# Laboratory 6

# 6. Single-Cycle MIPS CPU Design (3): 16-bits version – One clock cycle per instruction

## 6.1. Objectives

Study, design, implement and test

- **Instruction Decode Unit for the 16-bit Single-Cycle MIPS CPU**
- **Main control Unit for the 16-bit Single-Cycle MIPS CPU**

Familiarize the students with

- Single-Cycle CPU design: Defining the instructions / writing the test program (MIPS assembly language, machine code)
- Xilinx® ISE WebPack
- Digilent Development Boards **(DDB)**
  - Digilent Basys Board – Reference Manual
  - Digilent Basys 2 Board – Reference Manual
  - Digilent Basys 3 Board – Reference Manual

## 6.2. Reduced Size MIPS Processor Description

(!) Read Lectures 3 and 4 in order to understand the works needed in this laboratory.

Remember that an instruction execution cycle (lecture 4) has the following phases:

- IF – Instruction Fetch
- ID/OF – Instruction Decode / Operand Fetch
- EX – Execute
- MEM – Memory
- WB – Write Back

Your own Single-Cycle MIPS 16 processor implementation (that you will continue in this laboratory and finish in the next ones) will be partitioned in 5 (five) components (new entities). These components will be declared and instantiated in the “test_env” project.

The utility of this implementation will be understood in the future laboratories, when you will implement the pipeline version of your 16-bit MIPS processor!

In this laboratory you will design, VHDL description, implement and test the Instruction Decode Unit of your own single cycle MIPS processor together with the main control unit for your processor.

The data-path of the processor (32-bit version), including the control unit and the necessary control signals, is presented in the next figure. In order to reduce the complexity of the data-path, the control signals were not explicitly connected, but rather they can be easily identified by their names.

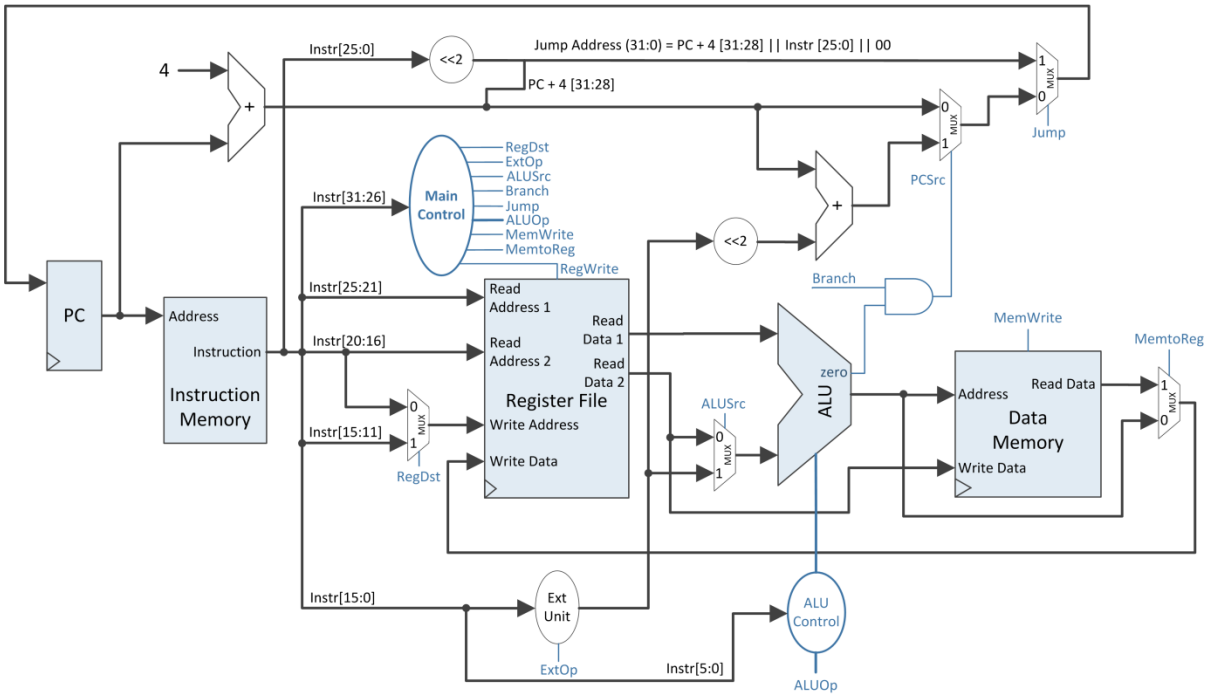


Figure 6-1: MIPS 32 Single-Cycle Data-Path + Control

As a reminder, the instruction formats for your MIPS 16 processor are presented below:

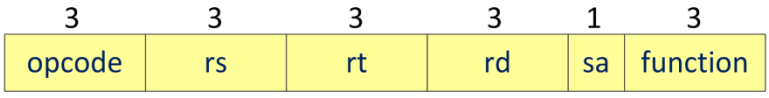


Figure 6-2: R-type Instruction format

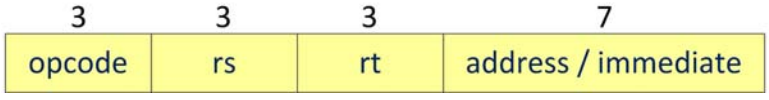


Figure 6-3: I-type Instruction format


Figure 6-4: J-type Instruction format

The ID (Instruction decode) unit consists in the following components:

- Register File
- Multiplexer
- Sign/Zero Extender

Please refer to laboratory 4 for the characteristics of these components for your Single-Cycle MIPS 16 processor. Remember (laboratory 3) that the Register File read operations are asynchronous, only the Register File Writes are synchronous.

The data-path of the Instruction Decode Unit is presented in Figure 6-5.

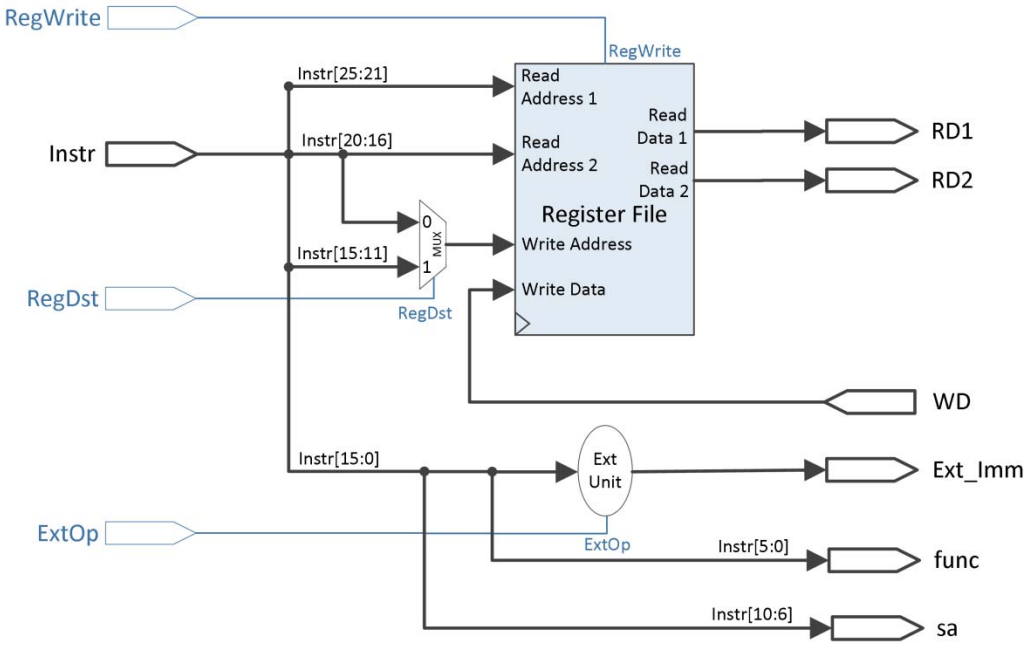


Figure 6-5: Instruction Decode Data-Path for MIPS 32

The ID unit provides, as output, the data fields (Read Data 1, Read Data 2 and Extended Immediate) used by the processor in the next execution phases. In addition, the function field is also provided to the ALU Control Unit and the shift amount is used as an additional input to the ALU for shift operations.

The inputs of the ID unit are:

- The clock signal (for the Register File Writes)
- The 32-bit instruction
- The 32-bit Write Data for the Register File
- Control Signals:
  - RegWrite – Write Enable signal for the Register File
  - RegDst – Selects the write address for the Register File
  - ExtOp – selects between Sign and Zero extension of the immediate field

The outputs of the ID unit are:

- Register from rs address: 32-bit Read Data 1
- Register from rt address: 32-bit Read Data 2
- 32-bit Extended Immediate
- 6-bit function field of the instruction
- 5-bit shift amount for R-type shift instructions

The meaning of the control signals:

- RegDst = 1 → the Write Address for the Register File is the rd field of the instruction (Instr[15:11])
- RegDst = 0 → the Write Address for the Register File is the rt field of the instruction (Instr[20:16])

- RegWrite = 1 → write the value provided by the Write Data Signal into the Write Address Register in the Register File.

- ExtOp = 0 → perform Zero Extension of the 16-bit immediate
- ExtOp = 1 → perform Sign Extension of the 16-bit immediate

The control signals of the Main Control Unit are presented in Figure 6-6. Please refer to Lecture 04 for the full description of the control signals for the MIPS processor.

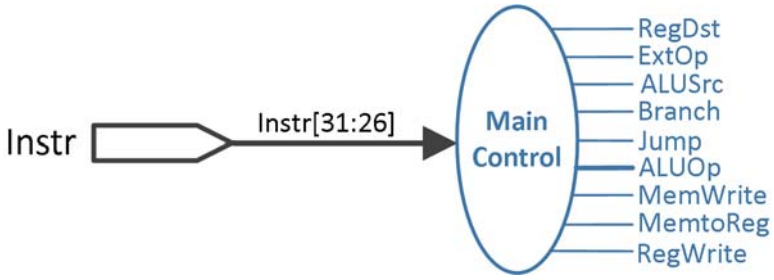


Figure 6-6: MIPS 32 Single Cycle Main Control Unit

The input of the Main Control Unit consists in the 6-bit opcode field of the instruction while the outputs are represented by the main data-path control signals (except for the ALUCtrl signal). There are 8 x 1-bit control signals and ALUOp which can be 2 or more bits wide depending on the 15 instructions that you have chosen. As it can be seen in the figure above the ALUOp line is thicker, meaning that it has more than 1-bit.

## 6.3. Laboratory Assignments

Read carefully and completely each activity before you begin!

Prerequisites:

- All the assignments from the laboratories 3 and 4 completed
- The instruction fetch unit implemented and tested on the Digilent Development Board.
- Xilinx project with “test_env” including the IF unit (Laboratory 5 Assignment 5.3.2)

Attention: If the homework from the previous laboratories is not completed, you will receive a 1 for this and all future laboratories until the homework is done without the possibility of any corrections to the mark!

### 6.3.1. Instruction Decode design

Taking into account the instruction decode data-path from Figure 6-5 design a new component (new entity) in the “test_env” project for your own single-cycle MIPS 16. All the data fields are 16-bits wide.

The ID entity will contain the hardware components described in Figure 6-5, that will not be implemented with other components (use behavioral VHDL description), except for the Register File (see laboratory 3).

Use one line of code for the extension unit (sign / zero extension), as in laboratory 2.

**Attention!** Be careful when transforming the data fields from Figure 6-5 (MIPS 32) into your own single-cycle MIPS 16 implementation.

### 6.3.2. Main Control Unit Design

The first part of this assignment is to identify the control signals for all your 15 instructions. See the table completed at assignment 4.3.3 from laboratory 4. In case you have not completed this table yet, see lecture 4 for reference (control signals table: add, lw, sw, beq, etc.) draw a table with all the control signals for each of the 15 instructions. In order to test the implementation on the Digilent Development board it is not mandatory to finish the whole control unit during the laboratory hours: 4-6 instructions are enough for testing, the rest is considered homework).

You can implement the Main Control Unit either as a new entity in the “test_env” project or as a simple decoder process in the “test_env” architecture. A supplementary

suggestion here is to declare the control signals individually, for a better reading of the VHDL code.

### 6.3.3. Testing of the Instruction Decode Unit and Main Control Unit

In the “test_env” entity declare and instantiate your Instruction Decode Unit (and also the Main Control Unit if you have implemented it as a separate entity).

Connect the Instruction Decode Unit together with your previously implemented Instruction Fetch Unit. The output of the IF unit – i.e. the 16-bit instruction will be the input to the ID unit.

Connect the necessary control signals generated by the Main Control Unit to the IF and ID units.

The next data-path components will be implemented in the next laboratories. For this reason, in order to test the writing in the Register File in this laboratory, use the adder from laboratory 3 to connect the outputs of the Instruction Decode Unit, RD1 (Read Data 1) and RD2 (Read Data 2) signals and generate the WD (Write Data) signal for the Register File (input to the ID unit).

Attention! At this point, RegWrite is asserted by the Main Control Unit, so the writing in the Register File is done only for those instructions in your program that require a writing of a result in the Register File.

Attention! For writing in the Register File it is necessary to control the writing mechanism from one of the buttons available on the board in the same manner as it was controlled for the PC register. The RegWrite Control Signal must be validated (use an AND logic gate) with one of the MPG outputs, i.e. use the same enable signal as for write validation in the PC register (from the IF Unit). Do not forget about the other MPG output that resets the PC register.

You are required to have both the IF and ID units together in the “test_env” project and the process/component instantiation for the Main Control Unit.

For connecting to the SSD use, all the signals present on the data-path, i.e. the outputs of the IF unit and the inputs and outputs of the ID unit:

Use 3 switches in order to control the display on the SSD (multiplexor):

- sw(7:5) = 000 → display the instruction on the SSD
- sw(7:5) = 001 → display the next sequential PC (PC + 1 output) on the SSD
- sw(7:5) = 010 → display the RD1 signal on the SSD
- sw(7:5) = 011 → display the RD2 signal on the SSD
- sw(7:5) = 100 → display the WD signal on the SSD
- …

On the LEDs, you will display the control signals from the Main Control Unit. You have 8 x 1-bit control signals and ALUOp.

For Basys 1 & 2 – use another switch to control the display on the LEDs:

- sw(0) = 0 → Display the 1-bit control signals on the LEDs. The order of the control signals is your own choice.
- sw(0) = 1 → Display the n-bit ALUOp signal on the LEDs (the rest of LEDs will have the value '0' for now)

For Basys 3 – display the control signals on the available 16-LEDs.

As in the previous laboratory, use two outputs of the MPG, one to reset the PC register and the other one to control the writing in the PC register and the RF. You will simulate the normal, sequential execution of the instructions.

Use "hard-coded" values for the branch target address and jump address inputs of the instruction fetch unit (as in the previous laboratory):

- You can use x"0000" for jump as an alternative reset mechanism for the PC register (jump to the first instruction in the ROM)
- Use an intermediate address for the branch target address. This address must be an address of an instruction in your program (within the range of your previously implemented program in the ROM – machine code)

## 6.4. References

[1] Computer Architecture Lectures 3 & 4 slides.
[2] D. A. Patterson, J. L. Hennessy, "Computer Organization and Design: The Hardware/Software Interface", 5th edition, ed. Morgan–Kaufmann, 2013.
[3] D. A. Patterson and J. L. Hennessy, "Computer Organization and Design: A Quantitative Approach", 5th edition, ed. Morgan-Kaufmann, 2011.
[4] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012
[5] MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02
[6] MIPS32® Architecture for Programmers Volume IV-a: The MIPS16e™ Application-Specific Extension to the MIPS32™ Architecture, Revision 2.62.

- Chapter 3: The MIPS16e™ Application-Specific Extension to the MIPS32® Architecture.