VIETNAM NATIONAL UNIVERSITY OF HO CHI MINH CITY
UNIVERSITY OF INFORMATION TECHNOLOGY
FACULTY OF COMPUTER ENGINEERING

FINAL PROJECT REPORT

DATA MEMORY SECURITY ON FPGA

Lecturer: Hồ Ngọc Diễm

NGUYỄN TUẤN DŨNG - 21520746
TRƯƠNG CÔNG THÀNH – 21522606
TRẦN ĐẶNG TOÀN – 21522688

HO CHI MINH CITY, 18/ 12/ 2023

Table of Contents

CHAPTER 1: GENERAL INTRODUCTION

1.1 Motivation:

Storing information using FIFO and LIFO mechanisms is to enhance storage performance. Automating the data management process according to these principles can help reduce the complexity of the information storage and retrieval process. In addition, the increasing need for information security poses a challenge. great knowledge in data management. Using FIFO and LIFO mechanisms can help increase information security by exploiting the way it works and creating additional encryption functions for data before storing it. For these reasons, we propose a system model for storing and encoding stored information using FIFO and LIFO memory by using the Verilog language.

1.2 System goals

We aim to design a data storage system according to the First-In-First-Out or Last-In-First-Out principle. Design security mechanisms to ensure the safety and integrity of data during storage and retrieval. The integration of these security layers helps prevent unauthorized access and ensures that important information is protected. In addition, this is to provide opportunities for project participants to develop skills in hardware design, Verilog programming, and a solid understanding of the operating principles of storage mechanisms.

1.3 Function's description:

Some main functions of the Data Memory Security designed in this project are as follows:

- SENDER:

+ Enter a Message (from SENDER want to send to RECEIVER)
+ The Message is converted to BINARY then put the data to module
+ We use XOR - CIPHER to encrypt data (XOR with K- the default value)
+ Then we reverse data by LIFO (Y) and store in FIFO, we have value Y in FIFO.

- RECEIVER:

+ RECEIVER put " KEY" to module
+ If right, RECEIVER has the right Message (which SENDER sent to RECEIVER)
+ If wrong, RECEIVER has the wrong Message.

1.4 Structural components of the system:

1.4.1 FIFO module:

A "Memory FIFO" refers to a First-In-First-Out memory buffer, which is a specific type of memory structure designed to temporarily store data in a way that preserves the order in which it was received. This structure operates on the principle that the first data item written into the buffer is the first one.

1.4.2 LIFO module:

"LIFO" stands for "Last-In-First-Out," and a "Memory LIFO" refers to a type of memory structure designed to store and retrieve data in a way that follows the Last-In-First-Out principle. In other words, the most recently added item is the first one to be removed or processed.

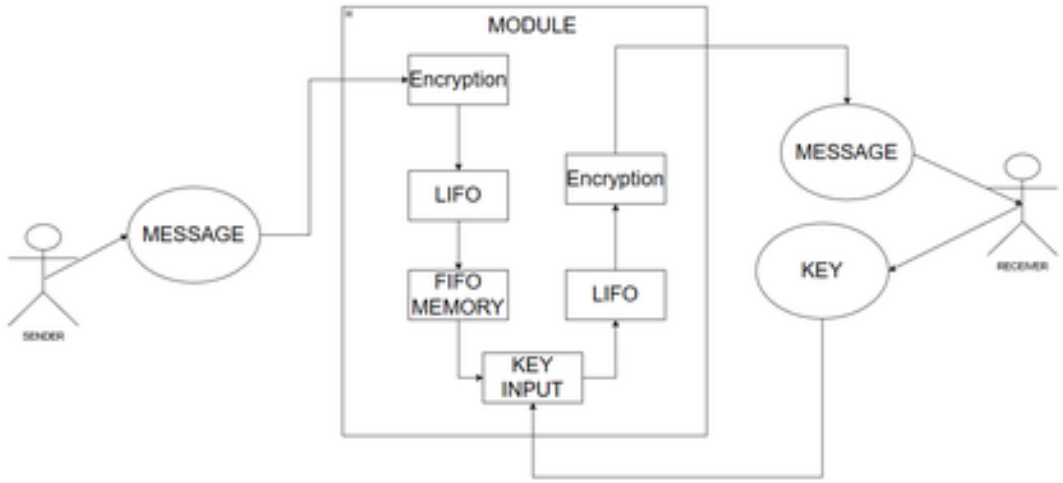
1.4.3 Encryption

Encryption is a module to encrypt data before putting it into LIFO memory to invert and store it in LIFO memory with the purpose of increasing the security of information against external threats. We use the XOR function to encrypt data in this project.

1.4.4 Decryption

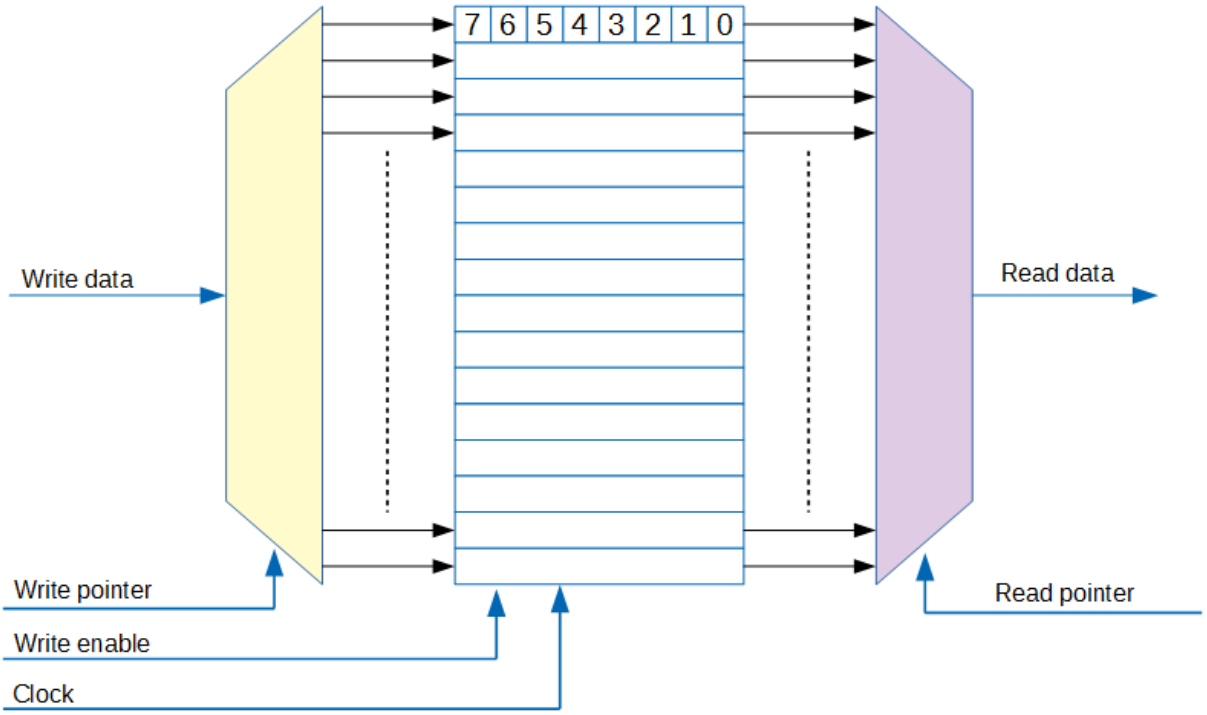
Decryption is a module that functions to decrypt data when the user uses the correct key previously provided, from which the desired data can be received without interference caused by the encryption module.We use the XOR function to decrypt data in this project.

1.5 Block diagram of system:



CHAPTER 2: THE STEPS TO IMPLEMENT THE SYSTEM

2.1 FIFO module:



2.1.1 Fifo_mem:

- **Verilog code:**

```
module fifo_mem(data_out,fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow,clk, rst_n, wr, rd, data_in
);
  input wr, rd, clk, rst_n;
  input[31:0] data_in;
  output[31:0] data_out;
  output fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow;
  wire[9:0] wptr,rptr;
  wire fifo_we,fifo_rd;
  write_pointer top1(wptr,fifo_we,wr,fifo_full,clk,rst_n);
  read_pointer top2(rptr,fifo_rd,rd,fifo_empty,clk,rst_n);
  memory_array top8(data_out, data_in, clk,fifo_we, wptr,rptr);
  status_signal top9(fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow, wr, rd, fifo_we, fifo_rd, wptr,
rptr,clk,rst_n);
 endmodule
```

Module fifo_mem:

+ This is the main module that defines the ports (inputs/outputs) and connects the sub-modules to build the FIFO.

+ The ports include:

- data_out: Output data from the FIFO.
- fifo_full: Signal indicating whether the FIFO is full or not.
- fifo_empty: Signal indicating whether the FIFO is empty or not.
- fifo_threshold: Signal indicating whether the FIFO has reached the threshold (near its limit) or not.
- fifo_overflow: Signal indicating whether the FIFO has overflowed or not.
- fifo_underflow: Signal indicating whether the FIFO has underflowed (lack of data) or not.
- clk: Clock signal.
- rst_n: Asynchronous reset signal (active-low).
- wr: Signal for writing data into the FIFO.
- rd: Signal for reading data from the FIFO.
- data_in: Input data into the FIFO.

+ This module uses the sub-modules write_pointer, read_pointer, memory_array, and status_signal to control and store the data of the FIFO.

2.1.2 Memory_array:

- **Verilog code:**

```
module memory_array(data_out, data_in, clk,fifo_we, wptr,rptr,fifo_rd);
 input[31:0] data_in;
 input clk,fifo_we,fifo_rd;
 input[9:0] wptr,rptr;
 output[31:0] data_out;
 reg[32767:0] data_out2[1023:0];
 reg [31:0] data_out;
 always @(posedge clk)
 begin
  if(fifo_we)
    data_out2[wptr[8:0]] <= data_in ;
 end
  always @(posedge clk)
 begin
  if(fifo_rd)
    data_out = data_out2[rptr[8:0]] ;
 end
endmodule
```

Module memory_array:

+ This is the module that stores the actual data in the FIFO

+ The ports include:

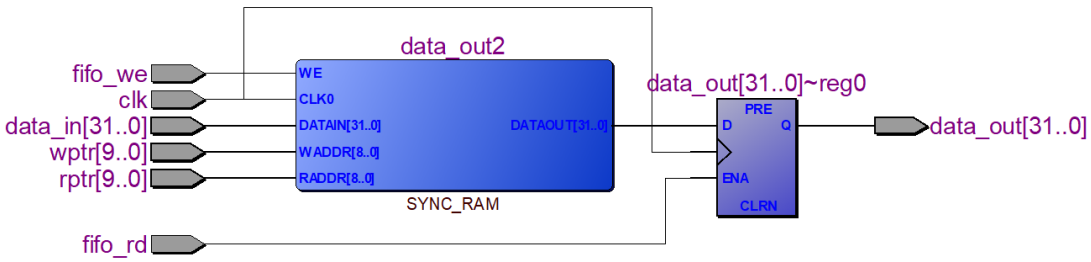
- data_out: Output data from the FIFO.
- data_in: Input data into the FIFO.
- clk: Clock signal.
- fifo_we: Signal for writing data into the FIFO memory.
- fifo_rd: Signal indicating whether the FIFO can be read.
- wptr: Write pointer for data.
- rptr: Read pointer for data.

+ This module uses an array called data_out2 with a size of 32768 * 1023 byte (approximately 3MB) to store the data.

+ When the fifo_we signal is activated (active), the data from data_in is written into the data_out2 array at the position specified by the wptr.

+ Check whether the fifo_rd(allow readings) is high on the positive edge of the watch; If it does, it reads the data from data_out2 at the index specified by the rptr, sending this data to the output.

- **RTL Viewer:**



- **Testbench:**

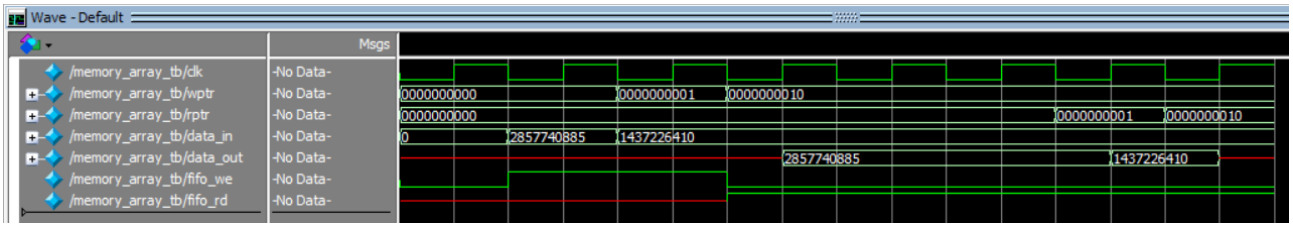
```
module memory_array_tb;

  // Parameters
  parameter DATA_WIDTH = 32;
  parameter ADDR_WIDTH = 10;
  parameter MEM_SIZE = 2**ADDR_WIDTH;

  // Signals
  reg [DATA_WIDTH-1:0] data_in;
  wire [DATA_WIDTH-1:0] data_out;
  reg clk;
  reg fifo_we;
  reg [ADDR_WIDTH-1:0] wptr;
  reg [ADDR_WIDTH-1:0] rptr;
  reg fifo_rd;

  // Instantiate the module
  memory_array dut (
    .data_in(data_in),
    .data_out(data_out),
    .clk(clk),
    .fifo_we(fifo_we),
    .wptr(wptr),
    .rptr(rptr),
    .fifo_rd(fifo_rd)
  );

  // Clock generation
  always begin
    #5 clk = ~clk;
  end

```

+ Parameter Definitions:

- DATA_WIDTH: The width (number of bits) of each word in the memory array.
- ADDR_WIDTH: The width (number of bits) of the memory array address.
- MEM_SIZE: The size of the memory array

+ Signal Declarations:

- data_in: Input signal for the data of the memory array.
- data_out: Output signal for the data of the memory array.
- clk: Clock signal.
- fifo_we: Write enable signal for the memory array.
- fifo_rd: Signal indicating whether the FIFO can be read
- wptr: Write pointer signal (index of the next write position in the memory array).
- rptr: Read pointer signal (index of the next read position in the memory array).



⇨ In this testbench, we load data into FIFO as 2857740885 and 1437226410 respectively using data_in signal, when storing data thfi wptr increases by 1 bit and when exporting data, rptr increases by 1 bit until equal to fifo. When there is a signal fifo_we = 1, data from data_in will be entered into memory and fifo_rd = 1, data will be output from memory.

2.1.3 Read pointer:

- **Verilog code:**

```
module read_pointer(rptr,fifo_rd,rd,fifo_empty,clk,rst_n);
 input rd,fifo_empty,clk,rst_n;
 output[9:0] rptr;
 output fifo_rd;
 reg[9:0] rptr;
 assign fifo_rd = (~fifo_empty)& rd;
 always @(posedge clk or negedge rst_n)
 begin
  if(~rst_n) rptr <= 10'b0000000000;
  else if(fifo_rd)
   rptr <= rptr + 10'b0000000001;
  else
   rptr <= rptr;
 end
endmodule
```

Module read_pointer:

+ This is the module that controls the read pointer.

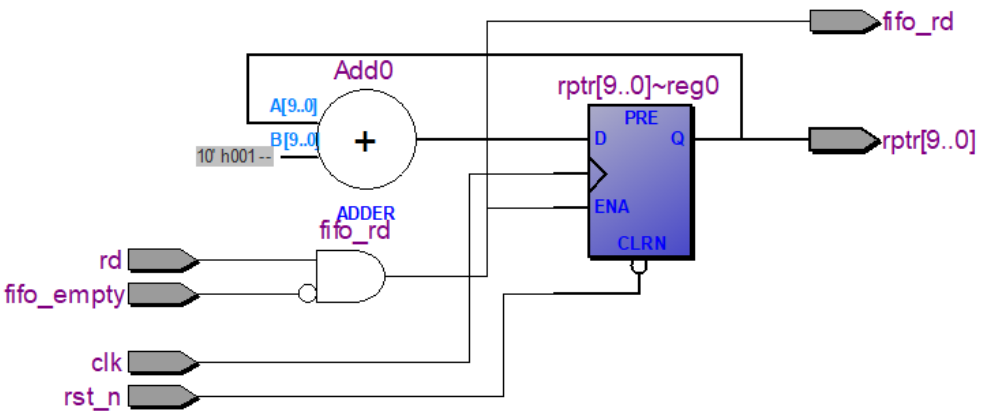
+ The ports include:

- rptr: Read pointer for data.
- fifo_rd: Signal indicating whether the FIFO can be read or not.
- rd: Signal for reading data from fifo_mem.
- fifo_empty: Signal indicating whether the FIFO is empty or not.
- clk: Clock signal.
- rst_n: Asynchronous reset signal (active-low).

+ This module controls the read pointer, rptr, based on the signals rd, fifo_empty, clk, and rst_n.

+ When both fifo_empty and rd signals are activated, rptr is incremented by 1 to indicate the next read position in the data array.

- **RTL Viewer:**



- **Testbench:**

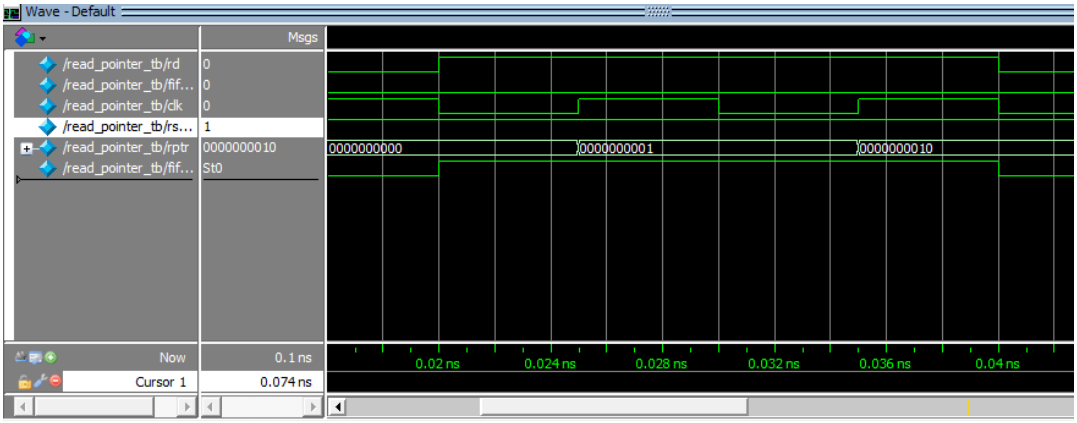
```
module read_pointer_tb;
  reg rd, fifo_empty, clk, rst_n;
  wire [9:0] rptr;
  wire fifo_rd;

  // Instantiate the module
  read_pointer ul (
    .rptr(rptr),
    .fifo_rd(fifo_rd),
    .rd(rd),
    .fifo_empty(fifo_empty),
    .clk(clk),
    .rst_n(rst_n)
  );

  // Clock generator
  always begin
    #5 clk = ~clk;
  end

  // Test sequence
  initial begin
    rd = 0;
    fifo_empty = 0;
    clk = 0;
    rst_n = 0;
    #10 rst_n = 1;  // Release reset
    #10 rd = 1;     // Start reading
    #20 rd = 0;     // Stop reading
    #10 fifo_empty = 1;  // FIFO becomes empty
    #20 fifo_empty = 0;  // FIFO is not empty
    #10 rd = 1;     // Start reading again
    #20 $finish;    // End of test
  end
endmodule

```

+ Declaration and connection of variables and signals:

- reg rd, fifo_empty, clk, rst_n: Declare the variables rd, fifo_empty, clk, rst_n as reg (register) type variables, which hold data in the testbench.
- wire [9:0] rptr: Declare the variable rptr as a wire signal with a size of 10 bits.
- wire fifo_rd: Declare the variable fifo_rd as a wire signal.

+ Test sequence:

- initial begin: Begin the initial block.
- rd = 0; fifo_empty = 0; clk = 0; rst_n = 0: Set initial values for the variables and signals in the testbench.
- rst_n = 1: Set rst_n to 1 to release the reset.
- rd = 1: Set rd to 1 to start reading.
- rd = 0: Set rd to 0 to stop reading.
- fifo_empty = 1: Set fifo_empty to 1 to indicate that the FIFO becomes empty.
- fifo_empty = 0: Set fifo_empty to 0 to indicate that the FIFO is not empty.
- rd = 1: Set rd to 1 to start reading again.



⇨ Regarding the output value, rptr will increment by 1 each time rd is set to 1 and fifo_empty is 0. This happens twice in the testbench, so rptr will increase from 0000 (0 in decimal) to 00001 (1 in decimal) after the first read, and then to 00010 (2 in decimal) after the second read.

2.1.4 Write pointer:

- **Verilog code:**

```
45  module write_pointer(wptr,fifo_we,wr,fifo_full,clk,rst_n);
46   input wr,fifo_full,clk,rst_n;
47   output[9:0] wptr;
48   output fifo_we;
49   reg[9:0] wptr;
50   assign fifo_we = (~fifo_full)&wr;
51   always @(posedge clk or negedge rst_n)
52   begin
53    if(~rst_n) wptr <= 10'b0000000000;
54    else if(fifo_we)
55     wptr <= wptr + 10'b0000000001;
56    else
57     wptr <= wptr;
58   end
59  endmodule
60
```

Module write_pointer:

+ This is the module that controls the write pointer.

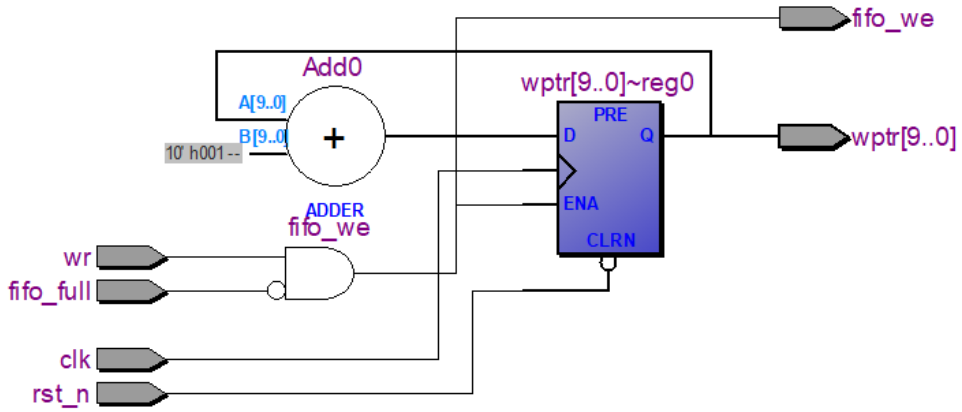
+ The ports include:

- wptr: Write pointer for data.
- fifo_we: Signal indicating whether the FIFO can be written to or not.
- wr: Signal for writing data into fifo_mem.
- fifo_full: Signal indicating whether the FIFO is full or not.
- clk: Clock signal.
- rst_n: Asynchronous reset signal (active-low).

+ This module controls the write pointer, wptr, based on the signals wr, fifo_full, clk, and rst_n.

+ When both fifo_full and wr signals are activated, wptr is incremented by 1 to indicate the next write position in the data array.

- **RTL Viewer:**



- **Testbench:**

```
1   module write_pointer_tb;
2     reg [9:0] data_in;
3     reg wr, fifo_full, clk, rst_n;
4     wire [9:0] wptr;
5     wire fifo_we;
6
7     // Instantiate the module
8     write_pointer u1 (
9       .wptr(wptr),
10      .fifo_we(fifo_we),
11      .wr(wr),
12      .fifo_full(fifo_full),
13      .clk(clk),
14      .rst_n(rst_n)
15    );
16
17    // Clock generator
18    always begin
19      #5 clk = ~clk;
20    end
21
22    // Test sequence
23    initial begin
24      data_in = 0;
25      wr = 0;
26      fifo_full = 0;
27      clk = 0;
28      rst_n = 0;
29      #10 rst_n = 1;  // Release reset
30      #10 wr = 1; data_in = 10'b0000000001;  // Start writing value 1
31      #20 wr = 1; data_in = 10'b0000000010;  // Write value 2
32      #10 wr = 0;     // Stop writing
33      #10 fifo_full = 1;  // FIFO becomes full
34      #20 fifo_full = 0;  // FIFO is not full
35      #10 wr = 1; data_in = 10'b0000000011;  // Start writing value 3
36      #20 $finish;    // End of test
37    end
38  endmodule
39
```

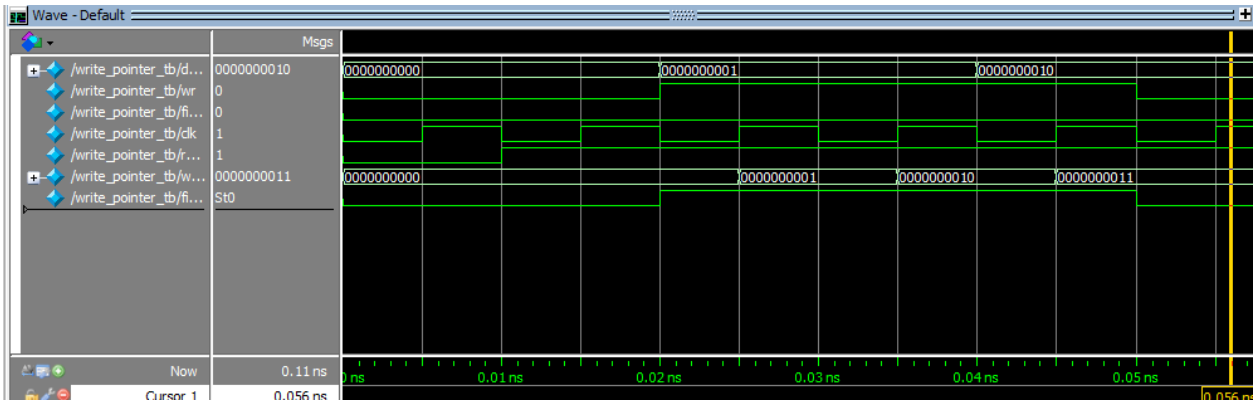
+ Declaration and connection of variables and signals:

- reg [9:0] data_in: Declare the variable data_in as a register of size 10 bits.
- reg wr, fifo_full, clk, rst_n: Declare the variables wr, fifo_full, clk, rst_n as registers.
- wire [9:0] wptr: Declare the variable wptr as a wire signal with a size of 10 bits.

- wire fifo_we: Declare the variable fifo_we as a wire signal.

+ Test sequence:

- initial begin: Begin the initial block.
- data_in = 0; wr = 0; fifo_full = 0; clk = 0; rst_n = 0: Set initial values for the variables and signals in the testbench.
- rst_n = 1: Set rst_n to 1 to release the reset.
- wr = 1; data_in = 10'b0000000001: Set wr to 1 to start writing, and assign the value 1 to data_in.
- wr = 1; data_in = 10'b0000000010: Continue writing by setting wr to 1 and assign the value 2 to data_in.
- wr = 0: Stop writing by setting wr.
- fifo_full = 1: Set fifo_full to 1 to indicate that the FIFO becomes full.
- fifo_full = 0: Set fifo_full to 0 to indicate that the FIFO is not full.
- wr = 1; data_in = 10'b0000000011: Start writing again by setting wr to 1 and assign the value 3 to data_in.



2.1.5 Status signal:

- **Verilog code:**

```
61  module status_signal(fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow, wr, rd, fifo_we, fifo_rd, wptr,
    rptr,clk,rst_n);
62   input wr, rd, fifo_we, fifo_rd,clk,rst_n;
63   input[9:0] wptr, rptr;
64   output fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow;
65   wire fbit_comp, overflow_set, underflow_set;
66   wire pointer_equal;
67   wire[9:0] pointer_result;
68   reg fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow;
69   assign fbit_comp = wptr[9] ^ rptr[9];
70   assign pointer_equal = (wptr[8:0] - rptr[8:0]) ? 0:1;
71   assign pointer_result = wptr[9:0] - rptr[9:0];
72   assign overflow_set = fifo_full & wr;
73   assign underflow_set = fifo_empty&rd;
74   always @(*)
75   begin
76    fifo_full =fbit_comp & pointer_equal;
77    fifo_empty = (~fbit_comp) & pointer_equal;
78    fifo_threshold = (pointer_result[9]||pointer_result[8]) ? 1:0;
79   end
80   always @(posedge clk or negedge rst_n)
81   begin
82   if(~rst_n) fifo_overflow <=0;
83   else if((overflow_set==1)&&(fifo_rd==0))
84    fifo_overflow <=1;
85    else if(fifo_rd)
86     fifo_overflow <=0;
```

```
87        else
88         fifo_overflow <= fifo_overflow;
89      end
90      always @(posedge clk or negedge rst_n)
91      begin
92      if(~rst_n) fifo_underflow <=0;
93      else if((underflow_set==1)&&(fifo_we==0))
94       fifo_underflow <=1;
95       else if(fifo_we)
96        fifo_underflow <=0;
97        else
98         fifo_underflow <= fifo_underflow;
99      end
100   endmodule
101
```

Module status_signal:

+ This is the module that controls the status signals of the FIFO.

+ The ports include:

- fifo_full: Signal indicating whether the FIFO is full or not.
- fifo_empty: Signal indicating whether the FIFO is empty or not.
- fifo_threshold: Signal indicating whether the FIFO has reached the threshold or not.
- fifo_overflow: Signal indicating whether the FIFO has overflowed or not.
- fifo_underflow: Signal indicating whether the FIFO has underflowed or not.
- wr: Signal for writing data into fifo_mem.
- rd: Signal for reading data from fifo_mem.
- fifo_we: Signal indicating whether the FIFO can be written to or not.
- fifo_rd: Signal indicating whether the FIFO can be read or not.
- wptr: Write pointer for data.
- rptr: Read pointer for data.
- clk: Clock signal.
- rst_n: Asynchronous reset signal (active-low).

+ This module controls the status signals of the FIFO based on the signals and pointers from other sub-modules.

+ fifo_full is set to 1 when the FIFO is full, and 0 otherwise.

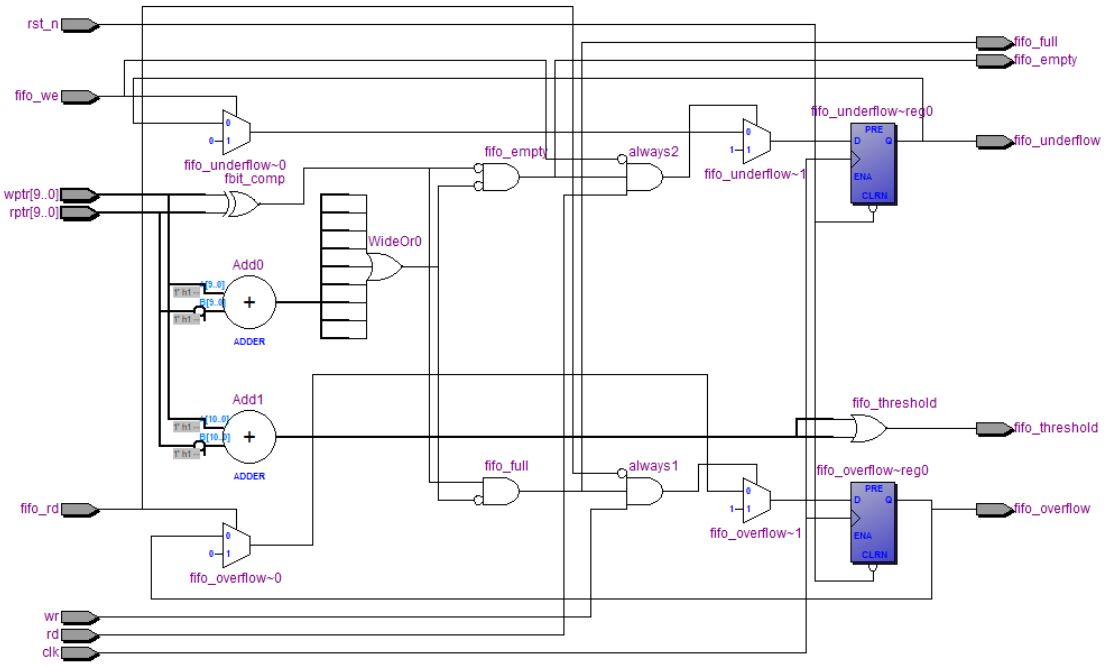
+ fifo_empty is set to 1 when the FIFO is empty, and 0 otherwise.

+ fifo_threshold is set to 1 when the FIFO has reached the threshold (number of elements near the limit), and 0 otherwise.

+ fifo_overflow is set to 1 when the FIFO has overflowed (writing data when full), and 0 otherwise.

+ fifo_underflow is set to 1 when the FIFO has underflowed (reading data when empty), and 0 otherwise.

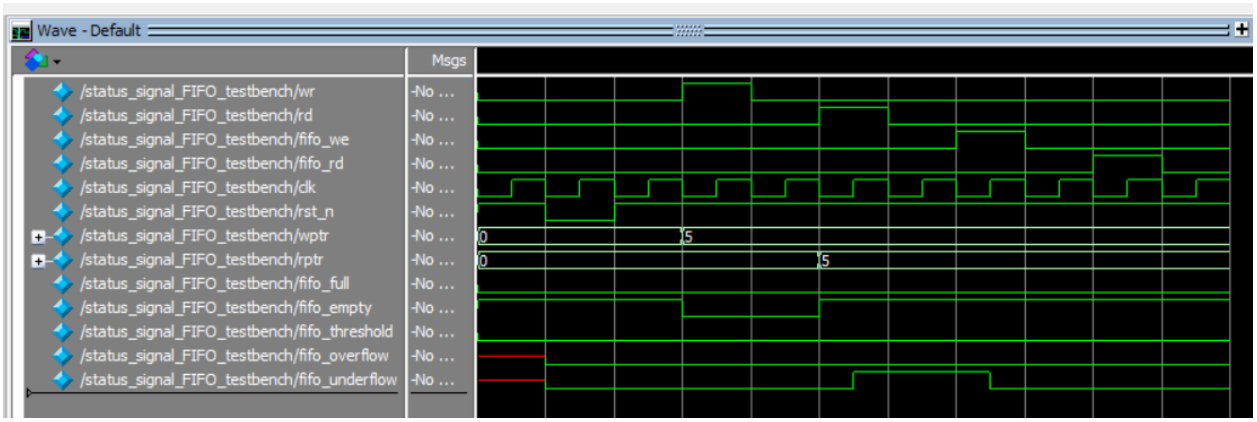
- **RTL Viewer:**



- **Testbench:**

```
`timescale 1ns/100ps

module status_signal_FIFO_testbench;
  // Signals
  reg wr, rd, fifo_we, fifo_rd, clk, rst_n;
  reg[9:0] wptr, rptr;
  wire fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow;

  // Instantiate the module under test
  status_signal_FIFO sFF(
    .fifo_full(fifo_full),
    .fifo_empty(fifo_empty),
    .fifo_threshold(fifo_threshold),
    .fifo_overflow(fifo_overflow),
    .fifo_underflow(fifo_underflow),
    .wr(wr),
    .rd(rd),
    .fifo_we(fifo_we),
    .fifo_rd(fifo_rd),
    .wptr(wptr),
    .rptr(rptr),
    .clk(clk),
    .rst_n(rst_n)
  );

  // Clock generation
  always #5 clk = ~clk;

  // Initial block for stimulus
  initial begin
    // Initialize inputs
    wr = 0;
    rd = 0;
    fifo_we = 0;
    fifo_rd = 0;
    wptr = 0;
    rptr = 0;
    clk = 0;
    rst_n = 1;

    // Apply reset
    #10 rst_n = 0;

    // Release from reset
    #10 rst_n = 1;

    // Test scenario
    #10 wr = 1; wptr = 5;
    #10 wr = 0;
    #10 rd = 1; rptr = 5;
    #10 rd = 0;
    #10 fifo_we = 1;
    #10 fifo_we = 0;
    #10 fifo_rd = 1;
    #10 fifo_rd = 0;

    // Add more test scenarios as needed

    #10 $stop; // End simulation
  end

endmodule
```

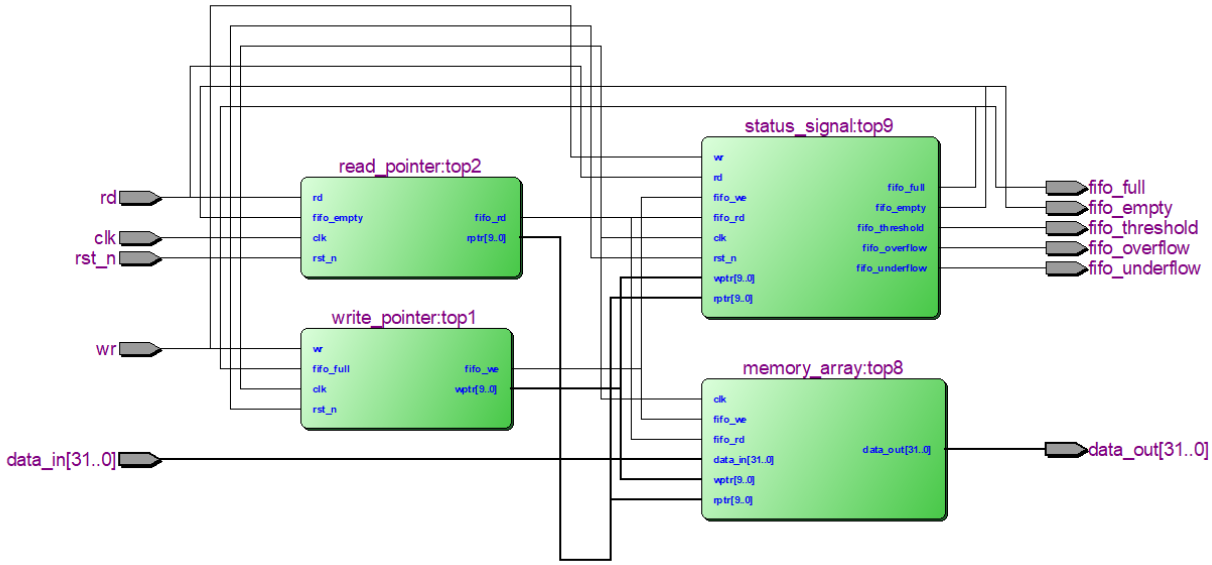
- wr: Write signal, data is written when wr = 1.
- rd: Read signal, data is read when rd = 1.
- fifo_we: Write signal (fifo_we = (~fifo_full) & wr) is active when the FIFO is not full (~fifo_full) and write signal is asserted (wr = 1).
- fifo_rd: Read signal (fifo_rd = (~fifo_empty) & rd) is active when the FIFO is not empty (~fifo_empty) and read signal is asserted (rd = 1).
- clk: Clock signal.
- reset_n: Reset signal, resets fifo_overflow and fifo_underflow to 0 when reset_n = 0.
- fifo_full: Indicates FIFO full when fifo_full = 1.
- fifo_empty: Indicates FIFO empty when fifo_empty = 1.
- fifo_overflow: Indicates FIFO overflow (fifo_overflow = 1) when the FIFO is full but data is still being written.
- fifo_underflow: Indicates FIFO underflow (fifo_underflow = 1) when the FIFO is empty but data is still being read.
- fifo_threshold: Reaches value 1 when the number of data in the FIFO is less than a specific threshold; otherwise, it is low (fifo_threshold = (pointer_result[24] || pointer_result[23]) ? 1 : 0).



+ Explanation of Testbench:

- In the status_signal_FIFO module, there are 4 input signals: wr, rd, fifo_we, and fifo_rd. Specifically, fifo_we and fifo_rd are outputs of another module. Examining the waveform, when reset_n is asserted (reset_n = 0), it activates fifo_overflow and fifo_underflow, setting them to 0.
- Fifo_Empty is 0 when there is a gap between the wptr and rptr addresses, and fifo_empty is 1 when wptr equals rptr (4th bit is the same).
- Fifo_Full is 1 when the 4th bit is different and the lowest 4 bits are the same. This means wptr equals rptr, but wptr has completed one full rotation.
- Fifo_Overflow is 0 because fifo_full is 0, while fifo_underflow is 1 due to fifo_empty = 1, fifo_rd = 1, and fifo_we = 0.
- Fifo_Threshold is 0 because the data in the FIFO is not lower than a specific threshold.

⇨ **When we have submodules like fifo_mem, memory_array, read_pointer, write_pointer and status_signal. We proceed to connect them to get a complete FIFO as follows:**



- **Verilog code:**

```
module fifo_mem(data_out,fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow,clk, rst_n, wr, rd, data_in
);
  input wr, rd, clk, rst_n;
  input[31:0] data_in;
  output[31:0] data_out;
  output fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow;
  wire[9:0] wptr,rptr;
  wire fifo_we,fifo_rd;
  write_pointer top1(wptr,fifo_we,wr,fifo_full,clk,rst_n);
  read_pointer top2(rptr,fifo_rd,rd,fifo_empty,clk,rst_n);
  memory_array top8(data_out, data_in, clk,fifo_we, wptr,rptr);
  status_signal top9(fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow, wr, rd, fifo_we, fifo_rd, wptr,
rptr,clk,rst_n);
 endmodule

 module memory_array(data_out, data_in, clk,fifo_we, wptr,rptr);
  input[31:0] data_in;
  input clk,fifo_we;
  input[9:0] wptr,rptr;
  output[31:0] data_out;
  reg[32767:0] data_out2[1023:0];
  wire[31:0] data_out;
  always @(posedge clk)
  begin
   if(fifo_we)
      data_out2[wptr[8:0]] <=data_in ;
  end
 assign data_out = data_out2[rptr[8:0]];
endmodule

module read_pointer(rptr,fifo_rd,rd,fifo_empty,clk,rst_n);
 input rd,fifo_empty,clk,rst_n;
 output[9:0] rptr;
 output fifo_rd;
 reg[9:0] rptr;
 assign fifo_rd = (~fifo_empty)& rd;
 always @(posedge clk or negedge rst_n)
 begin
  if(~rst_n) rptr <= 10'b0000000000;
  else if(fifo_rd)
   rptr <= rptr + 10'b0000000001;
  else
   rptr <= rptr;
 end
endmodule

module write_pointer(wptr,fifo_we,wr,fifo_full,clk,rst_n);
 input wr,fifo_full,clk,rst_n;
 output[9:0] wptr;
 output fifo_we;
 reg[9:0] wptr;
 assign fifo_we = (~fifo_full)&wr;
 always @(posedge clk or negedge rst_n)
 begin
```

```
53      if(~rst_n) wptr <= 10'b0000000000;
54      else if(fifo_we)
55       wptr <= wptr + 10'b0000000001;
56      else
57       wptr <= wptr;
58     end
59    endmodule
60
61    module status_signal(fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow, wr, rd, fifo_we, fifo_rd, wptr
      ,rptr,clk,rst_n);
62     input wr, rd, fifo_we, fifo_rd,clk,rst_n;
63     input[9:0] wptr, rptr;
64     output fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow;
65     wire fbit_comp, overflow_set, underflow_set;
66     wire pointer_equal;
67     wire[9:0] pointer_result;
68     reg fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow;
69     assign fbit_comp = wptr[9] ^ rptr[9];
70     assign pointer_equal = (wptr[8:0] - rptr[8:0]) ? 0:1;
71     assign pointer_result = wptr[9:0] - rptr[9:0];
72     assign overflow_set = fifo_full & wr;
73     assign underflow_set = fifo_empty&rd;
74     always @(*)
75     begin
76      fifo_full =fbit_comp & pointer_equal;
77      fifo_empty = (~fbit_comp) & pointer_equal;
78      fifo threshold = (pointer result[9]||pointer result[8]) ? 1:0;
79     end
80     always @(posedge clk or negedge rst_n)
81     begin
82     if(~rst_n) fifo_overflow <=0;
83     else if((overflow_set==1)&&(fifo_rd==0))
84      fifo_overflow <=1;
85      else if(fifo_rd)
86       fifo_overflow <=0;
87       else
88        fifo_overflow <= fifo_overflow;
89     end
90     always @(posedge clk or negedge rst_n)
91     begin
92     if(~rst_n) fifo_underflow <=0;
93     else if((underflow_set==1)&&(fifo_we==0))
94      fifo_underflow <=1;
95      else if(fifo_we)
96       fifo_underflow <=0;
97       else
98        fifo_underflow <= fifo_underflow;
99     end
100   endmodule
101
102
```

⇨ This Verilog code implements a FIFO using submodules to manage read and write pointers, store data, and control the state of the FIFO.

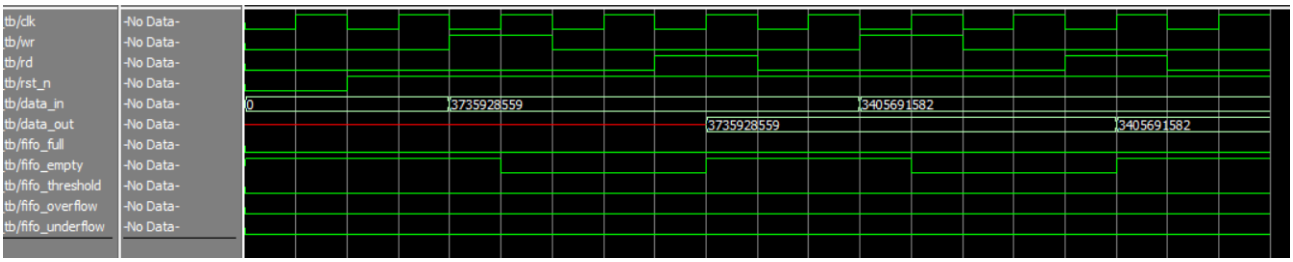
- **Testbench:**

```
1   module fifo_mem_tb;
2       reg wr, rd, clk, rst_n;
3       reg [31:0] data_in;
4       wire [31:0] data_out;
5       wire fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow;
6
7       // Khởi tạo DUT
8       fifo_mem DUT (
9           .data_out(data_out),
10          .fifo_full(fifo_full),
11          .fifo_empty(fifo_empty),
12          .fifo_threshold(fifo_threshold),
13          .fifo_overflow(fifo_overflow),
14          .fifo_underflow(fifo_underflow),
15          .clk(clk),
16          .rst_n(rst_n),
17          .wr(wr),
18          .rd(rd),
19          .data_in(data_in)
20      );
21
22
23      always begin
24          #5 clk = ~clk;
25      end
26
27
28      initial begin
29          // Khởi tạo tín hiệu
30          clk = 0; rst_n = 0; wr = 0; rd = 0; data_in = 0;
31
```

```
32          #10 rst_n = 1; // Thả reset
33          #10 wr = 1; data_in = 32'hDEADBEEF; // Ghi dữ liệu vào FIFO
34          #10 wr = 0;
35          #10 rd = 1; // Đọc dữ liệu từ FIFO
36          #10 rd = 0;
37
38          #10 wr = 1; data_in = 32'hCAFEBABE; // Ghi dữ liệu khác vào FIFO
39          #10 wr = 0;
40          #10 rd = 1; // Đọc dữ liệu từ FIFO
41          #10 rd = 0;
42
43          #10 $finish; // Kết thúc mô phỏng
44      end
45  endmodule
46
```

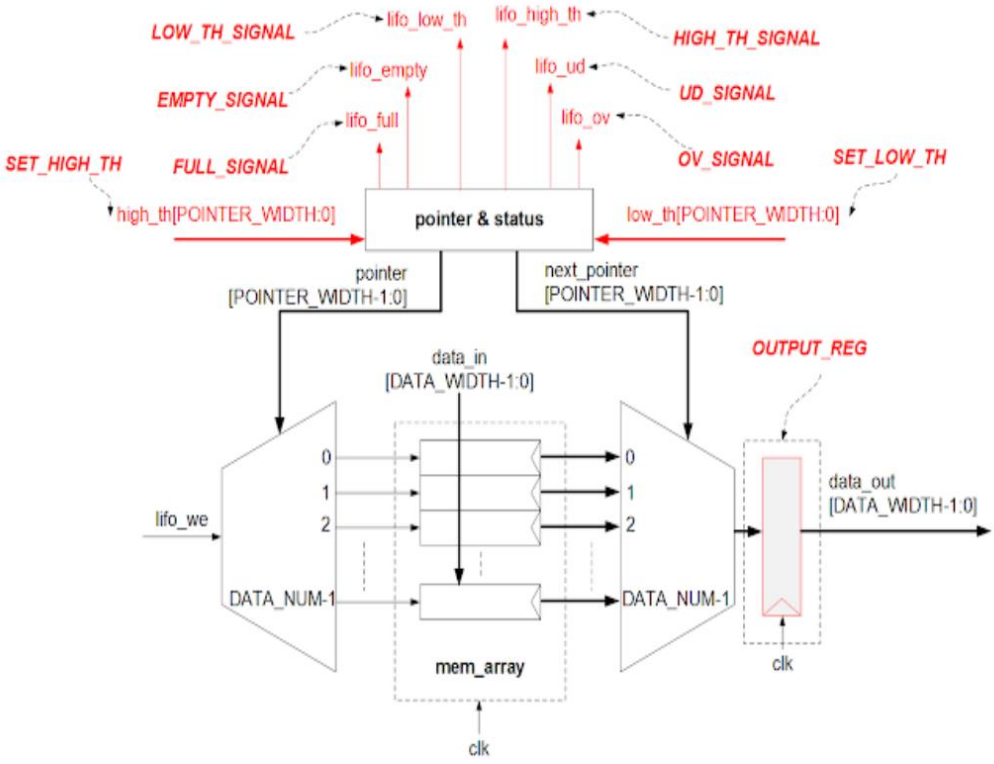
+ Declaration and initialization of signals:

- wr, rd, clk, rst_n are reg-type (registers) input signals representing write enable, read enable, clock, and reset signals respectively.
- data_in is a 32-bit-width reg-type (register) input signal representing the input data to the FIFO.
- data_out is a 32-bit-width wire-type (wire) output signal representing the output data from the FIFO.
- fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow are wire-type (wire) output signals representing the state of the FIFO, such as full, empty, threshold, overflow, and underflow.



⇨ We load data into FIFO as 3735928559 and 3405691582 respectively using data_in signal, when storing data thfi wptr increases by 1 bit and when exporting data, rptr increases by 1 bit until equal to fifo. When there is a signal fifo_we = 1, data from data_in will be entered into memory and fifo_rd = 1, data will be output from memory. Signals fifo_full and fifo_empty indicate the empty or full status of MEMORY. The fifo_threshold = 0 signal indicates that MEMORY has not passed a specific threshold.

2.2 LIFO module



2.2.1 Lifo_memory:

- **Verilog code:**

```
module lifo_memory(clk, rst_n, wr, rd,lifo_empty,lifo_full,data_in,data_out,pointer1);
//inputs
input clk;
input rst_n;
input wr;
input rd;
input [31:0] data_in;
//outputs
output  reg [31:0] data_out;
output  lifo_empty;
output  lifo_full;
output [9:0]pointer1;
//pointer
reg [9:0] pointer;
wire [9:0] next_pointer, add_value;
//memory 3MB
reg [32767:0] mem_array[1023:0];
wire lifo_re, lifo_we,lifo_en;
//pointer
assign pointer1 = pointer;
assign lifo_we = wr & ~lifo_full;
//
assign lifo_re = rd & ~lifo_empty;
//
assign lifo_en = lifo_re ^ lifo_we;
//
assign add_value[9:0] = lifo_re? {{9{1'b1}}, 1'b0}: {10{1'b0}};
//
assign next_pointer[9:0] = pointer[9:0] +add_value[9:0] + 1'b1;
//
```

```
always @ (posedge clk) begin
  if (~rst_n)
   pointer[9:0] <= {10{1'b0}};
  else if (lifo_en)
   pointer[9:0] <= next_pointer[9:0];
end
//Status
//
assign lifo_full = pointer[9];
//
assign lifo_empty= ~|pointer[9:0];
//memory array
always @ (posedge clk) begin
  if (lifo_we)
   mem_array[pointer[9:0]] <= data_in[31:0];
end

always @ (posedge clk) begin
   if(lifo_re) begin
   data_out[31:0] <= mem_array[next_pointer[9:0]];
   end
end

endmodule
```

Module lifo_memory:

+ Inputs:
  - clk (clock): Clock signal.
  - rst_n (reset_n): Asynchronous reset signal, active low.
  - wr (write): Write signal for writing data into the memory.
  - rd (read): Read signal for reading data from the memory.
  - data_in (data_in): Input data to be written into the memory.

+ Outputs:
  - data_out (data_out): Output data read from the memory.
  - lifo_empty (lifo_empty): Signal indicating that the LIFO memory is empty.
  - lifo_full (lifo_full): Signal indicating that the LIFO memory is full.
  - pointer1 (pointer1): Value of the current pointer in the memory.

+ Reg and Wire:
  - pointer: reg [9:0] variable used to store the current value of the pointer.
  - next_pointer: wire [9:0] variable used to calculate the next value of the pointer.
  - add_value: wire [9:0] variable used to determine the value to be added to the pointer when a read operation occurs.
  - mem_array: reg [32767:0] array used to store data in the memory.

+ Control Signals:
  - lifo_we: Signal indicating a write operation to the memory.
  - lifo_re: Signal indicating a read operation from the memory.
  - lifo_en: Signal indicating the activation of the LIFO memory.

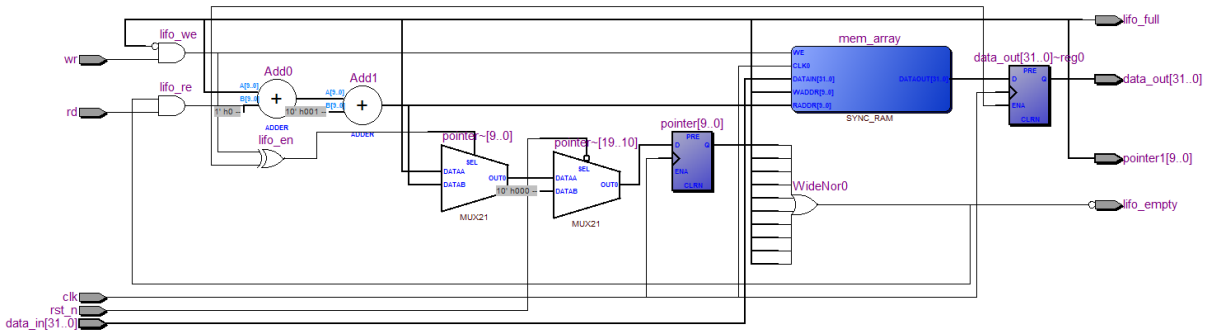
+ Assignments:
  - add_value: If a read operation (lifo_re) occurs, add_value is set to {10'b1111111110} (10 bits of 1 and 1 bit of 0), otherwise it is set to {10'b0000000000}.
  - next_pointer: The next value of the pointer is calculated by adding the current value of the pointer (pointer) to add_value and 1.
  - pointer: The value of the pointer is updated based on the clk and rst_n signals. When rst_n = 0, the pointer is reset to 0. When lifo_en = 1, the value of the pointer is updated to next_pointer.

+ Status Signals:
  - lifo_full: Determined by the most significant bit of the pointer (pointer[9]).
  - lifo_empty: Determined by the OR operation on all bits of the pointer (pointer[9:0]).

+ Memory:

- Data is written into the memory (mem_array) when lifo_we = 1 and stored at the current pointer position.
- Data is read from the memory (mem_array) when lifo_re = 1 and stored in the data_out variable.

- **RTL Viewer:**

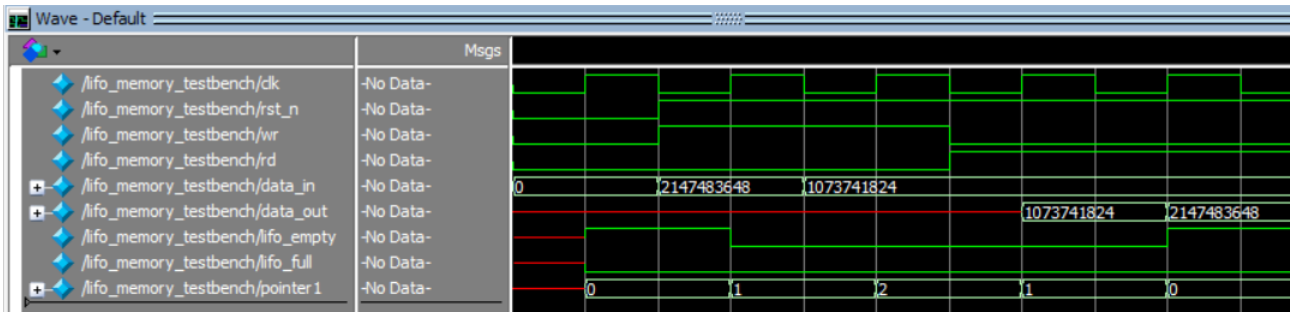


- **Testbench:**

```
1   `timescale 1ns/1ps
2
3   module lifo_memory_testbench;
4
5     reg clk, rst_n, wr, rd;
6     reg [31:0] data_in;
7     wire [31:0] data_out;
8     wire lifo_empty, lifo_full;
9     wire [9:0] pointer1;
10
11    lifo_memory uut(
12      .clk(clk),
13      .rst_n(rst_n),
14      .wr(wr),
15      .rd(rd),
16      .lifo_empty(lifo_empty),
17      .lifo_full(lifo_full),
18      .data_in(data_in),
19      .data_out(data_out),
20      .pointer1(pointer1)
21    );
22  always begin
23      #5 clk = ~clk;
24  end
25
26    initial begin
27      // Initialize inputs
28      clk = 0;
29      rst_n = 0;
30      wr = 0;
31      rd = 0;
32      data_in = 0;
33
34      // Apply reset
```

```
35      rst_n = 0;
36      #10;
37      rst_n = 1;
38      wr = 1;
39      data_in = 32'b10000000000000000000000000000000;
40      #10;
41      data_in = 32'b01000000000000000000000000000000;
42      #10;
43      wr = 0;
44      rd = 1;
45      #30;
46
47      $stop; // Stop simulation
48    end
49
50  endmodule
```

+ Clk: Clock signal.
+ rst_n: Reset signal for the FIFO, sets fifo_ov = 0 and fifo_ud = 0 when rst_n = 0 (activates the fifo_ov and fifo_ud signals).
+ wr: Write signal for data input when wr = 1.
+ rd: Read signal for data output when rd = 1.
+ data_in: 32-bit input data.
+ data_out: 32-bit output data.
+ lifo_full: Indicates that the LIFO is full when fifo_full = 1 (input provided).
+ lifo_empty: Indicates that the LIFO is empty when fifo_empty = 1 (input provided).
+ pointer1: Pointer stack of the LIFO.
+ 32'b10000000000000000000000000000000 = 2147483648
+ 32'b01000000000000000000000000000000 = 1073741824



=> With wr = 1 and the clock rising edge for the first time, data_in = 2147483648 is written into the LIFO MEMORY. On the second rising edge of the clock, data_in = 1073741824 is written into the LIFO MEMORY. When there is data in the LIFO MEMORY, lifo_empty transitions from 1 to 0. Then, with rd = 1 and consecutive rising edges of the clock twice, data_out has the values 1073741824 and 2147483648 respectively. Pointer 1 runs from 0 to 2 when data is loaded and runs from 2 back to 0 when data is read out.

2.2.2 Status signal of LIFO:

- **Verilog code:**

```
module status_signal_LIFO (clk, rst_n, wr, rd,lifo_empty,lifo_full,lifo_ov,lifo_ud,lifo_low_th,lifo_high_th,pointer);

//inputs
input clk;

input rst_n;
input wr;
input rd;
input lifo_empty;
input lifo_full;
input [10:0] pointer;

//outputs
output reg lifo_ov;
output reg lifo_ud;
output lifo_low_th;
output lifo_high_th;

//Internal signals


wire lifo_re, lifo_we, lifo_en;
//pointer
assign lifo_we = wr & ~lifo_full;
//
assign lifo_re = rd & ~lifo_empty;
//
assign lifo_en = lifo_re ^ lifo_we;
//The low threshold signal
assign lifo_low_th = (pointer[10:0] < 8);
assign lifo_high_th = (pointer[10:0] >= 8);

always @ (posedge clk) begin
    if (~rst_n) lifo_ov <= 1'b0;
    else if (lifo_re) lifo_ov <= 1'b0;
    else if (wr & lifo_full) lifo_ov <= 1'b1;
end

 //Underflow


always @ (posedge clk) begin
    if (~rst_n) lifo_ud <= 1'b0;
    else if (lifo_we) lifo_ud <= 1'b0;
    else if (rd & lifo_empty) lifo_ud <= 1'b1;
end
 endmodule
```

+ Inputs:
    - clk (clock): Clock signal.
    - rst_n (reset_n): Asynchronous reset signal, active low.
    - wr (write): Write signal for writing data into the LIFO memory.
    - rd (read): Read signal for reading data from the LIFO memory.
    - lifo_empty: Signal indicating that the LIFO memory is empty.
    - lifo_full: Signal indicating that the LIFO memory is full.
    - pointer: Value of the current pointer in the LIFO memory.

+ Outputs:
    - lifo_ov: Signal indicating LIFO memory overflow.
    - lifo_ud: Signal indicating LIFO memory underflow.
    - lifo_low_th: Signal indicating that the pointer is below a low threshold value.
    - lifo_high_th: Signal indicating that the pointer is at or above a high threshold value.
    - Control Signals:
    - lifo_we: Signal indicating a write operation to the LIFO memory.
    - lifo_re: Signal indicating a read operation from the LIFO memory.

+ Assignments:
    - lifo_we: The lifo_we signal is assigned the value of wr & ~lifo_full, which means it is active (1) when there is a write operation and the LIFO memory is not full.
    - lifo_re: The lifo_re signal is assigned the value of rd & ~lifo_empty, which means it is active (1) when there is a read operation and the LIFO memory is not empty.

- lifo_low_th: The lifo_low_th signal is assigned based on the comparison of the pointer value (pointer[9:0]) with a low threshold value (8 in this case). It is active (1) when the pointer is below the threshold.
- lifo_high_th: The lifo_high_th signal is assigned based on the comparison of the pointer value (pointer[9:0]) with a high threshold value (8 in this case). It is active (1) when the pointer is at or above the threshold.

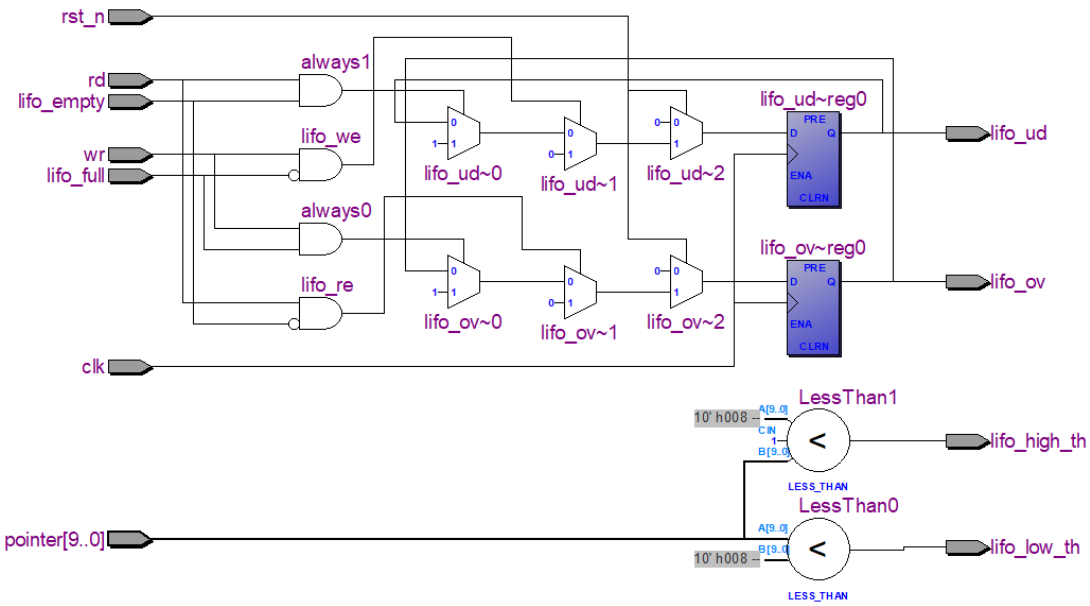
+ Overflow:

- The lifo_ov signal is set to 1 (overflow) under the following conditions:
- During reset (when ~rst_n), lifo_ov is set to 0 (no overflow).
- When there is a read operation (lifo_re), lifo_ov is set to 0 (no overflow).
- When there is a write operation (wr) and the LIFO memory is full (lifo_full), lifo_ov is set to 1 (overflow).

+ Underflow:

- The lifo_ud signal is set to 1 (underflow) under the following conditions:
- During reset (when ~rst_n), lifo_ud is set to 0 (no underflow).
- When there is a write operation (lifo_we), lifo_ud is set to 0 (no underflow).
- When there is a read operation (rd) and the LIFO memory is empty (lifo_empty), lifo_ud is set to 1 (underflow).

- **RTL Viewer:**

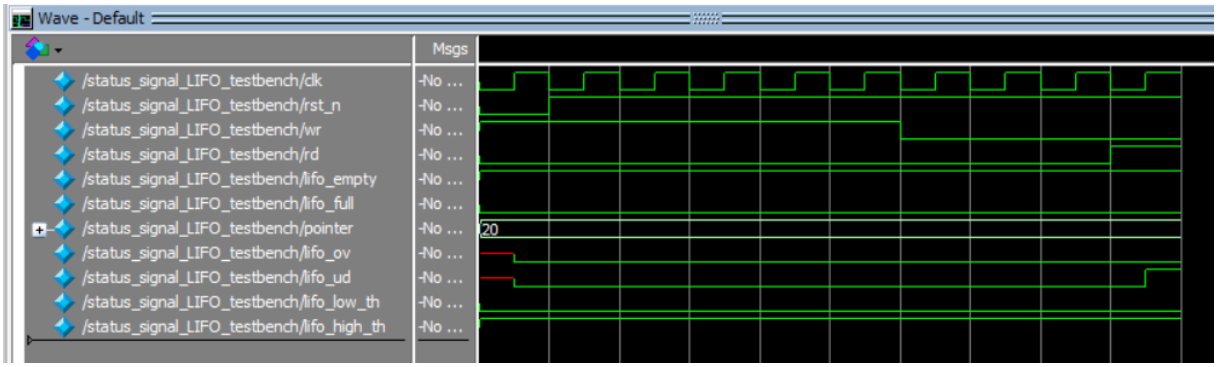


- **Testbench:**

```
1  `timescale 1ns/100ps
2
3  module status_signal_LIFO_testbench;
4
5    // Parameters
6    parameter DATA_WIDTH    = 16;
7    parameter POINTER_WIDTH = 4;
8    parameter TH_LEVEL      = (2**POINTER_WIDTH)/2;
9
10   // Inputs
11   reg clk = 0;
12   reg rst_n = 1;
13   reg wr = 0;
14   reg rd = 0;
15   reg lifo_empty = 1;
16   reg lifo_full = 0;
17   reg [24:0] pointer;
18
19   // Outputs
20   wire lifo_ov;
21   wire lifo_ud;
22   wire lifo_low_th;
23   wire lifo_high_th;
24   // Instantiate the module under test
25   status_signal_LIFO uut (
26     .clk(clk),
27     .rst_n(rst_n),
28     .wr(wr),
29     .rd(rd),
30     .lifo_empty(lifo_empty),
31     .lifo_full(lifo_full),
32     .lifo_ov(lifo_ov),
33     .lifo_ud(lifo_ud),
34     .lifo_low_th(lifo_low_th),
35     .lifo_high_th(lifo_high_th),
36     .pointer(pointer)
37   );
38
39   // Clock generation
40   always begin
41     #5 clk = ~clk;
42   end
43
```

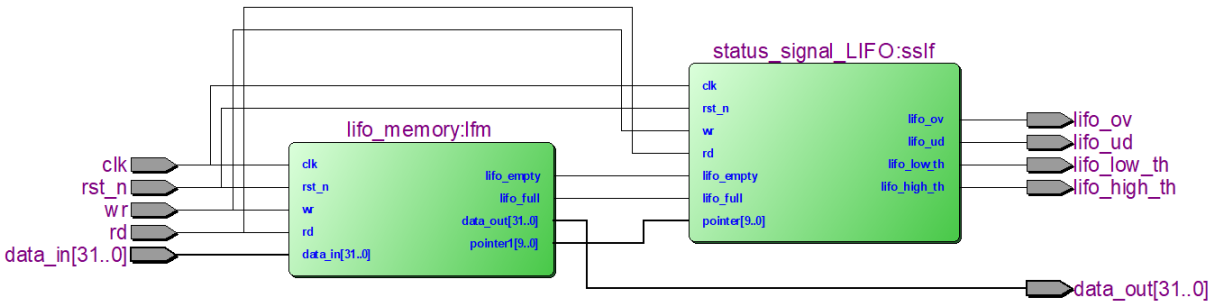
```
44      // Stimulus generation
45      initial begin
46          // Initialize signals
47          wr = 1; // Enable write
48          rd = 0; // Disable read initially
49          lifo_empty = 1;
50          lifo_full = 0;
51          pointer = 20;
52
53          // Reset
54          rst_n = 0;
55          #10 rst_n = 1;
56
57          // Test case 1: Trigger Overflow
58          #10 wr = 1;
59          #10 wr = 1;
60          #10 wr = 1;
61          #10 wr = 1; // Overflow should be triggered here
62
63          // Test case 2: Trigger Underflow
64          #10 wr = 0;
65          #10 wr = 0;
66          #10 wr = 0;
67          #10 rd = 1; // Underflow should be triggered here
68
69          // Add more test scenarios as needed
70
71          // Finish simulation
72          #10 $stop;
73      end
74
75  endmodule
```

+ Clk: Clock signal.
+ rst_n: Reset signal that sets fifo_ov = 0 and fifo_ud = 0 when rst_n = 0.
+ wr: Write signal for data input when wr = 1.
+ rd: Read signal for data output when rd = 1.
+ lifo_full: Indicates that the LIFO is full when fifo_full = 1 (input provided).
+ lifo_empty: Indicates that the LIFO is empty when fifo_empty = 1 (input provided).
+ pointer: Address of the stack slot (input) being provided.
+ lifo_ov: Indicates LIFO overflow when data is written into the LIFO even when it is full, set fifo_ov = 1.
+ lifo_ud: Indicates LIFO underflow when data is read from the LIFO even when it is empty, set fifo_ud = 1.
+ lifo_high_th: Indicates high threshold when the pointer's stack slot address is greater than or equal to 8.
+ lifo_low_th: Indicates low threshold when the pointer's stack slot address is less than 8.



=> For example, lifo_empty and lifo_full are both 0, showing that LIFO is neither empty nor full. In the case of this module, lifo_full and lifo_empty are signals generated from the LIFO MEMORY module. The rst_n signal is the reset signal for lifo_ov and lifo_ud to 0. Set the pointer address = 20 to test the two signals lifo_high_th and lifo_low_th, we see lifo_high_th = 1 and lifo_low_th = 0. Signal lifo_ov = 0 because rd & lifo_empty = 0 and signal lifo_ud = 1 because rd & lifo_empty = 1.

=> When we have submodules like lifo_mem, status_signal_LIFO. We proceed to connect them to get a complete LIFO as follows



- **Verilog code:**

```
1  module LIFO(data_out, lifo_ov, lifo_ud,lifo_low_th, lifo_high_th, clk, rst_n, wr, rd, data_in);
2    input wr, rd, clk, rst_n;
3    input[31:0] data_in;
4    output[31:0] data_out;
5    output lifo_low_th,lifo_high_th,lifo_ov,lifo_ud;
6    wire [9:0] pointer1;
7    wire lifo_full,lifo_empty;
8    wire fifo_we,fifo_rd;
9    lifo_memory lfm(clk, rst_n, wr, rd,lifo_empty,lifo_full,data_in,data_out,pointer1);
10   status_signal_LIFO sslf(clk, rst_n, wr, rd,lifo_empty,lifo_full,lifo_ov,lifo_ud,lifo_low_th,lifo_high_th,pointer1);
11 endmodule
```

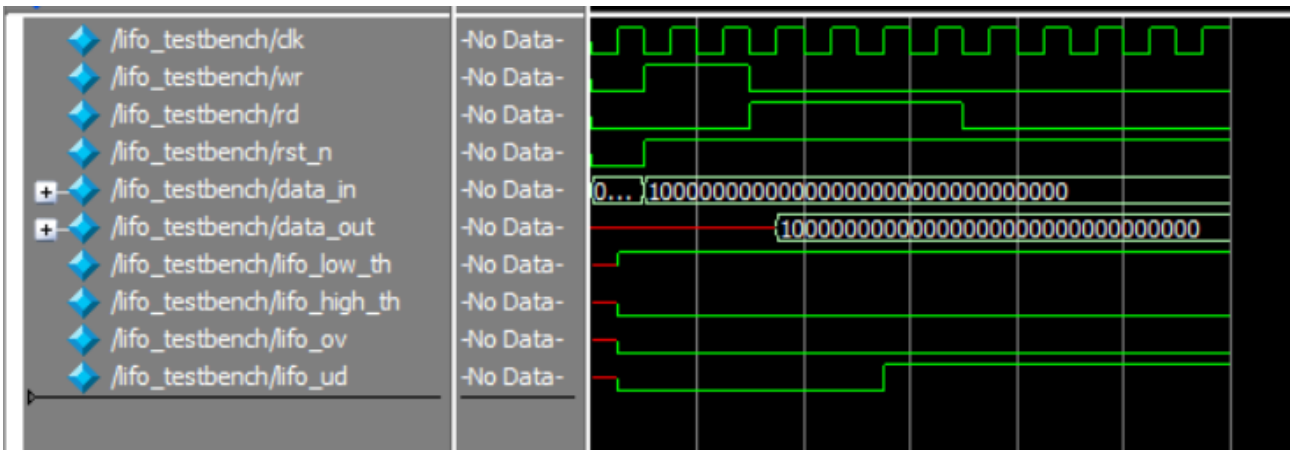
⇨ The main "LIFO" module instantiates the "lifo_memory" module to implement the LIFO memory functionality, and the "status_signal_LIFO" module to generate status signals based on the LIFO memory state. These modules work together to handle data storage and retrieval while providing status information about the LIFO memory.

- **Testbench:**

```
`timescale 1ns/1ps

module lifo_testbench;

  reg wr, rd, clk, rst_n;
  reg [31:0] data_in;
  wire [31:0] data_out;
  wire lifo_low_th, lifo_high_th, lifo_ov, lifo_ud;

  LIFO lifo1(
   .data_out(data_out),
   .lifo_ov(lifo_ov),
   .lifo_ud(lifo_ud),
   .lifo_low_th(lifo_low_th),
   .lifo_high_th(lifo_high_th),
   .clk(clk),
   .rst_n(rst_n),
   .wr(wr),
   .rd(rd),
   .data_in(data_in)
  );

  always begin
    #5 clk = ~clk;
  end
initial begin
    // Initialize inputs
    wr = 0;
    rd = 0;
    clk = 0;
    rst_n = 0;
    data_in = 0;

    // Apply reset
    rst_n = 0;
    #10;
    rst_n = 1;
    wr = 1;
    data_in = 32'b10000000000000000000000000000000;
    #20;
    wr = 0;
    rd = 1;
    #40;
    rd = 0;
    #50; // Allow for some simulation time

    $stop; // Stop simulation
end

endmodule
```

+ clk: Clock signal.
+ rst_n: Reset signal. When rst_n = 0, the fifo_ov and fifo_ud signals are set to 0 (activating the signals).
+ wr: Write signal. When wr = 1, data is written into the LIFO.
+ rd: Read signal. When rd = 1, data is read from the LIFO.
+ data_in: 32-bit input data.
+ data_out: 32-bit output data.
+ lifo_ov: Indicates LIFO overflow. If data is written into a full LIFO, lifo_ov = 1.

+ lifo_ud: Indicates LIFO underflow. If data is read from an empty LIFO, lifo_ud = 1.
+ lifo_high_th: Indicates a high threshold. lifo_high_th = 1 when the pointer's stack slot address is greater than or equal to 8.
+ lifo_low_th: Indicates a low threshold. lifo_low_th = 1 when the pointer's stack slot address is less than 8.



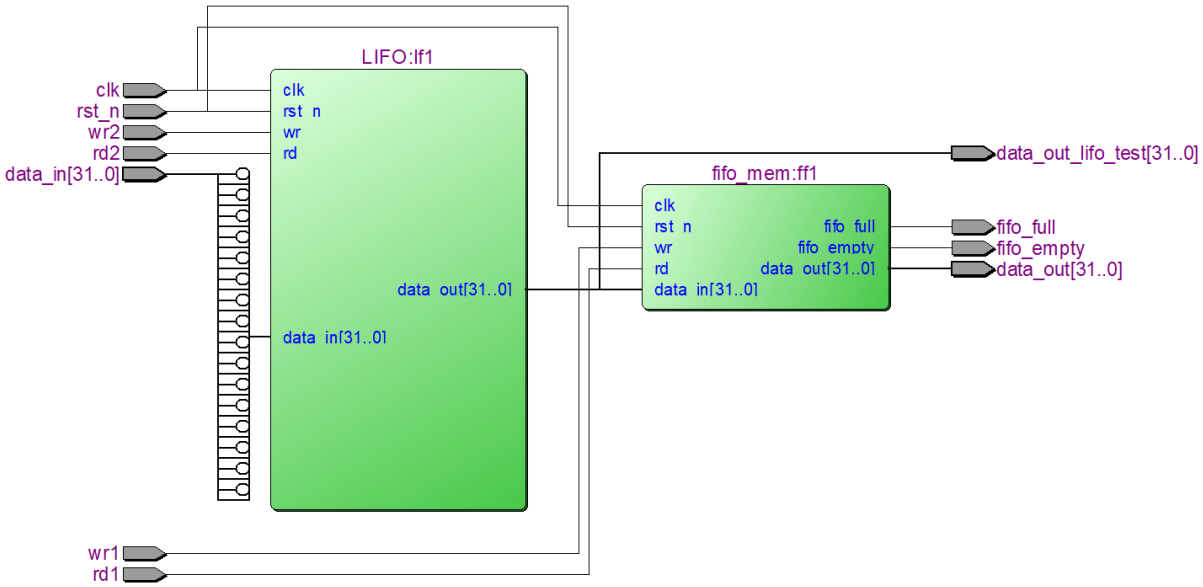
⇨ When wr = 1 and data_in = 32'b10000000000000000000000000000000, it means that the value 32'b10000000000000000000000000000000 is written into the LIFO_MEMORY. When rd = 1 and the clock signal (clk) is triggered for the first time, data_out will be 32'b10000000000000000000000000000000, as it reads the previously written data from the LIFO_MEMORY. From the second clock cycle onwards, the LIFO_MEMORY becomes empty, and lifo_ud = 1, indicating that there is no more data to be read. Additionally, lifo_low_th = 1 because the value of the pointer is less than 8, indicating that the LIFO is approaching empty.

2.3 Security (memory) :

- **Verilog code:**

```
module security(data_in,clk,rst_n,wr1, rd1,wr2, rd2,data_out,data_out_lifo_test,fifo_full, fifo_empty);
//fifo
input wr1, rd1, clk, rst_n;
wire[31:0] data_out_lifo;
wire fifo_threshold, fifo_overflow, fifo_underflow;
output fifo_full, fifo_empty;
//lifo
input wr2, rd2;
wire lifo_low_th,lifo_high_th,lifo_ov,lifo_ud;
output [31:0] data_out_lifo_test;
//key and input
input [31:0] data_in;
output[31:0] data_out;
//input "xin chao" = data_in
//xin = 00000000 01111000 01101001 01101110
//chao = 01100011 01101000 01100001 01101111
wire [31:0] X;

assign data_out_lifo_test = data_out_lifo;
assign X = data_in ^ 32'b10101010101010101010101010101010;
//KEY = 10101010101010101010101010101010

//fifo_mem ff1(data_out,fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow,clk, rst_n, wr, rd, X);
LIFO lf1(data_out_lifo, lifo_ov, lifo_ud,lifo_low_th, lifo_high_th, clk, rst_n, wr2, rd2, X);
//wr = 1 , rd = 0 => wr = 0, rd = 1
fifo_mem ff1(data_out,fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow,clk, rst_n, wr1, rd1, data_out_lifo);
//wr = 1 , rd = 0 => wr = 0, rd = 1
endmodule
```

- **RTL Viewer:**

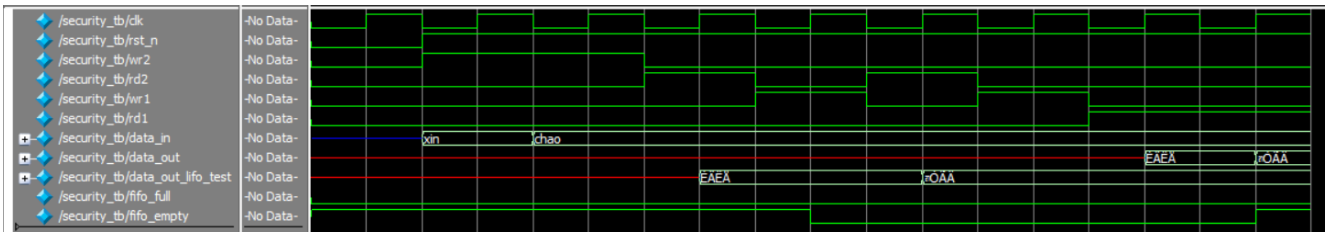


- **Testbench:**

```
1   `timescale 1ns/1ps
2
3   module security_tb;
4
5     // Inputs
6     reg clk;
7     reg rst_n;
8     reg wr1, rd1, wr2, rd2;
9     reg [31:0] data_in;
10
11    // Outputs
12    wire [31:0] data_out, data_out_lifo_test;
13    wire fifo_full, fifo_empty;
14
15    // Instantiate the security module
16    security sec_inst (
17      .data_in(data_in),
18      .clk(clk),
19      .rst_n(rst_n),
20      .wr1(wr1),
21      .rd1(rd1),
22      .wr2(wr2),
23      .rd2(rd2),
24      .data_out(data_out),
25      .data_out_lifo_test(data_out_lifo_test),
26      .fifo_full(fifo_full),
27      .fifo_empty(fifo_empty)
28    );
29
```

+ Clk: Clock signal.
+ rst_n: Reset signal. When rst_n = 0, the fifo_ov and fifo_ud signals are set to 0 (activating the signals).
+ wr1: Write signal to write data into the FIFO.
+ rd1: Read signal to read data from the FIFO.
+ wr2: Write signal to write data into the LIFO (data is inverted before storing).
+ rd2: Read signal to read data from the LIFO.
+ data_in: Input data to be written.

+ data_out: Data content stored in the memory.
+ fifo_empty: Indicates whether the memory is empty. fifo_empty = 1 when the memory is empty; otherwise, it is 0.
+ fifo_full: Indicates whether the memory is full. fifo_full = 1 when the memory is full; otherwise, it is 0.
+ data_out_lifo_test: Used to test the output of the LIFO after the inversion process before storing it into the FIFO.



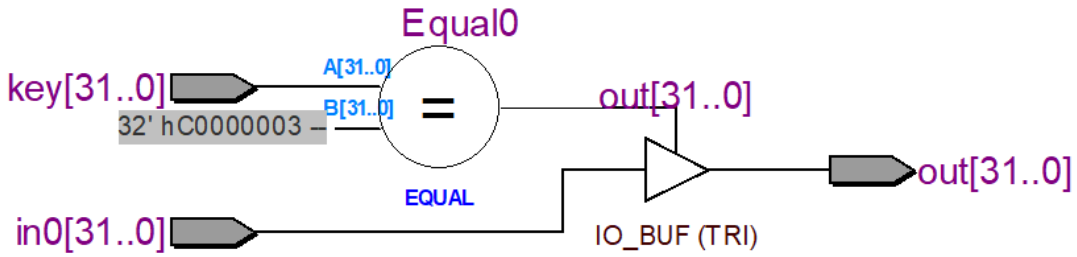
⇨ The four signals, wr1, rd1, wr2, and rd2, are used to write and encode data. The FIFO acts as the main memory, while the LIFO serves as the data encoding module. The data_in signal represents the input data to be encoded and written into the memory, while the data_out signal holds the output data that has not been decoded yet. The fifo_empty and fifo_full signals are used to indicate the empty and full status of the memory.

2.4 USER_KEY

- **Verilog code:**

```
module user_key (
    input [31:0] in0,
   input [31:0] key,
   output  reg [31:0] out
);

   always @(*) begin
      if (key == 32'b11000000000000000000000000000011)
         out <= in0;
      else
         out <= 32'bz;
   end

endmodule
```

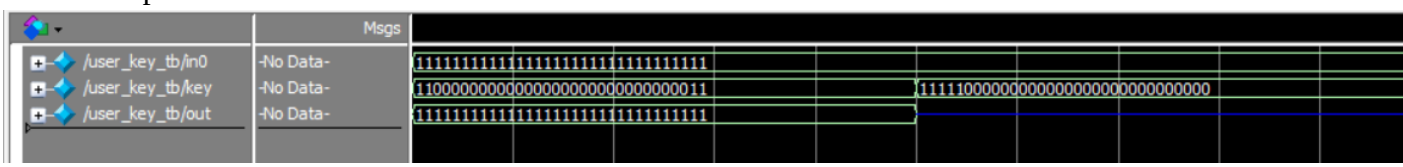
- **RTL Viewer:**



- **Testbench:**

```
1  `timescale 1ns / 1ps
2
3  module user_key_tb;
4
5    // Inputs
6    reg [31:0] in0;
7    reg [31:0] key;
8
9    // Outputs
10   wire [31:0] out;
11
12   user_key uut (
13     .in0(in0),
14     .key(key),
15     .out(out)
16   );
17
18   // Initial block
19   initial begin
20     // Initialize inputs
21     in0 = 32'b11111111111111111111111111111111;
22     key = 32'b11000000000000000000000000000011;
23     #10;
24     key = 32'b11111000000000000000000000000000;
25     #10;
26     $stop;
27   end
28
29 endmodule
```

+ Clk: Clock signal.
+ In0: Input data.
+ Key: Key input used to unlock the data.
+ Out: Output data

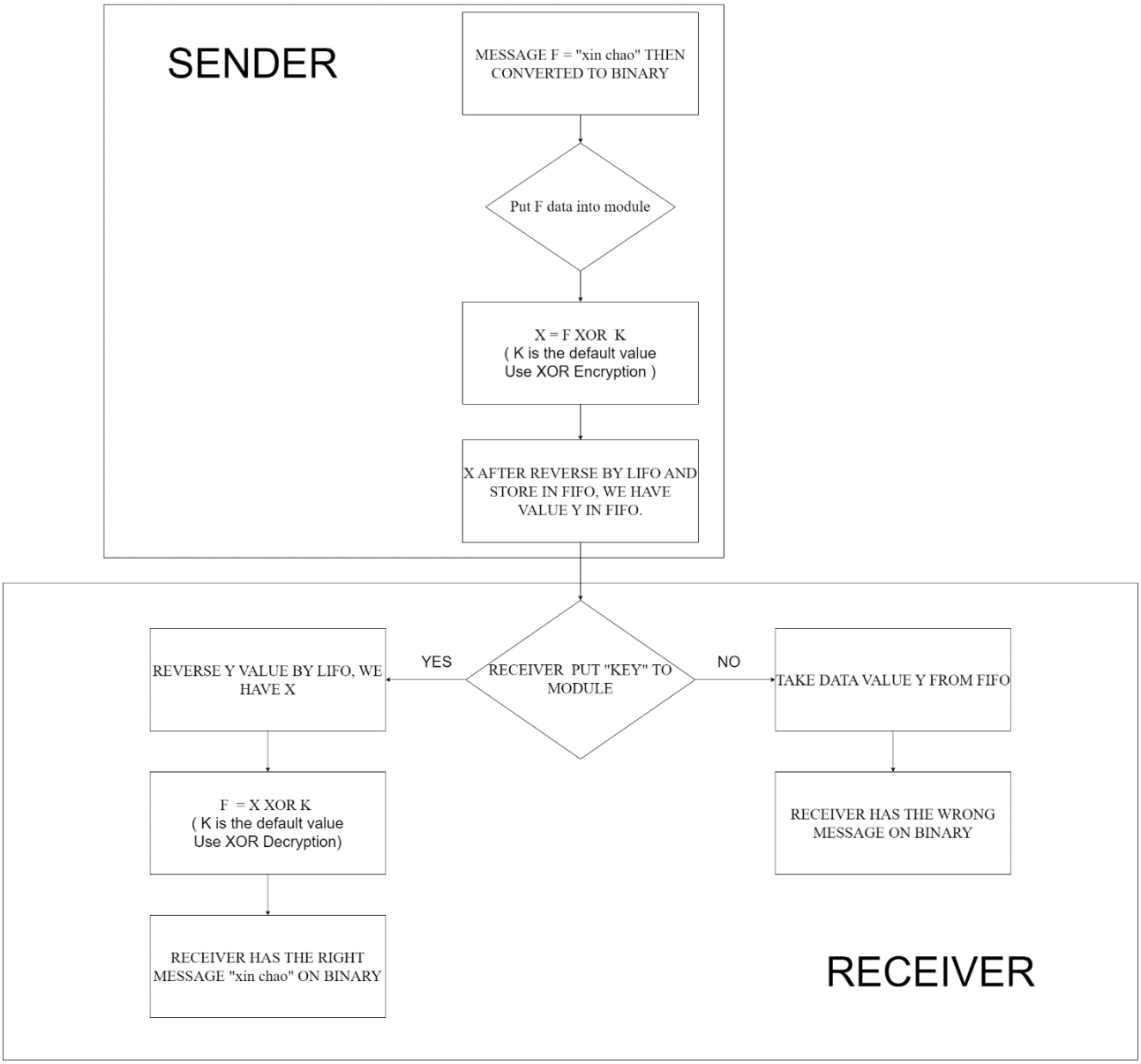


⇨ If the correct KEY is entered, the desired output data will be provided. However, if an incorrect KEY is entered, the output data will be represented as 32'bZ.

CHAPTER 3: PROPOSED SYSTEM

3.1 Flowchart

FLOW CHART

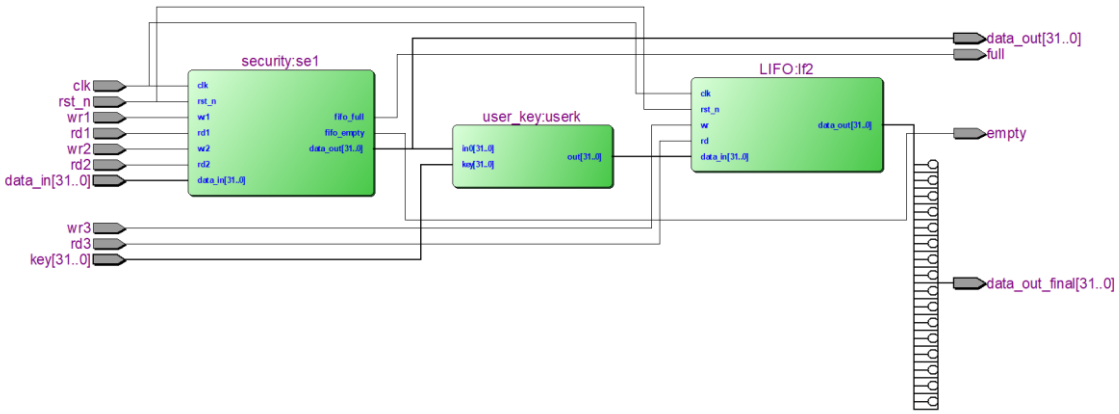


3.2 Security_FPGA system:

- **Verilog code:**

```
module security_FPGA(data_in,wr1, rd1, clk, rst_n,wr2, rd2,wr3,rd3,key,data_out,data_out_final,empty,full);
//security
input [31:0] data_in;
output[31:0] data_out;
output empty,full;
input wr1, rd1, clk, rst_n,wr2, rd2;
wire lifo_low_th,lifo_high_th,lifo_ov,lifo_ud;
wire [31:0] data_out_lifo_test;
output wire [31:0] data_out_final;
wire fifo_full, fifo_empty;

//key
wire [31:0]out;
input [31:0]key;
wire [31:0] X;
//LIFO2
input wr3,rd3;
assign empty = fifo_empty;
assign full =fifo_full;

security se1(data_in,clk,rst_n,wr1, rd1,wr2, rd2,data_out,data_out_lifo_test,fifo_full, fifo_empty);

user_key userk(data_out,key,out);

LIFO lf2(X, lifo_ov, lifo_ud,lifo_low_th, lifo_high_th, clk, rst_n, wr3, rd3, out);

assign data_out_final = X ^ 32'b10101010101010101010101010101010;
endmodule
```

- **RTL Viewer:**

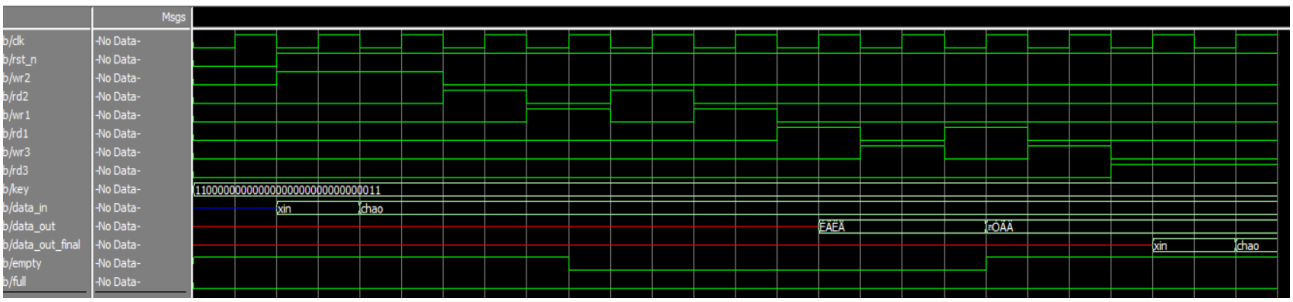


- **Testbench:**

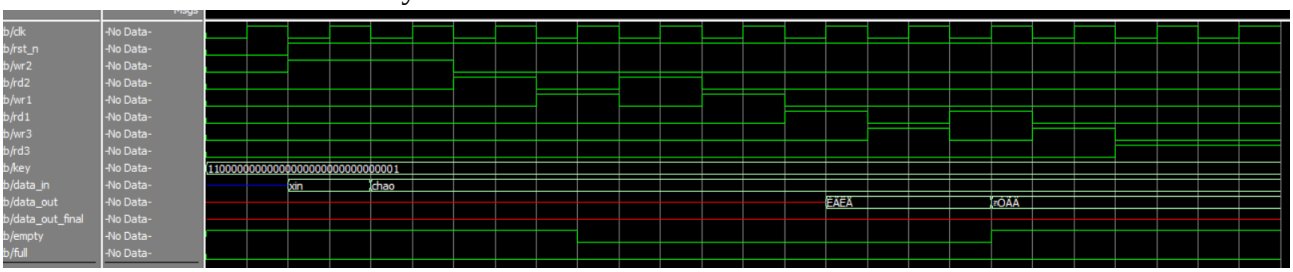
```
`timescale 1ns / 1ps

module security_FPGA_tb;

  // Parameters

  // Inputs
  reg [31:0] data_in;
  reg wr1, rd1, clk, rst_n, wr2, rd2, wr3, rd3;
  reg [31:0] key;
  wire empty,full;
  // Outputs
  wire [31:0] data_out, data_out_final;

  // Instantiate the module under test
  security_FPGA uut (
    .data_in(data_in),
    .wr1(wr1),
    .rd1(rd1),
    .clk(clk),
    .rst_n(rst_n),
    .wr2(wr2),
    .rd2(rd2),
    .wr3(wr3),
    .rd3(rd3),
    .key(key),
    .data_out(data_out),
    .data_out_final(data_out_final),
    .empty(empty),
    .full(full)
  );

  // Clock generation
```

```
34    always begin
35        #5 clk = ~clk;
36    end
37
38    // Initial block
39    initial begin
40        // Initialize inputs
41        data_in = 32'bz;
42        wr1 = 0;
43        rd1 = 0;
44        clk = 0;
45        rst_n = 0;
46        wr2 = 0;
47        rd2 = 0;
48        wr3 = 0;
49        rd3 = 0;
50        key = 32'b11000000000000000000000000000011;
51
52        #10;
53        rst_n = 1;
54        wr2 = 1;
55        data_in = 32'b00000000_01111000_01101001_01101110;
56        #10;
57        data_in = 32'b01100011_01101000_01100001_01101111;
58        #10;
59        wr2 = 0;
60        rd2 = 1;
61        #10;
62        rd2 = 0;
63        wr1 = 1;
64        #10;
65        rd2 = 1;
66        wr1 = 0;
67        #10;
68        rd2 = 0;
69        wr1 = 1;
70        #10;
71        wr1 = 0;
72        rd1 = 1;
73        #10;
74        rd1 = 0;
75        wr3 = 1;
76        #10;
77        rd1 = 1;
78        wr3 = 0;
79        #10;
80        rd1 = 0;
81        wr3 = 1;
82        #10;
83        wr3 = 0;
84        rd3 = 1;
85        #20;
86
87
88        $stop;
89    end
90
91  endmodule
```

+ Clk: clock signal
+ rst_n: reset signal that sets fifo_ov = 0 and fifo_ud = 0 when reset_n = 0 (activates the fifo_ov and fifo_ud signals)
+ wr1: write data to the FIFO
+ rd1: read data from the FIFO
+ wr2: write data to the LIFO (reverse the data)
+ rd2: read data from the LIFO
+ wr3: write data from the FIFO to the LIFO
+ rd3: read data from the LIFO
+ key: user-provided data for data retrieval
+ data_in: input data content for information encoding
+ data_out: content stored in memory after information encoding
+ data_out_final: content stored in memory after information decoding
+ empty: indicates whether the memory is empty (empty = 1) or not (empty = 0)
+ full: indicates whether the memory is full (full = 1) or not (full = 0).



⇨ The security system is composed of three main modules. One module combines FIFO and LIFO, where FIFO acts as the main memory, another module handles USER_KEY input, and a LIFO module is responsible for reversing the order and outputting the data. The sender writes data into the memory using signals wr2, rd2, wr1, and data_in. The receiver retrieves the signals using rd1, wr3, rd3, and key. If the correct key is entered, rd3 will output the desired result provided by the sender. The signal Data_out represents the data stored in the memory (information that has been encoded), and the receiver must enter the correct USER_KEY to decode the information. The signals empty and full indicate the status of the memory.



⇨ The security system produces a result of "32'bz" if the receiver enters the wrong KEY.

CHAPTER 4: CONCLUSION:

4.1 Result:

- The combination of FIFO and LIFO in the system allows for flexible data writing and reading. FIFO is used to store data in the order it arrives, while LIFO can reverse the order of the data. This provides high flexibility for data processing and shuffling within the system.
- The system also supports information encryption. By using input data and storing the encoded information in memory, the system protects important information from unauthorized access.
- The system also ensures information security. The receiver must enter the correct USER_KEY to decrypt the information, ensuring that only those with the accurate key can access and decode the data.

4.2 Drawbacks:

- Vulnerability to Key Guessing Attacks: If the system relies solely on a single USER_KEY for decryption, it may be vulnerable to key guessing attacks. An attacker could potentially guess or brute-force the key, compromising the security of the system.
- Limited Compatibility: The system may have difficulties in being compatible and working seamlessly with other systems, causing challenges in data exchange or integration with other applications.
- Subpar Performance: The system may fail to meet performance requirements, resulting in slow response times or consuming more system resources than anticipated.

4.3 Direction of development:

- Scaling and expandability: If the system was designed for a small scale, consider scaling up to support more users and handle larger data volumes. Use technologies like cloud and distributed servers to enhance the system's scalability.
- Performance optimization: Continuously optimize the system's performance, improve response times, and optimize resource utilization. Explore the application of optimization algorithms and techniques to enhance performance and optimize processes and workflows.
- Enhancing security: Ensure that the system is tightly secured and compliant with security standards. Update to the latest security measures and conduct regular testing to detect and address security vulnerabilities.
- Integration of new features: Consider integrating new technologies such as Artificial Intelligence(AI), Machine learning, virtual reality, or blockchain to enhance the system's capabilities and value.