DUBLIN CITY UNIVERSITY
SCHOOL OF ELECTRONIC ENGINEERING

EE540 VHDL Design & Synthesis

Assignment

Aidan Smyth

ID Number: 13452192

29/11/17

Declaration on Plagiarism

Assignment Submission Form

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I have read and understood the Assignment Regulations. I have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

I have read and understood the referencing guidelines found at
http://www.dcu.ie/info/regulations/plagiarism.shtml ,

https://www4.dcu.ie/students/az/plagiarism and recommended in the assignment

guidelines.

Name: Aidan Smyth Date: 26/11/17

Table of Contents

Introduction

Digital Cryptography

Digital cryptography is essential to providing security for digital applications. Computationally complex, digital cryptography require hardware implementation, typically using FPGAs, to accelerate the algorithms and provide adequate performance. A typical cryptography application is the implementation of a cryptographic co-processor tailored for generic cryptographic functions. Regardless of mathematical complexity, modern symmetric algorithms employ similar techniques for encryption and decryption. These include:

- Key Addition
- Substitution
- Permutation
- Look up tables
- Shifter
- ALU operations

Assignment Structure

This assignment implements some of these functions following the structure outlined in the assignment criteria [1]. Section 1 covers the design of the combinational components which make up the structural logic of the crypto-processor. Section 2 details the synchronous registers and memories which are required for storing the data processed by the combinational logic. The complete crypto-processor is tested using a test program detailed in the assignment criteria.

Run Instructions

To run the simulation for Section 1, set “Top_tb” as the top simulation source. Run for an additional 5ms after simulation.

To run Section 2 simulation, set “Test Program” as the top simulation source. No additional run time is required after simulation

1. Combinational Logic

To remain flexible and adaptable for several cryptographic applications, the co-processor must include standard instructions as well as dedicated functions specific for security. The three main components of the co-processor include:

- ALU
- Lookup Tables
- Shifter

1.1 N-Bit Adder Design in Verilog

The ALU requires addition and subtraction functionality. The N-bit full adder, shown in Figure 1, can be implemented by N full-adders in series. This can be designed in Verilog or VHDL, and the criteria of this assignment requires a Verilog module design.

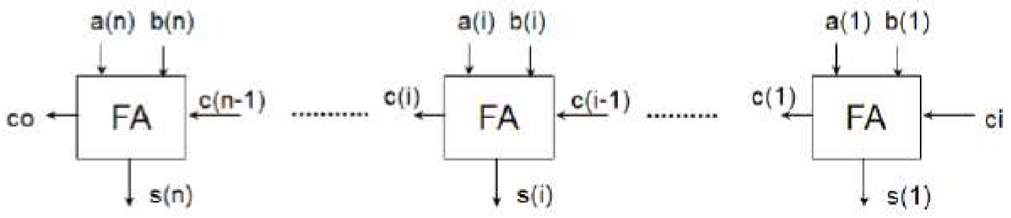


Figure 1 - N-bit Adder [1]

First, a full adder Verilog module must be designed. Full adder logic for the sum and carry outputs for two inputs A, B and a carry in (CIN):

```
SUM = (A xor B) xor CIN

COUT = (A and B) or (A and CIN) or (B and CIN)
```

In Verilog, the code for this module is quite short, and is listed in the code snippet below. The next step is to generate the N-bit adder in Verilog, instantiating and generating 16 full adders in series. The first full adder takes the carry input, which is high if subtraction is required (A - B == A + not(B) + 1). Verilog generate command generates the remaining 15 full adders, with the final internal carry out assigned to the N-bit adder carry out value.

```
generate
   for(i=0;i<N;i=i+1)
     begin: generate_N_adder
   if(i==0)
  full_adder_verilog f0 (A_BUS[0],B_BUS[0],CIN,SUM[0],c_internal[0]);
   else
  full_adder_verilog fi (A_BUS[i],B_BUS[i]
        ,c_internal[i-1],SUM[i],c_internal[i]);
     end
  assign cout = c_internal[N-1];
   endgenerate
```

Vivado elaborates the design to provide a schematic representation of the design. The elaborated schematics are displayed in Figure 2, Figure 3, and Figure 4. N is configured to 16 in this instantiation.

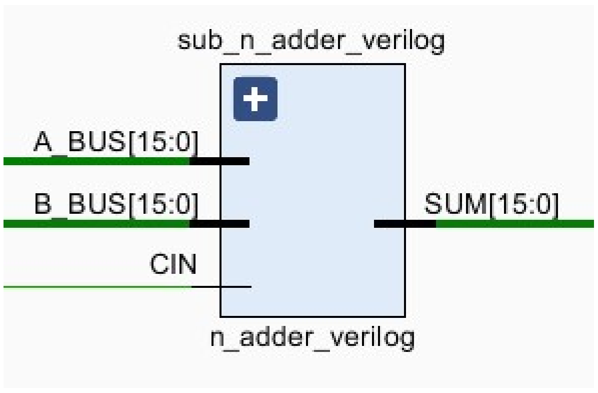


Figure 2 - N-bit adder block component

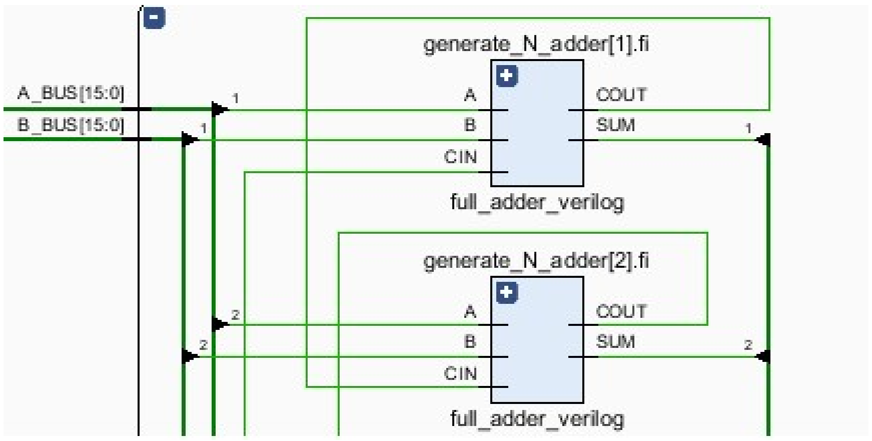


Figure 3 - Close up of internal structure of N-bit adder

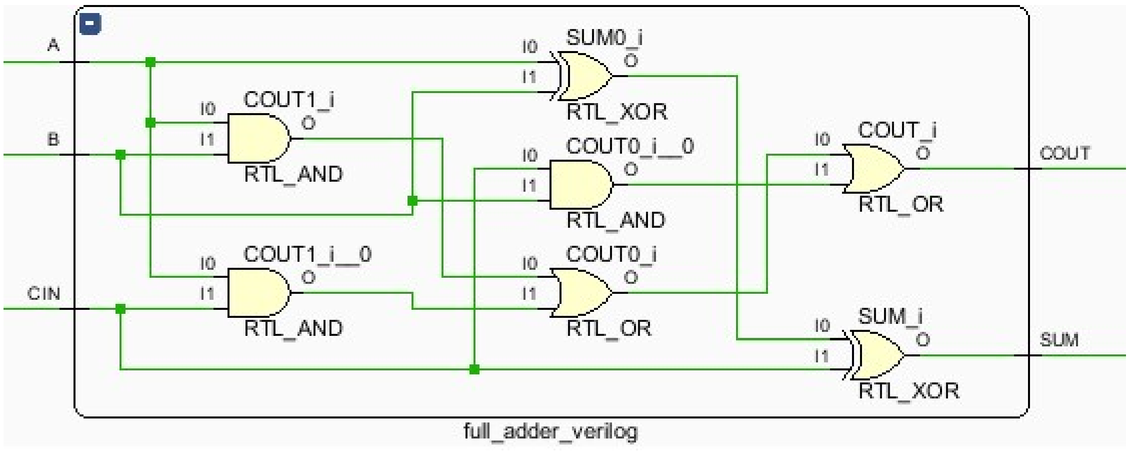


Figure 4 - Internal logic of one bit full adder

1.2 ALU Design

The Arithmetic Logic Unit (ALU) shown in Figure 5 is a key component of any co-processor. The ALU performs operation on the data buses given a control operation defined by its architecture. Vivado supports mixed language design, so the ALU can be designed in VHDL and instantiate the previously designed Verilog N-bit adder for addition and subtraction operations.

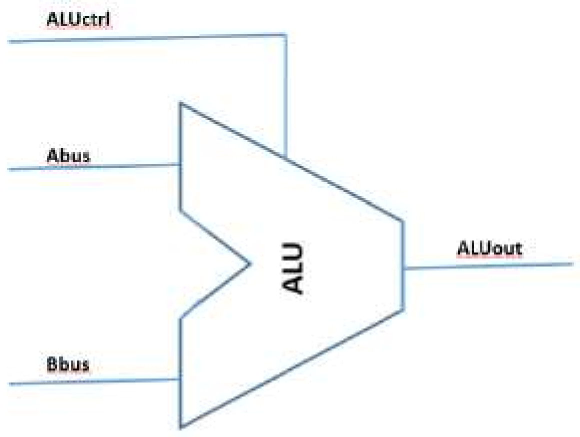


Figure 5 – ALU [1]

The mnemonics and instruction functions for the ALU operations are displayed in the table below.

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

It is straightforward to instantiate mixed language modules in VHDL, the only additional step is to ensure that all port mappings are explicitly declared (this is good practice anyway).

```
add_n_adder_verilog: n_adder_verilog port map(
A_BUS=>A_BUS,B_BUS=>B_BUS,CIN=>'0',SUM=>sum_op);
sub_n_adder_verilog: n_adder_verilog port map(
A_BUS=>A_BUS,B_BUS=>not_b,CIN=>'1',SUM=>sub_op);
```

To prevent multiple drivers from driving the ALU output to unknown 'X' values, each ALU operation output is stored in signals. The ALU is then assigned an output depending on what the control value case is. The subtraction uses the N-bit full adder, inverting the negative value and adding one to perform two's compliments conversion.

```
ALU_OUT<=    sum_op when CTRL = "0000" else
             sub_op when CTRL = "0001" else
             and_op when CTRL = "0010" else
             or_op when CTRL = "0011" else
             xor_op when CTRL = "0100" else
```

```
        not_op when CTRL = "0101" else
        mov_op when CTRL = "0110" else
        (others => '0');
```

The elaborated schematics for the design are presented in Figure 6 and Figure 7. There seven operations in total, which are multiplexed to the ALU output using the control operator. The carry out of the adder and subtractor elements is ignored.

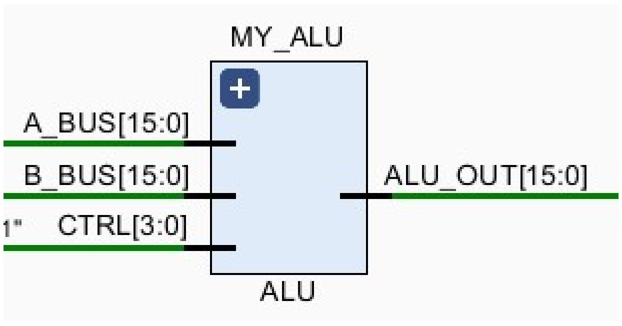


Figure 6 - ALU block component

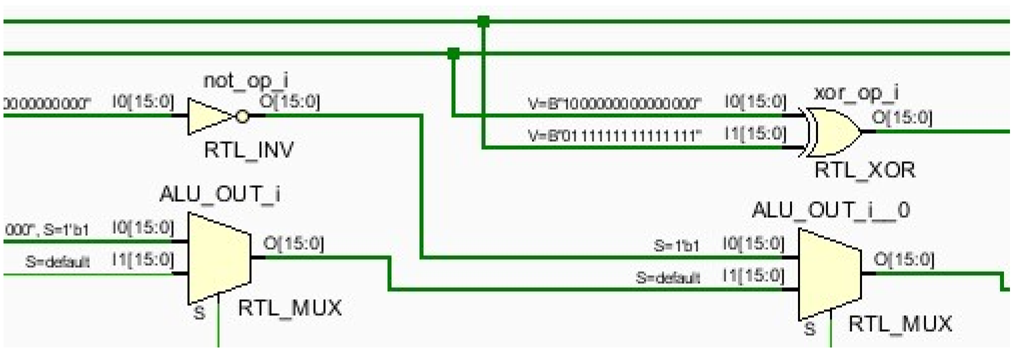


Figure 7 - NOT and XOR functions inside ALU

1.3 Shifter Design in VHDL

The shifter is commonly used in the permutation and transposition of ciphers, shifting and rotating data. It is an essential component for any crypto-processor.

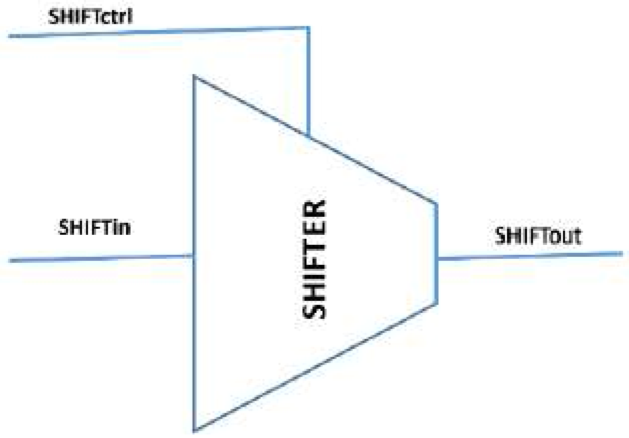
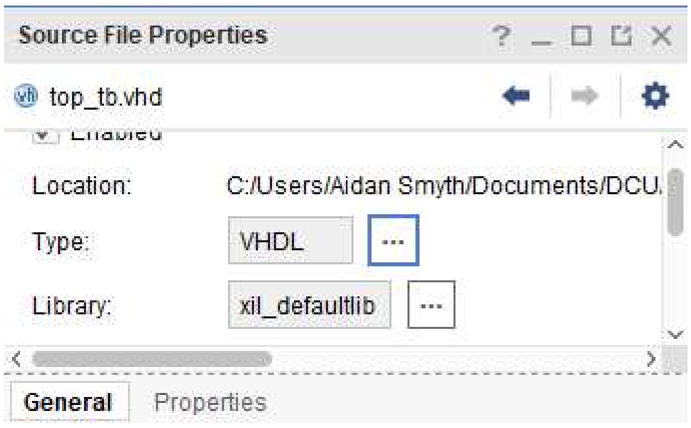


Figure 8 – Shifter [1]

This assignment implements a simple shift and rotate functions that act on nibbles from both data buses, rather than implementing a complex shifter. The mnemonics and controls for the various shifter operations are listed in the table below.

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 bits

The VHDL 2008 language supports the use of rotate and shift mnemonics. VHDL 2008 is not by default used in Vivado, and must be set in the "Source File Properties" found by right clicking the source file.



Location: C:/Users/Aidan Smyth/Documents/DCU.

Type: VHDL

Library: xil_defaultlib

Figure 9 - Source file properties window

Similarly to the ALU, a case statement multiplexes the shifter signals to the output, assigning zeros to the output if the CTRL does not match any shift operations.

```
process(SHIFTIN, SHIFTCTRL)
    begin
    case SHIFTCTRL is
    when "1000" => --ror4
          SHIFTOUT <= SHIFTIN ror 4;
    when "1001" => --rol4
          SHIFTOUT <= SHIFTIN rol 4;
    when "1010" => --sll4
          SHIFTOUT <= SHIFTIN sll 4;
    when "1011" => --slr4
          SHIFTOUT <= SHIFTIN srl 4;
    when others =>
          SHIFTOUT <= (others => '0');
    end case;
    end process;
```

The elaborated design of the 16 bit shifter reveals the shifter schemtaics shown in Figure 10 and Figure 11.

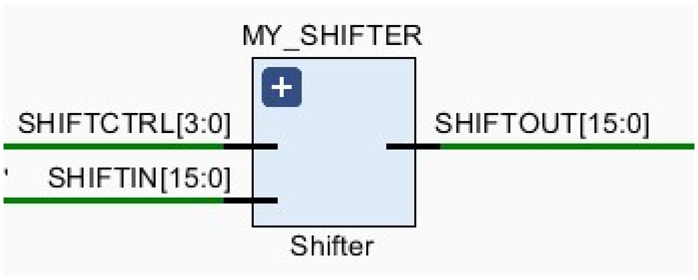


Figure 10 - Shifter schematic

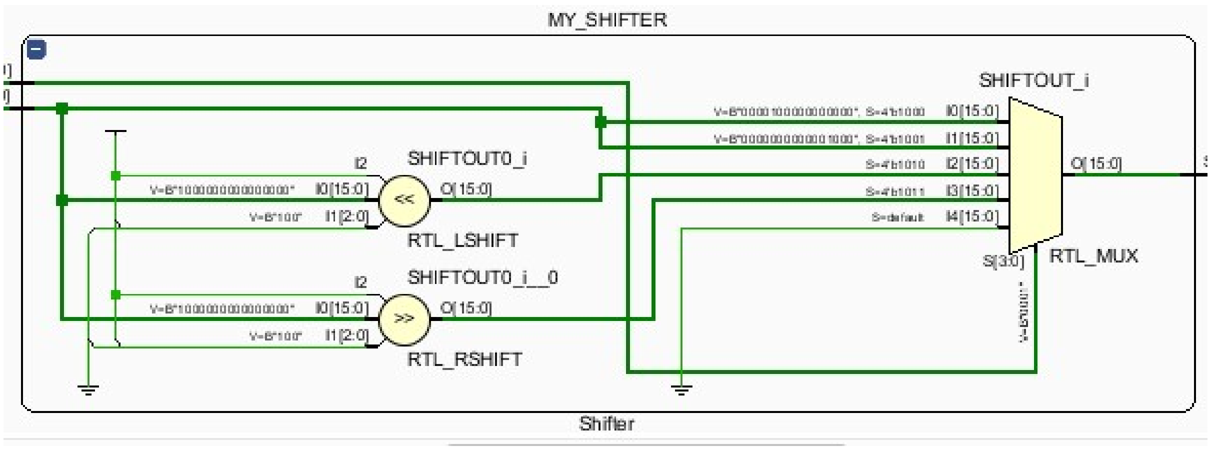


Figure 11 - Inside the shifter component

The shift functions can be clearly identified as components. Note that the rotate functions are simply re-wiring of the input logic.

1.4 Non-Linear Lookup Operations

Non-linear mapping obscures the relationship between inputs and outputs in digital symmetric encryption algorithms. The basic operation of this lookup function is to take one byte or nibble and map it to an output pre-defined by the function.

Lookup tables can be implemented using complex mathematical functions, a single large memory lookup table, or a series of smaller lookup tables as demonstrated in this assignment. The lookup tables S1 and S2 are displayed in the tables below.

S1 for 4 bit input																
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

S2 for 4 bit input																
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 12 shows the lookup table substitution operator. LUTin is a 16 bit input. The least significant byte is used in the shift operation, with the least significant nibble of the byte passing through S2 and the most significant nibble of the byte passing through S1. The most significant byte of the 16 bit input is unaffected. The lookup substitution is performed only when LUTen is high.

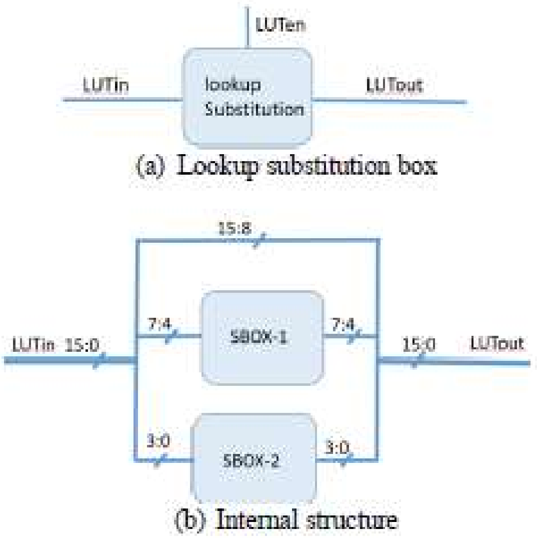


Figure 12 - Lookup table substitution [1]

LUTen is high only when **CTRL is "1100"**.

In VHDL design, the two lookup substition functions, S1 and S2, are designed as components to be instantiated in a final wrapper shifter component. The LUT is defined as an array of std_logic_vectors of size 4 bits. A temporary signal converts the input bits to unsigned ints, which are substitued for corresponding values pre-defined in the array.

```
architecture LUT of LUT_S1 is
    subtype LUKOUT is std_logic_vector (3 downto 0);
    type LUT is array (natural range 0 to 15) of LUKOUT;
    constant lookuptable1:   LUT := (       --S2
        "0001", "1011", "1001", "1100",
        "1101", "0110", "1111", "0011",
        "1110", "1000", "0111", "0100",
        "1010", "0010", "0101", "0000"
        );
    signal tmp: std_logic_vector (3 downto 0);
begin
    tmp <= lookuptable1(TO_INTEGER(unsigned(LUTIN)));

    (LUTOUT(3),LUTOUT(2),LUTOUT(1),LUTOUT(0))<=LUKOUT'(tmp(3), tmp(2),
tmp(1), tmp(0));
end architecture;
```

These two lookup tables are included in the main lookup table design

```
  --Assign values to input signals
    LINS1 <= LUTIN(7 downto 4);
    LINS2 <= LUTIN(3 downto 0);

    -- Perform Lookup table operations
    Lookup_S1:  LUT_S1 port map(LINS1,LOUTS1);
    Lookup_S2:  LUT_S2 port map(LINS2,LOUTS2);


    -- assign signals to output when CTRL mathces, else output is zeros
    LUTOUT(15 downto 8) <= LUTIN(15 downto 8) when CTRL = "1100" else
(others => '0');
    LUTOUT(7 downto 4) <= LOUTS1 when CTRL = "1100" else (others => '0');
    LUTOUT(3 downto 0) <= LOUTS2 when CTRL = "1100" else (others => '0');
```

Similarly to the ALU and LUT, when the CTRL is not the correct value, the LUT output is zeros. This is for the purpose of the next section’s structural design which incorporates all three components.

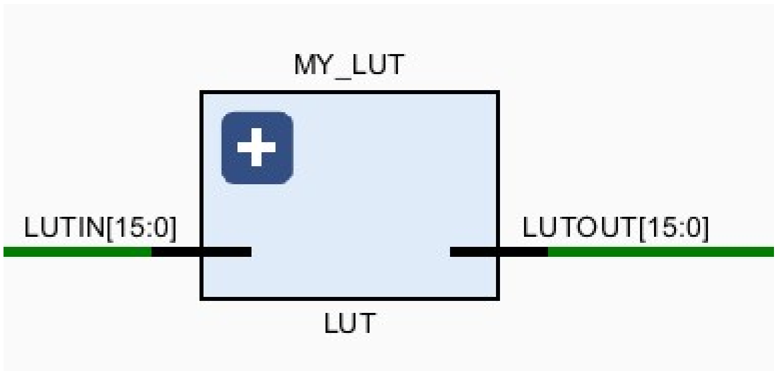


Figure 13 - LUT Component

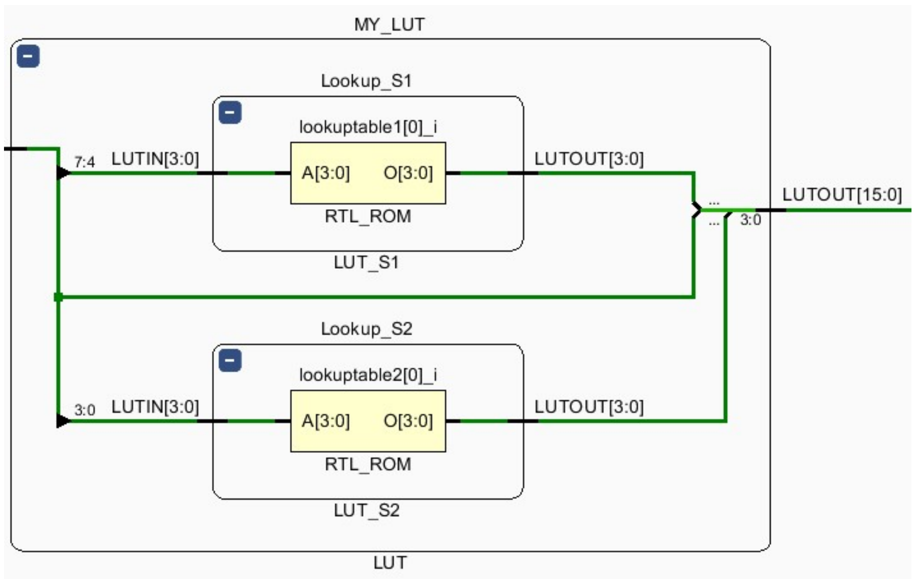


Figure 14 - S1 and S2 LUT operating on lower 8 bits

1.5 Structural VHDL Model

The previous sections have developed VHDL & Verilog components with 12 different operational modes for two configurable N-bit outputs. These modes are controlled by the CTRL input. Structural VHDL combines the logic functions as shown in Figure 15.

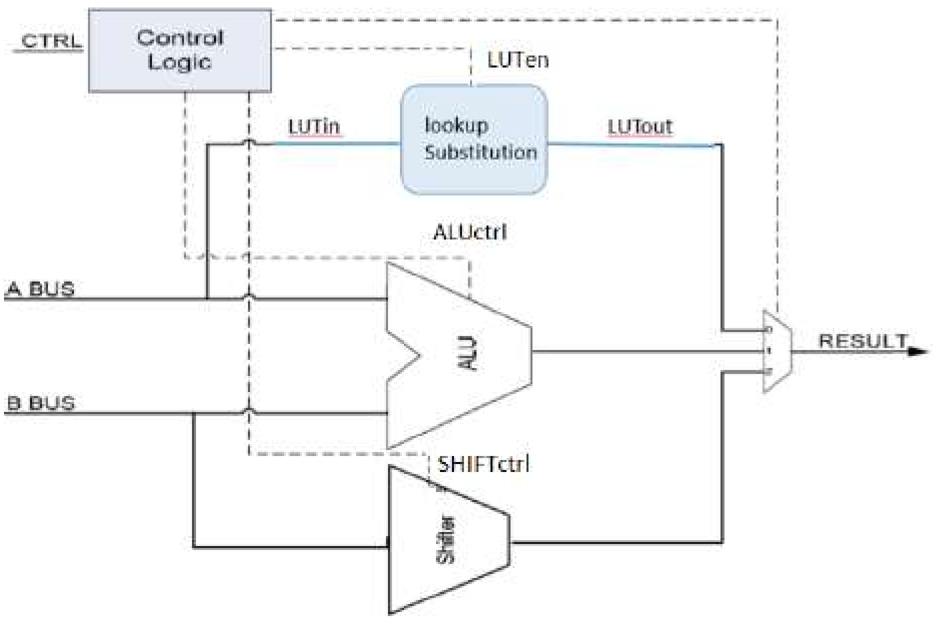


Figure 15 – Structural model design [1]

The design consists of two 16-bit bus inputs, a 4-bit control bus and a 16-bit output. All possible 16 input control states are designed. The structural VHDL is the top combinational logic component of the crypto-processor.

A difficulty which may arise when designing the structural VHDL is that multiple driver may set the output to unknown ‘X’ states. It is important that the output is assigned values in a way that ensures the outputs of different CTRL functions do not interfere with each other.

This is ensured by the three components (ALU, shifter and LUT) returning zeros if their CTRL is not enabled. The three components are instantiated and their outputs are assigned to signals. The output of the Structural design is simply an OR operation between these three signals, as only one of the signals can have non-zeros value at any given CTRL operation.

```
MY_ALU:       ALU port map(CTRL,A_BUS,B_BUS,ALU_RESULT);
MY_SHIFTER:   SHIFTER port map(CTRL,A_BUS,SHIFTER_RESULT);
MY_LUT:       LUT port map(A_BUS,CTRL,LOOKUP_RESULT);

process(ALU_RESULT,SHIFTER_RESULT,LOOKUP_RESULT)
begin
        -- use of brackets for faster timing & efficient gates
        RESULT <= ALU_RESULT or (SHIFTER_RESULT or LOOKUP_RESULT);
end process;
```

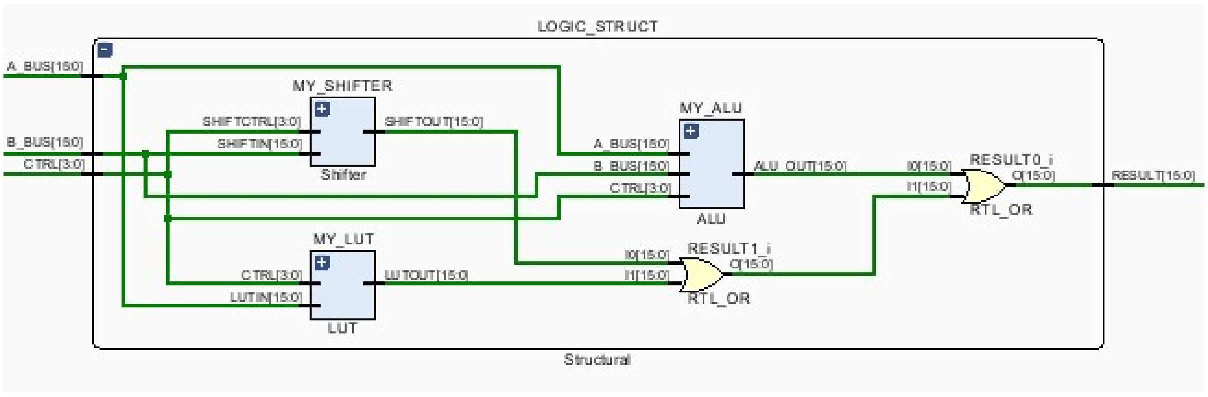


Figure 16 - Structural elaborated design

The schematics of the elaborated design are shown in Figure 16. The inputs, LUT, Shifter and ALU components can be clearly seen, and the two OR gates assign their outputs to the result.

1.6 VHDL Test Bench for Structural Design

A VHDL testbench is designed for verifying the operation of the structural combinational logic of the processor. The top testbench module connects a stimulus generator component to the structural design component.

```
  -- Component Instantiations
  my_STRUCT: STRUCTURAL

  PORT MAP (
             A_BUS     => A_BUS,
             B_BUS    => B_BUS,
             CTRL    => CTRL,
             RESULT  => RESULT
 );

  tb_stim_gen: stimgen
  PORT MAP (
            A_BUS      => A_BUS,
            B_BUS    => B_BUS,
            CTRL    => CTRL,
            RESULT  => RESULT
           );
```

In the stimulus all twelve modes of operation are tested in for loops, typically 32768 iterations per loop. With approximately 400,000 tests taken, a large amount of permutations are covered. The buses are controlled by incrementing integer variables that are then converted to unsigned std_logic_vectors and assigned to the buses. The output of the buses are then converted to hex format. There are two options, print “true/false” console messages, or output the results to a text file for verification using an external program.

The second option is preferred, due to the massive amount of vectors tested. A C++ program can then convert the hex values to integers and compute the pass/fail rate relatively straight forward. The following code snippet shows how to open a text file for logging the test bench results:

```
file outFile : text is out "C:/Users/Public/tb_bus.txt";  -- set output
file name
  variable NL: line;
  variable A_temp, B_temp, i : integer;

  BEGIN
        write(NL, string'("A_BUS  B_BUS CTRL  Result"));  -- include
heading in file
        writeline(outFile,NL);
```

The following code snippet demonstrates the loop for each control mode:

```vhdl
A_temp := 0;
B_temp := 0;
i := 1;
------------------------
---- ADD ---------------
------------------------
ASSERT FALSE
REPORT "BEGIN TEST FOR ALU ADD" SEVERITY NOTE;

CTRL_TB <= "0000";
wait for T1;

for i in 1 to 65536/2 loop

    A_BUS <= STD_LOGIC_VECTOR(to_unsigned(A_temp,N));
    B_BUS <= STD_LOGIC_VECTOR(to_unsigned(B_temp,N));
    RESULT_TB <= RESULT;
    wait for T1;
    hwrite(NL, STD_LOGIC_VECTOR(to_unsigned(A_temp,N)));
    write(NL, string'(" "));
    hwrite(NL, STD_LOGIC_VECTOR(to_unsigned(B_temp,N)));
    write(NL, string'(" "));
    hwrite(NL, CTRL_TB);
    write(NL, string'(" "));
    hwrite(NL, RESULT_TB);
    writeline(outFile,NL);
    A_temp := A_temp + 2;
    B_temp := B_temp + 1;
  end loop;
```

The stimulus must be run for 5ms after simulation to cover all test bench scenarios. This run time can be observed from the simulation wave in Figure 17. Note the changes in CTRL value every operational mode loop. The test bench covers all possible modes of operation for the structural design.

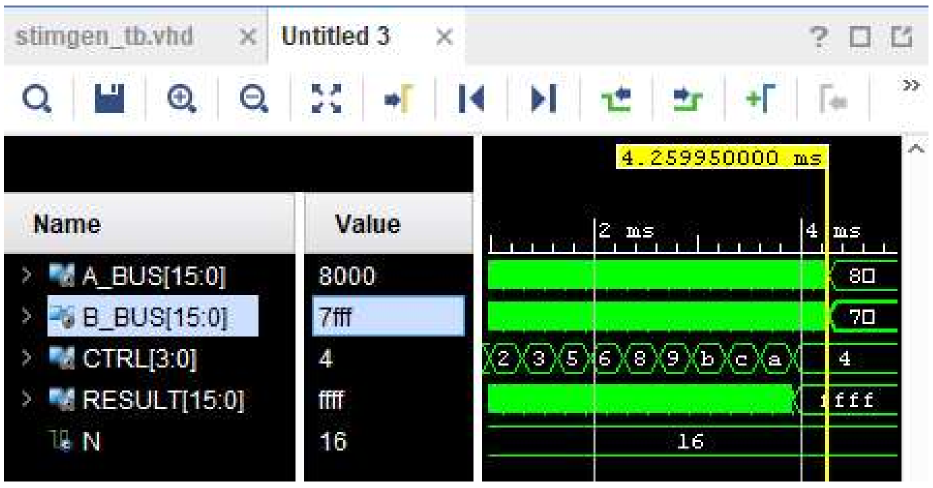
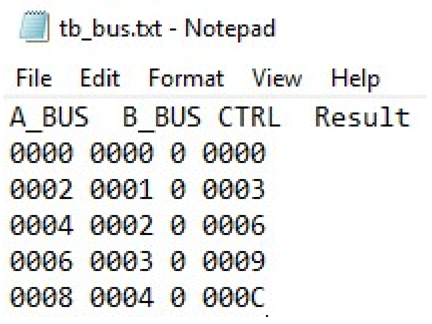


Figure 17 - Stimulus ran after simulation

Taking a look at the text file, we can see the correct operation of each of the 12 operation modes.



```
A_BUS  B_BUS CTRL  Result
0000 0000 0 0000
0002 0001 0 0003
0004 0002 0 0006
0006 0003 0 0009
0008 0004 0 000C
```

Figure 18 – Testbench file output add results

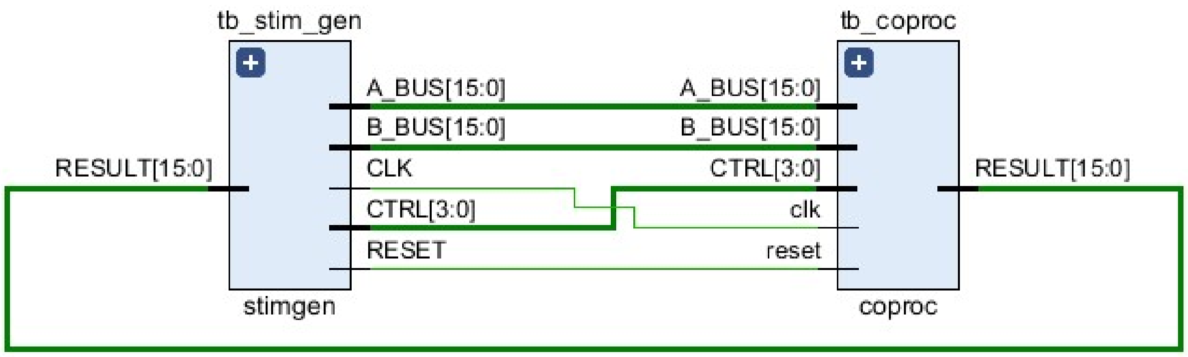


Figure 19 - Coprocessor and test bench schematic

2. Synchronous Logic Design

This section builds on the design in Section 1, implementing synchronous storage elements such as registers and memory. The final part of this section combines the components in the final crypto-processor and designs a test program to validate the design.

2.1 Registers

Digital systems rely on synchronicity to control data flow within a logic device. A clock signal schedules the data flow, preventing glitches from the input propagating to the output instantaneously. Synchronicity also removes the issue of knowing when an input signal results in an output that is valid. Edge triggered registers are implemented as processes which respond to the edges of a global clock signal.

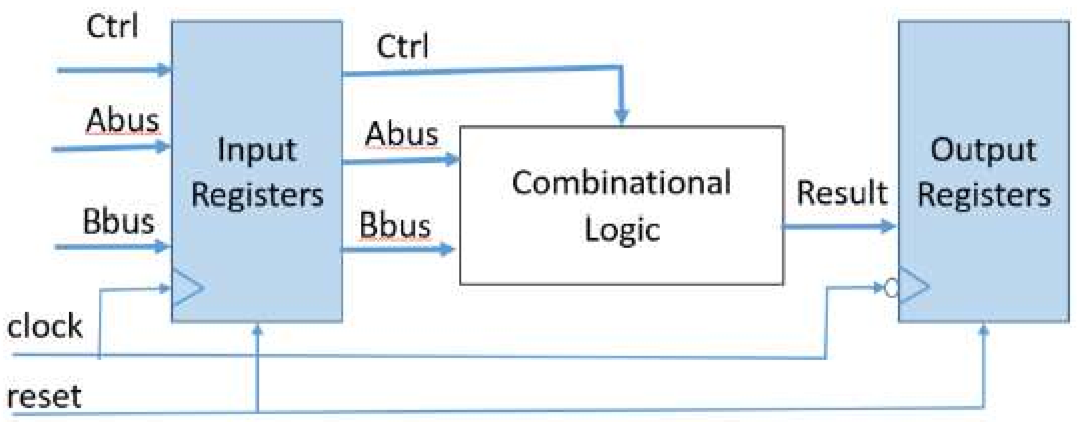


Figure 20 – Synchronous register system [1]

Figure 20 shows a synchronous coprocessor system. The input values are registered on the rising edge of the clock, and the output registers are falling-edge triggered. The existing N-bit combinational logic designed in Section 1 is the central component to the synchronous system. If the reset is high, the register values are set to zero.

The input register architecture is outlined in the following code snippet:

```
ARCHITECTURE Behavioural OF OutputReg IS
BEGIN
        PROCESS (clk, RESET)
        BEGIN
                -- asynchronous reset (active high)
                IF RESET = '1' THEN
                        RES_OUT <= (OTHERS => '0');

                ELSIF (falling_edge(clk)) THEN
                        RES_OUT <= RES_IN;
                END IF;

        END PROCESS;

END Behavioural;
```

The output register architecture is described in the following code snippet:

```
ARCHITECTURE Behavioural OF InputReg IS
BEGIN
        PROCESS (clk, RESET)
        BEGIN
                -- asynchronous reset (active high)
                IF RESET = '1' THEN
                        A_OUT <= (OTHERS => '0');
                        B_OUT <= (OTHERS => '0');
                        CTRL_OUT <= (OTHERS => '0'); --reset to ADD
operation
                ELSIF (rising_edge(clk)) THEN
                        A_OUT <= A_IN;
                        B_OUT <= B_IN;
                        CTRL_OUT <= CTRL_IN;
                END IF;
        END PROCESS;
END Behavioural;
```

The elaborated schematic of the two registers is shown in Figure 21, and the internal layout of the input register is displayed in Figure 22.

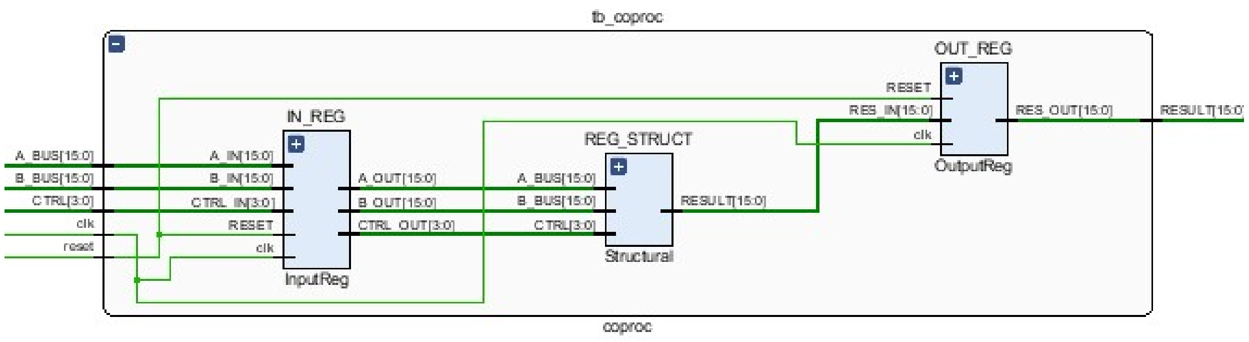


Figure 21 - Input and output register schematics

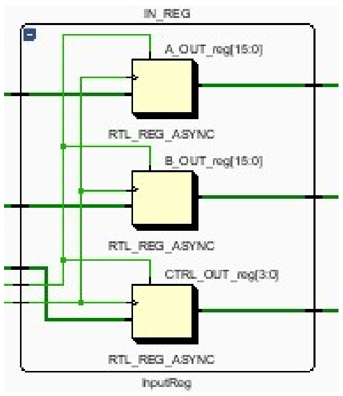


Figure 22 - Input register schematic

The two registers can be combined in memory, which will be explained in the next section.

2.2 Memory

Register modelling inevitably leads onto the modelling of memory elements within the digital system. Register files are integral to programmable processors. The register file is an array of registers that which are used to store operands and variables. In VHDL design this can be easily designed using an M-bit array of N-bit vectors. In this case both M and N will be 16.

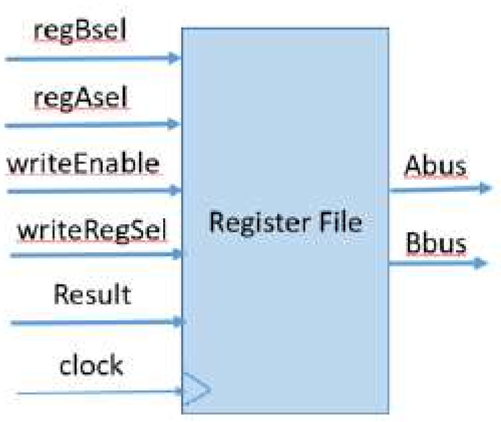


Figure 23 - Register file design [1]

Vivado supports RAM initialization in VHDL. The RAM is a user-defined memory type array of std_logic_vectors. The register file is initialized by giving it initial values, which are listed below. These values are essential to the validation of the testing program in Section 2.4.

```
-- Register File Initial 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'));
```

The write and read processes for the register file are synchronous and rising edge triggered. Each cycle, two registers are read and the result is written (if writing is enabled). An additional reset option is included for resetting the values in registers to zero if reset triggered.

```
IF (rising_edge(clk)) THEN
        -- Read and reset functions
        IF (RESET = '1') THEN
                ReadA <= x"0000";
                ReadB <= x"0000";
        ELSE
                ReadA <= reg_file(to_integer(unsigned(RegA)));
                ReadB <= reg_file(to_integer(unsigned(RegB)));
        END IF;
```

A data bypass exists so that the value written is forwarded directly to the output if read and write are from the same register in a single cycle.

```
-- Write functions
IF (En = '1') THEN -- write high
        reg_file(to_integer(unsigned(RegR))) <= RES;
        IF (RegA = RegR) THEN – Data Bypass for read A
                ReadA <= RES;
        END IF;
        IF (RegB = RegR) THEN – Data Bypass for read B
                ReadB <= RES;
        END IF;
END IF;
```

The elaborated schematic for the register file is shown in Figure 24

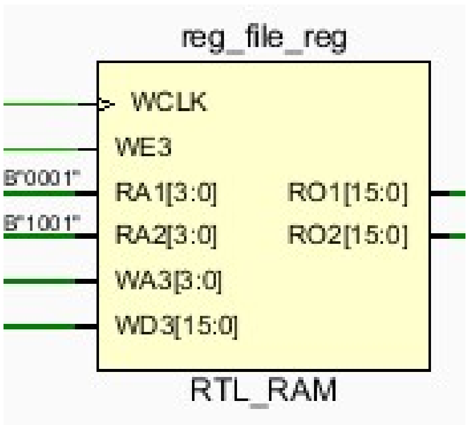


Figure 24 - RAM schematic

2.3 Complete Crypto Processor

The register file is combined with the structural combination logic design from Section 1 to create a complete crypto-processor, as illustrated in the assignment specification shown in Figure 25.

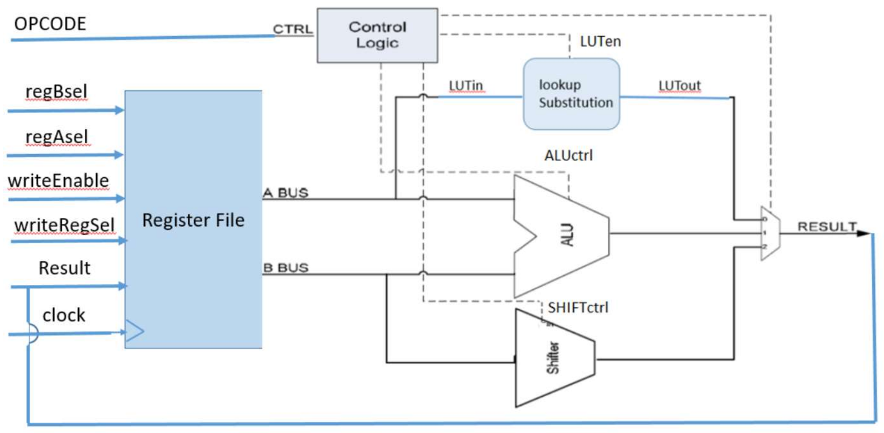


Figure 25 - Complete crypto-processor design [1]

The inputs to the crypto-processor include a 16-bit instruction word, reset, clock, and write enable. The 16 bit instruction word takes the form of four separate instruction inputs:

OPCODE	RegA index	RegB index	RegR (write) index

The synchronous system also handles an addition CTRL OPCODE mnemonic:

OPCODE	MNEMONIC	Instruction Description
0111	NOP	No Operation Performed

The output of the combinational logic is not stored in memory when the OPCODE is NOP. This is handled in the following code snippet:

```
IF (CTRL = "0111") THEN -- NOP Instruction
        Write_En <= '0';
ELSE
        Write_En <= '1';
END IF;
```

Flaw: The assignment outlines a flaw in the circuit design in Figure 25. As the instruction word is loaded on a rising clock edge, the destination index will be present for the current instruction but will not be present for the next clock cycle, by which time another instruction will be loaded. This will lead to the result being delayed and written to the wrong address. To ensure single cycle execution, this must be addressed in the design.

Solution: The flaw is fixed using temporary signals that act as wires between the register file and structural file. The signals are assigned the destination register on the rising edge, and are able to write a result to the register on the same clock cycle as the instruction word is loaded. The same idea is used to update the CTRL value on the rising edge.

The following code snipper outlines the solution to the potential design flaw:

```
if(rising_edge(clk)) then
        if(reset = '1') then
                CTRL_tmp <= x"0";
                RegR_tmp <= x"0";
        else
                RegR_tmp <= RegR;       -- Handles flaw
                CTRL_tmp <= CTRL;
        if(CTRL = "0111") then          -- NOP Instruction
                Write_En <= '0';
        else
                Write_En <= '1';
```

The design flaw solution is tested in the test bench program in the following section, where it is proven to operate correctly.

The complete coprocessor is elaborated and its schematics shown in Figure 26 and Figure 27. As expected, the register file and structural design are connected, with the processor output being stored in the register file.

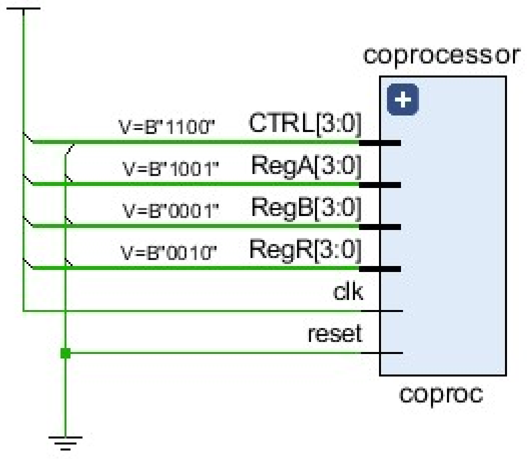


Figure 26 - Coprocessor component

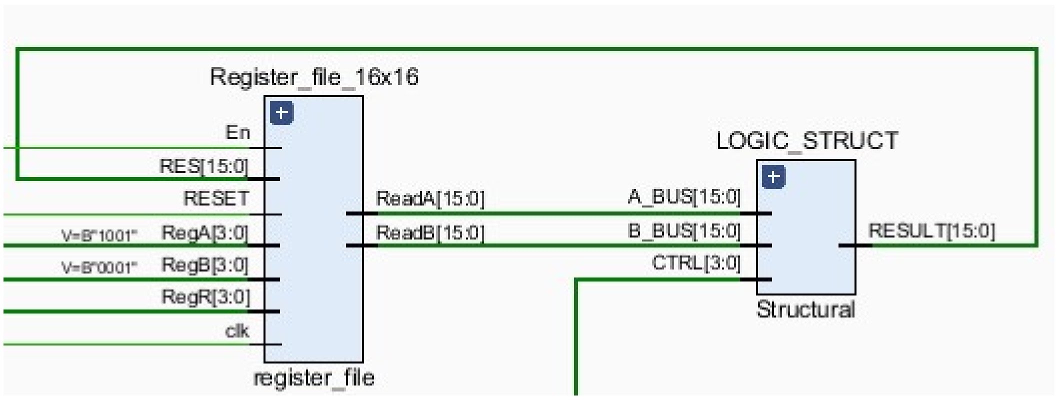


Figure 27 - Coprocessor internal structure

2.4 Test Program

After a system reset, the register file contains the values given in the table below. The following instruction words are implemented in a stimulus generator (and verified manually) to verify that the register and structural logic elements of the crypt-processor are working as expected:

MNEMONIC	RegA	RegB	RegR (destination)
ADD	R5 (f407)	R4 (1186)	R12 (058d)
XOR	R1 (c505)	R8 (6808)	R7 (ad0d)
ROR4	-	R12 (058d)	R0 (d058)
SLL 4	-	R9 (baa0)	R3 (aa00)
ADD	R0 (d058)	R7 (ad0d)	R10 (7d65)
SUB	R7 (ad0d)	R3 (aa00)	R12 (030d)
NOP	-	-	-
AND	R12 (030d)	R10 (7d65)	R9 (0105)
NOP	-	-	-
LUT	R9 (0105)	-	R2 (011e)

The vectors are tested directly from the stimulus generator, as is shown in the code snippet below. There is a wait of 1 clock period between each operation. This is allow the synchronous register file to operate correctly. All three correct register values are instantiated, and the wait of the additional clock cycle is enough time for the correct output to be assigned to the destination register.

A clock process is created, with a period of 20ns, plenty of cycle time for the crypto-processor to operate.

```
constant clk_period : time := 20 ns;

CLK_PROC: process
  begin
    clk <= '0';
    wait for clk_period/2;
    clk <= '1';
    wait for clk_period/2;
  end process;
```

The coprocessor component is declared in the test program:

```
coprocessor: coproc port map(
    clk => clk, reset => reset, RegA => RegA, RegB => RegB,
    ctrl => ctrl, regR => regR  );
```

The test program 16 bit instruction word is read from the text file “test_prog.txt”. The word is split and assigned to the CTRL and separate registers. A for loop repeats this instruction word read for all 10 instructions in the file.

Open file to read:

```
  file_open(test_prog, "C:/Users/Public/test_prog.txt", read_mode);
```

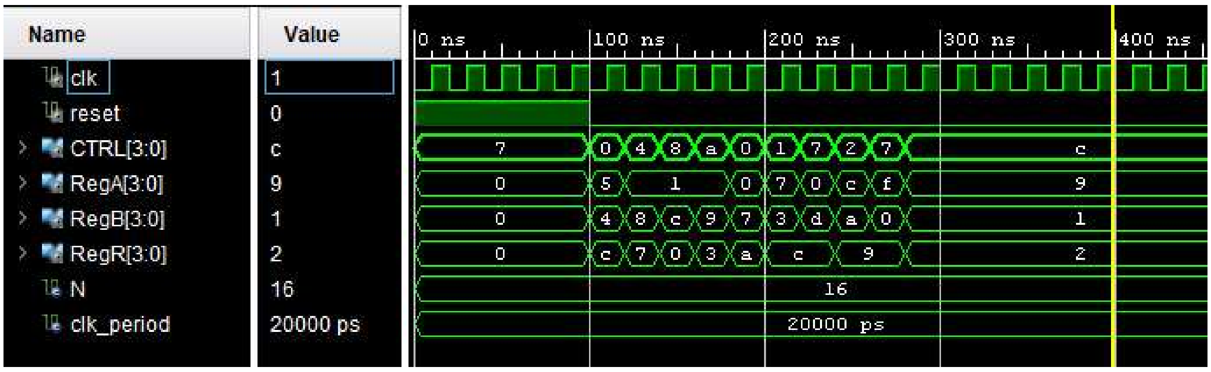
Read from the file:

```
for i in 1 to 10 loop
    -- Read instruction word
    readline(test_prog, v_ILINE);
    read(v_ILINE, v_instr);
    -- Assign to registers
    RegA <= v_instr(11 downto 8);
    RegB <= v_instr(7 downto 4);
    RegR <= v_instr(3 downto 0);
    CTRL <= v_instr(15 downto 12);
    wait for clk_period;
 end loop;
```

The expected/predicted transition of the register's hexadecimal values is described in the table below.

Register Index	Hex Values
R0	0001 -> d058
R1	c505
R2	3c07 -> 011e
R3	d405 -> aa00
R4	1186
R5	f407
R6	1086
R7	4706 -> ad0d
R8	6808
R9	baa0 -> 0105
R10	c902 -> 7d65
R11	100b
R12	c000 -> 058d -> 030d
R13	c902
R14	100b
R15	b000

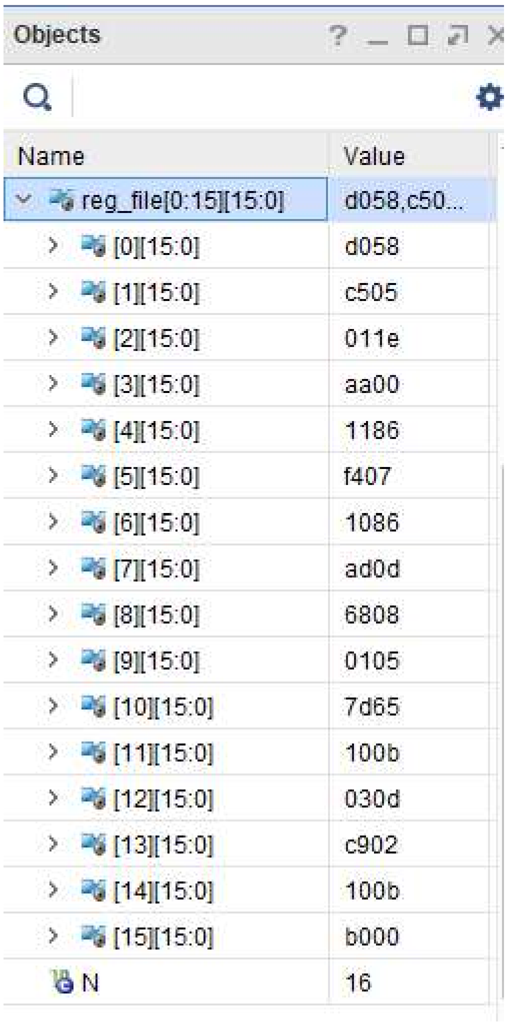
The test program must be set as top before simulation. After simulation, the waves in Figure 28 can be observed. The waves verify that the correct register values and CTRL instructions are being assigned.



Name	Value
clk	1
reset	0
CTRL[3:0]	c
RegA[3:0]	9
RegB[3:0]	1
RegR[3:0]	2
N	16
clk_period	20000 ps

Figure 28 - Test program simulation waves

The predicted register file values after simulation are compared to the final register file memory. The predicted register values match the actual register file values in Figure 29. It is clear that the test program has successfully verified the operation modes chosen in the stimulus generator. It is observable that the destination register flaw has been corrected, there is no delay with the assignment of the destination register.



Objects

Name	Value
reg_file[0:15][15:0]	d058,c50...
[0][15:0]	d058
[1][15:0]	c505
[2][15:0]	011e
[3][15:0]	aa00
[4][15:0]	1186
[5][15:0]	f407
[6][15:0]	1086
[7][15:0]	ad0d
[8][15:0]	6808
[9][15:0]	0105
[10][15:0]	7d65
[11][15:0]	100b
[12][15:0]	030d
[13][15:0]	c902
[14][15:0]	100b
[15][15:0]	b000
N	16

Figure 29 - Register file values after simulation

3. Code

Code is attached in the zip file, along with VHDL projects for Part 1 and Part 2.

4. References

[1] X. Wang, “EE540 Assignment 2017/2018,” 2017. [Online]. [Accessed 26 11 2017].