Проектирование систем на кристалле с программируемой архитектурой

Лабораторная работа №2

Цели работы:

1. Изучить особенности описания modport интерфейсов в HDL SystemVerilog;
2. Изучить дополнительные конструкции HDL SystemVerilog (package, typedef, enum);
3. Изучить алгоритм работы последовательного CRC-16;
4. Дополнить HDL модуль из Л.Р. №1 регистровой картой APB3 с использованием изученных доп. конструкций HDL SystemVerilog;
5. Имплементировать расчёт CRC-16 в соответствии с вариантом в модуль из HDL Л.Р. №1.

Теоретическая часть

Modport

Modport – конструкция языка описания аппаратуры SystemVerilog, описывающая направления сигналов внутри интерфейса в зависимости от типа используемого интерфейса (Master, Slave или др.). Данная конструкция необходима для явного определения направления сигналов для модуля, использующего интерфейс, поскольку все сигналы, находящиеся внутри SV-интерфейса без modport по-стандарту являются типом inout.

```
1  modport [identifier] (
2      input  [port_list],
3      output [port_list]
4  );
```

Рисунок 1 – Синтаксис modport (описывается внутри необходимого интерфейса)

Авторы:
Любавин Кирилл Дмитриевич
Калистратов Олег Александрович

```
interface ms_if (input clk);
  logic sready;      // Indicates if slave is ready to accept data
  logic rstn;        // Active low reset
  logic [1:0] addr;  // Address
  logic [7:0] data;  // Data

  modport slave ( input addr, data, rstn, clk,
                 output sready);

  modport master ( output addr, data,
                   input  clk, sready, rstn);
endinterface
```

Рисунок 2 – Пример определения интерфейса с использованием modport

Для назначения конкретного modport инстанцируемого модуля используется следующая конструкция (рисунок 3).

```
module d_top (ms_if tif);
    // Pass the "master" modport to master
    master  m0 (tif.master);

    // Pass the "slave" modport to slave
    slave   s0 (tif.slave);
endmodule
```

Рисунок 3 – Назначение modport интерфейса для подключаемого модуля

Struct, typedef

Конструкция struct в SystemVerilog позволяет нам создать группу из нескольких переменных (даже разного типа). На всю группу можно ссылаться как на одно целое, либо на отдельные её части можно ссылаться по имени, используя «иерархическое» обращение (как и в интерфейсах). Это очень удобно при написании RTL-кода, когда у вас есть набор сигналов, которые вам нужно передать по всему дизайну вместе, но вы хотите сохранить читаемость и доступность каждого отдельного сигнала.

Существуют структуры типа packed и unpacked. При объявлении структуры без присваивания конкретного типа она по умолчанию будет представлена как unpacked. Синтезируемой конструкцией является только лишь структура типа packed!

```
struct [struct_type] {
    [list_of_variables]
} struct_name;
```

Рисунок 4 – Синтаксис конструкции struct

```
localparam  YES = 1'b1,
            NO  = 1'b0;

struct packed {
    logic   [31:0]  fruit_id;
    bit             is_on_sale;
    byte            expiry_date;
} fruit_t;

fruit_t apple;
fruit_t orange;

always_comb apple.expiry_date   = 8'h7;
always_comb apple.fruit_id      = 32'h0000_01FCD;
always_comb apple.is_on_sale    = YES;
```

Рисунок 5 – Пример объявления структуры и использования её параметров.

В языке SystemVerilog можно создавать новые типы данных. Для этого используется конструкция typedef. В большинстве случаев мы просто используем typedef для присвоения имени объявлению типа, которое мы хотим использовать в нескольких местах вашего кода. Это полезно, так как мы можем создавать довольно сложные типы данных в SystemVerilog. Когда мы используем typedef вместо повторения сложного объявления типа, мы упрощаем наш код для понимания и поддержки.

```
typedef data_type type_name [range];
```

Рисунок 6 – Синтаксис конструкции typedef

Крайне удобным решением является объявление определённых регистров проектируемого модуля через typedef struct. Пример показан на рисунке 7.

```
typedef struct packed {
    logic   [13:0]      calculation_result;
    logic   [1:0]       expression_type;
    logic   [7:0]       value_2;
    logic   [7:0]       value_1;
} summ_register_t;
```

Рисунок 7 – Пример использования typedef для определения структуры регистра

Package

Package реализуют механизм хранения и передачи структур, методов, переменных, параметров и других конструкций HDL SystemVerilog между модулями для дальнейшего использования.

```
package registers_pkg;

    typedef struct packed {
        logic   [13:0]      calculation_result;
        logic   [1:0]       expression_type;
        logic   [7:0]       value_2;
        logic   [7:0]       value_1;
    } summ_register_t;

    enum logic {
        IDLE,
        PROCESSING,
        FINISH
    } fsm_calc_state_t;

endpackage : registers_pkg
```

Рисунок 8 – Пример объявления package с перечнем необходимых конструкций для определённого регистра

```
module register_example(
    APB3.Slave  APB3,
);

    import registers_pkg::*;

    summ_registers_t    REG1;
    summ_registers_t    REG2;

    fsm_calc_state_t    fsm_state_current;
    fsm_calc_state_t    fsm_state_next;

    /* some logic */

endmodule : register_example
```

Рисунок 9 – Подключение package внутри модуля и использование его переменных

CRC Алгоритм

Cyclic Redundancy Code (Циклически избыточный код) – Одна из разновидностей контрольной суммы, необходимая для проверки целостности данных. Алгоритм вычисления подразумевает передачу пакета данных через устройство, реализующее мат. функцию (полином). На рисунке 10 представлен пример HDL кода обработки данных для CRC-16-CCITT с использованием полинома $x^{16} + x^{12} + x^5 + 1$ (0x1021).

```
always @ (posedge clk or negedge nrst)
if (!nrst)
    crc     <= 16'h0000;
else begin
    crc[0]  <= ~(crcbitin ^ ~crc[15]);
    crc[1]  <= crc[0];
    crc[2]  <= crc[1];
    crc[3]  <= crc[2];
    crc[4]  <= crc[3];
    crc[5]  <= ~(~crc[4] ^ crcbitin ^ ~crc[15]);
    crc[6]  <= crc[5];
    crc[7]  <= crc[6];
    crc[8]  <= crc[7];
    crc[9]  <= crc[8];
    crc[10] <= crc[9];
    crc[11] <= crc[10];
    crc[12] <= ~(~crc[11] ^ crcbitin ^ ~crc[15]);
    crc[13] <= crc[12];
    crc[14] <= crc[13];
    crc[15] <= crc[14];
end
```

Рисунок 10 – Пример HDL кода вычисления последовательного CRC для CRC-16-CCITT (полином 0x1021)

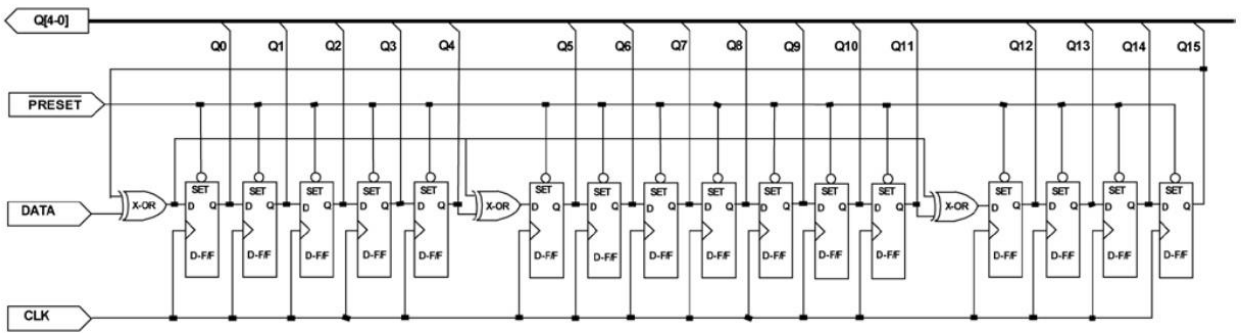

Рисунок 11 – Пример схемотехнической реализации последовательного расчёта CRC-16-CCITT

Варианты заданий

№	CRC-16 Тип	Полином	Начальное значение	RefIn	RefOut	XorOut
1	CRC-16/MAXIM	0x1021	0xFFFF	FALSE	FALSE	0x0000
2	CRC-16/ARC	0x8005	0x0000	TRUE	TRUE	0xFFFF
3	CRC-16/AUG-CCITT	0x8005	0x0000	TRUE	TRUE	0x0000
4	CRC-16/BUYPASS	0x1021	0x1D0F	FALSE	FALSE	0x0000
5	CRC-16/CDMA2000	0x8005	0x0000	FALSE	FALSE	0x0000
6	CRC-16/DDS-110	0xC867	0xFFFF	FALSE	FALSE	0x0000
7	CRC-16/DECT-R	0x8005	0x800D	FALSE	FALSE	0x0000
8	CRC-16/DECT-X	0x0589	0x0000	FALSE	FALSE	0x0001
9	CRC-16/DNP	0x0589	0x0000	FALSE	FALSE	0x0000
10	CRC-16/EN-13757	0x3D65	0x0000	TRUE	TRUE	0xFFFF
11	CRC-16/GENIBUS	0x3D65	0x0000	FALSE	FALSE	0xFFFF

- RefIn — порядок поступления битов из буфера данных: false — начиная со старшего значащего бита (MSB first), true – LSB first;
- RefOut – признак инвертирования порядка битов на выходе: true – инвертировать.
- XorOut – значение, с которым необходимо произвести XOR результата вычисления до сохранения в регистр.

Результаты вычислений можно сверить на https://crccalc.com

Если адрес не совпадает с регистровой картой, необходимо поднимать PSLVERR.

Регистровая карта должна выглядеть следующим образом (все регистры 32 бита).

Название регистра	Адрес регистра (сдвиг)	Функционал
Input Data	0x0	Исходные данные для вычисления (32 бита)
Result	0x4	Результат вычислений (16 бит)
Flags	0x8	Флаги событий. 0 бит – начать вычисление (выставляет master-устройство). 15 бит – вычисление закончено (задаётся slave-устройством).
Status	0xC	Необходим для дальнейшей отладки в Л.Р. 3

Используемая литература

1. SystemVerilog Modport. Режим доступа: https://www.chipverify.com/systemverilog/systemverilog-modport
2. CRC Calculation. Режим доступа: https://www.crccalc.com
3. Нововведения языка SystemVerilog. Режим доступа: https://habr.com/ru/post/221265/