Multicycle RISC-V Processor

Learning Goal: Simple multicycle processor architecture.

Requirements: Verilator, GTKWave

1 Introduction

In this lab you will implement a multicycle RISC-V (RV32I) processor. You will implement it step-by-step beginning with a **CPU** that executes a few basic instructions and extending it progressively to cover all the requested functionalities of the RISC-V standard. You will also use some of the components you built in the previous sessions.

> **Important!** Please read the entire assignment before starting to implement your CPU.

2 32-bit Register File

The primary component you will implement for this CPU lab is the **Register File**. This is a crucial element of the CPU architecture. Figure 1 illustrates the module of the **Register File**. The **Register File** contains 32 registers, each 32 bits wide. It's important to note that the first register (i.e., the register at address 0) always maintains a fixed value of 0.

Figure 1: Module of the **Register File**.

2.1 Reading the Register File

For this **Register File**, the read process is *asynchronous*. The inputs **aa** and **ab** select which two registers to read. Their values are sent on the outputs **a** and **b**, respectively. The diagram in Figure 2 illustrates a typical read process.

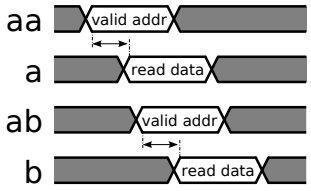

Figure 2: A read process of the **Register File**.

2.2 Writing to the Register File

The write process is *synchronous*. Input **wren** enables writing **wrdata** in the register addressed by **aw**. Writing a value in the register at address 0 has no effect – its value is fixed to 0. The diagram of Figure 3 illustrates a typical write process. It has essentially the same behaviour as flip-flops. The address, the data and **wren** are set during cycle 0. At the rising edge of the clock, the data is saved in the **Register File** at address **aw**. A new writing process can start during cycle 1.

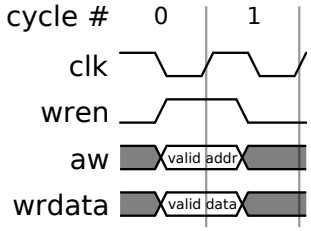

Figure 3: A write process of the **Register File**.

2.3 Exercise

Implement the **Register File** described in Section 2. For that, you can use an array of registers. The following code gives an example of the Verilog module structure and array declaration.

```
module register_file (
  ...
);
    reg [31:0] reg_array_r [32];
    // Rest of the implementation...
endmodule
```

To read a value from the array, you can use a simple assignment with the index:

```
assign a_o = reg_array_r[aa_i];
assign b_o = reg_array_r[ab_i];
```

- Implement the **Register File** in the file named `register_file.v`.
  - Do not forget that register 0 must have a fixed value of 0.
- To simulate the **Register File**, use Verilator and write your own testbench inside the `testbench` directory. Name it `tb_register_file.v`.

- Compile your Verilog code using Verilator:

```
verilator --binary --trace -Wno-fatal --top-module tb_register_file
↪  -o Vtb_register_file testbench/tb_register_file.v
↪  verilog/register_file.v
```

- Run the simulation:

```
./obj_dir/Vtb_register_file
```

- Use GTKWave to visualize the simulation results:

```
gtkwave dump/tb_register_file.vcd
```

3 Multicycle CPU Description

The first implementation of the **CPU** only executes several ALU operations (e.g., **`addi`**, **`and`**), one handy instruction called **`lui`** and the **`lw`** and **`sw`** instructions. A **`break`** instruction is also used to stop the execution of the program.

To execute an instruction, the multicycle **CPU** needs 4 to 5 cycles, depending on the instruction. Figure 4 shows the basic state machine of the **CPU**'s controller, which you will progressively extend. It illustrates the different steps of the execution of an instruction.

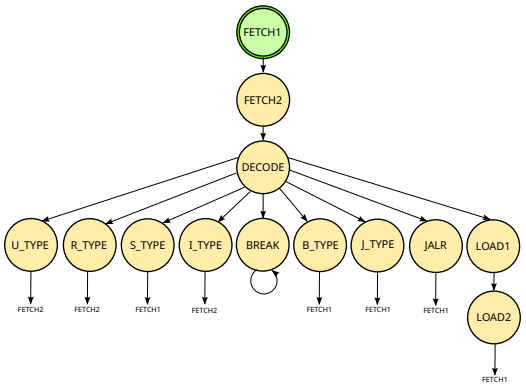

Figure 4: The state machine of the **CPU**'s controller.

During **FETCH1** and **FETCH2**, the **CPU** reads the next instruction to execute. During **DECODE**, the **CPU** identifies the instruction and determines the next state. During the next states, the instruction is executed. These last states are collectively called *Execute* states. You will add new *Execute* states as you implement new instruction types.

The next subsections describe each state, and progressively introduce the internal units and signals of the **CPU**.

3.1 FETCH1

During this first state of the execution, the address of the next instruction is prepared for reading. The instruction word will be available during the next cycle (recall that the RAM's reading process has a one-cycle latency: **FETCH1** prepares the address to be read, and **FETCH2** receives the corresponding instruction during the next cycle). Figure 5 shows the components used for the **FETCH1** state.

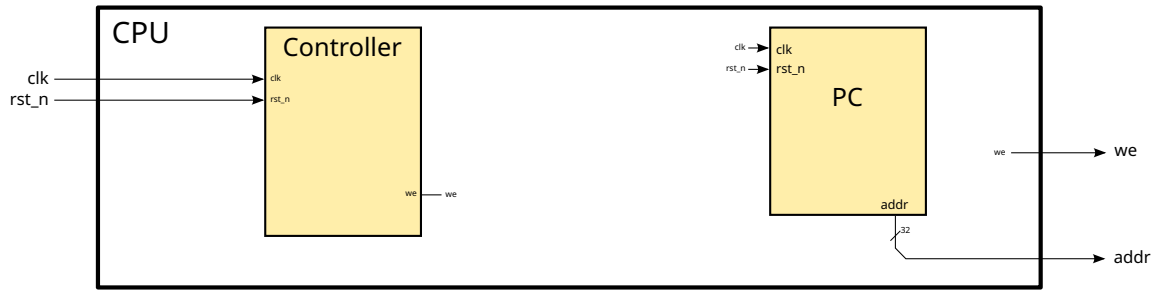

Figure 5: Components used for the **FETCH1** state.

The **Controller** controls the state machine. The input **rst_n**, synchronous and active low, initializes the state machine to **FETCH1**. The **PC** holds the address of the next instruction. The address is stored in a 32-bit register. The address must always be valid, thus the two least significant bits should remain at '0'.

- The input **clk** is the clock signal.
- The output **addr** is the current value of the address that is read from the memory.
- The input **rst_n** initializes the address register to 0x80000000.

During FETCH1, the memory read operation is implicitly initiated when the **we** (write enable) signal is low. The controller ensures that **we** is set to 0 during this state to initiate the read operation for the next instruction.

3.2 FETCH2

During the **FETCH2** state, the instruction word is read from the input **rdata** and saved in a register. The **Controller** enables the **PC**, so that it increments the address by 4. Figure 6 shows the components used for the **FETCH2** state.

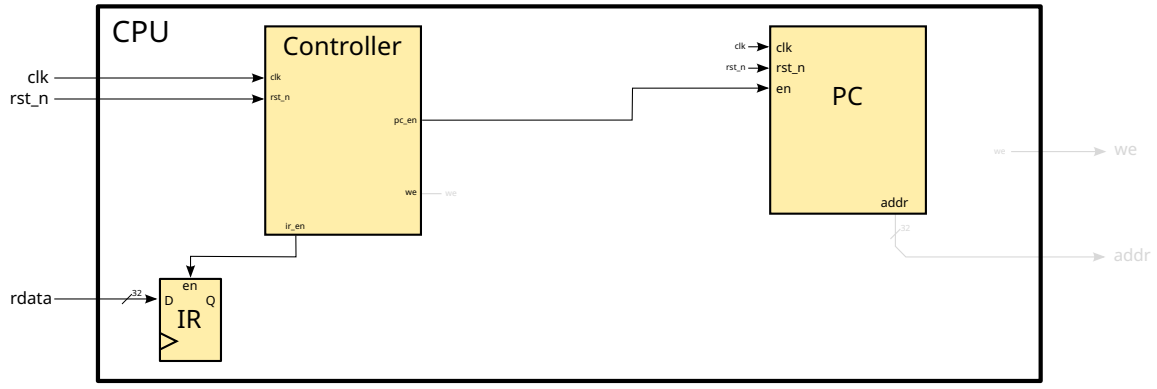

Figure 6: Components used for the **FETCH2** state.

The **Instruction Register** (**IR**) is a 32-bit register that stores the instructions coming from the memory.

- The input **clk** is the clock signal.
- The output **q** is the current value of the register.
- The input **en** enables to write the input **d** in the register at the next rising edge of the clock. In other words, at every rising edge of **clk**, the value of **d** is passed over to **q** if **en** is enabled.

3.3 DECODE

During the **DECODE** state, the **Controller** reads the opcode of the instruction to identify the current instruction and determines the next *Execute* state. The RISC-V instructions are progressively described in the following subsections. Figure 7 shows the components used for this state.

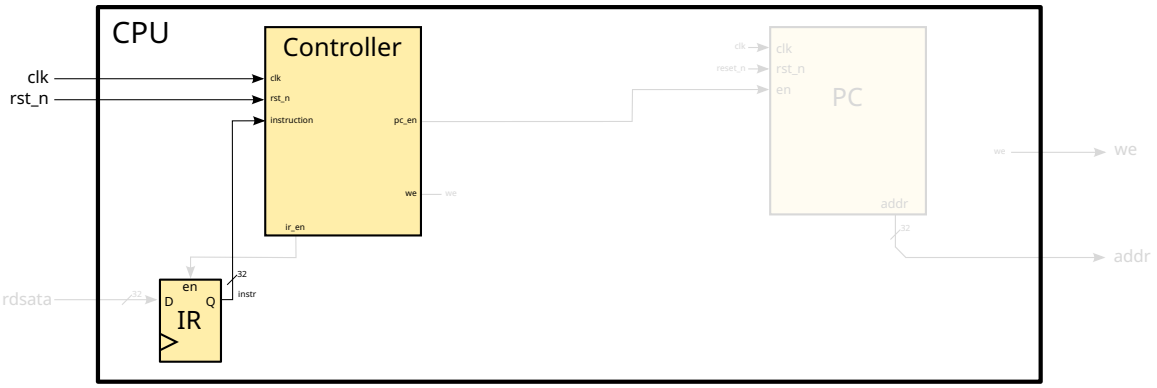

Figure 7: Components used for the **DECODE** state.

3.4 I_TYPE

The **I_TYPE** state executes operations between a register and an *immediate* value that is embedded in the instruction word, and saves the result in another register. Such instructions with an embedded 12-bit

immediate value are **I-type** (Immediate type) instructions in RV32I. Figure 8 shows the general **I-type** instruction format in detail.

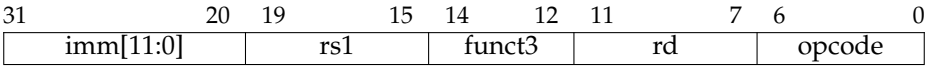

Figure 8: The general **I-type** instruction format in RV32I.

The fields **rs1** and **rd** are register addresses for the source and destination registers, respectively. The field **imm[11:0]** is the 12-bit immediate value. The **funct3** field provides additional information about the specific operation, and **opcode** identifies the instruction type.

It's important to note that while `lw` (load word), `jalr` (jump and link register) and `break` (repurposed `ebreak`) instructions are I-type instructions, they are handled in separate states due to their unique behaviors. The **I_TYPE** state primarily deals with immediate arithmetic and logical operations.

In RV32I, all immediates are sign-extended to 32 bits. This sign extension is handled directly in the **Controller**. The sign extension is performed by replicating the most significant bit (bit 11) of the immediate field to fill the upper 20 bits.

During the **I_TYPE** state, the **alu_op** signal is set by the **Controller** to perform the required operation in the **ALU**. The **alu_op** signal is derived from the **opcode** and **funct3** fields of the instruction. Figure 9 shows the components used for this state.

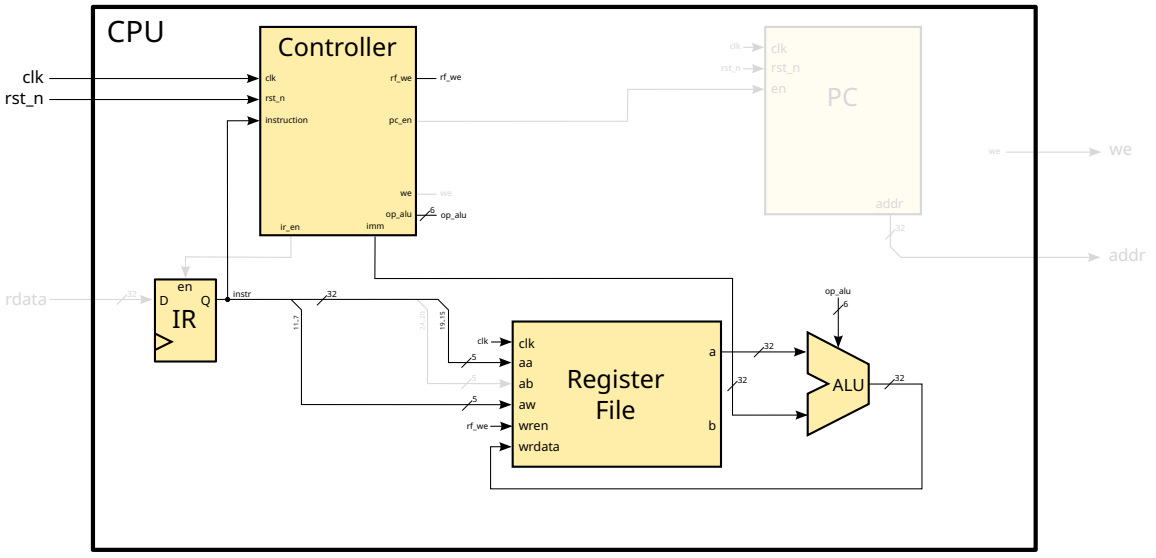

Figure 9: Components used for the **I_TYPE** state.

The **Register File** and the **ALU** are the same units that you have implemented during the previous labs. The **Controller** selects the operation to execute in the **ALU** with the signal **alu_op**. The **alu_op** signal depends on the current instruction (e.g., an *addition* for `addi`, a *logical AND* for `andi`, or a *logical right shift* for `srli`).

The **ALU** operation codes are summarized in Table 1, which has been updated to reflect the RV32I instruction set.

Table 1: ALU operations and their encoding.

Operation	Operation Type	Opcode
$A + B$	Add/Sub	000$\phi\phi\phi$
$A - B$		001$\phi\phi\phi$
$A = B$	Comparison	011000
$A \neq B$		011001
$A < B$ (signed)		011100
$A \geq B$ (signed)		011101
$A < B$ (unsigned)		011110
$A \geq B$ (unsigned)		011111
$A \oplus B$ (XOR)	Logical	100100
$A \vee B$ (OR)		100110
$A \wedge B$ (AND)		100111
$A \ll B$ (SLL)	Shift	110001
$A \gg_l B$ (SRL)		110101
$A \gg_a B$ (SRA)		111101

ϕ = *don't care*, **0**/**1** = special bit

3.5 R_TYPE

The **R_TYPE** state executes operations between two registers and saves the result in a third register. Such instructions with three register addresses are **R-type** (Register type) instructions. Figure 10 shows the general **R-type** instruction format in detail.

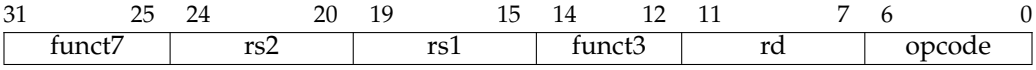

Figure 10: The general **R-type** instruction format in RV32I.

The fields **rs1** and **rs2** are source register addresses, and **rd** is the destination register address. The **funct7** and **funct3** fields provide additional information to specify the exact operation. The **opcode** field identifies the instruction type.

During the **R_TYPE** state, the **alu_op** signal is set by the **Controller** to perform the required operation in the **ALU**. The **alu_op** is determined based on the **opcode**, **funct3**, and **funct7** fields of the instruction.

Figure 11 shows the components used for the **R_TYPE** state.

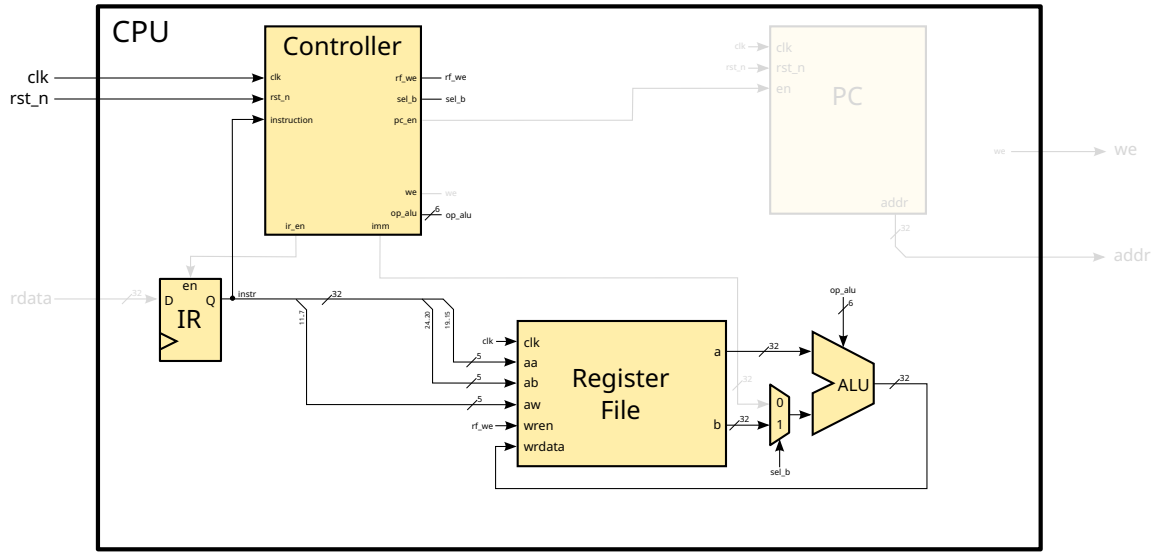

Figure 11: Components used for the **R_TYPE** state.

The **Register File** reads the values of the two source registers (**rs1** and **rs2**) and writes the result to the destination register (**rd**). The **ALU** performs the operation specified by **alu_op** on the two source register values. The result of the **ALU** operation is then written back to the **Register File** at the address specified by **rd**.

The **Controller** manages the entire process, interpreting the instruction and generating the appropriate control signals for the **Register File** and **ALU**.

3.6 U_TYPE

The **U_TYPE** state executes the LUI (Load Upper Immediate) instruction, which is a U-type instruction in RV32I. Figure 12 shows the U-type instruction format in detail.

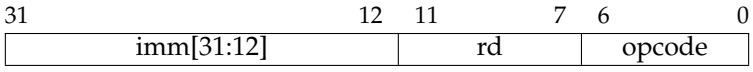

Figure 12: The U-type instruction format in RV32I.

The LUI instruction loads a 20-bit immediate value into the upper 20 bits of the destination register, setting the lower 12 bits to zero. This is useful for constructing large constants efficiently.

Table 2 describes the LUI instruction.

Instruction	opcode	Description
`lui rd, imm`	0110111	rd = imm $\ll$ 12

Table 2: The LUI instruction.

During the **U_TYPE** state, the following operations occur:

- The 20-bit immediate value from the instruction is shifted left by 12 bits, effectively placing it in the upper 20 bits of a 32-bit word.
- This 32-bit value is then written to the destination register (`rd`).
- No ALU operation is required for this instruction.

Figure 13 shows the components used for the **U_TYPE** state.

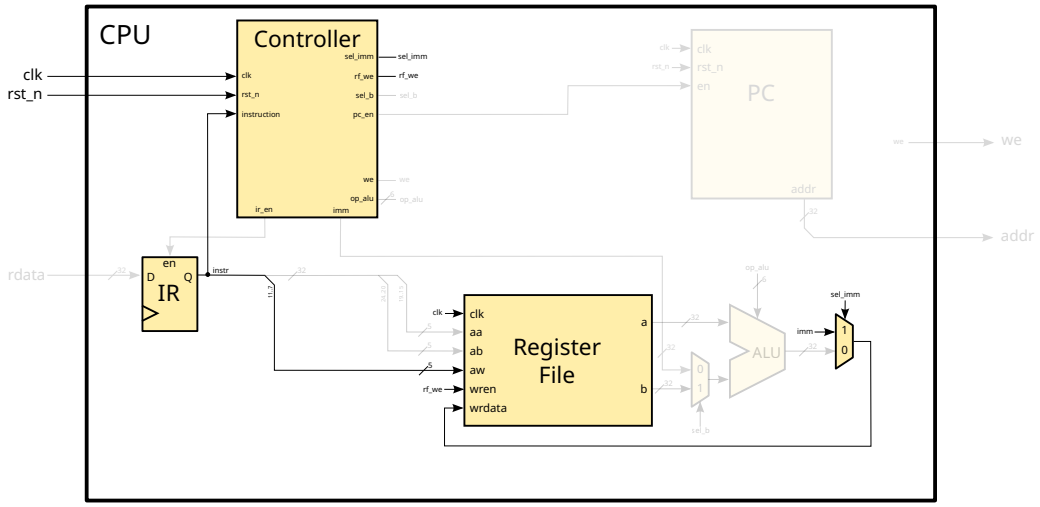

Figure 13: Components used for the **U_TYPE** state.

The **sel_imm** signal is set to select the immediate value instead of the ALU result. The **rf_we** signal enables writing to the register file, storing the computed immediate value in the destination register specified by `rd`.

LUI is often used in conjunction with an ADDI instruction to create arbitrary 32-bit constants or to form complete addresses for accessing memory. For example:

```
lui  x1, 0x12345    # x1 = 0x12345000
addi x1, x1, 0x678  # x1 = 0x12345678
```

This combination allows for the efficient creation of any 32-bit constant using just two instructions.

3.7 LOAD

The `lw` (load word) instruction is an **I-type** instruction in RV32I. Figure 14 shows the **LOAD** instruction format in detail.

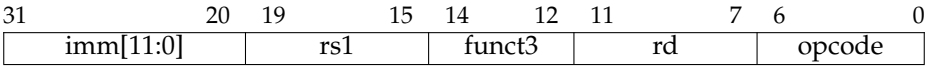

Figure 14: The **LOAD** instruction format in RV32I.

The load operation takes 2 cycles to complete due to the memory read latency. This process is divided into two states: **LOAD1** and **LOAD2**.

Figure 15 shows the components used for the **LOAD1** state.

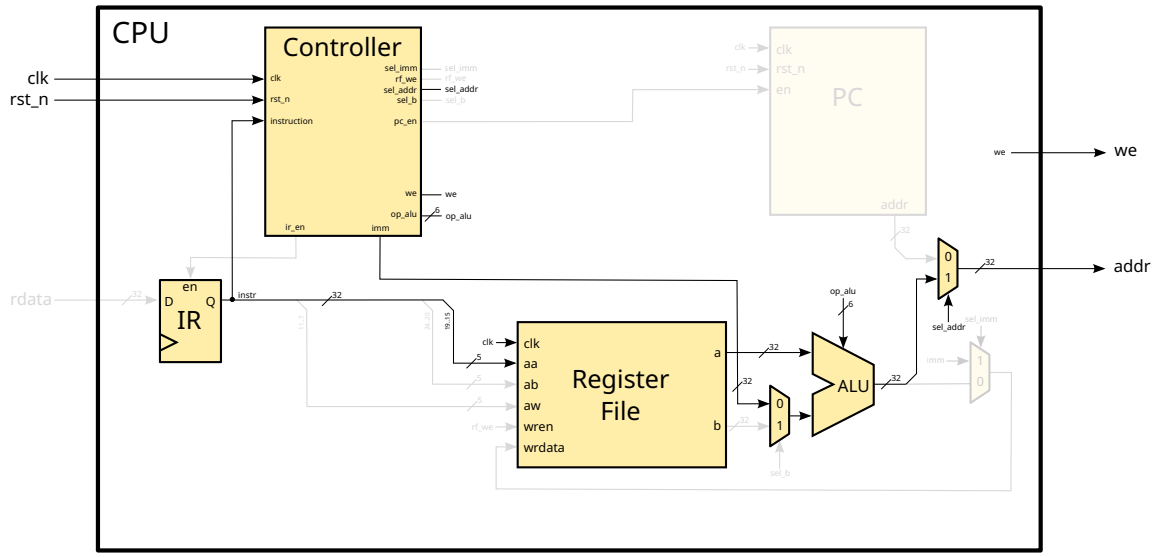

Figure 15: Components used for the **LOAD1** state.

During the **LOAD1** state:

- The address to read is computed by the ALU (adding the sign-extended immediate value to the value in **rs1**).
- The **sel_addr** signal is set to select the ALU result as the memory address.
- The memory read operation is initiated (implicitly, by keeping the **we** signal low).

Figure 16 shows the components used for the **LOAD2** state.

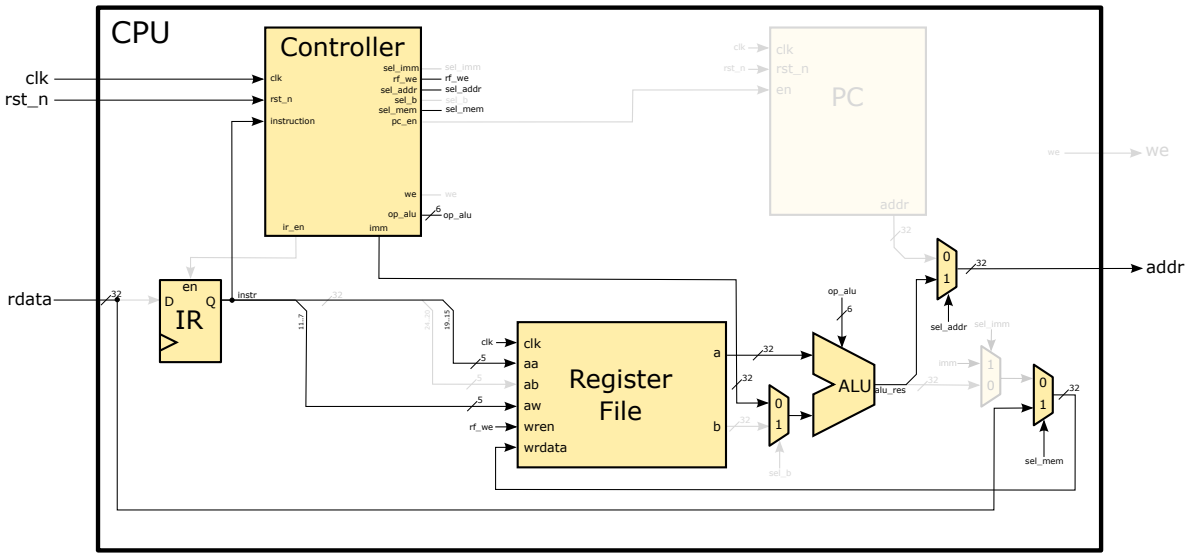

Figure 16: Components used for the **LOAD2** state.

During the **LOAD2** state:

- The signals from **LOAD1** are maintained:
  - The **imm_o** signal continues to provide the sign-extended immediate value.
  - The **sel_addr** signal remains set to select the ALU result as the memory address.
- The data read from memory becomes available on the **rdata** input.
- The **sel_mem** signal is set to select the memory data.
- The **rf_we** signal is activated to enable writing to the **Register File**.
- The selected data is written to the **Register File** at the address specified by **rd**.

By maintaining the address-related signals from **LOAD1**, the **LOAD2** state ensures that the decoder and multiplexer continue to select the correct **rdata** source. This approach guarantees that the memory read operation initiated in **LOAD1** completes successfully in **LOAD2**, with the correct data being written to the appropriate register.

The **Controller** manages the entire process, generating the appropriate control signals for the **ALU**, memory interface, and **Register File** in each state.

3.8 S_TYPE

The `sw` (store word) instruction is an **S-type** instruction in RV32I. Figure 17 shows the **S-type** instruction format in detail.

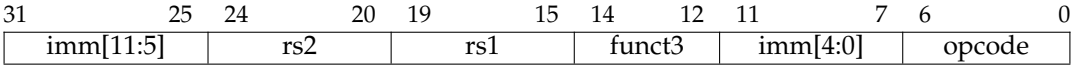

Figure 17: The **S-type** instruction format in RV32I.

In S-type instructions, the immediate value is split into two parts:

- **imm[11:5]** occupies bits 31-25 of the instruction
- **imm[4:0]** occupies bits 11-7 of the instruction

To form the full 12-bit immediate, these parts are concatenated as **imm[11:5] —— imm[4:0]**. This immediate is then sign-extended to 32 bits for use in address calculation.

During the **S_TYPE** state:

- The **ALU** computes the memory address by adding the sign-extended immediate to the value in **rs1**.
- The data to be stored (from **rs2**) is placed on the **wdata** output.
- The **Controller** activates the **we** (write enable) output signal to initiate a write operation.

Figure 18 shows the components used for the **S_TYPE** state.

The **Controller** manages the process by:

- Forming the correct immediate value from the instruction's fields

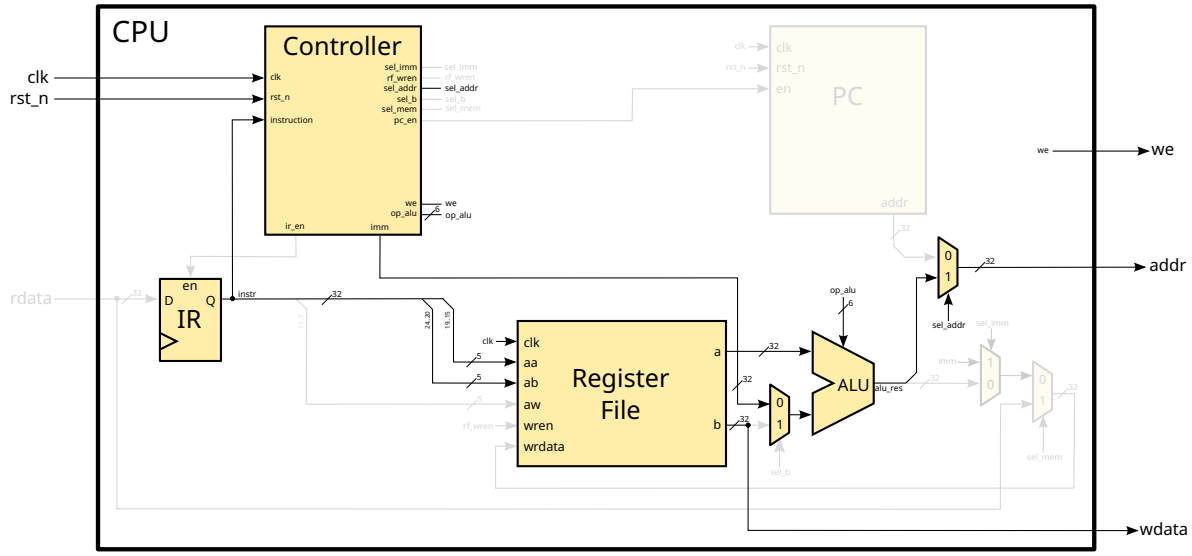

Figure 18: Components used for the **S_TYPE** state.

- Setting **sel_addr** to select the ALU result as the memory address
- Activating the **we** signal for the duration of the state

Unlike load operations, store operations complete in a single cycle, as there's no need to wait for data to be read from memory.

Reflexion Moment

Consider the `sb` (store byte) and `sh` (store halfword) instructions in the RV32I instruction set. If we were to support these instructions, would it be possible to complete all S-type instructions in a single cycle as we do now with `sw`? Why or why not?

3.9 BREAK

The **ebreak** instruction in our implementation is a repurposed version of the EBREAK (Environment BREAK) instruction from RV32I. It follows the I-type instruction format, as shown in Figure 8.

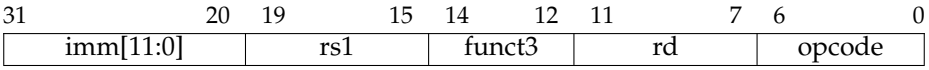

Figure 19: The I-type instruction format used for EBREAK in RV32I.

The EBREAK instruction is identified by the following specific field values in Figure 3.

Instruction	Immediate	funct3	opcode
ebreak	000000000001	000	1110011

Table 3: EBREAK instruction encoding values in RV32I.

In a standard RISC-V implementation, EBREAK is used to transfer control to a debugging environment. It's typically used for setting breakpoints in code during debugging. However, in our simplified CPU, we repurpose this instruction to stop the CPU execution entirely.

The BREAK instruction is identified by these specific values in its fields. When the CPU encounters this instruction, it enters the **BREAK** state, which serves as a termination point for the program, halting further execution.

This allows us to have a defined way to stop the CPU at a predetermined point in the code, which can be useful for testing and demonstration purposes. In our implementation, when the controller detects these specific field values during the DECODE stage, it transitions to the BREAK state and remains there, effectively stopping the program execution.

4 Exercise

- For this exercise, you will use modules you should have implemented during the previous labs. Make sure to review, understand, and import these modules before proceeding:
  - From the ALU lab:
    - `alu.v`: The Arithmetic Logic Unit
    - `add_sub.v`: The adder/subtractor module
    - `comparator.v`: The comparison module
    - `logic_unit.v`: The logical operations module
    - `shift_unit.v`: The shift operations module
    - `mux4x32.v`: The 4-to-1 multiplexer module
- Implement the simple **mux2x32** module in the file named `mux2x32.v`.
- Implement the **IR** (Instruction Register) in the file named `ir.v`.
- Implement a first version of the **PC** (Program Counter) in the file named `pc.v`. In this first version, the next address is always the current address incremented by 4. Remember to set the reset value to `32'h80000000`.

Instruction	Type	Opcode	State	Description
`and rd, rs1, rs2`	R-type	0110011	R_TYPE	`rd` ← `rs1` *AND* `rs2`
`srl rd, rs1, rs2`	R-type	0110011	R_TYPE	`rd` ← `rs1` ≫ `rs2`[4:0]
`addi rd, rs1, imm`	I-type	0010011	I_TYPE	`rd` ← `rs1` + `imm`
`lui rd, imm`	U-type	0110111	U_TYPE	`rd` ← `imm` ≪ 12
`lw rd, imm(rs1)`	I-type	0000011	LOAD	`rd` ← Mem[`rs1` + `imm`]
`sw rs2, imm(rs1)`	S-type	0100011	S_TYPE	Mem[`rs1` + `imm`] ← `rs2`
`ebreak`	I-type	1110011	BREAK	Stops the program execution

Table 4: Initial instructions for the RV32I CPU.

- Implement a first version of the **Controller** in the file named `controller.v`. In this first version, it should be able to decode the instructions from Table 4.
- Implement the described state machine, which controls all the control signals except **alu_op**. The **alu_op** signal is independent of the current state (i.e., it should be stateless) and should be generated in a separate process that depends on the instruction fields. See Section 5.4 for details.
- To test the current (incomplete) **Controller**, you can create an early version of the testbench that would test it for the couple instructions from Table 4.

5 Extending the multicycle CPU with flow control

In this section, you will add flow control to the CPU. This enables the CPU to do conditional jumps in the code using the *branch* instructions, and to call procedures using the `call` and `ret` instructions. To implement these instructions, you will add five new *Execute* states (i.e., the states coming from **DECODE** and going to **FETCH1**) to the state machine. These states are described in the following subsections. Do not forget to update the state transitions in your FSM process!

5.1 B_TYPE

The **B_TYPE** state executes conditional branch instructions, which are B-type instructions in RV32I. Figure 20 shows the general branch instruction format in detail.

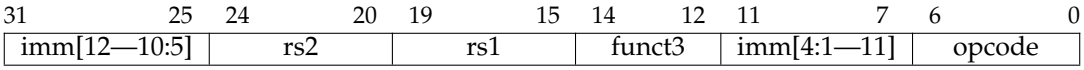

Figure 20: The B-type instruction format in RV32I.

All B-type instructions share the same opcode:

Instruction Type	Opcode
B_TYPE	1100011

Table 5: Opcode for B-type instructions in RV32I.

The immediate value for B-type instructions is formed in a complex manner to allow for a larger branch offset while maintaining the regular instruction length. The 12-bit immediate is constructed as follows:

- **imm[12]** is taken from instruction[31]
- **imm[11]** is taken from instruction[7]
- **imm[10:5]** are taken from instruction[30:25]
- **imm[4:1]** are taken from instruction[11:8]
- **imm[0]** is always 0

This immediate is then sign-extended to 32 bits, effectively creating a 13-bit signed offset.

Table 6 describes the different branch instructions in RV32I.

Instruction	funct3	Branches if:
`beq rs1, rs2, offset`	000	rs1 = rs2
`bne rs1, rs2, offset`	001	rs1 $\neq$ rs2
`blt rs1, rs2, offset`	100	rs1 $<$ rs2 (signed)
`bge rs1, rs2, offset`	101	rs1 $\geq$ rs2 (signed)
`bltu rs1, rs2, offset`	110	rs1 $<$ rs2 (unsigned)
`bgeu rs1, rs2, offset`	111	rs1 $\geq$ rs2 (unsigned)

Table 6: Branch instructions in RV32I.

During the **B_TYPE** state, the **ALU** compares the values of the registers **rs1** and **rs2**. If the condition is met, the **PC** must be updated with $PC \leftarrow PC + imm$. Remember that the **PC** has already been incremented by 4 during the **FETCH2** state, so this needs to be accounted for in the branch offset calculation.

Figure 21 shows the components used for the **B_TYPE** state.

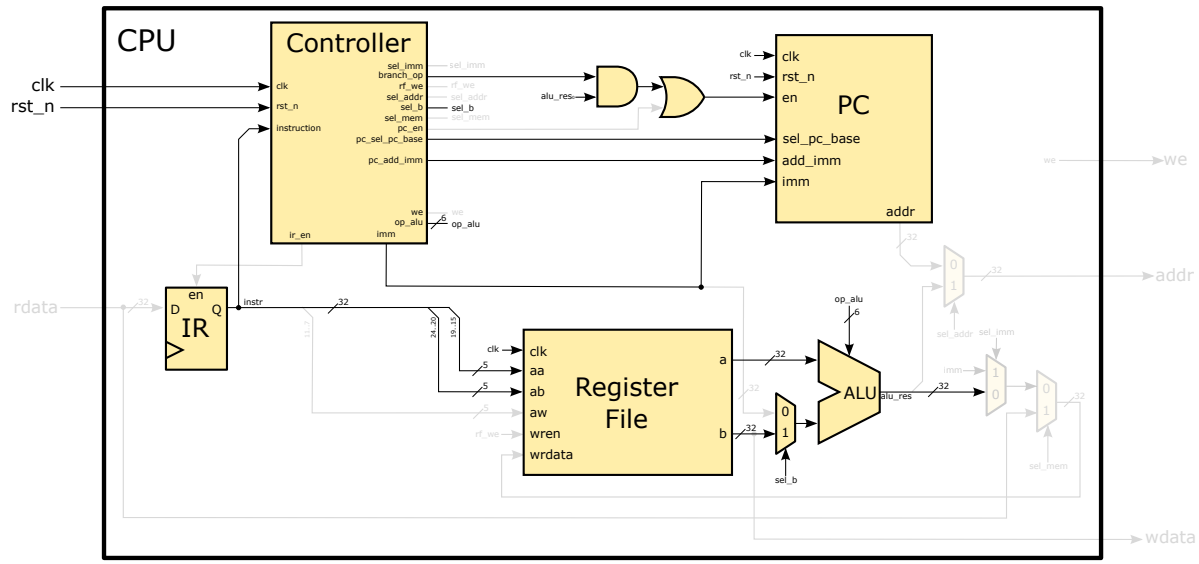

Figure 21: Components used for the **B_TYPE** state.

The **branch_op** signal and the least significant bit of the **ALU** result are used to determine if the branch should be taken. The **pc_add_imm** signal instructs the **PC** to add the sign-extended immediate value instead of 4 for the next instruction address. The **pc_sel_pc_base** signal in the controller is activated to

indicate that the current instruction address should be used as the base address for the branch calculation, rather than the next sequential instruction address. This ensures that the branch offset is applied to the address of the current instruction, allowing for correct relative branching.

5.2 J_TYPE

The **J_TYPE** state executes the `jal` (Jump and Link) instruction, which is a J-type instruction in RV32I. Figure 22 shows the `jal` instruction format in detail.

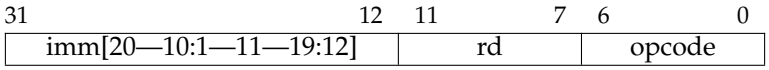

Figure 22: The J-type instruction format in RV32I.

The immediate value for J-type instructions is formed in a complex manner to allow for a larger jump offset. The 20-bit immediate is constructed as follows:

- **imm[20]** is taken from instruction[31]
- **imm[10:1]** are taken from instruction[30:21]
- **imm[11]** is taken from instruction[20]
- **imm[19:12]** are taken from instruction[19:12]
- **imm[0]** is always 0

This immediate is then sign-extended to 32 bits, effectively creating a 21-bit signed offset.

Table 7 describes the `jal` instruction.

Instruction	opcode	Description
`jal rd, offset`	1101111	Jump to PC + offset, save PC + 4 to rd

Table 7: The `jal` instruction.

During the **J_TYPE** state, the current **PC** value plus 4 is saved in the destination register (`rd`). The next address of **PC** is calculated by adding the sign-extended immediate to the current PC.

Figure 23 shows the components used for the **J_TYPE** state.

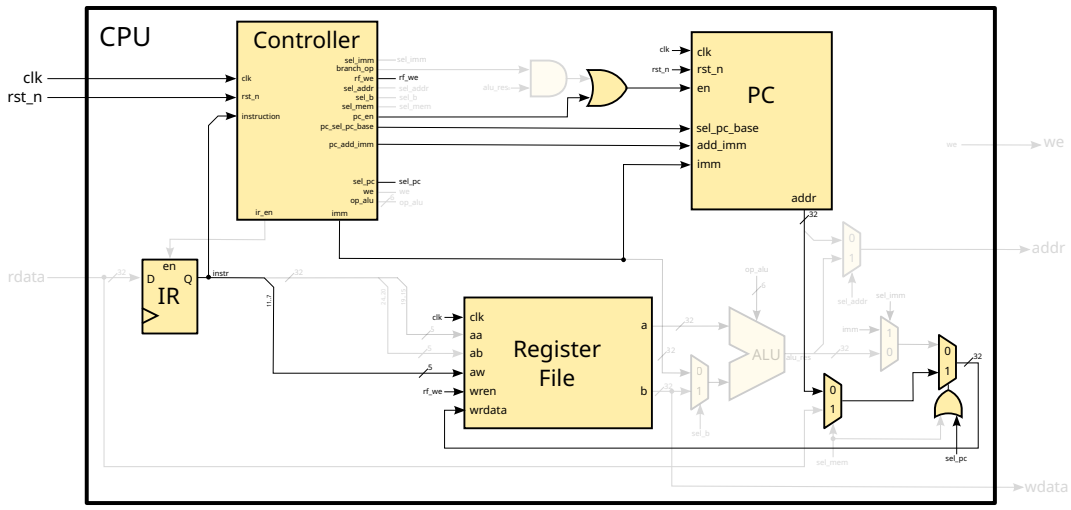

Figure 23: Components used for the **J_TYPE** state.

The **pc_add_imm** signal instructs the **PC** to add the sign-extended immediate value to the current PC for the next instruction address. The **sel_pc** signal selects PC + 4 as the value to be written to the destination register. The **rf_we** signal enables writing to the register file, storing the return address (PC + 4) in the destination register specified by `rd`. The **pc_sel_pc_base** signal in the controller is activated to use the current PC as the base address for the jump calculation. This ensures that the jump offset is applied to the address of the current instruction, allowing for correct relative jumping and maintaining the PC-relative addressing mode of the `jal` instruction.

5.3 JALR

The **JALR** state executes the Jump and Link Register instruction, which is an I-type instruction in RV32I. Figure 24 shows the JALR instruction format in detail.

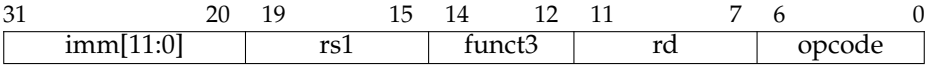

Figure 24: The JALR instruction format in RV32I.

Table 8 describes the JALR instruction.

Instruction	opcode	Description
`jalr rd, rs1, imm`	1100111	Jump to rs1 + imm, save PC + 4 to rd

Table 8: The JALR instruction.

During the **JALR** state, the following operations occur:

- The address of the next instruction (PC + 4) is saved in the destination register (`rd`).
- The next PC value is calculated by adding the sign-extended 12-bit immediate to the value in `rs1`.

- The two least significant bits of the calculated address are set to zero to ensure word alignment, as the PC should always contain a word-aligned address.

Note that, unlike the JAL instruction, JALR calculates the jump target address based on a register value plus an immediate, allowing for more flexible jump targets. The word alignment requirement (setting the two LSBs to zero) ensures that the jump always targets a valid instruction address.

Figure 25 shows the components used for the **JALR** state.

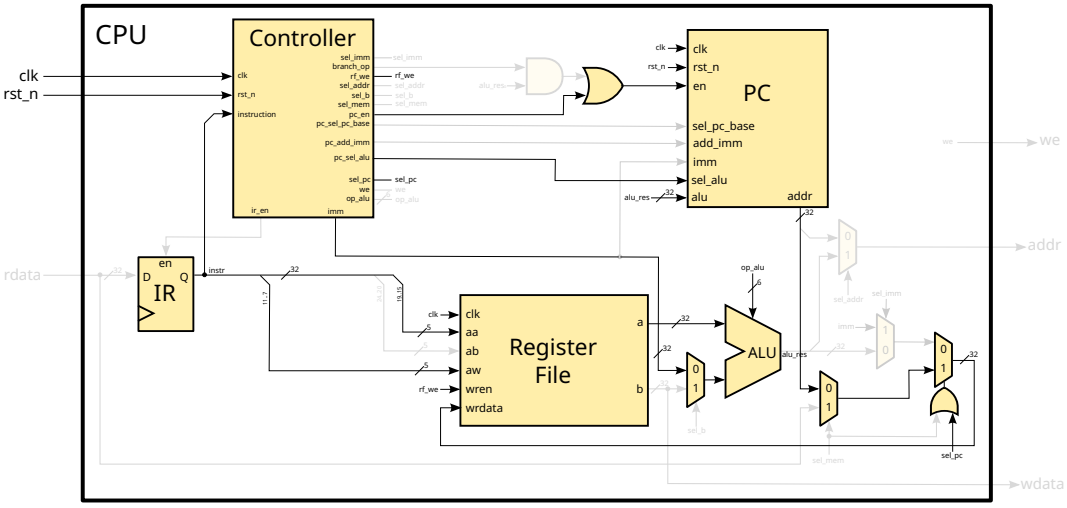

Figure 25: Components used for the **JALR** state.

The **pc_sel_alu** signal selects the ALU output (which computes rs1 + imm) as the next PC value. The **sel_pc** signal selects PC + 4 as the value to be written to the destination register. The **rf_we** signal enables writing to the register file, storing the return address (PC + 4) in the destination register specified by `rd`.

JALR can be used for various purposes, including:

- Implementing function returns (when rd = x0 and rs1 = x1, assuming x1 holds the return address)
- Implementing computed jumps
- Implementing more complex control flow by combining register and immediate values

5.4 Hint for the generation of the alu_op signal

The generation of the **alu_op** signal is crucial for the correct operation of the ALU. Here are some hints to help you implement this part:

- The **alu_op** signal is 6 bits wide, allowing for a variety of operations.
- For arithmetic and immediate arithmetic instructions (R-type and I-type), consider how the **funct3** field relates to the lower 3 bits of **alu_op**.
- Pay attention to special cases where the operation might differ between R-type and I-type instructions with the same **funct3** value (e.g., ADD vs SUB).
- For shift operations, consider how the **funct7** field might influence the operation (e.g., logical vs arithmetic shift).

- For comparison operations (SLT, SLTU), think about how these might be implemented using the ALU and how this affects the **alu_op** encoding.
- For branch instructions, consider how you can use the ALU to perform the necessary comparisons. How might the **funct3** field of branch instructions map to **alu_op**?
- Remember that some bits of **alu_op** might have different meanings for different operation types. The upper bits might control the operation category, while the lower bits specify the exact operation within that category.
- Implement the **alu_op** generation in a separate, combinational process that depends only on the instruction fields (**opcode**, **funct3**, **funct7**). This will make your design more modular and easier to extend.
- For instructions that don't use the ALU, you can set **alu_op** to a default value, as its result won't be used.

Remember to refer to the ALU operation codes in Table 1 when designing your **alu_op** generation logic. The key is to create a mapping between the instruction fields and the ALU operations that correctly implements the RV32I instruction set.

5.5 Exercise

- Modify the Verilog files of your **Controller** (`controller.v`) and the **PC** (`pc.v`) to add flow control to your CPU.
- Implement the immediate generation logic for B-type and J-type instructions in the **Controller**.
- Write your own testbenches to verify the correct operation of the new instructions supported by your CPU.

6 Completing the Multicycle CPU with the Remaining Instructions

In this final section, you will complete your CPU with the remaining operations. Most of the work is to generate, from the instruction, the correct value of the **alu_op** signal.

6.1 Immediate Operations

Table 9 lists the immediate arithmetic and logical instructions that can be handled by the **I_TYPE** state.

Instruction	funct3	Description
`addi rd, rs1, imm`	000	rd ← rs1 + imm
`slti rd, rs1, imm`	010	rd ← (rs1 < imm)? 1 : 0 (signed)
`sltiu rd, rs1, imm`	011	rd ← (rs1 < imm)? 1 : 0 (unsigned)
`xori rd, rs1, imm`	100	rd ← rs1 ⊕ imm
`ori rd, rs1, imm`	110	rd ← rs1 ∨ imm
`andi rd, rs1, imm`	111	rd ← rs1 ∧ imm

Table 9: Immediate arithmetic and logical instructions in RV32I.

Table 10 lists the immediate shift instructions that are also handled by the **I_TYPE** state but require special attention to the upper bits of the immediate value.

Instruction	funct3	imm[11:5]	Description
slli rd, rs1, imm	001	0000000	rd $\leftarrow$ rs1 $\ll$ imm[4:0]
srli rd, rs1, imm	101	0000000	rd $\leftarrow$ rs1 $\gg_l$ imm[4:0]
srai rd, rs1, imm	101	0100000	rd $\leftarrow$ rs1 $\gg_a$ imm[4:0]

Table 10: Immediate shift instructions in RV32I.

Important

For shift instructions, the controller should preserve all 12 bits of the immediate value (bits 31 to 20 of the instruction). The actual 5-bit shift amount should be extracted in the shifter unit, not in the controller.
For example, if you have an **srai** instruction with a shift amount of 2, the immediate output from the controller will be 0x402. This is because one of the bits in the funct7 field (contained within the immediate value) is set to indicate that the shift is arithmetic.
The shifter unit will then use only the lower 5 bits (0x02 in this case) as the actual shift amount, while the upper bits will be unused by the shifter inside the ALU.

6.2 Register Operations

Table 11 lists all the register-register instructions that can be handled by the **R_TYPE** state.

Instruction	funct3	funct7	Description
add rd, rs1, rs2	000	0000000	rd $\leftarrow$ rs1 + rs2
sub rd, rs1, rs2	000	0100000	rd $\leftarrow$ rs1 $-$ rs2
sll rd, rs1, rs2	001	0000000	rd $\leftarrow$ rs1 $\ll$ rs2[4:0]
slt rd, rs1, rs2	010	0000000	rd $\leftarrow$ (rs1 $<$ rs2)? 1 : 0 (signed)
sltu rd, rs1, rs2	011	0000000	rd $\leftarrow$ (rs1 $<$ rs2)? 1 : 0 (unsigned)
xor rd, rs1, rs2	100	0000000	rd $\leftarrow$ rs1 $\oplus$ rs2
srl rd, rs1, rs2	101	0000000	rd $\leftarrow$ rs1 $\gg_l$ rs2[4:0]
sra rd, rs1, rs2	101	0100000	rd $\leftarrow$ rs1 $\gg_a$ rs2[4:0]
or rd, rs1, rs2	110	0000000	rd $\leftarrow$ rs1 $\vee$ rs2
and rd, rs1, rs2	111	0000000	rd $\leftarrow$ rs1 $\wedge$ rs2

Table 11: Register-register instructions in RV32I.

Note that in RV32I, there are no separate states needed for unsigned operations or for immediate shift operations, as these are handled within the I-type and R-type instruction formats respectively.

When implementing these instructions, pay careful attention to the **funct3** and **funct7** fields, as they determine the specific operation to be performed. The **alu_op** signal should be generated based on these fields along with the **opcode**.

6.3 Exercise

- Complete the **Controller** (`controller.v`) to implement the remaining instructions.

- Complete your own testbenches to verify the correct operation of the remaining instructions supported by your CPU.

7 Submission

Submit all Verilog files related to the exercises in sections 2.3 , 4, 5.5 and 6.3. (`register_file.v`, `ir.v`, `pc.v`, `controller.v` and `mux2x32.v`) and the required files from the previous labs (`add_sub.v`, `comparator.v`, `logic_unit.v`, `mux4x32.v` and `shift_unit.v`). The files from part 1 are needed to run the testbench of the CPU, so a fail in the testbench can be caused by a mistake in the files from part 1. The testbenches from part 1 will not be reruned for this submission and you'll only get feedback for the files from part 2.

Once you submit the files, you will receive a report describing the tests that were applied to your design and the results of those tests (success or failure). This is a preliminary submission so the result of the tests will not affect your grade and in case of failure, you will have some additional infos about what was the first test that failed of every testbench.