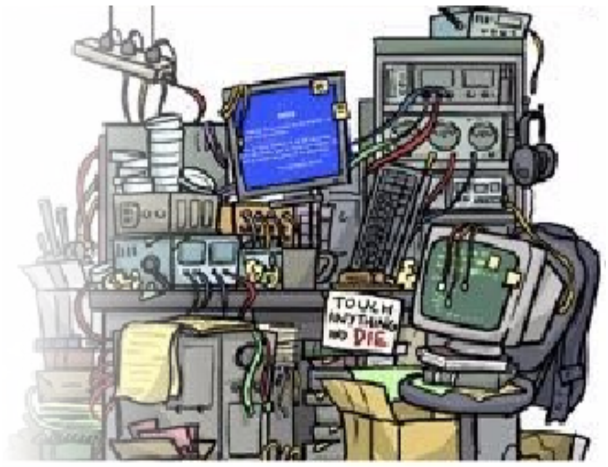
METU EE 446
Computer Architecture
Laboratory

Warming Up
for
Computer Design



Laboratory Work 1 - Warming Up for Computer Design

Objectives

The purpose of the first laboratory work is to construct a Verilog library composed of the fundamental modules to be used throughout the design of a computer. Moreover, simple datapath design is to be practiced through designing simple architectures from the modules in the constructed library to perform some simple tasks.

During this laboratory work, one will be familiar with designing modules with Verilog HDL and constructing architectures with schematic design. This laboratory work is a tool for getting familiar with the software, Quartus and cocotb, to be used throughout the semester. Finally, one will practice embedding the designs to a development board, DE1-SoC Board, equipped with a field programmable gate array (FPGA) and several peripheral units such as switch inputs, general purposed I/O pins, LED and 7-segment outputs etc.

1 Preliminary Work

To fulfill the requirements of this laboratory work, the following tasks should be performed. **Note: You should only include important parts of the code in the pdf report, the full code should be submitted separately.**

1.1 Reading Assignment

The laboratory manual where the regulations and some other useful information exist is available at the ODTUClass course page. Read that manual thoroughly. If you feel yourself unfamiliar with Verilog HDL programming, please refer to the corresponding lecture notes of the EE445 course, which are available on the course page.

1.2 Module Design with Verilog HDL (40% Credits)

For this part, you will implement fundamental modules in Verilog. These modules will be then used to construct more complex modules and computer architectures. More importantly, each module you design is to be added to your module library so that you will make use of them in future laboratory works. Thus, consider this part of the preliminary work as building your own Verilog module library.

For each item in this part, you should submit your design codes seperately from the report. You should also submit the corresponding cocotb test bench codes alongside the makefile for the relevant items. Additionally, remember to attach your explanation for the 2^{nd} item of the ALU design in 1.2.5.

1.2.1 Constant Value Generator (1% Credits)

Implement a module to put a constant value to a data bus. The module should have two configurable parameters one of which is for the data width and the other is for the value. If the data width is specified as W, for example, the module should continuously output the specified constant value to a data bus of W-bit width.

1.2.2 Decoder (2% Credits)

Implement a 2 to 4 and a 4 to 16 decoder.

1.2.3 Multiplexers (2% Credits)

Implement 2 to 1, 4 to 1, and 16 to 1 multiplexers all of which have W-bit data input/outputs, where W is a parameter of the module specifying the data width of the input.

1.2.4 Combinational Shifter (7% Credits)

Implement a W-bit combinational shifter that has 3 inputs, a W-bit data input, a 5-bit shift input, called shamt, which describes the shift amount and a 2 bit control input, where W is a parameter specifying the data width of the input. Shifter should be able to do logical shift left, logical shift right, arithmetic shift right and rotate right, for which the control signals are given in the Table 1

Table 1: Shifter Control Descriptions

Shifter Control [1:0]	**Shifter Operation**
00	LSL
01	LSR
10	ASR
11	RR

1.2.5 Arithmetic Logic Unit (ALU) (10% Credits)

1. Implement a W-bit ALU for 2's complement arithmetic, where W is a parameter specifying the data width of its operands.

   The ALU has 3 data inputs two W-bit for operands and one 1-bit for *carry*. The ALU should be equipped with 12 operations controlled by a 4-bit control input. In addition to the W-bit result output, the ALU should have 4 other status output bits: Carry out (CO), overflow (OVF), negative (N), and zero (Z). Negative and zero bits are affected by all the ALU operations; whereas, carry-out and overflow can only be affected by arithmetic operations. The specifications of the ALU operations and the ALU status outputs are provided in Table 2 and Table 3, respectively.

Table 2: ALU Operation Control

ALU Control [3:0]	**ALU Operation**	**Symbol**
0000	AND	$A \wedge B$
0001	EXOR	$A \oplus B$
0010	SubtractionAB	$A - B$
0011	SubtractionBA	$B - A$
0100	Addition	$A + B$
0101	Addition Carry	$A + B + carry$
0110	SubtractionAB Carry	$A - B + carry - 1$
0111	SubtractionBA Carry	$B - A + carry - 1$
1100	ORR	$A \vee B$
1101	Move	B
1110	Bit Clear	$A \wedge \neg B$
1111	Move Not	$\neg B$

Table 3: ALU Status Descriptions

Status	**Description**
CO	1 if there is a Carry Out from add or subtract operations; 0 for logic operations
OVF	1 if the add or subtract operation results in overflow; 0 for logic operations
Z	1 if the result is zero
N	1 if the result is negative

2. Explain your method to detect overflow.
3. Write a test bench module to test your implementation.
4. Comment the test cases in your code such that it is easily understandable which case you are testing: Addition, subtraction, AND, OR, overflow, etc.
5. Verify that your implementation is correct.
6. Provide you Makefile

1.2.6 Registers (8% Credits)

For this step, you will implement 3 different W-bit registers, where W is a parameter specifying the data width of the parallel input to the register and output of the register. Note that the **registers should have a clock input**, even though it is not mentioned in the following items explicitly.

1. Simple register with synchronous reset: Implement a positive edge-triggered register with parallel load and a synchronous reset. If the reset signal is 1, the content of the register is cleared at the next rising edge of the clock. If the reset signal is 0, the content of the register is loaded with

the input data at the next rising edge of the clock. The specifications of the simple register with synchronous reset is provided in Table 4.

Table 4: Simple Register (A) with Reset

Reset	Operation
0	$A \leftarrow DATA$
1	$A \leftarrow 0$

2. Register with synchronous reset and write enable: Implement a positive edge triggered register with parallel load, write enable and synchronous reset. If the reset signal is 1, the contents of the register is cleared at the next rising edge of the clock. If the reset signal is 0 and write enable signal is 1, the contents of the register is loaded with the input data at the next rising edge of the clock. Finally, if the reset signal is 0 and write enable signal is 0, the register retains its content. The specifications of the register with synchronous reset and write enable is provided in Table 5.

Table 5: Register (A) with synchronous reset and write enable

Reset	Write Enable	Operation
0	0	Retain
0	1	$A \leftarrow DATA$
1	X	$A \leftarrow 0$

1.2.7 Memory Unit (10% Credits)

For this step, you will implement a byte-addressable memory. The module has the inputs of a clock, write enable, write data and address, and the output of read data. W is a parameter specifying the data width of write data and read data in **bytes**, input address width is up to you. Clock and write enable is 1-bit.

Memory addressing should be combinational, read data should change the moment input address changes. When write-enable is given as 1, W-byte write-data input should be written to the location specified by the address input at the next positive clock edge. **You should use Little Endian convention to be consistent with ARM**

1.2.8 7-Segment Display Converter (Not Graded)

Important: Although not part of the design, you will need a 7-segment converter to be able to properly show the operation of your design in the lab sessions.

This module should take a 4-byte input and output the input as a hex number for the 7-segment display. Specifics of the conversions is up to you but a simple case or else-if statement should be sufficient. You can see the details of the 7-segment module in Figure 1. For more info please check the DE1-SoC user manual.

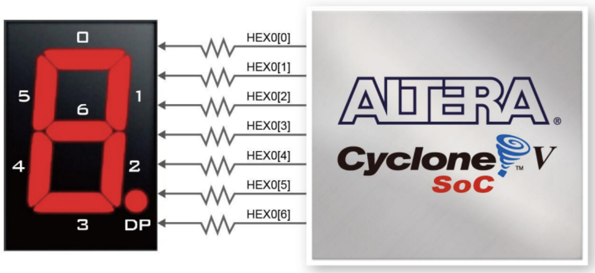


Figure 1: 7-Segment Display Signals for the DE1-SOC Board

1.3 Register File (20% Credits)

For this part, you will use your modules available from Part 1.2 to design a W-bit register file of 16 registers, where W is a parameter specifying the data width of the registers. The register file will be the central storage of the computer you will design in the future laboratory works.

For the design of the register file, you will use your **decoder**, **multiplexer** and **register** implementations. The design should be according to the desired operation of the register file.

The register file has one data input and two data outputs. The sources of the outputs and the destination of the input can be one of the 16 registers in the register file. Therefore, there should be 3 address inputs of width 4: one for destination select and two for source select. Finally, a control signal is required to enable write operation and a synchronous reset signal is required to clear the contents of the all registers in the register file. Note that the register file should inherently have a clock input, thus it is not mentioned explicitly.

The content of a register in the register file should be able to be modified without affecting the contents of the other registers. To modify a register, it should be addressed and write enable control of the register file module should be 1. If write enable of the register file is 0, then the contents of the registers cannot be modified except for the reset condition. Note that the write operation is synchronous; however, read operation should be asynchronous so that the data outputs are available as soon as their sources are addressed.

According to the aforementioned desired operation of the register file:

1. Design and sketch (on a paper) a datapath for a register file design using your decoder, multiplexer and register modules. You may use additional gates wherever necessary. For your sketch, you may present your modules with boxes.
2. Implement your design in Verilog HDL.
3. Write your testbench for your implementation using cocotb. **(10% Credit)**
4. Provide you Makefile
5. Verify that your implementation is correct.

You should submit pen and paper (or digital drawing) of the sketch of your design, the design code and the corresponding test bench code with the Makefile.

1.4 Datapath Design for an Architecture (40% Credits)

In this part, you will design a datapath for an architecture so that you can perform several tasks by applying proper control signals. The architecture to be completed is provided in Figure 2, the modules are the one's you designed in subsection 1.2. There are one 8-bit register with reset and write enable, one 8-bit ALU, 8-bit one shifter and two 8-bit multiplexers in the initial datapath. External data input is directly connected to the input of one of the MUXes.

Assuming that **the existing connections cannot be modified** you should complete the architecture by designing a datapath so that the following tasks can be performed with the desired constraints in less than 5 clock cycles each:

1. 2's Complement Load: Load the register with 2's complement of the input.
2. Multiply by 10: The register will be loaded with the input times 10.
3. Duplicate the First 4-bit: Given a byte such as $x_7x_6x_5x_4x_3x_2x_1x_0$ the input make the content of the register $x_7x_6x_5x_4x_7x_6x_5x_4$

According to the aforementioned tasks with specified constraints, design a datapath with as many additional MUXes as you want. You will implement your design in **Schematic Editor**. Thus, make sure that your Verilog modules are exported as symbols (see Laboratory Manual for how-to).

1. Implement your design in Verilog HDL.
2. Write your testbench for your implementation using cocotb. **(20% Credit)**
3. Provide your Makefile
4. Verify that your implementations are correct.

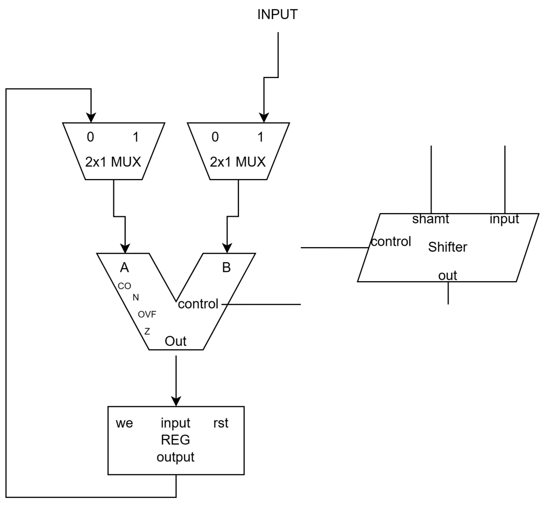


Figure 2: Architecture to which a datapath is to be designed

Considering your design, answer the following questions:

- How many control pins for the control signals does your architecture have?
- How many different control signals does your architecture use to perform the desired tasks?
- Can you reduce the number of the control pins? Why not or how?
- Write down the sequence of the control signals for all operations. How many clock cycles does these operations take?

For this part, should submit pen and paper (or digital drawing) of the sketch of your datapath design and your answers to the questions.

2 Experimental Work

2.1 Register File (40% Credits)

Load the register file module designed in the Preliminary Work Part 1.3 to the DE1-SoC Board board as a 8-bit register file. There are 10 switches and 4 push buttons in the DE1-SoC Board board. For the clock signal, use one of the push buttons of DE1-SoC Board board, debouncing exists in the push buttons of DE1-SoC Board. For other control signals since the board does not have enough switches and buttons you may connect some of the signals to the ground or VCC, everything is accepted as long as you can demonstrate proper operation of your design.

Verify the operation of the register file and demonstrate it to your lab instructor.

You must output each of the register file outputs using 7-segment display modules of the board. Use one 7-segment module for each hexadecimal digit

2.2 Datapath Design (60% Credits)

Load the custom architecture designed in the Preliminary Work Part 1.4 to the DE1-SoC Board board. Use 8 hardwired connections to the ground or VCC as the data input and use the 7-Segment Display of the DE1-SoC Board board to display the content of the register. For your control signals and the clock, you may use the push buttons and switches from DE1-SoC Board board.

Verify the operation of your design by performing the 2's Complement Load, Multiply by 10 and Duplicate the First 4-bit operations and demonstrate it to your lab instructor.

You must output the result signal using 7-segment display modules of the board. Use one 7-segment module for each hexadecimal digit

3 Parts List

`DE1-SoC Board`