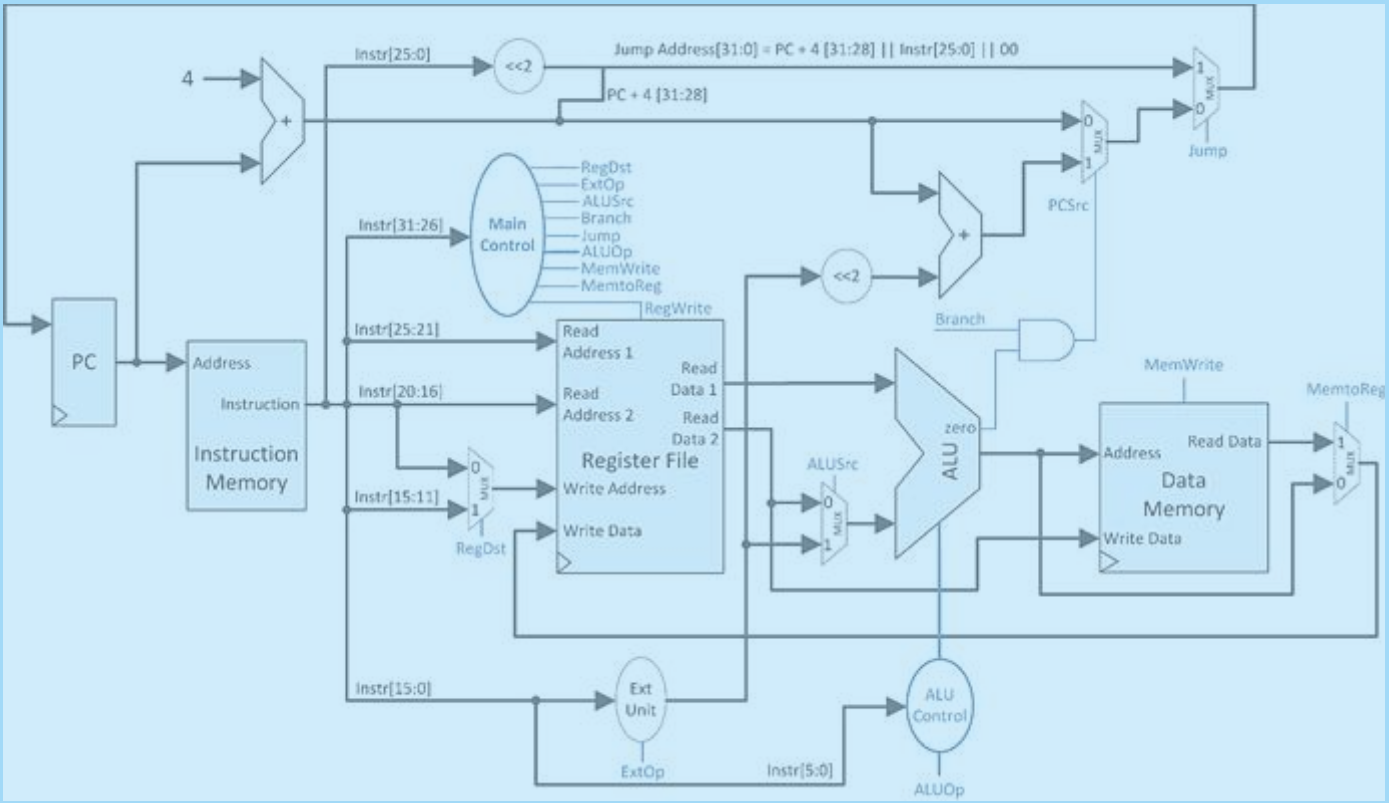
Mihai NEGRU **Florin ONIGA** **Sergiu NEDEVSCHI**

COMPUTER ARCHITECTURE

Laboratory Guide



UTPRESS

Cluj-Napoca, 2015

ISBN 978-606-737-123-9

Editura U.T.PRESS
Str.Observatorului nr. 34
C.P.42, O.P. 2, 400775 Cluj-Napoca
Tel.:0264-401.999 / Fax: 0264 - 430.408
e-mail: utpress@biblio.utcluj.ro
www.utcluj.ro/editura

Director: Ing. Călin D. Câmpean

Recenzia: Conf.dr.ing. Tiberiu Mariţa
Conf.dr.ing. Radu Dănescu

ISBN 978-606-737-123-9
Bun de tipar: 18.12.2015

Preface

This laboratory guide is intended for the 2nd year undergraduate students of the Automation and Computer Science Faculty, but can also be used by anyone who wants to grasp the basics of Computer Architecture. This laboratory guide is structured in 12 laboratory tutorials and 7 appendices. Each laboratory covers a part of the MIPS processor design process and the appendices provide more details and implementation examples for different hardware architectures. The reader is encouraged to go through the laboratories in the presented order, because each laboratory contains elements studied, designed and implemented in the previous ones.

This is the first printed edition of the Computer Architecture Laboratory Guide and consists in the resulting efforts of the authors over the past years. Special thanks goes to Professor Gheorghe Farkas who was our mentor in the field of Computer Architecture and had a great contribution in the teaching of this subject for more than 10 years.

The students from the Technical University of Cluj-Napoca, Computer Science field of study, will use this laboratory guide for fulfilling their knowledge about Computer Architecture. The laboratory guide is very tightly connected to the Computer Architecture lectures, so the attendance at the lectures is highly encouraged for a better understanding of the treated subjects. Each chapter of this laboratory guide starts with a short presentation of the necessary theoretical concepts, followed by practical design and implementation approaches. The laboratory assignments that a student must do are found at the end of each laboratory. The students are encouraged to carefully read all the laboratory material before attending the laboratory class, in order to be familiar with the tasks that must be designed and implemented throughout the laboratory. The first laboratories are meant for introducing the students to the VHDL programming language and the FPGA boards used for development and testing. Then, the difficulty of the laboratories increases gradually until the students design and implement a complete single-cycle MIPS processor and then transform this processor into a pipeline one. In the last laboratory works, the students are required to add input / output functionalities to the processor by means of a serial communication interface.

The authors wish you a pleasant reading!

Table of Contents

CA Laboratory general objectives

The laboratory exercises and homework are mandatory components of the Computer Architecture course. The main objective of the laboratory exercises is the developing of synthesizable VHDL models of simple MIPS CPUs using the Xilinx ISE tools and Digilent Development Boards (DDB).

The main laboratory themes are:

- Design with Xilinx ISE tools and Digilent Development Boards.
- Design synthesizable VHDL hardware components implemented and tested on the Digilent Development Boards.
- Understand the architecture of a single-cycle / multi-cycle / pipeline MIPS processor.
- VHDL design of MIPS single-cycle / multi-cycle / pipeline CPUs implemented and tested on Digilent Development Boards.
- Input / output serial communication for the single-cycle / multi-cycle / pipeline processors.

The associated homework helps to prepare the laboratory exercises and improve the specific problem solving ability of the students.

Some extracts of the recommended reading assignments are included in the laboratory materials, the original documents and VHDL examples are available on the web site.

Laboratory 1

1. Introduction to the Software/Hardware development environment for VHDL based designs.

1.1 Objectives

Familiarize the students with

- Xilinx ISE WebPack CAD tools – ***ISE Quick Start Tutorial***
- Xilinx® Synthesis Technology (XST) XST User Guide
- Xilinx Spartan 3E FPGA family
- Digilent Development Boards **(DDB)**
  - Digilent Basys Board – Reference Manual
  - Digilent Basys 2 Board – Reference Manual

1.2 Necessary resources (the kits are available and installed on the workstations from the laboratory)

Digilent Adept Software: download page

Digilent Basys Board:

- Reference Manual
- Schematic

Xilinx ISE WebPACK is a part of the Xilinx Design Suite, ISE Design Suite – 14.7 Full Product Installation

Xilinx ISE Software manual:

- XST User Guide

Online Help for VHDL programming

http://vhdl.renerta.com/

1.3 Basic Components

1.3.1. Logic Gates

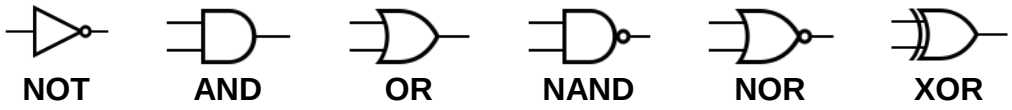


Figure 1.1: Logic Gates Diagrams

A	NOT
0	1
1	0

A	B	AND	OR	NAND	NOR	XOR
0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	1	0	0	0

Table 1.1: Logic Gates Truth Tables

1.3.2. Latches

A latch is an electronic circuit which has two stable states and thereby can store one bit of information. XST can recognize latches with asynchronous set/reset control signals. Latches can be described in VHDL by using: processes or concurrent statement assignment. **XST does not support wait statements (VHDL) for latch descriptions.**

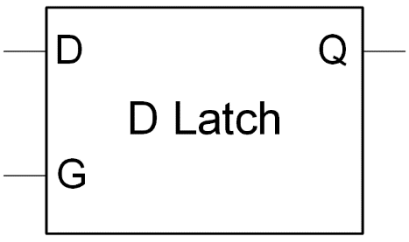


Figure 1.2: Latch with Positive Gate

IO Pins	Description
D	Data Input
G	Positive Gate
Q	Data Output

Table 1.2: Latch with Positive Gate Pin Description

1.3.3. Flip-Flops

A flip-flop is an electronic circuit that has two stable states and is capable of serving as one bit of memory. A flip-flop is usually controlled by one or two control signals and/or a gate or clock signal. XST recognizes flip-flops with the following control signals: asynchronous Set/Reset, synchronous Set/Reset or clock enable.

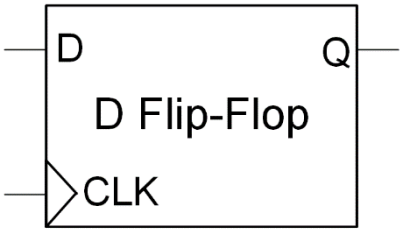


Figure 1.3: Flip-flop with Positive Edge Clock

IO Pins	Description
D	Data Input
CLK	Positive Edge Clock
Q	Data Output

Table 1.3: Flip-Flop with Positive-Edge Clock Pin Descriptions

When using VHDL for a positive-edge clock, instead of using:

```
if (C'event and C='1') then
```

you can also use:

```
if rising_edge(C) then
```

1.3.4. Multiplexers

A multiplexer or mux is a device that performs multiplexing; it selects one of many analog or digital input signals and outputs that into a single line. A multiplexer of 2^n inputs has n select bits, which are used to select which input line to send to the output. XST supports different description styles for multiplexers (MUXs), such as If-Then-Else or Case.

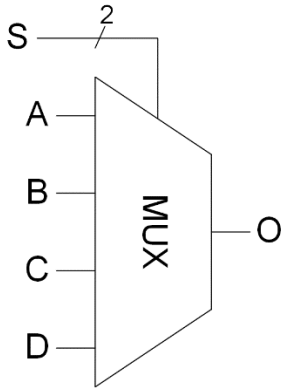


Figure 1.4: 4-to-1 1-Bit MUX

IO Pins	Description
A, B, C, D	Data Inputs
S	Mux Selector
O	Data Output

Table 1.4: 4-to-1 1-Bit MUX Pin Descriptions

1.3.5. Decoders

In digital electronics a decoder is a multiple-input, multiple-output logic circuit that converts coded inputs into coded outputs, where the input and output codes are different. E.g.: n-to-2^n decoders, BCD decoders.

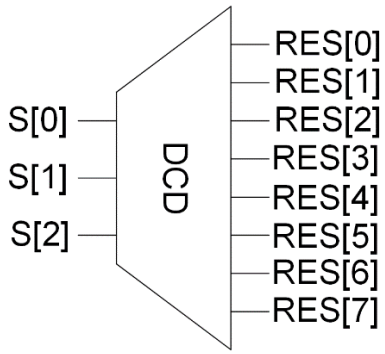


Figure 1.5: 3-to-8 Decoder

IO Pins	Description
S	Selector
RES	Data Output

Table 1.5: 3-of-8 Decoder Pin Descriptions

1.3.6. Counters

In digital logic and computing, a counter is a device which counts the number of times a particular event or process has occurred, often in relationship to a clock signal. XST recognizes counters with the following control signals: asynchronous Set/Reset, synchronous Set/Reset, asynchronous/synchronous Load (signal or constant or both), clock enable, modes (up, down, up/down) or a mixture of all.

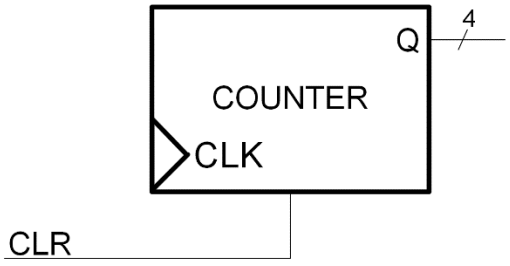


Figure 1.6: 4-Bit Up Counter with Asynchronous Reset

IO Pins	Description
CLK	Positive Edge Clock
CLR	Asynchronous Reset (Active High)
Q	Data Output

Table 1.6: 4-Bit Up Counter with Asynchronous Reset Pin Descriptions

1.4 Laboratory Assignments

Note: If necessary, you can consult the online help for VHDL indicated in the previous section.

1.4.1. Implement a simple VHDL design using Xilinx's ISE 14.7 and the DDB by carefully and completely covering the tutorial described in Appendix 1. You are also encouraged to read Appendix 2, which covers the description of basic digital components.

1.4.2. Add an 8-bit up-counter to your "*test_env*" design, by describing (in one process) the behavior of the counter in the "*test_env*" architecture. Try to control the counting process from a digital button present on the board.

Start by declaring an 8-bit signal (STD_LOGIC_VECTOR) in the architecture, before the begin statement.

If necessary (until you regain your full capacity in the VHDL programming language), use the *Language Templates* (Appendix 1) for extracting the behavioral description of the counter.

Use one of the buttons from the entity's ports in order to control the counting process, as an enable or count-up signal.

Display the counter values on the 8 LEDs available on the board.

Follow the steps in Appendix 1 in order to generate the bit file and re-program the Basys board. Control the counter from the button.

…Is there any problem?

1.4.3. Synchronized (1 clock period) mono pulse generator (MPG)

At this point, you have to work in the same "test_env" project.

In the future, you will need step-by-step control of sequential circuits, to trace and test the required data and control flow of your designs.
The necessary circuit that generates an ENABLE signal once per button push is given in the next figure.

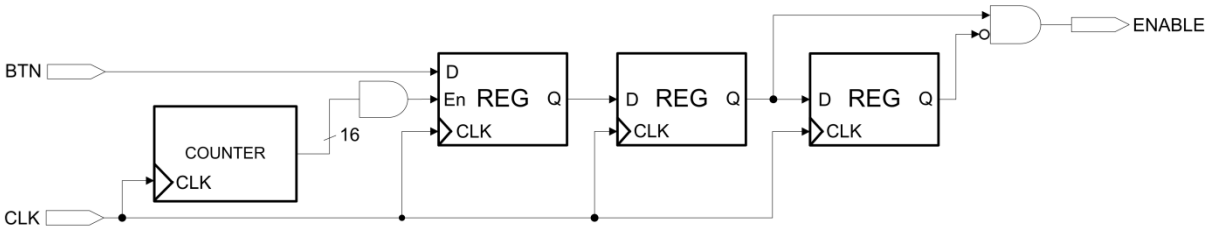


Figure 1.7: Synchronized (1 clock period) mono pulse generator

The role of the first register together with the 16-bit counter is to provide a delay necessary to de-bounce the buttons (physically worn out and / or low quality buttons). According to the "degradation" state of the button, you may have to increase the size of this 16-bit counter in order to increase the sampling interval for the button signal.

The mono pulse generator (MPG) will be implemented in a new entity / new file (menu **Project\New Source**) and will be used in the "*test_env*" by declaring as a component in the section for declaring signals and by instantiating this component with *port map* in the architecture body after begin.

The hardware components that implement the MPG (counter, registers, and logic gates) will be implemented using the behavioral description: by declaring the necessary signals and by describing the functionality using processes and concurrent assignments in the MPG architecture.

Work to do

a. Draw a timing diagram for the above circuit (paper and pencil / blackboard).
b. The BTN input is a signal from one of the DDB buttons (BTN0), CLK is the clock signal of the DDB (50 MHz, you can verify on the board that the jumper is not set for 25 MHz or 100 MHz).
c. Write and check the VHDL code for this circuit.
d. Include the MPG component in the *test_env.*
e. Use the ENABLE signal as count up for the previously implemented 8-bit counter, from section 3.2. You need to add the condition that ENABLE equals '1' where you test the count up condition for the 8-bit counter.

Synthesize your design and do not forget about **View RTL Schematic…**

Load your new design on the Basys board.

1.4.4. Create a new project, for example *test_new*, using the same ports as for the first project. You have to go again over the steps described in Appendix 1, without adding anything to the architecture *test_new*. Now implement the following circuit in the *test_new* architecture.

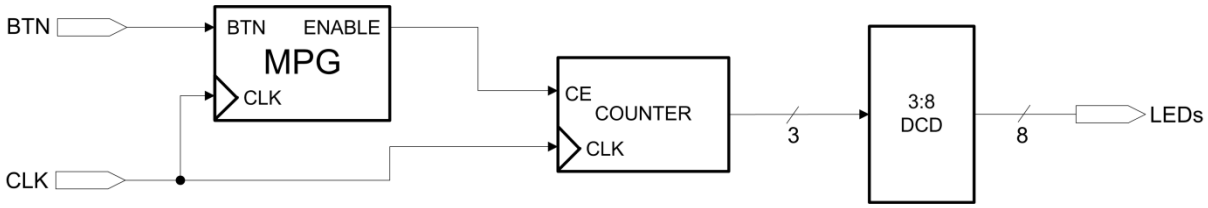


Figure 1.8: Problem 4.4 Schematic

You need to add the MPG source file to the new project (menu **Project\Add Copy of Source**). Import the MPG with *component / port map* in the *test_new* architecture and write the code (only processes) for the rest of the digital components, without any additional entities. Add a 3-bit counter and a 3-to-8 decoder, using only signals declared in the *test_new* architecture and concurrent processes / signal assignments.

Do not forget about **View RTL Schematic…**

Load the design on the Basys board.

1.4.5. Redesign the mono pulse generator in order to create a MPG for all the 4 buttons on the DDB.

Show all your designs to the TA

Homework

a. Finish all the laboratory assignments.
b. Re-read the Laboratory regulations, the tutorial from Appendix 1 – starting with laboratory 2 these concepts are considered learned. Be attentive about the aspects presented in the tutorial that were not, yet, relevant for this first designs. They will be important in the future
c. (this homework is considered implicit for the next labs) Read the material for the next laboratory (it is available on the web site).

1.5 References

[1] ISE Quick Start Tutorial (www.xilinx.com)
[2] Xilinx® Synthesis Technology (XST) User Guide
[3] Digilent Basys Board – Reference Manual
[4] Digilent Basys 2 Board – Reference Manual

Laboratory 2

2. Extending your design: Seven Segment display

2.1 Objectives

Design, implement and test

- **The Seven Segment Display**
- **A simple Arithmetic Logic Unit (ALU)**

Deeper Knowledge of:

- Xilinx® ISE Webpack CAD tools
- Xilinx® Synthesis Technology (XST) **XST User Guide**
  - **Chapter 2: XST HDL Coding Techniques**
  - **Chapter 6: XST VHDL Language Support**
- Digilent Development Boards (DDB)
  - **Digilent Basys Board – Reference Manual**

2.2 4-Digit Seven Segment Display

The Basys board comes equipped with a 4-digit Seven Segment Display (SSD). This interface uses seven LEDs for each digit; each digit is enabled by an anode signal. All the connections (7 common cathode and 4 distinct anode signals) to the SSD interface are active low. The cathode signals control the LEDs of the digit to be displayed, which is selected by the active anode signal.

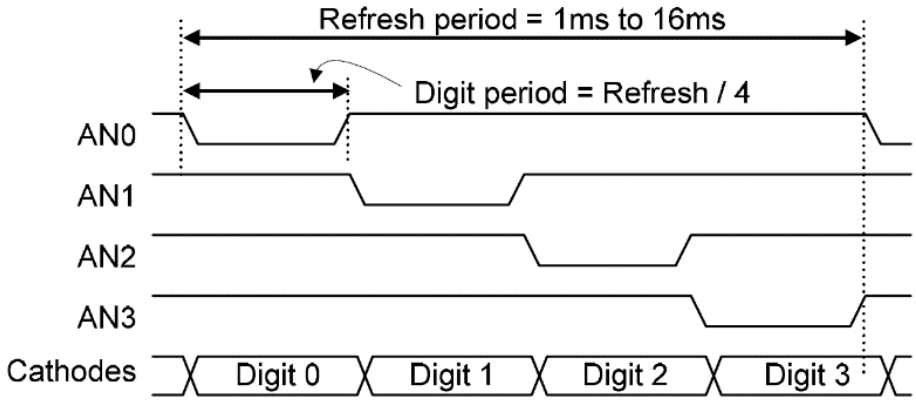


Figure 2.1: SSD Timing Diagram [3], [4]

In order to display 4 different digits on the SSD, you have to implement a circuit that sends the digits to the cathode signals of the SSD according to the timing diagram

presented in Figure 2.1. The refresh period is chosen in order to accommodate the human eye (at least 60 Hz refresh rate) with the cycling display of the digits (actually, only one digit is displayed at one time). Read the section about seven segment display methodology in the Basys boards Reference Manual [3], [4].

The figure below describes a possible implementation of the seven-segment display circuit. The inputs are 4 4-bit signals and the clock signal; the outputs are represented by the anode (an) and cathode (cat) signals (active low).

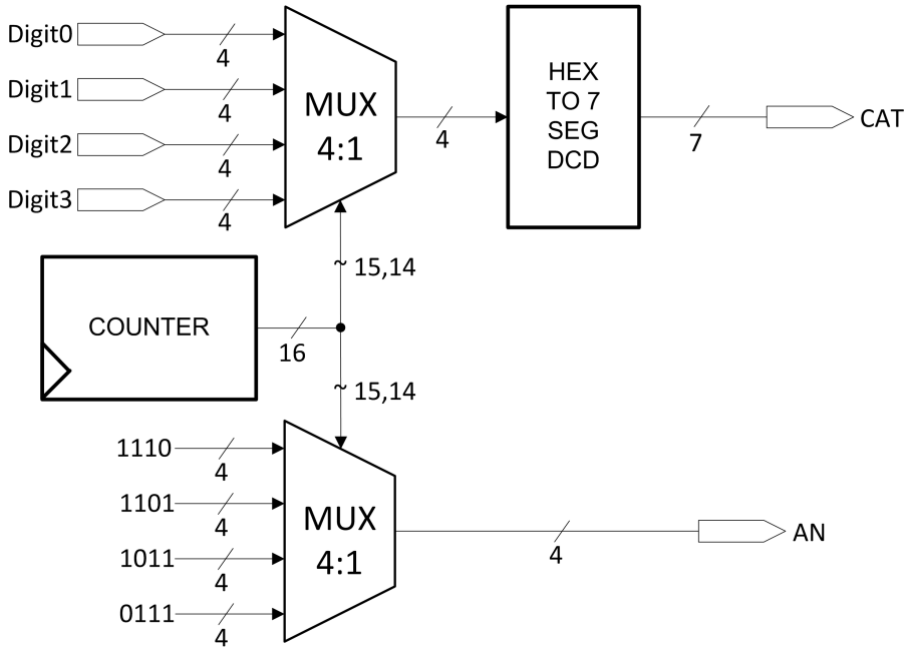


Figure 2.2: Schematic of the SSD circuit

2.3 General laboratory design for the Basys boards

Once you will complete assignment 2.5.1, all your future designs for this laboratory will resemble the next figure. This design will provide the necessary interfaces with the Basys boards. All of your circuit descriptions (behavioral or component instantiation) will be placed inside the “cloud”.

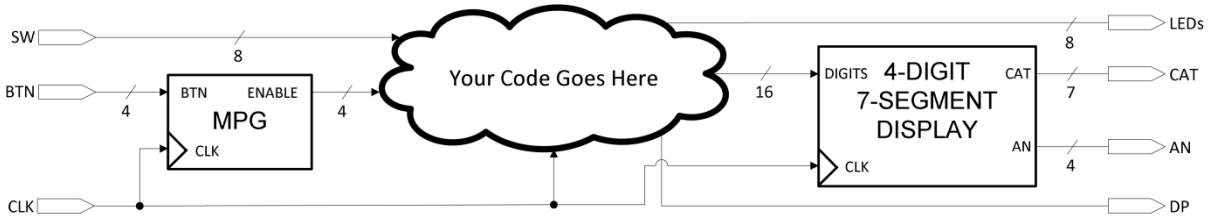


Figure 2.3: Laboratory Design

2.4 Simple ALU operations

Each of the following simple ALU operations can be implemented in VHDL in one line of code. You can also use processes for your implementations.

2.4.1 Adders

An adder is a digital circuit that performs addition of numbers. In modern computers, adders reside in the arithmetic logic unit (ALU) where other operations are performed. The equations for 1-bit full adder are given below:

$$Sum = A\ xor\ B\ xor\ Cin$$
$$Cout = (A\ and\ B) or\ (A\ and\ Cin)\ or\ (B\ and\ Cin)$$

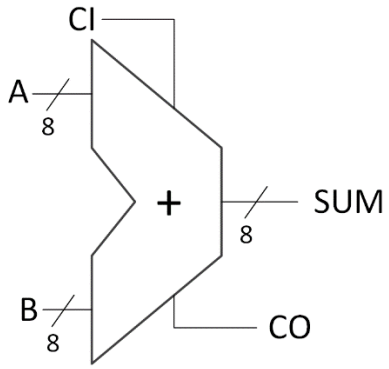


Figure 2.4: 8-Bit Adder with Carry in and Carry out

IO Pins	Description
A, B	Add Operands
CI	Carry In
CO	Carry Out
SUM	Add Result

Table 2.1: 8-Bit Adder with Carry In and Carry Out Pin Descriptions

Implementation in VHDL – Use simple assignment:

```
SUM <= A + B;
```

If the input signals are extended by at least one bit, the Xilinx Synthesizer takes care of the carry input and output signals.

2.4.2 Subtractors

A subtractor is a digital circuit that performs subtraction. In 2's complement subtraction is the same as adding the negative of the number and setting the Carry in to 1.

$$A - B = A + \overline{B} + 1$$

Implementation in VHDL – Use simple assignment:

```
DIFF <= A – B;
```

2.4.3 Shifters

A shifter is a digital circuit that can shift a word of data by a specified number of bits. There are two kinds of shifters:

- logical shifters – value shifted in is always 0
- arithmetic shifters – on right shifts sign extend

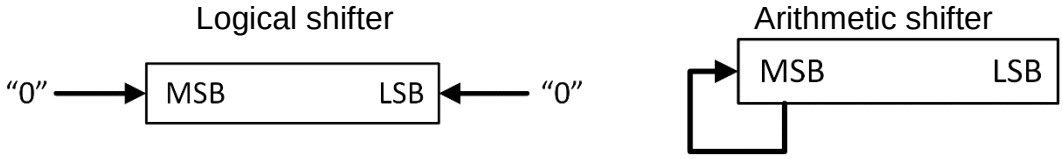


Figure 2.5: Logical and Arithmetic Shift Operations

Xilinx defines a shifter as a combinatorial circuit with two inputs and one output.

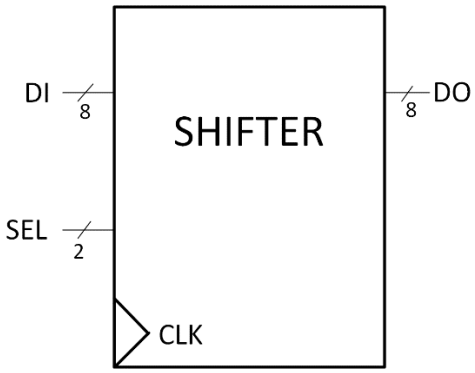


Figure 2.6: Shifter

IO Pins	Description
DI	Data Input
SEL	Shift Distance Selector
DO	Data Output

Table 2.2: Shifter Pin Description

Implementation in VHDL – For simple shifters when the shift amount is fixed use signal concatenation. Example << 2:

```
DO(7 downto 0) <= DI(5 downto 0) & "00";
```

A possible implementation for a combinational shifter with a variable shift amount (SEL) is presented in appendix 2.

2.4.4 Zero Detector

The n-bit zero detector is an n-bit input, 1-bit output NOR circuit, used in Arithmetic/Logic units especially for condition detection for Branch on Equal instructions.

Implementation in VHDL: use a simple line of code (mux 2:1 type):

Zero <= ‘1’ when DI=0 else ‘0’;

2.4.5 Sign/Zero Extender

The Sign and Zero extender for MIPS CPU are used in arithmetical/logical operations in which the 16-bit immediate value is implied. The extension is necessary because the ALU works on 32-bit operands, and the other involved operand has 32 bits.

Example: ADDI (ADD Immediate) instruction, memory access, branch address computation, logical operations with immediate, etc.

Implementation in VHDL: Use concatenation with zeros for the Zero Extension circuit and with the Sign Bit for the Sign Extension circuit.

2.4.6 Comparators

A comparator is a digital circuit that compares two numbers in binary form and generates a one or a zero at its output depending on whether they are the same or not. A comparator can be simulated by subtracting the two values (A & B) in question and checking if the result is zero.

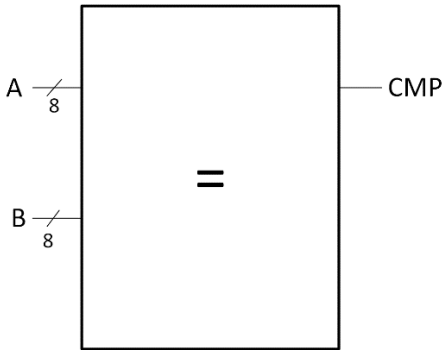


Figure 2.7: Unsigned 8-Bit Equal Comparator

IO Pins	Description
A, B	Comparison Operands
CMP	Comparison Result

Table 2.3: Unsigned 8-Bit Greater or Equal Comparator Pin Description

Implementation in VHDL – ALU performs subtraction between the two operands and the Zero Detection circuit is used over the ALU result. Another approach is to use the implementation provided in the Zero Detector:

CMP <= ‘1’ when A = B else ’0’;

2.5 Laboratory Assignments

2.5.1 4-Digit Seven Segment Display

You have to work in the previous design from laboratory 1 (*test_env* project). Design and implement a new component (a separate entity) for the 4-digit seven-segment display interface (see Figure 2.2 for details). **Use only processes to implement the counter and the two multiplexers, and use the Language Templates to implement the “HEX TO 7 SEG DCD” (the component that transforms a 4-bit hexadecimal digit to its corresponding seven-segment LEDs representation).**

Declare the component in the *test_env* project and instantiate it in the *test_env* architecture. Connect a 16-bit counter to the 4 input data signals of the seven-segment display. The counter is incremented when a button is pressed (use the MPG component to validate the increment of the counter). Your final design must resemble Figure 2.3.

2.5.2 Simple Arithmetic Logic Unit Circuit

Using the previous design (containing the MPG and the SSD), implement a simple ALU circuit with the following functions: ADD, SUB, SHIFT LEFT 2, and SHIFT RIGHT 2.

The results (16-bit) of the ALU operations are displayed on the SSD interface (Digit3...Digit0). Use a 2-bit counter (controlled by the MPG) to select the desired ALU operation.

The input operands for the ALU are the switches of the Basys board. The ALU design is presented in the figure below (describe it in the architecture of the *test_env* entity, no other components are required, use only internal signals, processes, and concurrent assignments).

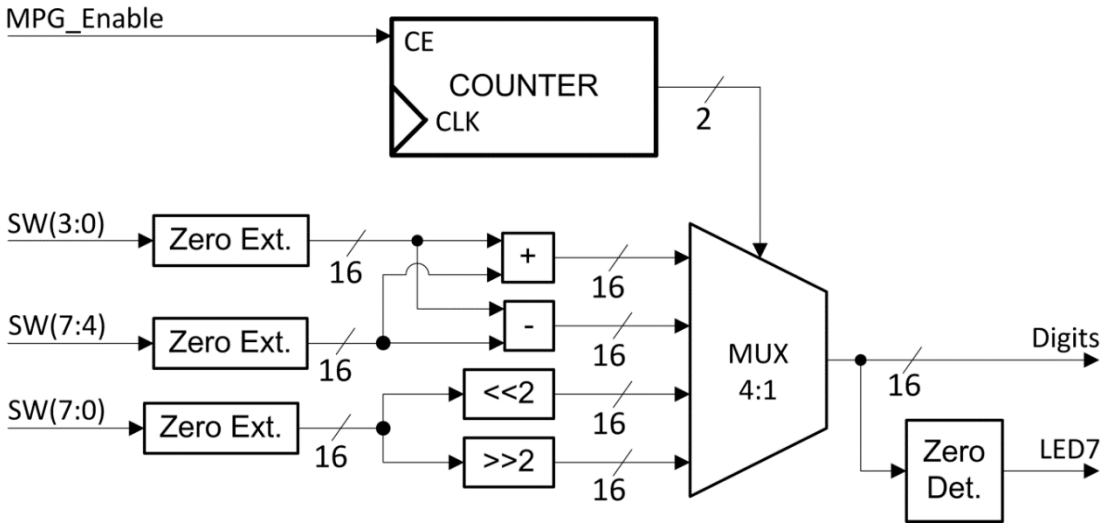


Figure 2.8: Simple ALU design.

2.5.3 Homework: Combinational Shifter with Variable Shift Amount

Read appendix 2 at the end of this laboratory guide. Draw (paper and pencil) the diagram of this shifter (described in the example from the appendix). Implement the circuit on the Basys board.

2.6 References

[1] XST User Guide, Chapter 2: XST HDL Coding Techniques
[2] XST User Guide, Chapter 6: XST VHDL Language Support
[3] Digilent Basys Board – Reference Manual
[4] Digilent Basys 2 Board – Reference Manual

Laboratory 3

3. Memory Components

3.1 Objectives

Design, implement and test

- **Register File**
- **Read only Memories – ROMs**
- **Random Access Memories – RAMs**

Familiarize the students with

- Xilinx® ISE Webpack
- Xilinx® Synthesis Technology (XST) **XST User Guide**
  - **Chapter 2: XST HDL Coding Techniques**
  - **Chapter 6: XST VHDL Language Support**
- Digilent Development Boards (DDB)
  - **Digilent Basys Board – Reference Manual**
  - **Digilent Basys 2 Board – Reference Manual**

3.2 Theoretical Background

3.2.1. Register File

The Register file is the central storage of a Microprocessor.

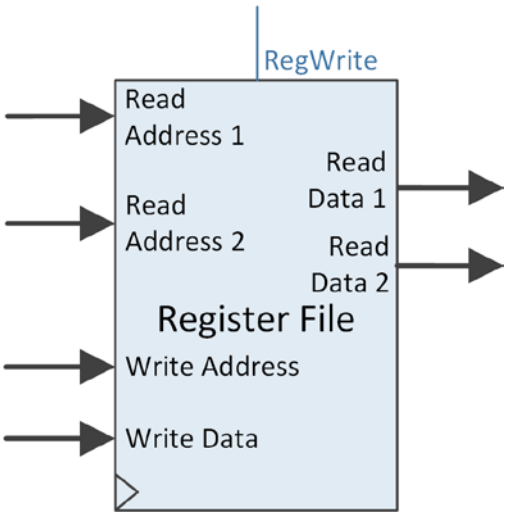


Figure 3.1: A Register File with 2 read ports and 1 write port

Most CPU operations involve using or modifying data stored in the register file. Since the register file runs at the full speed of the processor, it must be small and fast. The real register file is usually implemented as a small, fast **SRAM** memory with multiple accesses.

A register file (specific for MIPS) has two read addresses and one write address. The registers corresponding to the locations indicated by the two read addresses (Read Address 1 & Read Address 2) are delivered at the two output ports (Read Data 1 & Read Data 2). The data provided at the write data input port is written in the register indicated by the write address (Write Address, when the Write control signal (RegWrite) is asserted. The read operations are asynchronous, while the write operation is synchronous. Therefore, the register file supports 2 reads and one write in each clock cycle.

Appendix 3 presents a possible register file VHDL implementation.

3.2.2. ROMs and RAMs

Read-only memory (ROM) is a class of storage media used in computers and other electronic devices; they allow only read operations in usual operation mode. Random-access memories (RAM) are a form of computer data storage implemented as integrated circuits and allows the stored data to be accessed in any order; both read and write operations are permitted. These two memory types are essential for any microprocessor.

An FPGA device comes equipped with a certain amount of BRAM (Block RAM). The BRAM can be configured as either a ROM or a RAM. Depending on how you write the VHDL code, XST either can infer your RAM design as a distributed memory or directly mapped onto a Block RAM block. Distributed memories are built with registers, while Block RAM memories are mapped to available BRAM cell. Distributed RAMs occupy more space inside the FPGA and usually decrease the clock cycle rate, while BRAMs provide more space for auxiliary logic inside the FPGA. The type of inferred RAM depends on its description:

- RAM descriptions with an asynchronous read generate a distributed RAM macro.
- RAM descriptions with a synchronous read generate a block RAM macro. See appendix 4 for an implementation example.

XST covers the following RAM characteristics:

- Synchronous write
- Write enable
- RAM enable
- Asynchronous or synchronous read
- Reset of the data output latches
- Data output reset

- Single, dual or multiple-port read
- Single-port/Dual-port write
- Parity bits
- Block RAM with Byte-Wide Write Enable
- Simple dual-port BRAM

There are three possible modes of implementing a synchronous RAM [1], [2]: write-first, read-first and no change. These modes are reflected in the behavioral description of the RAM (VHDL code) regarding the read/write priority or order of operation. A possible "no change" implementation is presented in appendix 4.

To Do

- Use the language templates: VHDL → Synthesis Constructs → Coding Examples → RAM and see the differences in behavioral description between a distributed RAM and a Block RAM.
- Use the language templates: VHDL → Synthesis Constructs → Coding Examples → RAM → Block RAM → Single port in order to compare the read-first and write-first implementations.

3.2.3. Declaring an array in VHDL

An example of declaration and initialization for an array used in ROMs, RAMs and Register Files is presented below. First, we describe an array type having N locations of M bits each:

```
type <arr_type> is array (0 to N-1) of std_logic_vector(M-1 downto 0);
```

Next, we declare a signal of the same type as the previously declared one:

```
signal r_name: <arr_type>;
```

If one implements a ROM then the signal must be initialized. Initializations for the RAMs and Register File are also possible.

```
signal r_name: <arr_type> := (
"00…0", -- M bits, use hexadecimal representation when possible
"00…1", --
others => "00…0"
);
```

3.3 Laboratory Assignments

At this time, it is mandatory to have a functional "*test_env*" project that resembles with the description from the previous laboratory (laboratory 2: section 3 → figure 3). Your design must contain the 4-bit Mono Pulse Generator (MPG) and 4-digit Seven

Segment Display (SSD) components instantiated in the top-level entity of your "*test_env*" project; here you will write the code for this laboratory.

Use **View RTL Schematic** after each successful synthesis of your project. You can also view the implemented components in the **Design Summary** (see laboratory 1).

3.3.1. ROM Implementation

Include a 256 x 16 bits ROM memory in the test_env project (do not declare a new entity). Initialize the ROM with some arbitrarily chosen values (see 2.3). Use an 8-bit counter to generate the addresses for the ROM. The counter is controlled by the MPG component. The contents of the ROM is displayed on the Seven Segment Display. The behavior of the ROM is asynchronous – use only one line of code. The design is depicted in the figure below:

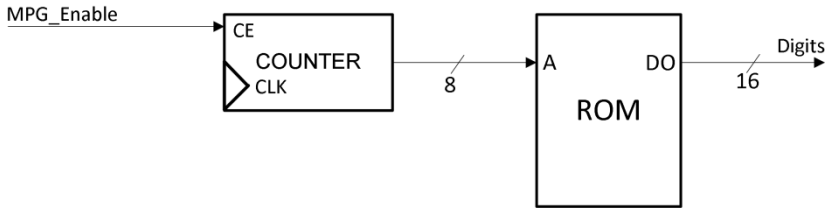


Figure 3.2: Simple ROM design

Test on the Basys board!

3.3.2. Register File Implementation

Do not delete the previously written code! Use comments if necessary!

Design and implement a 16x16-bit Register File on the Basys board (use a new component for the register file, in the "*test_env*" project). Initialize the Register File with some values. The design is presented in Figure 3.3.

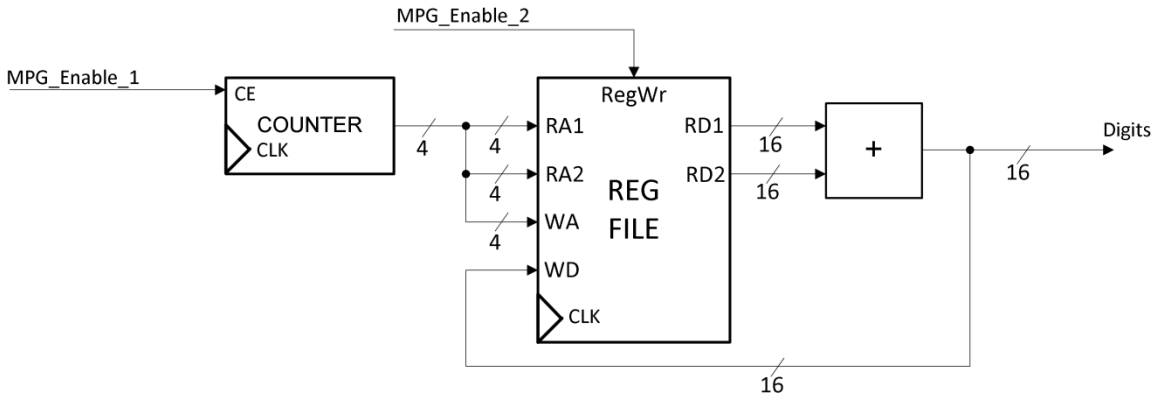


Figure 3.3: Simple Register File Design

Use a counter to generate the read and write address of the Register File. The counter is controlled by a MPG component. The outputs of the Register File are added together; the result of addition is displayed on the Seven Segment Display and written back to the Register File. You have to use another output of the MPG component to enable the write signal of the Register File (RegWr). The design should resemble a multiply by 2 circuit (a + a = 2a).

Add a synchronous reset mechanism for the counter that generates the Register File's address such that after going through a few of the Register File's locations you can reset the address counter (return to address 0) and check that the written results in Register File are correct.

Test on the Basys board!

Add some auxiliary components/elements to the design in order to use only one button for the counter increment and RegWr signal. The new circuit should work in the same manner (the results should be the same as in the previous case) and you must see the correct results of addition on the seven-segment display.

You can add the necessary circuit in order to display all the intermediate signal values on the SSD (Hint: use switches).

Test on the Basys board!

3.3.3. RAM Design

Replace the Register File previously designed with a RAM memory. Use a shift left 2 operation instead of the addition (Figure 3.3, shift is implemented with concatenation). Use only one address for the RAM. Use a write-first mode of implementation.

3.4 References

[1] XST User Guide, Chapter 2: XST HDL Coding Techniques
[2] XST User Guide, Chapter 6: XST VHDL Language Support
[3] XAPP463 (v2.0) March 1, 2005 Using Block RAM in Spartan-3 Generation FPGAs
[4] Digilent Basys Board – Reference Manual
[5] Digilent Basys 2 Board – Reference Manual

Laboratory 4

4. Single-Cycle MIPS CPU Design: 16-bits version – One clock cycle per instruction

4.1 Objectives

Study, design, implement and test

- **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**

4.2 Reduced Size MIPS Processor Description

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

In this laboratory, you will design and start the implementation of your own single cycle MIPS processor – MIPS 16.

The microprocessor will be a simpler version of the MIPS 32 microarchitecture described during the lectures. What does simpler mean? The instruction set will be smaller (fewer instructions to implement); the width of the instructions and data fields will be of 16-bits. Implicitly, the number of registers used in the register file will be smaller; the instruction and data memories will be smaller. The rest of the principles described during the lectures are the same (data-path and control).

The principal motive for implementing MIPS 16 is the reduced methodologies for data display (8 LEDs and 4-digit Seven Segment Display). In this manner, one avoids using other multiplexing mechanisms for signal display purposes (32 bits); and the on-chip debugging process is simplified (testing your program on the FPGA board).

The dimension/width of both instructions and data will be of 16-bits. The 3 instruction formats are given below. Compare this format with the 32-bit instruction format from the lectures. Observe the differences/limitations.

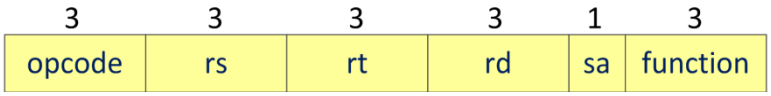


Figure 4.1: R-type Instruction format

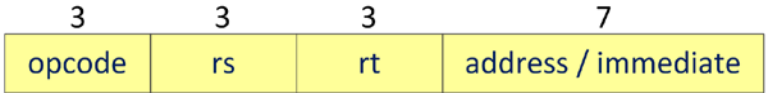


Figure 4.2: I-type Instruction format

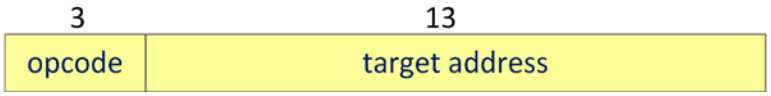


Figure 4.3: J-type Instruction format

These instruction formats obey the formats presented in the MIPS32 ISA, except the width of each field.

The opcode is encoded on 3-bits. For I-type and J-type instructions, the opcode uniquely encodes the instruction to be executed. In the case of R-type instructions, in accordance to the MIPS standard, the opcode is 0 and the function field identifies the ALU operation for each instruction. The function field is encoded on 3-bits. This means that your processor can implement at most 15 instructions:

- 8 R-type Instructions
- 7 I-type Instructions and J-type instructions.

The table below presents the minimum number of instructions, of each type, that will be implemented on the MIPS 16 processor. On the doted positions, you will choose or define new instructions for your MIPS processor (depending on the program that you will implement).

R-type Instructions	Addition	add
	Subtraction	sub
	Shift Left Logical (with shift amount – sa)	sll
	Shift Right Logical (with shift amount – sa)	srl
	Logical AND	and
	Logical OR	or
	….	…
	….	…
I-type Instructions	Add Immediate	addi
	Load Word	lw
	Store Word	sw
	Branch on Equal	beq
	….	…
	….	…
J-type Instruction	Jump	j

Table 4.1: Instructions for MIPS16

The description of each MIPS 16 data-path component characteristics is given below. (!) These characteristics are not only valid for this laboratory, but also for the future laboratory works.

Program Counter characteristics:

- 16-bit edge triggered D flip-flop

Instruction Memory (ROM) characteristics:

- One input bus: Instruction Address
- One output bus: Instruction Data
- Memory word is 16-bit (selected by instruction address)
- No control signals

Register File characteristics:

- Two read addresses and one write address
- Eight 16-bit registers (rs, rt, rd encoded on 3-bits)
- Two 16-bit data outputs: Read data 1 and Read data 2
- One 16-bit data input: Write Data
- Multiple accesses: 2 asynchronous reads and 1 synchronous (edge triggered) write. During read operation, the register file behaves as a combinational logic block.
- One control signal RegWrite. When RegWrite is asserted the value on the Write Data line is written in the register indicated by the write address line

Data Memory (RAM) characteristics:

- One 16-bit input address bus: Address
- One 16-bit input data bus: Write Data
- One 16-bit output data bus: Read Data
- One control Signal: MemWrite

Extension Unit characteristics:

- ExtOp = 1 → Sign Extension
- ExtOp = 0 → Zero Extension

ALU characteristics:

- ALU performs arithmetical / logical operations
- (!) You need to identify all the operations that the ALU needs to perform after completing the definition of the instructions from Table 4.1. You are encouraged to choose two more R-type instructions and two more I-type Instructions.
- You need to identify how many control bits are necessary to encode the ALU operations (ALUCtrl).

4.3 Laboratory Assignments

Read carefully and completely each activity before you begin!

4.3.1. Define the instructions for MIPS 16 – Paper and Pencil

Starting from the instruction formats presented in the previous section write (Paper and Pencil) the instruction format (on bits, including the opcode and function fields) for each of the instructions presented in Table 4.1.

Add two more R-type instructions and 2 more I-type instructions in order to complete the whole number of instructions that your processor is capable of performing.

Besides the lecture materials, you can also use Appendix 5 – MIPS Instruction Reference for a reference on MIPS instructions.

You need to specify the encoding (on bits) in all of the fields from the instruction.

Write the RTL abstract for all the 15 instructions from your MIPS 16 instruction set. Draw the processing diagram for all the instructions (add, and, sll, lw, beq, j – in the laboratory, the rest as homework).

For your MIPS 16 processor you will ignore the overflow exceptions that can appear during ALU operations (example: add instruction)

Give an example for each instruction including the bit encoding off all fields (including the instruction operands). Example: add $2, $4, $3 → "… the 16 bits…".

Attention: in order to increase the encoding readability of each instruction use the "_" symbol between the instruction fields (opcode, rs, etc.). This is also supported by VHDL and has no effect on the bit string. For VHDL it is mandatory to specify the binary encoding B before the string of bits (or X, O for hexadecimal and octal encoding respectively).

`B"001_010_011_100_1_111"` is equivalent to **`"0010100111001111"`**

4.3.2. MIPS 16 test program

Write a short program with the instructions that you have defined for your MISP 16 processor (paper and pencil). Describe the program in assembly language, then each instruction in machine code (16-bit encoding for each instruction, use "_" between the fields of the instructions).

Using assignment 3.1 from laboratory 3 (ROM memory whose addresses are generated by a Mono Pulse Generator controlled counter), introduce your program in

the ROM memory and trace it on the Basys board. When writing your program in the ROM memory you have to write a comment for each instruction, i.e. the assembly language description for each instruction. Your program should be visible in parallel to the machine code.

Optionally, you are invited to write a more complex program for your MIPS processor.

4.3.3. Data-Path for MIPS 16

Draw the Data-Path of your single-cycle MIPS 16 processor. Be sure to include all the necessary components on the data-path, such that all the 15 instructions execute correctly.

Starting from the RTL abstract description, identify the values of the control signals for each instruction. Draw a table with the control signals and their values (see lecture 4 for details).

4.4 References

[1] Computer Architecture Lectures 3 & 4 slides.
[2] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012
[3] MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02
[4] 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.

Laboratory 5

5. Single-Cycle MIPS CPU Design (2): 16-bits version – One clock cycle per instruction

5.1. Objectives

Study, design, implement and test

- **Instruction Fetch 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**

5.2. 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 start 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 Fetch Unit of your own single cycle MIPS processor – MIPS 16.

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.

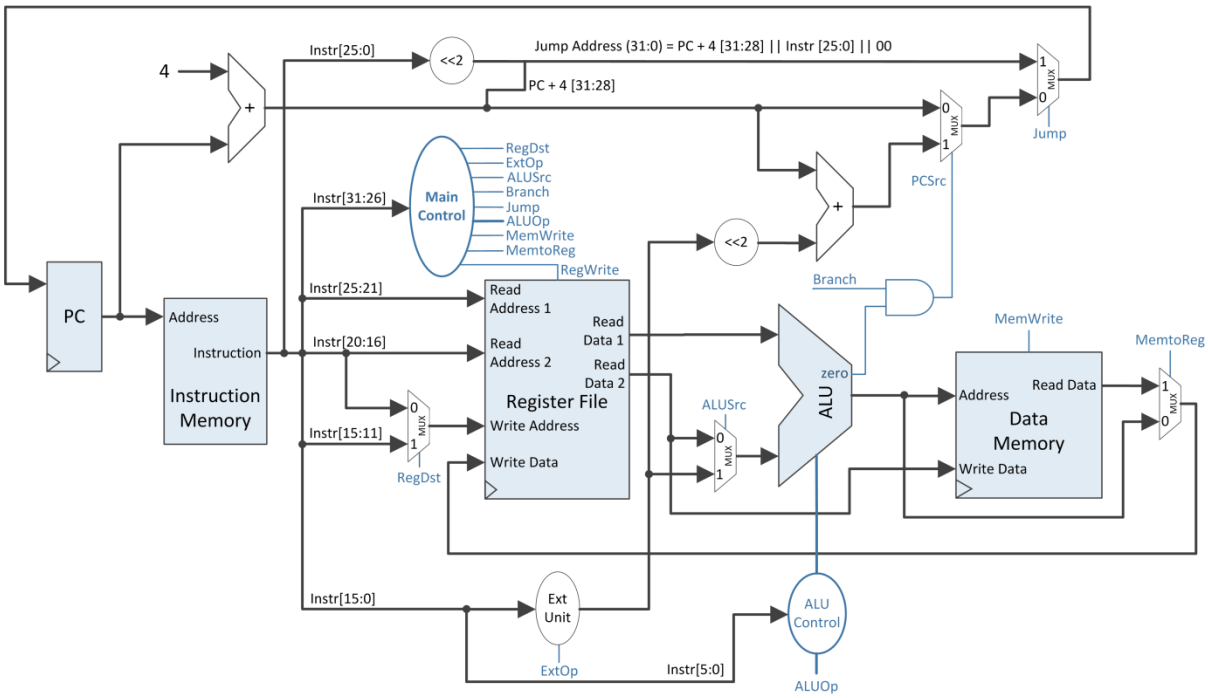


Figure 5.1: MIPS 32 Single-Cycle Data-Path + Control

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

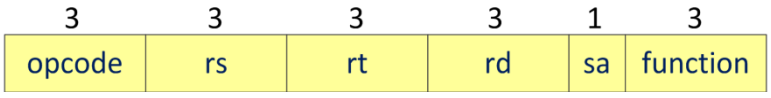


Figure 5.2: R-type Instruction format

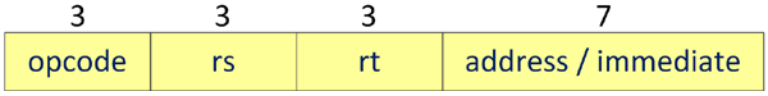


Figure 5.3: I-type Instruction format

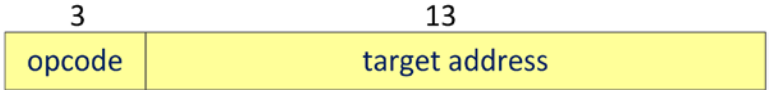


Figure 5.4: J-type Instruction format

The IF (Instruction Fetch) unit consists in the following components (you will not declare new entities):

- Program Counter
- Instruction Memory (ROM)
- Adder

In addition, there exist two multiplexers for selecting the next instruction address. Please refer to the previous laboratory for the characteristics of these components for your Single-Cycle MIPS 16 processor. The data-path of the Instruction Fetch Unit is presented in Figure 5.5.

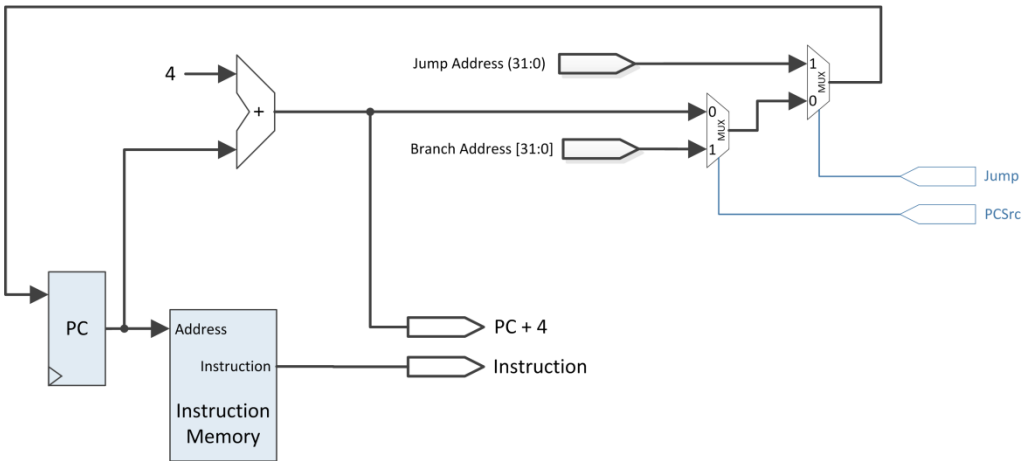


Figure 5.5: Instruction Fetch Data-Path for MIPS 32

Usually the IF unit provides, as output, the instruction to be executed as well as the next sequential instruction address. In the case of jump or branch instructions, the IF unit must also receive, as inputs, the branch target address and the jump address, together with the control signals that will select the next instruction address.

The inputs of the IF unit are:

- The clock signal (for the PC)
- The branch target address
- The jump address
- Jump Control signal
- PCSrc Control signal (for branch)

The outputs of the IF unit are:

- The instruction to be executed by the MIPS processor
- The next sequential instruction address (PC + 4)

The meaning of the control signals:

- Jump = 1 → PC ← jump address
- Jump = 0
  - If PcSrc = 0 → PC ← + 4
  - If PcSrc = 1 → PC ← branch target address

5.3. Laboratory Assignments

Read carefully and completely each activity before you begin!

Prerequisites:

- You need to have all the assignments from the previous lab completed
- 15 instructions for your own Single-Cycle MIPS 16 defined (Laboratory 4 Assignment 4.3.1)
- RTL abstract / instruction formats written on paper for all the 15 instructions
- Data-Path for MIPS 16 – paper and pencil (Laboratory 4 Assignment 4.3.3; use **Error! Reference source not found.** from this laboratory as reference, or lecture 4 for more details)
- Xilinx project with test_env including at least the ROM with the test program written in machine code (Instruction Memory) (Laboratory 4 Assignment 4.3.2)

Attention: If the homework from the previous laboratory 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!

5.3.1. Instruction Fetch design

Taking into account the instruction fetch data-path from Figure 5.5 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 IF entity will contain the hardware components described in Figure **5**.**5**, that will not be implemented with other components (use behavioral VHDL description).

The instruction memory will be the ROM memory from the previous laboratory with the program written in machine code. Do not increase the memory size to 2^{16} locations, but rather use a subset of the program counter to address the ROM memory (least significant 8 bits).

The adder will be implemented with + 1 in the VHDL description (not + 4 – MIPS32 case).

The program counter register will be a rising edge triggered D Flip-Flop with all the necessary input addresses (sequential operation (PC + 1), branch target address and jump address).

Attention! The new PC value will only be written in the PC register when a button from the Basys board is pressed (use one enable signal from the MPG as input to the IF Unit in order to activate the write of the PC register). Additionally use another button (MPG enable signal) to reset the PC register (another input for the IF Unit).

5.3.2. Testing of the Instruction Fetch Unit

In the “test_env” entity declare and instantiate your Instruction Fetch Unit. Connect the IF Unit together with the MPG and SSD components previously designed. Use two enable signals from the MPG to reset and to validate the writing in the PC register. For connecting to the SSD use, both outputs of the instruction fetch unit (instruction and PC+1).

Use sw(7) to control the display on the SSD (use a multiplexor):

- sw(7) = 0 → display the instruction on the SSD
- sw(7) = 1 → display the next sequential PC (PC + 1 output) on the SSD

For implementing and testing conditional and unconditional jumps, you will map the two control signals to two switches:

- Use sw(0) for Jump control signal.
- Use sw(1) for the PCSrc control signal.

Use “hard-coded” values for the branch target address and jump address inputs of the instruction fetch unit:

- 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)

5.4. References

[1] Computer Architecture Lectures 3 & 4 slides.
[2] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012
[3] MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02
[4] 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.

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**

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.

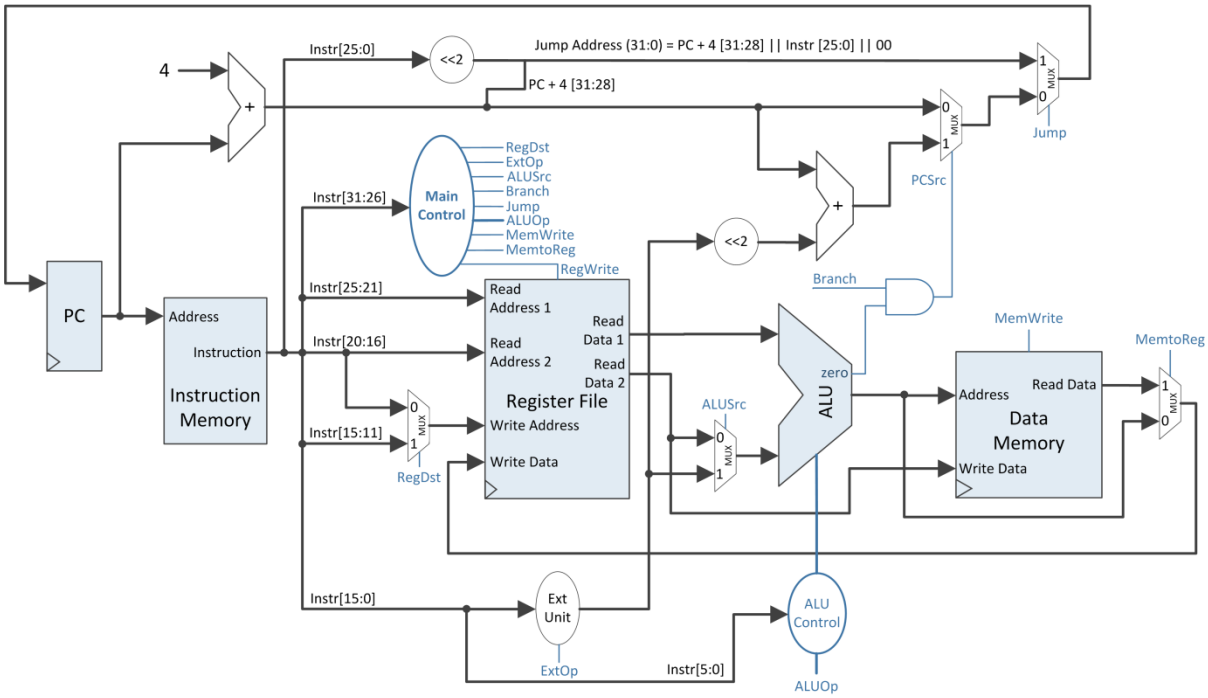


Figure 6.1: MIPS 32 Single-Cycle Data-Path + Control

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

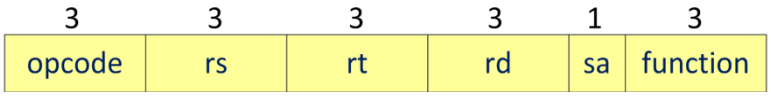


Figure 6.2: R-type Instruction format

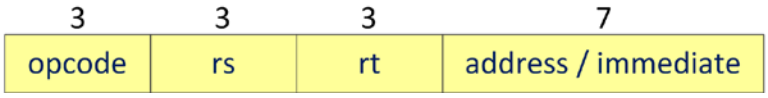


Figure 6.3: I-type Instruction format

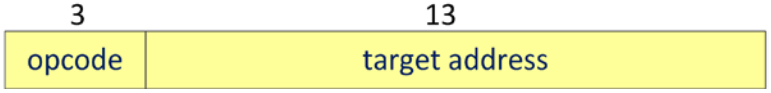


Figure 6.4: J-type Instruction format

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.

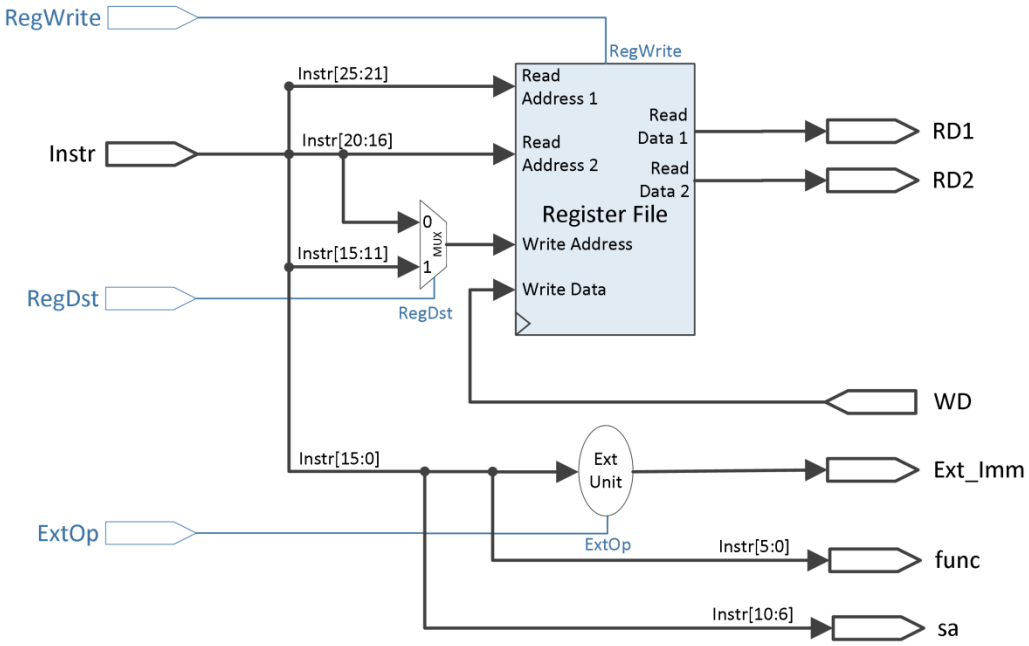


Figure 6.5: Instruction Decode Data-Path for MIPS 32

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.

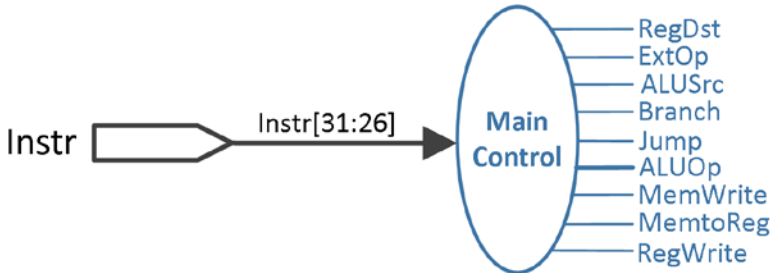


Figure 6.6: MIPS 32 Single Cycle Main Control Unit

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 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. 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)

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] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012
[3] MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02
[4] 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.

Laboratory 7

7. Single-Cycle MIPS CPU Design (4): 16-bits version – One clock cycle per instruction

7.1. Objectives

Study, design, implement and test

- **Instruction Execute Unit for the 16-bit Single-Cycle MIPS CPU**
- **Testing of the Arithmetical-Logical Instructions**

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**

7.2. 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 and hopefully finish in this laboratory) 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 Execute Unit of your own single cycle MIPS 16 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.

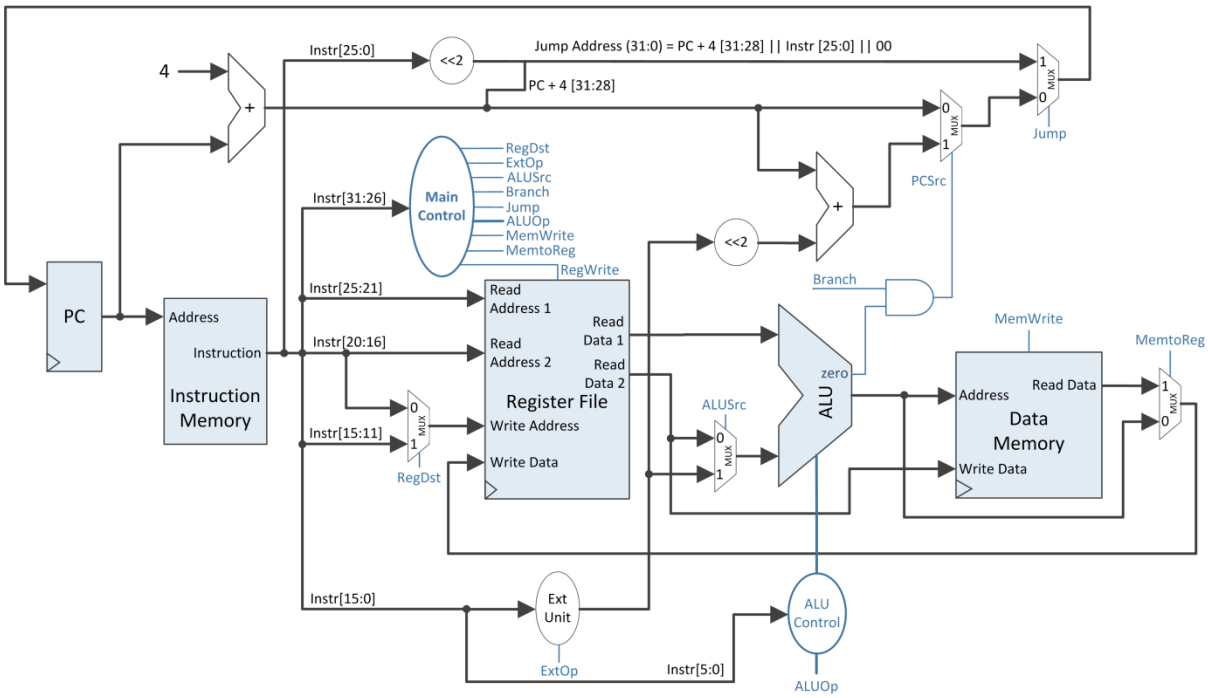


Figure 7.1: MIPS 32 Single-Cycle Data-Path + Control

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

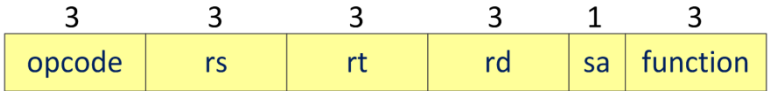


Figure 7.2: R-type Instruction format

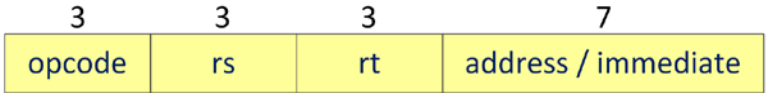


Figure 7.3: I-type Instruction format

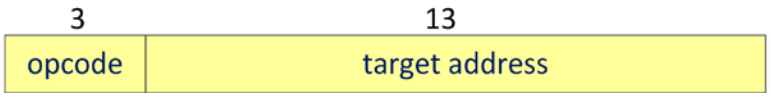


Figure 7.4: J-type Instruction format

The Execute Unit (Ex) consists in the following components:

- Arithmetic Logic Unit (ALU)
- ALU Control
- Multiplexer
- Shift Left 2 and adder for branch target address computation

Please refer to laboratories 2 and 4 for the characteristics of these components for your Single-Cycle MIPS 16 processor.

The data-path of the Instruction Execute Unit is presented in Figure 7.5.

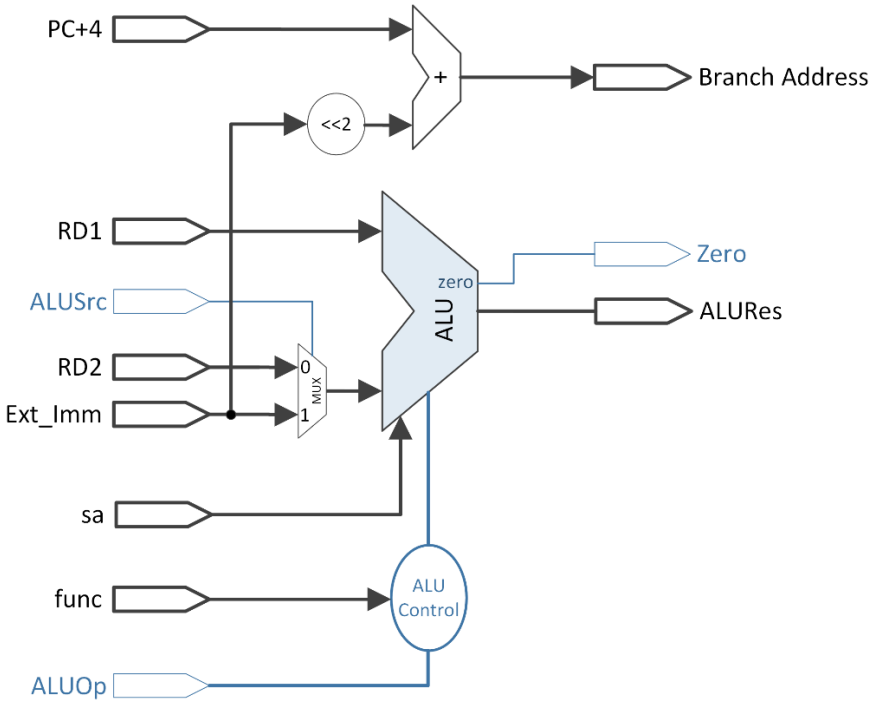


Figure 7.5: Instruction Execute Data-Path for MIPS 32

The EX unit provides, as output, the ALU Result used for writing the result of arithmetical/ logical instructions in the Register File or used as the address for the Data Memory in the case of lw and sw instructions. In addition the ALU provides another output, the Zero Signal, which indicates whether the result of the ALU is equal to zero or not (if the result is equal to zero the signal will have as value 1, otherwise 0). For simplicity of your future, first version, pipeline implementation, the EX unit also includes the branch target address computation.

The inputs of the Ex unit are:

- Next Sequential Instruction Address (PC+4)
- 32-bit Read Data 1 (RD1)
- 32-bit Read Data 2 (RD2)
- 32-bit Extended Immediate (Ext_Imm)

- 6-bit function field (func)
- 5-bit shift amount (sa)
- Control Signals:
  - o ALUSrc – selects between Read Data 2 and Extended Immediate as input to the second port of the ALU
  - o ALUOp – ALU operation code provided by the Main Control Unit

The outputs of the Ex unit are:

- 32-bit Branch target address
- 32-bit ALU result (ALURes)
- 1-bit Zero signal

The meaning of the control signals:

- ALUSrc = 0 → the Read Data 2 signal is the second input for the ALU
- ALUSrc = 1 → the Extended Immediate signal is the second input for the ALU
- ALUOp → is defined by the Main Control Unit according to the operations implemented in the ALU.

The branch target address is computed with the following formula:
Branch Address ← PC + 4 + S_Ext(Imm) << 2;

The Zero signal together with the Branch Control Signal is used in order to select between the normal sequential execution of the program (PC + 4) or the branch target address.

ALU Control Unit defines the ALU operations encoded in the ALUCtrl control signal. For I-type instructions, the encoding of the ALUCtrl is simply defined by the ALUOp signal. For R-type instructions, the value of ALUCtrl is defined by the fixed value ALUOp and the function field.

7.3. Laboratory Assignments

Read carefully and completely each activity before you begin!

Prerequisites:

- All the assignments from the laboratories 4, 5, 6 completed
- The instruction fetch unit implemented and tested on the Digilent Development Board.
- The instruction decode unit implemented and tested on the Digilent Development Board.
- Xilinx project with “test_env” including the IF and ID units (Laboratory 6 Assignment 6.3.3)

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!

7.3.1. Instruction Execute Unit design

Taking into account the instruction execution data-path from Figure 7.5 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 Ex entity will contain the hardware components described in Figure 7.5 that will not be implemented with other components (use behavioral VHDL description)

Use one line of code for the branch target address computation (no shift required only addition).

Use a process (with a case statement) for the ALU implementation as in laboratory 2. The 1-bit shift amount is used only for the logical and / or arithmetic shift operations.

Use a process (with a case statement) for the implementation of the ALU Control. The encoding of the ALUCtrl control signal is dependent on your own 15 instructions and defines the arithmetic-logical operations implemented by the ALU.

Attention! Be careful when transforming the data fields from Figure 7.5 (MIPS 32) into your own single-cycle MIPS 16 implementation.

7.3.2. Testing of the Instruction Execute Unit

Instantiate the Execution Unit in the "test_env" project. At this moment, you will connect the output of the Execution Unit (AluRes) to the Write Data port of the ID Unit (WD input)

You have to test all your arithmetical / logical instructions on the Digilent Development Board: Add, Sub, Shift left, Shift Right, And, Or, Addi, etc. Make sure that all the instructions perform correctly.

All the signals present on the data-path must be connecting to the SSD, i.e. the outputs of the IF, ID and EX units. Use 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 Ext_Imm signal on the SSD

- sw(7:5) = 101 → display the ALURes signal on the SSD
- sw(7:5) = 111 → 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. 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)

Trace your program (without any memory operation or conditional /unconditional jump) on the Digilent Development Board instruction by instruction. Be sure that all the control signals / data fields are correct.

7.4. References

[1] Computer Architecture Lectures 3 & 4 slides.
[2] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012
[3] MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02
[4] 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.

Laboratory 8

8. Single-Cycle MIPS CPU Design (5): 16-bits version – One clock cycle per instruction

8.1. Objectives

Study, design, implement and test

- **Memory Unit for the 16-bit Single-Cycle MIPS CPU**
- **Write Back Unit for the 16-bit Single-Cycle MIPS CPU**
- **Other necessary connections for branch / jump address computation**
- **Test the 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**

8.2. 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 and hopefully finish in this laboratory) 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 Memory Unit, Write Back Unit and the rest of the connections of your own single cycle MIPS 16 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.

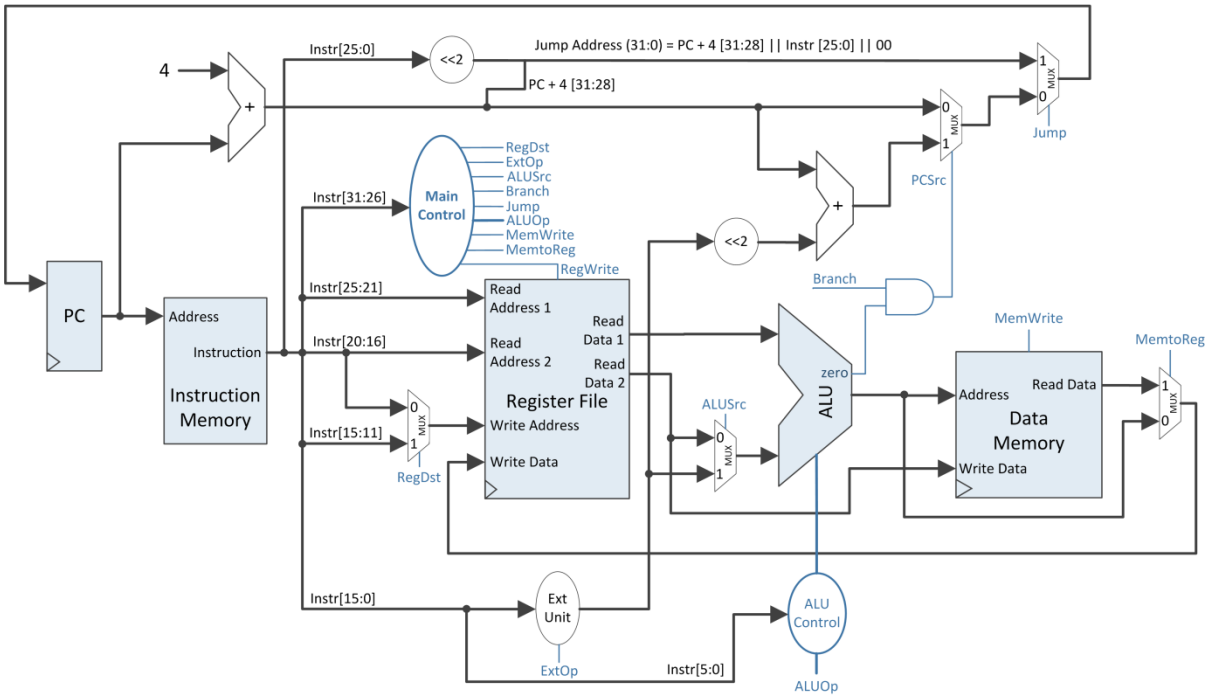


Figure 8.1: MIPS 32 Single-Cycle Data-Path + Control

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

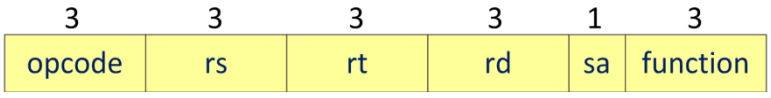


Figure 8.2: R-type Instruction format

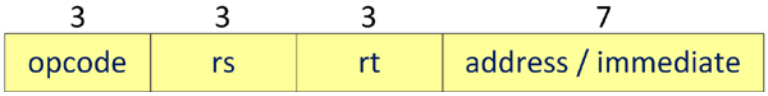


Figure 8.3: I-type Instruction format

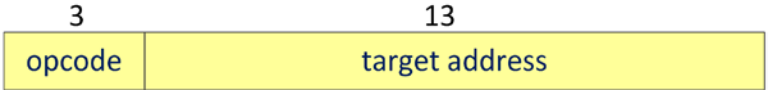


Figure 8.4: J-type Instruction format

The Memory Unit consist in the following component

- Data Memory

The data-path of the Memory Unit is presented in Figure 8.5.

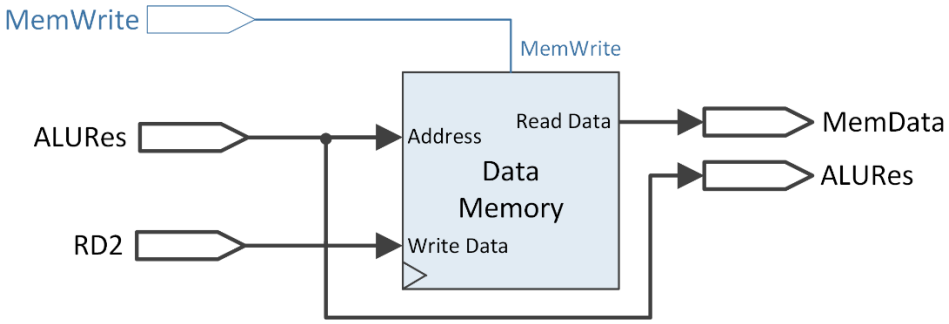


Figure 8.5: Memory Unit Data-Path for MIPS 32

The Data Memory is a RAM with asynchronous read and synchronous write operations. A similar RAM implementation (with synchronous read) was done in laboratory 3.

The inputs of the Memory Unit:

- The clock signal (for the Data Memory Writes)
- 32-bit ALURes signal – consists in the address for the Data Memory
- 32-bit RD2 signal – the second output of the Register File (used only for store word instructions) is the Write Data field for the Data Memory
- MemWrite control signal

The outputs of the Memory Unit:

- 32-bit MemData, the Read Data from the Data Memory (used only for load word instructions).
- 32-bit ALURes, this signal is also the result of the arithmetic-logical instructions that must be stored in Register File, so it is also fed as output for the Memory Unit and input to the Write Back Unit.

The only control signal present in this stage is the MemWrite Control Signal.

- MemWrite = 0 → Nothing is written in the Data Memory.
- MemWrite = 1 → The RD2 signal is written in the Data Memory at the address indicated by the ALURes signal.

The Write Back unit is simply the last multiplexor from Figure 8.1. The rest of the components are the AND gate for generating the PCSrc control signal, the jump address computation.

The control signal for the Write Back multiplexor is MemtoReg and identifies what value is fed to the Write Data port of the Register File in the ID state:

- MemtoReg = 0 → the ALURes signal is the input to the Write Data port of the Register File.
- MemtoReg = 1 → the MemData is the input to the Write Data port of the Register File.

The PCSrc signal identifies if the current instruction is a Branch instruction and if the content of the rs and rt registers are the same; i.e. RF[rs] == RF[rt].

PCSrc <= Branch and Zero;

8.3. Laboratory Assignments

Read carefully and completely each activity before you begin!

Prerequisites:

- All the assignments from the laboratories 4, 5, 6, 7 completed
- The instruction fetch unit implemented and tested on the Digilent Development Board.
- The instruction decode unit implemented and tested on the Digilent Development Board.
- The instruction execute unit implemented and tested on the Digilent Development Board
- The memory unit implemented and tested on the Digilent Development Board
- Xilinx project with “test_env” including the IF, ID, EX, MEM units (Laboratory 7 Assignment 7.3.3)

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!

8.3.1. Memory Unit Design

Describe a new component (new entity) for the Memory Unit. Use the RAM implementation from laboratory 3 and change the read operation to an asynchronous one. Write only with processes inside the Memory Unit.

Instantiate the Memory Unit in the “test_env” project. Connect all the signals from the Memory Unit in the data-path. The MemWrite signal should be validated with an output of the MPG component as it was previously implemented for the writing in the Register File (RegWrite signal).

8.3.2. Adding the Write Back Unit and the jump address computation

Add the write back multiplexor for your own MIPS processor. Use only one line of code to implement this multiplexor.

Complete your own MIPS processor implementation with the jump address computation and the PCSrc signal computation. Complete all the necessary connections for the data-path as in Figure 8.1 (jump address, branch target address, write back in the register file, etc.).

Test the LW, SW, BEQ and Jump Instructions for your MIPS processor.

8.3.3. Testing it ALL: You own Single-Cycle MIPS 16 processor

At this moment, you should have all the components form the data-path implemented in the “test_env” project.

All the signals present on the data-path must be connecting to the SSD, i.e. the outputs of the IF, ID, EX, M and WB units. Use 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 Ext_Imm signal on the SSD
- sw(7:5) = 101 → display the ALURes signal on the SSD
- sw(7:5) = 110 → display the MemData signal on the SSD
- sw(7:5) = 111 → 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. 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)

If needed for debugging the processor on the Digilent Development Board you can additionally display other signals on the SSD / LEDs: branch target address, jump address, ALUCtrl, etc.

Now trace your program on the Digilent Development Board instruction by instruction. Be sure that all the control signals / data fields are correct.

Present your Single-Cycle MIPS implementation to your TA.

8.4. References

[1] Computer Architecture Lectures 3 & 4 slides.

[2] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012
[3] MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02
[4] 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.

Laboratory 9

9. Pipeline MIPS CPU Design: 16-bits version

9.1. Objectives

Study, design, implement and test

- **MIPS 16 CPU, pipeline version**

Familiarize the students with

- Pipeline CPU design
- Xilinx® ISE Webpack
- Digilent Development Boards (DDB)
  - **Digilent Basys Board – Reference Manual**

9.2. Transforming the MIPS 16 Single-Cycle CPU to a Pipeline CPU

! You must attend/read lecture 8 in order to fully understand the Pipeline CPU

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

The data-path of the single-cycle 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.

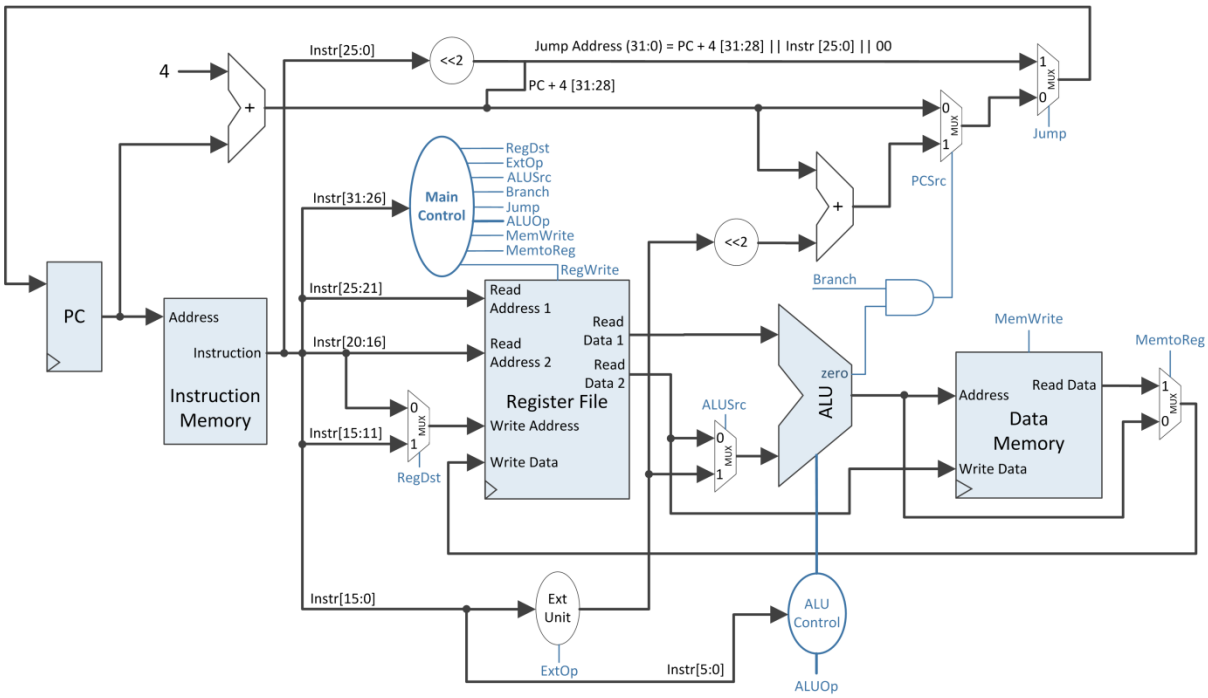


Figure 9.1: MIPS 32 Single-Cycle Data-Path + Control

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

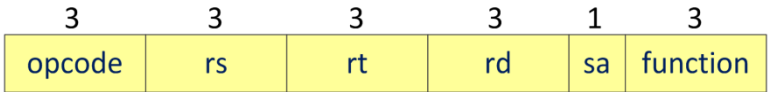


Figure 9.2: R-type Instruction format

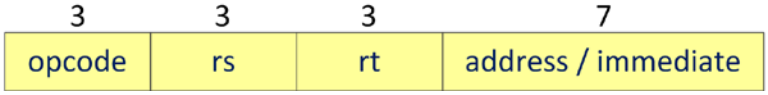


Figure 9.3: I-type Instruction format

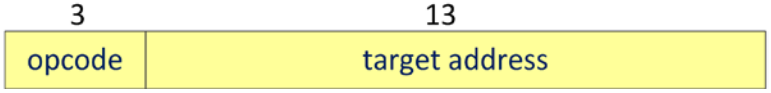


Figure 9.4: J-type Instruction format

The main issue with the single-cycle MIPS CPU is the length of the critical path, for the load word instruction (see lecture 04). The necessary time for transmitting the data along the critical path must be covered by the clock cycle time. This results in a long cycle time (slow clock).

In order to reduce the clock cycle time, the solution is to partition the data-path along the critical path with rising edge triggered registers (D flip-flops). These registers are inserted between the MIPS 32 functional units that coincide with the instruction execution phases: IF, ID, EX, MEM, WB. In this manner, one can simultaneously execute at most 5 instructions, each of them executing one of the five execution phases. The pipeline execution units are also referred to as **stages**.

The data-path together with the control unit for the pipelined MIPS 32 CPU is presented in Figure 9.5.

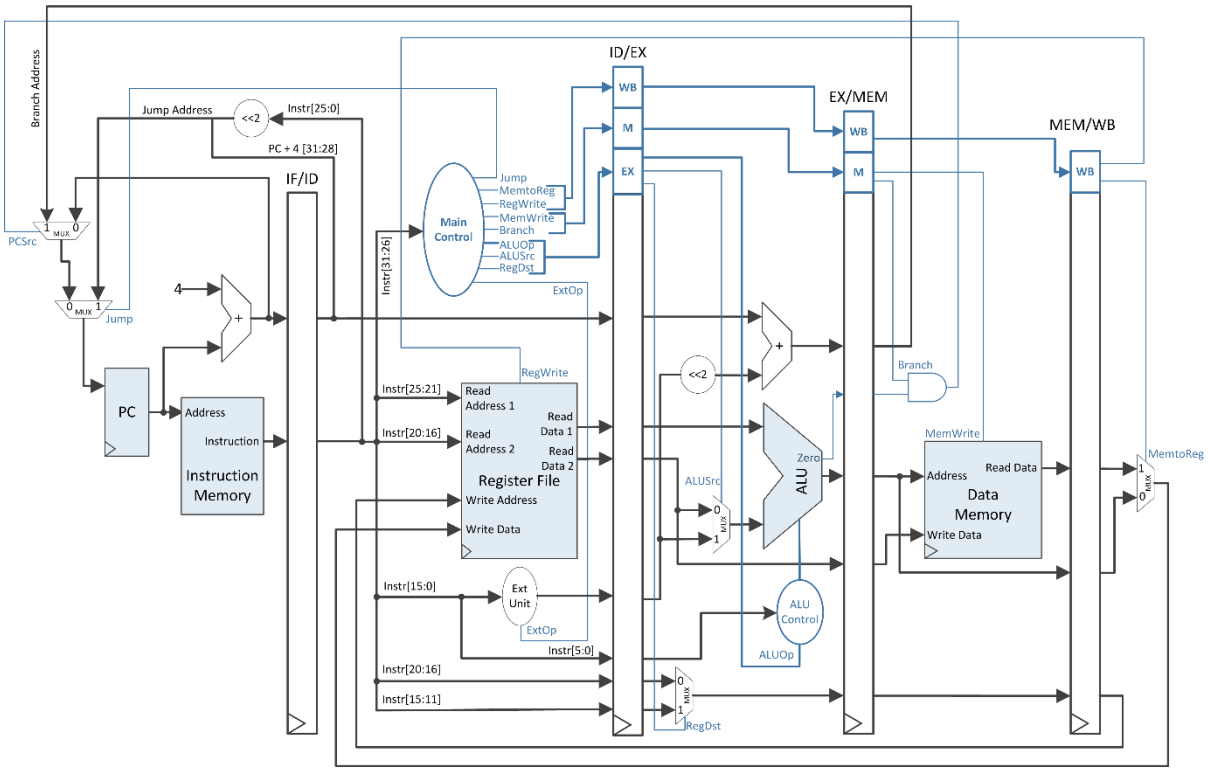


Figure 9.5: MIPS 32 Pipeline Data-Path + Control, obtained from the partitioning of the Single-Cycle Data-Path

Each intermediate register will be referred depending on its position between the pipeline stages. The register between the IF stage and the ID stage is IF/ID, the one between ID and EX is ID/EX, etc.

The role of these intermediate registers is to hold the intermediate results of the instruction execution in order to provide these results to the next stage, in the next clock cycle.

Furthermore, the execution on the data-path depends on the control signals values, which are specific for each instruction. So, through the intermediate registers (starting with the ID/EX register) the control signals will also be provided for the next stages. The control signals are symbolically grouped after the stage name where they belong.

The control signals are transmitted together with the intermediate results until the stages where they are needed.

Lecture 8 explains in more detail the design of the pipeline CPU; the details presented so far represent the necessary knowledge for transforming your own MIPS 16 single-cycle CPU into a pipeline one.

One notable difference between the two data-paths is that the multiplexer used for selecting the write address for the Register File is placed in EX, not in ID as in the single-cycle CPU case. There are 2 possible solutions:

a) Leave it in the ID stage. In this case, the RegDst signal will not be transmitted through ID/EX and will be connected directly from the control unit.
b) Move it to the EX stage, modifying the input / output ports of the ID and EX units, and transmit the RegDst signal according to the presented pipeline data-path.

Observation: The MemRead signal will be ignored, as in the single-cycle case.

9.3. Laboratory Assignments

Read carefully and completely each activity before you begin!

Prerequisites:

- Xilinx project with "test_env" including the complete and correct implementation of the single-cycle MIPS 16 CPU.

9.3.1. Verify the MIPS 16 CPU design

Before you begin transforming your single-cycle processor into a pipeline one, generate the *.bit file and investigate the clock frequency of your processor: **Processes → Design Summary/Reports.** Open **Detailed Reports → Synthesis Report.** In the synthesis report, locate the section about the clock frequency, similarly with the following text:

TIMING REPORT
...
Timing Summary:
…
  Minimum period: 13.284ns (Maximum Frequency: **75.280MHz**)
…

Write down the frequency of your single-cycle processor, so that you can compare it with the frequency of the pipeline version.

If you have not completed the testing of the single-cycle processor, you must complete it now, before you begin the pipeline implementation.

The pipeline implementation must start from a fully functional single-cycle MIPS 16 version.

9.3.2. Design of the intermediate registers (paper and pencil)

For each intermediate register identify the fields that it must store, taking into account your own MIPS 16 implementation.

Use the data-path from Figure 9.5 (!) but keep in mind your MIPS 16 processor's features: 16 bits, not 32; based on your own chosen instructions the data-path may contain additional elements.

For example, for the first intermediate register, IF/ID, one must memorize the following two fields (generic name according to the stage they belong):

- IF.PC+1 – 16 bits
- IF.Instruction – 16 bits

It results that the IF/ID register should contain 32-bits.

Similarly describe the ID/EX, EX/MEM, MEM/WB registers.

When describing the fields of the next intermediate registers, take a look at the associated functional units (the inputs unit and the destination unit respectively: example for IF/ID the IF and ID units respectively) in the laboratories 5, 6, 7 and 8. Identify the fields from the input/output ports of the functional units. This step will be used in the next assignment.

9.3.3. Describe the intermediate registers in VHDL

For this assignment, you will work in the "test_env" entity (where the components of the processor are instantiated and connected together).

Attention: When introducing the new pipeline registers, some small modifications in your functional units may appear. You will easily identify these modifications by carefully studying Figure 9.5 and the particular data-paths for each functional unit of the MIPS 16 processor – laboratories 5, 6, 7, 8. For example, the ID unit needs an additional input port for the write address of the Register File, address that will come from the last pipeline register MEM/WB.

You will not declare new entities for the pipeline registers. For each register, you have to declare a signal of appropriate length (according to the previous assignment) and the behavior of the pipeline register will be described with one process (synchronous data transfer on the rising edge of the clock signal). In order to ease the testing of the processor each register will be controlled with an enable signal (the same MPG output used for validating the writing in the PC register).

You will realize the partitioning of the single-cycle data-path with the pipeline registers and make the necessary correct connections in the mapping of the functional units.

For example, the value of the RegWrite control signal will be transmitted through the MEM/WB register and will be mapped at the input of the ID Unit.

Pipeline register example (one can also use concatenation): for IF/ID you must declare a 32-bit signal RegIF_ID and describe the behavior of the register in one process:

On rising edge of the clock
RegIF_ID(31..16) <= PC+1;
RegIF_ID(15..0) <= Instruction;
Where PC+1 and Instruction are the outputs of the IF unit.

Alternatively, you can declare new signals and concatenate the stage name to the signal name like:
On rising edge of the clock
PC_ID <= PC + 1;
Instruction_ID <= Instruction;

9.3.4. Homework – Paper and Pencil

Draw your own MIPS 16 pipeline CPU data-path together with the control unit and control signals.

9.4. References

[1] Computer Architecture Lectures 3 & 4 slides.
[2] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012
[3] MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02
[4] 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.

Laboratory 10

10. Pipeline MIPS CPU Design (2): 16-bits version

10.1. Objectives

Study, design, implement and test

- **MIPS 16 CPU, pipeline version with the modified program without hazards**

Familiarize the students with

- Pipeline CPU design
- Xilinx® ISE Webpack
- Digilent Development Boards (DDB)
  - **Digilent Basys Board – Reference Manual**

10.2. Transforming the MIPS 16 Single-Cycle CPU to a Pipeline CPU

! You must attend/read lecture 8 in order to fully understand the Pipeline CPU

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

The data-path of the single-cycle 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.

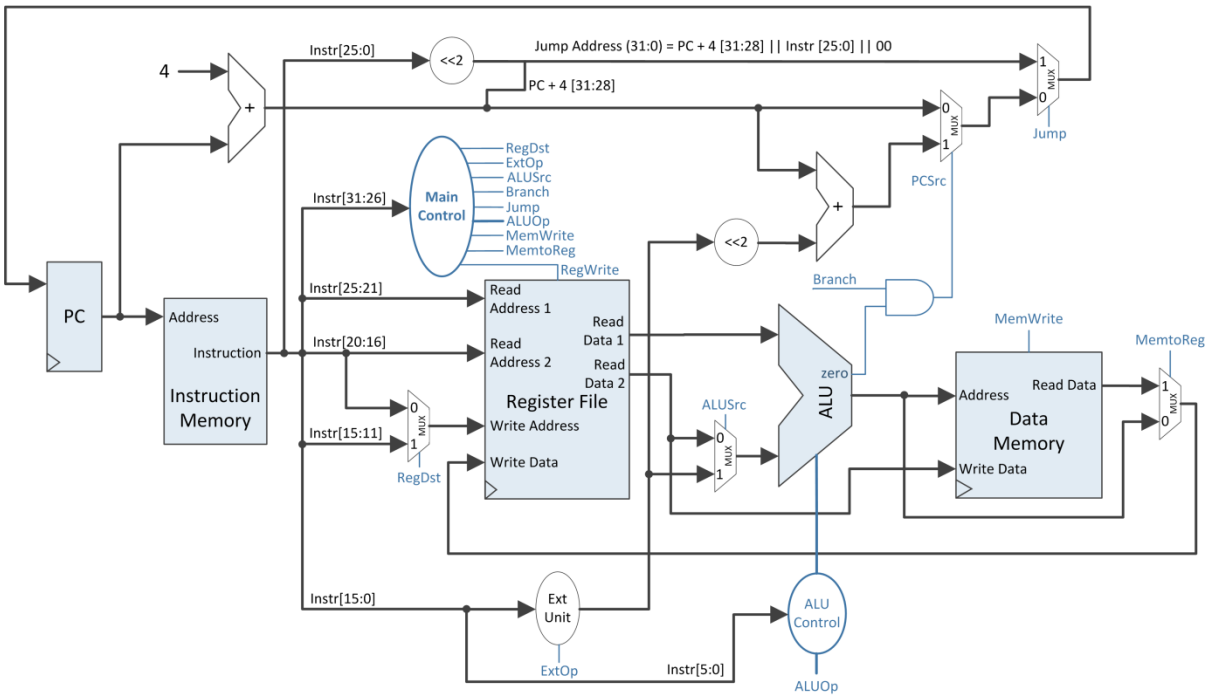


Figure 10.1: MIPS 32 Single-Cycle Data-Path + Control

The main issue with the single-cycle MIPS CPU is the length of the critical path, for the load word instruction (see lecture 04). The necessary time for transmitting the data along the critical path must be covered by the clock cycle time. This results in a long cycle time (slow clock).

In order to reduce the clock cycle time, the solution is to partition the data-path along the critical path with rising edge triggered registers (D flip-flops). These registers are inserted between the MIPS 32 functional units that coincide with the instruction execution phases: IF, ID, EX, MEM, WB. In this manner, one can simultaneously execute at most 5 instructions, each of them executing one of the five execution phases. The pipeline execution units are also referred to as **stages**.

The data-path together with the control unit for the pipelined MIPS 32 CPU is presented in Figure 10.2.

Each intermediate register will be referred depending on its position between the pipeline stages. The register between the IF stage and the ID stage is IF/ID, the one between ID and EX is ID/EX, etc.

The role of these intermediate registers is to hold the intermediate results of the instruction execution in order to provide these results to the next stage, in the next clock cycle.

Furthermore, the execution on the data-path depends on the control signals values, which are specific for each instruction. So, through the intermediate registers (starting with the ID/EX register) the control signals will also be provided for the next stages. The control signals are symbolically grouped after the stage name where they belong.

The control signals are transmitted together with the intermediate results until the stages where they are needed.

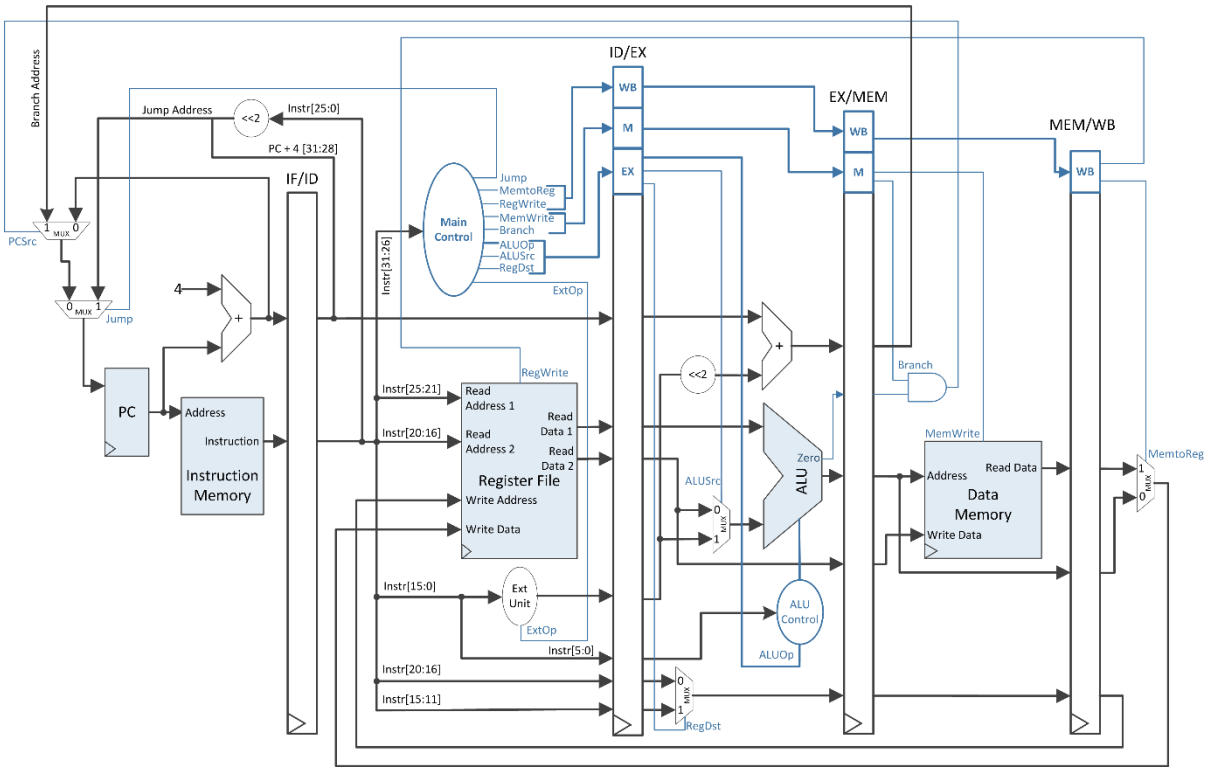


Figure 10.2: MIPS 32 Pipeline Data-Path + Control, obtained from the partitioning of the Single-Cycle Data-Path

Lecture 8 explains in more detail the design of the pipeline CPU; the details presented so far represent the necessary knowledge for transforming your own MIPS 16 single-cycle CPU into a pipeline one.
One notable difference between the two data-paths is that the multiplexer used for selecting the write address for the Register File is placed in EX, not in ID as in the single-cycle CPU case. There are 2 possible solutions:

c) Leave it in the ID stage. In this case, the RegDst signal will not be transmitted through ID/EX and will be connected directly from the control unit.
d) Move it to the EX stage, modifying the input / output ports of the ID and EX units, and transmit the RegDst signal according to the presented pipeline data-path.

Observation: The MemRead signal will be ignored, as in the single-cycle case.

10.3. Hazards in MIPS

Hazards are situations in which an instruction cannot be executed in the next clock period. The hazard can be classified as:

1. **Structural Hazards (resource dependency)**
   - 2 instructions try to use the same resource simultaneously for different purposes → resource constraints
2. **Data Hazards (data dependency)**
   - Attempt to use data before it is ready (available)
   - For an instruction in the ID phase, the operands might still be processed in other pipeline stages
3. **Control Hazards (condition and control dependency)**
   - The branch decision and branch target address are not known until the MEM stage. The jump address is computed in the ID stage.
   - Pipelining of jumps, branches and other instructions that modify the sequential flow of the program

These hazards have been thoroughly presented during lecture 8 (you are encouraged to read them!). Optimal solutions (in hardware) are relying on forwarding and stalling the pipeline (see the lecture notes for reference). For your MIPS 16 pipeline implementation, you should implement the software solution, modifying your program such that the data and control hazards are avoided.

The basic change in your program should be the following: introduce NoOp (No Operation) instructions between the instructions where the hazard exists.

NoOp instruction should not change anything in your processor (ex. sll $0, $0, 0; add $0, $0, $0, etc.)

10.3.1. Structural Hazards

Structural hazards occur when instructions from two different pipeline stages are trying to use the same resource in the same cycle.

Special attention should be given to the structural hazard that can occur when two instructions at distance of 3 are using the same register (from the RF). In the following example, we presume that instr1 and instr2 do not have hazard with other instructions.

Structural hazard at **RF**
add **$1**, $1, $2
instr1
instr2
add $3, **$1**, $4

Pipeline diagram (in each clock cycle we present the pipeline stage for each instruction):

Instr\Clk	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	…
add **$1**, $1, $2	IF	ID	EX	MEM	**WB**				
instr1		IF	ID	EX	MEM	WB			
instr2			IF	ID	EX	MEM	WB		
add $3, **$1**, $4				IF	**ID**	EX	MEM	WB	

During clock cycle CC5, the new value of $1, generated by the first instruction, is in WB stage, being unwritten yet in RF. Therefore, in ID stage, the 4th instruction will read the old value of $1, in cycle CC6 EX receiving the incorrect value.

There are 2 possible solutions:

1. Recommended: Modify RF block such that the writing is done in the middle of the clock cycle (test the falling edge – clk = 0 & clk'event). In this case, the RF read (being asynchronous), in the second part of CC5 the correct value of $1 occurs and it is propagated forward to EX at CC5-CC6 transition.
2. Introduce a NoOp instruction

Without Hazard
add **$1**, $1, $2
instr1
instr2
NoOp
add $3, **$1**, $4

Attention! In the following, it is assumed that you have chosen the 1st option. Otherwise, introduce an extra NoOp where necessary.

10.3.2. Data Hazards

Data hazards (Read After Write or Load Data Hazard) occur when the current instruction use as source(s) the register that will be written by other instructions that are still executing in the pipeline.

(!) In order to establish where these hazards occur you need to draw the pipeline diagram and to understand how pipelining is done (when operands are read).

The following example contains most of the data hazards that might occur in your pipelined MIPS.

Instr. Nb.	Program
1	add $1, $2, $3
2	add $3, $1, $2
3	add $4, $1, $2
4	add $5, $3, $2
5	lw $3, 5($5)
6	add $4, $5, $3
7	sw $3, 6($5)
8	beq $3, $4, -6

Hazard identification process is solved with NoOp insertions, starting from first instruction to the last one. Example:

Instr\Clk	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
add $1, $2, $3	IF	ID	EX	MEM	**WB($1)**				
add $3, $1, $2		IF	**ID($1)**	EX	MEM	**WB($3)**			
add $4, $1, $2			IF	**ID($1)**	EX	MEM	WB		
add $5, $3, $2				IF	**ID($3)**	EX	MEM	**WB($5)**	
lw $3, 5($5)					IF	**ID($5)**	EX	MEM	**WB($3)**

Instr\Clk	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11	…
add $5, $3, $2	IF	**ID($3)**	EX	MEM	**WB($5)**				
lw $3, 5($5)		IF	**ID($5)**	EX	MEM	**WB($3)**			
add $4, $5, $3			IF	**ID($3, $5)**	EX	MEM	**WB($4)**		
sw $3, 6($5)				IF	**ID($3)**	EX	MEM	WB	
beq $3, $4, -6					IF	**ID($4)**	EX	MEM	WB

Hazards are solved iteratively, starting from the first occurrence. Solving a hazard between 2 successive instructions implicitly solves the hazards between the first instruction and the instruction at distance +2. Example: between instruction 1 and 2, and 1 and 3, there is a RAW hazard (after $1). Hazard between 1 and 2 is solved first, delaying instruction 2 with 2 cycles (it should have ID on cycle CC5) => 2 NoOp. Therefore, all following instructions will be delayed with 2 cycles, so the hazard between 1 and 3 is also resolved.

There is a RAW hazard between the 2nd and 4th instructions, after $3, which can be solved by delaying with 1 cycle, inserting a NoOp after 2 or before 4.

All other hazards are being solved, resulting the following program:

Instr. Nb.	Program
1	add $1, $2, $3
2	NoOp
3	NoOp
4	add $3, $1, $2
5	NoOp
6	add $4, $1, $2
7	add $5, $3, $2
8	NoOp
9	NoOp
10	lw $3, 5($5)
11	NoOp
12	NoOp
13	add $4, $5, $3
14	NoOp
15	sw $3, 6($5)
16	beq $3, $4, -6

10.3.3. Control Hazards

Control hazards occur at instructions that alter the sequential flow of the program, when the next sequential instructions that follow (3 for BEQ and 1 for J) are implicitly executed.

For conditional jump instructions (beq, bne, etc.), the next 3 instructions will implicitly be executed, being already in the pipeline. Therefore, a simple (but not efficient) solution is to insert 3 NoOp's.

For unconditional jumps (j, jal, etc.), based on the data-path from Figure 5.5, these instructions are computing the jump address (and writing it in the PC register) in the ID stage. It means that only next instruction starts the execution, so one NoOp needs to be inserted after the j instruction. A better solution would be to insert the instruction that is after j before it, with condition that this instruction is not also a jump instruction (j, beq, etc.)

10.4. Laboratory Assignments

Read carefully and completely each activity before you begin!

Prerequisites:

- Xilinx project with "test_env" including the complete and correct implementation of the pipeline MIPS 16 CPU.

10.4.1. Verify the MIPS 16 Pipeline CPU design

You can evaluate the critical path by checking the clock frequency: Go to **Processes → Design Summary/Reports.** Open **Detailed Reports → Synthesis Report** and watch the section related to clock signal.

You should notice an increase in frequency (of 30-50%) caused by the pipelining of your MIPS (compared to the one observed in laboratory 9). One can observe that the increase in speed is not proportional to the number of pipeline stages. There is a multitude of reasons for that: stages are not balanced, the resulting circuit depends on the board's technology, the memories are implemented as distributed RAMs, etc.

10.4.2. Program analysis and hazard removal (paper and pencil)

Based on the example in section 3, identify the hazards in your program. Insert NoOp instructions where such an instruction is needed. Draw the pipeline diagram for at least 5 successive instructions in your program (for all, if there are not any hazards).

Note: By introducing the NoOp's you will need to adjust the addresses for the jump instructions in your program.

Modify the (assembly) program in the instruction memory

10.4.3. Test and evaluate the MIPS 16 pipeline

Test your design on the FPGA board. You have 2 options:

a. If your pipeline implementation was correct, without any mapping mistakes etc., then watching your final results is enough (results should be identical with your single cycle implementation).
b. If the results are different, then you should trace your program step-by-step.

Use the same display procedure as the one used for your single-cycle MIPS (with the multiplexor on switches for selecting different data to be displayed on the SSD). It is important to understand that now your outputs (for your switches configuration) will not be the same as in the single-cycle implementation. You have 5 instructions in the pipeline; some of them will be NoOp.

You can display the control signals on the LEDs. Use the delayed control signals, i.e. the control signals delayed to the stage where they are used.

If necessary, display other signals/change the displayed signals, from different stages, on the SSD.

10.4.4. Hardware optimizations for the MIPS Pipeline CPU (optional).

If you have finished and tested your MIPS pipeline CPU, you can modify your solution in order to implement the following components of the complete pipeline processor:

a. Hazard detection unit
b. Forwarding unit
c. Move the branch in the ID stage
d. Hardware stalls for the LW (RAW hazard), BEQ and J instructions.

In the end, you should have a complete pipeline implementation, as it is presented in the lecture material.

10.5. References

[1] Computer Architecture Lectures 3 & 4 slides.
[2] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012
[3] MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02
[4] 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.

Laboratory 11

11. Finite State Machines and Serial Communication

11.1. Objectives

Study, design, implement and test

- **Finite State Machines**
- **Serial Communication**

Familiarize the students with

- Xilinx® ISE Webpack
- Digilent Development Boards (DDB)
  - **Digilent Basys Board – Reference Manual**
  - **Digilent Basys 2 Board – Reference Manual**

11.2. Theoretical Background

11.2.1. Finite State Machines

A finite state machine or FSM is a model of behavior composed of a finite number of states, transitions between those states, and actions. A finite state machine is used to describe an abstract model of a control unit. XST proposes a large set of templates to describe FSMs. By default, XST tries to distinguish FSMs from VHDL or Verilog code, and apply several state encoding techniques to obtain better performance or less area.

XST supports the following state encoding techniques:

- Auto – the best suited encoding algorithm for each FSM.
- One-hot – associate one code bit and one flip-flop per state. At a given clock cycle during operation, one and only one bit of the state variable is asserted. Only two bits toggle during a transition between two states. One-hot encoding is appropriate with most FPGA targets where a large number of flip-flops are available. It is also a good alternative when trying to optimize speed or to reduce power dissipation.
- Gray – guarantees that only one bit switches between two consecutive states. It is appropriate for controllers exhibiting long paths without branching. In addition, this coding technique minimizes hazards and glitches. Very good results can be obtained when implementing the state register with T flip-flops.

- Compact – consists of minimizing the number of bits in the state variables and flip-flops. Compact encoding is appropriate when trying to optimize area.
- Johnson – like Gray, it shows benefits with state machines containing long paths with no branching.
- Sequential – consists of identifying long paths and applying successive radix two codes to the states on these paths. Next state equations are minimized.
- Speed1 – oriented for speed optimization. The number of bits for a state register depends on the particular FSM, but generally, it is greater than the number of FSM states.
- User – original encoding specified in the HDL file.

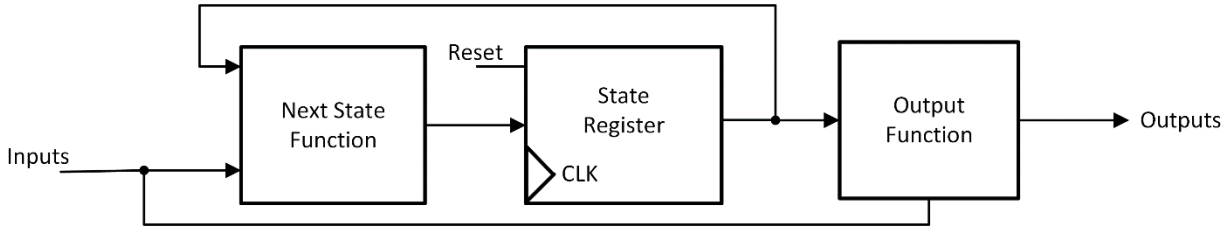


Figure 11.1: FSM Representation Incorporating Mealy and Moore Machines

When describing a finite state machine in VHDL you may have several processes (1, 2 or 3) depending upon how you consider and decompose the different parts of the preceding model. Appendix 6 (adapted from [1]) describes the VHDL finite state machine implementations.

11.2.2. Serial Communication – UART

Serial communication is the transmission or reception of data one bit at a time. Today's computers generally address data in bytes or some multiple thereof. A serial port is used to convert each byte to a stream of ones and zeroes as well as to convert streams of ones and zeroes to bytes. The serial port contains an electronic chip called a Universal Asynchronous Receiver/Transmitter (UART) that actually does the conversion. Serial transmission of digital information (bits) through a single wire or other medium is much more cost effective than parallel transmission through multiple wires.

When transmitting a byte, the UART first sends a START BIT followed by the data (generally 8 bits, but could be 5, 6, 7, or 8 bits), followed by STOP BITs. The sequence is repeated for each byte sent.

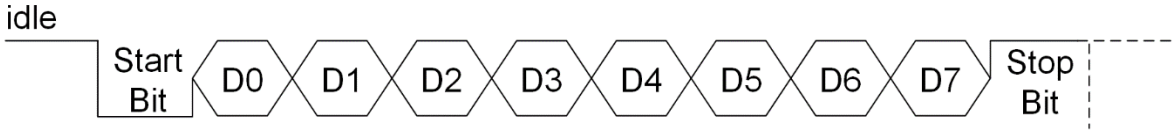


Figure 11.2: Serial Transmission Timing Diagram

Serial transmission does not involve a clock signal. The information is included in the baud rate (**number of bits per second**). Common baud rates are 2400, 4800, 9600 and 19200. This means that a bit transmitted through the serial line is valid for a given time period (the inverse of the baud rate).

The start bit is always 0, the data bits are transmitted with the LSB (least significant bit) first and MSB (most significant bit) last and the stop bit is always 1. In serial communication, the stop bit duration can have multiple values: 1, 1.5 or 2 bit periods in length. Besides the synchronization provided by the use of start and stop bits, an additional bit called a parity bit may optionally be transmitted along with the data. A parity bit affords a small amount of error checking, to help detect data corruption that might occur during transmission. One can choose even parity, odd parity, mark parity, space parity or none at all. When even or odd parity is being used, the number of marks (logical 1 bits) in each data byte is counted, and a single bit is transmitted following the data bits, to indicate whether the number of 1 bits just sent is even or odd.

The data sent through serial communication is encoded using ASCII codes (Appendix 7). Assume we want to send the letter 'A' over the serial communication channel. The binary representation of the letter 'A' is 01000001 ($0x41_{hex}$). Remembering that bits are transmitted from least significant bit (LSB) to most significant bit (MSB), the bit stream transmitted would be as follows for the line characteristics 8 bits, no parity, 1 stop bit, 9600 baud: **LSB (0 1 0 0 0 0 0 1 0 1) MSB**. This represents (Start Bit) (Data Bits) (Stop Bit). For a binary two-level signal, a data rate of one bit per second is equivalent to one Baud. To calculate the actual byte transfer rate simply divide the baud rate by the number of bits that must be transferred for each byte of data. In the case of the above example, each character requires 10 bits to be transmitted for each character. As such, at 9600 baud, up to 960 bytes can be transferred in one second.

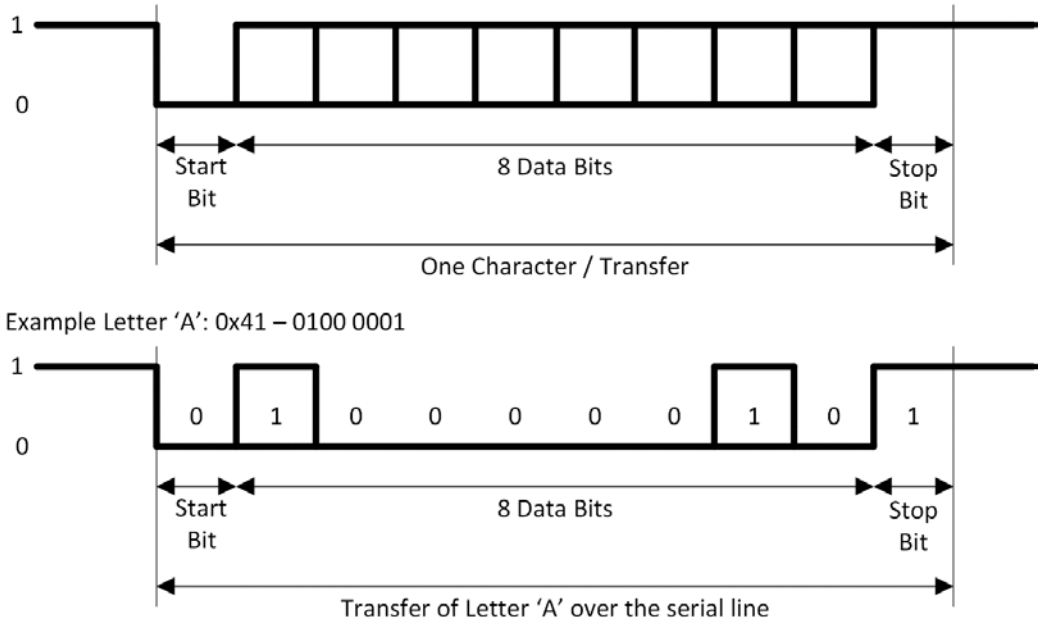


Figure 11.3: Serial Transmission Example (8 data bits, no parity)

For accurate serial communication, on the receiving end, an oversampling scheme is commonly used to locate the middle position of the transmitted bits, i.e., where the actual sample is taken. The most common oversampling rate is 16 times the baud rate. Therefore, each serial bit is sampled 16 times but only one sample is saved.

Each UART contains a shift register which is the fundamental method of conversion between serial and parallel forms.

11.3. Laboratory Assignments

11.3.1. Pmod USB-UART

Read the Pmod USB-UART reference manual. The figure below shows the connection of the USB-UART peripheral module to the FPGA board.

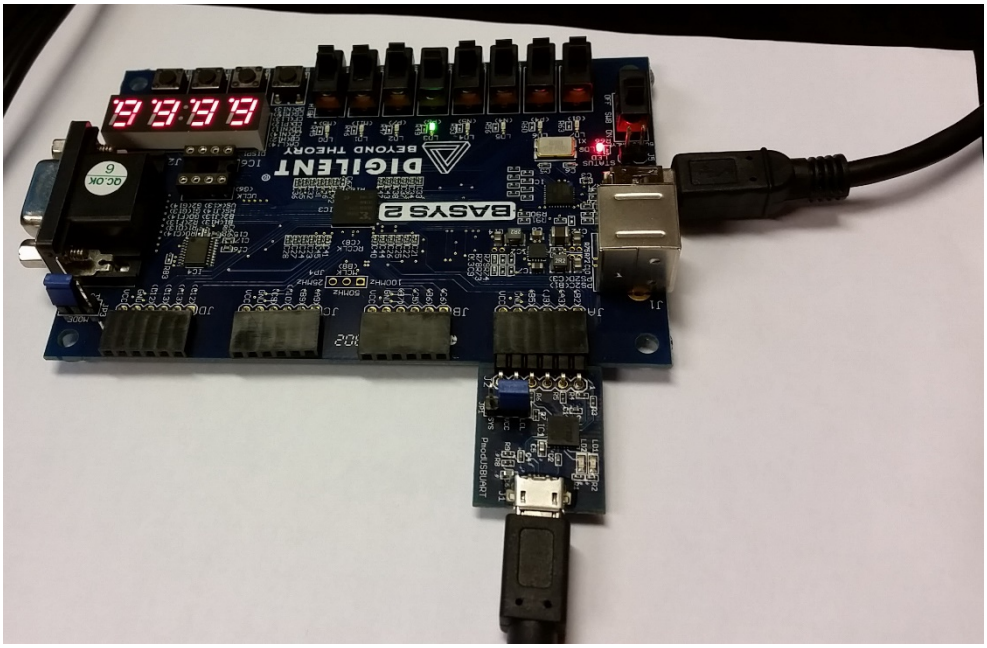


Figure 11.4: Pmod USB-UART connection to the FPGA board

Use the USB-Mini USB cable to power the board and the USB-Micro USB cable for serial data communication.

Download and open the HTERM terminal program. Alternatively, you can use the hyper-terminal software available in windows or download any other terminal software known to you / available on the web.

You need to define the RX (input) and TX (output) ports into your “test_env” project and in the UCF file. Use your board’s reference manual to locate the correct pin numbers. Attention: The TX of the FPGA board is the RX of the Pmod USB-UART module and the RX of the FPGA board is the TX of the Pmod USB-UART module.

11.3.2. Serial Transmit FSM

Design a baud rate generator that would ensure a 9600 baud rate (9600 bits per second) communication over the serial cable. Use a counter to generate the BAUD_ENable signal (generate a '1' every bit time interval).

Baud rate generation:

- For 25 MHz, clock period ~40 ns, input clock must be divided by 2604.
- For 50 MHz, clock period ~20 ns, input clock must be divided by 5208.
- For 100 MHz, clock period ~10 ns, input clock must be divided by 10416.

Define a new entity for the transmission FSM. The next figure presents the ports of this entity.

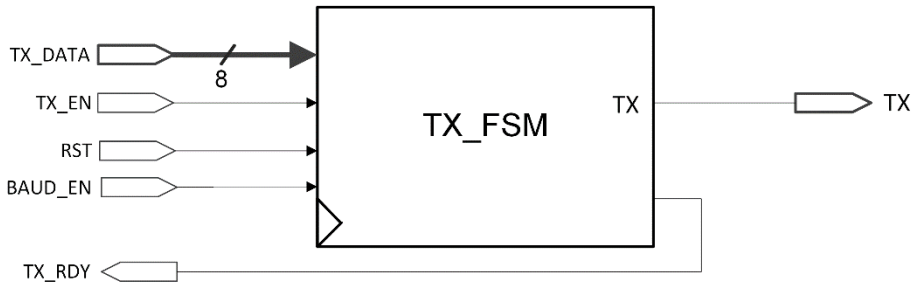


Figure 11.5: TX_FSM Entity Description

The detailed FSM implementation is presented in the figure below. A state transition is triggered only in the clock cycle when BAUD_ENable is '1'. This ensures that a bit will be valid for the baud rate period. The BIT_CNT is a signal with the functionality of a counter inside the FSM; it holds the current transmitting bit value. It should be incremented in the bit state and should be reset after each serial transfer (you can do that in the idle state, or in all states except the bit state).

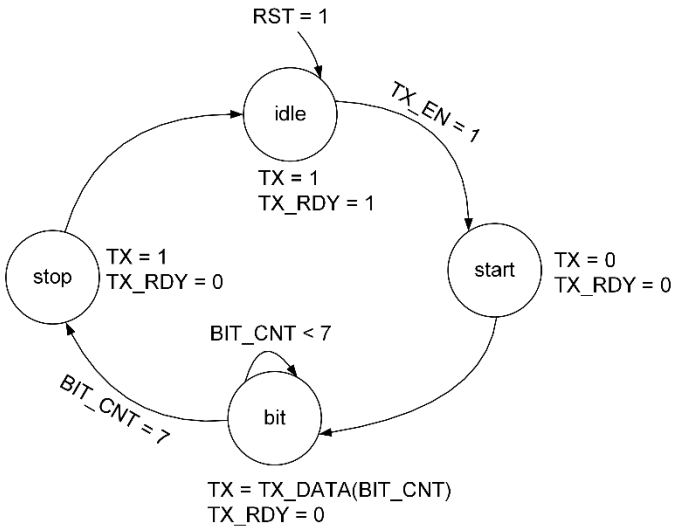


Figure 11.6: TX_FSM Implementation

Write the VHDL code and implement in the "test_env" project the TX_FSM state machine. Use a FSM with 2 or 3 processes (see appendix 6). Test the communication between the FPGA board and the PC. The parameters of the serial communication are: 1 start bit, 8 data bits, 1 stop bit, no parity bit, 9600 baud rate. Make sure that these settings are also configured in the HTERM / hyper-terminal application.

In order to test the serial transmission from the FPGA board to the PC, connect the TX_DATA input to the switches, the TX_EN signal to a MPG enable, RST to '0' or another MPG enable. Make sure that the switches show a valid ASCII code.

Define the correct methodology of asserting the TX_EN signal in order to initiate a single serial data transfer (use a D flip-flop with a set and a reset).

11.3.3. I/O from the MIPS CPU

Connect the TX_FSM into your own MIPS processor implementation. At this point, you are allowed to use your finished and complete processor (single-cycle or pipeline).

You have to send 16-bits of data from your MIPS processor to the PC. Depending on the result of your program, define what field you will send (register with the final result, memory location, etc.).

Example:
When your program has finished execution the result is in R7 and the PC is 0x0020. Add a new instruction to your program: addi R7, R7, 0. Define a 16-bit register whose value will be written from the RD1/ALURes signal, write it in this register (write enable with the value of the PC) and initiate the serial transfer.

Remember that when sending over the serial line the 8-bits represent an ASCII character, hence you are required to make 4 transfers in order to send the alphanumerical encoding of the 4 x 4-bit hexadecimal value (use a decoder/ROM to generate the 8-bit ASCII representation for a hexadecimal value).

Define the methodology to send the 16-bit data over the serial line. Use the TX_RDY signal to control the 4 serial transfers.

11.4. References

[1] XST User Guide
[2] Digilent Basys Board – Reference Manual
[3] Digilent Basys 2 Board – Reference Manual
[4] Digilent Pmod USB-UART – Reference Manual
[5] http://www.asciitable.com/
[6] http://www.der-hammer.info/terminal/

Laboratory 12

12. Finite State Machines and Serial Communication (2)

12.1. Objectives

Study, design, implement and test

- **Finite State Machines**
- **Serial Communication**

Familiarize the students with

- Xilinx® ISE Webpack
- Digilent Development Boards (DDB)
  - **Digilent Basys Board – Reference Manual**
  - **Digilent Basys 2 Board – Reference Manual**
  -

12.2. Theoretical Background

Oversampling mechanism for UART Receive

When transmitting a byte, the UART first sends a START BIT followed by the data (general 8 bits, but could be 5, 6, 7, or 8 bits), followed by STOP BITs. The sequence is repeated for each byte sent.

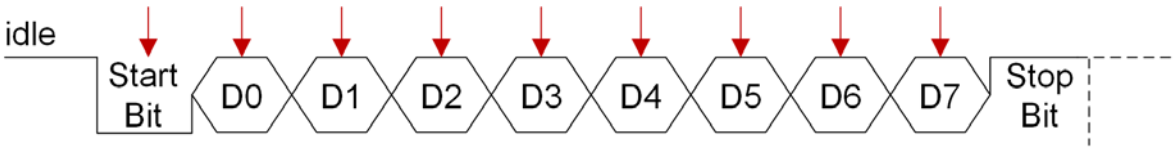


Figure 12.1: Timing Diagram for serial transmission (8-bit Data Example). The red arrows indicate when the bits of data should be read at the receiver.

Serial transmission does not involve a clock signal. The information is included in the baud rate (**number of bits per second**). Common baud rates are 2400, 4800, 9600 and 19200. This means that a bit transmitted through the serial line is valid for a given time period (the inverse of the baud rate). More details on the transmission over the serial line can be found in the previous laboratory.

When receiving a UART packet, one must read (sample) the input signal and extract the data bits sent over the serial line bit by bit. At a first glance, the sample rate for the receiver should coincide with the sample rate (baud rate) of the transmitter; i.e. the rate at which the data was sent. However this is WRONG and can yield in bad transfers at the receiver end, due to imperfect synchronizations (the receiver and the transmitter are in two different clock domains, the baud rate is generated independent at the receiver and the transmitter, asynchronous communication – no common clock signal) between the receiver and the transmitter (double reading the same bit, missing the start bit, not reading the first bit and reading the sign bit, etc.). The frequency at which such events can occur depends on the difference between the sampling rates of the transmitter and receiver. Even if the differences would be very small at a significant number of samplings for successive bits, the error in communication can occur. For example, a difference of 0.1% between the two sampling rates, when transmitting 1000 bits the error appears once. When we perform the serial transfer with 10-bits per character (1 start bit, 8 data bits and 1 stop bit) it results that one character from 100 will be falsely received.

This problem is tackled using oversampling: the input receive signal is read (sampled) at a higher rate than the one used at the transmitter. This permits the detection of the middle of the start bit interval, thus allowing the data bits to be read approximately in the middle of the bit interval, thus eliminating the risk of gaps and receiving false data. For each new character, the middle of the start bit will be determined so this is the only synchronization mechanism used between the receiver and the transmitter.

Oversampling rates are multiple of the transmitter baud-rate: 2, 4, 8, etc. The most usual oversampling rate is 16 times the baud rate of the sender. Each bit that is received over the serial line is sampled (read) 16 times, but only one of the samples is saved (the middle one). The maximum delay for detecting the start bit is 1/16 from the bit interval.

Any UART circuit contains a shift register that is used for converting the received serial data into its parallel form.

12.3. Laboratory Assignments

12.3.1. Serial Receive FSM

Design a baud rate generator that would ensure a 9600 baud rate (9600 bits per second) communication over the serial cable. Use a counter to generate the BAUD_ENable signal (generate a ‘1’ every bit time interval). For the serial receive communication you need to implement an oversampling mechanism of 16.

Baud rate generation for oversampling of 16:

- For 25 MHz, clock period ~ 40 ns, input clock must be divided by ~ 163.

- For 50 MHz, clock period ~ 20 ns, input clock must be divided by ~ 326.
- For 100 MHz, clock period ~ 10 ns, input clock must be divided by ~ 651.

Define a new entity for the receive FSM. The next figure presents the ports of this entity.

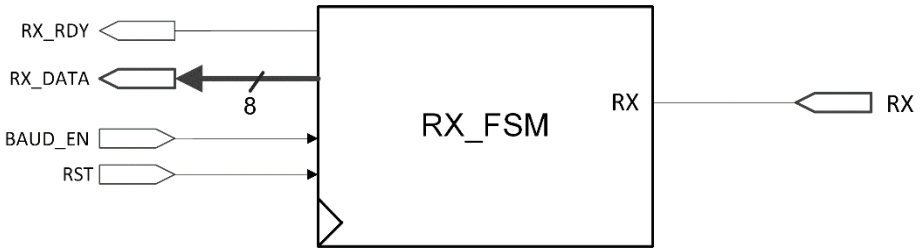


Figure 12.2: RX_FSM Entity Description

The detailed FSM implementation is presented in the figure below. A state transition is triggered only in the clock cycle when BAUD_ENable is '1'. This ensures that a bit will be valid for the baud rate period.

For the RX_FSM you have to use two auxiliary counters: BAUD_CNT and BIT_CNT.

The BIT_CNT is similar to the one in the TX_FSM, i.e. a signal with the functionality of a counter inside the RX_FSM; it holds the current transmitting bit number. It should be incremented in the bit state and should be reset after each serial transfer (you can do that in the idle state, or in all states except the bit state).

The BAUD_CNT is a signal is a signal with the functionality of a counter inside the RX_FSM; it counts the number of BAUD_ENables in order to ensure a correct oversampling mechanism. Remember that you use an oversampling factor of 16.

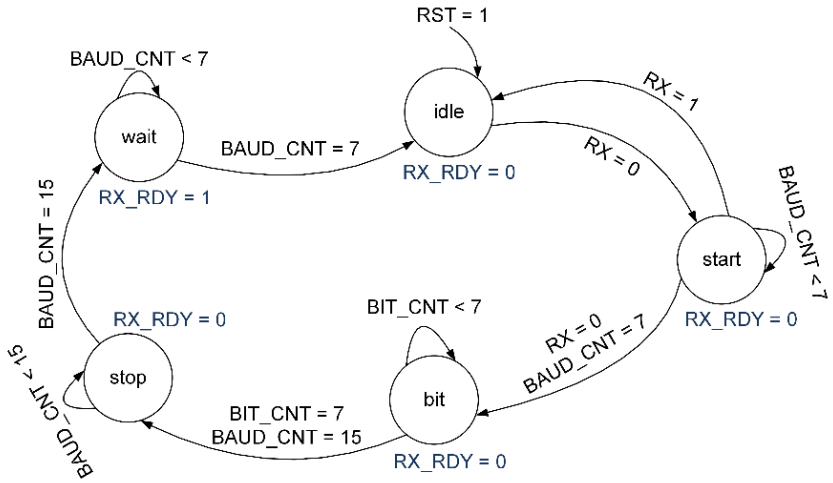


Figure 12.3: RX_FSM Implementation

Write the VHDL code and implement in the “test_env” project the RX_FSM state machine. Use a FSM with 2 or 3 processes (see appendix 6, laboratory 11). You also have to implement a shift register in order to receive the correct data from the serial input line. The RX signal will be shifted in this shift register only once per bit interval; i.e. in the middle of the transmitting interval. Test the communication between the FPGA board and the PC. The parameters of the serial communication are: 1 start bit, 8 data bits, 1 stop bit, no parity bit, 9600 baud rate. Make sure that these settings are also configured in the HTERM / hyper-terminal application. You have to identify the serial port where the module is connected – exactly like in the previous lab.

In order to test the serial transmission from the computer to the FPGA board, connect the RX_DATA output to the SSD (2 digits), RST to ‘0’ or a MPG enable signal. On the SSD, you will see the 8-bit ASCII code representation of the characters that you are sending from the PC.

12.3.2. I/O from the MIPS CPU – optional

Connect the RX_FSM into your own MIPS processor implementation. At this point, you are allowed to use your finished and complete processor (single-cycle or pipeline).

You have to receive 16-bits of valid data from the computer and feed this data into your MIPS processor. Depending on your program you can define what fields will be written with the data coming from the computer (register from the Register File, Data Memory location or even the Instructions from the Instruction Memory).

Remember that when receiving data from the serial RX line the 8-bits from a data transfer represent an ASCII character, hence you are required to make 4 transfers in order to receive the alphanumerical encoding of the 4 x 4-bit hexadecimal value (use a decoder/ROM to generate the 4-bit hexadecimal data and then concatenate 4 receive transfers in order to obtain the correct 16-bit data that will be fed to your processor).

Define the methodology to receive the 16-bit data over the serial RX line. Use the RX_RDY signal to control the writing of the data into your processor.

12.4. References

[1] XST User Guide
[2] Digilent Basys Board – Reference Manual
[3] Digilent Basys 2 Board – Reference Manual
[4] Digilent Pmod USB-UART – Reference Manual
[5] http://www.asciitable.com/
[6] http://www.der-hammer.info/terminal/

A. Appendix 1 – ISE Quick Start Tutorial

ISE Quick Start Tutorial, adapted to ISE 14.7

Starting the ISE Software

Double click the desktop icon, or go to **Start→Programs→ Xilinx Design Tools → ISE Design Suite 14.7 → ISE Design Tools → Project Navigator**

Attention (!) Be careful to use the latest version of ISE not the 9.2i version that may be installed on the computers from the lab.

Accessing Help

At any time during the tutorial, you can access online help for additional information about the ISE software and related tools.
To open Help, do either of the following:

- Press **F1** to view Help for the specific tool or function that you have selected or highlighted.
- Launch the **ISE Help Contents** from the Help menu. It contains information about creating and maintaining your complete design flow in ISE.

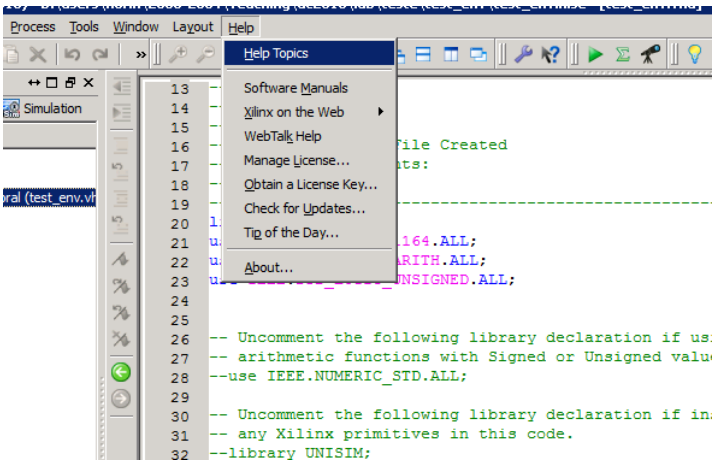


Figure A.1: ISE Help Topics

Create a New Project

Create a new ISE project, which will target the FPGA device on the Basys development board, Spartan 3E.
To create a new project:

1. Select **File → New Project...** The New Project Wizard appears.
2. Type **test_env** in the Entity Name field.

3. Enter or browse to a location (directory path) for the new project (remember the laboratory rules). A test_env subdirectory is created automatically.
4. Verify that **HDL** is selected from the Top-Level Source Type list.
5. Click **Next** to move to the device properties page.
6. Fill in the properties in the table as shown below:
   - Product Category: **All**
   - Family: **Spartan 3E**
   - Device: **XC3S100E**
   - Package: **TQ144 (for Basys) / CP132 (for Basys 2)**
   - Speed Grade: **-4**
   - Top-Level Module Type: **HDL**
   - Synthesis Tool: **XST (VHDL/Verilog)**
   - Simulator: **ISim (VHDL/Verilog)**
   - Preferred language: **VHDL**
   - Leave the default values in the remaining fields.

Creating a VHDL Source

Create a VHDL source file for the project as follows:

1. Click the menu **Project/New Source**.
2. Select VHDL Module as the source type.
3. Type in the file name that you want to create. For example “test_env”.
4. Verify that the Add to project checkbox is selected.
5. Click Next.
6. Declare the ports for your design by filling in the port information as in the following figure. **These ports are particularly defined for the Basys board, being enough for the majority of the laboratory designs for this semester.**

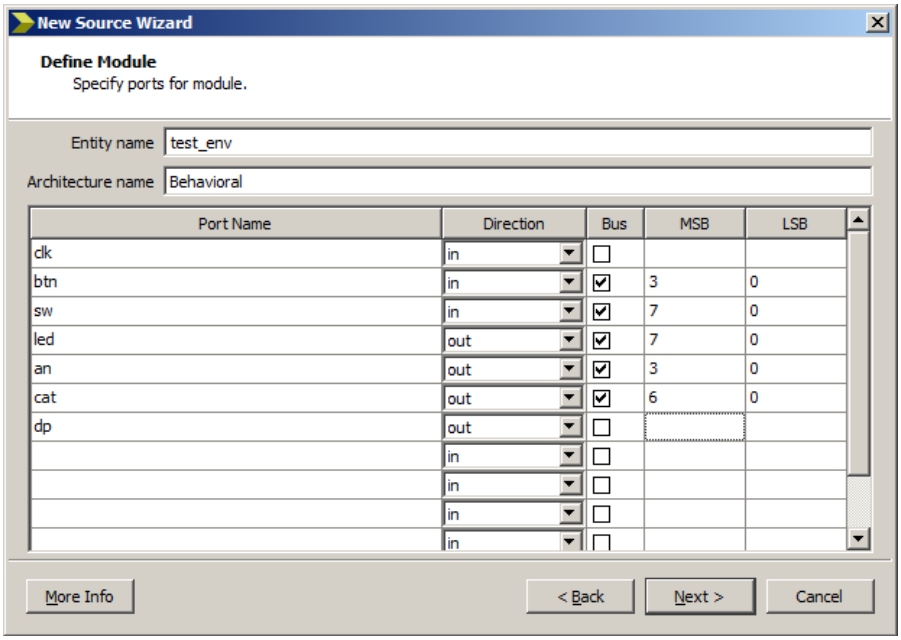


Figure A.2 Port definition through the Xilinx Interface

7. Click **Next** (re-verify the summary of the port declarations), and then **Finish** to complete the new source file template.

The source file containing the entity *test_env* and its architecture is displayed in the ISE environment, and in the **Hierarchy** tab appears as **Top Module** for the current design.

Remember, in projects containing multiple source files, if one accidentally changes the top module entity, you can reset it as a top module by right click on a source in **Hierarchy**, and select **Set as Top Module**.

Attention the parent of the test_env entity in the hierarchy is formed by the properties of the FPGA target device. For Basys, one must see **xc3s100e-4tq144.** If it does not coincide, this means that you have probably skipped step 6 from **Creating a VHDL Source**. Double click on the parent and enter the target device properties.

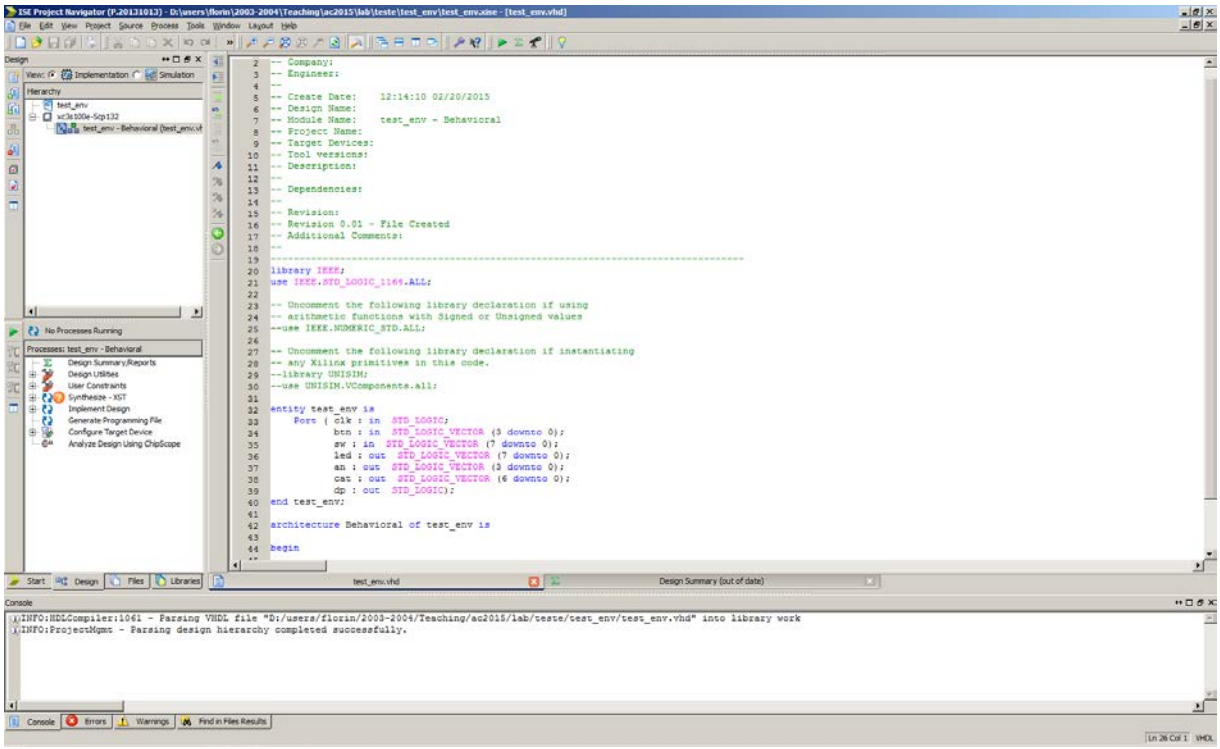


Figure A.3: The new ISE Project

Make shore that the following libraries are included in the source file header:

```
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

If they are missing from any VHDL file, include them!

Using Language Templates (VHDL) – optional (*You will probably have to use this in the future…*)

Language Templates includes VHDL synthesizable examples that you can use in your designs. The "Light Bulb" takes you directly to Language Templates tab" or you can do the following"

1. Place the cursor under the begin statement of your architecture.
2. Open Language Templates by selecting the menu **Edit → Language Templates…**
3. Navigate in the hierarchy "+", to the coding examples:
   VHDL → Synthesis Constructs → Coding Examples → …
4. Select the desired component in the hierarchy, then right click → **Use in File**. This step will copy the model code to your source file at the place of your cursor.
5. Close the **Language Templates**.
6. Change the signal names so that they will match the signals in your entity.

Editing the VHDL Source Code

1. Add component and/or signal declarations between the architecture and the begin statements.
2. Add the rest of the code (component instantiation, behavioral description, etc.) after the begin statement and before the end statement.
3. For the first example add the following statements after begin.

```
led <= sw;
an  <= btn;
cat <= (others=>'0');
dp  <= '0';
```

4. Save the file by selecting **File → Save or Ctrl + S**.
5. Select the top-level entity in the Hierarchy tab: *test_env*.
6. Verify that your VHDL syntax is correct: in the **Processes** zone: **Synthesize – XST → Check Syntax → Run**
7. Correct the errors if they appear in the bottom part of the ISE environment. Start from the top with the first error.
8. Synthesize your design: double click **Synthesize – XST**
9. View the resulting circuit: double click **Synthesize – XST → View RTL Schematic.** In the next dialog be sure to select the second variant (*Start with a schematic of the top-level block*), press **OK**. The top-level entity will appear. Double click to view its internal organization. **You should recognize at least a part of the declared entity.** This is a first method to verify that your code is correct and implements the desired circuits.

You have now created the VHDL source for the "test_env" project with no errors.
Note: You can also create a UCF file for your project by selecting **Project → Create New Source**.

Assigning Pin Location Constraints

Specify the pin locations for the ports of the design so that they are connected correctly on the DDB. You can Edit the User Constraints File (*.ucf) manually (**Users Constraints → Edit Constraints (Text)**). You can find the user constraints file for the Basys Board here and for the Basys 2 Board here. Download the file and add it to your design. Open the constraints file and see the syntax for every port (net).

For the future, you can add new ports to the constraints file.

Implement Design and Verify Constraints

Implement the design and verify that it meets all constraints.

1. Double-click the **Implement Design** process in the Processes tab.
2. Notice that after Implementation is complete, the Implementation processes have a green check mark next to them indicating that they completed successfully without Errors or Warnings. If there are errors or warnings, you can correct them.
3. Open **Design Summary/Reports**. Analyze the reports of your design (Summary, Timing Constraints, etc.). In the next designs, these reports will be relevant.

Generate Programming File

1. Before generating the programming file, you must set the start-up clock option to JTAG clock: **Generate Programming File → Properties → Startup Options → FPGA Start-Up Clock → JTAG Clock.**
2. Generate the programming file: double click **Generate Programming File.** The bit file for the DDB configuration is created.

If you notice that one or more processes have an orange question mark next to them, it indicates that they are out-of-date with one or more of the design files. You will have to re-run these processes.

If there are no errors at this time the file “test_env.bit” should be in the project folder.

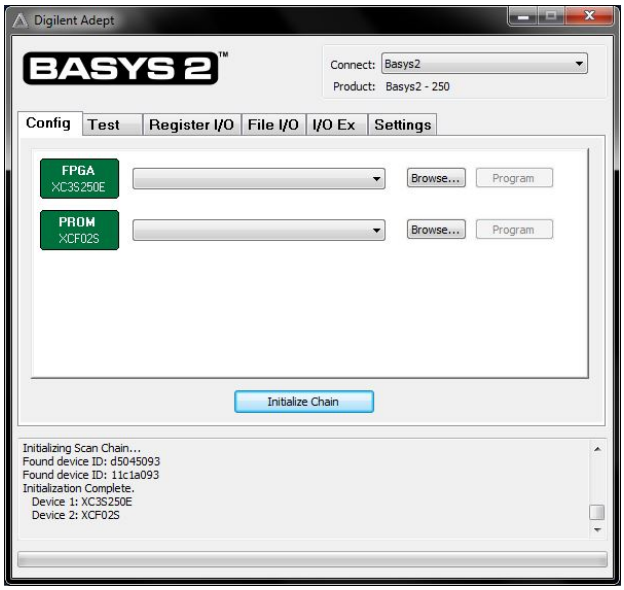


Figure A.4: Digilent Adept tool

Download Design to the Spartan™-3E Demo Board → Basys

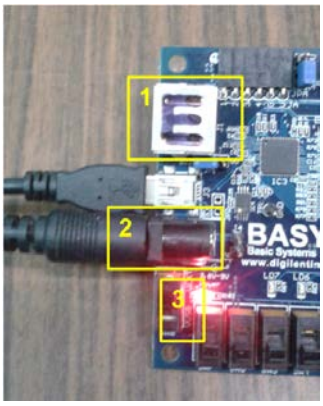
If you encounter problems during the programming of the board please go to the end of this tutorial (after the following figure).

1. Connect the Basys board to the USB port.
2. Start the Adept Tool from Adept programming software: Start → Programs → Digilent → Adept (Figure A.4).
3. Press the Initialize Chain button
4. Browse for the project's bit file.
5. Program the FPGA device.

Possible problems when connecting the board and solutions:

Problem: The Basys board is not recognized.
Solutions (Start in order and restart Adept after every fail, ask the TA to assist you):

a) Try a different USB port (front or rear of the computer). If a driver install process initiates, call your TA. You will need administrative privileges.



b) Verify that the board does not require external power.
   - If it does not require external power (no "E" sign – see the figure below position 1), make shore that the switch (Position 3 in the image) is in the VUSB position
   - If it requires external power, do the following
     - ➢ Use a 3.3 V power supply in the external power socket (position 2 in the image)
     - ➢ Move the switch (position 3 in the image) to the VEXT position

c) Try a new programming cable
d) Try a new board (report this to your TA)
e) Change the workstation

B. Appendix 2 – Combinational Shifter Implementation

A shifter can also be implemented as a sequence of multiplexers. In such an implementation, the output of one MUX is connected to the input of the next MUX in a way that depends on the shift distance. The number of multiplexers required for an n-bit word is $n * \log_2 n$.

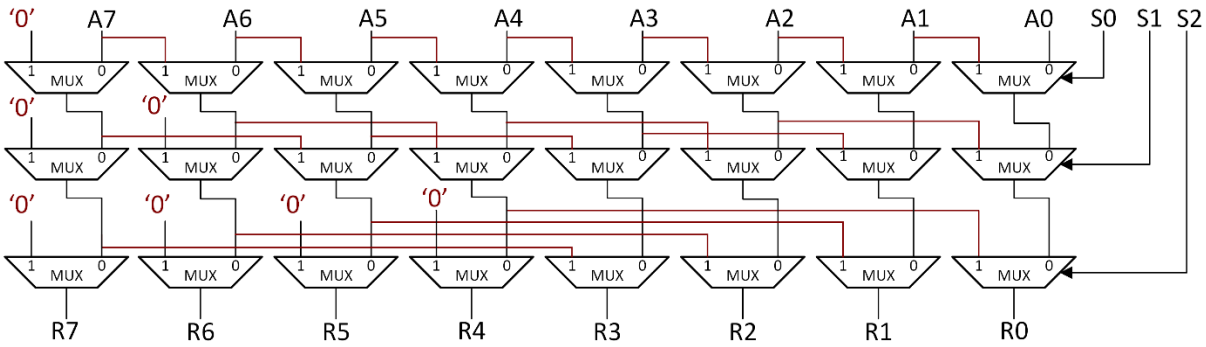


Figure B.1: Multi-level (logarithmic) 8-bit right shifter

Example:

- sw(4 downto 0) is a 5-bit signal that can be shifted left or right arithmetic
- sw(6:5) is the shift amount: 0, 1, 2 or 3 positions
- sw(7) is the shift direction
  - sw(7) = 0 – shift left
  - sw(7) = 1 – shift right arithmetic
- the result is displayed on the LEDs from the Basys board

The code in VHDL implemented with two processes:

```
process(sw)
begin
   if sw(5) = '1' then    -- shift with 1 position
      if sw(7) = '0' then
         shift1 <= sw(3 downto 0) & '0';           -- shift left
      else
         shift1 <= sw(4) & sw(4 downto 1);       -- shift right arithmetic
      end if;
   else
      shift1 <= sw(4 downto 0);
   end if;
end process;

process(sw, shift1)
begin
   if sw(6) = '1' then    -- shift with 2 position
      if sw(7) = '0' then
         shift2 <= shift1(2 downto 0) & "00";                    -- shift left
      else
         shift2 <= shift1(4) & shift1(4) & shift1(4 downto 2);   -- shift right arithmetic
      end if;
   else
      shift2 <= shift1;
   end if;
end process;

led <= shift2;
```

C. Appendix 3 – Register File Implementation

```
entity reg_file is
	port (
		clk	: in std_logic;
		ra1	: in std_logic_vector (2 downto 0);
		ra2	: in std_logic_vector (2 downto 0);
		wa	: in std_logic_vector (2 downto 0);
		wd	: in std_logic_vector (7 downto 0);
		wen	: in std_logic;
		rd1	: out std_logic_vector (7 downto 0);
		rd2	: out std_logic_vector (7 downto 0)
	);
end reg_file;

architecture Behavioral of reg_file is

	type reg_array is array (0 to 7) of std_logic_vector(7 downto 0);
	signal reg_file : reg_array;

begin
	process(clk)
	begin
		if rising_edge(clk) then
			if wen = '1' then
				reg_file(conv_integer(wa)) <= wd;
			end if;
		end if;
	end process;

	rd1 <= reg_file(conv_integer(ra1));
	rd2 <= reg_file(conv_integer(ra2));

end Behavioral;
```

D. Appendix 4 – RAM Implementation

The following example is a RAM with “no change” policy

```
entity rams_no_change is
        port (  clk     : in std_logic;
                we      : in std_logic;
                en      : in std_logic;
                addr    : in std_logic_vector(7 downto 0);
                di      : in std_logic_vector(15 downto 0);
                do      : out std_logic_vector(15 downto 0));
end rams_no_change;

architecture syn of rams_no_change is

        type ram_type is array (0 to 255) of std_logic_vector (15 downto 0);
        signal RAM: ram_type;
begin

        process (clk)
        begin
                if clk'event and clk = '1' then
                        if en = '1' then
                                if we = '1' then
                                        RAM(conv_integer(addr)) <= di;
                                else
                                        do <= RAM( conv_integer(addr));
                                end if;
                        end if;
                end if;
        end process;
end syn;
```

E. Appendix 5 – MIPS Instruction Reference

Note: ALL immediate values should be sign extended.
Exception: For logical operations immediate values should be zero extended.
After extensions, you treat them as signed or unsigned 32-bit numbers.

For the non-immediate instructions, the only difference between signed and unsigned instructions (ex ADD vs. ADDU) is that signed instructions can generate an overflow.

The instruction formats are given, you can figure out the binary instruction codes. The instruction descriptions are given below. Additional details can be found here: "MIPS Single Cycle Processor", John Alexander, Barret Schloerke, Daniel Sedam, Iowa State University

ADD – Add

Description:	Adds two registers and stores the result in a register
Operation:	$d ← $s + $t; advance_pc (4);
Syntax:	add $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0000

ADDI – Add immediate

Description:	Adds a register and a signed immediate value and stores the result in a register
Operation:	$t ← $s + imm; advance_pc (4);
Syntax:	addi $t, $s, imm
Encoding:	0010 00ss ssst tttt iiii iiii iiii iiii

ADDIU – Add immediate unsigned

Description:	Adds a register and an unsigned immediate value and stores the result in a register
Operation:	$t ← $s + imm; advance_pc (4);
Syntax:	addiu $t, $s, imm
Encoding:	0010 01ss ssst tttt iiii iiii iiii iiii

ADDU – Add unsigned

Description:	Adds two registers and stores the result in a register
Operation:	$d ← $s + $t; advance_pc (4);
Syntax:	addu $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0001

AND – Bitwise and

Description:	Bitwise ands two registers and stores the result in a register
Operation:	$d ← $s & $t; advance_pc (4);
Syntax:	and $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0100

ANDI – Bitwise and immediate

Description:	Bitwise ands a register and an immediate value and stores the result in a register
Operation:	$t ← $s & imm; advance_pc (4);
Syntax:	andi $t, $s, imm
Encoding:	0011 00ss ssst tttt iiii iiii iiii iiii

BEQ – Branch on equal

Description:	Branches if the two registers are equal
Operation:	if $s == $t advance_pc (offset << 2); else advance_pc (4);
Syntax:	beq $s, $t, offset
Encoding:	0001 00ss ssst tttt iiii iiii iiii iiii

BGEZ – Branch on greater than or equal to zero

Description:	Branches if the register is greater than or equal to zero
Operation:	if $s >= 0 advance_pc (offset << 2); else advance_pc (4);
Syntax:	bgez $s, offset
Encoding:	0000 01ss sss0 0001 iiii iiii iiii iiii

BGEZAL – Branch on greater than or equal to zero and link

Description:	Branches if the register is greater than or equal to zero and saves the return address in $31
Operation:	if $s >= 0 $31 = PC + 8 (or nPC + 4); advance_pc (offset << 2); else advance_pc (4);
Syntax:	bgezal $s, offset
Encoding:	0000 01ss sss1 0001 iiii iiii iiii iiii

BGTZ – Branch on greater than zero

Description:	Branches if the register is greater than zero
Operation:	if $s > 0 advance_pc (offset << 2); else advance_pc (4);
Syntax:	bgtz $s, offset
Encoding:	0001 11ss sss0 0000 iiii iiii iiii iiii

BLEZ – Branch on less than or equal to zero

Description:	Branches if the register is less than or equal to zero
Operation:	if $s <= 0 advance_pc (offset << 2)); else advance_pc (4);
Syntax:	blez $s, offset
Encoding:	0001 10ss sss0 0000 iiii iiii iiii iiii

BLTZ – Branch on less than zero

Description:	Branches if the register is less than zero
Operation:	if $s < 0 advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bltz $s, offset
Encoding:	0000 01ss sss0 0000 iiii iiii iiii iiii

BLTZAL – Branch on less than zero and link

Description:	Branches if the register is less than zero and saves the return address in $31
Operation:	if $s < 0 $31 = PC + 8 (or nPC + 4); advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bltzal $s, offset
Encoding:	0000 01ss sss1 0000 iiii iiii iiii iiii

BNE – Branch on not equal

Description:	Branches if the two registers are not equal
Operation:	if $s != $t advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bne $s, $t, offset
Encoding:	0001 01ss ssst tttt iiii iiii iiii iiii

DIV – Divide

Description:	Divides $s by $t and stores the quotient in $LO and the remainder in $HI
Operation:	$LO ← $s / $t; $HI ← $s % $t; advance_pc (4);
Syntax:	div $s, $t
Encoding:	0000 00ss ssst tttt 0000 0000 0001 1010

DIVU – Divide unsigned

Description:	Divides $s by $t and stores the quotient in $LO and the remainder in $HI
Operation:	$LO ← $s / $t; $HI ← $s % $t; advance_pc (4);
Syntax:	divu $s, $t
Encoding:	0000 00ss ssst tttt 0000 0000 0001 1011

J – Jump

Description:	Jumps to the calculated address
Operation:	PC ← nPC; nPC = (PC & 0xf0000000) | (target << 2);
Syntax:	j target
Encoding:	0000 10ii iiii iiii iiii iiii iiii iiii

JAL – Jump and link

Description:	Jumps to the calculated address and stores the return address in $31
Operation:	$31 ← PC + 8 (or nPC + 4); PC = nPC; nPC = (PC & 0xf0000000) | (target << 2);
Syntax:	jal target
Encoding:	0000 11ii iiii iiii iiii iiii iiii iiii

JR – Jump register

Description:	Jump to the address contained in register $s
Operation:	PC ← nPC; nPC = $s;
Syntax:	jr $s
Encoding:	0000 00ss sss0 0000 0000 0000 0000 1000

LB – Load byte

Description:	A byte is loaded into a register from the specified address.
Operation:	$t ← MEM[$s + offset]; advance_pc (4);
Syntax:	lb $t, offset($s)
Encoding:	1000 00ss ssst tttt iiii iiii iiii iiii

LUI – Load upper immediate

Description:	The immediate value is shifted left 16 bits and stored in the register. The lower 16 bits are zeroes.
Operation:	$t ← (imm << 16); advance_pc (4);
Syntax:	lui $t, imm
Encoding:	0011 11-- ---t tttt iiii iiii iiii iiii

LW – Load word

Description:	A word is loaded into a register from the specified address.
Operation:	$t ← MEM[$s + offset]; advance_pc (4);
Syntax:	lw $t, offset($s)
Encoding:	1000 11ss ssst tttt iiii iiii iiii iiii

MFHI – Move from HI

Description:	The contents of register HI are moved to the specified register.
Operation:	$d ← $HI; advance_pc (4);
Syntax:	mfhi $d
Encoding:	0000 0000 0000 0000 dddd d000 0001 0000

MFLO – Move from LO

Description:	The contents of register LO are moved to the specified register.
Operation:	$d ← $LO; advance_pc (4);
Syntax:	mflo $d
Encoding:	0000 0000 0000 0000 dddd d000 0001 0010

MULT – Multiply

Description:	Multiplies $s by $t and stores the result in $Hi and $LO.
Operation:	$Hi, $LO ← $s * $t; advance_pc (4);
Syntax:	mult $s, $t
Encoding:	0000 00ss ssst tttt 0000 0000 0001 1000

MULTU – Multiply unsigned

Description:	Multiplies $s by $t and stores the result in $Hi and $LO.
Operation:	$Hi, $LO ← $s * $t; advance_pc (4);
Syntax:	multu $s, $t
Encoding:	0000 00ss ssst tttt 0000 0000 0001 1001

NOOP – no operation

Description:	Performs no operation.
Operation:	advance_pc (4);
Syntax:	noop
Encoding:	0000 0000 0000 0000 0000 0000 0000 0000

Note: The encoding for a NOOP represents the instruction SLL $0, $0, 0 which has no side effects. In fact, nearly every instruction that has $0 as its destination register will have no side effect and can thus be considered a NoOP instruction.

OR – Bitwise or

Description:	Bitwise logical ors two registers and stores the result in a register
Operation:	$d ← $s | $t; advance_pc (4);
Syntax:	or $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0101

ORI – Bitwise or immediate

Description:	Bitwise ors a register and an immediate value and stores the result in a register
Operation:	$t ← $s | imm; advance_pc (4);
Syntax:	ori $t, $s, imm
Encoding:	0011 01ss ssst tttt iiii iiii iiii iiii

SB – Store byte

Description:	The least significant byte of $t is stored at the specified address.
Operation:	MEM[$s + offset] ← (0xff & $t); advance_pc (4);
Syntax:	sb $t, offset($s)
Encoding:	1010 00ss ssst tttt iiii iiii iiii iiii

SLL – Shift left logical

Description:	Shifts a register value left by the shift amount listed in the instruction and places the result in a third register. Zeroes are shifted in.
Operation:	$d ← $t << h; advance_pc (4);
Syntax:	sll $d, $t, h
Encoding:	0000 00ss ssst tttt dddd dhhh hh00 0000

SLLV – Shift left logical variable

Description:	Shifts a register value left by the value in a second register and places the result in a third register. Zeroes are shifted in.
Operation:	$d ← $t << $s; advance_pc (4);
Syntax:	sllv $d, $t, $s
Encoding:	0000 00ss ssst tttt dddd d--- --00 0100

SLT – Set on less than (signed)

Description:	If $s is less than $t, $d is set to one. It gets zero otherwise.
Operation:	if $s < $t $d ← 1; advance_pc (4); else $d ← 0; advance_pc (4);
Syntax:	slt $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 1010

SLTI – Set on less than immediate (signed)

Description:	If $s is less than immediate, $t is set to one. It gets zero otherwise.
Operation:	if $s < imm $t ← 1; advance_pc (4); else $t ← 0; advance_pc (4);
Syntax:	slti $t, $s, imm
Encoding:	0010 10ss ssst tttt iiii iiii iiii iiii

SLTIU – Set on less than immediate unsigned

Description:	If $s is less than the unsigned immediate, $t is set to one. It gets zero otherwise.
Operation:	if $s < imm $t ← 1; advance_pc (4); else $t ← 0; advance_pc (4);
Syntax:	sltiu $t, $s, imm
Encoding:	0010 11ss ssst tttt iiii iiii iiii iiii

SLTU – Set on less than unsigned

Description:	If $s is less than $t, $d is set to one. It gets zero otherwise.
Operation:	if $s < $t $d ← 1; advance_pc (4); else $d ← 0; advance_pc (4);
Syntax:	sltu $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 1011

SRA – Shift right arithmetic

Description:	Shifts a register value right by the shift amount (shamt) and places the value in the destination register. The sign bit is shifted in.
Operation:	$d ← $t >> h; advance_pc (4);
Syntax:	sra $d, $t, h
Encoding:	0000 00-- ---t tttt dddd dhhh hh00 0011

SRL – Shift right logical

Description:	Shifts a register value right by the shift amount (shamt) and places the value in the destination register. Zeroes are shifted in.
Operation:	$d ← $t >> h; advance_pc (4);
Syntax:	srl $d, $t, h
Encoding:	0000 00-- ---t tttt dddd dhhh hh00 0010

SRLV – Shift right logical variable

Description:	Shifts a register value right by the amount specified in $s and places the value in the destination register. Zeroes are shifted in.
Operation:	$d ← $t >> $s; advance_pc (4);
Syntax:	srlv $d, $t, $s
Encoding:	0000 00ss ssst tttt dddd d000 0000 0110

SUB – Subtract

Description:	Subtracts two registers and stores the result in a register
Operation:	$d ← $s - $t; advance_pc (4);
Syntax:	sub $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0010

SUBU – Subtract unsigned

Description:	Subtracts two registers and stores the result in a register
Operation:	$d ← $s - $t; advance_pc (4);
Syntax:	subu $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0011

SW – Store word

Description:	The contents of $t is stored at the specified address.
Operation:	MEM[$s + offset] ← $t; advance_pc (4);
Syntax:	sw $t, offset($s)
Encoding:	1010 11ss ssst tttt iiii iiii iiii iiii

SYSCALL – System call

Description:	Generates a software interrupt.
Operation:	advance_pc (4);
Syntax:	syscall
Encoding:	0000 00-- ---- ---- ---- ---- --00 1100

XOR – Bitwise exclusive or

Description:	Exclusive ors two registers and stores the result in a register
Operation:	$d ← $s ^ $t; advance_pc (4);
Syntax:	xor $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d--- --10 0110

XORI – Bitwise exclusive or immediate

Description:	Bitwise exclusive ors a register and an immediate value and stores the result in a register
Operation:	$t ← $s ^ imm; advance_pc (4);
Syntax:	xori $t, $s, imm
Encoding:	0011 10ss ssst tttt iiii iiii iiii iiii

F. Appendix 6 – Finite State Machine Implementations

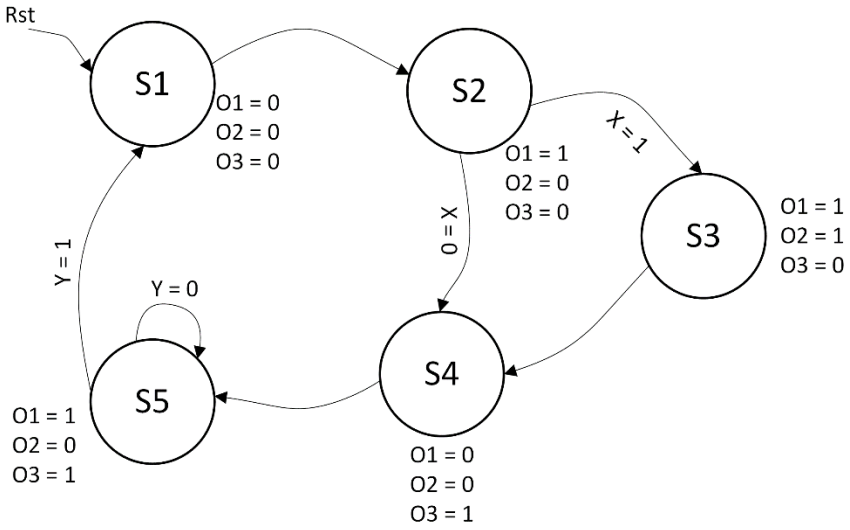


Figure F.1: Finite State Machine Example (XST User Guide)

IO Pins	Description
clk	Positive Edge Clock
Rst	Asynchronous Reset (Active High)
X, Y	FSM Inputs
O1, O2, O3	FSM Outputs

Table F.1: FSM Pin Descriptions

The Xilinx Synthesis technology recognizes Finite State Machines written in VHDL with 1, 2 or 3 processes. A coding example for the Finite State Machine presented in Figure F.1, for each kind of implementation, is given on the next pages. You have to adapt the Finite State Machine implementation to your own FSM description.

VHDL Coding Example: FSM with One Process

```
entity fsm_1 is
        port (
                clk, rst, x, y    : IN std_logic;
                o1, o2, o3        : OUT std_logic
        );
end entity;

architecture beh1 of fsm_1 is
        type state_type is (s1, s2, s3, s4, s5);
        signal state    : state_type ;
begin

        process (clk, rst, x, y)
        begin
                if (rst ='1') then
                        state <=s1;
                        o1<='0'; o2<='0'; o3<='0';
                elsif (clk='1' and clk'event) then
                        case state is
                                when s1 =>  state <= s2;
                                            o1<='1'; o2<='0'; o3<='0';
                                when s2 =>  if x = '1' then
                                                    state <= s3;
                                                    o1<='1'; o2<='1'; o3<='0';
                                            else
                                                    state <= s4;
                                                    o1<='0'; o2<='0'; o3<='1';
                                            end if;
                                when s3 =>  state <= s4;
                                            o1<='0'; o2<='0'; o3<='1';
                                when s4 =>  state <= s5;
                                            o1<='1'; o2<='0'; o3<='1';
                                when s5 =>  if y = ‘1’ then
                                                    state <= s1;
                                                    o1<='0'; o2<='0'; o3<='0';
                                            else
                                                    state <= s5;
                                                    o1<='1'; o2<='0'; o3<='1';
                                            end if;
                        end case;
                end if;
        end process;

end beh1;
```

VHDL Coding Example: FSM with Two Processes

```
entity fsm_2 is
        port (
                clk, rst, x, y    : IN std_logic;
                o1, o2, o3        : OUT std_logic
        );
end entity;

architecture beh1 of fsm_2 is
        type state_type is (s1, s2, s3, s4, s5);
        signal state    : state_type ;
begin

        process1: process (clk, rst, x, y)
        begin
                if (rst ='1') then
                        state <=s1;
                elsif (clk='1' and clk'Event) then
                        case state is
                                when s1 =>   state <= s2;
                                when s2 =>   if x = '1' then
                                                        state <= s3;
                                             else
                                                        state <= s4;
                                             end if;
                                when s3 =>   state <= s4;
                                when s4 =>   state <= s5;
                                when s5 =>   if y = '1' then
                                                        state <= s1;
                                             else
                                                        state <= s5;
                                             end if;
                        end case;
                end if;
        end process process1;


        process2: process (state)
        begin
                case state is
                        when s1 => o1<='0'; o2<='0'; o3<='0';
                        when s2 => o1<='1'; o2<='0'; o3<='0';
                        when s3 => o1<='1'; o2<='1'; o3<='0';
                        when s4 => o1<='1'; o2<='0'; o3<='0';
                        when s5 => o1<='1'; o2<='0'; o3<='1';
```

```
                end case;
        end process process2;
end beh1;
```

VHDL Coding Example: FSM with Three Processes

```
entity fsm_3 is
        port (
                clk, rst, x, y    : IN std_logic;
                o1, o2, o3        : OUT std_logic
        );
end entity;

architecture beh1 of fsm_3 is
        type state_type is (s1, s2, s3, s4, s5);
        signal state, next_state      : state_type ;
begin
        process1: process (clk, rst)
        begin
                if (reset ='1') then
                        state <=s1;
                elsif (clk='1' and clk'Event) then
                        state <= next_state;
                end if;
        end process process1;


        process2 : process (state, x, y)
        begin
                case state is
                        when s1 =>   next_state <= s2;
                        when s2 =>   if x = '1' then
                                             next_state <= s3;
                                     else
                                             next_state <= s4;
                                     end if;
                        when s3 =>   next_state <= s4;
                        when s4 =>   next_state <= s5;
                        when s5 =>   if y = ‘1’ then
                                             next_state <= s1;
                                     else
                                             next_state <= s5;
                                     end if;
                end case;
        end process process2;


        process3 : process (state)
        begin
                case state is
```

```
                    when s1 => o1<='0'; o2<='0'; o3<='0';
                    when s2 => o1<='1'; o2<='0'; o3<='0';
                    when s3 => o1<='1'; o2<='1'; o3<='0';
                    when s4 => o1<='1'; o2<='0'; o3<='0';
                    when s5 => o1<='1'; o2<='0'; o3<='1';
            end case;
    end process process3;
end beh1;
```

G. Appendix 7 – ASCII Codes Table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Figure G.1: ASCII Codes (http://www.asciitable.com/)