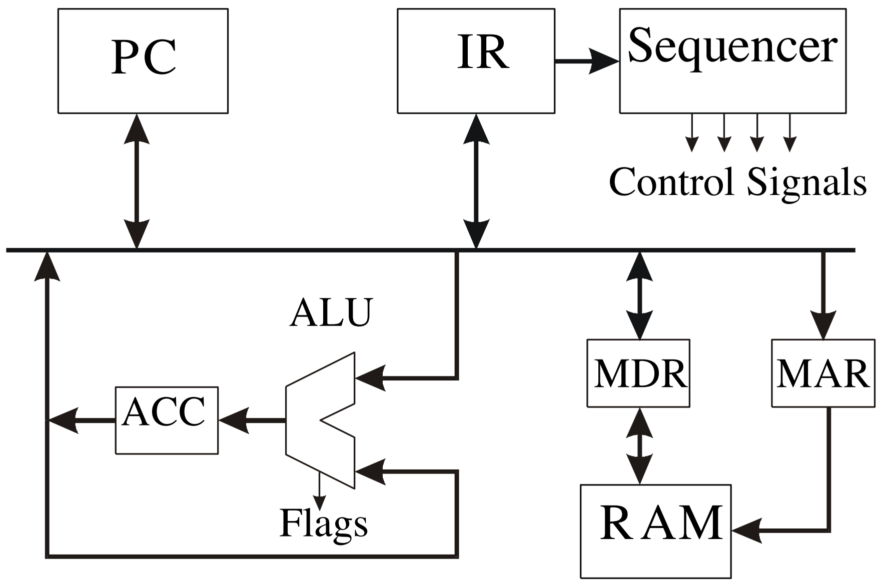
D2

Digital Systems and Microprocessor Design Exercise

This exercise contributes 10% of the marks for ELEC1202.

This exercise will be done individually. This version of the exercise has been revised for remote working in 2021.

In this design exercise, you will apply your knowledge of digital design and microprocessors. You will modify the instruction set of a very basic microprocessor in order to perform a simple encryption and decryption task.



Schedule

Preparation time	:	6 hours
Remote lab time	:	6 hours in 2x 3-hour sessions

Items provided

Tools	:	
Components	:	
Equipment	:	
Software	:	ModelSim, Quartus

Items to bring

Not applicable

Before your first scheduled session, it is essential that you read through this document and complete ***all*** of the preparation work in section 2. Only preparation which is recorded in your laboratory logbook will contribute towards your mark for this exercise. This is an individual exercise, so you should complete the preparation yourself, but you are free to discuss the problem with other students, subject to the academic integrity conditions below. Before starting your preparation, read through all sections of these notes so that you are fully aware of what you will have to do.

Academic Integrity – *If you undertake any of the work in this exercise jointly with other students, it is important that you acknowledge this fact in your logbook. Similarly, you may want to use sources from the internet or books to help answer some of the questions. Again, record any sources in your logbook.*

Revision History

January 14, 2021	Mark Zwolinski (mz)	Revised for remote working
October 15, 2019	Mark Zwolinski (mz)	Corrected minor errors
January 7, 2019	Mark Zwolinski (mz)	Revised with new hints section
January 3, 2018	Mark Zwolinski (mz)	New version

Mark Scheme

Preparation	
0	No preparation completed before the lab
1	Partially completed, no understanding
2	Partially completed, minimal understanding
3	Partially completed, good understanding *or* All completed, minimal understanding
4	All completed, good understanding
5	All completed, excellent understanding
Progress & Quality	
0	No progress made/lab not attended
1	Minimal progress (<25% of section 3)
2	Completion of <u>some</u> of section 3
3	Completion of section 3.1 – string decrypted
4	Good solution to sections 3.1 and 3.2
5	Good solution to section 3.1 and excellent solution to section 3.2
Understanding	
0	No learning or understanding
1	Very little understanding of section 3
2	Some understanding of section 3
3	Good understanding of section 3
4	Very good understanding of <u>all</u> of section 3
5	Excellent understanding of <u>all</u> of section 3
Logbook	
0	No work recorded in the logbook (or proper logbook not used)
1	Minimal record of the lab in the logbook, of very little benefit
2	Minimal record of the lab in the logbook, of some benefit
3	Good record of some activities/problems in the logbook, but sporadic coverage
4	Complete record of activities/problems in the logbook, some improvement possible
5	Excellent and complete record in logbook

Note that the handin for preparatory work and your final log book is under ELEC1029. This is the same as for other lab sessions.

1 Aims, Learning Outcomes and Outline

This laboratory design exercise aims to:

- test your knowledge of state machines, digital building blocks, buses and microprocessors through a design exercise.

Having successfully completed the lab, you will:

- understand memory-mapped input and output;
- be able to modify a basic microprocessor design to include new opcodes.

---

2 Preparation

You may wish to remind yourself how to use ModelSim and Quartus. You will also find it helpful to look again at the X5 and T4 experiments. The preparation is in three parts and your log book notes should be submitted for assessment before the first scheduled session. The *maximum* time that you are expected to spend on each part is shown. You may find it easier to do section 2.3 before section 2.2.

2.1 Basic Operation (about 1 hour)

Download the CPU design files to your own account. You should use the ECS remote desktop for your work, but keep your files on the H: drive. The zip file includes the CPU components and a basic testbench that implements a clock and reset (cpu1).

The file "rom.sv" contains a program. What does this program do? Where does it store data?

What happens at address 0 and why? Where does the program loop back to?

The memory address registers in the ram and rom have 5 bits. How are the ram and rom organised to ensure that they do not overlap? How are the addresses decoded?

Compile the SystemVerilog CPU files in ModelSim and simulate the CPU over a number of clock cycles. Note that it takes 6 clock cycles to execute a single opcode, so you will need to run the simulation over at least 30 clock cycles to see the whole program execute once and a lot longer than that to understand the operation in full.
To observe "sysbus" in the simulation, click on "c1 cpu1" in the "sim" pane. "sysbus" will appear in the Objects pane. Drag "sysbus" into the "Wave" pane.

How is the high impedance, 'Z', state of sysbus shown in the simulation?

SystemVerilog and ModelSim can represent ‘Z’ and ‘X’ states, but are these states “real”? In other words, is it possible to design logic that would detect ‘Z’ or ‘X’ states and display them? (This question is repeated from T4!)

It would be possible to take module "contention" from the "components.sv" file used in T4 and modify it to detect 'Z' and 'X' states on the sysbus. This is not usually done. In the subsequent sections, it is your responsibility to ensure that there is no contention on the bus.

2.2 Memory-Mapped Input and Output (about 2 hours)

It is strongly suggested that you create a new folder for this part of the exercise. Copy the necessary SystemVerilog files to this new folder.

Simply observing the state of the bus is not particularly helpful. As in the T4 lab exercise, switches and 7-segment displays could be connected to the bus using clocked registers to hold data. *Because of remote working in 2021, you will not, of course do this with real switches and displays. The aim here is to show how input and output devices can be mapped to memory* *locations, but because it is difficult to interpret the output of a 7-segment decoder in simulation, we will simply use the binary output here.*

Module “cpu2” has an input called “switches” and an output called “display”. Both have WORD_W bits. Module “buffer” is designed as an interface between the input switches and the system bus (sysbus) and is “hard-wired” to be at address 30. Module “register” is designed as an interface between the system bus and the output display and is hard-wired at address 31. Both of these modules are more complicated that the modules in T4, but the principle is the same.

This method of interfacing inputs and outputs is known as *memory mapping* and is common in many microprocessor systems.

Sketch a diagram of the memory, going from address 0 to address 31, and showing which components are at each address.

Add the appropriate lines to cpu2 to instantiate one buffer and one register .

If the CPU attempts to read from address 30, both the buffer and the RAM will be enabled. Why is this going to be a problem?

The following line in the ram module copies the contents of mdr to sysbus:

```
assign sysbus = (MDR_bus & mar[WORD_W-OP_W-1]) ? mdr : 'z;
```

Modify this line so that mdr is not written to the bus when mar contains addresses 30 or 31. The RAM should continue to work correctly for other addresses.

Modify testcpu so that it uses cpu2 instead of cpu1. You will also need to include switches and display as variables in the testbench.

Run the simulation again. You should be able to see display and switches.

What do you see in the Waveform window for display and switches?

Note: If you see an X state (red line) on sysbus, you are getting contention between the RAM and the buffer. You will need to fix the assignment in the ram module (see above).

You can check your modification to the ram module by submitting your modified ram.sv file to an online checker. Go to the handin page for ELEC1029 and find the 1st code check entry for D2. You can submit one file, ram.sv, in the usual way. A simulation will be run and a summary of the results will be emailed back to you. If your ram module would cause contention, or if it cannot store data at all of the first 14 addresses, you will see an error message. You will also see error messages if the file does not compile.

Include the email in the record of your preparatory work. Although you can submit as many times as you wish, you should not use this method to debug your code. The number of times you submit to the system is recorded and this will be noted when your preparation is marked.

You can now validate the correct functioning of the buffer and register hardware by writing a program in the ROM that uses the buffers and registers.

Modify the program in the rom module such that the result in the accumulator is STOREd to address 31 (display) as well as address 16. Note that you will have to move the contents of the ROM at addresses 4 to 6 up by one place and modify the ADD and BNE instructions. What do you now see in the simulation?

Make a second change to the program in the rom module such that the data for the ADD operation comes from address 30 – the switches. Modify testcpu to apply different values to the switches during the simulation. What do you now see in the simulation?

2.3 Extending the Microprocessor(about 1 hour)

It is strongly suggested that you create a new folder for this part of the exercise. Copy the necessary SystemVerilog files to this new folder.

The CPU has two arithmetic operations – ADD and SUBtract. The aim of this part of the exercise is to include another logic operation – XOR. Assume that the interpretation of an XOR N instruction is to XOR the contents of the accumulator with the contents of address N – in other words it works just as the ADD instruction. To do this, 5 steps will need to be completed:

1. The bit pattern for the new opcode, XOR, is added to opcodes.h
2. A new control signal, ALU_xor, is included in the cpu module. Create a new version of the cpu module – cpu3. The module headers of alu and sequencer will need to include this control signal.
3. ALU_xor is set to 1 in state s8 of the sequencer. This is done in exactly the same way as for ALU_add and ALU_sub. Don't forget to assign a default value at the start of the always_comb block.
4. The code in the alu module is extended to perform the XOR operation. Again, this is exactly the same as for the add and subtract operations.
5. Write a new program in the rom module to use the XOR opcode.

Add the XOR opcode and functionality to the CPU by following these 5 steps.

How have you verified the correct operation of the new XOR opcode? Have you simulated its behaviour?

You can check your modifications to the alu and sequencer module by submitting your modified files to an online checker. Go to the handin page for ELEC1029 and find the 2nd code check entry for D2. You can submit two files, alu.sv and sequencer.sv, in the usual way. A simulation will be run and the code will be checked in Quartus and a summary of the results will be emailed back to you. As with the ram, you will see error messages, if your code is incorrect. Note that for this check to work, the opcode must be XOR (but the bit pattern is not important) and the control signal must be ALU_xor.
Include the email in the record of your preparatory work. Again, do not use this as the main method for debugging your code.

3 Laboratory Work

3.1 Encryption and Decryption

We can represent alphabetical characters in this microprocessor using a "5-bit ASCII" code as shown in TABLE 1.

Binary	Char	Binary	Char	Binary	Char	Binary	Char
0 0000	@	0 1000	h	1 0000	p	1 1000	x
0 0001	a	0 1001	i	1 0001	q	1 1001	y
0 0010	b	0 1010	j	1 0010	r	1 1010	z
0 0011	c	0 1011	k	1 0011	s	1 1011	[
0 0100	d	0 1100	l	1 0100	t	1 1100	/
0 0101	e	0 1101	m	1 0101	u	1 1101	]
0 0110	f	0 1110	n	1 0110	v	1 1110	.
0 0111	g	0 1111	o	1 0111	w	1 1111	_

TABLE 1 – 5-bit character encoding

A simple encryption technique is to XOR the code for each character with a key. For example, "a" (00001) can be encrypted by XORing with a key (for example, 10101) to give an encrypted character "t" (10100).
Using the opcodes that you now have, including XOR, write routines to implement encryption and decryption on a block of 8 characters in memory. Can you fit these routines into the space that you have? If not, what can you do? (See section 3.2.)

Hints

- Do you need any other opcodes? Are there any opcodes that you do not need?
- The memory is split equally between RAM and ROM. Is that still appropriate, or do you need an uneven split?
- Note that the ROM can only contain programs and data. You should not modify the structure of the SystemVerilog code to implement tests or branches. If you do that, you are not modelling memory, but something else that does not correspond to real hardware.

Testing your design
A string of 8 characters has been encrypted using an XOR operation. This is the encrypted string: "oiytmmvk".
Using your XOR opcode, write a program to decrypt this string. Then follow the decrypted string.

✏ *You <u>must</u> fully record all your work in this section.*

3.2 Extending your design

When you decrypt the string from section 3.1, you will find some comments on the encryption procedure. For this part of the exercise, you are free to extend the microprocessor design in any way you choose in order to enhance the encryption and decryption. For example:

- How could you use the XOR function to provide better encryption?
- How can you use the switches and display to move data into and out of the CPU core?
- Can you implement such a scheme in the limited memory that you have?
- Can you create larger ROM and RAM spaces by using a larger value for WORD_W (and OP_W)? Is your design fully parameterised?
- Can you also decrypt?
- Can you think of a better encryption technique?
- Do you need more opcodes?

To obtain full marks, you must show that you have validated your design using ModelSim. Thus you will have to write one or more testbenches. You should use the first initial analysis phase of Quartus to show that your design would synthesise to an FPGA without creating latches or similar structures.

✏ *You <u>must</u> fully record all your work in this section.*

Appendix Additional Notes and Hints

Mapping Switches, displays, RAM and ROM.

It will help if you open the .sv files in a text editor with line numbers.

First of all, note that we can only read from the ROM, so we don't need to worry about writing to ROM addresses. Similarly, we will only read from the switches and write to the displays. Also note that only one element can write to the sysbus at a time.

Note also that the same control signals are used to control the RAM and the ROM, namely:

clock, n_reset, MDR_bus, load_MDR, load_MAR, CS, R_NW

So the only way to distinguish between RAM and ROM access is to use address values. In the same way, we will use the same signals for reading from the switches and writing to the displays, but no others. So we don't change the sequencer or any other parts of the system.

How do the RAM and ROM fit together?

The RAM has 16 addresses - see line 31. [0:(2**WORD_W-OP_W-1))-1] translates to the range [0: (2**4)-1] or [0:15]. The ROM has up to 16 addresses too, but we only use 7.

So, we need 4 bits to address each of the 16 address ranges, but the MAR has 5 bits. Look at line 33 of rom.sv and line 35 of ram.sv. What's the difference? In the ROM, mar[4] is tested to see if it's 0; in the RAM, mar[4] is tested to see if it's 1. Therefore, only the ROM or the RAM can write to the sysbus, but not both. Note also that on line 52 of ram.sv, only the bottom 4 bits of mar are used for the address in the ram array.

Similarly, for writing to memory, mar[4] is tested in line 50 of ram.sv and the bottom 4 bits are referenced on line 52.

Putting all this together means that the ROM sits between addresses 0 and 15, while the RAM is between addresses 16 and 31 and we achieve this simply by using mar[4] to distinguish between the two parts. But we need to be careful to ensure that we don't accidentally reference two parts of memory at the same time.