Multicycle Nios II Processor

Learning Goal: Simple multicycle processor architecture.

Requirements: Gecko4Education, Quartus II Web Edition and ModelSim-Altera.

1 Introduction

In this lab you will implement a multicycle Nios II 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 Nios II. You will also use some of the components you built in the previous sessions.

2 Multicycle CPU Description

The first implementation of the **CPU** only executes several ALU operations (e.g., **`addi`**, **`and`**) and the **`ldw`** and **`stw`** instructions. A **`break`** instruction is also used to stop the execution of the program. As shown in Figure 1, the **CPU** is connected into the same system you built for the Memories lab with an additional **Button** module, which interfaces the four buttons of the Gecko4Education. In particular,`SW1` to `SW4` of the Gecko board are mapped to `buttons_in[0]` to `buttons_in[3]` of the system, whereas `reset_n` is connected to `SW6`.

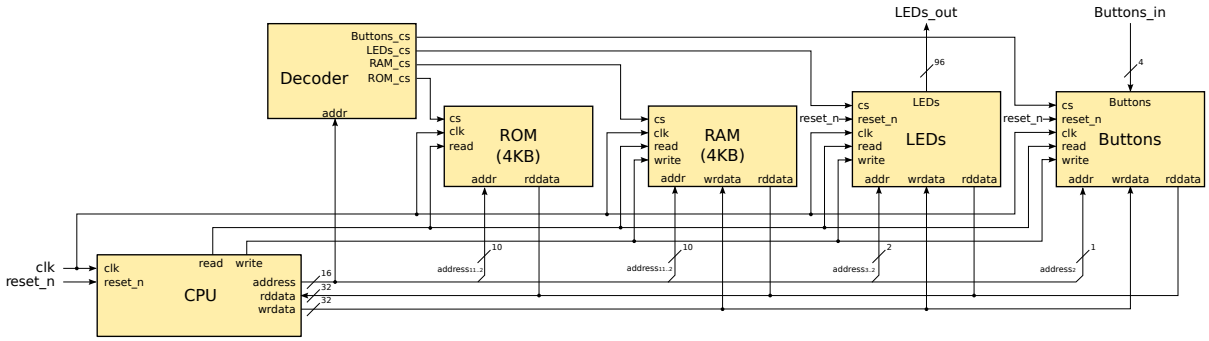

Figure 1: Connection of the multicycle Nios II processor to the other components of the system.

To execute an instruction, the multicycle **CPU** needs 4 to 5 cycles, depending on the instruction. Figure 2 shows the state machine of the **CPU**'s controller. It illustrates the different steps of the execution of an instruction.

During **FETCH1** and **FETCH2**, the **CPU** reads the next instruction to execute. During **DECODE**, the **CPU** identifies the instruction and determine the next state. During the next states, the instruction is executed. These last states are called *Execute* states.

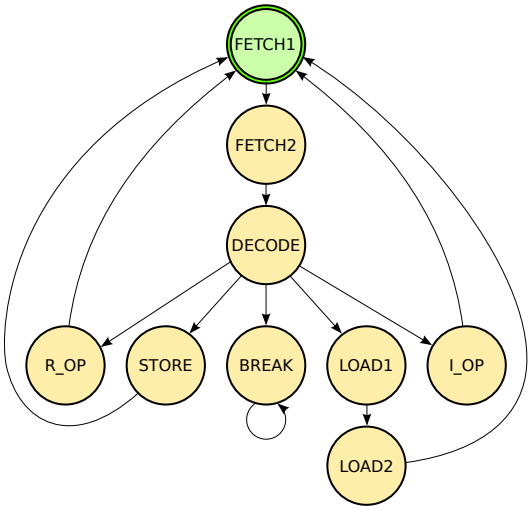

Figure 2: The state machine of the **CPU**'s controller.

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

2.1 FETCH1

During this first state of the execution, the address of the next instruction and the signal **read** are set to start a new read process. The instruction word is available during the next cycle. Figure 3 shows the components used for the **FETCH1** state.

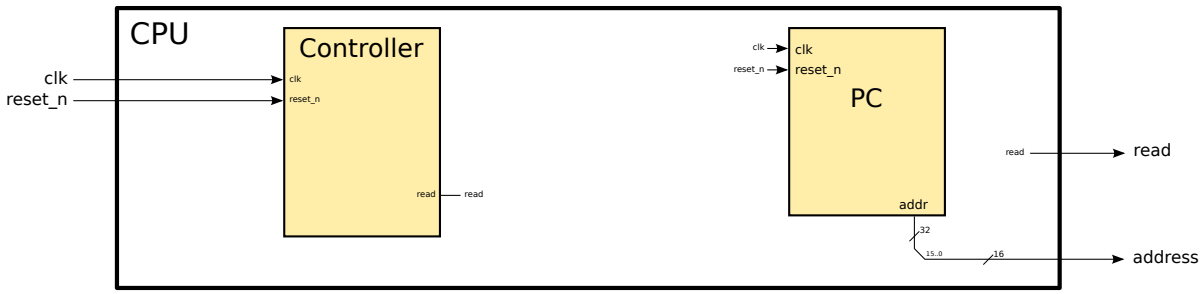

Figure 3: Components used for the **FETCH1** state.

The **Controller** controls the state machine. The input **reset_n**, asynchronous and active low, initializes the state machine to **FETCH1**. The **PC** holds the address of the next instruction. The address is stored in a 16-bit register. The address must always be valid, thus the two least significant bits should remains at '0'.

The first version of the **CPU** is purely sequential: the next address is the current address incremented by 4.

- The input **clk** is the clock signal.
- The output **addr** is the current 16-bit register value extended to 32 bits. The 16 most significants bits are set to 0.
- The input **reset_n** initializes the address register to 0.

- The input **en** (see FETCH2 figure) enables the **PC** to switch to the next address (i.e., **addr**+4 for the moment).

2.2 FETCH2

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

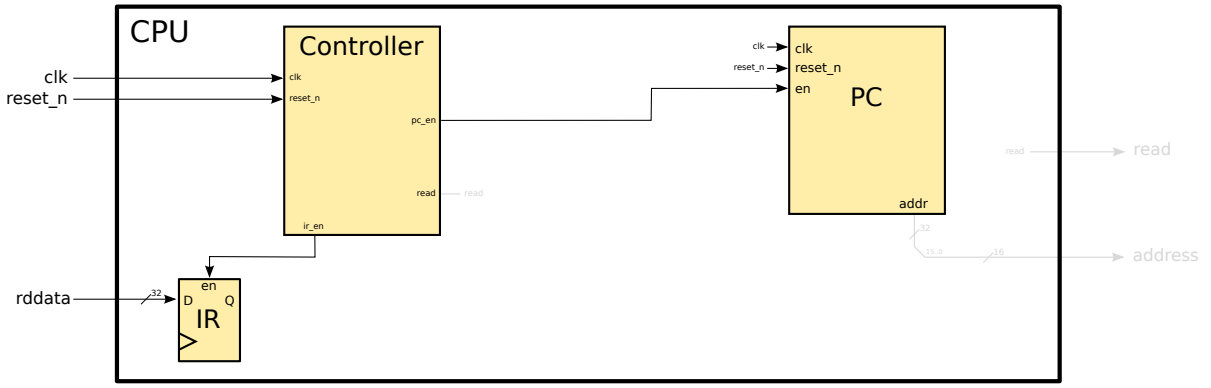

Figure 4: Components used for the **FETCH2** state.

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.

2.3 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 Nios II instructions are progressively described in the following subsections. Figure 5 shows the components used for this state.

2.4 I_OP

The **I_OP** 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 16-bit immediate value are **I-type** (Immediate type) instructions. Figure 6 shows the general **I-type** instruction format in details.

The fields **A** and **B** are register addresses. In most of the cases, **A** is a register operand and **B** is the destination register. The field **IMM16** is the 16-bit immediate value. The field **OP** is the opcode of the instruction.

During the state **I_OP**, the **op_alu** signal is set by the **Controller** to perform the required operation in the **ALU**. The result of the **ALU** is saved in the **Register File**. Figure 7 shows the components used for this state.

The **Register File** and the **ALU** are the same units that you have implemented during the previous labs. The **Extend** unit extends the width of the 16-bit field **IMM16** to 32 bits. The sign is extended or not depending on the signal **signed**.

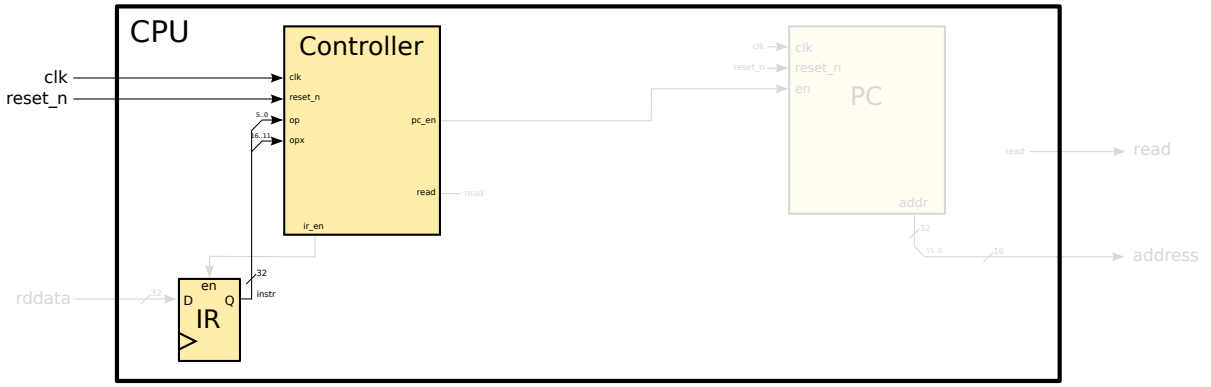

Figure 5: Components used for the **DECODE** state.

31 30 29 28 27	26 25 24 23 22	21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6	5 4 3 2 1 0
A	B	IMM16	OP

Figure 6: The general **I-type** instruction format.

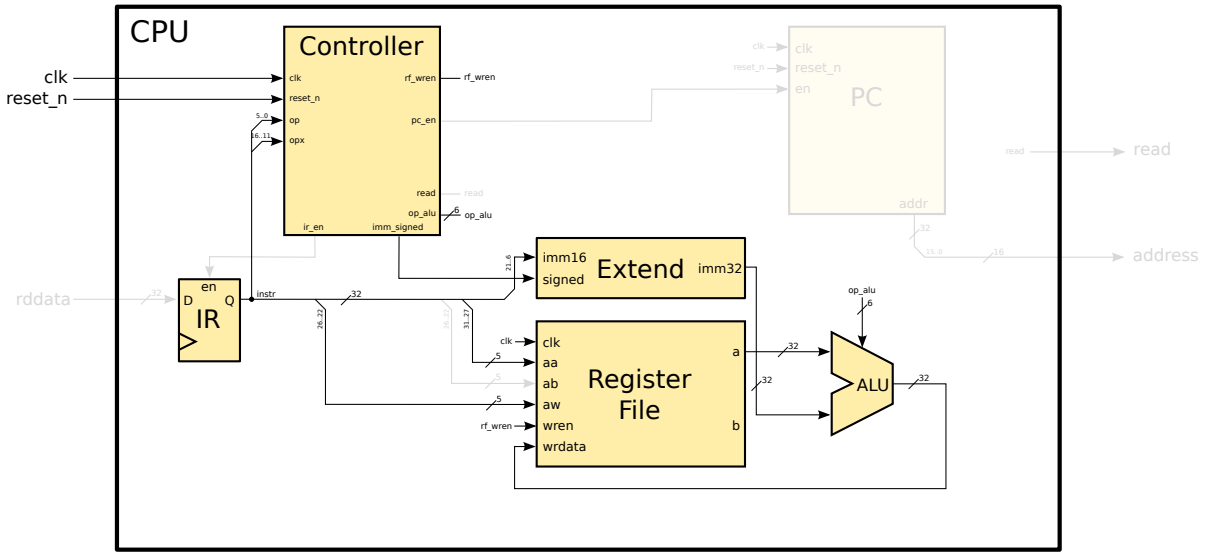

Figure 7: Components used for the **I_OP** *Execute* state.

The **Controller** selects the operation to execute in the **ALU** with the signal **op_alu**. The **op_alu** signal depends on the current instruction (e.g., an *addition* for **`addi`**, **`stw`** and **`ldw`**, a *logical AND* for **`and`**, or a *logical right shift* for **`srl`**).

The **ALU** opcode is summarized in Table 1.

Operation	Operation Type	Opcode
$A + B$	Add/Sub	$000\phi\phi\phi$
$A - B$		$001\phi\phi\phi$
$A \leq B$ (signed)	Comparison	011001
$A > B$ (signed)		011010
$A \neq B$		011011
$A = B$		011100
$A \leq B$ (unsigned)		011101
$A > B$ (unsigned)		011110
$A\ nor\ B$	Logical	$10\phi\phi00$
$A\ and\ B$		$10\phi\phi01$
$A\ or\ B$		$10\phi\phi10$
$A\ xnor\ B$		$10\phi\phi11$
$A\ rol\ B$	Shift/Rotate (Optional)	$11\phi000$
$A\ ror\ B$		$11\phi001$
$A\ sll\ B$		$11\phi010$
$A\ srl\ B$		$11\phi011$
$A\ sra\ B$		$11\phi111$

ϕ = *don't care*

Table 1: **ALU** opcode.

2.5 R_OP

The **R_OP** 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 8 shows the general **R-type** instruction format in details.

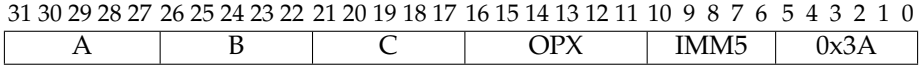

Figure 8: The general **R-type** instruction format.

The fields **A**, **B** and **C** are register addresses. In most of the cases, **A** and **B** are register operands, and **C** is the destination register. The field **IMM5** is a 5-bit immediate value that is used only by a few **R-type** instructions. The field **OP** is always set to `0x3A` and it identifies the **R-type** instructions. The field **OPX** is an extension of the field **OP** and is the actual opcode of the **R-type** instructions.

During the state **R_OP**, the signal **op_alu** is set by the **Controller** to perform the required operation in the **ALU**. Register **b** is selected as the second operand and the result of the **ALU** is saved in the **Register File**. Figure 9 shows the components used for the **R_OP** state.

The multiplexer controlled by the signal **sel_b** selects the second operand of the **ALU**: either register **b** (for R-type instructions) or the immediate value (for I-type instructions). The multiplexer controlled by the **sel_rC** signal selects the write address (**aw**) from either the **B** (for I-type instructions) or **C** (for R-type instructions) instruction field.

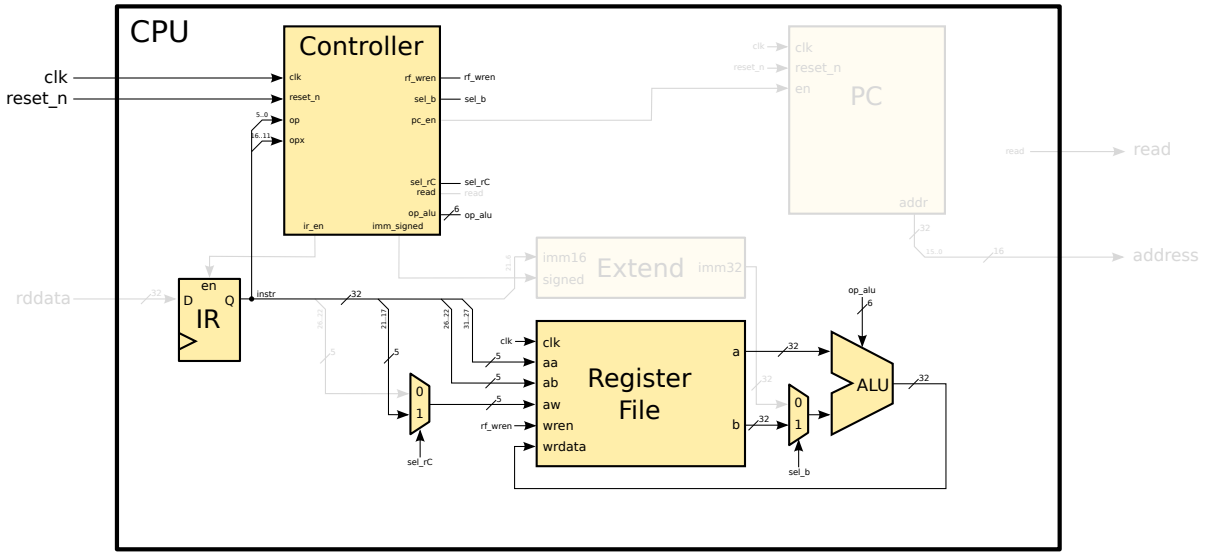

Figure 9: Components used for the **R_OP** *Execute* state.

2.6 LOAD

The **ldw** instruction is an **I-type** instruction with **OP**=0x17. Figure 10 shows the **LOAD** instruction format in details.

31 30 29 28 27	26 25 24 23 22	21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
A	B	IMM16	0x17

Figure 10: The **LOAD** instruction format.

Figure 11 shows the components used for the **LOAD1** state.

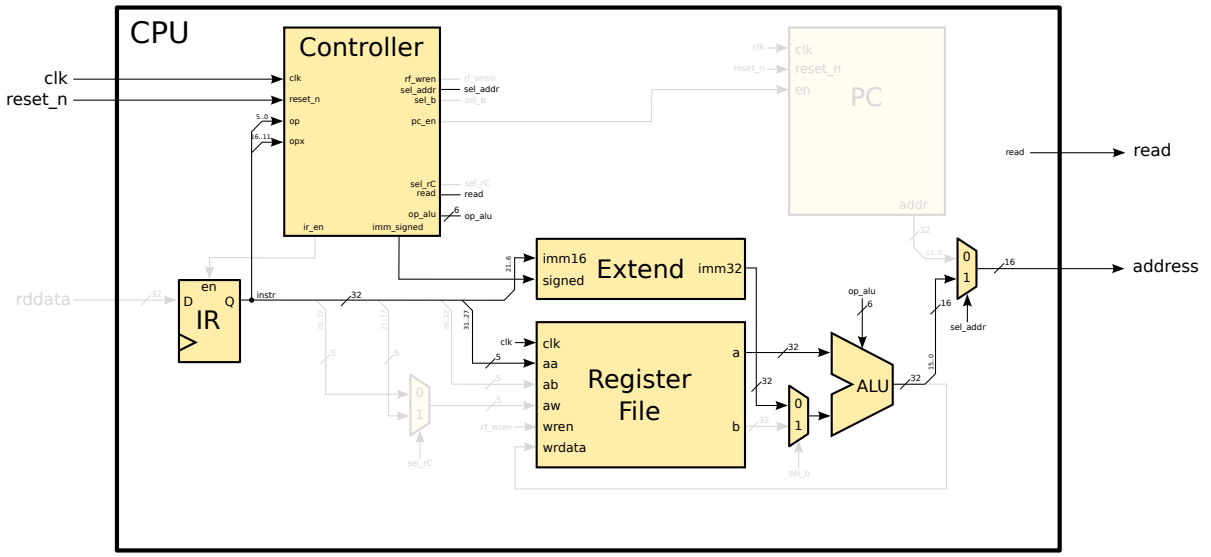

Figure 11: Components used for the **LOAD1** *Execute* state.

The load operation takes 1 more cycle than the other instructions. This is caused by the read process, which has a 1-cycle latency. During the state **LOAD1**, the address to read is computed by the ALU (adding the signed *immediate* value to **a**) and the signal **read** is set to start a read process. The read value will be available during **LOAD2**. The multiplexer controlled by the signal **sel_addr** selects the memory address from either the **PC** address or the result of the **ALU**.

During the state **LOAD2**, the memory data is written to the **Register File** at the address specified by **B**. The multiplexer controlled by the signal **sel_mem** selects the data to write to the **Register File** from either the result of the **ALU** or the **rddata** input. Figure 12 shows the components used for the **LOAD2** state.

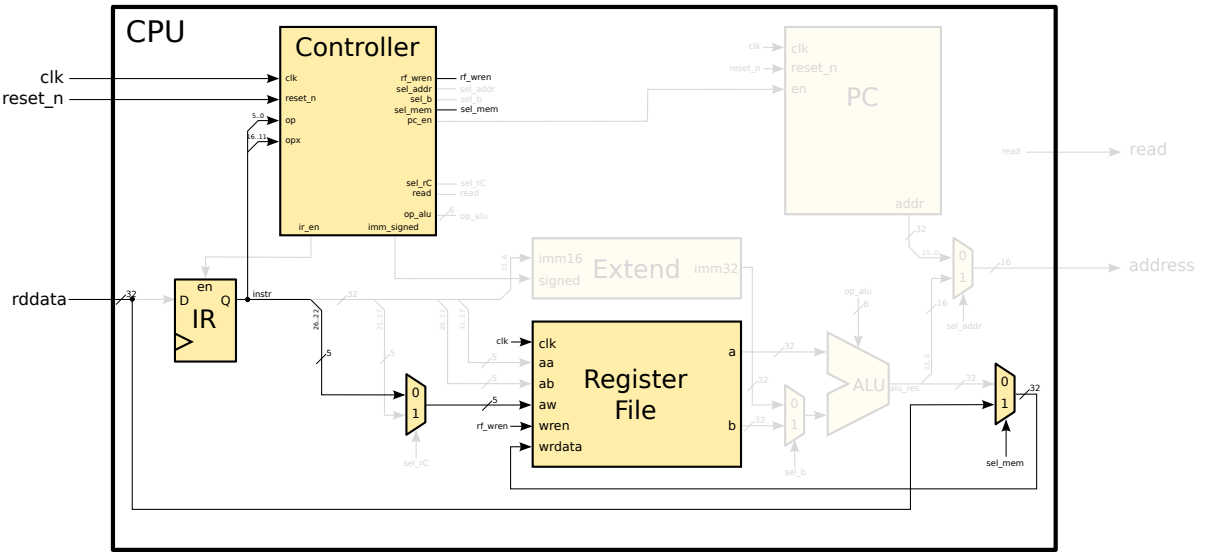

Figure 12: Components used for the **LOAD2** *Execute* state.

2.7 STORE

The **stw** instruction is a **I-type** instruction with **OP**=0x15. Figure 13 shows the **STORE** instruction format in details.

31 30 29 28 27	26 25 24 23 22	21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
A	B	IMM16	0x15

Figure 13: The **STORE** instruction format.

During the state **STORE**, the **ALU** computes the memory address as for a **ldw** instruction, and the **Controller** activates the **write** output signal to start a write process. The data to write is held in the register **b**. Figure 14 shows the components used for the **STORE** state.

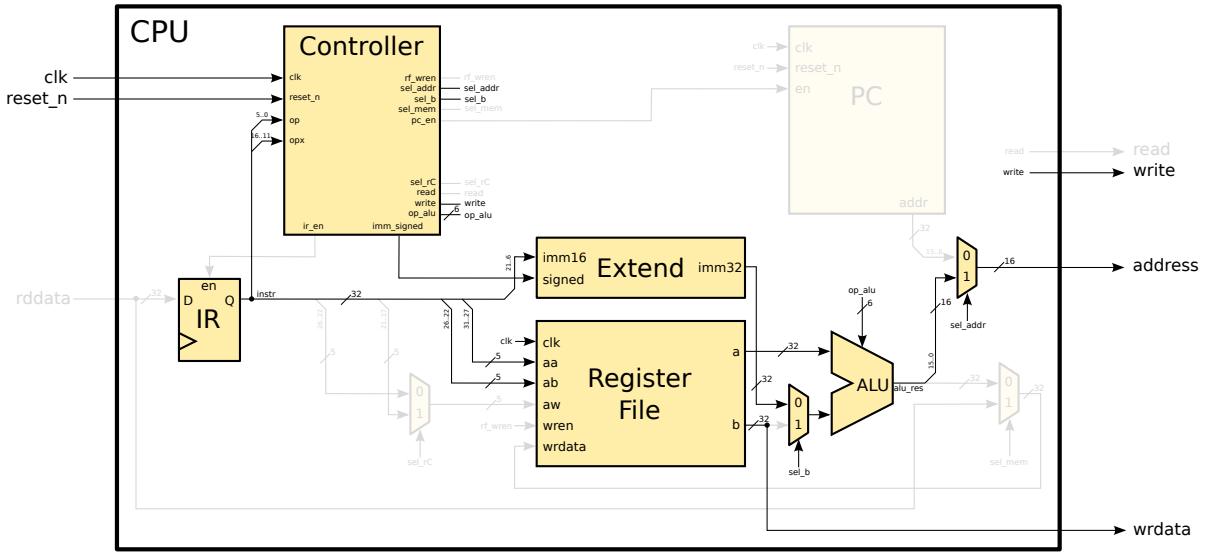

Figure 14: Components used for the **STORE** *Execute* state.

2.8 BREAK

The **`break`** instruction is a **R-type** instruction with **OPX**=`0x34`. Figure 15 shows the **BREAK** instruction format in details.

31 30 29 28 27	26 25 24 23 22	21 20 19 18 17	16 15 14 13 12 11	10 9 8 7 6	5 4 3 2 1 0
0x00	0x00	0x00	0x34	0x00	0x3A

Figure 15: The **BREAK** instruction format.

This instruction stops the CPU execution (note that this is not the official purpose of this instruction). The state **BREAK** is simply a dead end.

3 Exercise

- Download the project template.
- Open the Quartus project, and open the `GECKO.bdf` file. Notice that the CPU is connected into the same system that you used during the **Memories** lab.
- For this exercise, you will use some units you should have implemented during the previous labs. **We recommend you to make sure that they pass all the tests on submissions 01 and 02 of lapsubmit before moving on to this project, as otherwise it will be difficult to debug the multicycle processor.** Copy the following files into the **vhdl** folder of this project and make sure the filenames and entities match.
  - For the **ALU**, copy all the VHDL files (`add_sub.vhd`, `ALU.vhd`, `comparator.vhd`, `logic_unit.vhd`, `multiplexer.vhd`, `shift_unit.vhd`) from the **vhdl** folder of your **ALU** project.
  - For the **Register File**, copy the `register_file.vhd` from the **vhdl** folder of your **Memories** project.

  - For the memory system, copy the `ROM.vhd`, `RAM.vhd`, `ROM_Block.vhd` and `decoder.vhd` files from the **vhdl** folder of your **Memories** project.
  - Modify the Decoder to add a new `cs_buttons` output that is activated when we access the **Buttons** module (addresses `0x2030` to `0x2034`).

- The general architecture of the **CPU** is already given in the `CPU.bdf` file. This file contains the architecture for the complete version of the CPU. You will find extra control signals in addition to the ones discussed until now. Ignore them for the moment (set them to 0 while you implement the **Controller**).
- Implement the multiplexers: `mux2x5`, `mux2x16` and `mux2x32` in the corresponding VHDL files from the project template. These multiplexers only differ on their bitwidth.
- Implement the **Extend** unit in the file named `extend.vhd`.
- Implement the **IR** in the file named `IR.vhd`.
- Implement a first version of the **PC** in the file named `PC.vhd`. In this first version, the next address is always the current address incremented by 4.
- Implement a first version of the **Controller** in the file named `controller.vhd`. In this first version, it should be able to decode the instructions from Table 2.

Instruction	State	Type	OP	OPX	Description
and rC, rA, rB	R_OP	R-type	0x3A	0x0E	rC ← rA *AND* rB
srl rC, rA, rB	R_OP	R-type	0x3A	0x1B	rC ← (*unsigned*)rA ≫ $rB_{4..0}$
addi rB, rA, imm	I_OP	I-type	0x04	-	rB ← rA + (*signed*)imm
ldw rB, imm(rA)	LOAD	I-type	0x17	-	rB ← Mem[rA + (*signed*)imm]
stw rB, imm(rA)	STORE	I-type	0x15	-	Mem[rA + (*signed*)imm] ← rB
break	BREAK	R-type	0x3A	0x34	Stops the program execution

Table 2: Initial instructions for the CPU.

Implement the described state machine, which controls all the control signals except **op_alu**. The **op_alu** signal is independent of the current state (i.e. it should be stateless) and should be generated in a separated process that depends only on **OP** and **OPX**. This simplifies the introduction of additional operations.

- Compile the Quartus project and correct the syntax errors.
- Open the Modelsim project `multicycle_niosII.mpf` in the **modelsim** subfolder and compile it. Ignore the warnings that you may get on the `tb_*.vhd` and `check_functions.vhd` files.
- To test the current (incomplete) **Controller** use the `test_Controller0.do` file from the **modelsim** folder by typing `do test_Controller0.do` in the Transcript window on the bottom of the Modelsim interface. A `.do` file is a macro of Modelsim commands; you are encouraged to check `test_Controller0.do` yourself to learn how compilation and simulation can be started from command line and how the inputs to be applied to a circuit can be imported from a file. Once the controller is fully implemented, you can also use the other provided testbenches to verify other components. For that, open the **modelsim** folder and execute the corresponding do files (for example, to test the Extend unit, execute the `test_Extend.do` file).

To test the CPU, you will write a short machine language program.

- Download the Nios2Sim simulator from the web page of the course.

- The simulator is a Java executable (`.jar`). If you want to execute it on your own machine, make sure you have installed a Java Runtime Environment (JRE). For more details go to the Java web site (`http://www.java.com`).
- Double click on the `nios2sim.jar` file to run it.
- Copy the following Nios II assembly code to the Nios2Sim text editor and save it to a `program.asm` file.

```
addi    t0, zero, 0x55AA
stw     t0, 0x2000(zero)
break
```

- Select Nios II >Assemble to assemble the code. This verifies that the syntax is correct.
- Select File >Export to Hex File to generate the initialization file of your **ROM**. Save it in your `ROM.hex` that is in the **quartus** folder.
- Compile your Quartus project to update the **ROM** content. *If you want to update the ROM content without recompiling everything, select* Processing >Update Memory Initialization File.
- Program the FPGA and verify that the behavior is consistent with the one of the Nios2sim simulator. If this is not the case, first make sure to test each module separately with the respective `test_[unit].do` macro. If all tests passed but the system still does not work correctly on the board, check the **ModelSim debugging video tutorial** on the course Moodle to learn how to debug the project as a whole by using the `tb_GECKO.vhd` file from the **testbench** folder.
- Modify the assembly program to test all implemented instructions. For example, you can write a program that displays the result of an addition onto the LEDs.

4 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 create three new *Execute* states (i.e., the states coming from **DECODE** and going to **FETCH1**) to the state machine. These three states are described in the following subsections.

4.1 BRANCH

The **BRANCH** state executes *branch* instructions, which are **I-Type** instructions. Figure 16 shows the general branch instruction format in details.

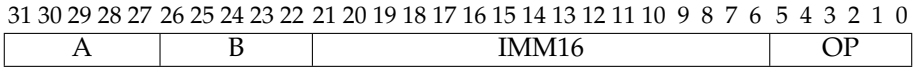

Figure 16: The general branch instruction format.

Table 3 describes the different branch instructions.

During the **BRANCH** state, the **ALU** compares the values of the registers **a** and **b**. If the comparison is verified, the **PC** must take the value $PC \leftarrow PC+4+IMM16$ (PC is the address of the current instruction). However, remember that the **PC** has already been incremented by 4 during the state **FETCH2**. Therefore, we only need to add the signed immediate value to the current address stored in the **PC**. Figure 17 shows the components used for the **BRANCH** state.

Instruction	Type	OP	Jumps to `label` if:
br `label`	I-type	0x06	*no condition*
ble `rA, rB, label`	I-type	0x0E	`rA` $\leq$ `rB`
bgt `rA, rB, label`	I-type	0x16	`rA` $>$ `rB`
bne `rA, rB, label`	I-type	0x1E	`rA` $\neq$ `rB`
beq `rA, rB, label`	I-type	0x26	`rA` $=$ `rB`
bleu `rA, rB, label`	I-type	0x2E	(*unsigned*)`rA` $\leq$ (*unsigned*)`rB`
bgtu `rA, rB, label`	I-type	0x36	(*unsigned*)`rA` $>$ (*unsigned*)`rB`

Table 3: Branch instructions.

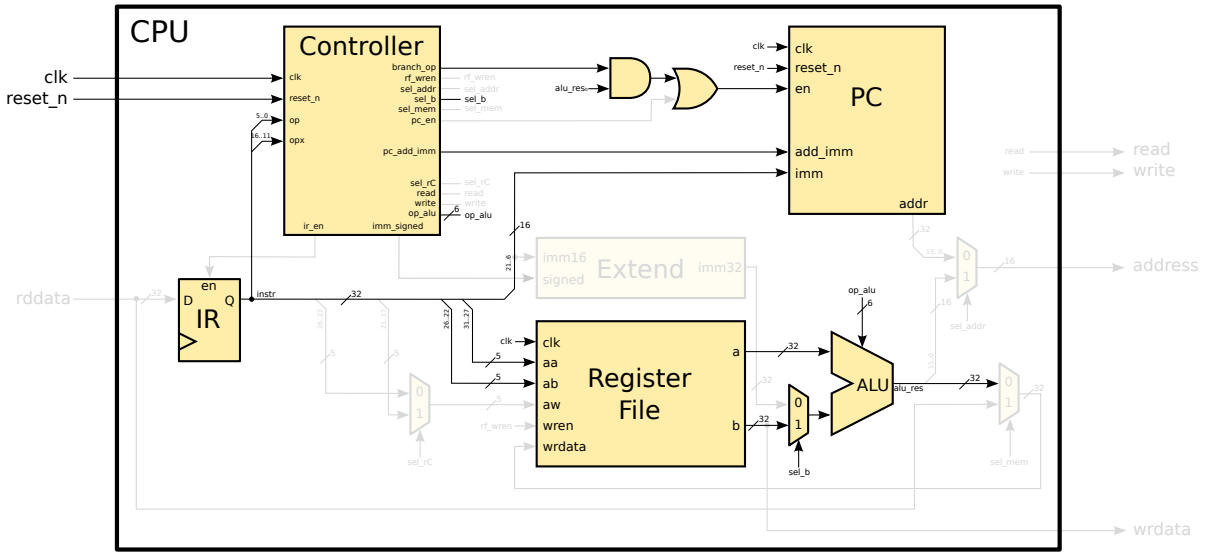

Figure 17: Components used for the **BRANCH** *Execute* state.

Two logic gates are added to enable the **PC** when the **branch_op** signal and the least significant bit of the result of the **ALU** are active. By default, the value that is added to the **PC** is 4. The **pc_add_imm** signal tells to the **PC** to selects the immediate value instead of 4 for the addition.

Following is represented a small example of a loop with a branch instruction.

```
      addi  r2, r2, 32
loop: addi  r2, r2, -1
      ...
      bgt   r2, r0, loop
```

4.2 CALL

The **CALL** state executes the **call** instruction, which is an **I-type** instruction with **OP**=`0x00`. Figure 18 shows the **call** instruction format in details.

31 30 29 28 27	26 25 24 23 22	21 20 19 18 17 16 15 14 13 12 11 10	9 8 7 6 5 4 3 2 1 0
0x00	0x00	IMM16	0x00

Figure 18: The **call** instruction format.

Table 4 describes the **call** instruction.

Instruction	Type	OP	Description
call label	I-type	0x00	Call procedure label

Table 4: The **call** instruction.

During the **CALL** state, the current **PC** value is saved in the *return address* register (ra). The next address of **PC** is the value of the IMM16 field shifted to the left by 2. Figure 19 shows the components used for the **CALL** state.

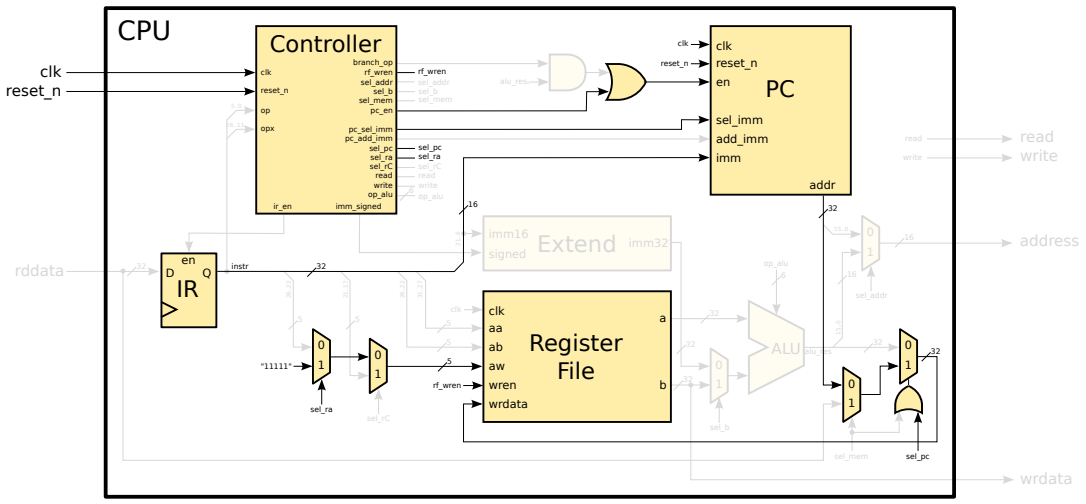

Figure 19: Components used for the **CALL** *Execute* state.

The **pc_sel_imm** signal selects the immediate field as the next value of the **PC**. In the **call** instruction, the embedded address is byte aligned. Since the **PC** is word aligned, the immediate value must be shifted to the left by 2.

The multiplexer controlled by the **sel_ra** signal selects the write address register from either the **B** instruction field or the address of the ra register located on address 31 in the **Register File**.

4.2.1 CALLR

During a **callr** instruction, the current **PC** address is saved in the ra register, and the next value of the **PC** takes its new value from the register **a**. Figure 20 shows the **callr** instruction format in details.

31 30 29 28 27	26 25 24 23 22	21 20 19 18 17	16 15 14 13 12	11 10 9 8 7 6	5 4 3 2 1 0
A	0x00	0x1F	0x1D	0x00	0x3A

Figure 20: The **callr** instruction format.

The field **C** of the **callr** instruction is implicitly set to ra locate on address 31 in the **Register File**. Table 5 describes the **callr** instruction.

Instruction	Type	OPX	Description
callr rA	R-type	0x1D	ra ← PC; PC ← rA

Table 5: The **callr** instruction.

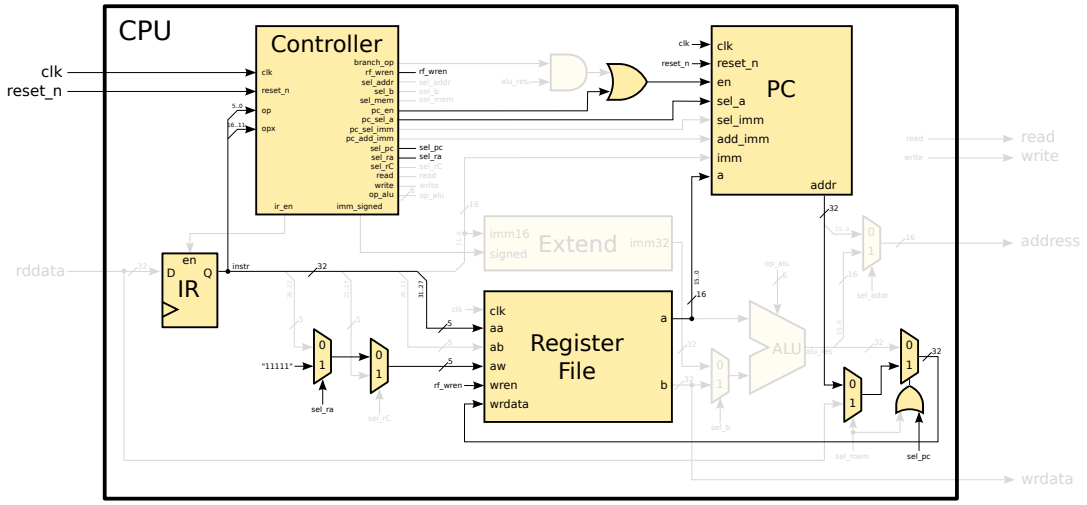

Figure 21: Components used for the **CALLR** *Execute* state.

4.3 JMP

The **JMP** state executes the **jmp** and **ret** instructions, which are **R-type** instructions with **OPX**=`0x0D` and **OPX**=`0x05`, respectively. Figure 22 shows the **jmp** and **ret** instruction formats in details.

jmp instruction format:

31 30 29 28 27	26 25 24 23 22	21 20 19 18 17	16 15 14 13 12 11	10 9 8 7 6	5 4 3 2 1 0
A	0x00	0x00	0x0D	0x00	0x3A

ret instruction format:

31 30 29 28 27	26 25 24 23 22	21 20 19 18 17	16 15 14 13 12 11	10 9 8 7 6	5 4 3 2 1 0
0x1F	0x00	0x00	0x05	0x00	0x3A

Figure 22: The **jmp** and **ret** instruction formats.

Note that for the instruction **ret**, the field **A** is implicitly set to the register `ra` located on address 31 in the **Register File**. Thus, **ret** is equivalent to **jmp** `ra`.

Table 6 describes the **ret** and **jmp** instructions.

Instruction	Type	OPX	Description
ret	R-type	0x05	PC ← `ra`
jmp `rA`	R-type	0x0D	PC ← `rA`

Table 6: The **ret** and **jmp** instructions.

During the **JMP** state, the **PC** address takes the value from the register **a**. The **pc_sel_a** signal selects the value coming from the register **a** as the next value of the **PC**. Figure 23 shows the components used for the **JMP** state.

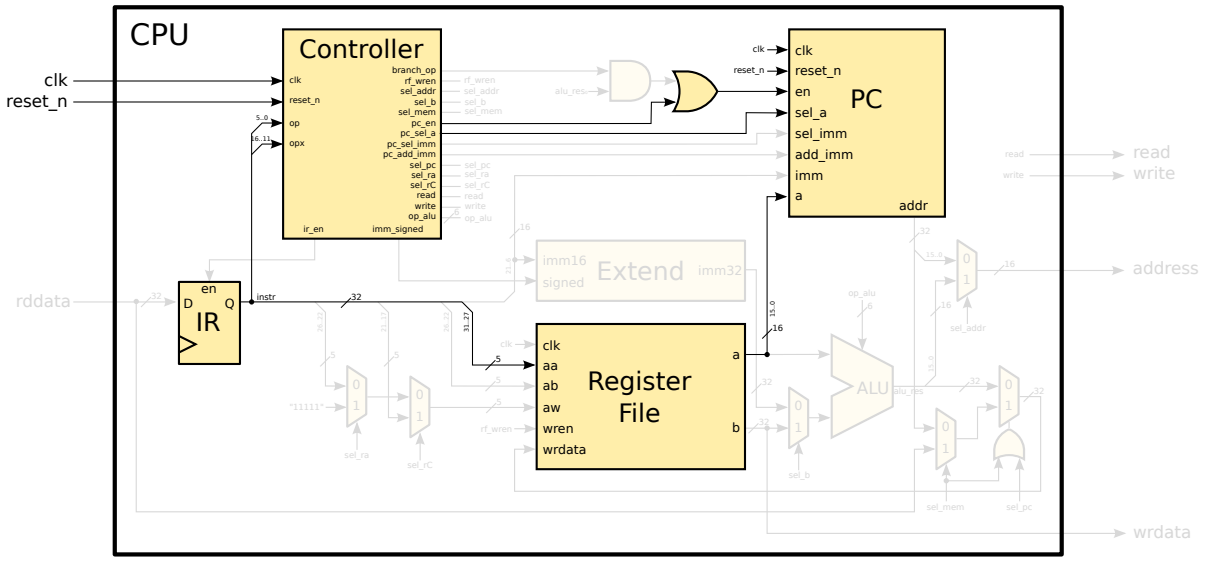

Figure 23: Components used for the **JMP** *Execute* state.

4.3.1 JMPI

During a **`jmpi`** instruction, the **PC** takes the value of the immediate field shifted to the left by 2, as for the **`call`** instruction. Table 7 describes the **`jmpi`** instruction.

Instruction	Type	OP	Description
`jmpi` `label`	I-type	0x01	Jumps to `label`

Table 7: The **`jmpi`** instruction.

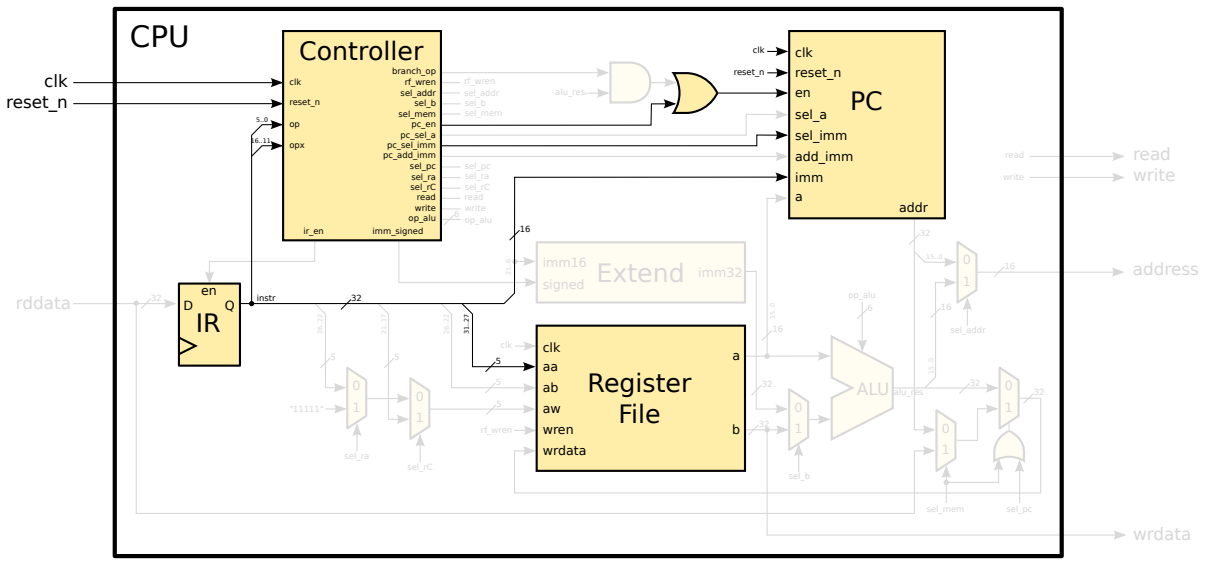

Figure 24: Components used for the **JMPI** *Execute* state.

4.4 Exercise

- In Quartus, modify the **Controller** and the **PC** to add flow control to your CPU.
- Compile and correct any errors that you find.
- Modify your assembly program `program.asm` to test the new instructions of the CPU.
- Generate the new `ROM.hex` file.
- Compile your Quartus project and program the FPGA. Use ModelSim and the provided testbenches for debugging.

5 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 **op_alu** signal. Section 5.3 give you some hints to generate it efficiently.

5.1 Immediate Operations

Table 8 lists the addition instruction that can be handled by the **I_OP** state.

Instruction	Type	OP	Description
addi rB, rA, imm	I-type	0x04	rB ← rA + (*signed*)imm

Table 8: An **I-type** addition instruction handled by the **I_OP** state.

The immediate operations listed in Table 9 require their immediate value to be considered as an *unsigned* number. Thus, it is recommended to create a new *Execute* state for these instructions.

Instruction	Type	OP	Description
andi rB, rA, imm	I-type	0x0C	rB ← rA *and* (*unsigned*)imm
ori rB, rA, imm	I-type	0x14	rB ← rA *or* (*unsigned*)imm
xnori rB, rA, imm	I-type	0x1C	rB ← rA *xnor* (*unsigned*)imm

Table 9: Additional **I-type** instructions handled by a new *Execute* state.

Table 10 lists the *comparison* instructions that can be handled by the **I_OP** state:

Instruction	Type	OP	Description
cmplei rB, rA, imm	I-type	0x08	rB ← (rA ≤ (*signed*)imm)? 1 : 0
cmpgti rB, rA, imm	I-type	0x10	rB ← (rA > (*signed*)imm)? 1 : 0
cmpnei rB, rA, imm	I-type	0x18	rB ← (rA ≠ (*signed*)imm)? 1 : 0
cmpeqi rB, rA, imm	I-type	0x20	rB ← (rA = (*signed*)imm)? 1 : 0

Table 10: Comparison instruction handled by the **I_OP** state.

The immediate comparison operations listed in Table 11 require their immediate value to be considered as an *unsigned* number. Thus, they can be handled by the new *Execute* state.

Instruction	Type	OP	Description
cmpleui rB, rA, imm	I-type	0x28	rB ← (*unsigned*)rA ≤ (*unsigned*)imm
cmpgtui rB, rA, imm	I-type	0x30	rB ← (*unsigned*)rA > (*unsigned*)imm

Table 11: Comparison instruction handled by the new *Execute* state.

Instruction	Type	OPX	Description
add rC, rA, rB	R-type	0x31	rC ← rA + rB
sub rC, rA, rB	R-type	0x39	rC ← rA − rB
cmple rC, rA, rB	R-type	0x08	rC ← (rA ≤ rB)? 1 : 0
cmpgt rC, rA, rB	R-type	0x10	rC ← (rA > rB)? 1 : 0
nor rC, rA, rB	R-type	0x06	rC ← rA *nor* rB
and rC, rA, rB	R-type	0x0E	rC ← rA *and* rB
or rC, rA, rB	R-type	0x16	rC ← rA *or* rB
xnor rC, rA, rB	R-type	0x1E	rC ← rA *xnor* rB
sll rC, rA, rB	R-type	0x13	rC ← rA ≪ $rB_{4..0}$
srl rC, rA, rB	R-type	0x1B	rC ← (*unsigned*)rA ≫ $rB_{4..0}$
sra rC, rA, rB	R-type	0x3B	rC ← (*signed*)rA ≫ $rB_{4..0}$

Table 12: The **R-type** instruction handled by the **R_OP** state.

5.2 Register Operations

Table 12 lists all the instructions that can be handled by the **R_OP** state.

The *shift* operations listed in Table 13 are R-type instructions, but they use a 5-bit immediate value for the second operand. It is recommended to create a new *Execute* state for these instructions.

Instruction	Type	OPX	Description
slli rC, rA, imm	R-type	0x12	rC ← rA ≪ $imm_{4..0}$
srli rC, rA, imm	R-type	0x1A	rC ← (*unsigned*)rA ≫ $imm_{4..0}$
srai rC, rA, imm	R-type	0x3A	rC ← (*signed*)rA ≫ $imm_{4..0}$

Table 13: Additional **R-type** instruction handled by a new *Execute* state.

Table 14 lists additional instructions that can be handled by the **R_OP** state.

Instruction	Type	OPX	Description
cmpne rC, rA, rB	R-type	0x18	rC ← (rA ≠ rB)? 1 : 0
cmpeq rC, rA, rB	R-type	0x20	rC ← (rA = rB)? 1 : 0
cmpleu rC, rA, rB	R-type	0x28	rC ← ((*unsigned*)rA ≤ (*unsigned*)rB)? 1 : 0
cmpgtu rC, rA, rB	R-type	0x30	rC ← ((*unsigned*)rA > (*unsigned*)rB)? 1 : 0
rol rC, rA, rB	R-type	0x03	rC ← rA *rol* $rB_{4..0}$
ror rC, rA, rB	R-type	0x0B	rC ← rA *ror* $rB_{4..0}$

Table 14: Additional instruction handled by the **R_OP** state.

The *rotate* operation listed in Table 15 is a R-type instruction, but uses a 5-bit immediate value for the second operand. Thus, this instruction can be handled by the new *Execute* state introduced in Section 5.2.

Instruction	Type	OPX	Description
`roli rC, rA, imm`	R-type	0x02	rC ← rA *rol* $\text{imm}_{4..0}$

Table 15: A rotate operation handled by the new *Execute* state from Section 5.2.

5.3 Hint for the generation of the op_alu signal

Look carefully at the **OP** or **OPX** fields of the instructions and compare it to the corresponding **ALU** opcode.

- For **I-type** instructions, the 3 most significant bits of the **OP** field can directly be mapped on the 3 least significant bits of the **op_alu** signal.
- For **R-type** instructions, the 3 most significant bits of the **OPX** field can directly be mapped on the 3 least significant bits of the **op_alu** signal.
- Do not take into account the instructions that do not use the **ALU**. This simplifies the generation of **op_alu**.

For the *unconditional* branch, you need to somehow make the **ALU** output `1` for the branch to be taken. What operation can you instruct the **ALU** to perform to always obtain `1` as an output ? Check the value of the operand fields **A** and **B** of the *unconditional* branch instruction for a hint.

5.4 Exercise

- In Quartus, complete the **Controller** to implement the remaining instructions.
- Use `test_Controller.do` to test the complete controller with ModelSim.
- Modify your assembly program `program.asm` to test some of the new instructions.
- Generate the new `ROM.hex` file.
- Compile your Quartus project and program the FPGA. Use ModelSim and the provided testbenches for debugging. See Section 3 for information on how to use `tb_GECKO.vhd` for system-level debugging.

6 Submission

Submit all VHDL files related to the exercises in sections 3, 4.4 and 5.4. (`CPU.vhd`, `IR.vhd`, `PC.vhd`, `buttons.vhd`, `controller.vhd`, `extend.vhd`, `mux2x16.vhd`, `mux2x32.vhd` and `mux2x5.vhd`) and the required files from the previous labs (`add_sub.vhd`, `ALU.vhd`, `comparator.vhd`, `decoder.vhd`, `RAM.vhd`, `logic_unit.vhd`, `multiplexer.vhd`, `register_file.vhd` and `shift_unit.vhd`). **Please note that the files from the previous labs will be tested and checked against plagiarism exactly like the files developed for the first time in this lab.**