Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и кибербезопасности
Высшая школа компьютерных технологий и информационных систем

# Отчёт по лабораторной работе № 5

Дисциплина: Автоматизация проектирования дискретных устройств (на английском языке).

Выполнил студент гр. 5130901/10101 _____ Д.Л. Симоновский
(подпись)

Руководитель _____ А.А. Федотов
(подпись)

“09” марта 2024 г.

Санкт-Петербург
2024

**Оглавление**

# 1. Список иллюстраций:

# 2. Цель упражнения:

Пройти цикл проектирования в рамках пакетов Quartus и ModelSim, включая следующие этапы:

- Создание проекта.
- Разработка описания модулей с использованием конструкций расширения SystemVerilog.
- Разработка теста на языке SystemVerilog и моделирование.
- Отладка проекта.

# 3. Алгоритм работы проекта:

Разработать описание LFSR по следующему полиному (вариант 18):

$$x^7 + x^6 + x^3 + x + 1$$

Типом реализации логического элемента в обратной связи – XOR.

# 4. Решение:

Разработаем LFSR модуль:

```systemverilog
`timescale 1ns / 1ns
module LFSR_7_6_3_1_0_F (
  input  bit       CLK,
  input  bit       RST,
  input  bit       ENA,
  output bit [7:1] LFSR_out
);
  always_ff @(posedge CLK, posedge RST)
    if (RST) LFSR_out <= 1'b1;
    else if (ENA)
      if (LFSR_out == '0) LFSR_out <= 1'b1;
      else LFSR_out <= {LFSR_out[6:1], LFSR_out[7] ^ LFSR_out[6] ^ LFSR_out[3] ^ LFSR_out[1]};
endmodule
```

Данный модуль – сдвигающий регистр, с асинхронным сбросом. Сброс выполняется в 1 т.к. в противном случае LFSR перестанет работать.

Получившаяся RTL структура выглядит следующим образом:

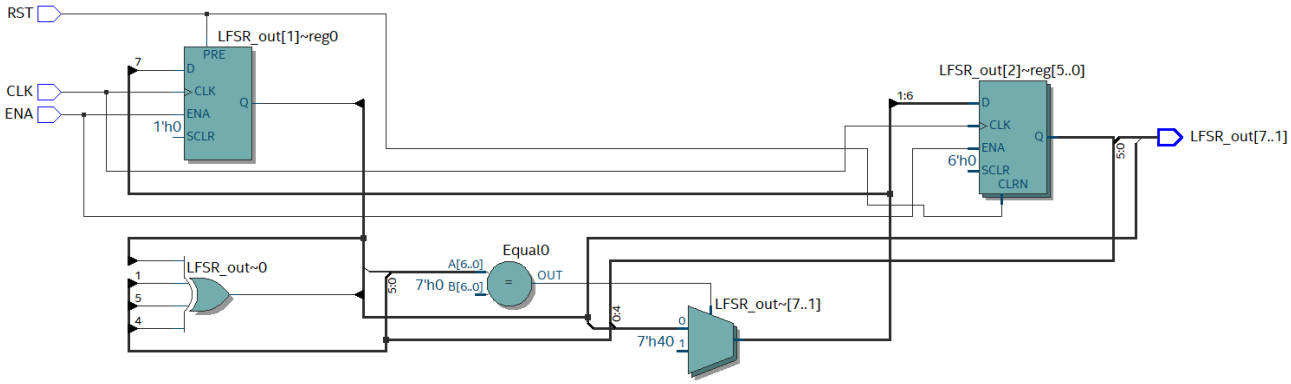


*Рис. 4.1. RTL структура LFSR модуля.*

Теперь составим тестовый модуль для данного модуля:

```sv
`timescale 1ns / 1ns
module tb_LFSR_7_6_3_1_0_F ();
  bit       CLK;
  bit       RST;
  bit       ENA;
  bit [7:1] LFSR_out;

  LFSR_7_6_3_1_0_F LFSR_7_6_3_1_0_F_inst (.*);

  localparam CLK_PERIOD = 20;

  initial forever #(CLK_PERIOD / 2) CLK = ~CLK;

  bit [7:1] CNT_int = '0;

  bit [7:1] LFSR_out_start = '0;

  initial begin
    RST = '1;
    #(CLK_PERIOD * 3 / 4);
    RST = '0;
    #(CLK_PERIOD * 5 / 4);
    ENA = '1;
    forever begin
      @(negedge CLK);
      if (CNT_int == '0) LFSR_out_start = LFSR_out;
      else if (LFSR_out_start == LFSR_out) break;
      CNT_int += 1;
    end
    #(CLK_PERIOD * 5);
    $stop;
  end

endmodule
```

Данный модуль позволит нам посчитать период. Т.к. степень полинома 7, мы ожидаем период равный 127 ($2^7 - 1$).
Получившаяся вейформа выглядит следующим образом:

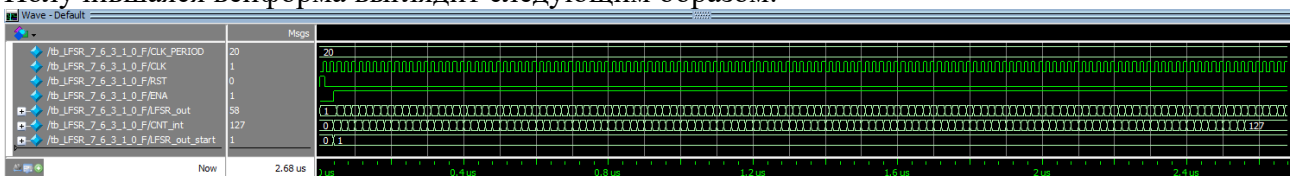


*Рис. 4.2. Вейформа для модуля.*

Как мы видим, мы получили ожидаемый период в 127 единиц.
Теперь создадим модуль для создания гистограммы:

```sv
`timescale 1ns / 1ns
module histogram_unit #(
  parameter MAX_NUMBER = ((1 << 7) - 1),
  parameter SIZE       = 7
) (
  input  bit                          CLK,
  input  bit [$clog2(MAX_NUMBER) - 1:0] d_in,
  input  bit                          RST,
  input  bit                          ENA,
  output bit [             SIZE - 1:0] mem_out
);
  bit [SIZE - 1:0] mem_arr[0:MAX_NUMBER];

  bit [SIZE - 1:0] mem_in;
  bit [$clog2(MAX_NUMBER) - 1:0] adr_in, adr_clear;

  initial for (int i = 0; i <= MAX_NUMBER; i++) mem_arr[i] = 0;

  assign mem_in = RST ? '0 : (ENA ? mem_arr[adr_in] + 1'b1 : mem_arr[adr_in]);
  assign adr_in = RST ? adr_clear : d_in;

  always @(posedge CLK) begin : building_histogram
    mem_arr[adr_in] <= mem_in;
    mem_out         <= mem_arr[adr_in];
  end

  always_ff @(posedge CLK, negedge RST) begin : clearing_array
    if (~RST) adr_clear <= '0;
    else adr_clear <= adr_clear + 1'b1;
  end

endmodule
```

Здесь MAX_NUMBER – параметр, определяющий максимальное число в гистограмме, а SIZE – размерность данных в гистограмме.
Данный модуль использует память mem_arr, в которой каждый такт по введенному адресу (d_in) добавляется единица, также присутствует возможность отчистки памяти, однако для полной отчистки понадобится MAX_NUMBER тактов.
Получившаяся RTL схема выглядит следующим образом:

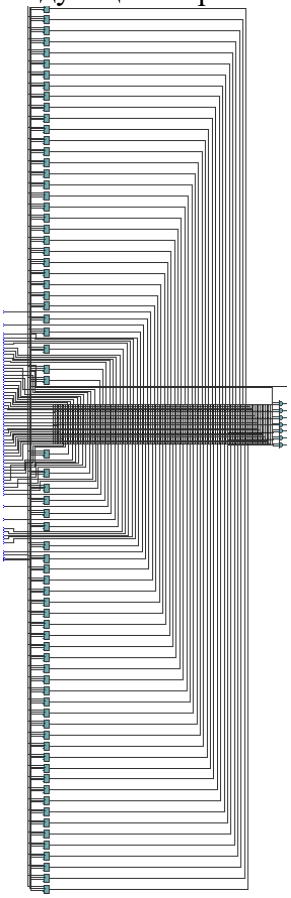

*Рис. 4.3. RTL Viewer для разработанного модуля.*

Как можно заметить, тут нет как таковой памяти, все строится на регистрах. Причина этого в том, что мы пытаемся одновременно читать значения из памяти и записывать в неё, из-за чего квартус решает, что это лучше сделать, используя регистровые схемы.
Это можно исправить, заменив строку 19 на следующую:

```
1  assign mem_in = RST ? '0 : (ENA ? mem_out + 1'b1 : mem_out);
```

Однако это приведет к тому, что вместо увеличения заданного значения на 1, мы будем записывать результат суммы в следующую цифру, поданную на вход. В контексте нашей задачи это практически не повлияет на результат (если мы изначально ожидаем корректность разработанного модуля), однако по моему мнению лучше использовать схему на регистрах. Как от неё избавиться будет рассмотрено позже.
Теперь напишем тест для созданного модуля:

```
`timescale 1ns / 1ns
module tb_histogram_unit ();
  parameter SIZE = 5;
  parameter MAX_NUMBER = 127;
  bit [$clog2(MAX_NUMBER) - 1:0] d_in;
  bit                            CLK;
  bit                            ENA;
  bit                            RST;
  bit [             SIZE - 1:0] mem_out;

  localparam CLK_PERIOD = 20;

  initial forever #(CLK_PERIOD / 2) CLK = ~CLK;

  histogram_unit #(
    .SIZE      (SIZE),
    .MAX_NUMBER(MAX_NUMBER)
  ) histogram_unit_inst (
    .*
  );

  initial begin
    ENA = '1;
    for (int i = 0; i < (MAX_NUMBER + 1) * 8; i++) begin
      @(negedge CLK) d_in += 18;
    end
    @(negedge CLK)
    RST = '1;
    @(negedge CLK)
    $stop;
  end

endmodule
```

Запустим этот тестовый модуль:

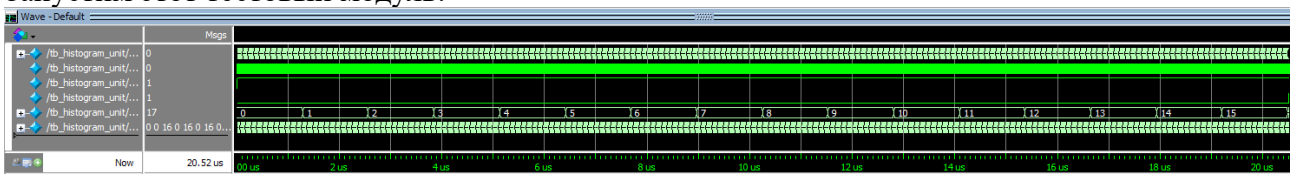


*Рис. 4.4. Результат тестирования модуля.*

Посмотрим данные в памяти:

Memory Data - /tb_histogram_unit/histogram_u

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 00000000 | 0 | 0 | 16 | 0 | 16 | 0 | 16 | 0 |
| 00000008 | 16 | 0 | 16 | 0 | 16 | 0 | 16 | 0 |
| 00000010 | 16 | 0 | 16 | 0 | 16 | 0 | 16 | 0 |
| 00000018 | 16 | 0 | 16 | 0 | 16 | 0 | 16 | 0 |
| 00000020 | 16 | 0 | 16 | 0 | 16 | 0 | 16 | 0 |
| 00000028 | 16 | 0 | 16 | 0 | 16 | 0 | 16 | 0 |
| 00000030 | 16 | 0 | 16 | 0 | 16 | 0 | 16 | 0 |
| 00000038 | 16 | 0 | 16 | 0 | 16 | 0 | 16 | 0 |
| 00000040 | 16 | 0 | 16 | 0 | 16 | 0 | 16 | 0 |
| 00000048 | 16 | 0 | 16 | 0 | 16 | 0 | 16 | 0 |
| 00000050 | 16 | 0 | 16 | 0 | 16 | 0 | 16 | 0 |
| 00000058 | 16 | 0 | 16 | 0 | 16 | 0 | 16 | 0 |
| 00000060 | 16 | 0 | 16 | 0 | 16 | 0 | 16 | 0 |
| 00000068 | 16 | 0 | 16 | 0 | 16 | 0 | 16 | 0 |
| 00000070 | 16 | 0 | 16 | 0 | 16 | 0 | 16 | 0 |
| 00000078 | 16 | 0 | 16 | 0 | 16 | 0 | 16 | 0 |

*Рис. 4.5. Данные в памяти.*

Как мы видим, модуль гистограммы корректно обрабатывает входную последовательность, а также очищает данные по RST.

Теперь напишем модуль верхнего уровня, объединив LFSR и модуль гистограммы, чтоб проверить, что в LFSR все случайные числа равновероятны:

```systemverilog
`timescale 1ns / 1ns
module lab_MS_SV_3 (
  input  bit       CLK,
  input  bit       RST,
  input  bit       ENA,
  output bit [6:0] mem_out
);

  bit [6:0] LFSR_out;
  bit [6:0] d_in;

  assign d_in = LFSR_out;

  LFSR_7_6_3_1_0_F LFSR_7_6_3_1_0_F_inst (
    .CLK,
    .RST,
    .ENA,
    .LFSR_out
  );

  histogram_unit histogram_unit_inst (
    .CLK,
    .d_in,
    .RST,
    .ENA,
    .mem_out
  );

endmodule
```

Теперь напишем тестовый модуль:

```systemverilog
`timescale 1ns / 1ns
module tb_lab_MS_SV_3 ();
  bit       CLK;
  bit       RST;
  bit       ENA;
  bit [6:0] mem_out;

  lab_MS_SV_3 lab_MS_SV_3_unit (.*);

  localparam CLK_PERIOD = 20;

  initial forever #(CLK_PERIOD / 2) CLK = ~CLK;

  initial begin
    RST = '1;
    ENA = '1;
    #(CLK_PERIOD / 2);
    RST = '0;
    repeat(127 * 5) @(negedge CLK);
    $stop;
  end

endmodule
```

Мы 5 раз повторяем период, который был получен ранее, ожидая, что в гистограмме все значения от 1 до 127 будут равны 5. Проверим это:

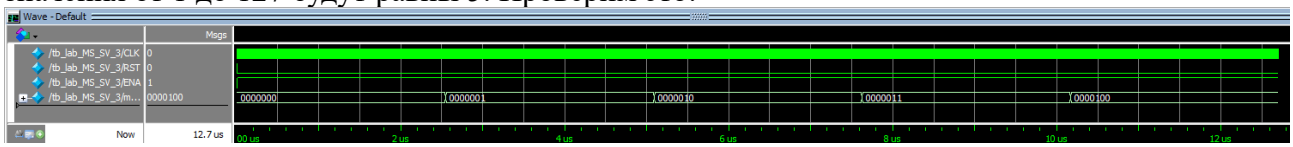


*Рис. 4.6. Результат тестирования модуля.*

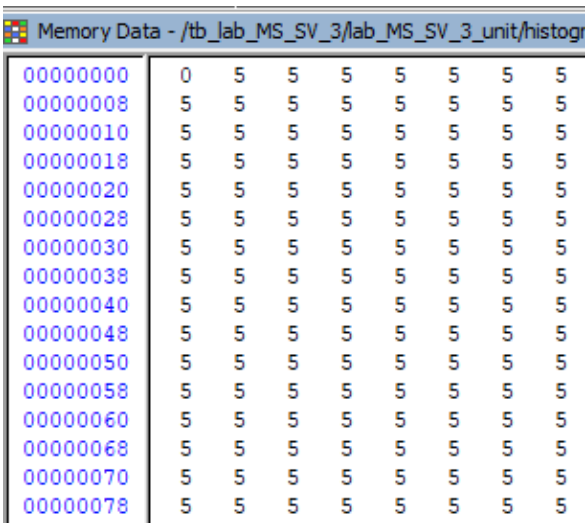


*Рис. 4.7. Данные в памяти гистограммы.*

Как мы видим, результат соответствует ожиданиям.

Теперь разработаем модуль для тестирования на плате:

```
module db_lab_MS_SV_3 (
  (* altera_attribute = "-name IO_STANDARD \"3.3-V LVCMOS\"", chip_pin = "23" *)
  input bit CLK
);

  bit       RST = 1'b1;
  bit       ENA = 1'b1;
  bit [6:0] mem_out;

  lab_MS_SV_3 lab_MS_SV_3_ints (.*);
  SP_unit SP_unit_inst (
    .source    ({RST, ENA}),
    .source_clk(CLK)
  );
endmodule
```

Используя ISSP, будем редактировать сигналы RST и ENA, а благодаря Signal Tap II наблюдать за результатом:

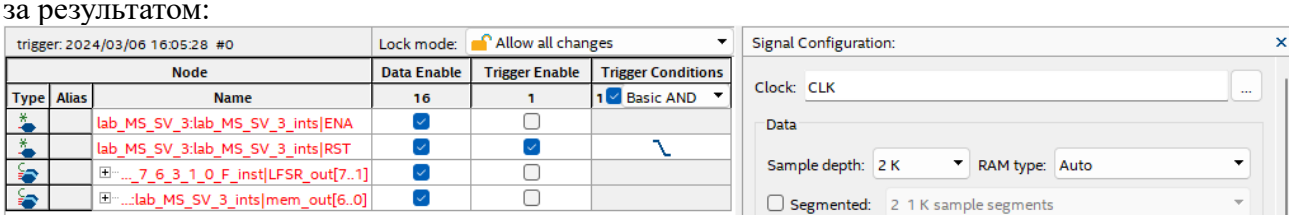


*Рис. 4.8. Настройки Signal Tap II.*

Выполним загрузку разработанного модуля на плату и запустим тестирование, переведя RST в 0:

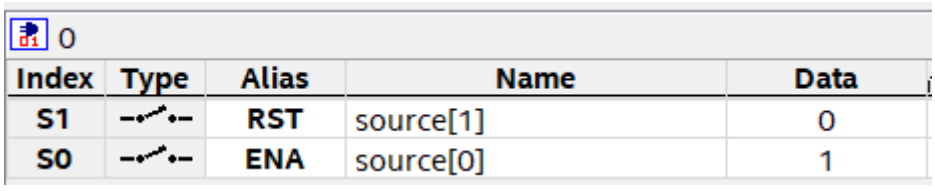


*Рис. 4.9. Настройки ISSP.*

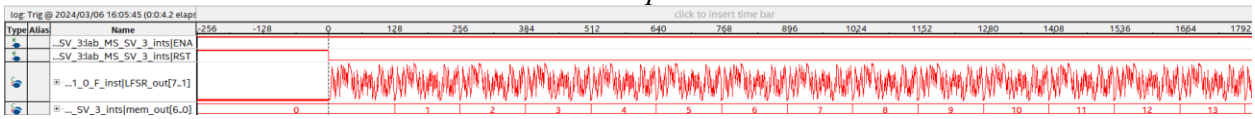

*Рис. 4.10. Signal Tap II, результат.*

Как мы видим, LFSR выдает случайные числа, а mem_out увеличивается с 0 до 13 т.е. ведется подсчет циклов с начала работы.

Теперь в соответствии с заданием в модуле histogram необходимо заменить mem_arr на однопортовую память. Проблема состоит в том, что в один такт необходимо выполнить считывание значения из памяти и на основании этого записать в ту же ячейку новые данные. Выполнить это за один такт нереально, поэтому необходимо добавить PLL, который умножит внутреннюю частоту. Схема будет следующей: на фронте clk сохраняется значение на входе d_in, после чего на спаде clk_50 (clk с частотой в 2 раза большей) мы загрузим на вход памяти адрес, а на второй спад мы на вход памяти поместим обновленные данные:

```
`timescale 1ns / 1ns
module histogram_unit #(
  parameter MAX_NUMBER = ((1 << 7) - 1),
  parameter SIZE       = 7
) (
  input  bit                          CLK,
  input  bit [$clog2(MAX_NUMBER) - 1:0] d_in,
  input  bit                          RST,
  input  bit                          ENA,
  output bit [             SIZE - 1:0] mem_out
);

  bit [SIZE - 1:0] mem_in;
  bit [$clog2(MAX_NUMBER) - 1:0] adr_in, adr_clear;

  bit [$clog2(MAX_NUMBER) - 1:0] d_in_temp;

  bit                            clk_50;
  bit [SIZE - 1:0] mem_out_2;

  RAM RAM_inst (
    .address(adr_in),
    .data   (mem_in),
    .clock  (!clk_50),
    .wren   (ENA),
    .q      (mem_out_2)
  );

  always_ff @(posedge CLK) d_in_temp <= d_in;
  always_ff @(negedge CLK) mem_out <= mem_out_2;

  PLL PLL_inst (
    .inclk0(CLK),
    .c0    (clk_50)
  );

  assign mem_in = RST ? '0 : (ENA ? mem_out_2 + !CLK : mem_out_2);
  assign adr_in = RST ? adr_clear : d_in_temp;

  always_ff @(negedge clk_50) begin : clearing_array
    if (~RST) adr_clear <= '0;
    else adr_clear <= adr_clear + 1'b1;
  end

endmodule
```

Стоит отметить, что теперь память будет стираться в 2 раза быстрее (по 2 адреса за такт).
Посмотрим на то, как выглядит RTL схема модуля:

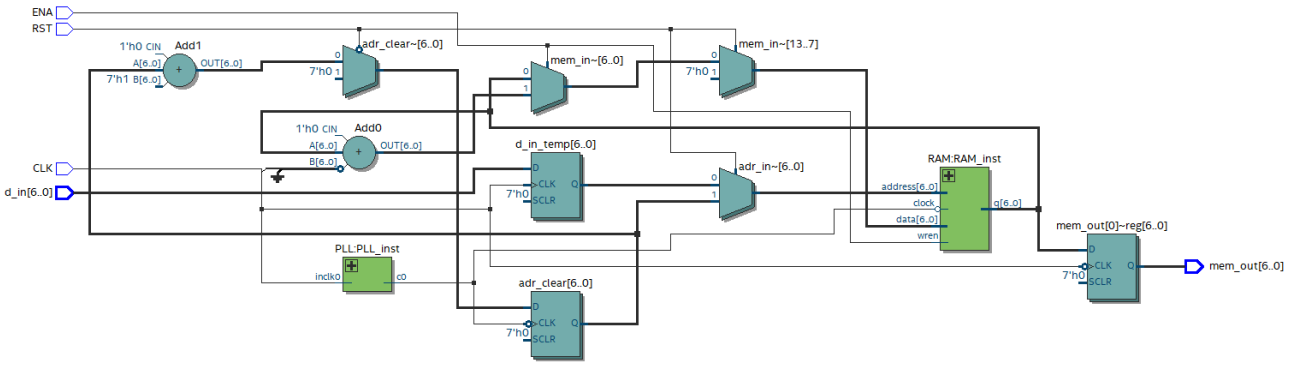


*Рис. 4.11. RTL схема модуля.*

Как мы видим, такой способ действительно помог избавиться от регистровой схемы памяти, однако сильно усложнил проект т.к. требует PLL.
Теперь необходимо повторить тестирование этого модуля, чтоб проверить корректность его работы:

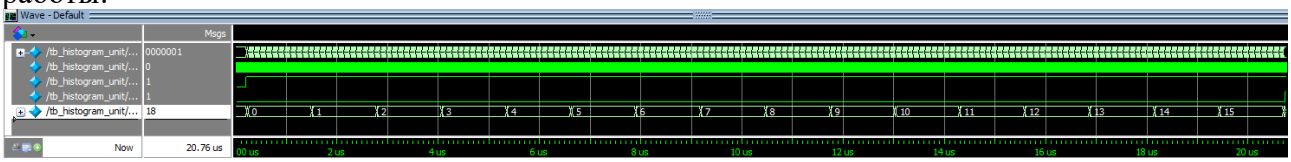


*Рис. 4.12. Результат тестирования модуля.*

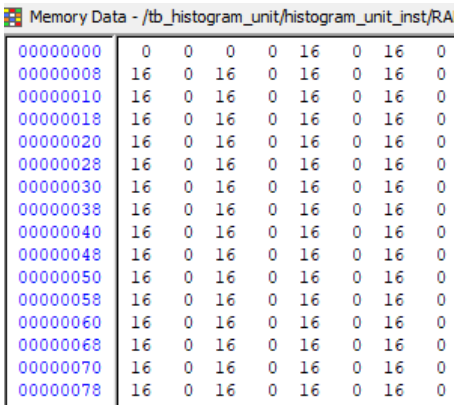


*Рис. 4.13. Данные в памяти.*

Как мы видим, результаты соответствуют ожиданиям.

Тестирование модуля верхнего уровня также повторило предыдущие тесты, поэтому перейдем сразу к тестированию на плате.

Настроим Signal Tap II следующим образом:

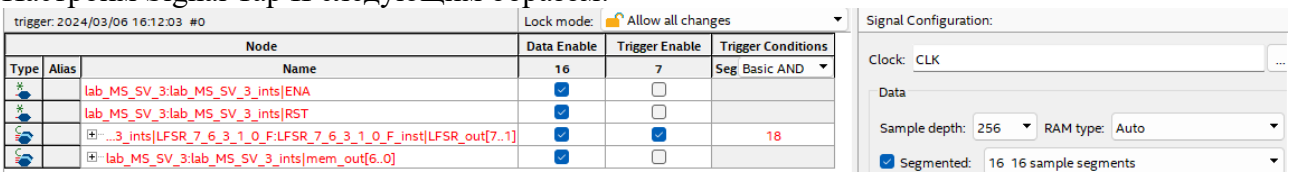


*Рис. 4.14. Настройки Signal Tap II.*

Важной особенностью новой памяти является то, что её можно посмотреть через ISMC. Если сделать это при RST = 1, мы увидим следующий результат:

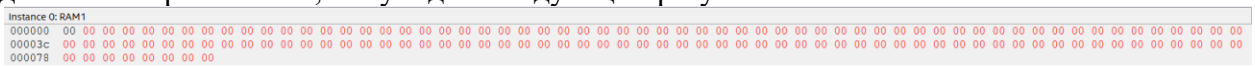


*Рис. 4.15. Память при RST = 1.*

И это логично т.к. память постоянно сбрасывается.

Теперь поставим RST в 0 и считаем память:

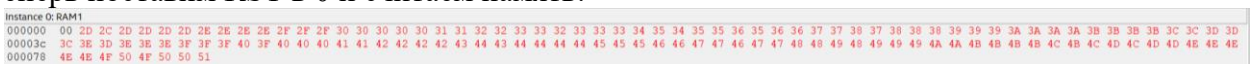


*Рис. 4.16. Память при RST = 0.*

Как мы видим, 0 ячейка осталась 0 т.к. наш генератор не должен выдавать это число, а вот остальные ячейки имеют случайные значения, хотя ожидалось что они будут одинаковыми. Это связано с тем, что ISMC не работает на достаточных частотах и не может выгрузить всю память сразу, а делает это постепенно, откуда и возникают отличия.

В Signal Tap II увидим:

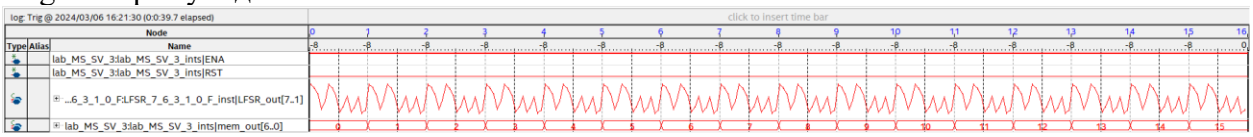


*Рис. 4.17. Signal Tap II.*

Мы видим, что было зафиксировано 16 отрезков, когда встречалось требуемое число. Оно встречается раз в цикл (как и ожидалось), а также соседние измерения имеют одинаковую форму т.к. цикл формирует одинаковую последовательность чисел.

# 5. Вывод:

В ходе лабораторной работы успешно пройден цикл проектирования, начиная с создания проекта и разработки модулей с использованием расширений SystemVerilog. Использование SystemVerilog предоставило широкий спектр новых возможностей по сравнению с Verilog, облегчая процесс разработки и улучшая читаемость кода.

Отладка проекта осуществлялась с помощью инструментов In-System Sources and Probes Editor и SignalTap II, что значительно повысило эффективность процесса. Эти инструменты позволили быстро выявить и исправить ошибки, что является ключевым аспектом при работе с любым проектом.