Pipelining The Nios II

Learning Goal: Processor pipeline.

Requirements: Quartus, ModelSim, Nios2Sim.

1 Introduction

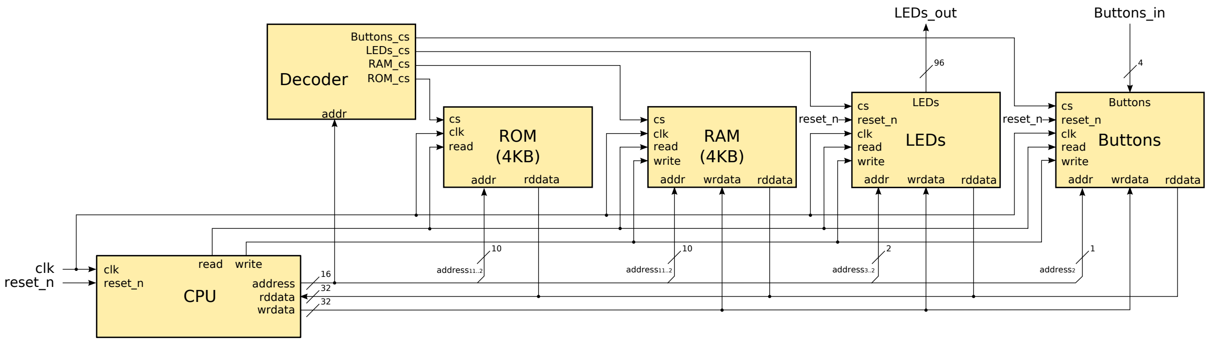
During this lab, you will create a relatively simple pipelined version of the Nios II. Starting from the multi-cycle processor you implemented in ArchOrd, you will reorganize the components, add some register stages, and modify the **Program Counter** and the **Controller**. If you did not attend ArchOrd, you will find a copy of the assignment in the Additional Material on Moodle.

2 The CPU Pipeline

You will implement a relatively simple 5-stage pipeline.

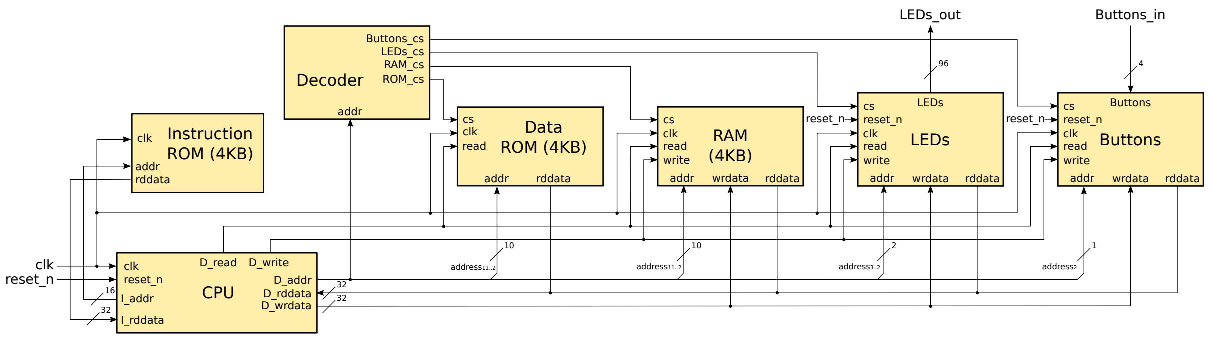
- *Harvard* architecture.
- 5 stages (*Fetch*, *Decode*, *Execute*, *Memory*, and *Writeback*).
- All instructions go through these 5 stages, even if some of them are not used.
- There are no forwarding paths, stalls or flushes.
- The branch instructions (including `br`) have 2 delay slots.
- The jump instructions (e.g., `jmp`, `call`, `ret`) have a single delay slot.

In the multi-cycle version of the Nios II (shown in the following figure), instruction and data memory accesses use the same memory port, since these different accesses never occurred at the same time.

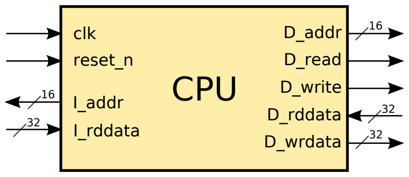


In the pipelined version of the Nios II, the processor reads the instruction memory at every cycle. Therefore, it is not possible to use a single memory port without stalling the pipeline when a data access occurs.

To simplify the implementation and avoid stalling, we will switch to a *Harvard* architecture (i.e., have separated ports for instructions and data). The original single memory port of the multi-cycle processor (i.e., the one connected to the ROM, the RAM and the peripherals) will become the data port. An instruction port is introduced and connected to a duplicate copy of the ROM (see `GECKO.bdf`).

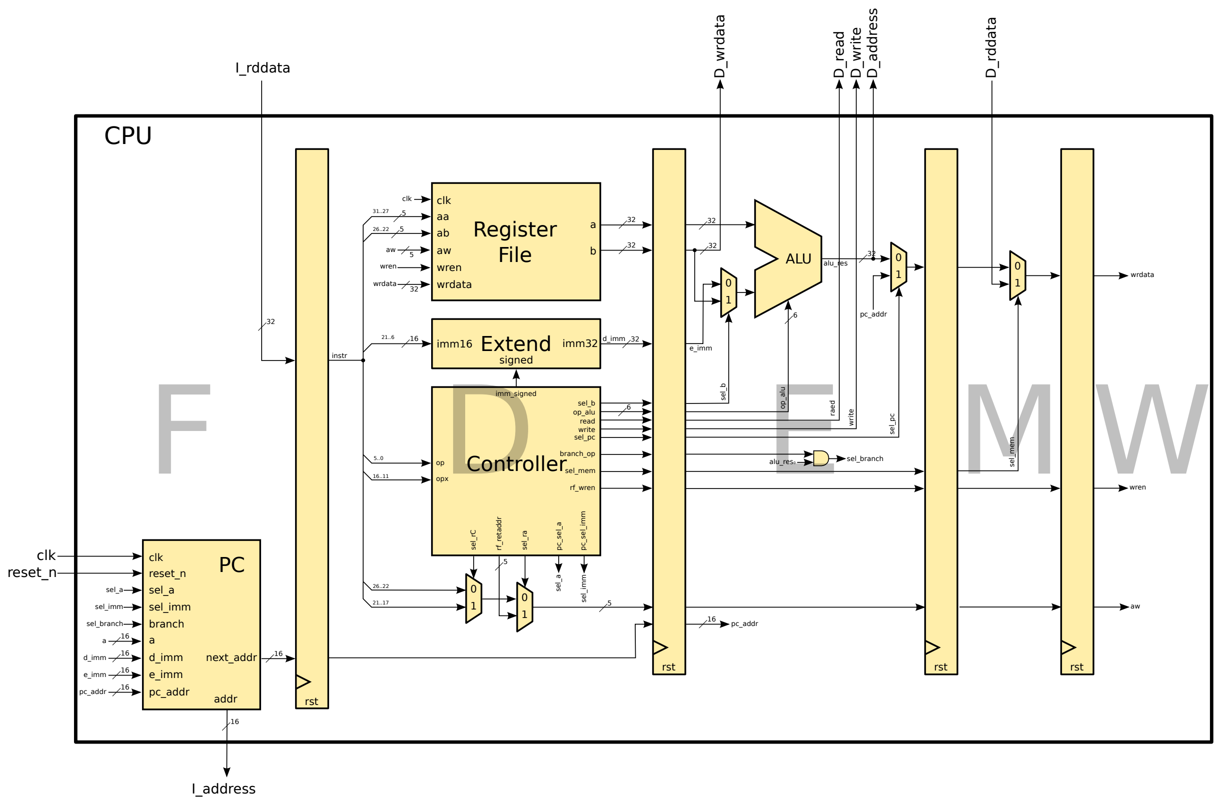


The following figure illustrates the new CPU entity.



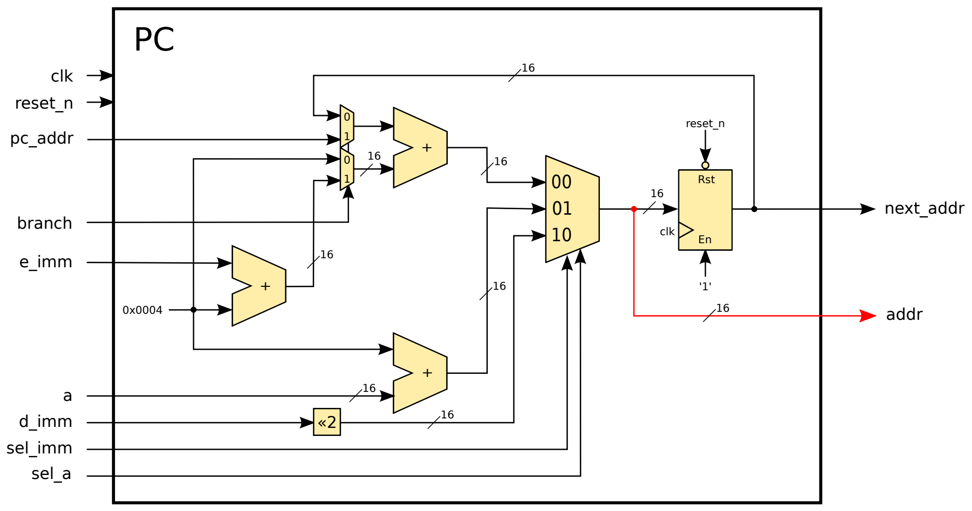
The following figure shows what the architecture of the your pipeline could be. You can see that most of the components implemented for the multi-cycle processor can be reused without any modification. The function of each of the five stages is very close to the states of the multi-cycle processor.

- Fetch: The next instruction and the next instruction address (i.e., PC+4) are fetched in the first register stage.
- Decode: The instruction is decoded by the **Controller**, which generates the control signals for the next stages and also for the **PC**, so that in the case of a jump instruction, the **PC** is directly updated during the *Decode* stage. Additionally, the register operands are read from the **Register File** and stored in the next register stage.
- Execute: This is where the **ALU** operations occur. It's during this cycle that the decision of taking the branch is done. The next instruction address and the immediate value are sent to the **PC**. The memory control signals are sent to the memory port, so that in the case of a read, the data is received during the next stage. The result of the **ALU** and some of the control signals are stored in the next register stage.
- Memory: In the case of a **`ldw`** instruction, the data from memory is stored in the next stage.
- Writeback: The control signals are sent to the **Register File** to write the result of the instruction.



The **Program Counter** and the **Controller** are the only modules that require to be modified. The following subsections will give you a description of what must be done.

2.1 The Program Counter



- In this pipelined version of the Nios II, the **PC** is always enabled and loads a new instruction at every cycle. Therefore, the **en** signal is not required.
- The **a** and **imm** input values must be provided by the *Decode* stage with their corresponding control signals (i.e., **sel_a** and **sel_imm**).

- Reduce the latency of the pipeline by providing the **addr** signal to the ROM directly after the next address selection, and **before** the counter register (see the figure above).

2.2 The Controller

- Now that we have a *Harvard* architecture, the **sel_addr** control signal becomes useless.
- The **PC** being always enabled, the **pc_en** signal becomes useless as well.
- The **pc_add_imm** signal cannot be provided directly by the **Controller**. Instead, it is generated in the *Execute* stage. In the new **Controller**, the **pc_add_imm** signal corresponds to the **branch** signal.
- The **Controller** becomes asynchronous. Instead of a state machine, you should compute the control signals in a combinatorial way.
- You can ignore the `break` instruction.

3 Exercise

- Open the Quartus project `quartus/GECKO.qpf`.
- Implement the **PC**, **Controller** and the **pipeline registers** in the respective `.vhd` files.
- Update the block diagram in `CPU.bdf` according to the schematic described in Section 2.
- To verify your design, write a simple program in Nios2Sim. This program should call a procedure, do some branches, and give some feedback (through the LEDs for example).
- Don't forget to take into account the fact that the current pipeline doesn't care about data hazards, and that there are delay slots for the branch and jump instructions. Insert `nop` instructions when its necessary.
- Generate the `hex` file and save it as both `data_ROM.hex` and `instruction_ROM.hex`. Compile your design and program your FPGA.
- If the program is not executed properly, simulate the modules you implemented in ModelSim by writing a testbench. Use the VHDL Testbench Tutorial under Additional material on Moodle and the previous labs as a reference.
- Would it be difficult to flush the first stages of the pipeline in the case of a jump or a branch? Think of a solution and propose it to an assistant. If you have the time, try to implement it.

4 Submission

Generate the VHDL file from the `CPU.bdf` block diagram:

- In the **Project Navigator**, select `CPU.bdf`.
- Click on **File** → **Create / Update** → **Create HDL Design File from Current File...**
- Select VHDL and choose a path for the generated `CPU.vhd` file.

Submit the files `CPU.vhd`, `PC.vhd` and `controller.vhd`.