Тествое задние на кафедру МЦСТ

Комиссаров Данил Андреевич

Март 2025

Сейчас требуется написать 2 проекта на Verilog: модель определения делимости на 3 и модуль памяти.

Из важных аспектов выделены:

1. Синтезируемость
2. Затраты аппаратных ресурсов
3. Читаемость

Опишу, как я понимаю эти пункты:

Первый видимо означает вообще работоспособность модуля, то есть компилируется ли код или же нет. Второе означает количество транзисторов или же размер на кристалле, который займет этот модуль, то есть следует писать как можно меньше переменных **reg** и использовать как можно меньше комбинационной логики, то есть сокращать количество операторов по возможности. Ну а третье, очевидно, говорит о понятности написанного кода и соответствии названий и форматов входных и выходных пинов.

Вроде более-менее понятно, перейдем к написанию модулей:

1 Определитель делимости на 3

Описать на Verilog параметризованный модуль, реализующий следующую функциональность: на выходной порт **divisibility** подаётся 1, если значение данных на входном порте **data** кратно числу 3 (в десятичной системе счисления); в остальных случаях подаётся 0.

Понятно, что придется в любом случае работать с двоичной формой представления числа, иначе было бы удобно просто сложить все цифры и определить делимость этой суммы.

Поэтому вспомним школьный факт, что число в бинарном представлении делится на 3, когда разность суммы битов на четных местах и на нечетных тоже делится на 3.

```
always @(*)
begin
    sum_even = 0; //Сумма битов на четных позициях
    sum_odd = 0;  //Сумма битов на нечетных позициях

    even_count = 0;
    odd_count = 0;

    for (i = 0; i < DATA_W; i = i + 1)
    begin
        if (i % 2 == 0)
        begin
            sum_even = sum_even + data[i]; //Здесь вычисляем соответственно сумму на четных
            even_count = even_count + 1;
        end
        else
        begin
            sum_odd = sum_odd + data[i];   //А здесь на нечетных
            odd_count = odd_count + 1;
        end
    end
    inverted_sum = ((even_count - sum_even) - (odd_count - sum_odd))%3;
    if (data[DATA_W-1] == 0 && (sum_even - sum_odd)%3==0 || data[DATA_W-1] == 1 && (inverted_sum == -1 || inverted_sum == 2))
    begin
        divisibility <= 1'b1;
    end
    else
    begin
        divisibility <= 1'b0;
    end
```

Здесь реализована проверка на делимость положительных чисел. Предлагаю добавить еще и проверку для отрицательных: для этого нужно просто перевести число из отрицательного представления в положительное, то есть убрать минус в десятичном представлении. Для этого нужно просто инвертировать все биты и добавить единицу к числу и проверить точно так же, как и предыдущее.

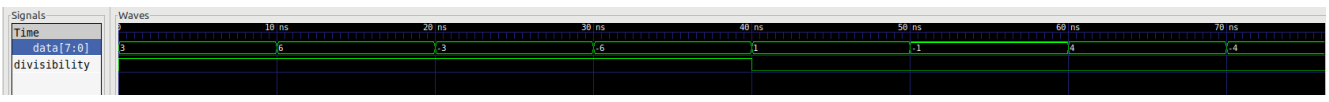
Считаю этот метод достаточно оптимальным в контексте затрат аппаратных ресурсов.

```
for (i = 0; i < DATA_W; i = i + 1)
begin
    if (i % 2 == 0)
    begin
        sum_even = sum_even + data[i]; //Здесь вычисляем соответственно сумму на четных
        even_count = even_count + 1;
    end
    else
    begin
        sum_odd = sum_odd + data[i];   //А здесь на нечетных
        odd_count = odd_count + 1;
    end
end
//debag = ((even_count - sum_even) - (odd_count - sum_odd))%3;
if (data[DATA_W-1] == 0 && (sum_even - sum_odd)%3==0 || data[DATA_W-1] == 1 && (((even_count - sum_even) - (odd_count - sum_odd)))%3==1)
begin
    divisibility <= 1'b1;
end
else
begin
    divisibility <= 1'b0;
end
```

Этим же и занимается вторая половина условия в условном операторе if.

Зачем так сложно инвертировать биты: то есть сначала подсчитывать все четные, а потом из них вычитать только те, в которых есть единица, а не, например, просто вычитать из **DATA_W /2**? Написано это с расчетом на то, что параметр **DATA_W** может быть любым, в том числе и нечетным; в этом случае не будет работать первый метод.

Вроде все очевидно, проверим на тестбенче, в котором в том числе надо написать и проверку на отрицательные числа.



Строка **data[7:0]** представлена в виде **signed decimal**.

Вот так вот реализована первая задача.

2 Память

Нужно реализовать модуль памяти с возможностью записи и считывания данных.

```
1  module memory #(
2      parameter MEM_SIZE = 6,
3      parameter DATA_W = 10,
4      parameter ADDR_SIZE = $clog2(MEM_SIZE)
5  )(
6      input wire clk,
7      input wire write_flag,
8      input wire [DATA_W-1:0] data_in,
9      input wire [ADDR_SIZE-1:0] addr_w,
10     input wire read_flag,
11     input wire [ADDR_SIZE-1:0] addr_r,
12     output reg [DATA_W-1:0] data_out
13 );
14
15
16 reg [DATA_W-1:0] memory [MEM_SIZE-1:0];
17
18 always @(posedge clk) begin
19     if (write_flag) memory[addr_w] <= data_in;
20 end
21
22 always @(posedge clk) begin
23     if (read_flag) data_out <= memory[addr_r];
24 end
25
26 endmodule
```

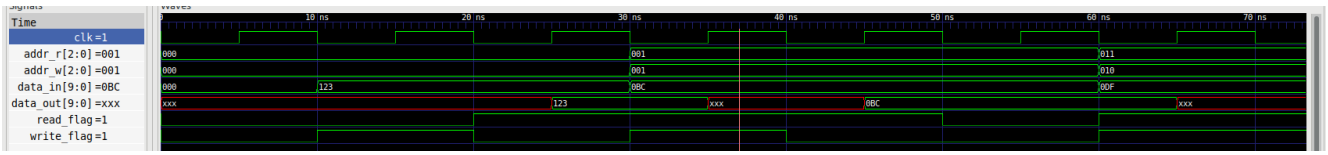
Вся суть модуля только лишь в двумерном массиве регистров, которые и являются ячейками памяти. Дольше описывали параметры и порты, чем код писали))

Параметры **MEM_SIZE** и **DATA_W** описываются как в ТЗ, помимо этого можно добавить параметр **ADDR_SIZE** который означает, сколько нужно выделить битов для описания адреса ячейки памяти. Так как нет верхней границы по предыдущим параметрам, считаю уместным его добавить.

Часто встречал в интернете, когда писал свой процессор на Verilog, следующие особенности в написании ячеек памяти:

```
reg [DATA_W-1:0] memory [0:MEM_SIZE-1];// Такую запись
reg [DATA_W-1:0] memory [MEM_SIZE-1:0];//Вместо такой
```

Утверждается, якобы это реализует принцип **LITTLE-ENDIAN** вместо **BIG-ENDIAN**, но, признаюсь честно, не могу в полной степени оценить всю суть этого шага, так как, при замене этих строк, ничего не меняется при проверке в тестбенче, поэтому оставлю тот вариант, в котором уверен.



Методом тщательного взгляда можно убедиться, что модуль работает корректно. В некоторых местах показывается сигнал *xxx*, это происходит из-за того, что нет значения при инициализации памяти. Можно добавить, например, сигнал сброса к модулю, но этого нет в ТЗ, оставим значит как есть.

Задание можно считать выполненным. Было интересно.

Спасибо за прочтение.)