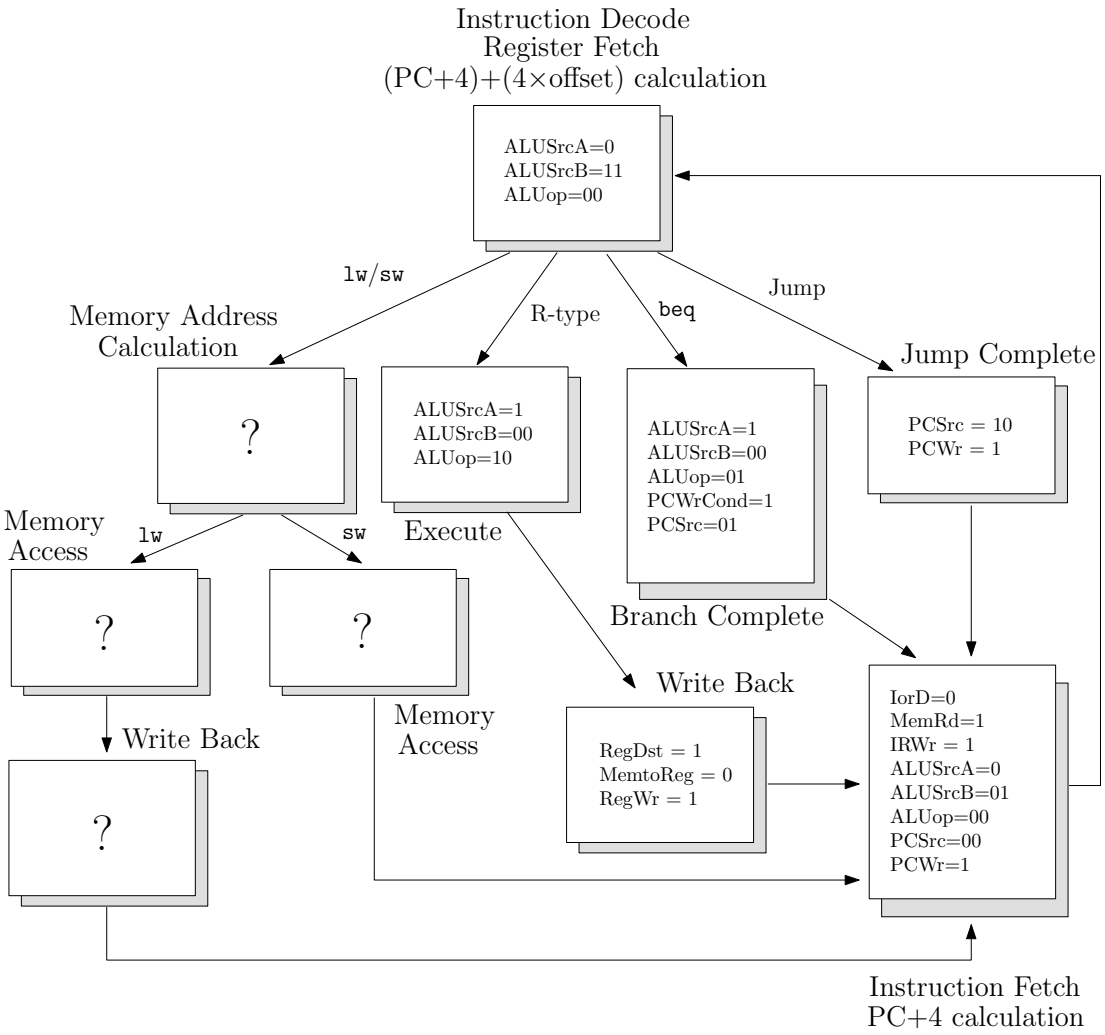

Figure 4.43: Control FSM, updated for R-type arithmetic-logic write back.

Data transfer memory address calculation

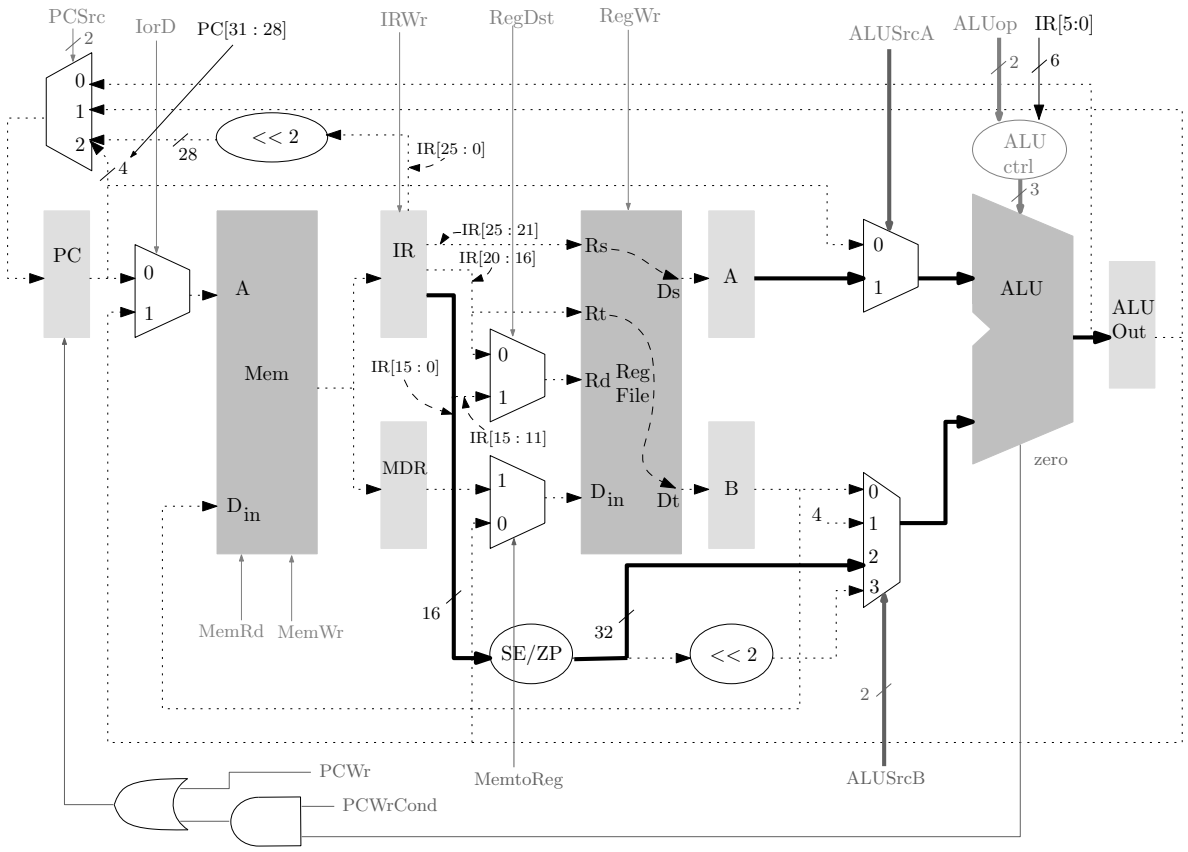

Figure 4.44: Datapath and relevant control signals for data transfer memory address calculation.

Adding the content of the register specified by the instruction's Rs field (available in register A) and the offset in IR[15:0] provides the memory address to access. This requires the following control signals:

1. ALUSrcA = 1 : feed A into the ALU;

2. ALUSrcB = 10 : feed the sign-extended offset into the ALU;

3. ALUop = 00 : add the two ALU inputs.

The FSM is updated in Figure 4.45.

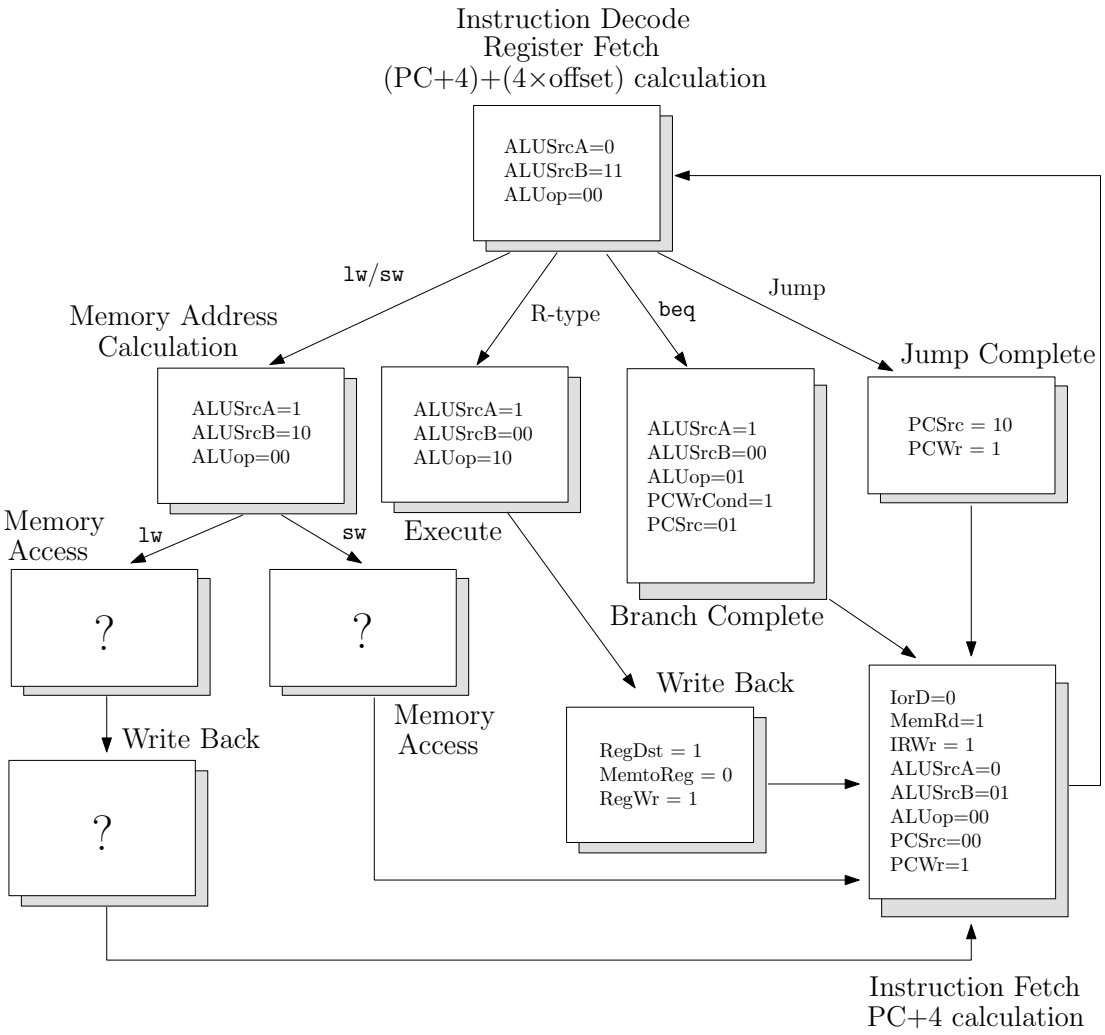

Figure 4.45: Control FSM, updated for data transfer memory address calculation.

Data transfer memory access

The memory access step for `sw` is illustrated in Figure 4.46: the contents of register B are applied at the input Din of the memory, to be written at the address provided by the register ALUout (applied to the input A of the memory). This requires the following control signals:

1. IorD = 1 : apply address provided by ALUout to the memory;
2. MemWr = 1 : enable writing into memory.

Enabling writing into memory allows the input Din to overwrite the contents of the memory at address A.

For `lw`, the memory access step is illustrated in Figure 4.47: the contents of the memory location at the address provided by the register ALUout (applied to the input A of the memory) are provided at the output of the memory and stored into the register MDR. This requires the following control signals:

1. IorD = 1 : apply address provided by ALUout to the memory;
2. MemRd = 1 : enable reading from memory.

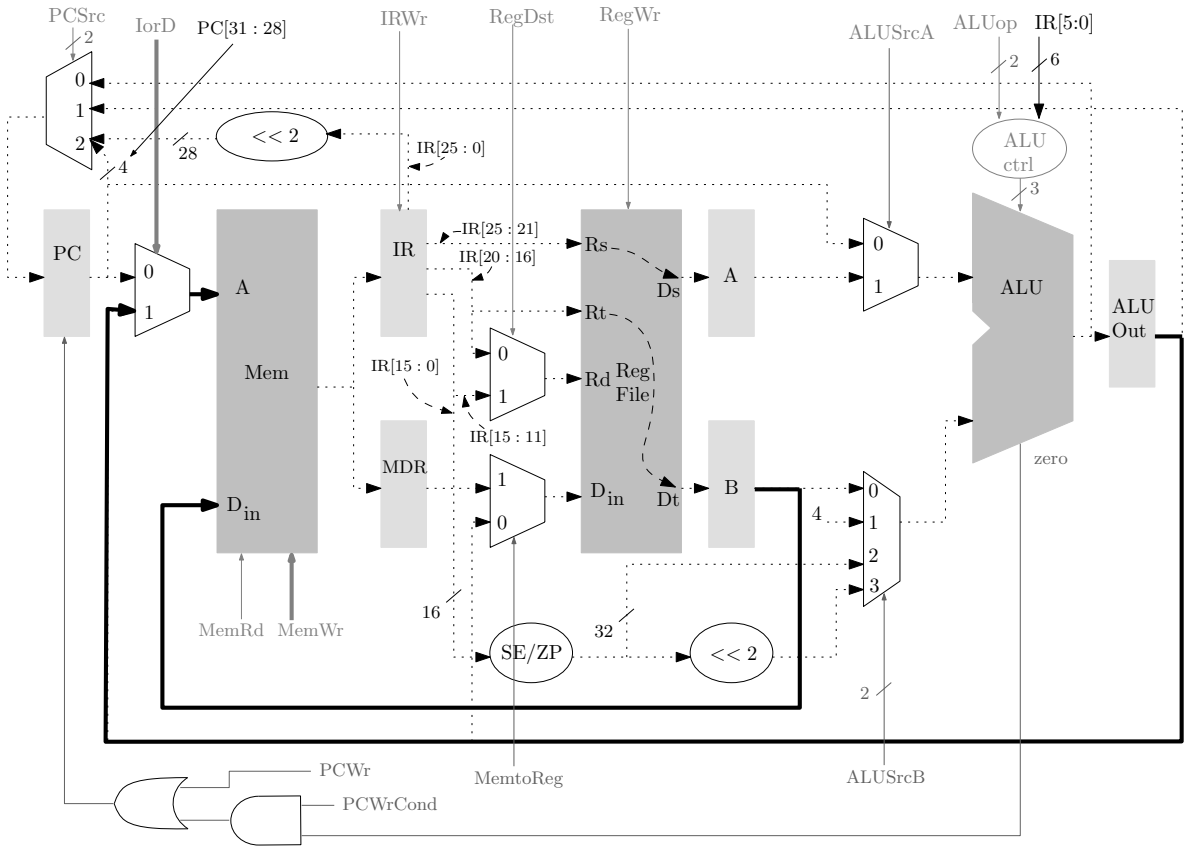

Figure 4.46: Datapath and relevant control signals for `sw` memory access.

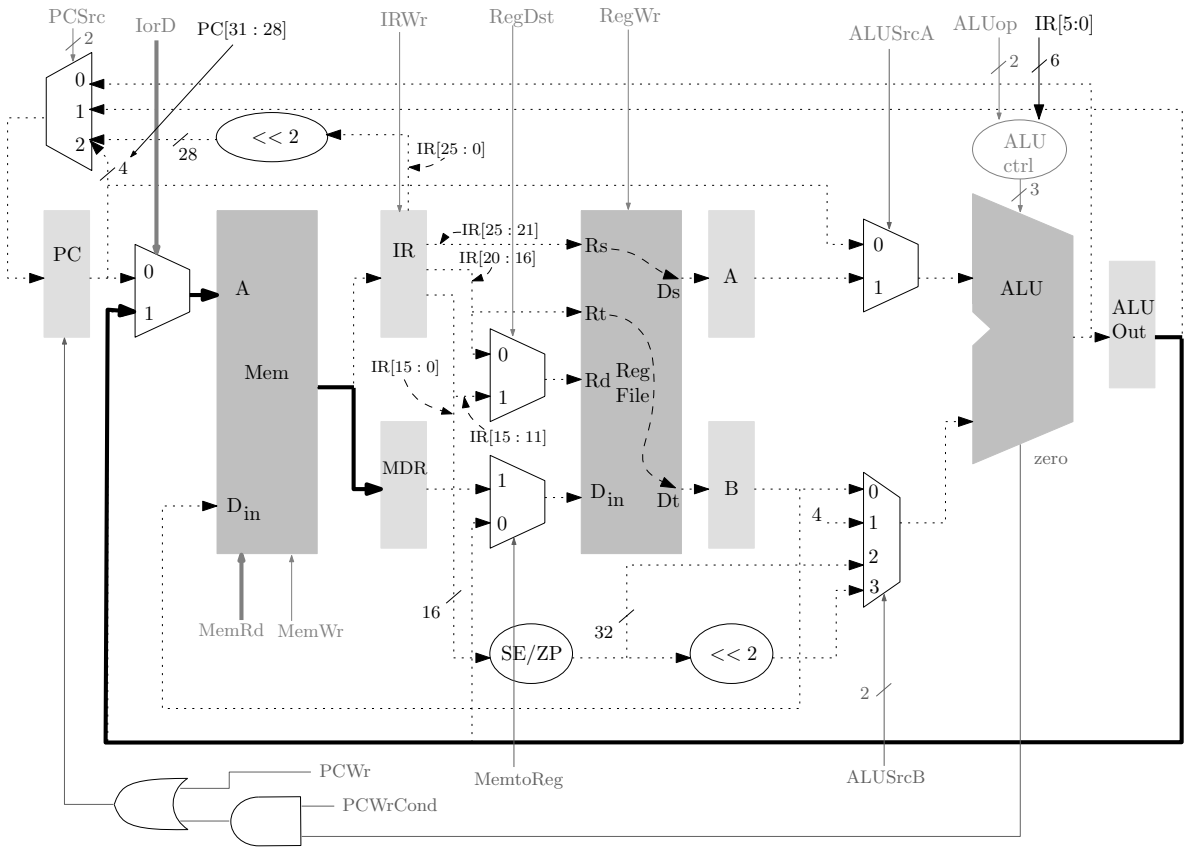

Figure 4.47: Datapath and relevant control signals for `lw` memory access.

Note that, even in case of an `lw` instruction, the contents of register B are applied to the Din input of the memory. However, since MemWr is now 0, the input Din is ignored. The updated FSM state diagram is shown in Figure 4.48.

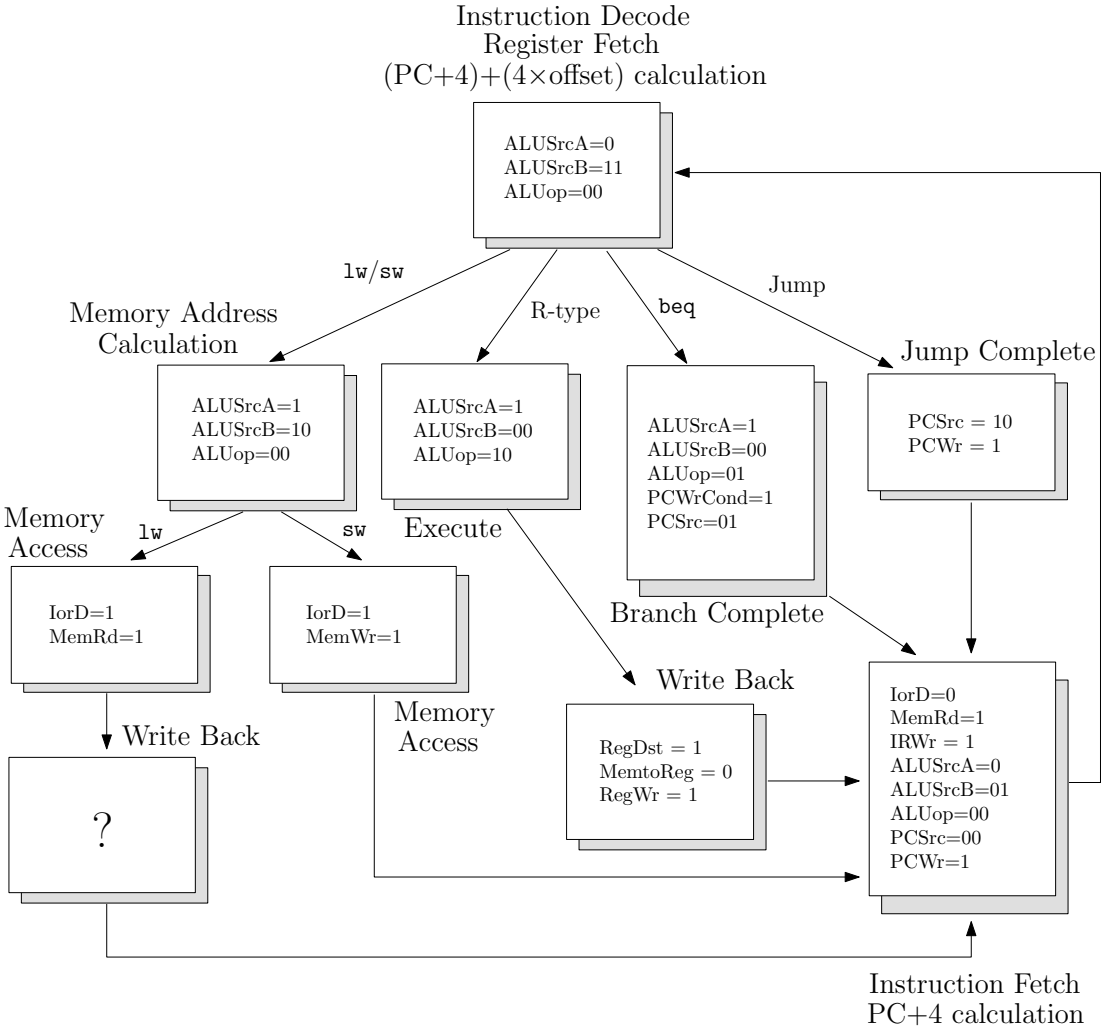

Figure 4.48: Control FSM, updated for data transfer memory access.

`lw` write back

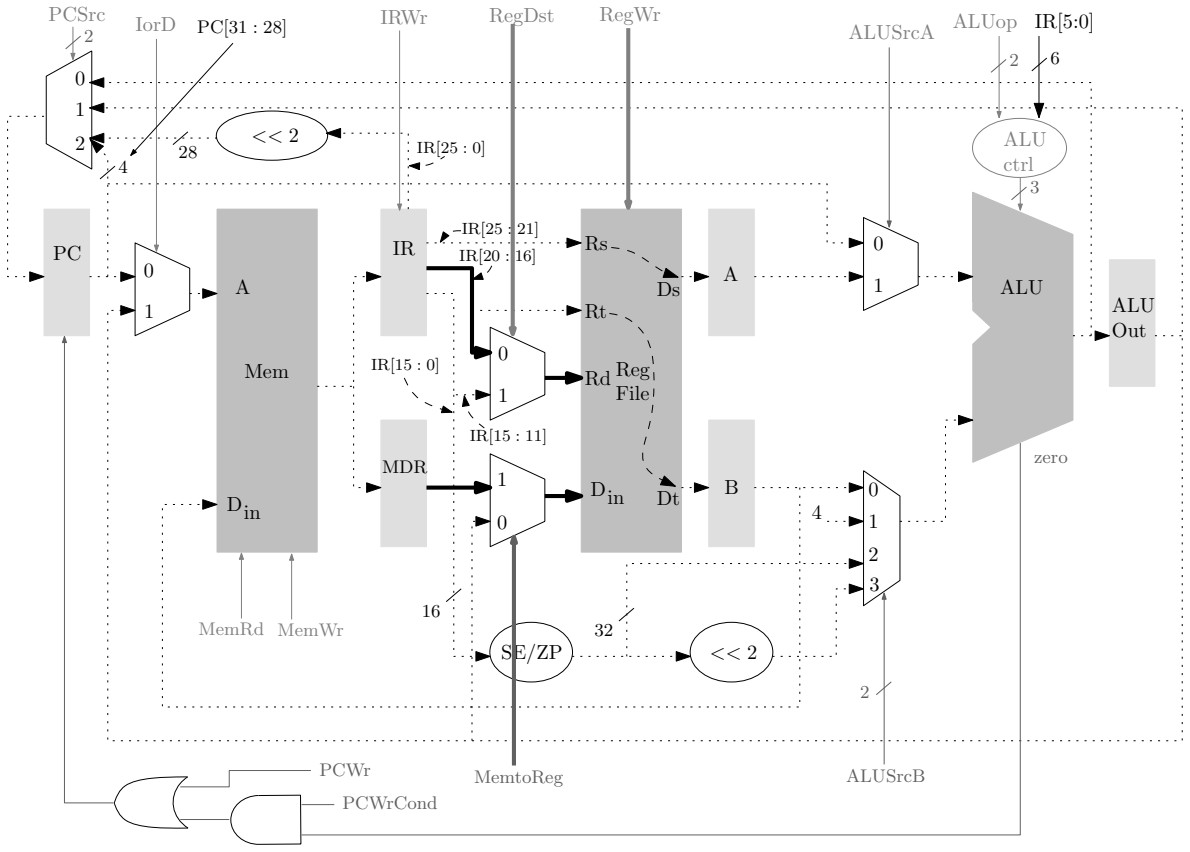

Figure 4.49: Datapath and relevant control signals for `lw` write back.

The data retrieved from memory, available in the MDR register, is now applied to the Din input of the register file, to be stored in the register specified by IR[20 : 16] (applied to the Rd input of the register file). The required control signals are set up as follows:

1. RegDst = 0 : select IR[20 : 16] as Rd input of the register file;

2. MemtoReg = 1 : apply contents of MDR to Din input of the register file;

3. RegWr = 1 : enable updating the appropriate register.

The completed FSM for the control unit is represented in Figure 4.50.

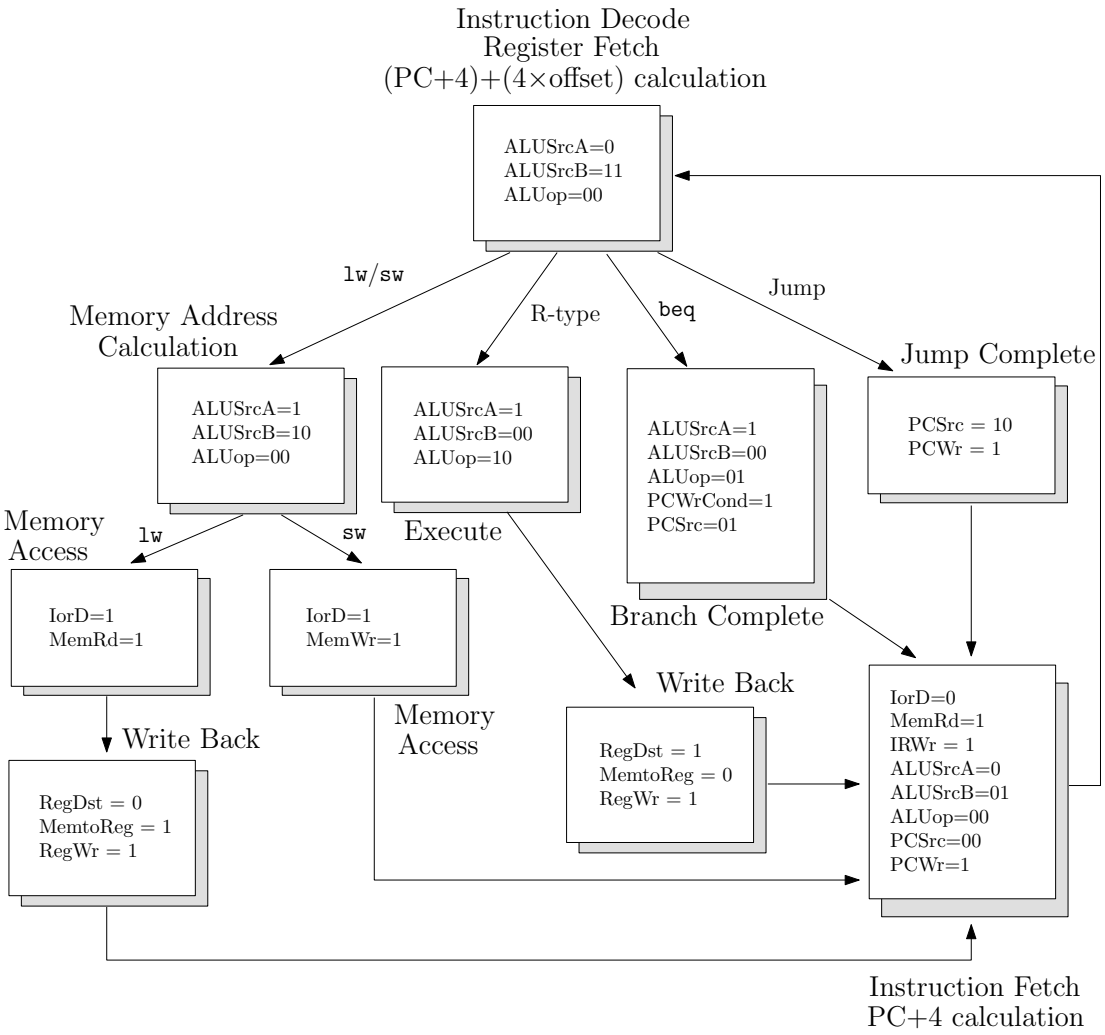

Figure 4.50: The control FSM.