数字电路课内实验

薛丞博 南方 马越

2025 年 4 月

数字电路课内实验

编　写　单　位　北京理工大学

学　院　名　称　集成电路与电子学院

作　者　姓　名　薛丞博 南方 马越

编　写　日　期　2025 年 4 月

表 0.1　版本更新说明

版本号	修改内容
1.0.0	初次发布
1.0.1~1.0.5	修正 FPGA 型号；增加 TODOlist
1.0.6	前言部分增加对安装路径、工程路径等的具体要求 补充第四章第五章实验描述，明确实验要求 修改第三章实验中 SW0 为 SW_0，以防止误解 修改代码字体为 Monaco 修改水印 上线所有在线检查系统

目 录

第 1 章　硬件平台介绍、软件安装及环境配置

1.1　写在最前面

在阅读本文时，最好已经学过或正在进行数字电路这门课程的学习，在每次遇到本文的代码片段时，请参考任一本 Verilog 语法书籍了解该片段涉及到的语法规则并仔细阅读本文提供的代码，Verilog 作为一个硬件描述语言，大多数情况下不需要在代码构筑上进行创新。

本实验涉及到的都是数字设计中最基本的概念与底层逻辑，希望同学们在阅读时多查阅相关资料，搞清楚数字电路的概念以及发展的原因与成果。

如果你有 FPGA 开发的基础，可以直接跳到第六章。完成其中一个附加题并通过 ppt 做一个简要陈述可以得到全部的实验分数。

建议在打开提供的工程前，先完整的阅读整个讲义并跟随文中提供的代码走完流程。会更好的帮助你理解。

参考 Verilog 资料：https://www.runoob.com/w3cnote/verilog-tutorial.html 参考硬件设计书籍：见乐学。

在乐学上，给出了今年实验的 Demo 程序，在阅读完第二章后，可以学会如何下载该程序至开发板。作用如下：1. 验证硬件功能是否正常。2. 可以作为提交作业的功能参考。但不要提交该程序。

在下载程序后，通过拨动小拨码开关 sw[7:0]（8 位合并）的第 6、5、4 位选择对应的实验，支持实验 2、3、4 的功能演示。实验 5 的演示功能总是开启的。

1.2　硬件平台介绍

本实验采用的硬件平台为 EGO1 开发板，其基于 Xilinx 公司开发的 Artix 7 系列 FPGA 设计。关于板卡的硬件资源可以参考 板卡介绍。在这个实验中可以不过分关注硬件平台的性能，学会阅读原理图即可。

本实验会用到板卡的 LED 灯、拨码开关，按键等基本资源。

1.3 EDA 简介

本实验中涉及的 EDA 软件主要有两个，一个是 Xilinx 公司的 Vivado FPGA 设计软件，其主要作用是将用户输入的设计文件转化成硬件可以执行的二进制文件。类似于 C 语言将.c/.h 文件编译成可执行文件的过程，但原理不尽相同。这个转化过程通常被称为“综合”，但狭义上综合仅是该软件编译过程的第一步，后面还包括“时序约束”、“实现”、“生成比特流”等步骤。在本系列实验中，会着重描述最基本的步骤“设计输入、综合、实现、生成 bit 流”，关于时序约束请在后续学习阶段中参考其他教材。需要注意的是，时序约束在复杂数字系统的设计中占比 50% 以上。

另外一个软件是 Mentor 公司的 Modelsim 仿真软件，其主要作用是完成用户输入的仿真，在通常的数字电路设计中，仿真在包括前仿真与后仿真，前仿真亦可细分成 RTL 仿真（功能仿真）以及门级仿真（网表级），这些不同的仿真会覆盖整个数字电路/芯片设计的不同阶段，实现对电路功能/性能的验证。

1.4 软件安装

在下载完软件后（为减少软件体积，本实验默认使用的是 Vivado 2017.4 版本和 Modelsim 2020.4 版本，其他版本请自行解决兼容问题），双击 setup 进行安装。与大多数 EDA 软件没有不同，选择默认选项即可完成安装。为了减少所需硬盘空间，可以在安装 Vivado 时只选择安装 Artix 7 系列支持。

为什么不同系列的硬件需要不同软件来支持？跟手机处理器一样，为了满足不同用户的需求，FPGA 也分为高中低端的产品，其中包含有不同的资源。

在本实验中，计算机名，用户名，安装路径，工程路径均只允许包含字母 A-Za-z 数字 0-9 以及下划线 _ 并以字母作为第一个字符。

受限于 Microsoft Windows 系统内部逻辑限制，文件名（含路径）总长不得超过 255 个字符。在 linux 下无此限制。

第 2 章　组合逻辑实验

实验目的：熟悉 Verilog 语法及仿真过程，掌握例化，testbench 概念，学习 EDA 工具的安装与使用。

2.1　二输入与门

这个实验将带领大家完成一个二输入与门的设计，并生成文件置入 FPGA 中执行，观察现象。

数字电路开发通常分为设计输入、电路实现、仿真以及实物验证四个阶段。请暂时忘掉看过的 Verilog 语法，跟随本文的进度。

2.2　设计输入

在这个阶段中，需要明确用户设计的需求，本实验的目的是完成一个二输入与门，选择的输入为开发板上的 SW0（开关 0）及 SW1（开关 1），输出的结果为 LED0。

2.3　电路实现

2.3.1　建立工程

打开 Vivado，新建一个工程，使用默认选项即可。注意，在选择器件时，应选择与开发板一致的 xc7a35tcsg324-1。

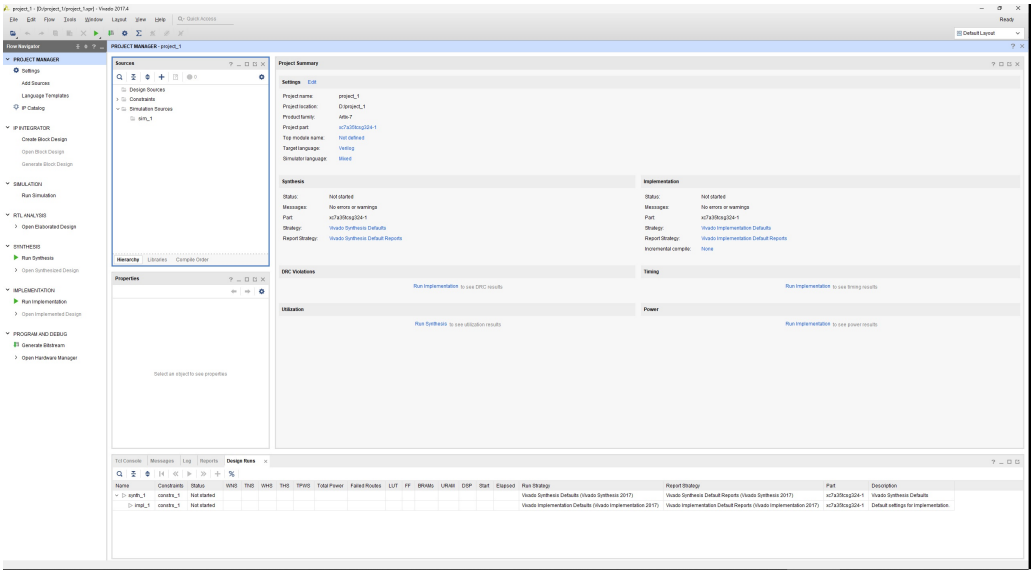


图 2.1　创建工程

2.3.2　创建文件并输入

点击 Add Sources，创建一个 Verilog 文件，命名为 and2.v，并打开该文件。

数字电路设计的理念为模块化思想，无论使用 Verilog 亦或 VHDL 都是如此，在本实验中我们选择流传行稍广的 Verilog 作为例子。

Verilog 中的设计单元为 module，我们设计一个二输入与门，这个与门包含两个输入与一个输出，因此在端口定义中，我们写下两个 input a,b 与一个 output o。

```
1    module and2
2        (
3        input a,
4        input b,
5        output o
6        );
7    endmodule
```

接下来完成二输入与门的功能，

```
1    module and2
2        (
3         input a,
4         input b,
```

```
5         output o
6         );
7         assign o = a & b;
8     endmodule
```

至此，我们完成了二输入与门电路实现的过程。

2.3.3 仿真

通常在 FPGA 开发时，会使用 Modelsim 进行仿真。当然，Vivado 自带仿真器的功能，但本实验中仍然选择 Modelsim 的原因是作为一个第三方仿真软件，其可以适配不同厂商 FPGA 的仿真过程，在低复杂度 FPGA 应用开发时，学会 Modelsim 已足够应付所有场景。

可以选择使用 Vivado 内置的仿真器并跳过编译 Modelsim 的部分。参考这个。

首先介绍使用 Vivado 调用 Modelsim 的一般流程。首先在 Vivado 环境中配置三方软件，打开 Tools->Settings->3rd Party Simulators，上方填写 Modelsim 的安装路径，下方的 Default Compiled Library Paths 中不应为空。

为了填写这个 Default Compiled Library Paths，我们暂且关闭这个窗口，打开 Tools-> Compile Simulation Libraries。填写 Compiled library location 以及 Simulator executable path，并完成编译。（此过程大概需要持续 40 分钟）

编译好后，回到图2.2的界面，填写 Modelsim Default Compiled Library Paths。

跳过 Modelsim 的从这里续上。

为了进行仿真，另外一个必须的文件是激励文件，通常被称为 Testbench/tb 文件。其作用是给我们设计的电路一个输入，以便于我们观察电路的输出是否与设计一致。我们新建一个文件，命名为 tb_and2.v。为了符合 Verilog 的语法，该文件仍然需要存在 Module 的引脚输入的括号，但作为一个 tb，它不需要存在输入输出，因此我们将括号留空即可。

```
module tb_and2
    (
     );

endmodule
```

接下来，我们做一个 Verilog 设计中非常常见的工作：例化（instance），该步骤的作用是将我们前面设计的 and2 模块置入 tb 中，为了帮助理解，我们在 tb 中例化 4 次。

```
module tb_and2(

    );
```

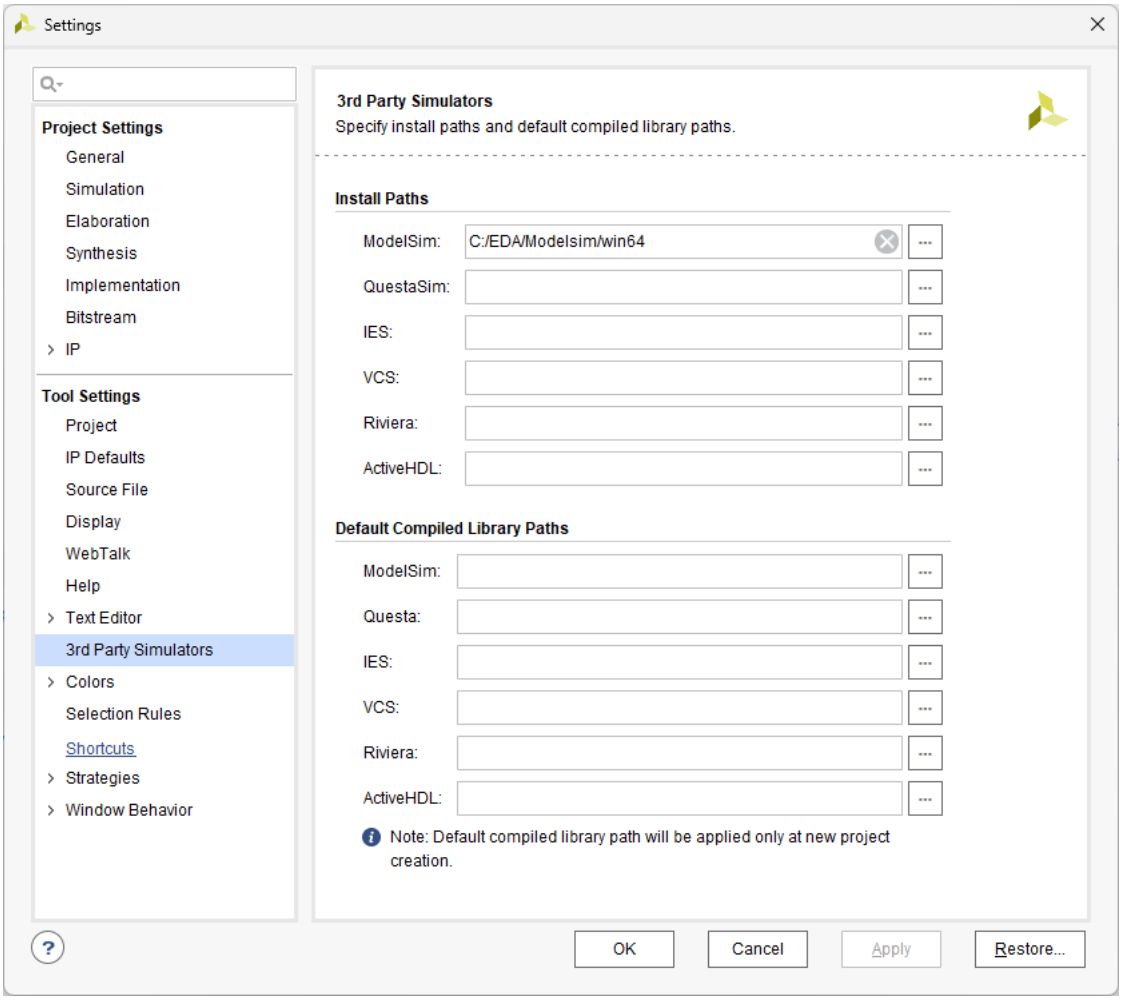


图 2.2　第三方仿真环境

```
4
5     wire [3:0] o;
6     wire [3:0] a;
7     wire [3:0] b;
8
9     and2 uut0 //注意这里是名字
10    (
11    // Outputs
12    .o                              (o[0]),
13    // Inputs
14    .a                              (a[0]),
15    .b                              (b[0]));
16
17    and2 uut1
18    (
19    // Outputs
20    .o                              (o[1]),
21    // Inputs
22    .a                              (a[1]),
23    .b                              (b[1]));
24
25    and2 uut2
26    (
27    // Outputs
28    .o                              (o[2]),
29    // Inputs
30    .a                              (a[2]),
31    .b                              (b[2]));
32
33    and2 uut3
34    (
35    // Outputs
36    .o                              (o[3]),
37    // Inputs
38    .a                              (a[3]),
```

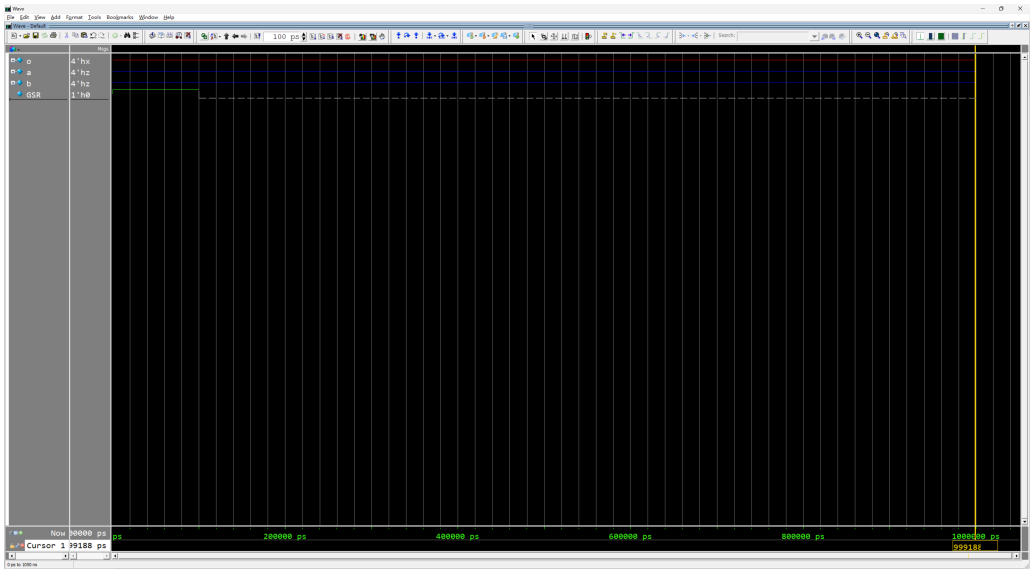


图 2.3　仿真截图

```
39      .b                                  (b[3]));
40
41  endmodule
```

这里想强调的概念是，我们设计了一个 and2 的单元，但是在实际应用时，可以重复使用，且每一个例化都是唯一的存在。假象我们做的是红绿灯，每个路口摆放的红绿灯类似于例化的过程，但每个红绿灯的颜色变化可以不尽相同。在后面我们会进一步强调这个概念。

目前让我们回到仿真这个过程上来，右击左侧的 Run Simulation，在设置中修改 Target Simulator 为 Modelsim, 注意这里 Simulation 顶层名字应为 tb_and2。

回到主界面，点击 Run Simulation，如图2.3。

注意，目前我们在 tb 中并未对输入赋值，因此输入的结果为默认的高阻态（蓝色），输出的结果为红色的未定义（undefined）。在仿真中出现红色还有另外一种可能是既赋值成了 0，又赋值成了 1。

下面我们修改一下 tb 文件，给电路添加一个激励。由于我们需要给输入赋值，**根据 verilog 的语法，我们需要把输入定义成 reg 型变量。**

我们在 tb_and2.v 中增加一段代码，在 0 时刻给所有输入赋值成 0，并在后面修改他们的值，观察输出的变化。首先，verilog 的仿真时刻是由 ‘timescale 1ns / 1ps 关键字控制的，前者表明其时间单位，后者表明变化的精度。**Verilog 是描述硬件电路而不是设计硬件电路的语言**。可以理解成仿真软件是按 Verilog 语法，在每个 1ps 时刻计算一次各输出的结果，并在 1ns 时刻更新波形，这个波形恰好与设计出的电路的功能一致（这个说法可能不太精确，但请务必了解这个概念）。在本实验中不会涉及到别的 timescale，因此在现在的学习阶段可以跳过这个问题。事实上，在绝大多数开发阶段，也不太会用到 timescale 本身的概念，但 **Verilog 是描述硬件电路而不是设计硬件电路的语言**，错误的仿真方式会使得代码与实际电路功能无法对应。后面我们会用一个例子来说明这个问题。回到代码中，先添加一个在 0 时刻的值，接着在不同时刻添加一些输入的变化。

```
module tb_and2(

    );
    wire [3:0] o;
    reg [3:0]  a;
    reg [3:0]  b;

    initial begin
        a = 4'h0;
        b = 4'h0;
        #100;
        a[0] = 1'b1;

        a[1] = 1'b1;
        b[0] = 1'b1;
        #100;
        b[1] = 1'b1;
        #300;
        a[3:2] = 2'b11;
        b[3:2] = 2'h2;
        #100;
        b[2] = 1'b1;
```

```
23      end
24
25      and2 uut0
26      (
27       // Outputs
```

注意这里我们不要关闭 Modelsim，切换至 Library 选项卡，找到 tb_and2（在 xil_defaultlib 下），右键选择 Recompile。

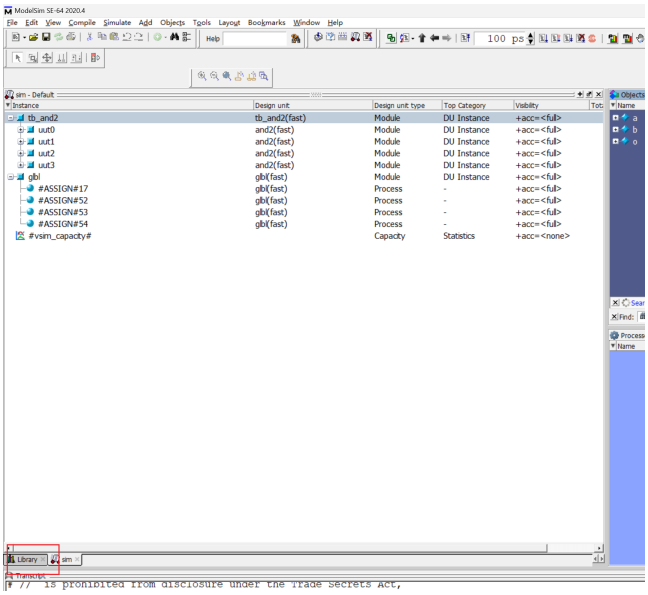


图 2.4　切换 Library

在命令行中输入 restart 并回车（或点击图形界面上的 Restart），重新开始仿真。在命令行中输入 run 1 us 回车（或通过图形界面）。

作业 1：对显示的波形进行截图，说明波形变化与 tb 中激励变化的对应关系。

2.3.4　实物验证

回到 Vivado 中，首先，tb_and2.v 是只在仿真中使用的文件，因此修改下该文件的属性，使其只在仿真中使用。

接着，我们将 a[3:0] 与 b[3:0] 连接到板卡的开关上，这里涉及的内容是约束文件的编写。通常的做法会直接编写约束文件，这里为了让大家了解基本概念，我们选择用图形界面的方式来打开约束文件。约束文件的后缀是 xdc。

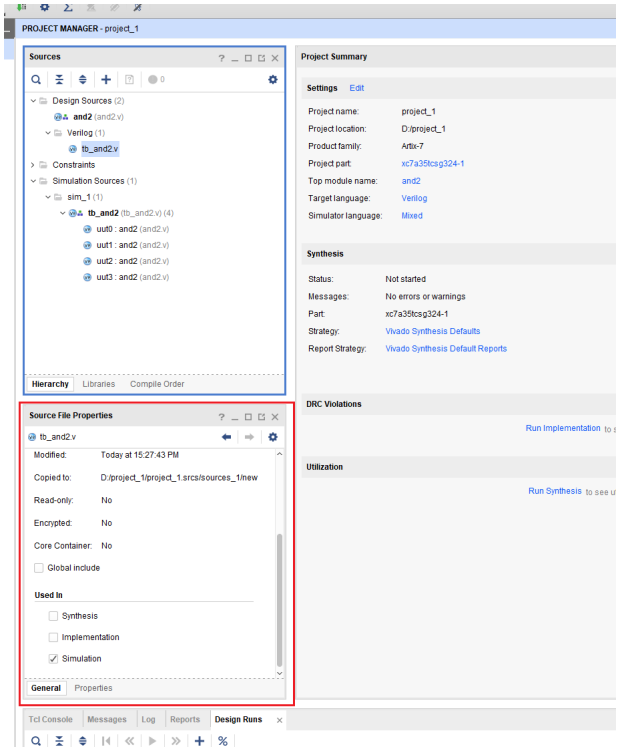


图 2.5　修改属性

在主界面左侧点击 RTL ANALYSIS 下的 Openn Elaborated Design，打开后选择菜单栏中的 Window->I/O Ports，如图2.6。

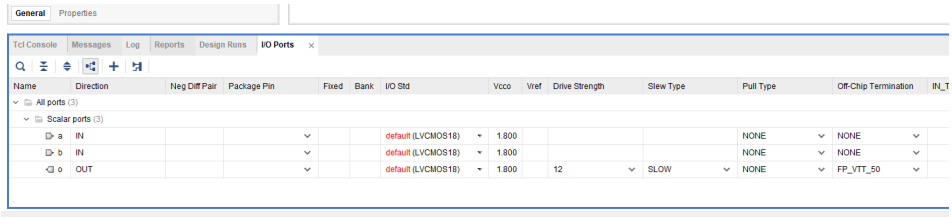


图 2.6　分配引脚

这里关注的参数主要有两个，一个 Package Pin，一个是 IO Std，观察原理图，并参考图2.7，将与门的输入连接到板卡的某两个开关上，输出连接到板卡的某个 LED 灯上，填写对应的电平标准 LVCMOS33，以及对应的位置信息并填写。

作业 2：查阅原理图，在界面上填写正确的管脚并截图。

注意，到这里应该已经学会阅读原理图了，之后本文出现的原理图三个字均指开发板的 schematic，并不再给出链接。

填写好后，保存，弹出对话框填写 xdc 文件的名字，**作业 3：打开该文件，记录文件中的内容。**在任何开发中，除非明确说明可以使用中文，建议在任何文件夹/ 文

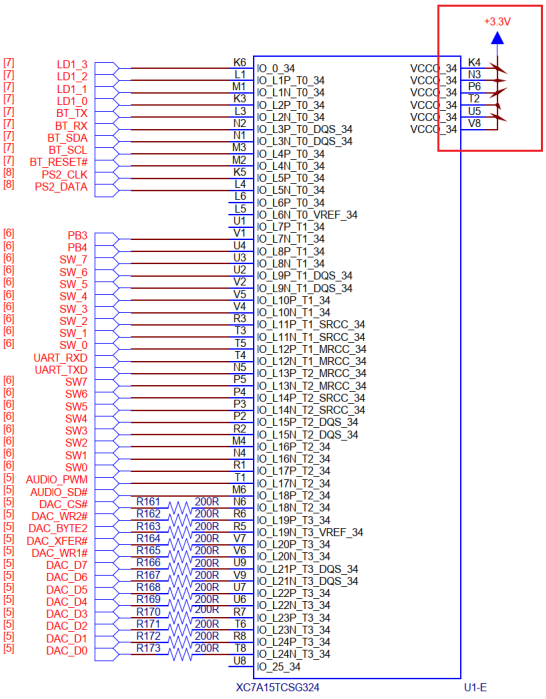


图 2.7　电平标准

件命名过程中总是使用英文小写字母 a-z、阿拉伯数字 0-9 与 _，且以字母开头。

接着我们点击 Run Implementation，并在执行结束后，点击 Open Implemented Design。

作业 4：观察 Device 窗口，找到高亮的 LUT。找到高亮的 IO，分析其相对位置关系。

请记住这个作业的截图，后续还会涉及到与其相关的概念。（FPGA 的结构不是本实验的重点，但想成为 FPGA 工程师，务必对 FPGA 的结构熟悉，参考 Xilinx 官方网站CLB的内容。）

在左侧的 Generate Bitstream 上右击，点击 Configure additional bitstream settings。将 General -> Enable Bitstream Compression 置为 True，将 Configuration -> Configuration Pin Settings during User Mode -> Unused IOB Pins 置为 PullNone。确认并保存。

作业 5：打开 xdc 文件，相对于作业 3，记录文件中的内容变化。

回到主界面，点击 Generate Bitstream。等待生成完毕。将板卡通过 USB 线接到电脑上，打开板卡电源。

点击 Open Hardware Manager，点击 Open target -> AUTO Connect。右击 xc7a35t_0 -> Program Device，弹出的对话框中应自动置入了 and2.bit 的路径，若没有，该文件默

认会生成在工程目录的.runs/impl_/下。

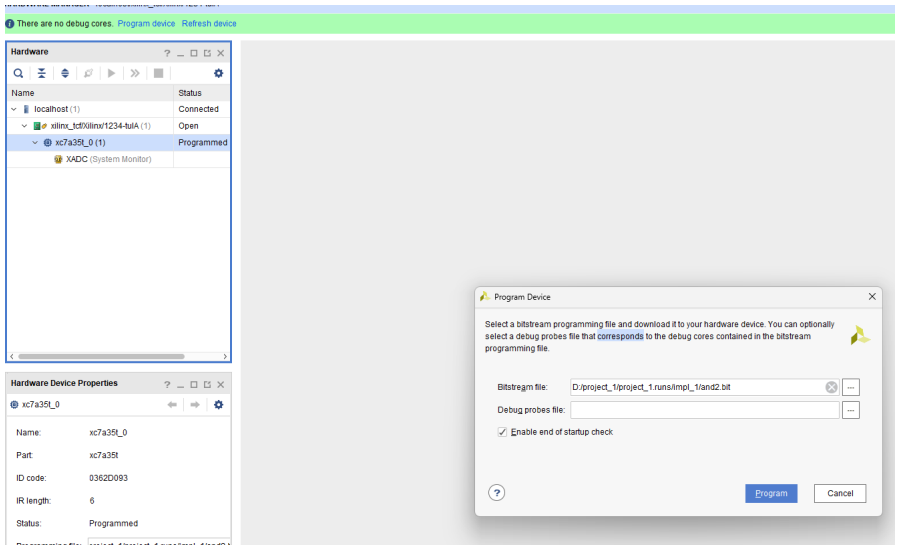


图 2.8　烧写

2.4　网站提交 bit 流：实验 1: 多个与门

阅读给出的工程，了解工程结构，修改 user_code.v。完成如下功能。

扩展至 SW0~SW7。即使用 SW0~SW7 完成 4 个与门，显示在 LD2_0 ~LD2_3 上。对应关系为 SW0、SW1 对应 LD2_0，SW2、SW3 对应 LD2_1，以此类推。

打开给出的工程，在 vivado 中新建一个 user_code.v 文件，

```
1
2  module user_code
3  (
4      input         clk,
5
6      input         reset,
7
8      input [3:0]   mode,
9      input [7:0]   sw7_sw0,
10     input [3:0]   sw,
11     input         push_button_middle,
12
13     output [7:0] led1,
```

```
    output [7:0] led2,
    output [7:0] segments_0,
    output [3:0] segments_0_sel,
    output [7:0] segments_1,
    output [3:0] segments_1_sel
    );

    //不用的输出需要给个默认值
    assign led1[7:0] = 0;
    assign led2[7:4] = 0;
    assign segments_1 = 0;
    assign segments_1_sel = 0;
    assign segments_0 = 0;
    assign segments_0_sel = 0;
    //ex1
    assign led2[0] = sw7_sw0[0] & sw7_sw0[1];
    assign led2[1] = sw7_sw0[2] & sw7_sw0[3];
    assign led2[2] = sw7_sw0[4] & sw7_sw0[5];
    assign led2[3] = sw7_sw0[6] & sw7_sw0[7];
```

打开综合后的 Schematic，截图。在硬件上实现并验证结果。

2.5 关于给出的工程

首先，由于采用了自动检查功能，必须使用本课程给出的框架完成最后代码提交。在顶层中给出了需要编写的模块的端口信息。编写该模块并综合提交 bit 文件。

修改给出工程 digital_circuits_lab_top.Verilog 中的班级号和学号。综合并提交至乐学。

```

// todo:  change
wire [63:0]  class_id = 64'hDEADBEEF;
wire [63:0]  student_id = 64'h6120220174;
```

随工程还给出了一个 bit 文件，供验证板卡硬件功能及确认最后实现的功能。

第 3 章　组合逻辑实验 2

实验目的：了解 FPGA 实现逻辑功能的底层原理，进一步学习 Verilog 语法，了解七段二进制数码管的显示逻辑。

上一章节主要解决的是软件使用的问题,在这一章节中,我们将进一步介绍 FPGA,并尝试使用更多板卡上的可视资源。

3.1　什么是 FPGA?

首先来考虑一个问题，在第2章中我们编写了一个二输入与门，并通过 FPGA 实现了功能，那么 FPGA 是如何用一个固定的结构，来完成不同的数字逻辑功能呢。我们生成的 bit 文件，又是如何使 FPGA 芯片产生对应的功能呢。

让我们通过一小段代码来稍微展开说说这个事，修改代码如下，并重新生成一次 bit 文件并下载测试（由于 Vivado 2017.4 版本与 Modelsim 的兼容性问题，可能默认状态下无法仿真，可以更换成 Vivado 自带的仿真器进行仿真）。

```
module and2
    (
     input  a,
     input  b,
     output o
     );
     // assign o = a & b;
     LUT2 #
       (
        .INIT(4'h8)  // Specify LUT Contents
        )
     LUT2_inst
       (
        .O(o),   // LUT general output
        .I0(a), // LUT input
        .I1(b)  // LUT input
        );
```

```
18 endmodule
```

在下载后，拨动拨码开关，我们发现程序的功能竟然跟上面写的 o = a&b 完全一致。那么这个 LUT2 是什么结构呢。简单来说，可以把 LUT2 假想成一个二输入一输出的黑盒子，里面有四个空间存放了四个小球，每个小球是 0 或者 1。当输入的 I0，I1 发生变化时，就会将对应的小球输出到 O 上。在这种情况下，我们规定，只有当 I0 = I1 = 1 的那个格子中存放的是等于 1 的小球，其余的三个小球都是 0，那么在功能上，这个黑盒子就相当于一个与门的作用。

这就是 FPGA 的基本原理，通过许多这种叫 LUT 的黑盒子，就可以完成不同的组合逻辑功能。请明确一个概念，只要输入和输出的数量是固定的，无论中间的逻辑如何，我们都可以通过 LUT 来完成组合逻辑功能。我们生成的 bit 流中，一部分内容就是将 LUT 的格子中放置好 0 或 1 的小球。

至于数字芯片设计的原理，请各位在后续的课程中了解。感兴趣的同学可以搜索关键词“标准单元库”。

作业 1：Open Implemented Design，定位使用的 LUT。截图。分析其与 IO 的相对位置关系。

3.2 可综合 Verilog

所谓可综合 Verilog，指能被综合成电路实现的 Verilog。再次强调一下这个概念，**Verilog 是硬件描述语言，而不是硬件设计语言**。以 FPGA 举例，Verilog 描述了 LUT 小格子中存放小球的逻辑值，而不是实打实的在 FPGA 中设计了一个 LUT。请务必明确这个概念。

目前位置，我们遇到了 reg / wire / 赋值 / 逻辑运算 / 例化等概念，至于 Verilog 中具体存在哪些语法，建议选择任何一本讲解 Verilog 的书籍，在本文中看到时查阅语法稍加了解即可。

在这一节中，我们将前面的 LUT 例化改写一下。首先是使用 case。

```
1  module and2
2  (
3   input      a,
4   input      b,
```

```
    output reg o
    );

    always@(*)
      case ({a,b}) //这里的 a,b 相当于把两个向量拼接在一起。
      //称为并置运算符
        2'b00  : begin
           o = 1'b0;
        end
        2'b01  : begin
           o = 1'b0;
        end
        2'b10  : begin
           o = 1'b0;
        end
        2'b11  : begin
           o = 1'b1;
        end
        default: begin
           o = 1'b0;
        end
      endcase

    // assign o = a & b;
    // LUT2 #
    // (
    // .INIT(4'h8) // Specify LUT Contents
    // )
    // LUT2_inst
    // (
    // .O(o), // LUT general output
    // .I0(a), // LUT input
    // .I1(b) // LUT input
    // );
endmodule
```

生成 bit 流并在硬件上验证。

作业 2：再次 Open Implemented Design，定位使用的 LUT。截图。分析其与 IO 的相对位置关系。

接下来我们换成使用 if 语句来实现。

```
module and2
  (
   input      a,
   input      b,
   output reg o
   );

   always @ ( * ) begin
      if(a & b)
        o = 1;
      else
        o = 0;
   end
endmodule
```

生成 bit 流并在硬件上验证。

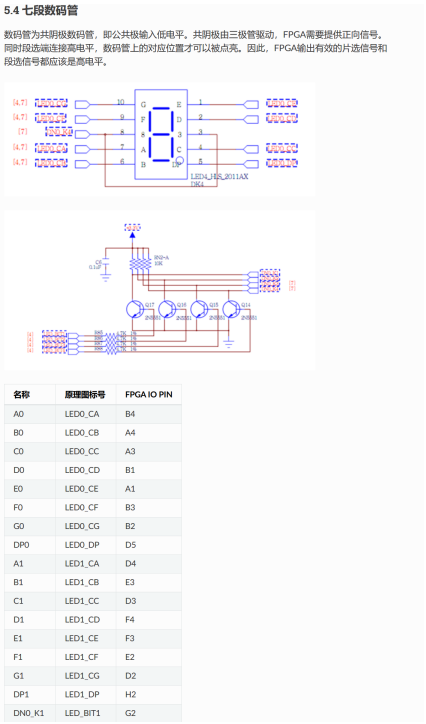
作业 3：再次 Open Implemented Design，再次定位使用的 LUT。截图。再次分析其与 IO 的相对位置关系。

对比这四种关于与门的实现方式中的 LUT，我们得到了一个结论，**不同的代码得到了同样的实现结果**。再次强调这个概念，**Verilog 是描述硬件电路而不是设计硬件电路的语言**。

暂时对可综合 Verilog 的介绍就到这里。在后面的实验中还会碰到这个环节。

3.3 七段二进制数码管显示

这一节中，我们尝试驱动板卡上另外一个可视化资源，七段二进制数码管。目前，先不去考虑如何让多个数码管同时点亮，只考虑单个数码管的情况，观察原理图，找



5.4 七段数码管

数码管为共阴极数码管，即公共极输入低电平。共阴极由三极管驱动，FPGA需要提供正向信号。同时段选端连接高电平，数码管上的对应位置才可以被点亮。因此，FPGA输出有效的片选信号和段选信号都应该是高电平。

名称	原理图标号	FPGA IO PIN
A0	LED0_CA	B4
B0	LED0_CB	A4
C0	LED0_CC	A3
D0	LED0_CD	B1
E0	LED0_CE	A1
F0	LED0_CF	B3
G0	LED0_CG	B2
DP0	LED0_DP	D5
A1	LED1_CA	D4
B1	LED1_CB	E3
C1	LED1_CC	D3
D1	LED1_CD	F4
E1	LED1_CE	F3
F1	LED1_CF	E2
G1	LED1_CG	D2
DP1	LED1_DP	H2
DN0_K1	LED_BIT1	G2

图 3.1　七段二进制数码管

到七段二进制数码管的部分。

这里我们先做一个点亮所有段的代码。注意这里在同一个工程下又新建了一个文件，并右击 Set top 了。

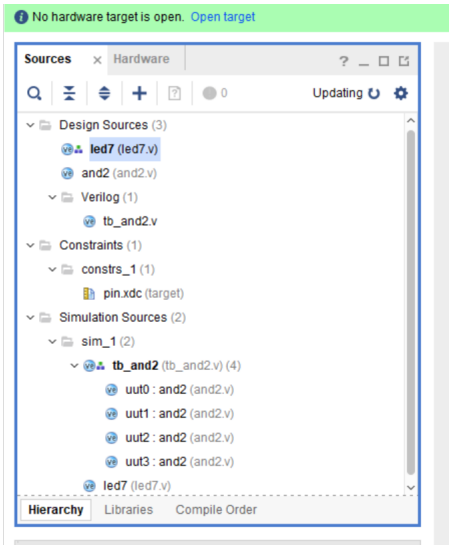


图 3.2　Set top

添加一个新的 xdc 文件并把原来的 xdc 文件置于无效（修改文件属性使其不在综合与实现过程中使用）。生成 bit 文件并观察现象。

```
module led7
(
 output [7:0] segments,
 output dn0_k1
 );

 assign segments = 8'hFF;
 assign dn0_k1 = 1;

endmodule
```

应该已经可以点亮一个七段二进制数码管了，现在让我们学习下一个可能用到的 verilog 语法，function。

```
module led7
(
```

```
output [7:0] segments,
output       dn0_k1
);

function  [7:0] gen_7ledoutput
  (input [3:0] binary_input);
   case (binary_input)
     4'h0  : gen_7ledoutput = 8'h3F;
     4'h1  : gen_7ledoutput = 8'h06;
     4'h2  : gen_7ledoutput = 8'h5B;
     4'h3  : gen_7ledoutput = 8'h4F;
     4'h4  : gen_7ledoutput = 8'h66;
     4'h5  : gen_7ledoutput = 8'h6D;
     4'h6  : gen_7ledoutput = 8'h7D;
     4'h7  : gen_7ledoutput = 8'h07;
     4'h8  : gen_7ledoutput = 8'h7F;
     4'h9  : gen_7ledoutput = 8'h6F;
     4'ha  : gen_7ledoutput = 8'h77;
     4'hb  : gen_7ledoutput = 8'h7C;
     4'hc  : gen_7ledoutput = 8'h39;

     4'hd  : gen_7ledoutput = 8'h5E;
     4'he  : gen_7ledoutput = 8'h79;
     4'hf  : gen_7ledoutput = 8'h71;
     default: begin
        gen_7ledoutput = 8'hFF;
     end
   endcase
endfunction

assign segments = gen_7ledoutput(4'h7);
assign dn0_k1 = 1;

endmodule
```

保存好这个 function，后面的实验中还会用到。

3.4 组合逻辑加法器

下面让我们来看另外一个组合逻辑的例子，组合逻辑加法器。

```
`timescale 1ns / 1ps
module adder
  (
   output [7:0] segments,
   output       dn0_k1
   );

   function  [7:0] gen_7ledoutput
     (input [3:0] binary_input);
      case (binary_input)
        4'h0  : gen_7ledoutput = 8'h3F;
        4'h1  : gen_7ledoutput = 8'h06;
        4'h2  : gen_7ledoutput = 8'h5B;
        4'h3  : gen_7ledoutput = 8'h4F;
        4'h4  : gen_7ledoutput = 8'h66;

        4'h5  : gen_7ledoutput = 8'h6D;
        4'h6  : gen_7ledoutput = 8'h7D;
        4'h7  : gen_7ledoutput = 8'h07;
        4'h8  : gen_7ledoutput = 8'h7F;
        4'h9  : gen_7ledoutput = 8'h6F;
        4'ha  : gen_7ledoutput = 8'h77;
        4'hb  : gen_7ledoutput = 8'h7C;
        4'hc  : gen_7ledoutput = 8'h39;
        4'hd  : gen_7ledoutput = 8'h5E;
        4'he  : gen_7ledoutput = 8'h79;
        4'hf  : gen_7ledoutput = 8'h71;
        default: begin
           gen_7ledoutput = 8'hFF;
```

```
30          end
31        endcase
32    endfunction
33
34    reg [31:0]   result = 0;
35
36    always @ ( * ) begin
37        result = result + 1'b1;
38    end
39
40    assign segments = gen_7ledoutput(result[3:0]);
41    assign dn0_k1 = 1;
42
43 endmodule
```

这里利用了前面的一段逻辑，单独仿真这个 module，观察波形。把这个文件当作顶层文件，生成 bit 流，下载到开发板上。观察板卡的现象。为了可以生成 bit 文件，请打开 xdc 文件，并添加

set_property ALLOW_COMBINATORIAL_LOOPS TRUE [get_nets result*]

为什么仿真和实际的现象对不上了，让我们来看下这段代码综合出的结果，点击主界面 RTL ANALYSIS -> Schematic，查看综合前的原理图。

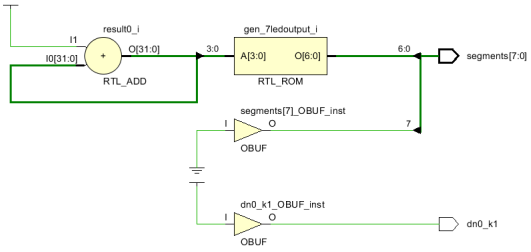


图 3.3　加法器

注意加法器的输出被直接连到输入上了。这与我们的代码设计是一致的，那么为什么仿真与实际电路的结果不一样，且都不符合我们的预期呢。前面我们在介绍 timescale 的时候提到过，可以理解成仿真软件是按 Verilog 语法，在每个 1ps 时刻计算一次各输出的结果，并在 1ns 时刻更新波形，这个波形恰好与设计出的电路的功能

一致。让我们稍微修改下代码：

```
1
2   `timescale 1ns / 1ps
3
4   module adder
5     (
6      output [7:0] segments,
7      output       dn0_k1
8      );
9
10     function  [7:0] gen_7ledoutput
11       (input [3:0] binary_input);
12        case (binary_input)
13          4'h0  : gen_7ledoutput = 8'h3F;
14          4'h1  : gen_7ledoutput = 8'h06;
15          4'h2  : gen_7ledoutput = 8'h5B;
16          4'h3  : gen_7ledoutput = 8'h4F;
17          4'h4  : gen_7ledoutput = 8'h66;
18          4'h5  : gen_7ledoutput = 8'h6D;
19          4'h6  : gen_7ledoutput = 8'h7D;
20          4'h7  : gen_7ledoutput = 8'h07;
21
22          4'h8  : gen_7ledoutput = 8'h7F;
23          4'h9  : gen_7ledoutput = 8'h6F;
24          4'ha  : gen_7ledoutput = 8'h77;
25          4'hb  : gen_7ledoutput = 8'h7C;
26          4'hc  : gen_7ledoutput = 8'h39;
27          4'hd  : gen_7ledoutput = 8'h5E;
28          4'he  : gen_7ledoutput = 8'h79;
29          4'hf  : gen_7ledoutput = 8'h71;
30          default: begin
31             gen_7ledoutput = 8'hFF;
32          end
33        endcase
```

```
34      endfunction
35
36      reg [31:0]   result = 0;
37
38      always @ ( clk ) begin
39         result = result + 1'b1;
40      end
41
42      reg clk = 1'b0;
43      always #1 clk = ~clk;
44
45
46      assign segments = gen_7ledoutput(result[3:0]);
47      assign dn0_k1 = 1;
48
49  endmodule
```

这里介绍一个概念，always block 是 verilog 一个组成单元，在仿真中，当括号中的变量（敏感列表）发生变化时，该 always block 的值会被重新计算并更新。

重新仿真该段代码，是不是奇怪的事情发生了。

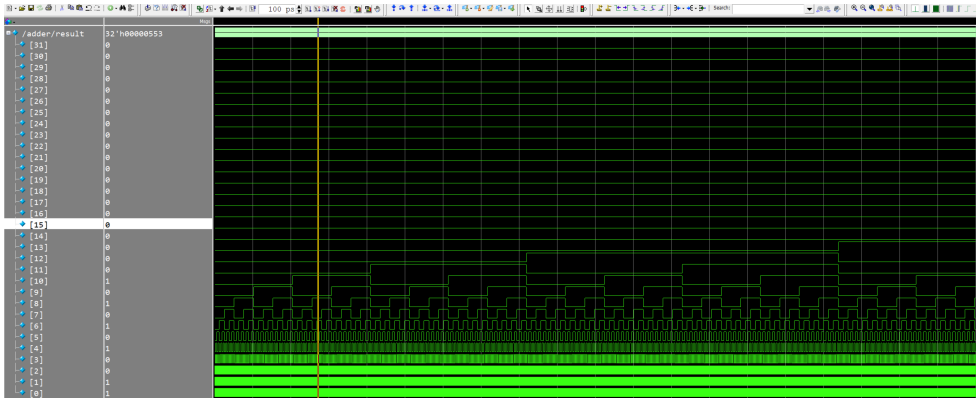


图 3.4　仿真结果

result 的结果发生变化了。这符合仿真软件的行为，当敏感列表发生变化时，重新计算了结果。

那么是不是硬件上也会发生变化呢？综合该代码并在硬件上进行测试。

作业 4：查看 RTL ANALYSIS 中的 Schematic。并截图。

通过对比，我们发现，Schematic 没有变化。那么自然的，硬件的结果也不会发生变化。

以上几个 testcase 证明了一个问题，仿真结果未必与真实电路一致。那么仿真软件在设计时就能充当什么角色，如何在二进制数码管上显示正确的自增的数字呢，我们在下一章进一步说明这个问题。

3.5 多路选择器

这一章最后一个问题，我们来看一下多路选择器（数据选择器，多路复用器，Multiplexer，mux）。通常多路选择器的原理图如图3.5所示，那么如何用代码实现这一功能呢。

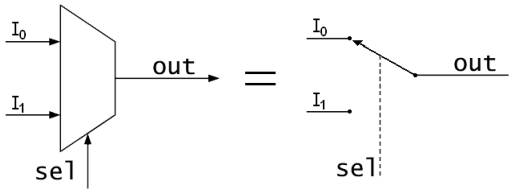


图 3.5 多路选择器

显然很容易，编写一段 verilog，通过 verilog 的三目运算符，很轻松的解决了这个问题。

```
1
2  `timescale 1ns / 1ps
3
4  module mux2
5  (
6   input I0,
7   input I1,
8   input sel,
9   output out
10   );
11
12   assign out = sel ? I1 : I0;
13 endmodule
```

那么实际上的电路呢，我们再次综合，打开综合后的 Schematic，如图3.6显示的那样。这个多路选择器被综合成了一个 LUT3，还记得前面我们提的小球的 LUT2 的

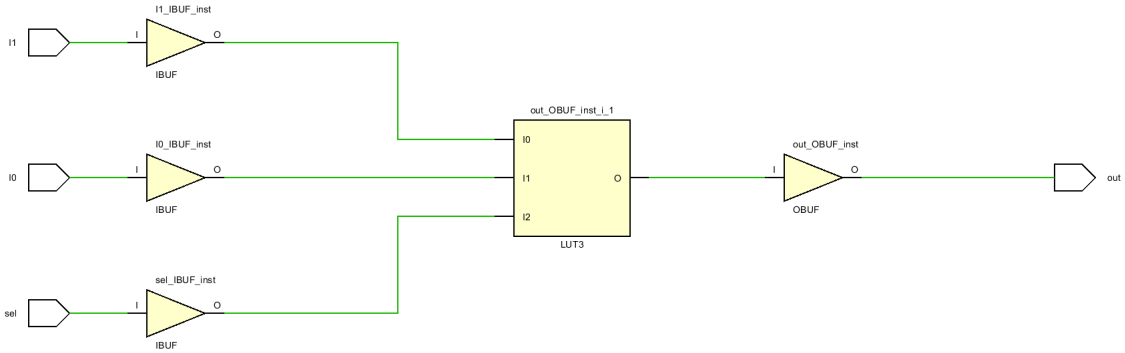


图 3.6　综合后的多路选择器

例子么？

作业 5：写出这个多路选择器使用的 LUT3 每个小球的值，以及其与对应输入的关系。类比真值表的格式。

与图3.5相比，真实的电路并没有一个单刀双掷的结构，却达到了同样的效果。

3.6　网站提交 bit 流：实验 2：七段二进制数码管

阅读给出的工程，了解工程结构，修改 user_code.v。完成如下功能。

根据 SW_0~SW_3 的值在板卡最左侧（对应丝印为 DK1）数码管上显示对应的结果。例 [SW3:SW0] 等于 4’b0001 时，在板卡最左侧数码管上显示 1，根据输入的 16 种可能，做 16 个数字的显示（0~f）。

第 4 章　时序逻辑实验

实验目的：1. 了解时序逻辑的基础概念及 D 触发器的 Verilog 描述方式，了解七段二进制数码管的扫描逻辑，会根据给出程序仿写出驱动七段二进制数码管的电路。2. 掌握按键消抖的概念。开始学习独立设计 Verilog 满足用户功能。

从这一章开始，我们终于进入了现代数字逻辑的范围。为什么要有时序逻辑呢？

4.1　时序逻辑加法

我们先来解决上一章中的加法的问题，如何顺利的在仿真和硬件结果上得到同样的结果。

为了阅读下面的部分，应该至少掌握了 D 触发器的概念。

其实在3.4一节中，我们通过添加 clk，已经在仿真里解决了一部分让 result 递增的效果。但是在硬件实现时，我们看到的电路中组合逻辑的输出与输入直接接到了一起，这会导致输出的结果振荡且不受控。现在我们把这段代码修改成正确的样子：板卡上提供了一个 100MHz 的晶振，我们引入这个晶振作为电路的输入。

```
1  `timescale 1ns / 1ps
2  module adder
3    (
4      input         clk,
5      input         reset_n,
6      output [7:0] segments,
7      output        dn0_k1
8      );
9
10     function  [7:0] gen_7ledoutput
11       (input [3:0] binary_input);
12       case (binary_input)
13         4'h0  : gen_7ledoutput = 8'h3F;
14         4'h1  : gen_7ledoutput = 8'h06;
15         4'h2  : gen_7ledoutput = 8'h5B;
16         4'h3  : gen_7ledoutput = 8'h4F;
```

```
        4'h4  : gen_7ledoutput = 8'h66;
        4'h5  : gen_7ledoutput = 8'h6D;
        4'h6  : gen_7ledoutput = 8'h7D;
        4'h7  : gen_7ledoutput = 8'h07;
        4'h8  : gen_7ledoutput = 8'h7F;

        4'h9  : gen_7ledoutput = 8'h6F;
        4'ha  : gen_7ledoutput = 8'h77;
        4'hb  : gen_7ledoutput = 8'h7C;
        4'hc  : gen_7ledoutput = 8'h39;
        4'hd  : gen_7ledoutput = 8'h5E;
        4'he  : gen_7ledoutput = 8'h79;
        4'hf  : gen_7ledoutput = 8'h71;
        default: begin
            gen_7ledoutput = 8'hFF;
        end
      endcase
    endfunction

    reg [31:0]   result = 0;

    // always @ ( clk ) begin
    // result = result + 1'b1;
    // end

    // reg clk = 1'b0;
    // always #1 clk =  clk;

    always @ ( posedge clk or negedge reset_n ) begin
      if(!reset_n)
        result <= 0;
      else
        result <= result + 1'b1;
    end
```

```
    assign segments = gen_7ledoutput(result[3:0]);
    assign dn0_k1 = 1;
endmodule
```

我们为这段代码编写一个 tb_adder.v。

```
module tb_adder(

  );

  reg clk = 1'b0;
  reg reset_n = 1'b0;
  wire [7:0] segments;
  wire       dn0_k1;

  adder uut
  (.clk(clk),
  .reset_n(reset_n),
  .segments(segments),
  .dn0_k1(dn0_k1)
  );
  always #5 clk = ~clk;

  initial begin
  reset_n = 1'b0;
  #100;
  @(posedge clk);
  reset_n = 1'b1;
  end

endmodule
```

作业 1，仿真上面这段代码，并截图，截图中务必保留 adder 中的 result 信号。这里刻意没有讲解如何查找到这个 result 信号，就是希望各位同学多熟悉 Modelsim 这个软件。

现在在仿真中，可以完成我们的需求了，在释放复位后的每一个时钟上升沿，result都可以自增，那么来看下电路图吧，打开 RTL ANALYSIS 中的 Schematic，观察原理图，发现与我们想的一致，D 触发器的输出 Q 反过来接到了加法单元的输入上。

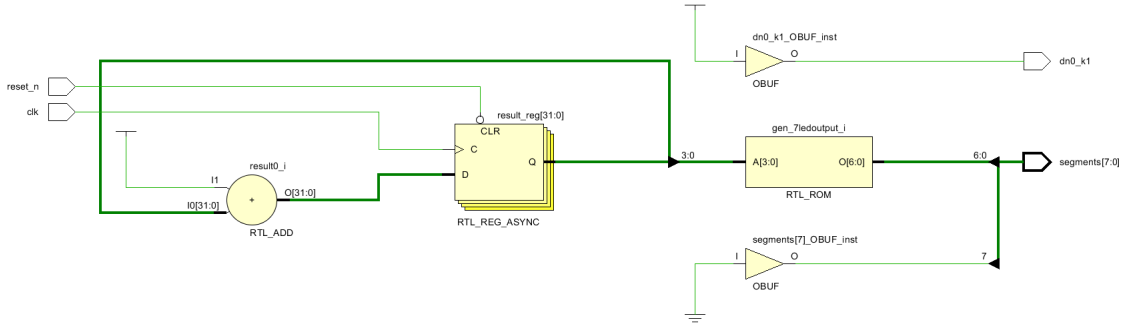


图 4.1　时序逻辑实现的加法器

从开始到现在，一直在说不需要掌握 Verilog 的语法，是因为绝大多数情况下，Verilog 的写法是相对固定的：你需要一个 D 触发器电路，就必须存在一个 always@posedge 的代码段；你需要一段组合逻辑帮你处理算法，那就必须存在一个加法/case/if 等等代码段。所以这篇文章的多数代码在看懂了的基础上，在以后的设计中照抄是最稳妥的办法。

我想在这里再强调一遍 Verilog 是描述硬件电路的语言，而不是硬件设计语言。

4.2　视觉残留与数码管显示

那么，现在这段代码生成 bit 文件会好用么，我们先来考虑视觉暂留这个概念。视觉暂留（英文：Persistence of vision）也称为正片后像，是光对视网膜所产生的视觉，在光停止作用后，仍然保留一段时间的现象，其具体应用是电影的拍摄和放映。原因是由视神经的反应速度造成的，其时值约是 1/16 秒，对于不同频率的光有不同的暂留时间。是现代影视、动画等视觉媒体制作和传播的根据。

这里直接给出答案，上一节的代码下到板卡上的现象不是我们想看到的，因为一个以 100MHz 频率自加的 LED 除了让灯看起来全亮并没有其他的功能，那么我们考虑将时钟降速。如何降速呢，看这段代码：

```
1  `timescale 1ns / 1ps
2
```

```
3  module adder
4    (
5     input         clk,
6     input         reset_n, //表示复位为低有效
7     output [7:0] segments,
8     output        dn0_k1
9     );
10
11    function  [7:0] gen_7ledoutput
12      (input [3:0] binary_input);
13       case (binary_input)
14         4'h0  : gen_7ledoutput = 8'h3F;
15         4'h1  : gen_7ledoutput = 8'h06;
16         4'h2  : gen_7ledoutput = 8'h5B;
17         4'h3  : gen_7ledoutput = 8'h4F;
18         4'h4  : gen_7ledoutput = 8'h66;
19         4'h5  : gen_7ledoutput = 8'h6D;
20         4'h6  : gen_7ledoutput = 8'h7D;
21         4'h7  : gen_7ledoutput = 8'h07;
22         4'h8  : gen_7ledoutput = 8'h7F;
23         4'h9  : gen_7ledoutput = 8'h6F;
24         4'ha  : gen_7ledoutput = 8'h77;
25         4'hb  : gen_7ledoutput = 8'h7C;
26         4'hc  : gen_7ledoutput = 8'h39;
27         4'hd  : gen_7ledoutput = 8'h5E;
28         4'he  : gen_7ledoutput = 8'h79;
29         4'hf  : gen_7ledoutput = 8'h71;
30         default: begin
31            gen_7ledoutput = 8'hFF;
32         end
33       endcase
34    endfunction // gen_7ledoutput
35
36    wire reset = !reset_n;
37    reg [31:0] result = 0;
```

```
38      reg [31:0] cnt = 0;
39      reg        clk_en = 1'b0;
40
41      always @ ( posedge clk or posedge reset ) begin
42         if(reset)begin
43            cnt <= 0;
44            clk_en <= 1'b0;
45         end else if(cnt == (32'd100_000_000 - 1)) begin
46            cnt <= 0;
47            clk_en <= 1'b1;
48         end else begin
49            cnt <= cnt + 1'b1;
50            clk_en <=1'b0;
51         end
52      end
53      always @ ( posedge clk or posedge reset ) begin
54         if(reset)
55           result <= 0;
56         else if(clk_en)
57           result <= result + 1'b1;
58      end
59
60      assign segments = gen_7ledoutput(result[3:0]);
61      assign dn0_k1 = 1;
62
63  endmodule
```

试一下，是不是现在可以按照我们想的那样，解决问题了。

4.3 实验 3：Let the Lights On

阅读给出的工程，了解工程结构，修改 user_code.v。完成如下功能。

从第一章开始，给出了全部的参考代码，现在轮到你们自己动手了，写一个模块，该模块的输入是 100MHz 的时钟以及高有效的复位，在解除复位后，可以以 250ms 的

周期、十六进制递增驱动左侧（对应 dk1~dk4）四个数码管，dk1 为高位。即显示 0000 -> 0001 -> ... -> 000F -> 0010 -> ... -> FFFF。要求数码管的刷新周期为 5ms，且由左至右刷新，即每 5ms，由 dk1 点亮变为 dk2 点亮。

在复位时，规定 dk1 灯点亮，dk2-dk4 熄灭。四个数码管的值为 0000（实际只有 dk1 显示）。并以此为基点开始动作。

注意提供的时钟为 100MHz，请不要修改。

检查内容如下：

1.（100MHz 时钟，25_000_000 * n 个上升沿，采样，25_000_000 个上升沿，再采样，n=rand(学号)；）并检查显示的值是否加 1；

2.（在复位解除后，100MHz 时钟，500_000 个上升沿）检查 dk[1:4] 的值，应由 1000 -> 0100;

生成 bit 文件并在板卡上测试。

4.4 网站提交 bit 流：实验 4：按键消抖

阅读给出的工程，了解工程结构，修改 user_code.v。完成如下功能。

以十六进制递增驱动左侧（对应 dk1~dk4）四个数码管。即复位后，显示 0000，一次按键后显示 0001，至 FFFF。请自行查阅按键消抖的概念，并实现相应功能：修改上一节的代码，去掉周期性的自加，修改为每次按键 push button，输出加 1。复位时数据清零，最左侧数码管（对应 dk1）显示 0。每 5ms 切换。定义每次按键动作为 push button 的值从 0 变成 1。按键频率最大为 5Hz。（每次按键时，按下及释放的状态不少于 100ms）。

生成 bit 文件并在板卡上测试。

检查内容如下：

1. 在复位解除后，产生一个 120ms 的按键脉冲，检验数据是否自增 1。

2. 在复位解除后，产生一个 1ms 的按键脉冲，检验数据是否不变。

第 5 章　时序逻辑实验 2

实验目的：了解 IP 生成及调用的过程，会使用 coe 文件给 bram 初始化。进一步训练独立设计 Verilog 的能力，形成根据时序图完成代码设计的初步思路。

5.1　原语 Primitive

在第3章中，我们介绍过 LUT 的概念，也看过 FPGA 的内部结构。在这一章中我们介绍在本课程中最后一个 FPGA 的概念：原语（Primitive）。

Primitive 指在 FPGA 中内置好的可以直接被例化的**硬件资源**，其具有固定的逻辑功能，比如 LUT 就是一个具有固定功能的硬件资源。作用是通过放置不同的小球实现不同的功能。

5.1.1　RAM

在这一章中要介绍的一个原语是 RAM。

假设有一个 10 输入的 LUT，LUT 的每个格子里面可以放 32 个小球。每次改变输入的值，就能取出对应的小球。且小球的值可以在 FPGA 运行时改变。

在现代的 FPGA 中就存在这样一种专门的硬件资源，SRAM（同步，随机，访问，存储器），其基本的概念在 SoC，嵌入式系统中都是非常重要的。

我们来看一下 ram 的时序图。

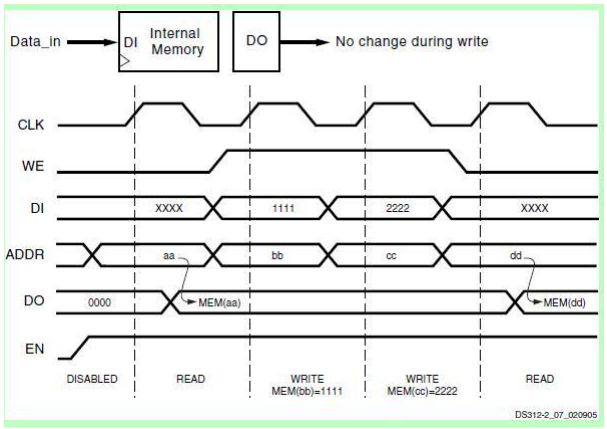


图 5.1　RAM 的时序图

RAM 中，选择小球的输入被称为地址（ADDR），由图中可以看出，在时钟上升沿来临时，如果 write _enable(we) 信号为高，那么对应格子的内容就会被改变。当 we 信号为低时，输出的才是对应格子里的内容。

上面的时序图应该记住。在目前的处理器结构中，该时序图是最常用的。

5.1.2 RAM 的例化

Xilinx（AMD）制造 FPGA 时，内置的 RAM 有两种大小，RAM36kb 与 RAM18kb（其实是 32kb 与 16kb，多出来的部分是用来做 ECC 校验的）（顺带一提的是没有 LUT2 和 LUT3，只有 LUT6 和 LUT7，但不影响理解），在 Vivado 中例化 RAM 的操作如下，打开 Tools -> language templates，搜索 RAM。即可查看到例化的示例。

5.1.3 IP 核生成与调用

那么当我们想使用超过 36kbRAM 时怎么办呢，重复例化并为每个 RAM 赋值显然复杂了，Vivado 中内置了另外的方式来帮助我们解决这个问题：IP 核。

点击主界面 PROJECT MANAGER -> IP Catalog，搜索 Block Memory Generator，就可以构筑更大的 BRAM，并且生成的 BRAM 在界面上可以找到例化的模板。

作业 1：生成一个深度 512，宽度 32bit 的 Single Port BRAM。并截图。

5.2 网站提交 bit 流：实验 5：呼吸灯

阅读给出的工程，了解工程结构，修改 user_code.v。完成如下功能。阅读下面的资料，了解占空比会影响 LED 亮度的原因及方式。完成 $y = e^{sin(2\pi t)} - e^{-1}$ 为亮度曲线的呼吸灯。

驱动的呼吸灯为 LD1[0]。

LED 亮度调节原理：人眼感受到的亮度可以理解为一定时间内光强的积分。在本次实验中，我们将利用这一点来完成这个呼吸灯的设计, 其原理是通过一定时间内 LED 的发光周期来调整亮度。来看这样一段 **Matlab** 代码。

```
% Define parameters
T = 8;
N = 512;
```

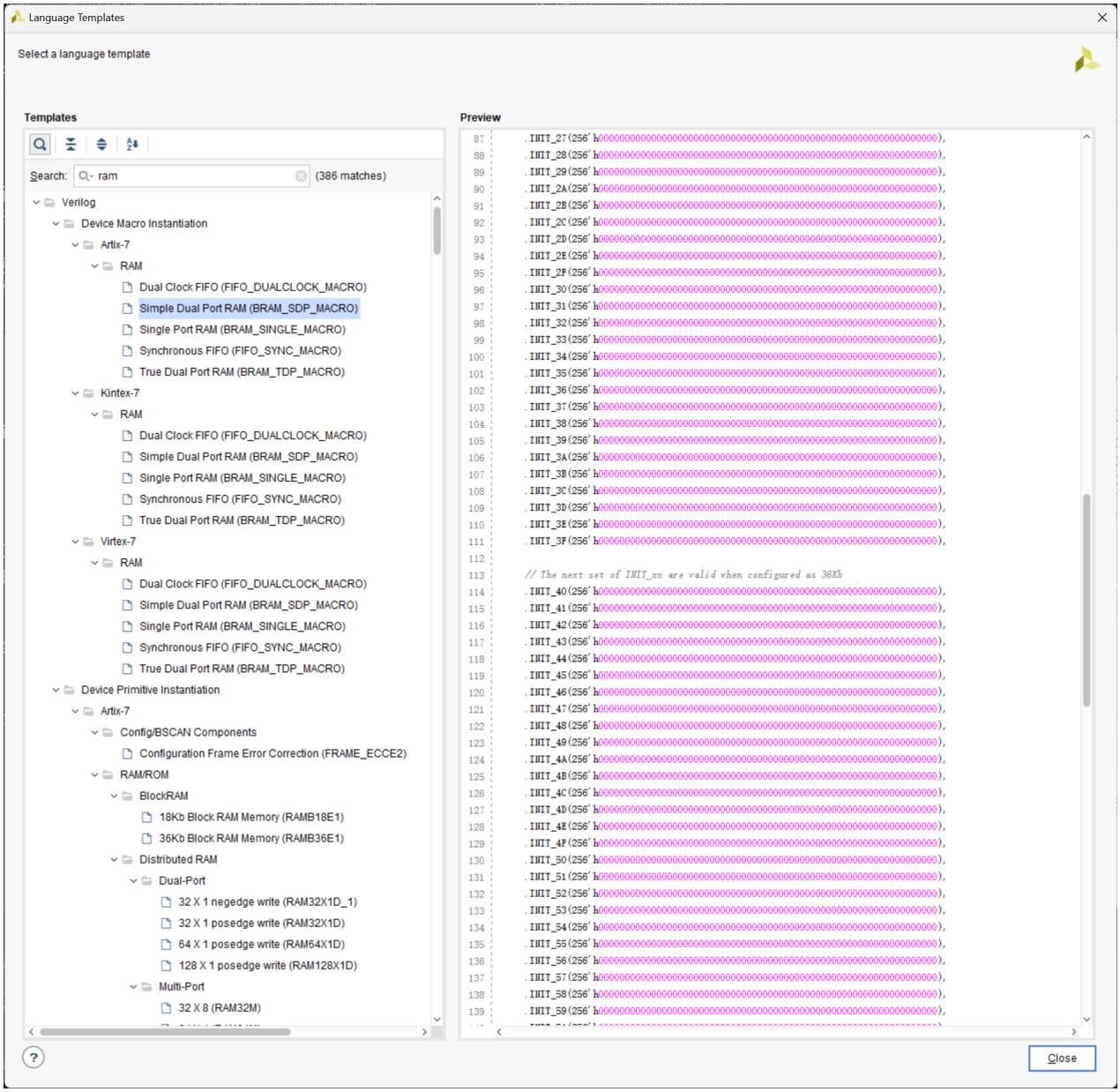


图 5.2　Language Templates

```
x = 0 : T/N : T- 1/N;
y1 = exp(sin(2*pi*x/T)) - exp(-1);
y_expand_1 = y1./ max(y1) .* 1600;
y_expand_1600 = uint32((y1./ max(y1)) .* 1600);
bar(x,y_expand_1600);
```

执行后，会生成这样一张柱状图5.3。这个实验是要按照这个柱状图编写程序：规定输入为 100MHz 时钟，300 个时钟周期为一个单位，最大亮度为 1600 个单位。（即最大亮度下 LED 会被点亮 1600*300*10ns=4.8ms），整个函数的周期为 512 个 4.8ms。LED 应当被循环点亮，即到达第 511 个后，从第 0 个重复。

按照柱状图，第一个 4.8ms 的亮度为 430。512 个柱状图的值在 breathing_freq.txt 给出。查阅资料，学习 BRAM 的初始化文件的编写方式，将 512 个柱状图的值置入本章作业 1 的 Single Port BRAM 中。

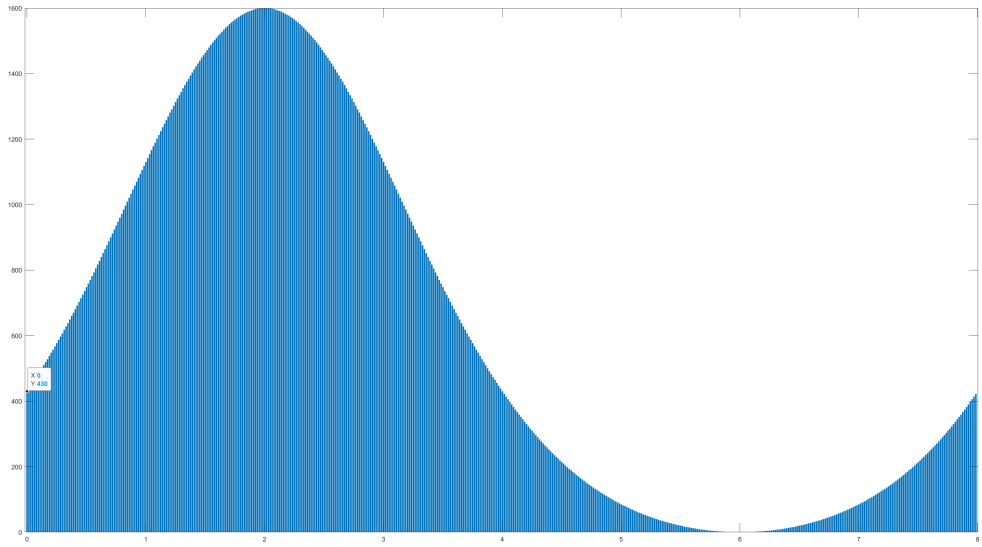


图 5.3　呼吸函数的柱状图

具体实现的电路功能的仿真波形如图5.4所示，为便于检查，规定复位时 LED 点亮，每 4.8ms 中，LED 先亮后灭。注意其中 5 个 Cursor 的相对关系。举例来说，Cursor2 - Cursor1 = 1290000ns，1290000 = 300 * 430 * 10ns；Cursor3 - Cursor2 = 3510000ns，1290000 + 3510000 = 4.8ms。并注意图5.5中的占空比（亮度）变化。

自动检查系统会依据柱状图的值，在解除复位后，随机产生 0~3s 时长的 100Mhz 时钟送给系统，并判断对应时刻的 LED 是否被点亮。

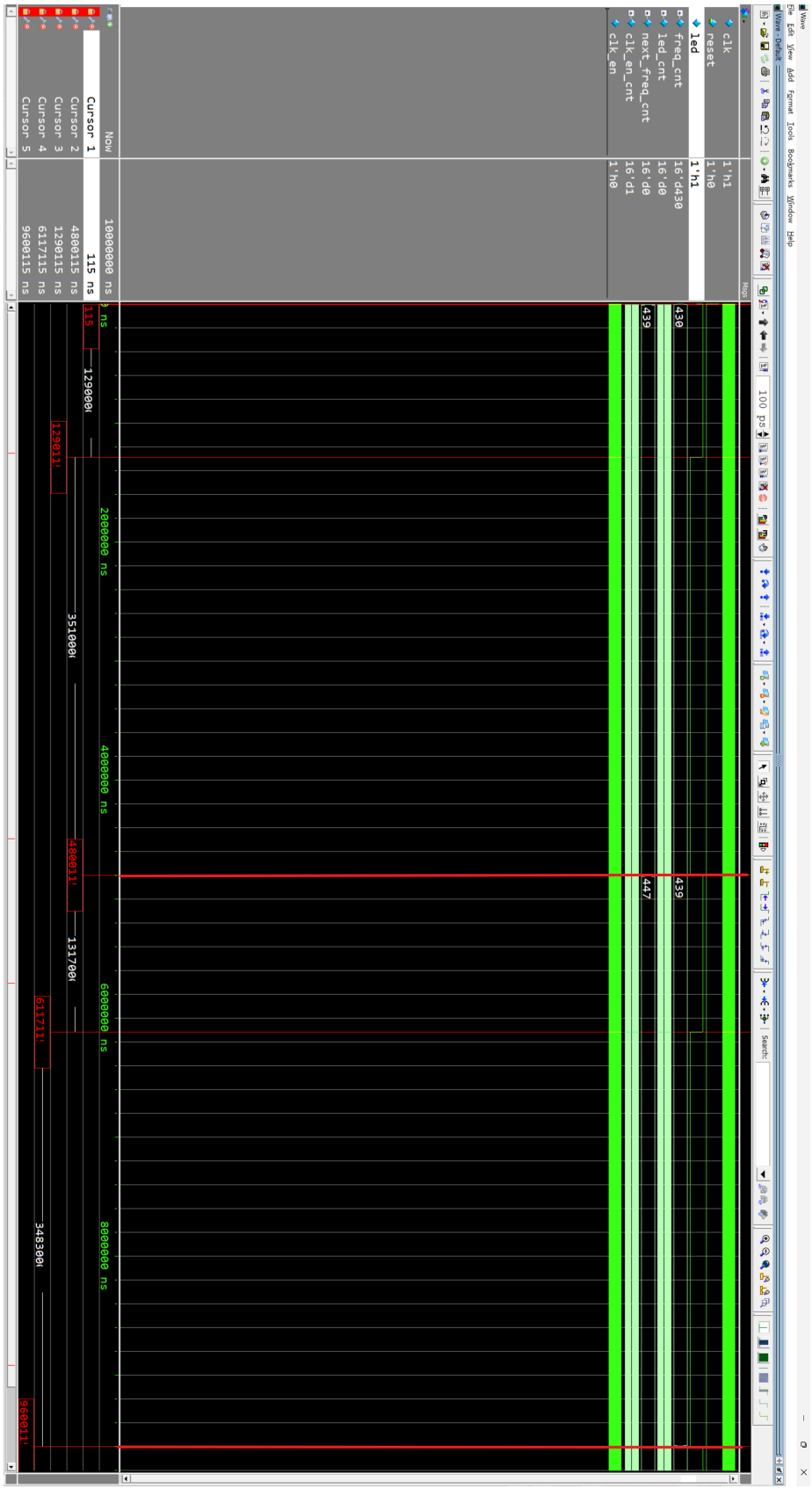

clk
reset
led
freq_cnt
led_cnt
next_freq_cnt
clk_en_cnt
clk_en
Now
Cursor 1
Cursor 2
Cursor 3
Cursor 4
Cursor 5
115 ns
480115 ns
1290115 ns
6117115 ns
9600115 ns
439
447
129011
351000
480011
131790
611711
348300
960011

图 5.4　仿真波形

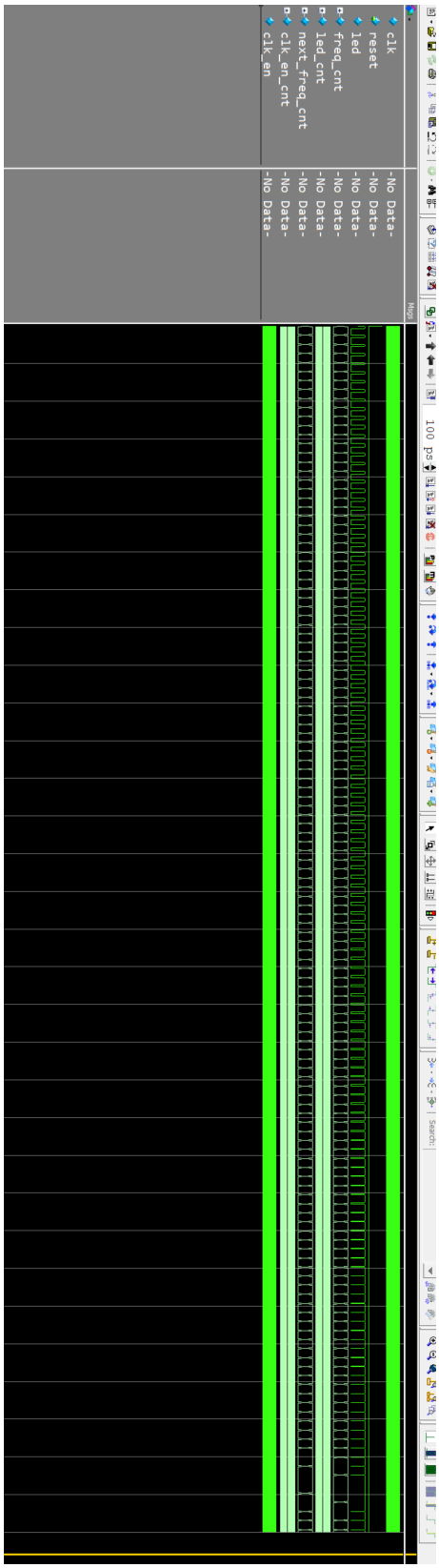

clk
reset
led
freq_cnt
led_cnt
next_freq_cnt
clk_en_cnt
clk_en
-No Data-

图 5.5　占空比变化

第 6 章　附加题

像第一章里说的那样，任选两道附加题中的一题，编写出代码通过验证，并制作 PPT 参与答辩，即可拿到全部实验分数。

需要提交的内容：1. 提供全部源代码。2. 自行设计硬件测试环境进行现场演示。

6.1　实验 6：UART 欺骗

已知电脑程序上的自动检查系统使用的是板卡的串口输出作为判断用户程序功能的依据。编写 FPGA 程序强制通过串口输出实验 1 的正确串口信息，骗过自动检查系统。

6.2　实验 7：GPS 信息解包

已知 GPS 模块通过串口输出的信息为 ASCII 字符串，且某一种只包含时间格式，且长度固定的字符串格式为 $GPZDA,204007.00,13,05,2022,,*62。编写 FPGA 程序，输出解析出的时分秒信息，例 56’h20220513204007。输入条件，波特率 115200，奇校验，停止位 2 位，数据位 8 位。