Task 1: Pipelined Microprocessor

NOTE: It is important to read the complete description of each exercise before starting with the implementation.

The objective of this task is to implement a pipelined version of the MIPS processor. A detailed description of the pipelined MIPS processor can be found in the book: *"Computer Organization and Design: The Hardware/Software Interface"*, by David A. Patterson and John L. Hennessy, available in the EPS library. It is recommended to follow the book to complete this task, in particular, chapter 6 (sections 6.2 y 6.3).

Design Overview

It is requested to implement the MIPS microprocessor in its single-cycle version, which will be used as the basis for the pipelined version. The processor does not need to support the full MIPS instruction set, but the following instructions: **ADD, ADDI, SUB, SLTI, AND, ANDI, OR, XOR, LUI, LW, SW, J, BEQ and NOP**, whose opcodes and description are included below. In any case, after pipelined the processor, the *beq* instruction, which involves control risks because it is a jump, will work "abnormally" in this basic pipelined version of the processor.

To make easier to complete this task, a simplified version of the single-cycle version of the processor is provided with the different sub-blocks of the processor already defined: the register bank, the arithmetic logic unit (ALU), the control unit (“Control" in Figures 2 and 3), the block that generates the control codes for the ALU ("ALU control") and the processor itself. Students must complete these files by writing the code for the corresponding VHDL architectures.

On the other hand, to verify the processor, a complete VHDL file is provided, with a very simple testbench that instantiates the micro and the instruction and data memories, for which a complete VHDL file is also included.

The hierarchical relationship in the design is the following (see figure 1):

- The testbench (processor_tb), instantiates the processor (processor) and the 2 memories (memory).
- The processor (processor) instantiates the register bank (reg_bank), the ALU (alu), the control unit (control_unit) and the block for controlling the ALU (alu_control).
- The runsim_arq.do file allows launching the simulation by invoking Modelsim / Questasim script from the modelsim console. This file compiles, simulates and opens the waveforms described in wave_arq.do

All register components must use the rising edge of the clock, and asynchronous active high reset. On the other hand, for the encoding of the processor it is recommended:

- Do not create additional components: make the program counter, the pipeline registers, multiplexers, etc. as code within the processor architecture.
- Use concurrent assignments (simple and conditional).

The provided material has the following structure, which must be maintained:

- Directorio *rtl/* : processor source code:

processor.vhd	processor, incomplete
alu.vhd	ALU, complete
reg_bank.vhd	Register bank, complete
control_unit.vhd	Control unit, incomplete
alu_control.vhd	ALU control, incomplete

- Directorio *sim/* : simulation tools, scripts and source code (all files are complete, except *wave.do*, where students must save their custom waveform):

processor_tb.vhd	Testbench
memory.vhd	Synchronous memory model
programa.s	Simple test assembler program
programa.lst	Program coding list
instrucciones	Data file for data memory
datos	Data file for instructions memory
runsim.do	Script to run ModelSim simulation
wave.do	Script de configure ModelSim Waveform

The testbench instantiates the memories, which read, in an initialization phase at time = 0, the data from the files "*instrucciones*" and "*datos*" (note that these memories are models that simplify what would be a scenario with real memories). These two files result from the assembly of *program.s*. In addition, the *program.lst* file allows us to know in which direction each instruction is and how it is encoded. This program executes all the instructions of the processor and, as its comments explain, it includes instructions that, due to data and control risks, cannot be correctly executed by this version of the processor. As an additional reference, in Moodle you can find a link with the equivalence between MIPS register names and their number 0-31 (*registers.html*).

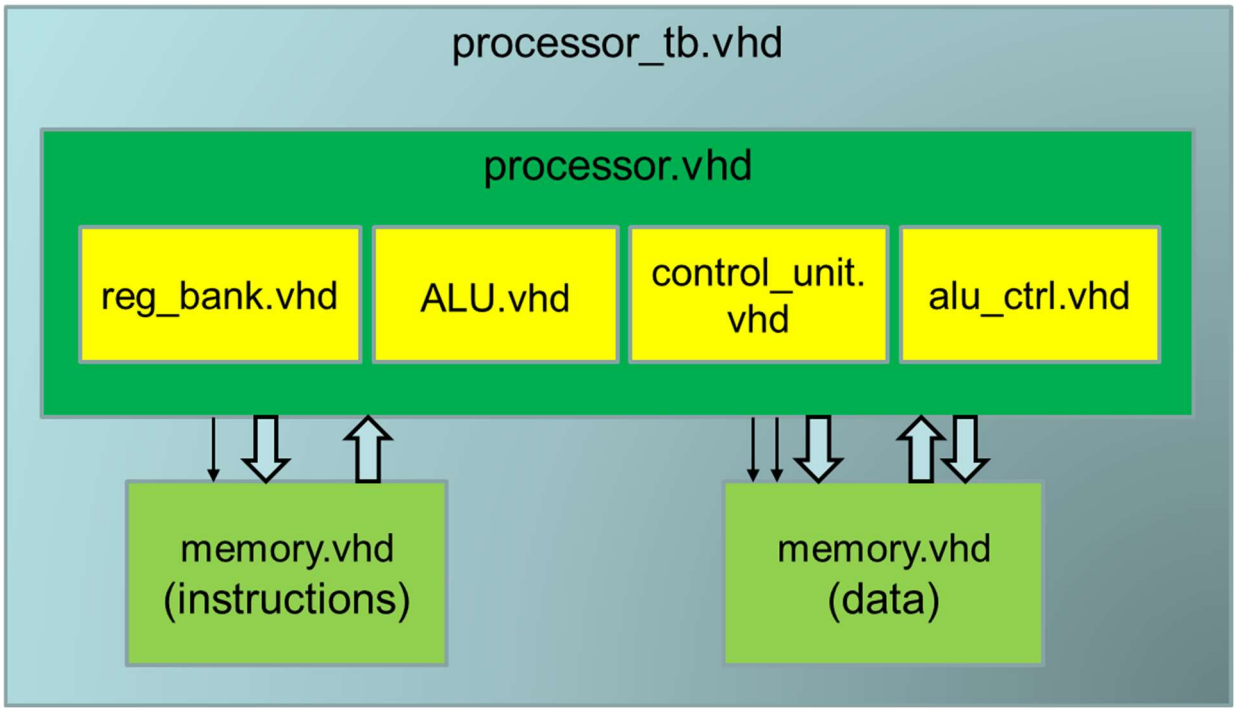


Figure 1. VHDL design file hierarchy

The diagram of the single-cycle processor to be implemented (completed) is shown in the following figure:

NOTA: To complete this exercise, the VDHL files from the course "*Estructura de Computadores*" (from the second semester of the first year) can be reviewed and reused. But you must use the provided files.

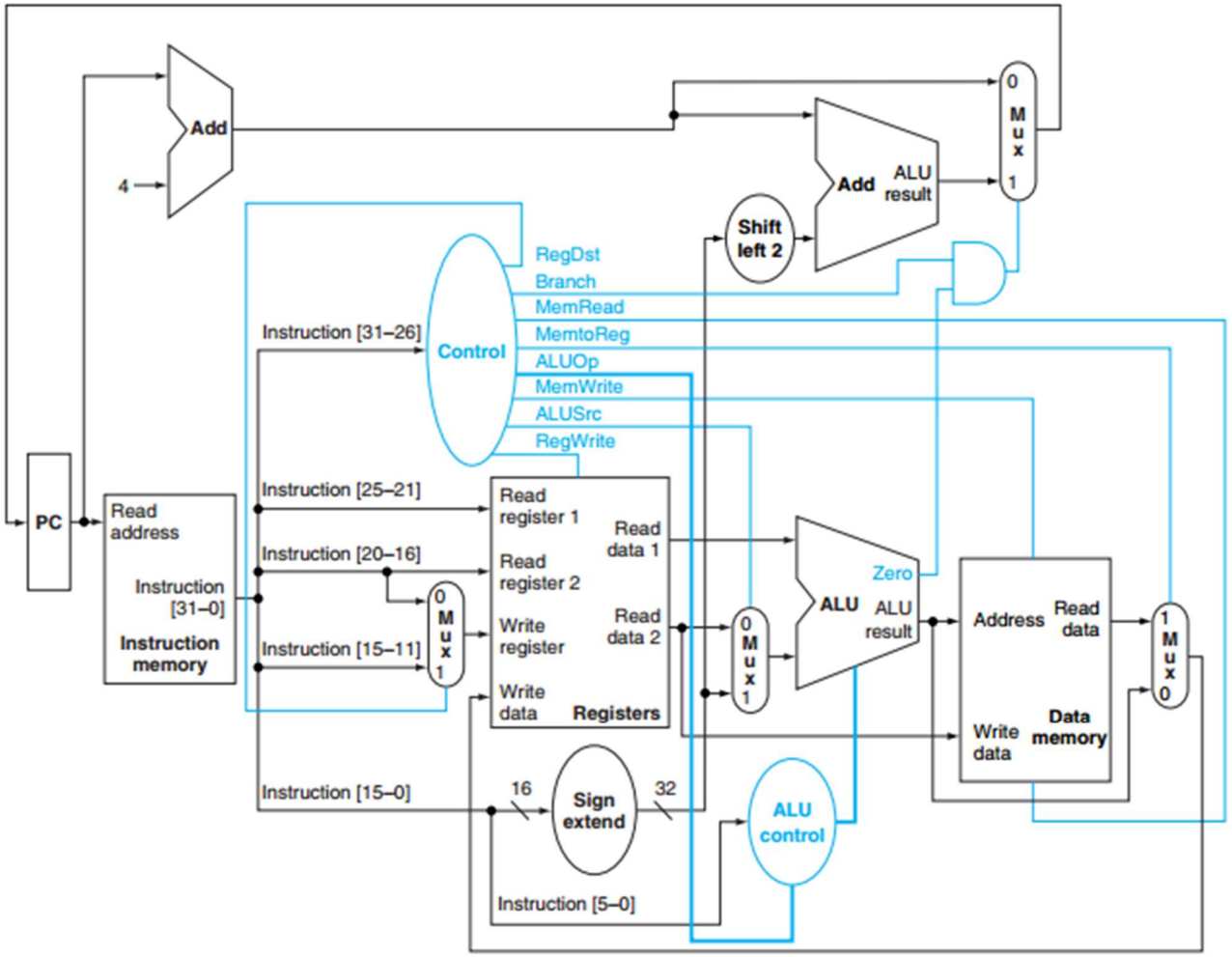
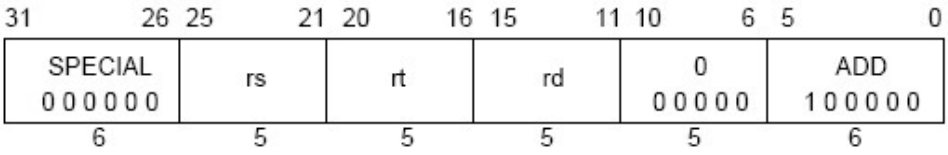


Figure 2. Single-cycle processor.

The instructions to implement have the following description:

ADD — **Add Word**



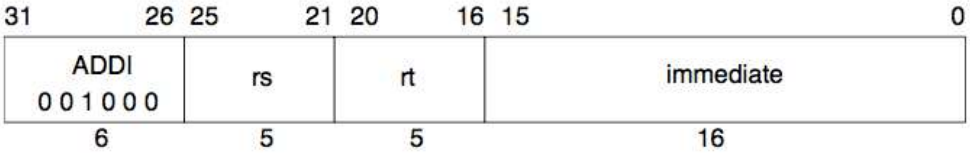
MIPS I

Format: ADD rd, rs, rt

Purpose: To add 32-bit integers. If overflow occurs, then trap.

Description: rd ← rs + rt

Add Immediate Word — **ADDI**



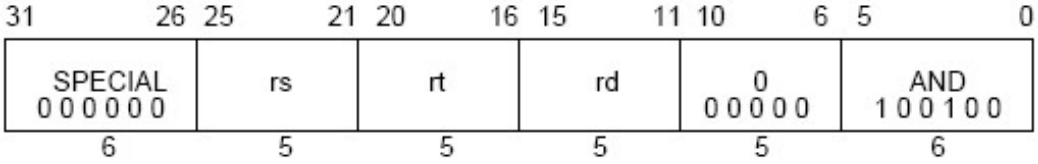
MIPS I

Format: ADDI rt, rs, immediate

Purpose: To add a constant to a 32-bit integer. If overflow occurs, then trap.

Description: rt ← rs + immediate

AND — **And**



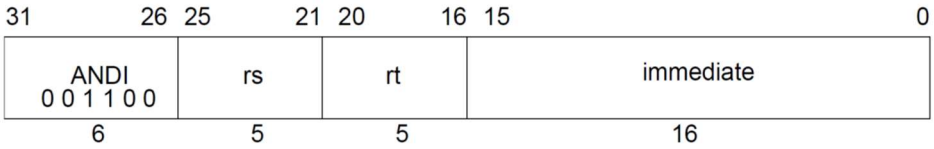
MIPS I

Format: AND rd, rs, rt

Purpose: To do a bitwise logical AND.

Description: rd ← rs AND rt

And Immediate **ANDI**

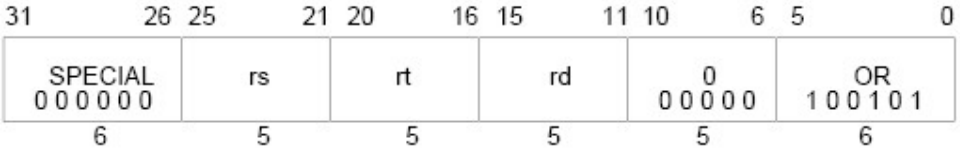


Format: ANDI rt, rs, immediate **MIPS I**

Purpose: To do a bitwise logical AND with a constant.

Description: rt ← rs AND immediate

OR **Or**

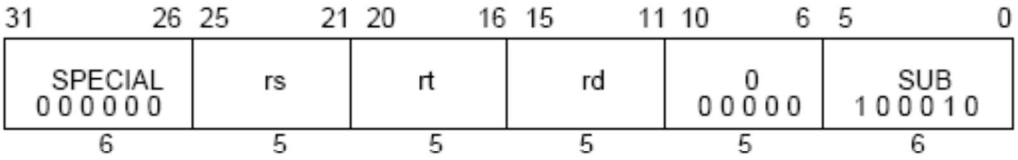


Format: OR rd, rs, rt **MIPS I**

Purpose: To do a bitwise logical OR.

Description: rd ← rs OR rt

SUB **Subtract Word**



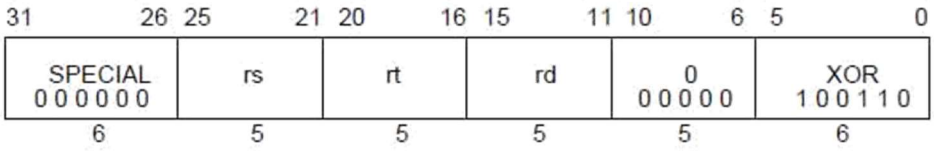
Format: SUB rd, rs, rt **MIPS I**

Purpose: To subtract 32-bit integers. If overflow occurs, then trap.

Description: rd ← rs - rt

XOR

Exclusive OR



MIPS I

Format: XOR rd, rs, rt

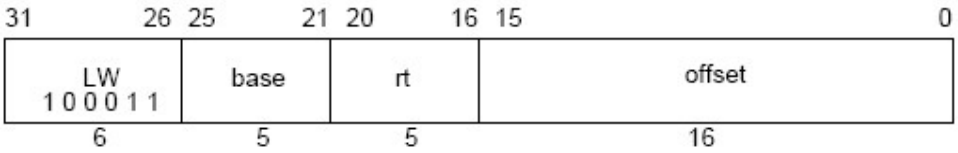
Purpose: To do a bitwise logical EXCLUSIVE OR.

Description: rd ← rs XOR rt

Combine the contents of GPR *rs* and GPR *rt* in a bitwise logical exclusive OR operation and place the result into GPR *rd*.

LW

Load Word



MIPS I

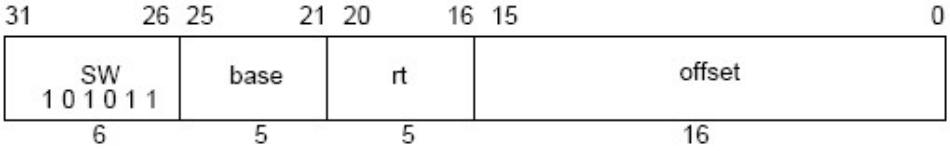
Format: LW rt, offset(base)

Purpose: To load a word from memory as a signed value.

Description: rt ← memory[base+offset]

SW

Store Word



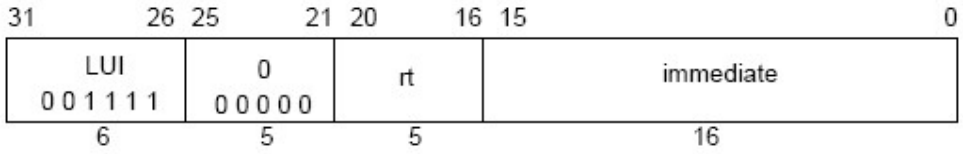
MIPS I

Format: SW rt, offset(base)

Purpose: To store a word to memory.

Description: memory[base+offset] ← rt

Load Upper Immediate **LUI**

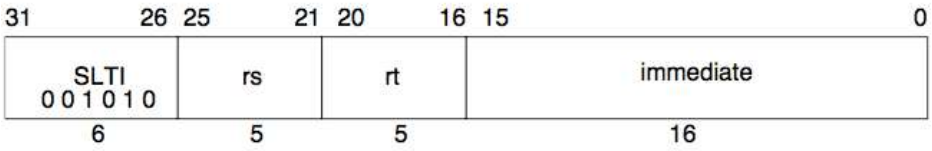


Format: LUI rt, immediate **MIPS I**

Purpose: To load a constant into the upper half of a word.

Description: rt ← immediate || 0^{16}

Set on Less Than Immediate **SLTI**



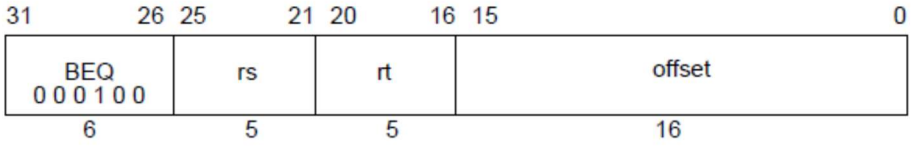
Format: SLTI rt, rs, immediate **MIPS I**

Purpose: To record the result of a less-than comparison with a constant.

Description: rt ← (rs < immediate)

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate* the result is 1 (true), otherwise 0 (false).

BEQ **Branch on Equal**



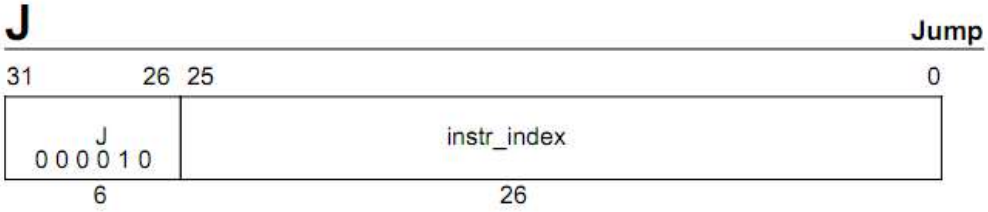
Format: BEQ rs, rt, offset **MIPS I**

Purpose: To compare GPRs then do a PC-relative conditional branch.

Description: if (rs = rt) then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (**not** the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are equal, branch to the effective target address after the instruction in the delay slot is executed.



Format: J target **MIPS I**

Purpose: To branch within the current 256 MB aligned region.

Description:

This is a PC-region branch (not PC-relative); the effective target address is in the "current" 256 MB aligned region. The low 28 bits of the target address is the *instr_index* field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (**not** the branch itself).

Jump to the effective target address. Execute the instruction following the jump, in the branch delay slot, before jumping.

The NOP instruction

The NOP instruction is not defined as an instruction in the MIPS instruction set, but as a particular case of another instruction. Regarding this task, the NOP instruction has all 32 bits set to '0'. The behavior of the processor for this instruction must be to not modify any of its resources (register bank, data memory).

Exercise 1 (2 points)

It is requested to complete the MIPS microprocessor in its single-cycle version. The provided design does not support the set of instructions described before (**ADD, ADDI, SUB, SLTI, AND, ANDI, OR, XOR, LUI, LW, SW, J, BEQ and NOP**).

Exercise 2 (1 point)

It is requested to make a simple assembler program capable of testing the instructions requested in Exercise 1.

Exercise 3 (7 points)

It is requested to implement the pipelined MIPS microprocessor. The result should be a processor capable of execute, in the ideal case (without risks), one instruction per clock cycle. For this exercise, it is not necessary the processor supports risks (except the structural one: access to separate memories of instructions and data).

All registers must meet the following characteristics:

1. Use on the rising edge of the clock (rising_edge (clk)).
2. Reset asynchronously using the "Reset" signal of the "processor_core" entity.

It is recommended to use the next processor diagram (slightly different from the one in the book).

Note: It is highly recommended to have a printed version of the diagram to make annotations (or a digital version to write down the names of signals).

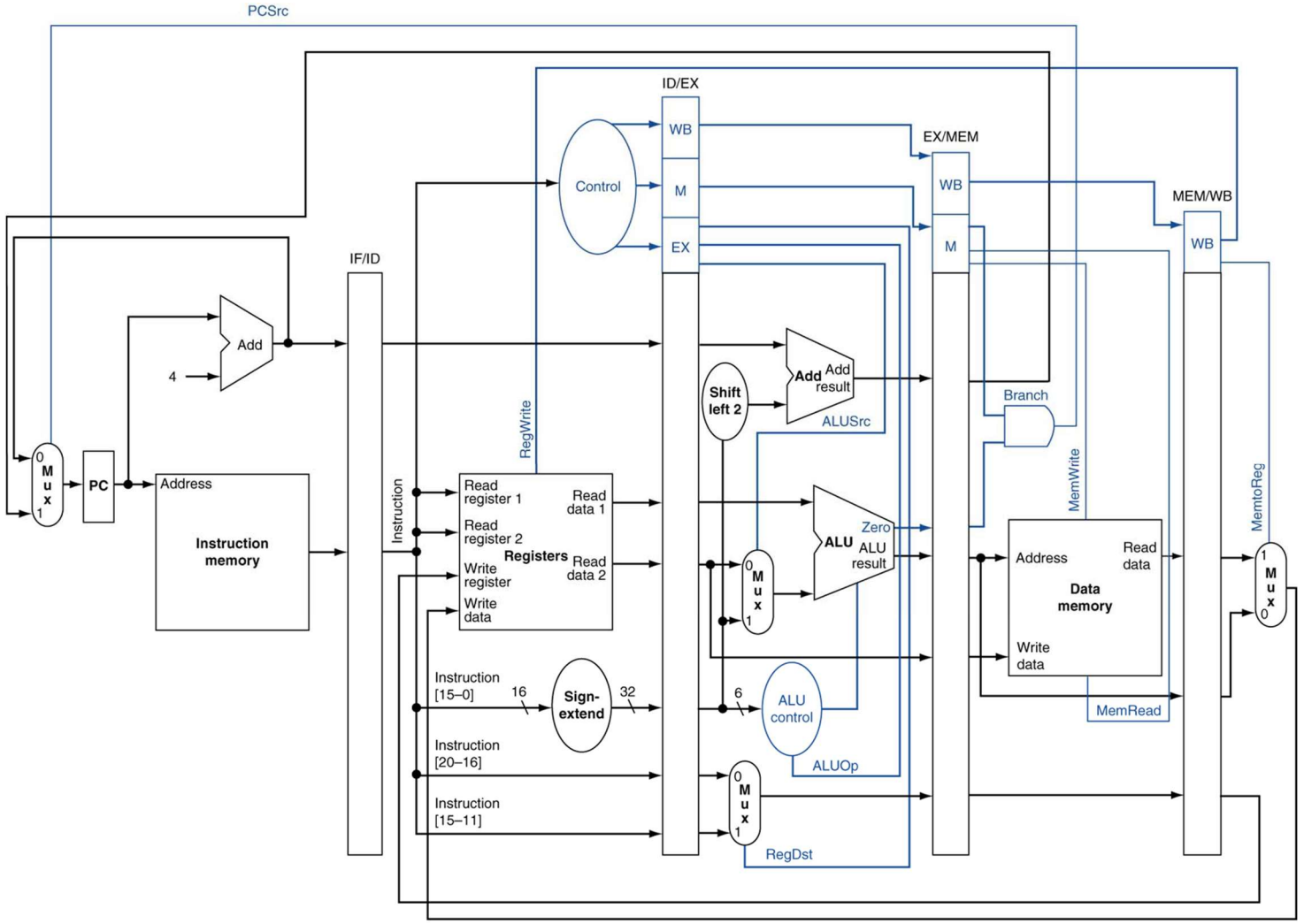


Figure 3. Pipelined microprocessor.

MATERIAL TO SUBMIT

VHDL files from exercises 1 and 3 and the assembler program from exercise 2. A text file named P1_grupoxxxx.txt where xxxx is the team number, and it must include the team members names and any explanation if necessary to understand the exercise.

Use 3 folders, one per exercise.

Submit a zip file through Moodle. The deadline is the last Friday of the last week of the task until 11:59 pm. For this task, Friday October 9, 2020.

* The professor could request a defense of the task as part of the grade.

HELP AND NOTICES

Files "instrucciones" and "datos" must be located in the same folder of the simulation project. If not the case, in file "processor_TB.vhd" the full path of these files can be given by changing the lines of code:

```
C_ELF_FILENAME        => "instrucciones",
...
C_ELF_FILENAME        => "datos",
```

By the complete path to both files:

```
C_ELF_FILENAME        => "X:\nombredirectorio\instrucciones",
...
C_ELF_FILENAME        => "X:\nombredirectorio\datos",
```

In the provided practice material, the folder "sw" contains a test assembly program, and the tools to compile it into the format used by our memories (Windows executable). The test program "programa.asm" provided does NOT include data risks, and it tests all the instructions for the basic exercise. To generate the files for the memory content from the assembler program, it is enough to double click on file "arqo_comp.bat".

File "registers.html" contains a table with the name of the registers used in assembler language, and the corresponding number in the bank register of the processor, from 0 to 31.