EEN1056 Computer Architecture and HDL

Assignment: Design of a Crypto-coprocessor (25% weighting of the final module mark)

Issued on 25 September 2024 (Wednesday of week 3)

Due date: 18 October 2024 (Friday of week 6)

The assignment is to design a crypto-coprocessor. All your non-testbench design code should be synthesisable, i.e., it should produce a logic circuit. (`Vivado command: RTL Analysis -> Open Elaborated Design`).

The Processing Unit in Figure 5 and the complete Crypto-coprocessor in Figure 8 should be simulated through testbench simulation.

SystemVerilog code for the assignment should be developed & simulated using the Xilinx design tool **Vivado 2024.1**.

Digital Cryptography

Implementing cryptographically secure algorithms is fundamental to providing adequate security for digital applications. Cryptographic algorithms are usually computationally intense applications that require hardware acceleration for acceptable performance. The primary task of hardware accelerators is to offload highly computational operations to dedicated hardware interfaced with the general-purpose processor. However, the implementation of such dedicated accelerators requires additional trade-offs. A typical solution is implementing a cryptographic co-processor tailored for generic cryptographic functions while not specific to one particular algorithm.

Regardless of the underlying structure, high-speed modern symmetric block encryption 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.

This assignment aims 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 logic for data processing. Then, section 2 covers the synchronous elements for data storage. The final section uses a test program to verify that the co-processor functions as expected.

Section 1: Processing Unit (Combinational Logic)

A flexible crypto co-processor should allow the implementation of different cryptographic applications. Therefore, it must include standard instructions and dedicated security-specific functions.

The Processing Unit consists of *three* combinational blocks. The **ALU** combinational block implements 8 arithmetic and logic operation instructions, the **Rotator** combinational block implements seven rotation operation instructions, and the **Lookup Substitution** combinational block implements the substitution box instruction.

Table 1 below shows the 16 instructions supported by the Processing Unit and their opcode encoding (ctrl[3:0]).

Table 1 – Instruction Set

Instruction ctrl[3:0]		Mnemonic	Instruction Function
ctrl[3]	ALUctr = ctrl[2:0]	ALUen = ~ctrl[3]	
0	000	ADD	ALUout = (Abus + Bbus) mod $2^{wordsize}$
0	001	SUB	ALUout = (Abus - Bbus) mod $2^{wordsize}$
0	010	MUL	ALUout = (Abus * Bbus) mod $2^{wordsize}$
0	011	AND	ALUout = Abus & Bbus
0	100	OR	ALUout = Abus | Bbus
0	101	XOR	ALUout = Abus ^ Bbus
0	110	NOT	ALUout = ~ Abus
0	111	MOV	ALUout = Abus
ctrl[3]	ROTctrl = ctrl[2:0]	ROTen = ctrl[3] & ~(ctrl[2] & ctrl[1] & ctrl[0]);	
1	000	ROL1	ROTout = Rotate left 1 bit of ROTin
1	001	ROL2	ROTout = Rotate left 2 bits of ROTin
1	010	ROL3	ROTout = Rotate left 3 bits of ROTin
1	011	ROL4	ROTout = Rotate left 4 bits of ROTin
1	100	ROL8	ROTout = Rotate left 8 bits of ROTin
1	101	ROR4	ROTout = Rotate right 4 bits of ROTin
1	110	ROR8	ROTout = Rotate right 8 bits of ROTin
Ctrl[3]	LUTen = ctrl[2:0]	LUTen = ctrl[3] & ctrl[2] & ctrl[1] & ctrl[0];	

1	111	LUT	LUTout = { S_box(LUTin[15:12]),
			S_box(LUTin[11:8]),
			S_box(LUTin[7:4]),
			S_box(LUTin[3:0]) };

1.1 ALU combinational block

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. The eight arithmetic and logic operations are encoded using three control bits. The arithmetic operations in cryptography algorithms are generally *unsigned* integers.

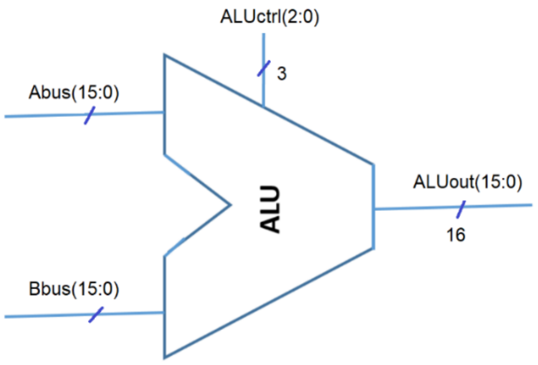

Figure 1: ALU

Given the instruction set defined in Table 2 below, implement a 16-bit version of the ALU shown in Figure 1.

Table 2 ALU Instruction Set and Control Encoding

ALUctrl	Mnemonic	Instruction Function
000	ADD	ALUout = (Abus + Bbus) mod $2^{wordsize}$
001	SUB	ALUout = (Abus - Bbus) mod $2^{wordsize}$
010	MUL	ALUout = (Abus * Bbus) mod $2^{wordsize}$
011	AND	ALUout = Abus & Bbus
100	OR	ALUout = Abus | Bbus
101	XOR	ALUout = Abus ^ Bbus
110	NOT	ALUout = ~ Abus
111	MOV	ALUout = Abus

Task: Implement the ALU described above in SystemVerilog.

Hint: The mod $2^{wordsize}$ operation is equivalent to keeping the rightmost wordsize bits of the results and dropping the other leftmost bits. In this project, the word size is 16 bits. (Abus + Bbus) produces a 17-bit number, mod 2^{16} means to keep the lower 16-bits and drop the leftmost carry-bit. (Abus * Bbus) produces a 32-bit result, mod 2^{16} means only keeping the lower 16-bits of the product and dropping the leftmost 16-bits.

1.2 Rotator combinational block

The ability to rotate data fast is a prerequisite for general-purpose processing and cryptographic operations, and it is commonly used in cyphers' permutation and transposition functions.

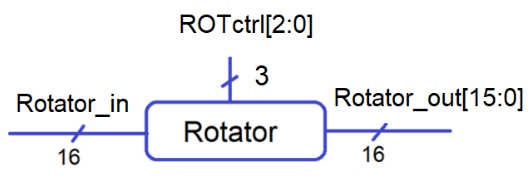

Figure 2: Rotator

The SIMON family of lightweight cyphers uses rotating left 1-bit, rotating left 2-bits, and rotating left 8-bits in each round of the encryption process. The SPECK family of lightweight cyphers uses 8-bit right-rotation and 3-bit left-rotation in the key schedule and round functions. Some other encryption algorithms primarily utilise *nibble* and *byte*-aligned shift and rotate. Therefore, we decided to implement a specific Rotator component, as shown in Figure 2, instead of a complex shifter/rotator.

Table 3 Rotator Instruction Set and Control Encoding

ROTctrl	Mnemonic	Instruction Function
000	ROL1	Rotate left 1 bit
001	ROL2	Rotate left 2 bits
010	ROL3	Rotate left 3 bits
011	ROL4	Rotate left 4 bits
100	ROL8	Rotate left 8 bits
101	ROR4	Rotate right 4 bits
110	ROR8	Rotate right 8 bits

Task: Implement the above Rotator in SystemVerilog.

1.3 The Lookup substitution combinational block

A non-linear substitution is usually employed in symmetric block encryption algorithms to obscure the relationship between inputs and outputs. The essential operation of a substitution function is to take a piece of n-bit information (usually n = 4) and map it to an n-bit output as defined by a non-linear substitution function called S-box.

```
Output <= Substitute(Input)
```

The substitution function (S-box) can be implemented using either a single table lookup operation utilising memory component or by constructing the S-box using logic gates. Table 4 shows the PRESENT, an ultra-lightweight block cypher's S-box. PRESENT is an ISO/IEC 29192-2:2012 standard lightweight block cypher optimised for hardware implementation.

Table 4 S-Box of the PRESENT cypher (in hexadecimal notation)

Sbox input x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Sbox output S[x]	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

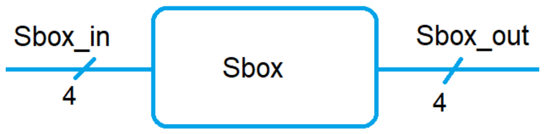

Figure 3 Non-linear S-box

Task: Implement the S-box Look-up substitution table in Table 3.

Figure 4 shows the 16-bit LUTin input goes through four Sboxes to produce 16-bit LUTout. Each 4-bit slice of the LUTin goes through one S-box to produce a 4-bit slice of the LUTout.

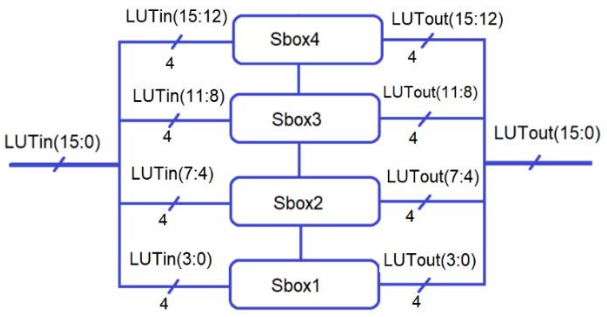

Figure 4 – 16-bit Non-linear Lookup substitution using 4-Sbox'es

Table 5 LUT instruction and control encoding

LUT ctrl	Mnemonic	Instruction Function

1111	LUT	LUTout = { S_box(LUTin[15:12]),
		S_box(LUTin[11:8]),
		S_box(LUTin[7:4]),
		S_box(LUTin[3:0]) };

Task: Implement the 16-bit non-linear lookup substitution in Figure 4 in SYSTEMVERILOG.

1.4 The Processing Unit

We can now combine the three SystemVerilog functional components developed above to create a processing unit in Figure 5, performing 16 operations. The 4-bit `ctrl` signal controls which function to achieve. As shown in Figure 5, **tri-state multiplexing** connects the outputs from the three functional blocks to the shared result bus.

The tristate logic can be defined using the ternary operator. The condition for each combinational block's output was its 'enable' line. These enable signals were provided by the "Control Logic" unit. If the enable signal was HIGH (at logic 1), the combinational block's output would be passed on to the *Result* output of the processor. If the enable signal was LOW (at logic 0), all bits of the connection would be high impedance (Z). **It is crucial that the output signal Result be a *wire* rather than a *logic* signal to have multiple drivers.** In this case, three sources are driving the *Result* wire. However, this does not lead to bus contention as *the enable signals are all mutually exclusive*. So, when one combinational block's output signal drives the *Result* signal, the other two outputs will always be high impedance (Z), meaning only one driver exists.

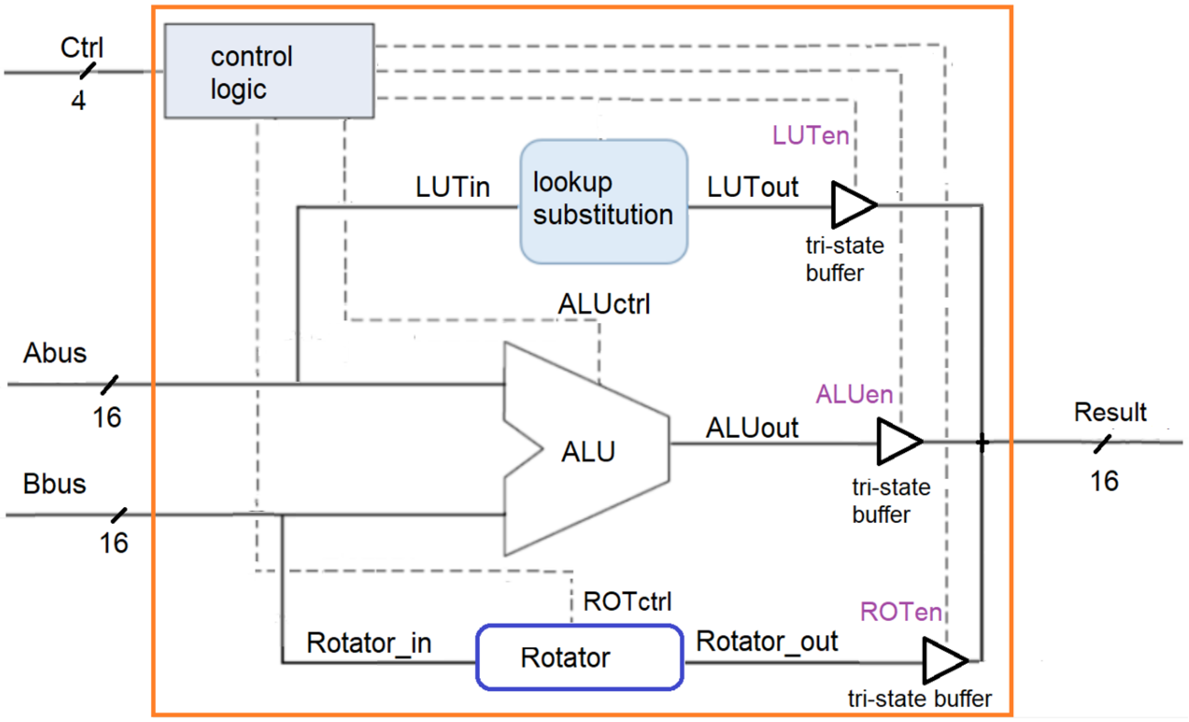

Figure 5 – The Processing Unit

The processing unit has two 16-bit data inputs (`Abus`, `Bbus`), a 4-bit control input (`ctrl`) and a 16-bit output (`result`). The Control Logic unit decodes the 4-bit `ctrl` codes into 16 sets of control signals for the three processing components to achieve the desired functions, as indicated in Table 1, and enable the connection of the relevant component outputs to the shared result bus.

Tasks:

1. Design the *control logic unit using a separate module* to derive the control signals required to implement the processing unit in Figure 5.
2. Develop a SystemVerilog structural description for the processing unit Figure 5. The structural description of the processing unit should instantiate the ALU, the Rotator, the Lookup Substitution combinational block, and the Control Logic unit as *components*.

1.5 SystemVerilog Test Bench for the Processing Unit

Task: Develop a SystemVerilog test bench for the processing unit you designed in section 1.4. There are a total of 36-bit inputs (16 (Abus) + 16(Bus) + 4(ctrl)), which have over 68 billion (68,719,476,736) input combinations. It is not feasible to test all the input combinations. You should select some test inputs and test each instruction in Table 1 at least once with those selected test inputs.

Test inputs with 16 consecutive binary bits, such as `16'b1010010100111100,` are hard to read. Instead, you can separate each four-bit group with an underscore, such as `16'b1010_0101_0011_1100`, which is much easier to read. The underscores do not change the value of a binary number. You can also use hexadecimal digits, such as `X"A53C"`.

At a minimum, test each of the 16 instructions (i.e., the instruction ctrl inputs from X"0" to X"F")at least once with two data inputs, Abus set to X"AC97" and Bbus set to "5368." The testbench should automatically verify if the processing unit produces the expected results for each operation with the specific operands.

*You should read the test instructions from a file named "**test_vector.mem**" and log the test results to another file named "**test_results.txt**". Include the "test_vector.mem" and the "test_results.txt" files in your lab report.*

Section 2: Single-cycle crypto-coprocessor

We need to design a memory for our crypto-coprocess first.

2.1 RAM

Fundamental to implementing any processor is random-access memory (RAM) - an addressable storage unit that provides read/write controls such that higher-level software can program sequences of operations. The RAM is an array of storage units that can store instructions, operands and results.

The FPGA (Field Programmable Gate Array) fabric includes embedded memory elements: dedicated block RAMs (BRAMs), distributed RAM built up with Look-Up-Tables (LUTs), and shift registers. Distributed RAM is faster than Block RAM and is used when not much RAM is needed.

Data is *written* into RAM *synchronously* at either rising or falling clock edges, whether distributed RAM or dedicated Block RAM. Data is usually read *asynchronously from Distributed RAM* and *synchronously from Block RAM*.

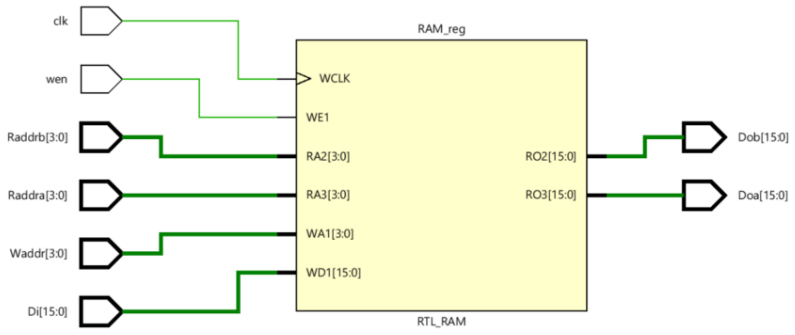

Figure 6 RAM

Task: Implement a Distributed 16x16-bit RAM shown in Figure 6. The RAM has two read ports and one write port and provides 1) *Asynchronous read* and 2) *falling-edge synchronous write*.

The specification of the RAM module ports is given below.

```
module RAM #(parameter N = 16, M = 4) ( // N-bit word, M-bit address
    input clk,
    input [M-1:0] Raddra, // read address A, M
    output [N-1:0] Doa,   // data A out
    input [M-1:0] Raddrb, // read address B
    output [N-1:0] Dob,   // data B out
    input [M-1:0] Waddr,  // write address
    input wen,            // write enable
    input [N-1:0] Di      // data in
);


// Add your RAM functionality here

endmodule
```

Vivado supports RAM initialisation in SystemVerilog. You can provide the initial values when the RAM is defined to initialise the RAM contents. The following SystemVerilog fragment defines and initialises the values of the 16 x 16-bit RAM.

RAM definition and Initialisation Values --Hexadecimal Values

```
// Define a memory array
reg [N-1:0] RAM [0:(2**M)-1];

// Initialize the RAM array with specific values
initial begin
    RAM[0] = 16'h0001;
    RAM[1] = 16'hc505;
    RAM[2] = 16'h3c07;
    RAM[3] = 16'hd405;
    RAM[4] = 16'h1186;
    RAM[5] = 16'hf407;
    RAM[6] = 16'h1086;
    RAM[7] = 16'h4706;
    RAM[8] = 16'h6808;
    RAM[9] = 16'hbaa0;
    RAM[10] = 16'hc902;
    RAM[11] = 16'h100b;
    RAM[12] = 16'hc000;
    RAM[13] = 16'hc902;
    RAM[14] = 16'h100b;
    RAM[15] = 16'hb000;
end
```

2.2 Single-cycle Crypto-Coprocessor

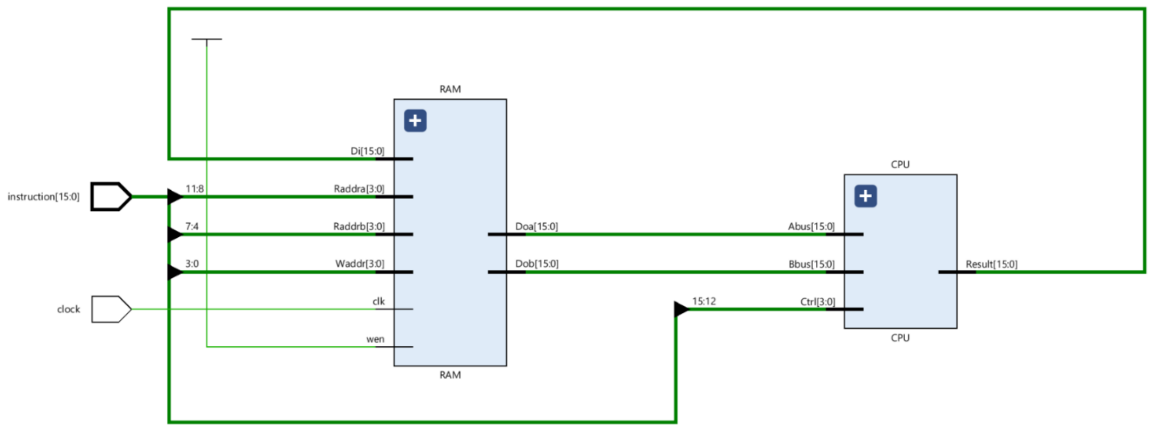

Figure 7 Crypto-coprocessor

Figure 7 shows the schematic of the Crypto-coprocessor. It consists of the RAM in Figure 6 as memory to store data and results and the Processing Unit in Figure 5 as CPU to execute instructions. The inputs to the Crypto-coprocessor are a 16-bit instruction word and a clock signal. The RAM *write-enable* signal `wen` is held high so the RAM can store the result of the operations.

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

15 0

Instruction[15:12]	Instruction[11:8]	Instruction[7:4]	Instruction[3:0]
OPCODE	Raddra	Raddrb	Waddr

Task: Implement the Crypto-processor in Figure 7.

2.4 Test Program

Assuming the RAM is initialised to the values in Section 2.1, test the Crypto-coprocessor by running the following sequence of instructions saved in a file called *`test_program.mem`*. (“`Add or create simulation sources`” `->` “`Create File`” `->` (Set File type to `Memory File`, Set File name to *`test_program`*)`.` Then, “`Simulation Sources`” `->` “`Memory File`” `->` click open the “`test_program.mem`” file and copy the following instructions into the `test_program.mem` file and save.).

```
054c   // ADD  R5   R4   R12   -->   [(0x“f407” + 0x“1186”) mod 2^16 = 0x“058d”]
5187   // XOR  R1   R8   R7    -->   [0x“c505” ^ 0x“6808” = 0x“ad0d”]
d0c0   // ROR4 --   R12  R0    -->   [ror4(0x“058d”)    = 0x“d058”]
b093   // ROL4 --   R9   R3    -->   [rol4(0x“baa0”)    = 0x“aa0b”]
007a   // ADD  R0   R7   R10   -->   [(0x“d058” + 0x“ad0d”) mod 2^16 = 0x“7d65”]
7c01   // MOV  R12  --   R1    -->   [move 0x“058d” in R12 to R1]
173c   // SUB  R7   R3,  R12   -->   [(0x“ad0d” - 0x“aa0b”)mod 2^16 = 0x“0302”]
2ca9   // MUL  R12  R10  R9    -->   [(0x“030d” * 0x“7d65”)mod 2^16 = 0x“29ca”]
c002   // ROL8 --   R0   R2    -->   [ROL8(0x“d058”)     = 0x“58d0”]
34b6   // AND  R4   R11  R6    -->   [0X“1186” and 0x“100b” = 0x”1002”]
f504   // LUT  R5   --   R4    -->   [LUT(0x“f407”) = 0x”29cd”]
6108   // NOT  R1   --   R8    -->   [NOT (0x“058d”) = 0x“fa72”]
a05f   // ROL3 --   R5   R15   -->   [rol3(0x”f407”) = “0x“a03f”]
80be   // ROL1 --   R11  R14   -->   [rol1(0x“100b”) = 0x“2016”]
904d   // ROL2 --   R4   R13   -->   [rol2(0x“29cd”) = 0x“a734”]
```

```
e044    // ROR8 --  R4   R4    -->    [ror8(0x‟29cd”) = 0x‟cd29”]
4beb    // OR   R11 R14  R11   -->    [0x‟100b” or 0x‟2016” = 0x‟301f”]
```

The 16-bit instructions are each represented using four **hexadecimal digits**, anything after `//` are comments for the line

The explanation of the first instruction `054c` is as follows:

```
054c   // ADD  R5  R4   R12   --> [(0x‟f407” + 0x‟1186”) mod 2^16 = 0x‟058d”]
```

Instruction[15:12]	Instruction[11:8]	Instruction[7:4]	Instruction[3:0]
OPCODE	Raddra	Raddrb	Waddr
0	**5**	**4**	**c**

From the instruction set in Table 1, you can see the encoding and the function of the ADD operation:

0	000	ADD	ALUout = (Abus + Bbus) mod $2^{wordsize}$

The following is the comment for this instruction.

```
// ADD  R5  R4   R12   --> [(0x‟f407” + 0x‟1186”) mod 2^16 = 0x‟058d”]
```

The first instruction is to add the contents in RAM address five and RAM address 4, then modulo 2^{16}, i.e. keep the least significant 16-bits and omit the carry bit. Store the result in RAM address 12.

At the system initialisation, the content of register 5 is `0x‟f407”`(the prefix `0x` represents a hexadecimal number), and RAM address 4 contains `0x‟1186”`.

```
                0x‟f407”       1111 0100 0000 0111
+               0x‟1186”       0001 0001 1000 0110
---------------------------------------------
               0x‟1058d”      10000 0101 1000 1101
```

mod 2^{16} operation drops the carry bit **1**

```
                0x‟058d”       0000 0101 1000 1101
```

Interpret the other instructions similarly.

Task: Develop a testbench to run the above sequence of instructions (stored in the *test_program.mem*) on the Crypto-coprocessor. The testbench should ***read the test program instructions from a file*** called *test_program.mem*. Write the final contents of the RAM after executing the test program to a file called “*RAM_final_contents.txt*.”

-- **End of Assignment** --

Submit the following

You should upload the following two separate files to Loop:

1. **File 1**: named **ID_Number-Surname-EEN1056-report.PDF** containing a well-structured assignment report**.** Include the complete SystemVerilog code for each section in your report. Note that 5% is for well-organised and presented reports. A messy or disorganised report will lose marks. The report should include:

    i. SystemVerilog code for each design (the ALU, the Rotator, the Sbox and the LUT lookup, the Control unit, etc..), with a brief explanation.
    ii. The elaborated logic circuit schematics from each design.
    iii. SystemVerilog code for the testbench for the CPU processing unit and the testbench for the crypto-coprocessor.
    iv. Relevant sections of test simulation waveforms with brief explanations.
    v. The *final contents of the RAM* in hexadecimal format *after running the test program* on the crypto-coprocessor. It would be best to verify the RAM values in your report.

2. **File 2:** named **ID_Number-Surname-EEN1056-code.ZIP** containing all the SystemVerilog source code files (**only those .sv files, .mem files, and .txt files).** Although your source codes have been included in the PDF format report, submitting separate source code files can facilitate the verification of your code using Vivado when necessary.

MARKING SCHEME

Section	Marks
1.1. ALU	10%
1.2. Rotator	10%
1.3. Sbox and LUT	5%
1.4. Processing unit (including control logic unit)	10%
1.5. Processing unit testbench	10%
2.1. RAM (Register file memory)	10%
2.2. Complete Crypto-Coprocessor	10%
2.3 Test Program Correct final RAM contents (checked by running the test program)	25% 5%
Well-structured and presented lab report, well-written, well-commented SystemVerilog code	5%