仿真与测试

唐紫健

2020.3.30

一．基本 testbench 编写

1. 示例

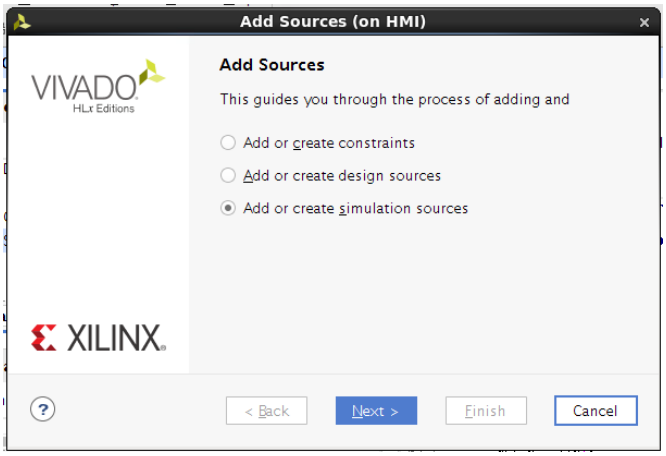
以简单 4bit 时序逻辑加法器为例，该 testbench 随机生成 2 个 4bit 加数，送入加法器后，比较加法器结果与实际期望的结果，并打印判断正误信息。

编写加法器的 Verilog 代码：

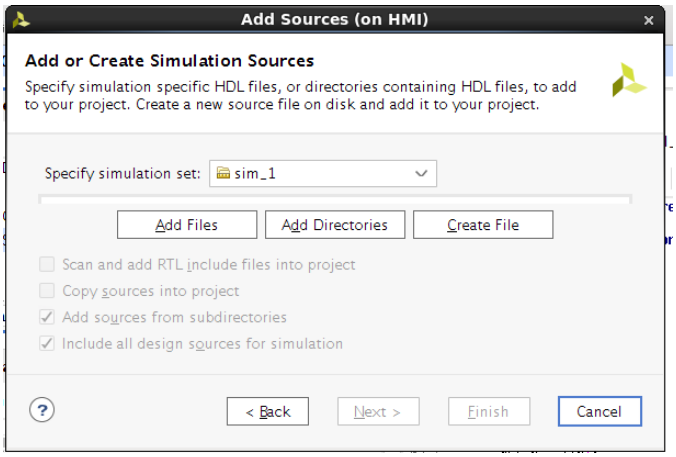
adder.v

```
module adder (
  input       clk    , // Clock
  input       rst_n   ,
  input  [3:0] data_in1, // Asynchronous reset active low
  input  [3:0] data_in2,
  output [3:0] data_out
);

  reg [3:0] data_out_q;

  assign data_out = data_out_q;

  always @(posedge clk or negedge rst_n) begin : proc_data_out
    if(~rst_n) begin
      data_out_q <= 0;
    end else begin
      data_out_q <= data_in1+data_in2;
    end
  end

endmodule
```

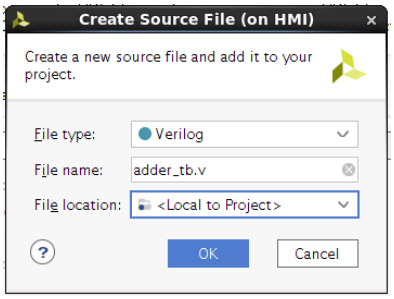
在 Vivado 中添加仿真文件



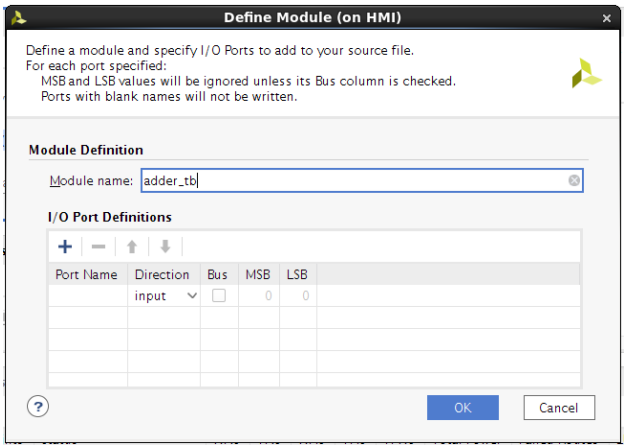
选择仿真文件的默认 fileset－sim_1，选择 create file （如果仿真文件已编写好，也可以选择 add files 将已有文件添加进 sim_1 即可）。



将仿真 testbench 文件命名为 adder_tb.v



点击 OK 后，回到图*界面选择 finish。出现如下界面，与之前添加 Verilog 源文件类似。因为 testbench 文件无需外部接口，此处选择不添加 I/O Port.



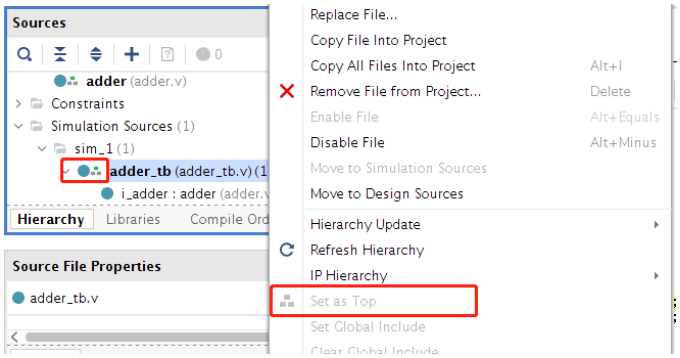
在 testbench 中添加如下代码，保存

adder_tb.v

```
`timescale 1ns/1ps
`define PERIOD 10

module adder_tb_simple();

  reg clk  ;
  reg rst_n;

  reg  [3:0] data_in1;
  reg  [3:0] data_in2;
  wire [3:0] data_out;

  // Generate the clock
  initial begin
    forever
      #(`PERIOD/2) clk = ~clk;
  end

  // Adder Instance
  adder i_adder (
    .clk     (clk     ),
    .rst_n   (rst_n   ),
    .data_in1(data_in1),
    .data_in2(data_in2),
    .data_out(data_out)
  );

  initial begin
    // Initialization
    rst_n = 1'b1;
```

```
    clk = 1'b0;

    data_in1 = 4'b0;
    data_in2 = 4'b0;

    // Reset the DUT
    #(`PERIOD*5)
      rst_n = 1'b0;
    #(`PERIOD*5)
      rst_n = 1'b1;

    #(`PERIOD)
      data_in1 = 4'b1110;
    data_in2 = 4'b0010;

    #(`PERIOD)
      data_in2 = 4'b0100;

    #(`PERIOD)
      data_in1 = 4'b0001;

    // ......

    #(`PERIOD*2);
    $finish;
  end

endmodule
```

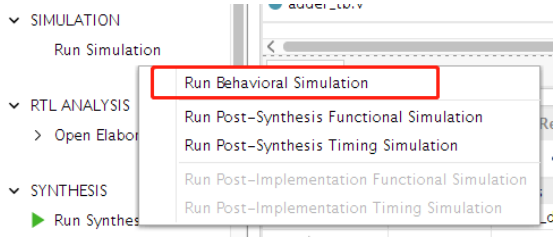
此时，文件结构如下



注意 adder_tb.v 文件前的符号，表明它是顶层仿真文件。如果没有该符号，请在文件处右键选择 set as top。

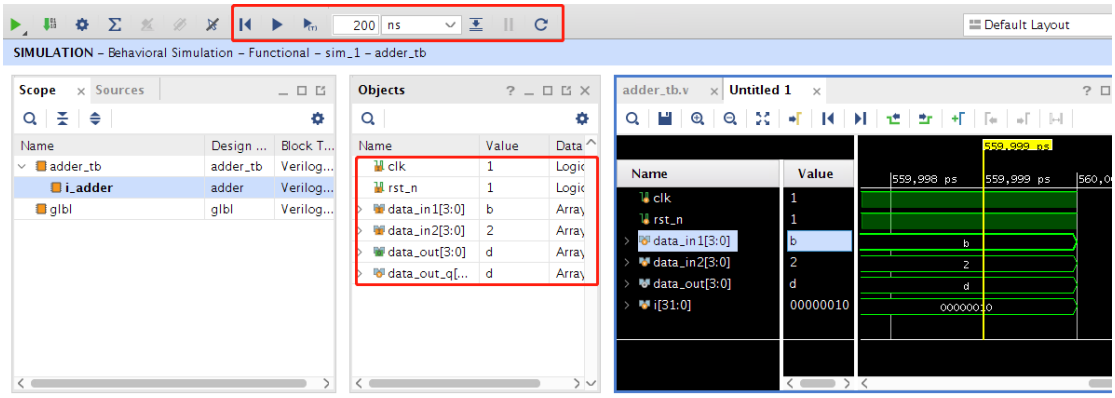
点击 run simulation，此时还未完成 synthesis 和 implementation，我们称之为行为级仿

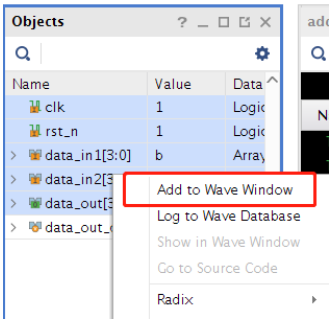
真（前仿），选择 run behavioral simulation。进入仿真界面

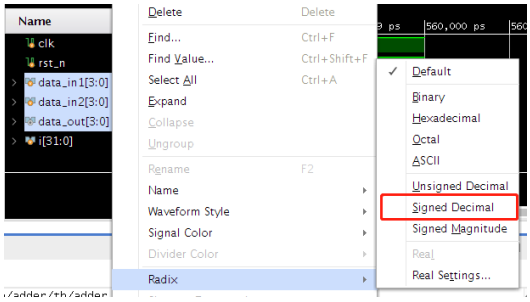


（注：若仿真 testbench 文件编写语法错误，则无法进入仿真界面，请查看错误并修正。）

上方为仿真控制按钮。在下方 Scope 中选择要查看的 module，在 Objects 中选择要查看的该 module 中的信号，右键选择将其添加进波形图像，点击上方 run 按钮即可开始仿真。可以更改信号的表示方式，例如改为有符号十进制数，便于我们查看。







（注 1：如果 testbench 中没有规定仿真结束时间，run all 会一直运行，需手动结束。可以使用 run for ** nm/um 按钮，设置仿真时间。或者在 testbench 中指定仿真结束时间（见后方

$finish $stop 内容）。
注 2：可使用快捷键 O 和 I 控制波形时间轴缩放，快速定位到需要查看的时间。）

2. 说明：

tb 文件使用的由于不对应真实硬件，因此不需要可综合。可使用#延时，在任意时刻对 module 端口信号，以及 module 内部寄存器赋值或者读取。

（1）仿真时间单位和时间精度

`timescale 1ns/1ps 定义仿真时间单位为 1ns，精度为 1ps。该预编译指令，对于前仿影响不大，但在后仿加入延时信息后，则对结果可能有影响。仿真精度应该比实际使用的时钟周期低数个量级。

（2）时钟设置

仿真时钟应与开发板上的实际时钟一致。以 Ego-1 开发板为例，时钟频率为 100MHz，因此使用

`define PERIOD 10

单位为`timescale 中设置的 ns，因此时钟频率为 100MHz，与实际时钟对应。使用 forever 语句，生成时钟

（注：若 DUT 不能够满足 10ns 的时钟约束，可以编写分频单元，加入 DUT 中。或者使用 Xilinx 的时钟 IP。）

（3）激励生成

在 initial 中生成激励。首先将 rst_n 拉低，复位 DUT。之后在确定的延时后，赋值输入信号即可。本示例中使用了 task 任务，可重复调用。

（4）testbench 结果查看

仿真结果可以在波形中手动查看检验，对于 debug 阶段比较有效。在 DUT 代码基本完善后，可以在 testbench 中写明检验语句，通过命令行输出来查看结果，见后方$display、$monitor。

（5）task 和 function

类似 C 语言的函数，数据传递到 task 或者 function 中，完成处理，并返回结果。必须使用数据输入和输出来专门调用，而不仅仅是将其连接到常规网表。可以多次调用，减少重复代码片段。function 与 task 的区别，在于 function 不能驱动多个输出，不能包含延迟。

可以参考 http://www.asic-world.com/verilog/task_func1.html

（6）for 循环

可综合代码中通常不使用 for 语句。但在 tb 中，可用来重复生成某一激励，使用方法与 C 语言类似。循环累计变量通常生命为 integer 类型。

二、常用系统函数

（1）打印与文件输出

类似 C 语言的 printf 函数，$display 和 $write 用于用于命令行打印信息。$display 输出后自动换行，$write 则不换行。常用格式控制及特殊字符如下

输出格式	说明	输出格式	说明
%h 或%H	以十六进制形式输出	%d 或%D	以十进制形式输出

%o 或%O	以八进制形式输出	%b 或%B	以二进制形式输出
%c 或%C	以 ASCII 码形式输出	%v 或%V	输出网格型数据信号轻度
%m 或%M	输出等级层次名字	%s 或%S	以字符串的形式输出
%e 或%E	以指数形式输出实型数	%f 或%F	以十进制的形式输出实型数
%g 或%G	以指数或者十进制数输出实型数，但是 无论何种格式都以较短的结果输出	%t 或%T	输出当前的时间格式

换码序列	功能	换码序列	功能
\n	换行	\"	双引号字符"
\t	横向跳格	\o	1-3 位八进制数代表的字符
\\	反斜杠字符	%%	百分符号%

https://www.cnblogs.com/SYoong/p/5897150.html

可以使用 $fopen 打开文件，使用$fdisplay 和 $fwrite 输出信息至文件，使用$fclose 关闭文件，将信息输出至文件以备后续处理。从文件读入数据可使用$fread 命令。

（2）$time 和$realtime

用于输出当前仿真时间，其中$time 返回类型为 64 位整数，$realtime 返回类型为实数。

（3）$monitor

$monitor 用于监控输出列表中的表达式或变量值的功能，仅当列表中变量的值发生变化时，才输出其结果。可结合$time 使用。

$monitor($time,,"mon1=%b mon2=%b",mon1,mon2) ；//可输出 mon1 或者 mon2 变化时，两个变量的值。

（4）$readmemb 和$readmemh

可从文件中将数据读入存储器（前面是否有介绍存储器？）中，其中$readmemb 读入 2 进制数，$readmemh 读入 16 进制数。调用格式为

$readmemb("<数据文件名>"，<存储器名>，<起始地址>，<结束地址>)； //其中起止地址为可选项。

（5）$finish 与$stop

用于结束与停止仿真。$finish 会结束仿真，而$stop 则会暂停仿真，可以在 GUI 界面选择继续仿真。建议使用该系统函数手动限制仿真时间长度。

（6）$random

$random 用于生成随机数，对于 ALU 测试比较常用。

```
reg [31:0] rand;
rand=$random%10;                    //生成-9~9 之间的随机数
rand={$random}%10;                  //生成 0~9 之间的随机数
```

（1）-（6）我们使用下面的较为完整的 tb 文件作为示例

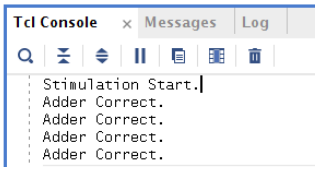
adder_tb.v

```
`timescale 1ns/1ps
`define PERIOD 10
```

```
module adder_tb();

  reg clk  ;
  reg rst_n;

  reg  [3:0] data_in1;
  reg  [3:0] data_in2;
  wire [3:0] data_out;

  // Loop Integer
  integer i;

  // Generate the clock
  initial begin
    forever
      #(`PERIOD/2) clk = ~clk;
  end

  // Adder Instance
  adder i_adder (
    .clk     (clk     ),
    .rst_n   (rst_n   ),
    .data_in1(data_in1),
    .data_in2(data_in2),
    .data_out(data_out)
  );

  // Generate Input
  task input_ramdomize();
    begin
      data_in1 = $random()%8;
      data_in2 = $random()%8;
    end
  endtask

  // Check Output
  task output_check();
    begin
      if(data_out == data_in1+data_in2)
        $display("Adder Correct.");
      else
        $display("Adder Error!",);
    end
  endtask
```

```
  initial begin
    // Initialization
    rst_n = 1'b1;
    clk = 1'b0;

    // Reset the DUT
    #(`PERIOD*5)
      rst_n = 1'b0;
    #(`PERIOD*5)
      rst_n = 1'b1;

    $display("Stimulation Start.");

    // Test Loop
    for (i = 0; i < 16; i=i+1) begin
      input_ramdomize();
      #(`PERIOD);
      output_check();
    end
    #(`PERIOD*2);
    $finish;
  end

endmodule
```

运行后，可以在 Tcl Console 出查看输出信息，该结果表面加法器功能正确。



（7）宏定义`define

与 C 语言宏定义类似，建议将宏定义大写，与普通变量区别。

注：宏定义只是简单字符串替换，使用不当时可能替换后的语句出现语法问题。

（8）条件编译命令`ifdef `else `endif

与 C 语言类似。通常与宏定义结合使用，可提高代码重用。

（9）`include 文件包含

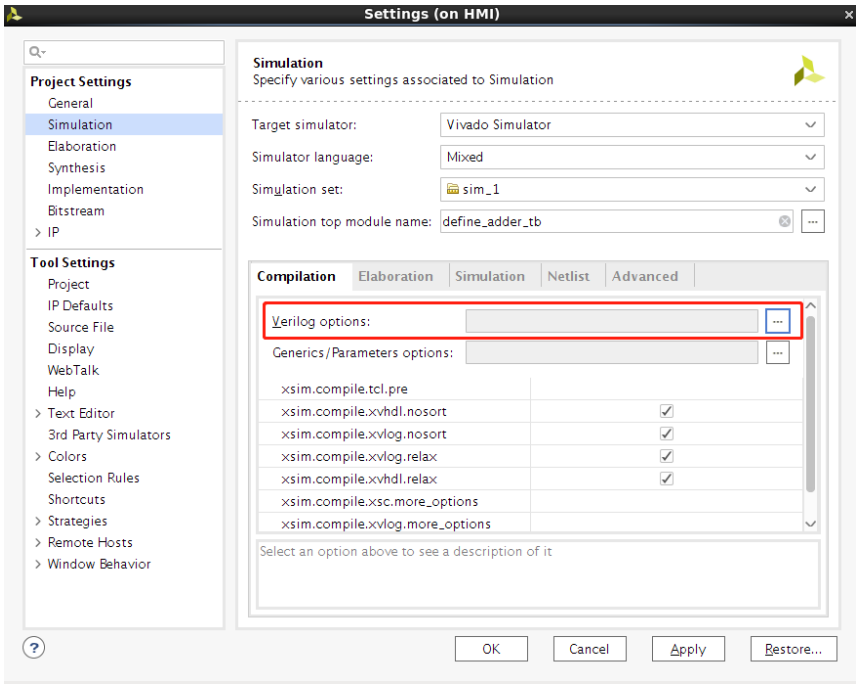
通常我们可以将宏定义写在一个文件中，类似 C 语言中头文件，在 tb 或者 dut 代码中，使用`include 将其包含。注意，包含的文件位置需要在 Vivado 中指定包含目录。

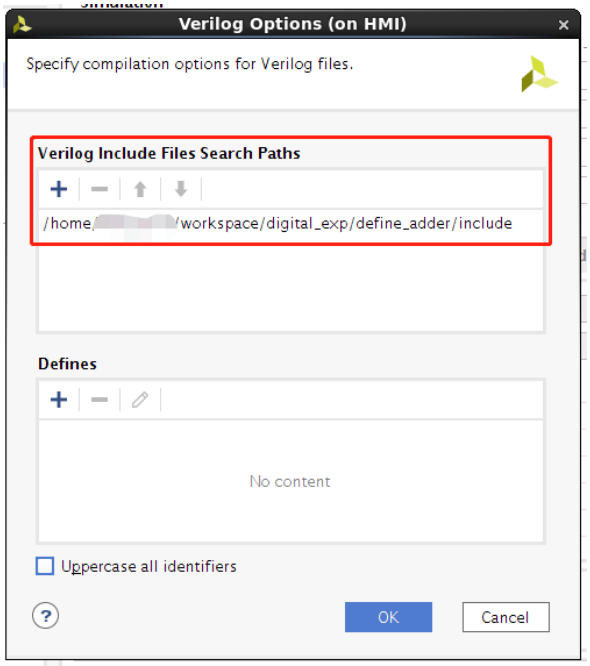
（7）、（8）、（9）我们用一些的例子介绍

例如，我们有如下代码片段，当定义了宏 FULL_ADDER 时，则编译全加器，否则编译半加器。将宏定义放在新文件 define.v 中，

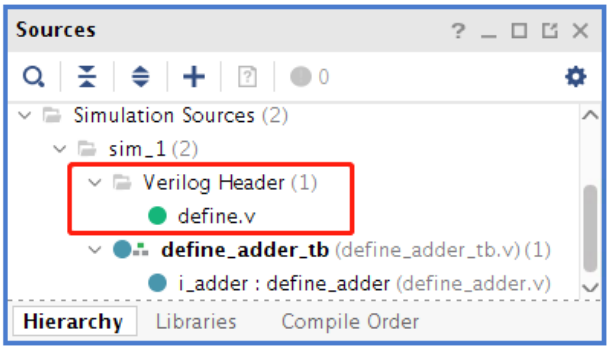
define.v
`define FULL_ADDER

vivado 中设置包含路径





修改完成后，可以看见新增 Verilog Header 文件



修改 dut 和 tb 代码条件编译。

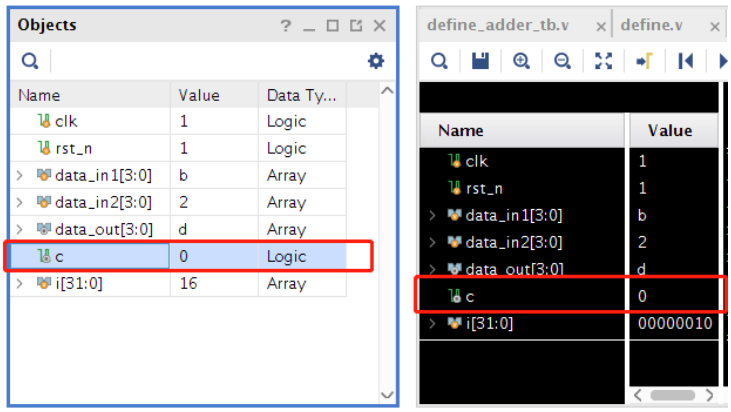
define_adder.v

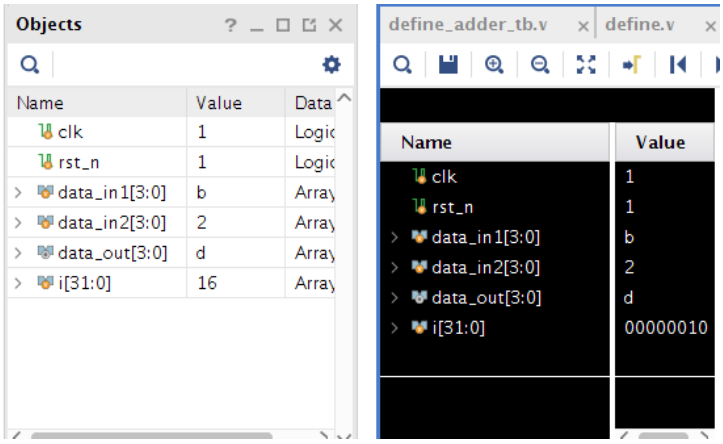
```
`include "define.v"
module define_adder (
  input       clk    , // Clock
  input       rst_n   ,
  input  [3:0] data_in1, // Asynchronous reset active low
  input  [3:0] data_in2,
  `ifdef FULL_ADDER
  output c,
  `endif
  output [3:0] data_out
);

  `ifdef FULL_ADDER
    reg [4:0] data_out_q;
    assign c = data_out_q[4];
  `else
    reg [3:0] data_out_q;
  `endif

  assign data_out = data_out_q[3:0];

  always @(posedge clk or negedge rst_n) begin : proc_data_out
    if(~rst_n) begin
      data_out_q <= 0;
    end else begin
      data_out_q <= data_in1+data_in2;
    end
  end

endmodule
```

define_adder_tb.v

```
`timescale 1ns/1ps
`define PERIOD 10

`include "define.v"

module define_adder_tb();

  reg clk  ;
  reg rst_n;

  reg  [3:0] data_in1;
  reg  [3:0] data_in2;
  wire [3:0] data_out;
  `ifdef FULL_ADDER
  wire c;
  `endif

  // ......

  // Adder Instance
  define_adder i_adder (
    .clk     (clk    ),
    .rst_n   (rst_n  ),
    .data_in1(data_in1),
    .data_in2(data_in2),
    `ifdef FULL_ADDER
    .c(c),
    `endif
    .data_out(data_out)
  );

  // ......

endmodule
```

define.v 分别定义或者不定义宏 FULL_ADDER，分别仿真。得到如下结果





（注：采用 include 读入的包含文件，在 synthesis 和 implementation 时需要分别设置包含路径。建议把多个管理模块中用到的宏统一存放在一个包含文件中。）

又例如，对于相同或相似的 tb，我们可以在不同的宏定义下，编译生成单周期处理器、多周期处理器，或是流水线处理器

define_tb2.v

```
module name ();

    // ......
    `ifdef MULT_PRD
        `ifdef PPLN
            pipeline_cpu cpu_i(
                .clk(clk),
                .rst_n(rst_n)
                // .....
            );
        `else
            mult_simple_cpu cpu_i(
                .clk(clk),
                .rst_n(rst_n)
                // .....
```

```
            );
        `endif
    `else
        single_cpu cpu_i(
            .clk(clk),
            .rst_n(rst_n)
            // ......
        );
    `endif
    // ......

endmodule
```

三、测试与验证基础

（1）

黑盒测试（black-box testing），也称黑箱测试，测试 DUT 的功能，而不是其内部结构或运作。测试者不需具备应用程序的代码、内部结构和编程语言的专门知识。测试者只需知道什么是系统应该做的事，即当键入一个特定的输入，可得到一定的输出。测试者选择有效输入和无效输入来验证是否正确的输出

白盒测试（white-box testing，又称透明盒测试 glass box testing、结构测试 structural testing 等），测试应用程序的内部结构或运作，而不是测试应用程序的功能（即黑盒测试）。在白盒测试时，以编程语言的角度来设计测试案例。测试者输入数据验证数据流在程序中的流动路径，并确定适当的输出，类似测试电路中的节点。

(Wikipedia: https://zh.wikipedia.org/wiki/%E8%BD%AF%E4%BB%B6%E6%B5%8B%E8%AF%95#%E9%BB%91%E7%9B%92%E6%B5%8B%E8%AF%95)

（2）

代码覆盖率测试一般包括行覆盖，条件覆盖，FSM 覆盖，翻转覆盖率等。例如：

1. RTL 中的每一行代码是否经过了仿真？
2. FSM 的每个状态是否经过了仿真？
3. if 条件语句的每种情况是否经过了仿真？
4. case 语句的每一种情况是否经过了仿真？

（https://www.cnblogs.com/dpc525/p/5071841.html）

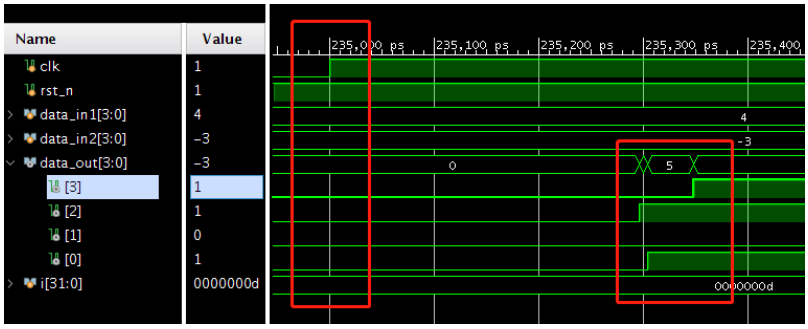
（3）

UVM（Universal Verification Methodology）通用验证方法学。大型的设计中，验证和设计几乎同等重要。随着集成电路集成度、系统复杂度、以及流片成本的提高，需要大量的工作对芯片进行完整可靠的验证。验证方法学应运而生，目前主流的 UVM 验证方法学基于 SystemVerilog。SystemVerilog 是一种 Verilog 的扩展集，它具有所有面向对象语言的特性：封装、继承和多态，同时还为验证提供了一些独有的特性，如约束（constraint）、功能覆盖

率（functional coverage）。同时提供了接口，可以把 C/C++的函数导入 SystemVerilog 代码中直接调用。
（详情可以参考 《UMV 实战：卷 I》 – 张强，机械工业出版社，2014）

四、前仿与后仿

添加约束文件，并进行 synthesis 和 implementation 后，Vivado 会用新生成的网表代替行为级 RTL 代码以及可能用到的 Xilinx IP，并加入延时信息。用生成的新网表做的仿真，可以统称为后仿真。由于延时的加入，要求此时的结果满足.xdc 文件中的时序约束条件。除了 STA 中的延时路径分析，我们可以通过后仿真波形，直观的看到实际延时对于性能的影响，并加以修正或优化。



以 4bit 加法器为例，可以看到后仿真的波形有两个不同之处。一是，输出信号不再对齐时钟上升沿，而是有加法器的延时。二是由于输出每一 bit 位置延时不同，导致波形中出现毛刺。