EE496 - VHDL Design & Synthesis Assignment 2

Due date: 28 November 2018 (week 10)

(20% weighting of the final module mark)

(Note: The lab report should be in PDF format, and all other files should be included in a single .zip file)
The purpose of this assignment is to build on the VHDL covered in lectures and to introduce the fundamentals of digital design using VHDL. Some relatively simple Verilog coding is also required. We will be covering the fundamentals of Verilog in an upcoming lecture (*Note: Verilog will be included in the final examination*). In the meantime you can find sufficient resources online in order to do the Verilog section of the assignment.

Both combinational logic and synchronous system design are included. Although you are not required to synthesize your design, you should ensure that all your non-testbench code is synthesizable. Functional (i.e. behavioural) simulation of your test benches must be used to verify your designs.
VHDL/Verilog code for the assignment should be developed & simulated using the free Webpack of the Xilinx design tool **Vivado 2018.2**, which can be downloaded from Xilinx website.

Digital Cryptography
Fundamental to providing adequate security for digital applications is the implementation of cryptographically secure algorithms. Unfortunately cryptographic algorithms are computationally intense applications which typically require some form of hardware acceleration to provide sufficient performance. The primary task of hardware accelerators is to off-load highly computational operations to dedicated hardware interfaced to the general purpose processor. However, the implementation of such dedicated accelerators requires additional trade-offs to be examined. A typical solution is to implement a cryptographic co-processor which has being tailored for generic cryptographic functions while not being specific to one particular algorithm.

Regardless of the underlying structure, high speed modern symmetric algorithms employ similar techniques to ensure security. Some of these methods include

- Substitution.
- Permutation.
- Key Addition.
- Addition over a finite field.
- Multiplication over a finite field.

The goal of this assignment is to implement some of these functions into a flexible 16-bit cryptographic co-processor.

The structure of the assignment is as follows, Section 1 covers the design of the combinational components which are utilised in our crypto-processor. Section 2 covers synchronous designs, as well as the components needed to ensure the data processed by our combinational logic can be stored. Finally a test program is given to ensure the final synchronous design works as expected.

Section 1: Combinational Logic

To remain flexible and to allow the implementation of many cryptographic applications, the co-processor must include standard instructions as well as dedicated functions specific for security.

1.1 ALU Design

The ALU (Arithmetic Logic Unit) shown in Figure 1 is the core combinational component of any flexible processing unit. Given the data and control inputs the ALU will perform an operation defined by its architecture..

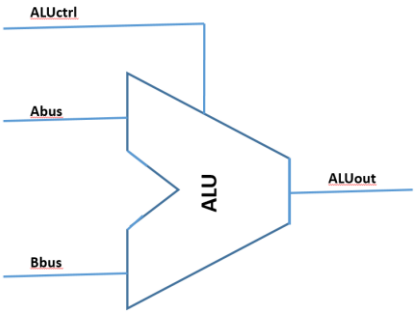

Figure 1: ALU

Given the instruction set defined in Table 1, implement a 16-bit version of the ALU shown in Figure 1.

Table 1 ALU Instruction Set

ALUctrl	Mnemonic	Instruction Function
0000	ADD	ALUout = Abus + Bbus
0001	SUB	ALUout = Abus - Bbus
0010	AND	ALUout = Abus & Bbus
0011	OR	ALUout = Abus | Bbus
0100	XOR	ALUout = Abus ^ Bbus
0101	NOT	ALUout = ~ Abus
0110	MOV	ALUout = Abus

Task: Implement the ALU described above in VHDL.
Notes: You **can** make use the arithmetic operators in package ieee.numeric_std.

1.2 Shifter in VHDL

Commonly used in the permutation and transposition functions of ciphers, the ability to shift and rotate data fast is a pre-requisite for both general purpose processing and cryptographic functions.

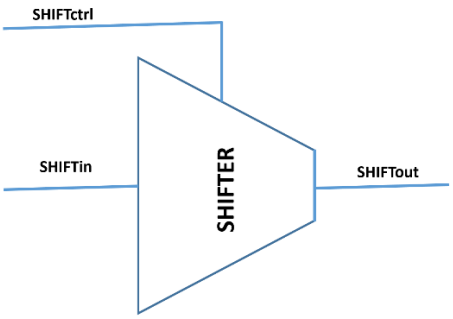

Figure 2: Shifter

Analysis of some algorithms has shown that the shift function is mostly utilised in *nibble* and *byte* aligned shift and rotates. For that reason we decide to implement a very simple shift and rotate function instead of a complex shifter.

SHIFT ctrl	Mnemonic	Instruction Function
1000	ROR4	Rotate right 4 bits
1001	ROL4	Rotate left 4 bits
1010	SLL4	Shift left logic 4 bits
1011	SRL4	Shift right logic 4 bit

Task: Implement the Shifting Unit described above in VHDL.
Notes: You can make use of the ROR, SLL and SRL function in package ieee.numeric_std.

1.3 Non-Linear Lookup Operations in VHDL

To obscure a relationship between inputs and outputs from an encryption algorithm, a method of non-linear substitution is usually employed in digital symmetric algorithms. The basic operation of a substitution function is to take an n-bit input (usually n = 4) and map it to an n-bit output as defined by the substitution function.

Output <= Substitute(Input)

The substitution function can be implemented using either complex maths functions, a single table lookup operation utilising large memory components or by breaking the function into smaller lookup operations interspersed with binary operations. The non-linear transforms which comprise SBox-1 and SBox-2 are given below.

S-Box1 (in integer equivalent of 4 binary bits)

Input	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Output	1	11	9	12	13	6	15	3	14	8	7	4	10	2	5	0

S-Box2 (in integer equivalent of 4 binary bits)

Input	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Output	15	0	13	7	11	14	5	10	9	2	12	1	3	4	8	6

Figure 3 illustrates the non-linear lookup substitution operation.

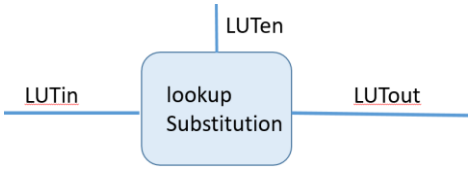

Figure 3 Lookup substitution box

Figure 4 shows the internal structure of the lookup substitution block in Figure 3. LUTin is a 16-bit input with index range (15:0). The input to Sbox-1 is 4-bit of LUTin with index range (7:4), the input to Sbox-2 is 4-bit of LUTin with index range (3:0). For simplicity we assume only the LEAST SIGNIFICANT BYTE of the 16-bit input is used. The most significant byte is simply passed through to the output unaltered. The lookup substitution operation is carried out when the lookup enable signal LUTen is high.

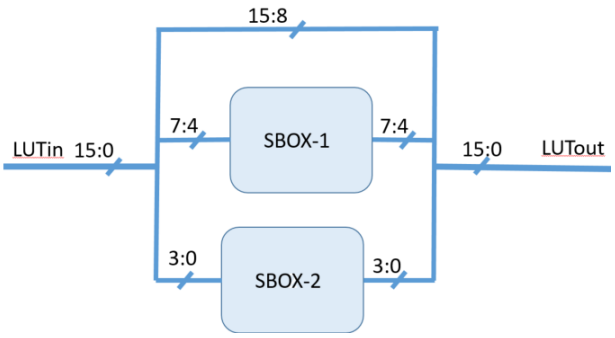

Figure 4 Internal structure of the non-linear Lookup Operation

SHIFT ctrl	Mnemonic	Instruction Function
1100	LUT	Subsitute(LUNin)

Task: Implement the hardware unit described in figure 3 and Figure 4 in VHDL.

1.4 Structural VHD model

Up to now we have developed three VHDL functional blocks which when combined create a logic design allowing for 12 operations. The operations are controlled by the **Ctrl** bits. Using the three VHDL functional blocks developed in the previous sections as components implement the basic combination logic system shown in Figure 4.

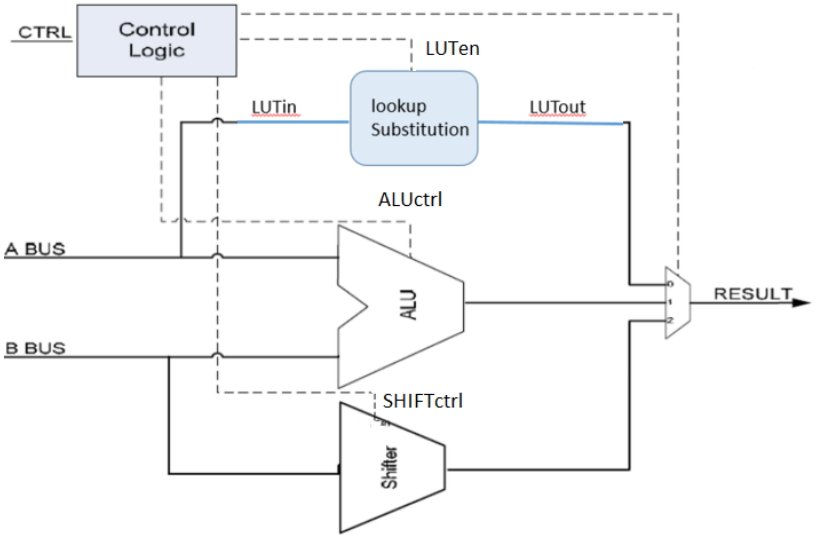

Figure 5 – Combinational logic system

Your design should have two 16-bit inputs (a_bus, b-bus) , a 4-bit control bus (ctrl) and a 16-bit output (result). The purpose of the control logic decoder is to map the 16 possible input codes to the control logic needed to control the three components in the combinational logic system. **All** signals can be determined from the 16 states. The design of the control logic decoder is left open to any implementation (Note: the control logic is not necessarily a separate entity)

Task: Implement the combinational logic system in Figure 5 in VHDL. You are **required** to **instantiate** the ALU, Shifter and the substitution functional blocks as **components** in your design**.**

1.5 VHDL Test bench for Structural Design

Task: Develop a VHDL test bench and stimulus generator scheme for the structural unit you implemented in section 1.4 which cycles through a number of different inputs and all 12 possible operations.

It is recommended to write test vectors in hex format. e.g. possible format would be as follows:-

Test Index (Integer)	**OPCODE (hex)**	**Ra (hex)**	**Rb (hex)**	**Result Expected (hex)**
1	2	F0F2	F0F1	F0F0
2	A	0000	13AC	AC00
etc				

For each test executed, your stimulus generator must print a suitable PASS/FAIL message to the console, including the value of the test index in the message. Optionally, you can log the test results to a file.

Section 2: Synchronous Logic Design

The purpose of this part of the assignment is to build on the fundamentals of VHDL examined in section 1. It is intended to examine the use of synchronous designs within VHDL.

2.1 Registers

Nearly all digital systems rely on some form of synchronicity to maintain data flow within a piece of logic. Without some form of a clock signal, data inputs and outputs from combinational logic would be very susceptible to input glitches with any changes in the input propagating to the output. Along with this, an additional problem is that it is difficult to know when a new input can be supplied and when an output is valid. To solve these problems we implement edge triggered registers which respond to the edges of a global clock signal.

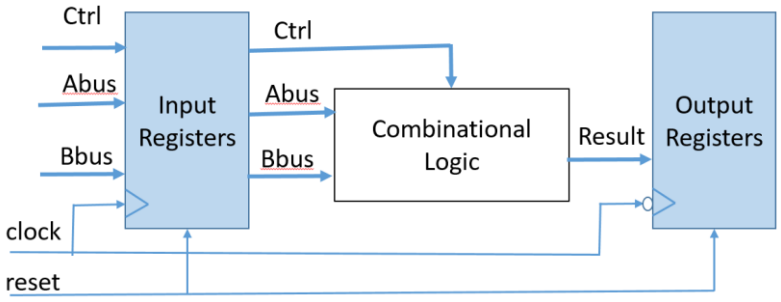

Figure 6 Synchronous system

Figure 6 shows the synchronous coprocessor. The input values and control value are **rising-edge** registered with a **falling-edge** triggered register used at the combinational output.

Task: Using the combinational system developed in section 1.4, implement the system shown in Figure 6. The registers should have an *asynchronous reset*. The design should take identical inputs as in section 1.4 but should produce a valid result after 1 clock cycle.

2.2 Memory

The ability to model synchronous systems in VHDL allows the modelling of memory elements within digital systems. Fundamental to the operation of any programmable processor is the register file. The register file is an array of registers which are used to store operands and variables. Register files are typically made to *write synchronous* to, either rising or falling clock edge, with *either synchronous or asynchronous read*. Figure 7 shows a diagram of a typical register file. In VHDL a register file can be implemented with only a few lines of code.

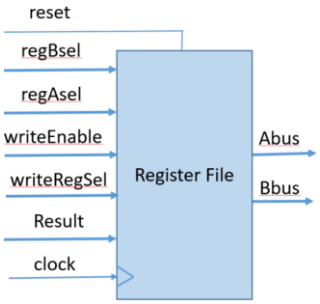

Figure 7 Register file

Task: Implement a 16 x 16-bit register file in VHDL similar to the structure shown in Figure 7. It should provide *rising edge synchronous read and write*. The register file should have the following input/output signals:

```
entity register_file is
    port
    (
    Abus          : out std_logic_vector(15 downto 0);
    Bbus          : out std_logic_vector(15 downto 0);
    result        : in  std_logic_vector(15 downto 0);
    writeEnable   : in std_logic;
    regAsel       : in std_logic_vector(3 downto 0);
    regBsel       : in std_logic_vector(3 downto 0);
    writeRegSel   : in std_logic_vector(3 downto 0);
    reset         : in std_logic;
    clock         : in std_logic
    );
end register_file;
```

The file contains 16 16-bit registers. Each cycle, two registers are read and one register is written (if writing is enabled). There should be a *data bypass (forwarding)* so that the value just written is forwarded directly to the output if we are reading from and writing to the same register in a single cycle.

Vivado supports RAM initialization in VHDL, so to initialize the contents of your register file (i.e. memory), you simply need to give your register file an initial value. The initial values are given below.

Register File Initial Values --Hexadecimal Values

```
type memory is array(0 to 15) of std_logic_vector(15 downto 0);
signal REG_FILE : memory := (
                            0 => x"0001",
                            1 => x"c505",
                            2 => x"3c07",
                            3 => x"d405",
                            4 => x"1186",
                            5 => x"f407",
                            6 => x"1086",
                            7 => x"4706",
                            8 => x"6808",
                            9 => x"baa0",
                            10=> x"c902",
                            11 => x"100b",
                            12 => x"C000",
                            13=> x"c902",
                            14 => x"100b",
                            15 => x"B000",
                            others => (others => '0'));
```

2.3 Complete Crypto Coprocessor

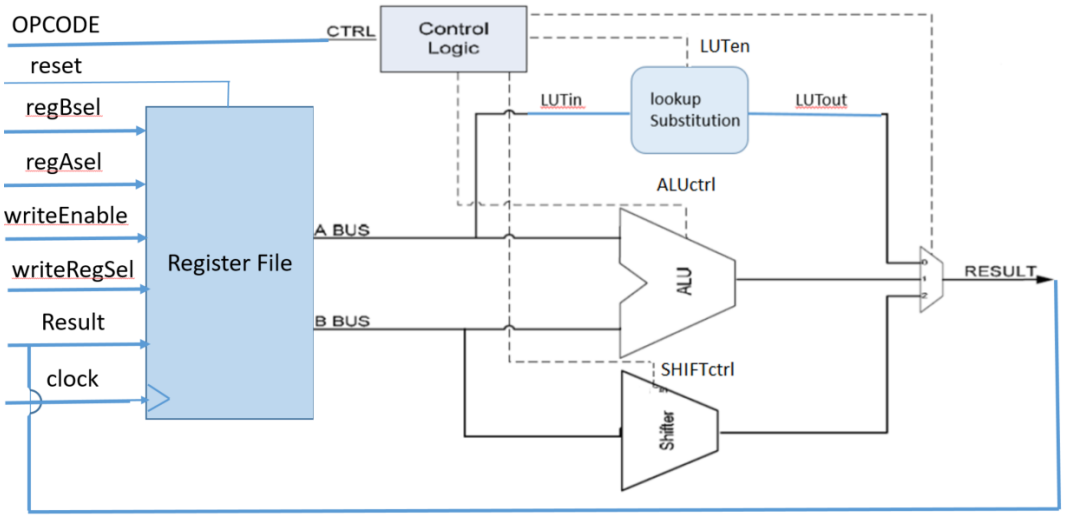

Figure 8 Complete Crypto Processor

The inputs to the Crypto processor are a 16-bit instruction word, and a clock and a reset signals. The register file write enable signal will normally be held high in order to store the result of the operation.

The 16-bit instruction word (15 downto 0) takes the form:

15			0
OPCODE	Ra index (regAsel)	Rb index (regBsel)	Rd index (writeRegSel)

This Crypto processor em also handles an additional CTRL OPCODE

CTRL OPCODE	**Mnemonic**	**Instruction Description**
0111	NOP	No Operation Performed

In other words, the output of the combinational logic must not be stored in memory when the OPCODE is NOP. Handling of the NOP instruction is not shown in Figure 8.

Task: Implement the design shown in Figure 8 which provides the register space needed to store variables and the sequential logic to control the circuit behaviour. You must also include the VHDL necessary to handle the NOP instruction.

Hint: Assuming an instruction word is loaded on a clock rising edge, there is a flaw in the circuit above. The destination register index, Rd, will be presented for the current instruction but won't be needed until after the next clock cycle, by which time another instruction will have been loaded. Since the result is delayed by a single clock cycle we would write the result to the wrong address.

Consider the following instruction sequence:

ADD R1, R2, R3
XOR R2, R5 R7

During clock period 1, the ADD instruction is presented and executed. After one clock period the result of the ADD instruction will be visible at the output of the combinational logic, however the XOR instruction will be loaded by this time. This will write (R1+R2) to R7 instead of R3 since R7

will be the Rd value present after one clock cycle. To ensure single cycle operation, **this must be solved in your design to ensure any program executes properly.**

2.4 Test Program

Assuming that **after a system reset,** the register file contains the values given in Section 2.2. Implement the following code in a stimulus generator:

```
ADD        R5, R4     R12
XOR        R1, R8     R7
ROR4 R12         R0
SLL4       R9         R3
ADD        R0, R7     R10
SUB        R7, R3     R12
NOP
AND        R12, R10   R9
NOP
LUT        R9         R2
```

Note: You can code the test vectors directly in your stimulus generator VHDL (Optionally, you can read them from a file).

---------- **End of Assignment** ----------

The following page contains information on what should be included in your report and the marking scheme.

Report Document

Your report must be in **PDF form**at, and should include a declaration state that the submitted report is **your own work**. Please refer to the following site regarding university policy on ***plagiarism***: https://www4.dcu.ie/students/az/plagiarism.

Also:
1. It must include instructions on how to run your structural test bench and your complete crypto processor test bench (i.e. the name of the Xilinx project file and the two test bench entities).

2. It should describe interesting snippets of your VHDL code

3. Block diagram illustration of your ALU design and complete crypto processor design (with flaw solved!)

4. Explain how you handle the NOP instruction and the flaw related to Rd.

5. It should include interesting simulation console logs from executions of your stimulus generator

6. It must include the contents of the register file in hex after running the complete crypto processor simulation (as mentioned earlier). You should manually verify the values in your report.

MARKING SCHEME

Section	Percentage
1.1. ALU in VHDL	10%
1.2. Shifter in VHDL	6%
1.3. NLLO in VHDL	6%
1.4. Structural VHDL	12%
1.5. Structural VHDL Test bench	6%
2.1. Registers	5%
2.2. Register file (memory)	10%
2.3. Complete Crypto Coprocessor NOP handled correctly Rd (flaw) handled correctly	10% 5% 5%
2.4 Test Program Correct final register file contents (checked by running simulation)	10% 5%
Well written, understandable VHDL/Verilog throughout the design	5%
Report document (including images of waveforms and final register file memory contents, etc)	5%