Multicycle Nios II Processor

Learning Goal: Simple multicycle processor architecture.

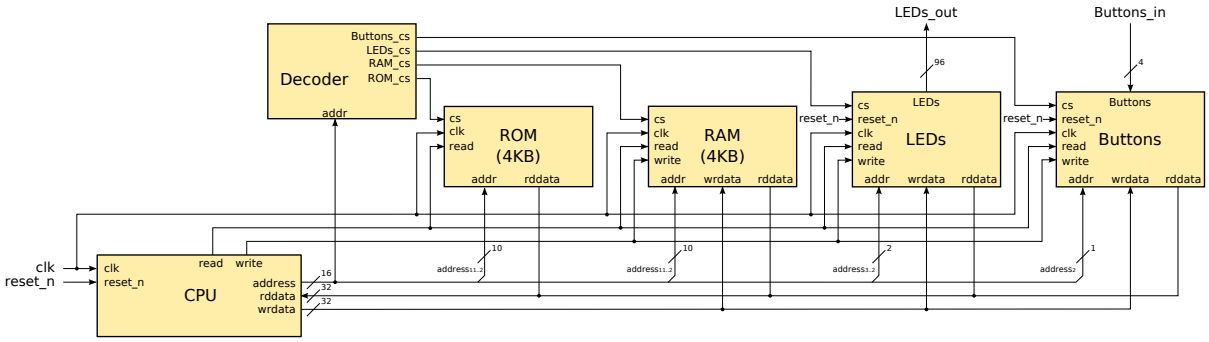
Requirements: FPGA4U, Quartus II Web Edition and ModelSim-Altera.

1 Introduction

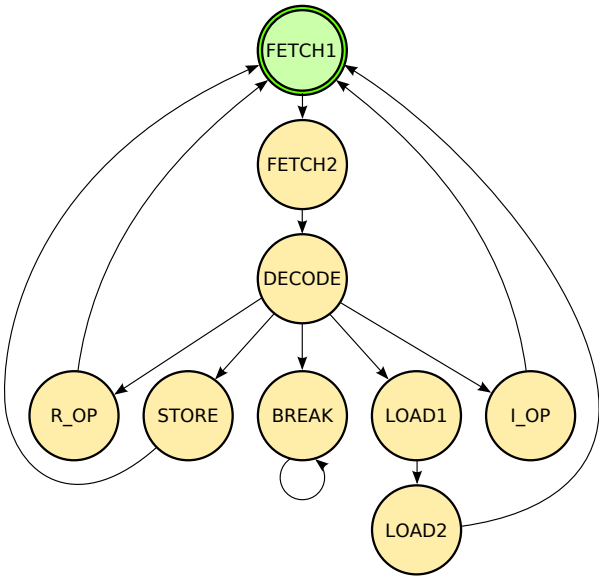
In this exercise you will implement a multicycle Nios II processor. You will use some of the components you built in the previous sessions. You will implement it step-by-step, beginning with a **CPU** that executes a few basic instructions, and extending it progressively to cover all the requested functionalities of the Nios II.

2 Multicycle CPU Description

The first implementation of the **CPU** will only execute several ALU operations (e.g., **`addi`**, **`and`**) and the **`ldw`** and **`stw`** instructions. A **`break`** instruction will also be used to stop the execution of the program. The **CPU** is connected into the same system you built for the **memories** project with an additional **Button** module, which interfaces the four buttons of the FPGA4U:



To execute an instruction, the multicycle **CPU** will need 4 to 5 cycles, depending on the instruction. The following figure is the state machine of the controller of the **CPU**. It illustrates the different steps of the execution of an instruction:

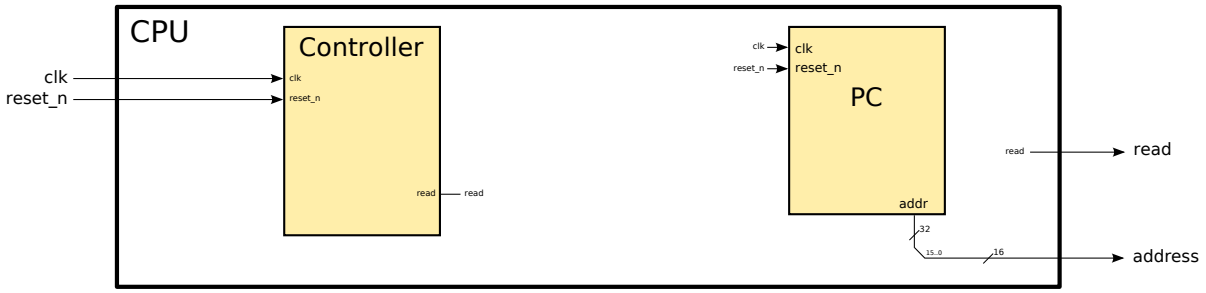


During **FETCH1** and **FETCH2**, the **CPU** reads the next instruction to execute. During **DECODE**, the **CPU** identifies the instruction and determine the next state. During the next states the instruction is executed. We will call these last states *Execute* states.

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 will be available during the next cycle.



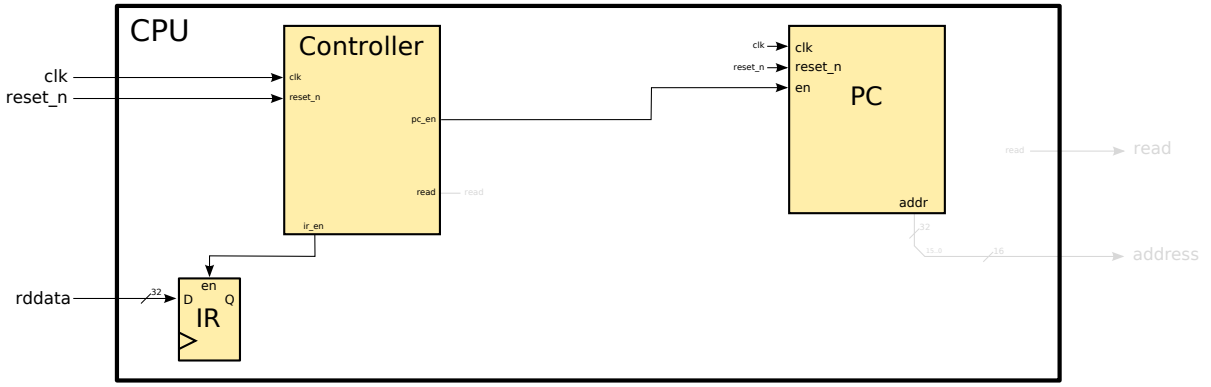
The **Controller** controls the state machine. The input **reset_n** initializes the state machine to **FETCH1**. The **PC** holds the address of the next instruction. The address is stored in a 16-bit register. The address must always be valid, thus the two least significant bits should remains at '0'.

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

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

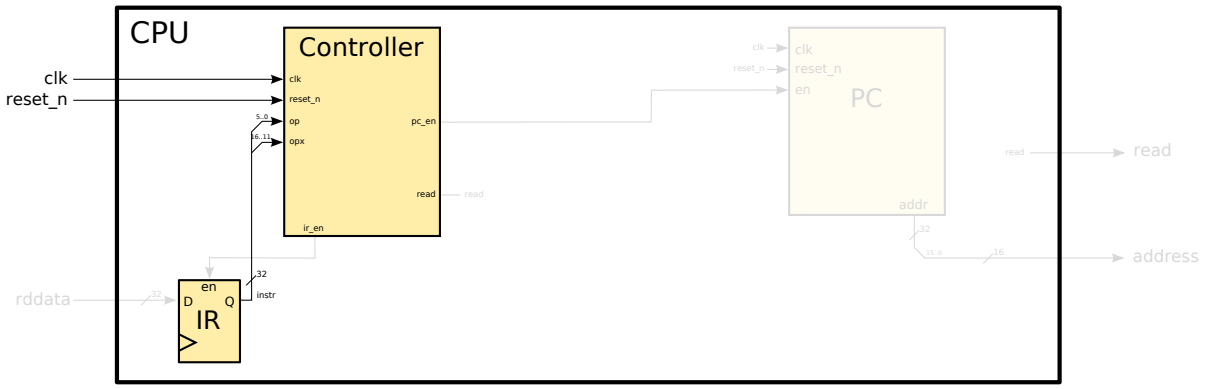


The **Instruction Register** (**IR**) is a 32-bit register that stores the instructions coming from the memory.

- The input **clk** is the clock signal.
- The input **en** enables to write the input **D** in the register at the next rising edge of the clock.
- The output **Q** is the current value of the register.

2.3 DECODE

During this state, the **Controller** reads the opcode of the instruction to identify the current instruction, and determines the next state. The Nios II instructions are progressively described in the following subsections.



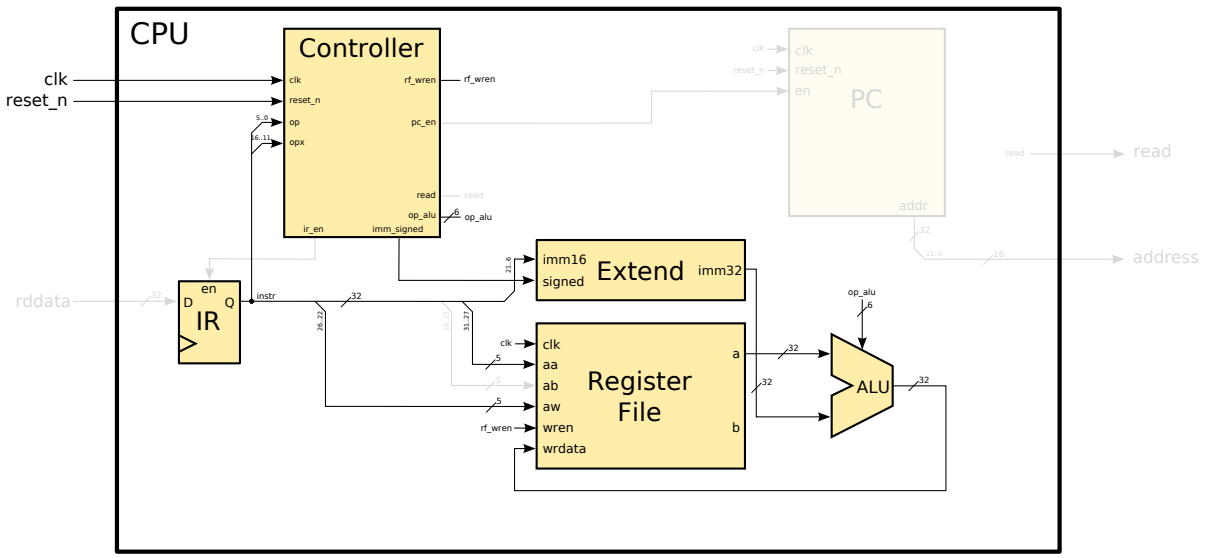
2.4 I_OP

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 a second register. Such instructions with a 16-bit *immediate* embedded value are **I-type** instructions (for Immediate type). The general **I-type** instruction format is detailed below.

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

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**.



The **Register File** and the **ALU** are the same units that you have implemented during the previous sessions. 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**.

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 the following table.

Operation	Operation Type	Opcode
$A + B$	Add/Sub	$000\phi\phi\phi$
$A - B$		$001\phi\phi\phi$
$A \geq B$ (signed)	Comparison	011001
$A < B$ (signed)		011010
$A \neq B$		011011
$A = B$		011100
$A \geq B$ (unsigned)		011101
$A < B$ (unsigned)		011110
$A\ nor\ B$	Logical	$10\phi\phi00$
$A\ and\ B$		$10\phi\phi01$
$A\ or\ B$		$10\phi\phi10$
$A\ xor\ B$		$10\phi\phi11$
$A\ rol\ B$	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*

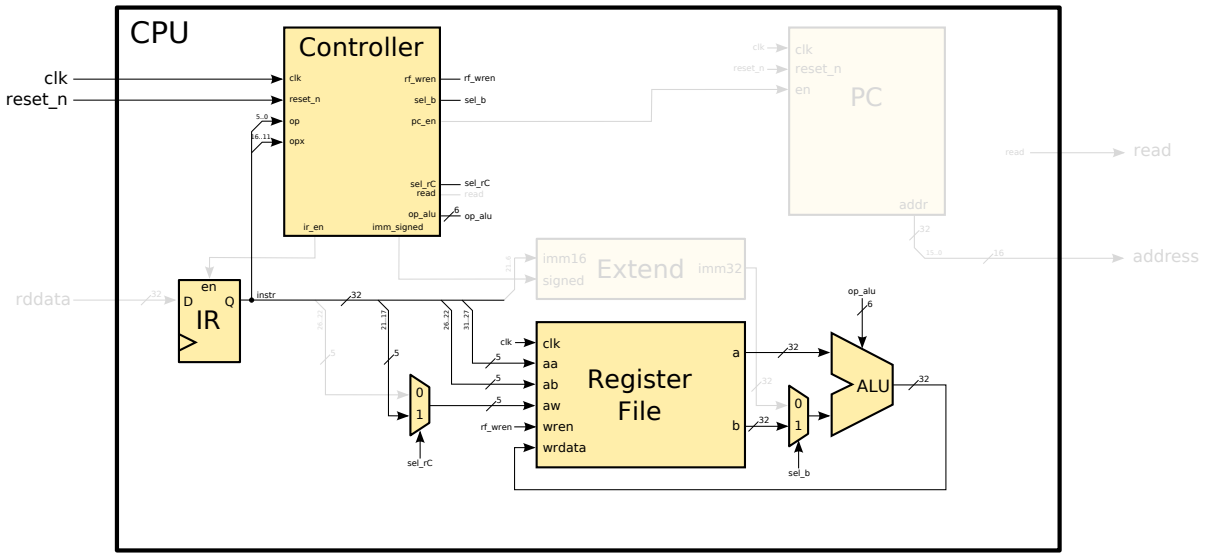
2.5 R_OP

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** instructions (for Register type). The general **R-type** instruction format is detailed below.

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	C	OPX	IMM5	0x3A

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 small 5-bit *immediate* value that is only used by a few **R-type** instructions. The field **OP** is always set to `0x3A` for **R-type** instructions, this is the way to identify them. 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**.



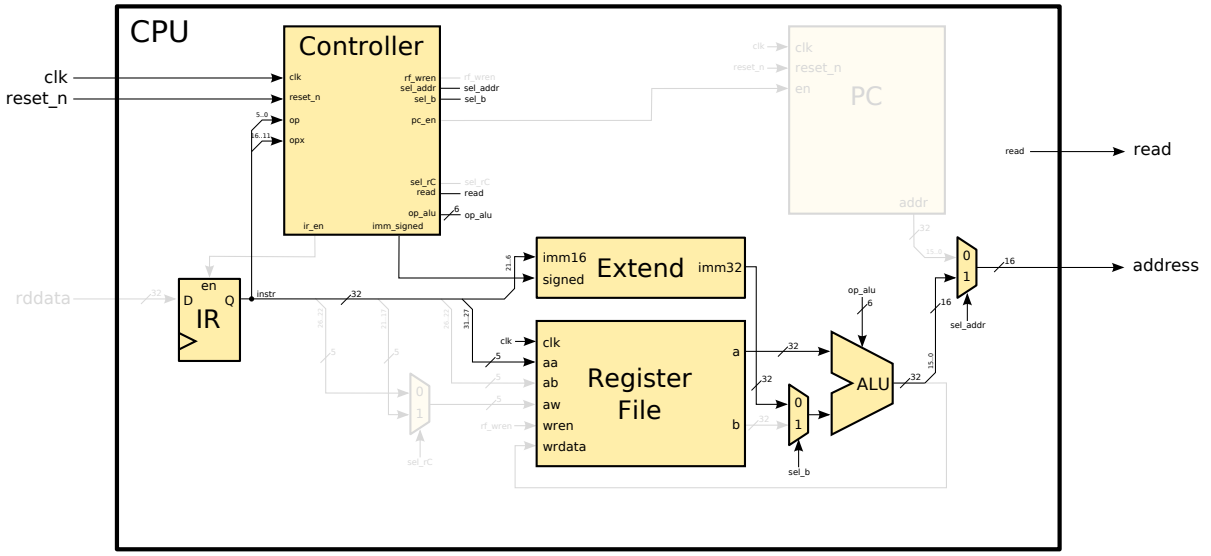
The multiplexer controlled by the signal **sel_b** selects the second operand of the **ALU**: either register **b** (for R-type instructions) or the immediate value (for I-type instructions). The multiplexer controlled by the **sel_rC** signal selects the write address (**aw**) from either the **B**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

2.6 LOAD

The **`ldw`** instruction is a **I-type** instruction with **OP**=`0x17`.

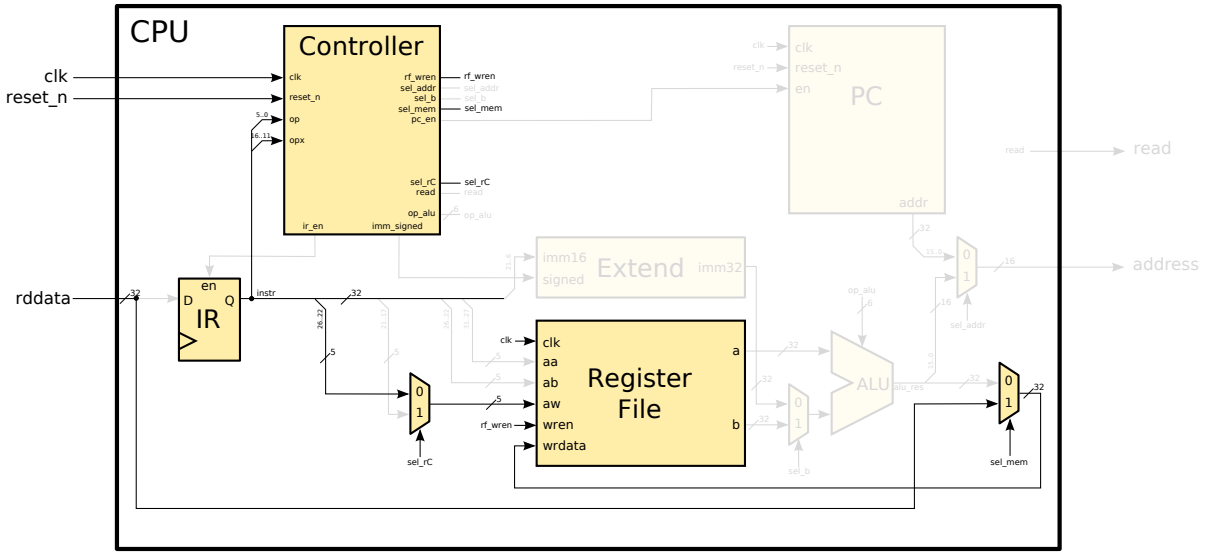
31 30 29 28 27	26 25 24 23 22	21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6	5 4 3 2 1 0
A	B	IMM16	0x17

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**.



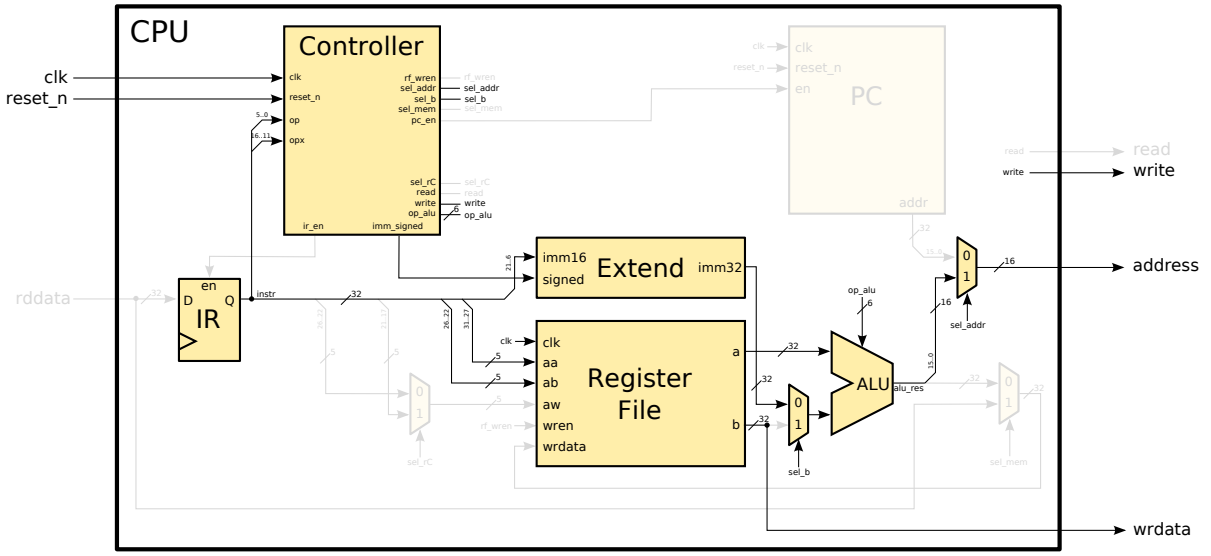
The multiplexer controlled by the signal **sel_mem** selects the data to write to the **Register File** from either the result of the **ALU** or the **rddata** input.

2.7 STORE

The **stw** instruction is a **I-type** instruction with **OP**=`0x15`.

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

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**.



2.8 BREAK

The **break** instruction is a **R-type** instruction with **OPX**=0x34.

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

This instruction will be used to stop 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 `system.bdf` file. Notice that the CPU is connected into the same system that you used during the project **memories**.
- For this exercise, you will use some units you should have implemented during the previous sessions. Copy the following files into the **vhdl** folder of this project, and make sure the filenames and entities matches:
  - For the **ALU**, copy all the VHDL files in the **vhdl** folder of your **ALU** project.
  - For the **Register File**, copy the `register_file.vhd` of your project **register_file**.
  - For the memory system, copy the `ROM.vhd`, `RAM.vhd`, `LEDs.vhd`, `ROM_Block.vhd` and `decoder.vhd` files from 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`).

  If you were not able to finish some of these design files during the previous exercise sessions, you can use their corresponding solution from the **solution** folder. However, you are strongly encouraged to take the time to finish them by yourself, or at least to fully understand them.
- 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`. These multiplexers only differ on their bitwidth.
- Implement the **Extend** unit.
- Implement the **IR**.
- Implement a first version of the **PC**. In this first version, the next address is always the current address incremented by 4.
- Implement a first version of the **Controller**. In this first version, it should be able to decode the following instructions:

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 ≫ $\text{rB}_{4..0}$
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

  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 is only dependent on **OP** and **OPX**. This will simplify the introduction of additional operations.
- Compile the Quartus project and correct the syntax errors.

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 will verify 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 correct. If necessary, use ModelSim to look at the behavior in details. For that you will find a `tb_system.vhd` file that only generates a clock signal in the **testbench** folder. The simulation should take into account the memory initialization file, if it does not you probably specified incorrectly the path to the file.

To test the current (incomplete) **Controller** you should use the `test_Controller0.bat`. You can also use the other testbenches that are provided to verify other components once the controller is fully implemented. For that, open the modelsim folder and execute the corresponding batch files (for example, to test the Extend unit, execute the `test_Extend.bat` file).

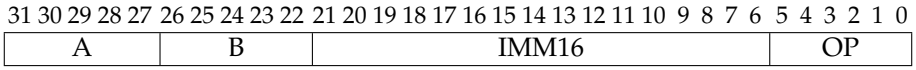
- Modify the assembly program to test all of the instructions that you have implemented. For example, you can write a program that displays the result of an addition onto the LEDs.

4 Extending the multicycle CPU with flow control

In this section you will add flow control to the CPU. This will enable the CPU to do conditional jumps in the code with the *branches* instructions and to call procedures with 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

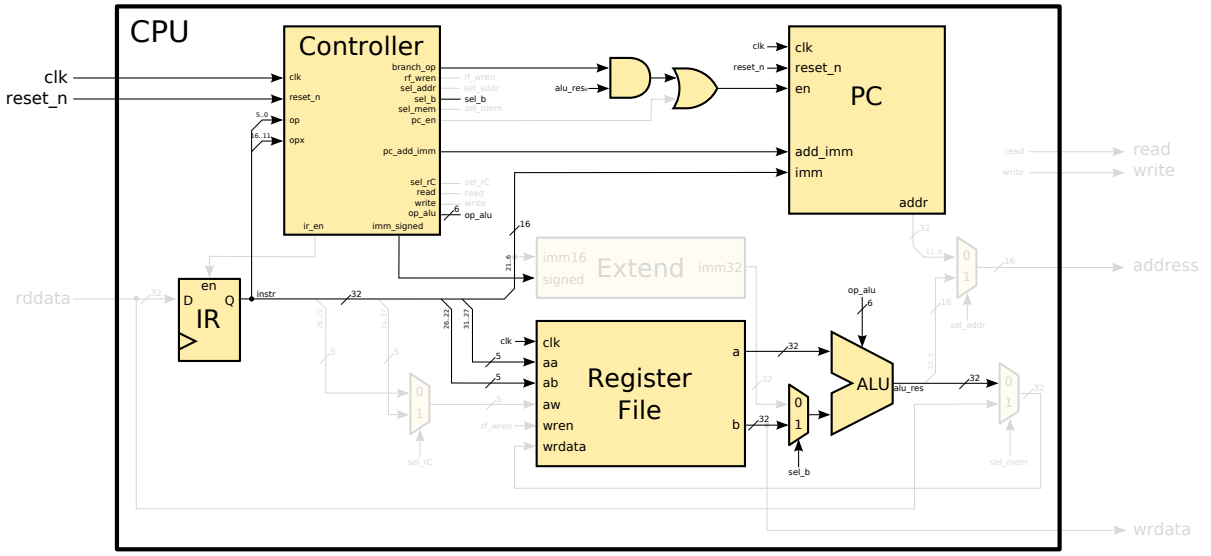
The **BRANCH** state executes *branch* instructions, which are **I-Type** instructions.



The following table describes the different *branch* instructions.

Instruction	Type	OP	Jumps to `label` if:
`br` `label`	I-type	0x06	*no condition*
`bge` `rA, rB, label`	I-type	0x0E	`rA` $\geq$ `rB`
`blt` `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`
`bgeu` `rA, rB, label`	I-type	0x2E	(*unsigned*)`rA` $\geq$ (*unsigned*)`rB`
`bltu` `rA, rB, label`	I-type	0x36	(*unsigned*)`rA` $<$ (*unsigned*)`rB`

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 being the address of the current instruction). But 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**.



Here, some logical gates have been added so that when the **branch_op** signal and the least significant bit of the result of the ALU are active, the **PC** is enabled. 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.

This is a small example of a loop made with a *branch* instruction:

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

4.2 CALL

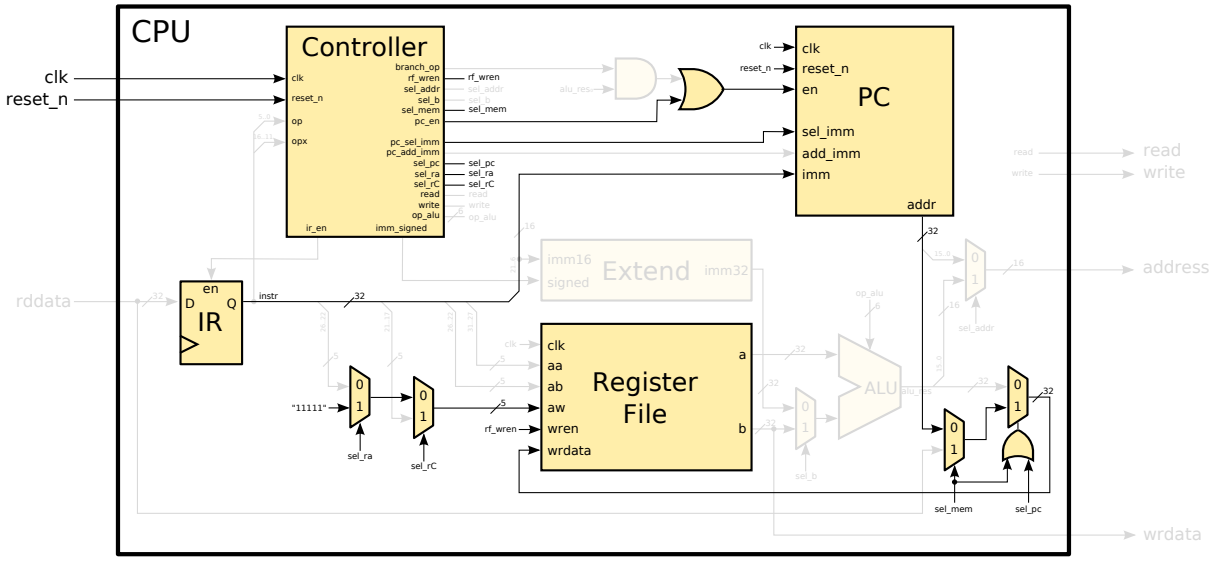
The **CALL** state executes the **call** instruction, which is a **I-type** instruction with **OP**=0x00.

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

The following table describes the **call** instruction.

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

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.



The **pc_sel_imm** signal selects the immediate field as the next value of the **PC**. In the `call` instruction, the embedded address is word aligned. Since the **PC** is byte 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 (address 31 in the **Register File**).

4.3 JMP

The **JMP** state executes the `jmp` and `ret` instructions, which are **R-type** instructions with **OPX**=`0x0D` and **OPX**=`0x05`, respectively.

`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:

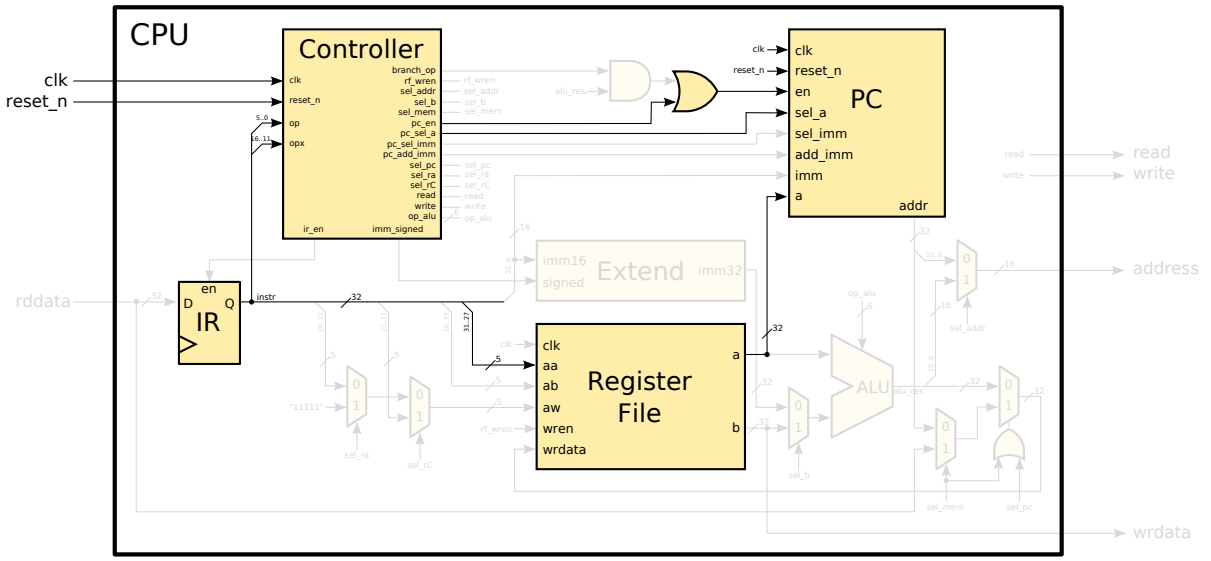
31 30 29 28 27	26 25 24 23 22	21 20 19 18 17	16 15 14 13 12 11	10 9 8 7 6	5 4 3 2 1 0
0x1F	0x00	0x00	0x05	0x00	0x3A

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

The following table describes the `ret` and `jmp` instructions.

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

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**.

4.4 JMPI and CALLR Instructions

Read the descriptions of the `jmpi` and `callr` instructions and implement them.

4.4.1 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.

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

4.4.2 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**.

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

The field **C** of the **`callr`** instruction is implicitly set to `ra` (address 31 in the **Register File**).

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

4.5 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 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 will be to generate, from the instruction, the correct value of the **op_alu** signal. We will give you some hints to generate it efficiently.

5.1 Immediate Operations

The following table lists all the instructions that can be handled by the **I_OP** state:

Instruction	Type	OP	Description
addi rB, rA, imm	I-type	0x04	rB ← rA + (*signed*)imm
cmpgei rB, rA, imm	I-type	0x08	rB ← (rA ≥ (*signed*)imm)? 1 : 0
cmplti 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

The following *immediate* operations can not be handled by the **I_OP** state. The immediate value must be considered as an *unsigned* number. You will have 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
xori rB, rA, imm	I-type	0x1C	rB ← rA *xor* (*unsigned*)imm
cmpgeui rB, rA, imm	I-type	0x28	rB ← (*unsigned*)rA ≥ (*unsigned*)imm
cmpltui rB, rA, imm	I-type	0x30	rB ← (*unsigned*)rA < (*unsigned*)imm

5.2 Register Operations

The following tables lists all the instructions that can be handled by the **R_OP** 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
cmpge rC, rA, rB	R-type	0x08	rC ← (rA ≥ rB)? 1 : 0
cmplt rC, rA, rB	R-type	0x10	rC ← (rA < rB)? 1 : 0
cmpne rC, rA, rB	R-type	0x18	rC ← (rA ≠ rB)? 1 : 0
cmpeq rC, rA, rB	R-type	0x20	rC ← (rA = rB)? 1 : 0
cmpgeu rC, rA, rB	R-type	0x28	rC ← ((*unsigned*)rA ≥ (*unsigned*)rB)? 1 : 0
cmpltu rC, rA, rB	R-type	0x30	rC ← ((*unsigned*)rA < (*unsigned*)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
xor rC, rA, rB	R-type	0x1E	rC ← rA *xor* rB
rol rC, rA, rB	R-type	0x03	rC ← rA *rol* $rB_{4..0}$
ror rC, rA, rB	R-type	0x0B	rC ← rA *ror* $rB_{4..0}$
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 ≫ $rB_{4..0}$

The following *shift* and *rotate* operations are R-type instructions, but use a small 5-bit immediate value for the second operand. These instructions can not be handled by the **R_OP** state. You will have to create a new *Execute* state for these instructions.

Instruction	Type	OPX	Description
roli rC, rA, imm	R-type	0x02	rC ← rA *rol* $\text{imm}_{4..0}$
slli rC, rA, imm	R-type	0x12	rC ← rA $\ll$ $\text{imm}_{4..0}$
srli rC, rA, imm	R-type	0x1A	rC ← (*unsigned*)rA $\gg$ $\text{imm}_{4..0}$
srai rC, rA, imm	R-type	0x3A	rC ← (*signed*)rA $\gg$ $\text{imm}_{4..0}$

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.
- Don't take into account the instructions that do not use the **ALU**, this will simplify the generation of **op_alu**.

5.4 Exercise

- Complete the **Controller** to implement these remaining instructions.
- Compile and correct any error.
- 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 testbenches that are provided for debugging.

6 Optimizing the state machine

Look at the state machine of the **Controller**. Is it always necessary to go back to **FETCH1** after the execution of an instruction? For example, from **I_OP**, could we directly go to **FETCH2**?

6.1 Exercise

- Explain why and how.
- Modify the **Controller** to bypass **FETCH1** when it is possible.

7 Submission

Submit all vhdl files related to the exercises in sections 3, 4.5, 5.4 and 6.1 (`CPU.vhd`, `IR.vhd`, `PC.vhd`, `System.vhd`, `buttons.vhd`, `controller.vhd`, `extend.vhd`, `mux2x16.vhd`, `mux2x32.vhd` and `mux2x5.vhd`) and the required files from the previous labs (`ALU.vhd`, `add_sub.vhd`, `comparator.vhd`, `logic_unit.vhd`, `multiplexer.vhd`, `register_file.vhd` and `shift_unit.vhd`).