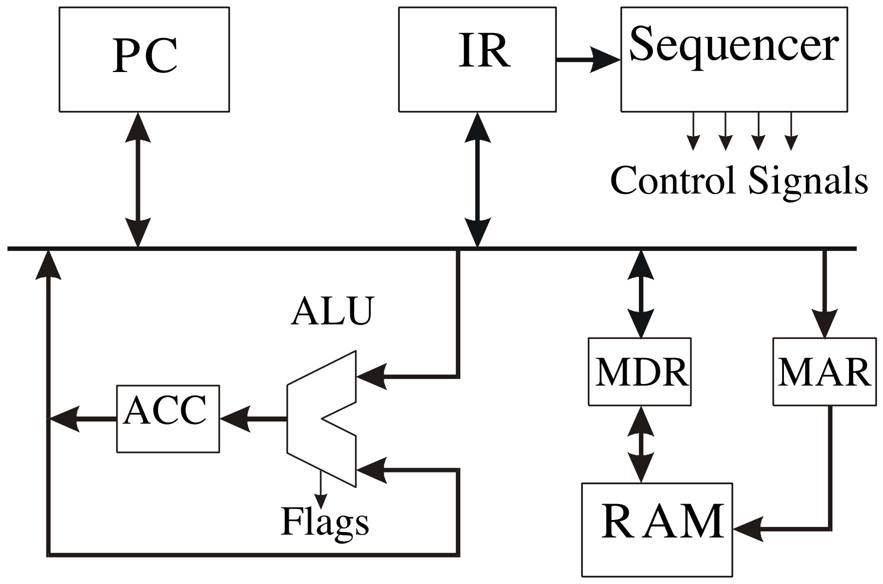
D2

Digital Systems and Microprocessor Design Exercise

This contributes 10% of the marks for ELEC1202.

This exercise will be done individually.

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
Lab time	:	6 hours

Items provided

Tools	:	
Components	:	
Equipment	:	DE1-SoC development board
Software	:	ModelSim, Quartus

Items to bring

Essentials. A full list is available on the Laboratory website at https://secure.ecs.soton.ac.uk/notes/ellabs/databook/essentials/

Before you come to the lab, 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. Before starting your preparation, read through all sections of these notes so that you are fully aware of what you will have to do in the lab.

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

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

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 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 may also find it helpful to look again at the X5 and T4 experiments.
Read through the course handbook statement on safety and safe working practices, and your copy of the risk assessment form. Make sure that you understand how to work safely. Read through all of this document so you are aware of what you will be expected to do in the lab.

2.1 Basic Operation

Download the CPU design files to your own account. The zip file includes the CPU components and a basic testbench that implements a clock and reset. Note that the design includes two 7-segment decoders that allow the state of the system bus (sysbus) to be displayed.

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?

Create a ModelSim project 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.
The outputs from the CPU are the signals to control the 7 segment displays. You will find it easier to observe "sysbus" in the simulation. To do this click on "c1 CPU" 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 show them on a 7-segment display? (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

Simply observing the state of the bus is not particularly helpful. As in the T4 lab exercise, switches and 7-segment displays can be connected to the bus using clocked registers to hold data.
Write SystemVerilog code to interface 8 switches and two 7-segment displays to the system bus. You can leave the existing displays in place to show the state of the bus. The switches and displays should be at addresses 30 and 31 (it does not matter which is which). The registers between the switches/displays and the bus should be controlled by the system clock and n_reset. The registers should be selected by decoding addresses and the CS and R_NW signals. *You*

should not modify the sequencer. You should create a new top-level module (for example CPU2) and you should create a new ModelSim project.

How have you verified the correct operation of the switches and displays? Have you simulated their behaviour?

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 in your system, going from address 0 to address 31, and showing which components are at each address.

You can validate the correct functioning of your hardware by writing a program that uses the switches and 7 segment displays.

Write a program that reads 2 numbers from the switches and displays the sum on the 7 segment displays.

2.3 Extending the Microprocessor

We can represent alphabetical characters in this microprocessor using a "5-bit ASCII" code as shown in TABLE 1. 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). Therefore, to implement this encryption, we need an XOR operator. Extend the microprocessor to implement an XOR operator. You will need to modify the sequencer, ALU and the control signals. You should create a new top-level module (for example CPU3) and you should create a new ModelSim project. Demonstrate the functionality by simulation.

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

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

Write a program that uses the new XOR opcode.

---

3 Laboratory Work

3.1 Basic Operation

You will find it helpful to have the simulation waveforms from the preparation on the screen for the next part, so restart ModelSim and display the simulation from section 2.1.

Create a Quartus project for the cpu, with CPU as the Top-level entity. As you have previously done, analyse these source files, then add the contents of “cpu_pins.qsf” to the “cpu.qsf” project file and do a full synthesis. The FPGA should be “5CSEMA5F31C6”.
Configure the FPGA with file “cpu.sof”. Assuming that you have followed all the steps correctly, the two rightmost 7-segment displays should show ‘00’. The clock is pushbutton KEY3 and the n_reset is KEY2. Clock the system through the program and observe the state of sysbus as shown on the 7-segment displays.

Does the behaviour follow the simulation? When the simulation shows a 'Z' state, what is shown on the 7 segment displays? Can you explain this?

3.2 Memory-Mapped Input and Output

Create a new Quartus project for the modified design (for example CPU2). You will also need to define the pins used to connect the switches and displays, and hence the qsf file will need to be modified. Consult the DE1-SoC User Manual for the pin numbers.

Is the behaviour on the FPGA board the same as the simulation?

3.3 Extending the Microprocessor

Implement the extended CPU on the FPGA board.

Is the behaviour on the FPGA board the same as the simulation?

3.4 Encryption and Decryption

Using the opcodes that you now have, 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?

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?
- 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.
- 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.

---

4 Optional Additional Work

<u>Marks will only be awarded for this section if you have already completed all of Section 3 to an excellent standard and with excellent understanding.</u>

- How could you use the XOR function to provide better encryption? Can you implement such a scheme in the limited memory that you have? Can you also decrypt? Can you think of a better encryption technique? Do you need more opcodes?
- Pushing a button to drive the clock is fairly boring. Modify your design (including the pin assignment) so that the CPU uses one of the built-in clocks and a counter to slow the clock down. (See exercise X5.) How can you enter data using the switches *and* have a continuously running clock?

Appendix Additional Notes and Hints

Mapping Switches, 7 Segment 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 7 segment 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.

1. How do the RAM and ROM fit together?

The RAM has 16 addresses - see line 31. [0:(1<<(WORD_W-OP_W-1))-1] translates to the range [0: (1<<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 53 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 49 of RAM.sv and the bottom 4 bits are referenced on line 51.

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.

2. How can we include a 7 segment display register?

Let's assume that we want to put the register at address 31. First of all, we need to ensure that we don't have RAM at this address. Look at line 32 of RAM.sv. This line is commented out, but it shows how the range of the RAM can be limited (without relying on parameters).

We also need to change lines 35, 49, 51 and 53 of RAM.sv, to ensure that address 15 (taking just the bottom 4 bits) is not accessed in the RAM.

If you look at the files playbus0.sv and components.sv from the T4 lab, you will see that a register is used to store the data displayed by the 7 segment display. We need to use a similar structure, but the control signals need to be the same as for the RAM. So, instead of a simple controlled register like:

```
always_ff @(posedge clk)
  if (OE)
    data_out <= data_in;
```

we need to go through several different steps:

```
always_ff @(posedge clock, negedge n_reset)
  begin
  if (~n_reset)
    begin
    mdr <= 0;
    mar <= 0;
    end
  else
    if (load_MAR)
      mar <= sysbus[WORD_W-OP_W-1:0];
    else if (load_MDR)
      mdr <= sysbus;
    else if (CS & ...) // look for address 31
      if (!R_NW) // Writing, so this signal needs to be low
        data_out <= mdr;
  end
```

Note that this takes 2 cycles for the write, so as to be consistent with the ASM chart.

3. How can we include input switches?

Let's assume the switches are at address 30. Again, look at the T4 files. This is simpler than the 7-segment case. The switches there are buffered with:

```
assign data = (OE & CS) ? data_in : 'Z;
```

Once again, we need to change this line to be consistent with the processor.

```
assign sysbus = (MDR_bus & ...) ? data_in : {WORD_W{1'bZ}};
```

So, here we include the logic to decode address 30 from mar.

We also need to load the mar:

```
always_ff @(posedge clock, negedge n_reset)
  begin
  if (~n_reset)
    begin
    mar <= 0;
    end
  else
    if (load_MAR)
      mar <= sysbus[WORD_W-OP_W-1:0];
  end
```

We don't use CS or R_NW for the ROM or the switches. We could put the switch values into a register, but it's not really necessary.

4. Putting it all together.

You could include all this in the RAM.sv module/file, but I would create two separate .sv modules/files, one for the switch buffer and one for the registers and instantiate them in the top level CPU module, as well as the 7 segment decoder. Note that you also need to include the input and output pins in the module header.