EL6463 Advanced Hardware Design

Lab #4

Name: Chen Shen

netID: cs5236

Test Cases

In my test bench, I used 3 `din` for encryption and decryption.

`din = x"0000000000000000"`

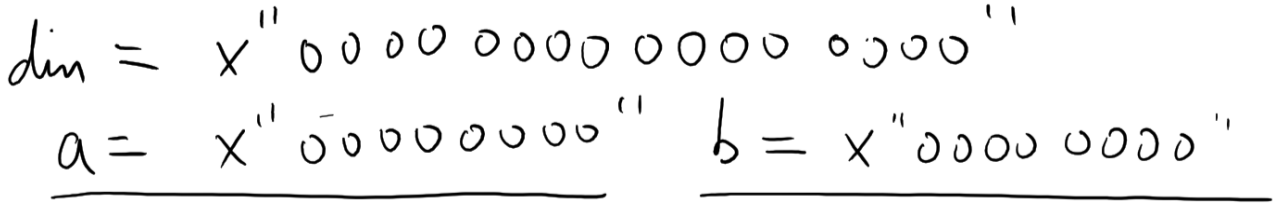
`din = x"0123456789abcdef"`

`din = x"ffffffffffffffff"`

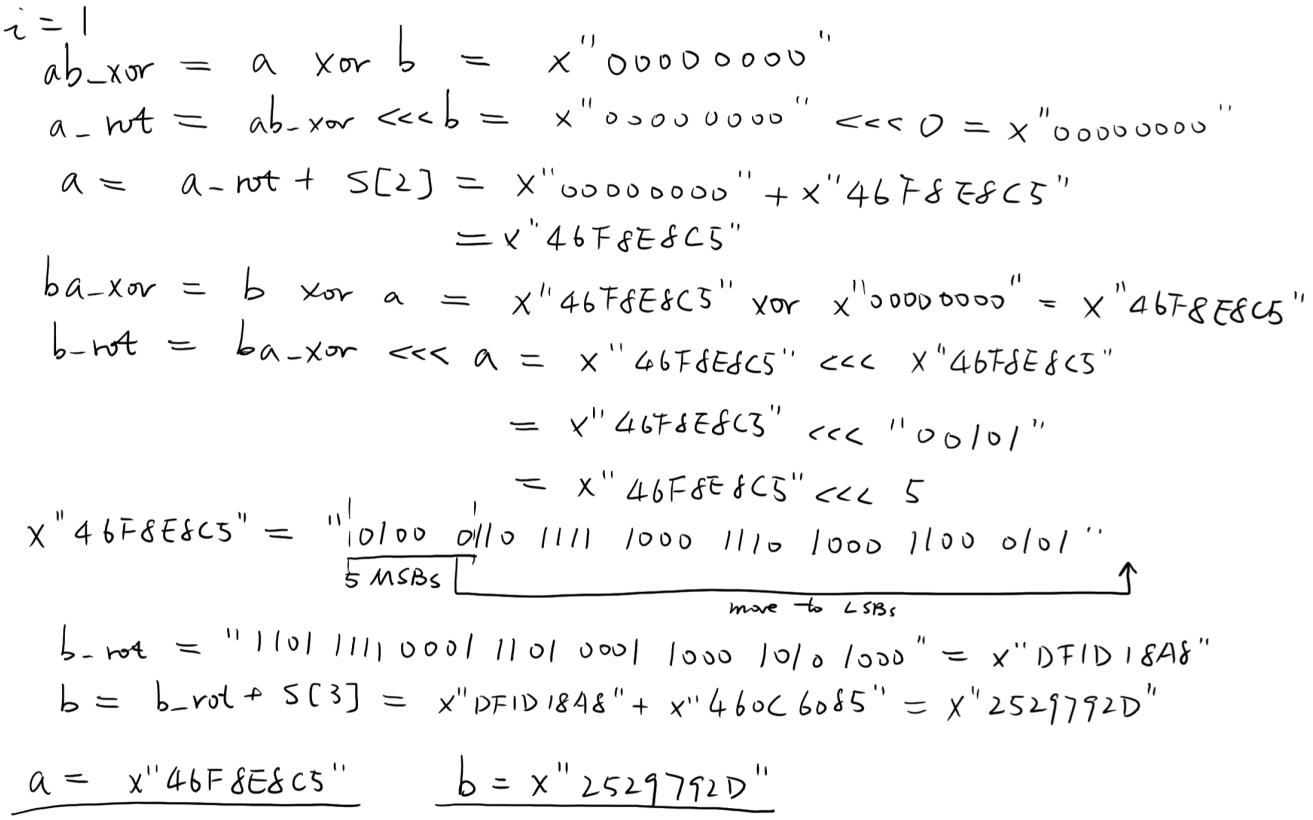
Hand Calculation

Encryption: `din = x"0000000000000000"`

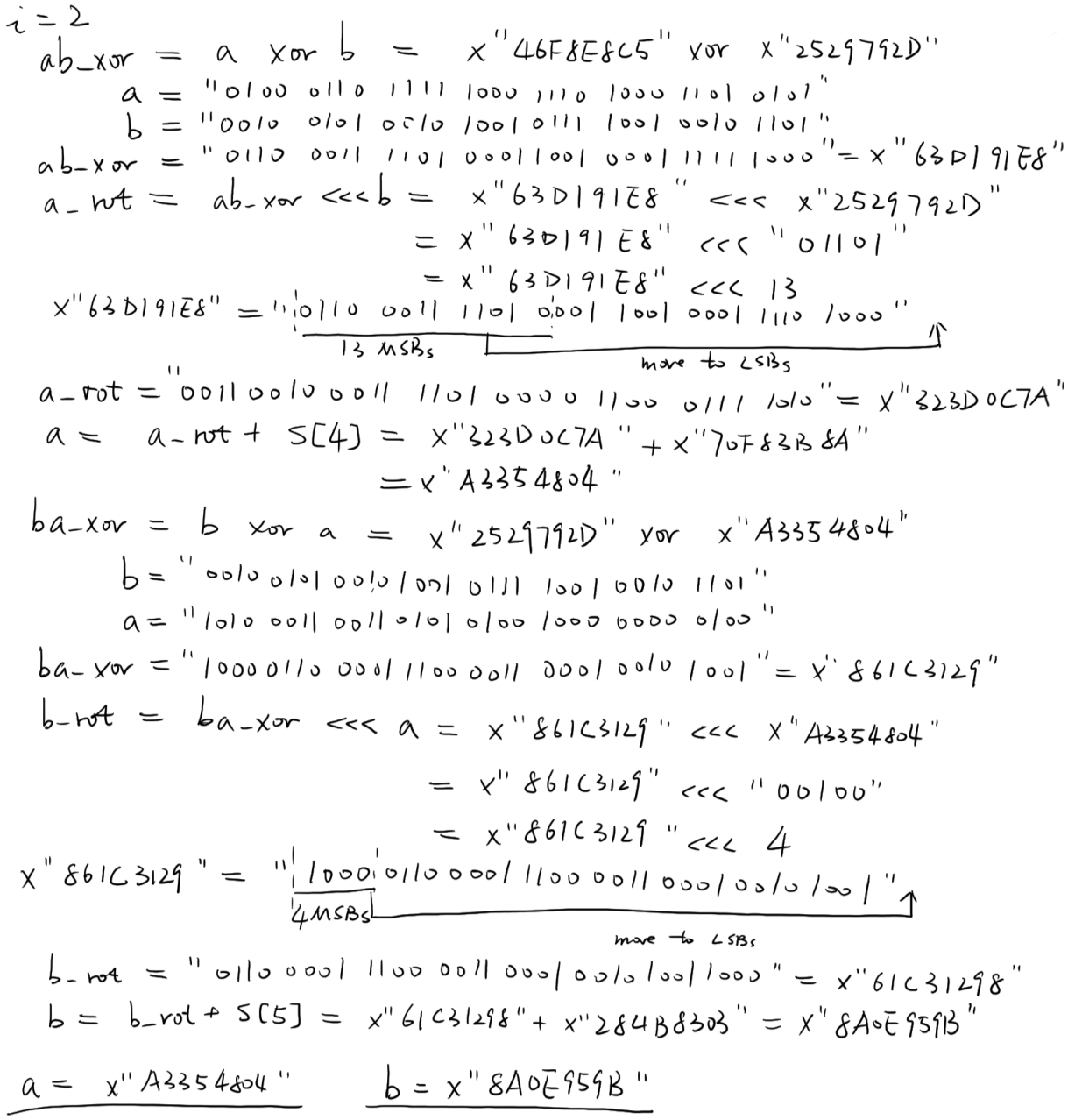
Before the main loop, we have `A = x"00000000"` and `B = x"00000000"`.



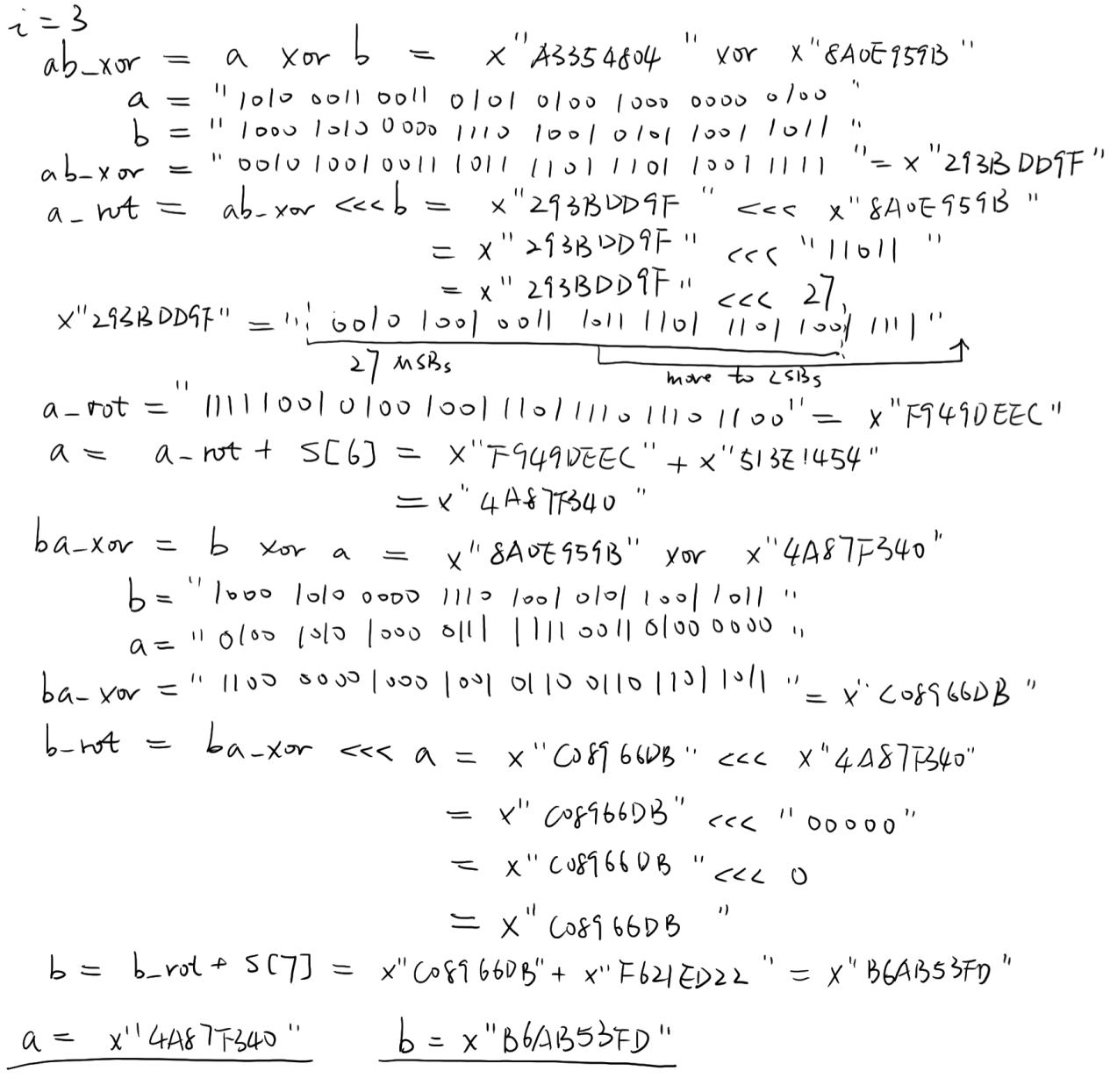
In the first loop, after the first line, we have `A = x"46f8e8c5"`. After the second line, we have `B = x"2529792d"`.



In the second loop, after the first line, we have `A = x"a3354804"`. After the second line, we have `B = x"8a0e959b"`.

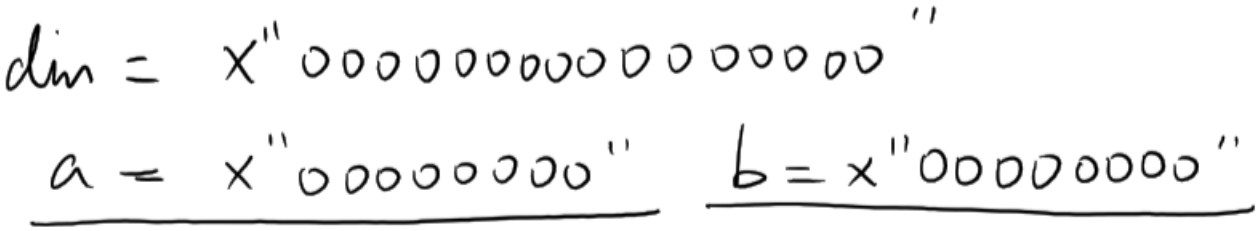


In the third loop, after the first line, we have `A = x"4a87f340"`. After the second line, we have `B = x"b6ab53fd"`.



Decryption: `din` = `x"0000000000000000"`

Before the main loop, we have `A` = `x"00000000"` and `B` = `x"00000000"`.



In the first loop, after the first line, we have `A` = `x"93c8774f"`. After the second line, we have `B` = `x"9afb9c80"`.

$i = 12$

$b\_minus = b - S[25] = x"00000000" - x"65046380" = x"9AFB9C80"$

$b\_rot = b\_minus >>> a = x"9AFB9C80" >>> x"00000000"$

$= x"9AFB9C80"$

$b = b\_rot \text{ xor } a = x"9AFB9C80" \text{ xor } x"00000000" = x"9AFB9C80"$

$a\_minus = a - S[24] = x"00000000" - x"F6CC1431"$

$= x"0933EBCF"$

$a\_rot = a\_minus >>> b = x"0933EBCF" >>> x"9AFB9C80"$

$= x"0933EBCF" >>> "00000"$

$= x"0933EBCF"$

$a = a\_rot \text{ xor } b = x"0933EBCF" \text{ xor } x"9AFB9C80"$

$a\_rot = "0000\ 1001\ 0011\ 0011\ 1110\ 1011\ 1100\ 1111"$

$b = "1001\ 1010\ 1111\ 1011\ 1001\ 1100\ 1000\ 0000"$

$a = "1001\ 0011\ 1100\ 1000\ 0111\ 0111\ 0100\ 1111"$

$= x"93C8774F"$

$a = x"93C8774F"$ $b = x"9AFB9C80"$

In the second loop, after the first line, we have A = x"e99c86aa". After the second line, we have B = x"6130d88b".

i = 11

b_minus = b - S[28] = x"9AFB9C80" - x"43192304" = x"57E2797C"

b_rot = b_minus >>> a = x"57E2797C" >>> x"93C8774F"

= x"57E2797C" >>> "01110"

= x"57E2797C" >>> 15

x"57E2797C" = "0101 0111 1110 0100 1011 1100 1001 0111" (move to MSBs; 15 LSBs)

b_rot = "1111 0010 1111 1000 1010 1111 0100 0011" = x"F2F8AFE4"

b = b_rot xor a = x"F2F8AFE4" xor x"93C8774F"

b_rot = "1111 0010 1111 1000 1010 1111 0100 0011"

a = "1001 0011 1100 1000 0111 0111 0100 1111"

b = "0110 0000 1011 0000 1011 1000 0110 1011" = x"6130D88B"

a_minus = a - S[22] = x"93C8774F" - x"30D76B0A" = x"62F10C45"

a_rot = a_minus >>> b = x"62F10C45" >>> x"6130D88B"

= x"62F10C45" >>> "11010"

= x"62F10C45" <<< 11

x"62F10C45" = "0110 0001 0000 1000 1111 0000 0100 0101" (move to MSBs; 11 LSBs)

a_rot = "1000 0100 0011 1010 0011 1010 0000 1010" = x"843A3A0A"

a = a_rot xor b = x"843A3A0A" xor x"6130D88B"

a_rot = "1000 0100 0011 1010 0011 1010 0000 1010"

b = "0110 0000 1011 0000 1011 1000 0110 1011"

a = "1110 0100 1001 1000 0011 1010 0110 0001" = x"E498361A"

a = x"E498361A" b = x"6130D88B"

In the third loop, after the first line, we have `A = x"4987f97f"`. After the second line, we have `B = x"20b04007"`.

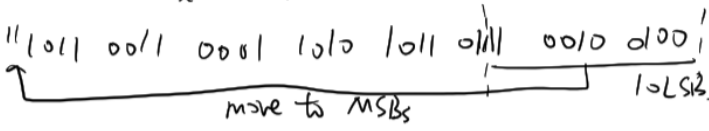
$i = 10$

$b\_minus = b - S[21] = x"61300888" - x"AE162167" = x"B31AB724"$

$b\_rot = b\_minus >>> a = x"B31AB724" >>> x"E99C86AA"$

$= x"B31AB724" >>> "01010"$

$= x"B31AB724" >>> 10$

$x"B31AB719" =$ 

$b\_rot = "1100\ 1001\ 0010\ 1100\ 1100\ 0110\ 1010\ 1101" = x"C92CC6AD"$

$b = b\_rot \text{ xor } a = x"C92CC6AD" \text{ xor } x"E99C86AA"$

$b\_rot = "1100\ 1001\ 0010\ 1100\ 1100\ 0110\ 1010\ 1101"$

$a = "1110\ 1001\ 1001\ 1100\ 0000\ 0110\ 0101\ 0111"$

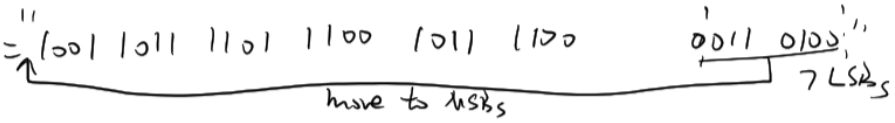
$b = "0010\ 0000\ 1011\ 0000\ 0000\ 0010\ 0000\ 0010" = x"20B04007"$

$a\_minus = a - S[20] = x"E99C86AA" - x"4DBFCA76" = x"9BDCBC34"$

$a\_rot = a\_minus >>> b = x"9BDCBC34" >>> x"20B04007"$

$= x"9BDCBC34" >>> "00111"$

$= x"9BDCBC34" >>> 7$

$x"9BDCBC34" =$ 

$a\_rot = "0110\ 1001\ 1011\ 1101\ 1110\ 0011\ 0100\ 0110" = x"69B7B946"$

$a = a\_rot \text{ xor } b = x"69B7B946" \text{ xor } x"20B04007"$

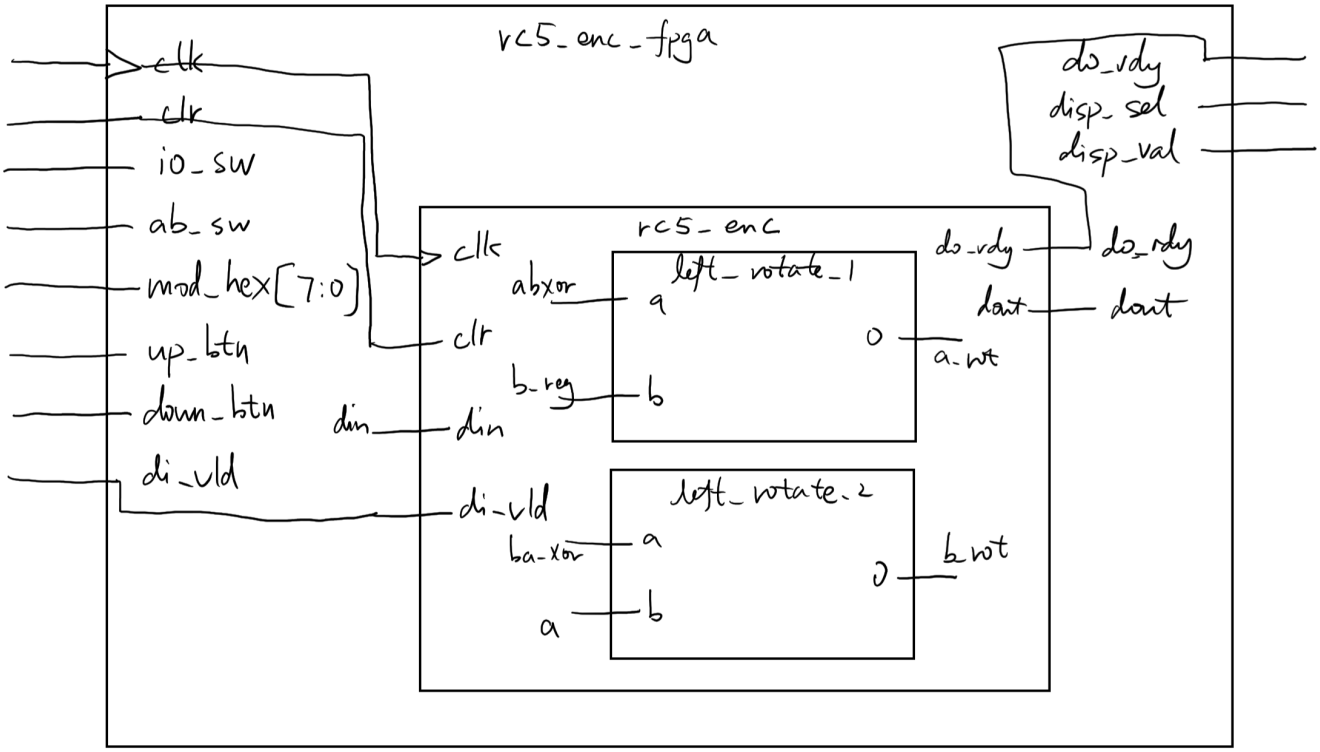
$a\_rot = "0110\ 1001\ 1011\ 1101\ 1110\ 0011\ 0100\ 0110"$

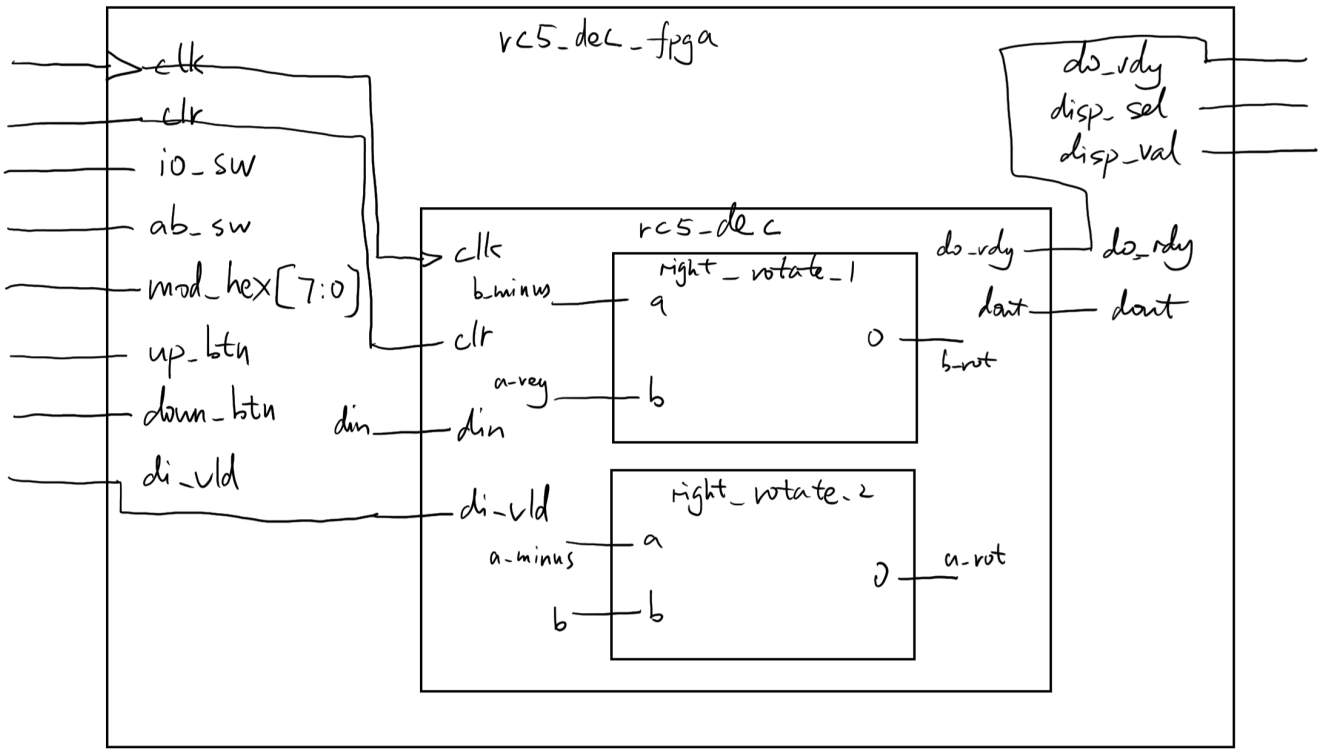
$b = "0010\ 0000\ 1011\ 0000\ 0000\ 0010\ 0000\ 0010"$

$a = "0100\ 1001\ 0000\ 1101\ 1110\ 0001\ 0100\ 0100" = x"490DE144"$

$a = x"490DE144"$ $\qquad$ $b = x"20B04002"$

Block Diagram





Resource Utilization

RC5 Encryption

	Synthesis stage	Place and Route stage
LUT and FF pairs usage	510 LUTs and 230 FFs	186
IOB usage	32	32
RAM/DSP blocks used (if any)	0	0

RC5 Decryption

	Synthesis stage	Place and Route stage
LUT and FF pairs usage	390 LUTs and 230 FFs	131
IOB usage	32	32
RAM/DSP blocks used (if any)	0	0

Simulation

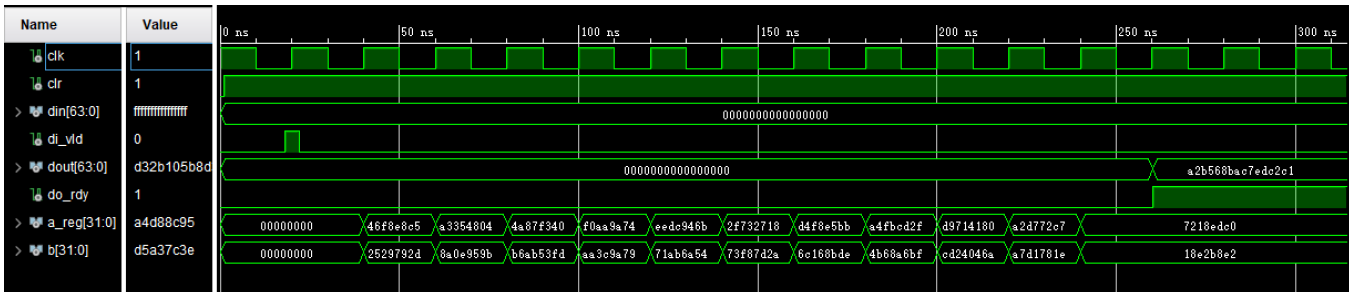
*Notice that all the simulations use the same clock peroid - 20 ns

Functional Simulation

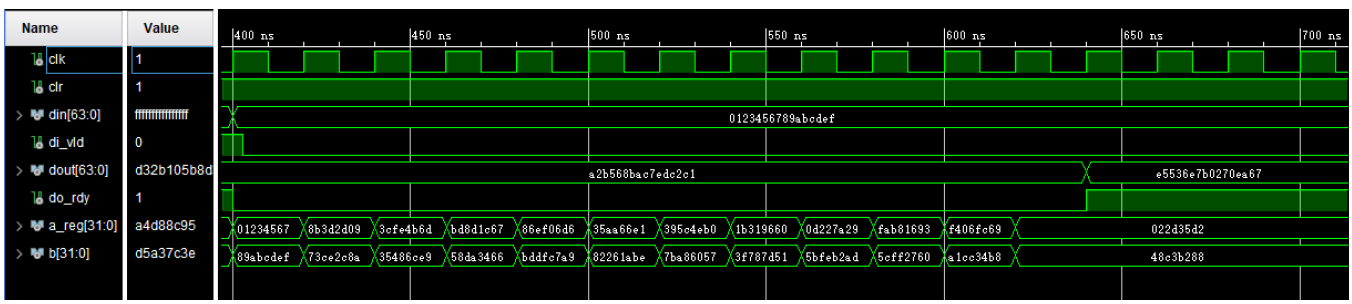
*In post-implementation simulation, the `a_reg`, `b_reg`, `a` and `b` are compiled to a lot of `wires`. So only some of these signals can be added to wave window.

RC5 Encryption

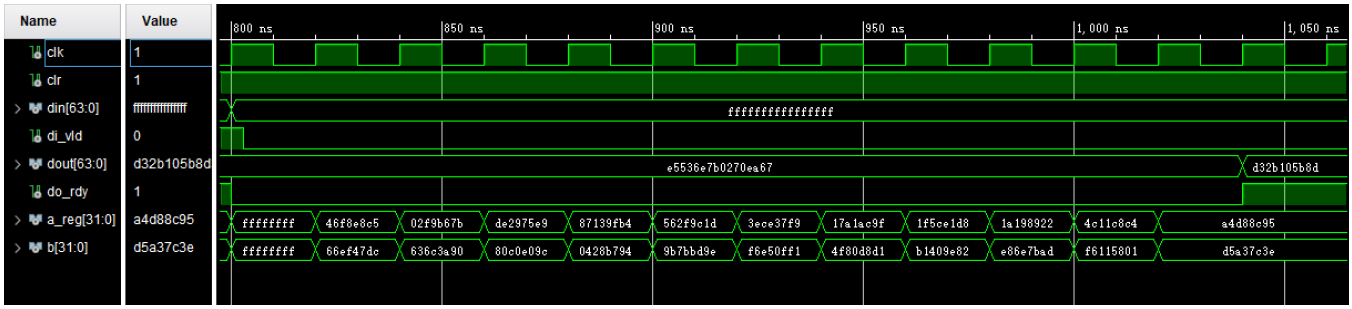
`din = x"0000000000000000"`



`din = x"0123456789abcdef"`

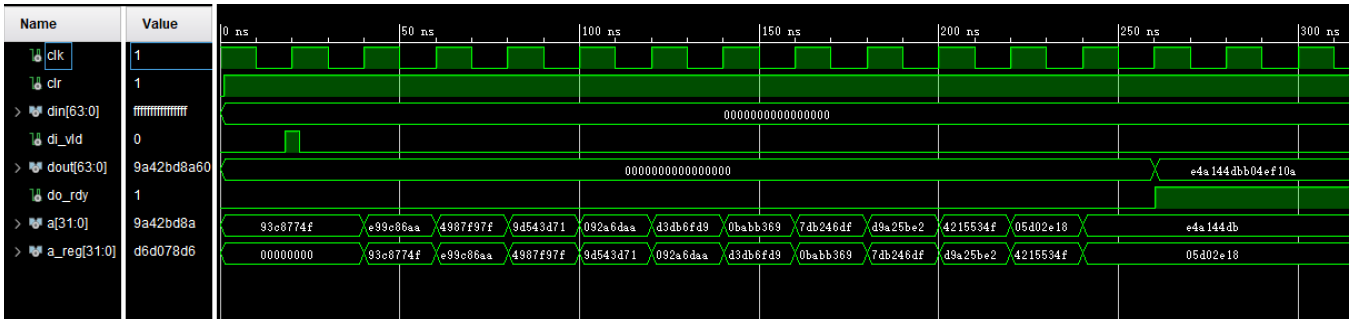


din = x"ffffffffffffffff"

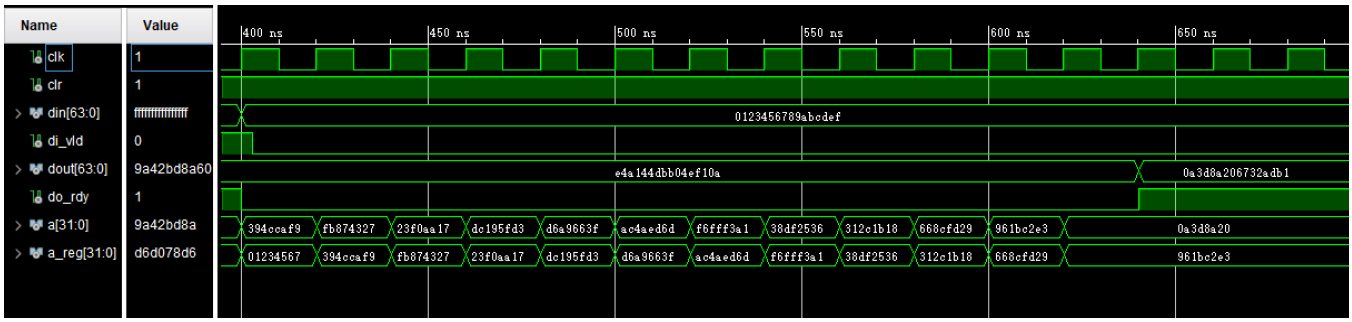


RC5 Decryption

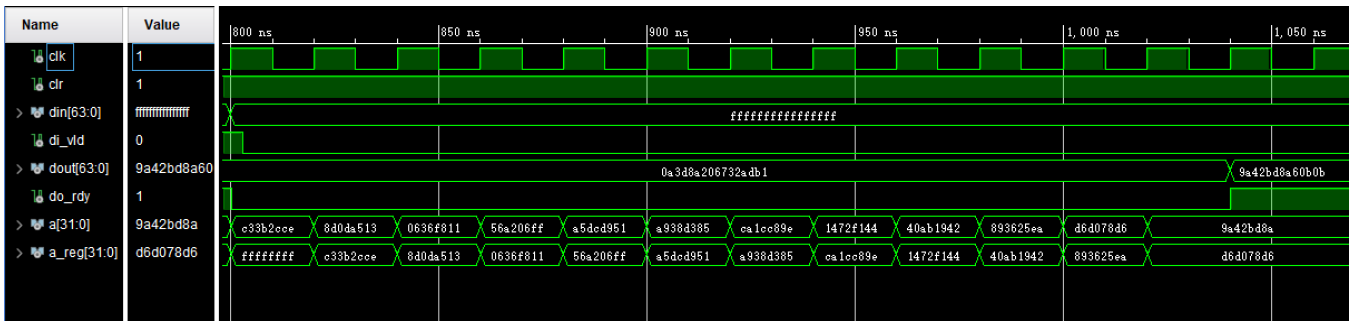
din = x"0000000000000000"



din = x"0123456789abcdef"



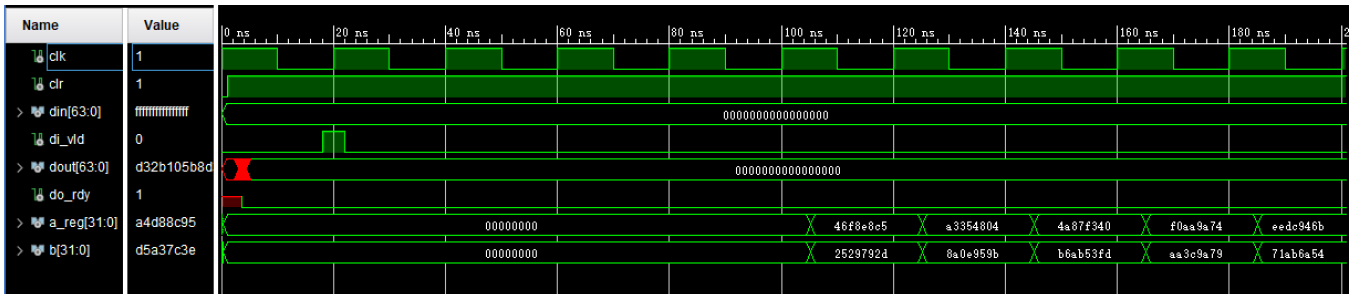
din = x"ffffffffffffffff"

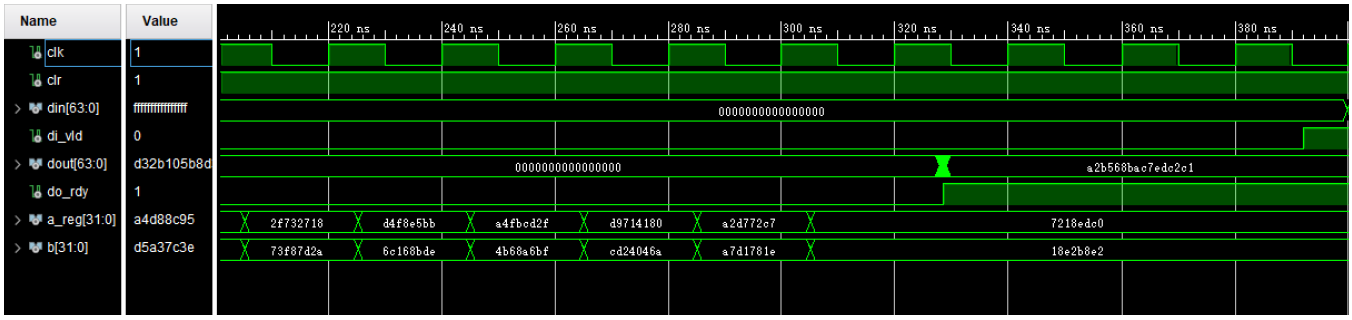


Timing Simulation

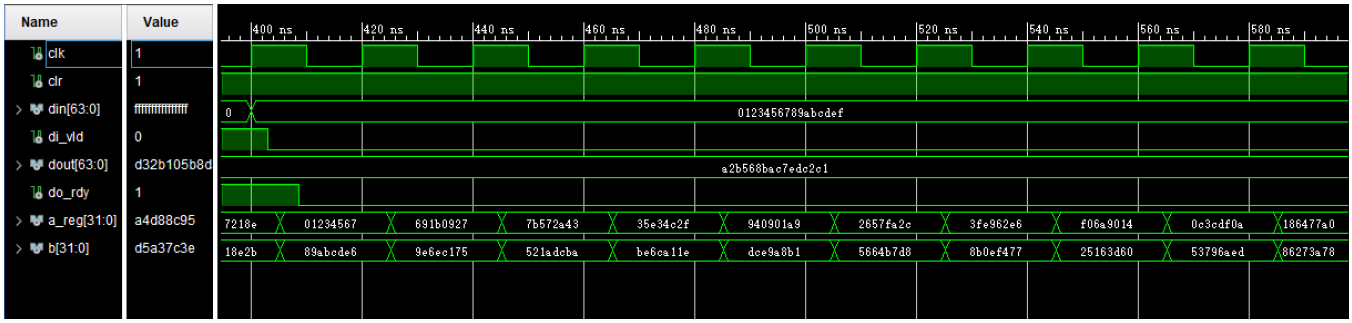
RC5 Encryption

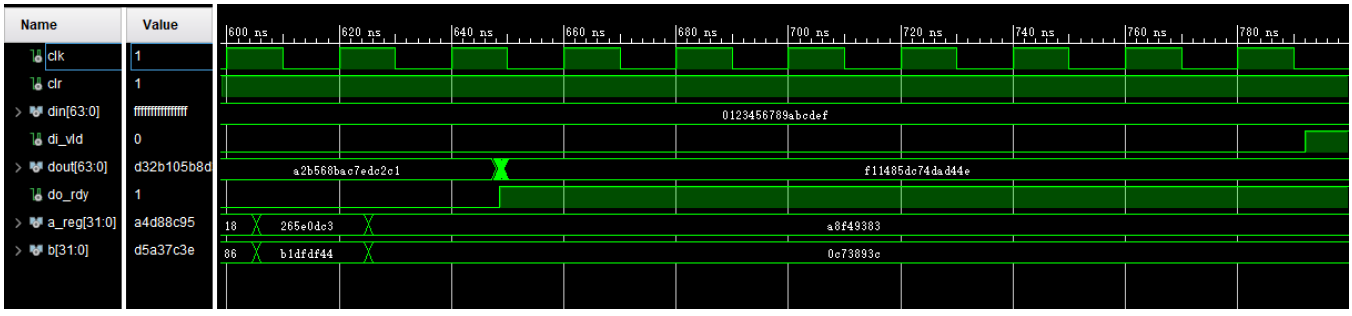
din = x"0000000000000000"



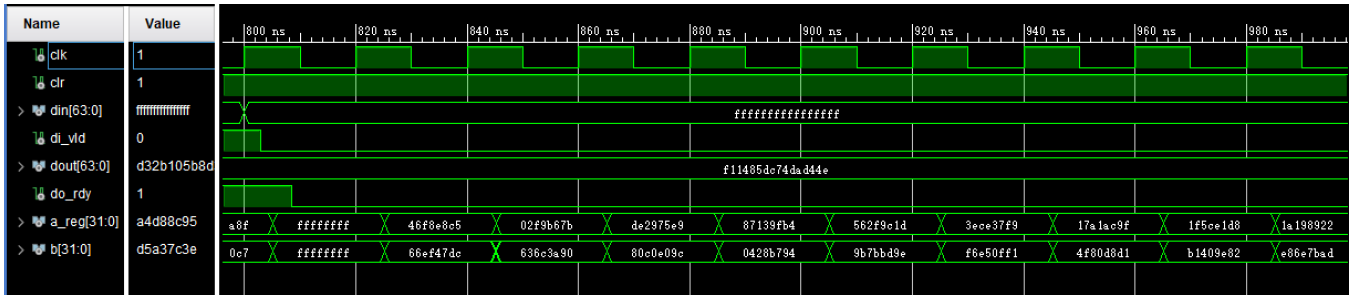


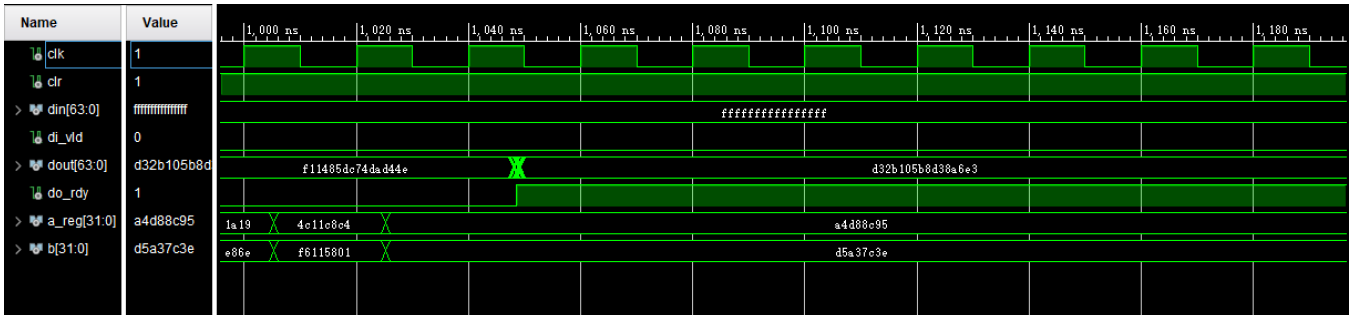
din = x"0123456789abcdef"





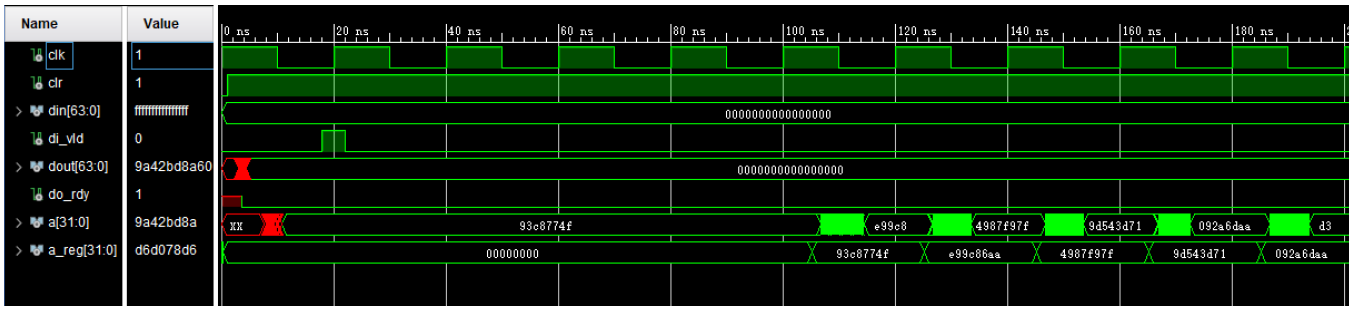
din = x"ffffffffffffffff"

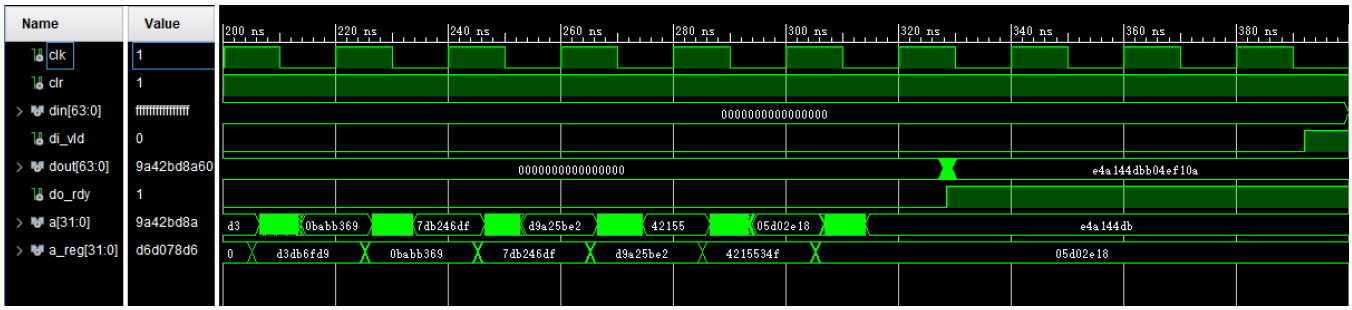




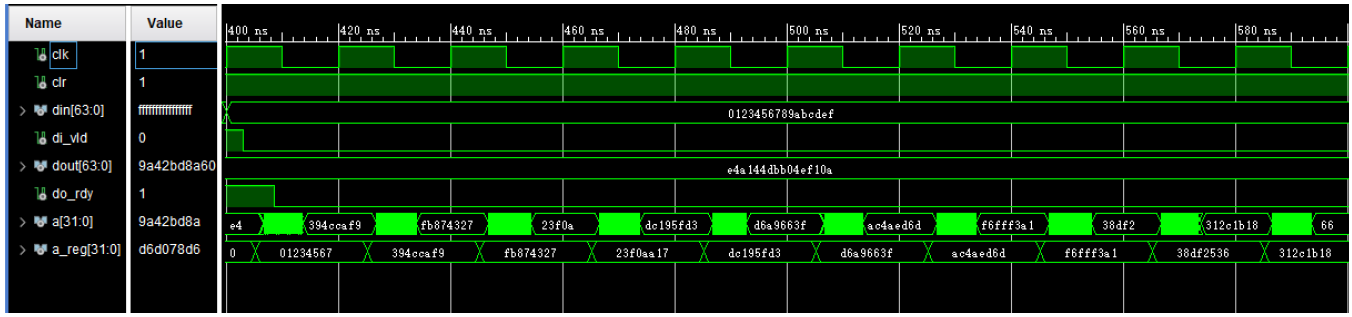
RC5 Decryption

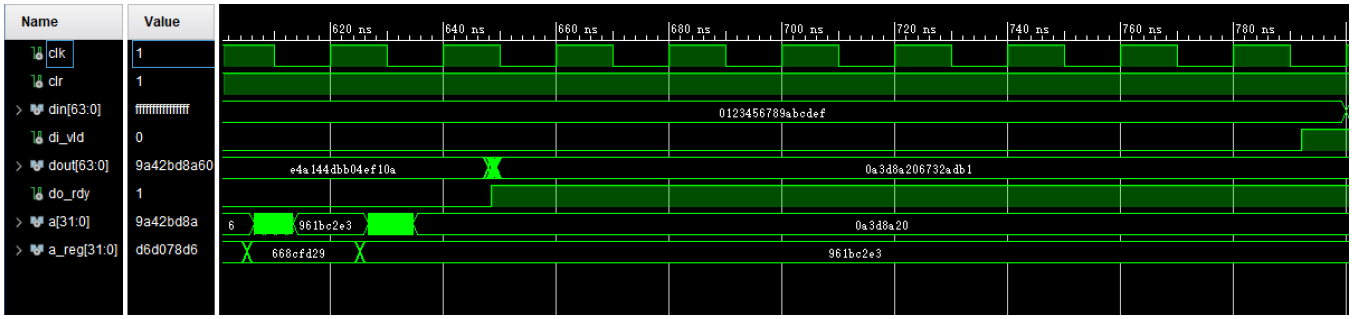
din = x"0000000000000000"



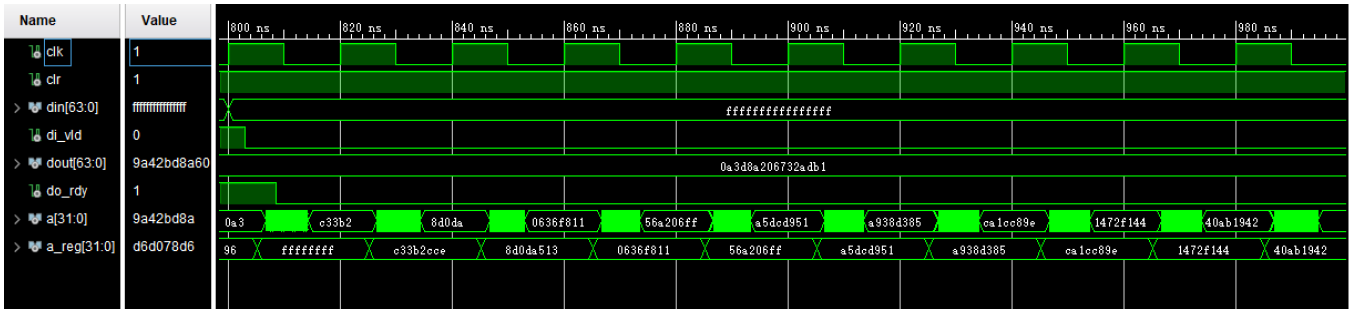


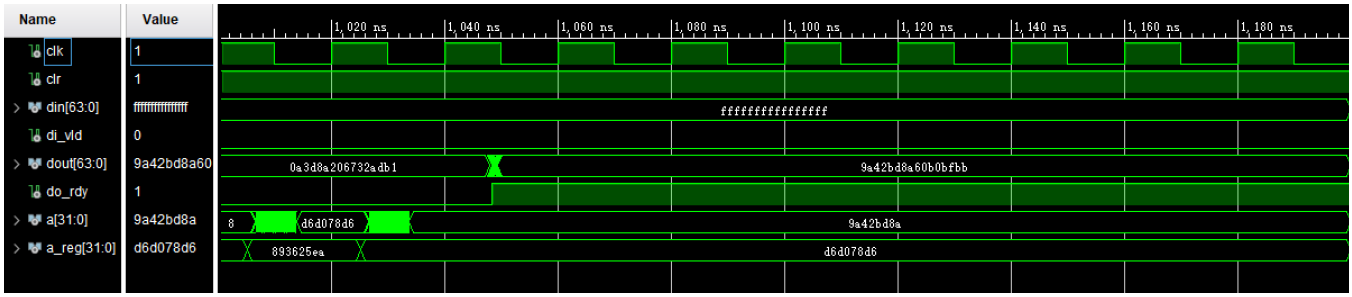
din = x"0123456789abcdef"





`din = x"ffffffffffffffff"`





Speed of the Design

*Remember that in Vivado, only WNS (Worst Negative Slack) is provided in the report, which represents the max delay path (required time - arrival time). In my design, I used a clock signal having a period of 20 ns. So the requirement was 20 ns, and the slack is `WNS` (note that it is positive), which means that we could have asked for a clock period `WNS` shorter and it would still be fine. Thus we can answer the minimum period and maximal frequency through this way. (I found this explanation on http://bilauer.co.il/blog/2017/01/vivado-minimal-period-timing/)

RC5 Encryption

Setup		**Hold**		**Pulse Width**	
Worst Negative Slack (WNS):	6.516 ns	Worst Hold Slack (WHS):	0.210 ns	Worst Pulse Width Slack (WPWS):	9.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns

Setup		Hold		Pulse Width	
Number of Falling Endpoints:	0	Number of Falling Endpoints:	0	Number of Falling Endpoints:	0
Total Number of Endpoints	412	Total Number of Endpoints	412	Total Number of Endpoints	219

Minimum period: 13.484 ns

Maximum clock frequency: 74.162 MHz

Latency: 12 clock cycles

RC5 Decryption

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	9.005 ns	Worst Hold Slack (WHS):	0.105 ns	Worst Pulse Width Slack (WPWS):	9.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Falling Endpoints:	0	Number of Falling Endpoints:	0	Number of Falling Endpoints:	0
Total Number of Endpoints	413	Total Number of Endpoints	413	Total Number of Endpoints	219

Minimum period: 10.995 ns

Maximum clock frequency: 90.950 MHz

Latency: 12 clock cycles

Port Map

Clock Signal

First, I generated a **clock signal** with period 20 ns, and mapped the clock signal `clk` (logic bit) in my design to it.

Because the hardware needs time to deal with the data in each clock cycle, if we use a too small period, for example, 10 ns, the time will be not enough to cover all the computations in each clock cycle. So we need set the clock cycle properly.

Buttons

In my design, 3 buttons (reset, up, down and center) are used.

The reset signal `clr` (logic bit) is mapped to **CPU reset button**, which serves as an asynchronous reset signal. When clicking on reset button, the signals including input (`din`), output (`dout`), registers (`a_reg` and `b_reg`) and some intermediate signals (`i_cnt`, `do_rdy`, .etc) will be initialized with a certain value (generally 0). Notice that the reset signal is low level effective. That is why it must be mapped to the CPU Reset button.

The signals `up_btn` (logic bit) and `down_btn` (logic bit) are mapped to **up button** and **down button**, which are used to modify the value of input vector. When clicking on up button, the displaying 8-bit hexadecimal number will increase by 1. Similarly, when clicking on down button, the displaying 8-bit hexadecimal number will decrease by 1. These two button only work when the input vector is being displayed. In addition, in order to avoid continued increasing/decreasing, I used a buffer signal for each button. By checking the value of button and the corresponding buffer at every clock rising edge, I can decide the rising edge of button signal is within which clock cycle. Thus, the function will be triggered only once at a time. Also, thanks to the high frequency clock signal, the delay cannot be detected by us human beings.

Center button is mapped to the signal `di_vld`, which is used to tell the system that input value is ready and computation can take place.

Switches

In my design, 10 switches (I/O switch, A/B switch, and 8 switches to decide the modifying bits) are used.

The signal `io_sw` (logic bit) is mapped to the first right-handed switch (**I/O switch**). This switch is used to switch the display (on 7 segments) between input vector and output vector.

The signal `ab_sw` (logic bit) is mapped to the second right-handed switch (**A/B switch**). This switch is used to switch the display (on 7 segments) between vector `A` (32 most significant bits) and vector `B` (32 least significant bits).

The signal `mod_hex` (8-bit logic vector) is mapped to 8 switches (**modifying switch**). Each bit of the vector corresponds to a certain segment. When modifying the hexadecimal value with the two buttons mentioned above, only the segments of which the corresponding switch is set to `1` will change.

LEDs

In my design, only one LED is used.

The signal `do_rdy` is mapped to the first left-handed LED. When output is ready, the LED will be on. Or it remains off.

7 Segment Display

In my design, all 7 segment display are used.

The current displaying value are determined by the two switch mentioned above (**I/O switch** and **A/B switch**).

In order to perform a proper function of displaying, I generate a slow clock, comparing with the clock `clk` (20 ns period). This clock signal for 7 segment display has a period of 20 * 2^16 ns and it is used to switch among all the 8 digits. At the rising edge of display clock signal (`disp_clk(15)`), the anode select (signal `seg_sel`) and corresponding value to be displayed (signal `seg_val`) will change. As a result, we can get a suitable refresh rate. With this rate, the 8 digits can be different and no overlapping occurs.

Brief Summary

By modifying the `.xdc` file, we can set the period of clock signal. In this way, each divide by two will use a flip-flop to implement. However, the number of flip-flop which can be used as frequency divider is limited. So the clock period can only be changed within a certain range. If we want a really slow clock, for example, a period of 1 sec. It can not be realized directly by flip-flop hardware. Thus, we need to declare a vector as counter to count the number of rising edges of clock signal.

The most significant difference between buttons and switches is that the switches have two stable states (`0` and `1`) while the buttons only have one (`0`). That's why I used buttons as triggers for each function and used switches to represent different states or modes.

There are 8 digits of 7 segment display but only 16 LEDs. As we all know, each digit of 7 segment display can show one bit of hexadecimal number, which takes 4 binary bits. 32 binary bits can be displayed at a time with 8 hexadecimal bits. So I chose to use 7 segment display to show the long vector (64 logic bits). Besides, the LEDs are used as indecators.

For more details, please go over my VHDL codes.

Demo Video

https://youtu.be/oqQJlu5xBXE