组成原理课程第三次实验报告

实验名称：数字逻辑电路设计基础

学号：2310422　姓名：谢小珂　　班次：　1078

一、实验目的

1.复习设计 CPU 必须掌握的数字逻辑电路和 Verilog 描述的知识。
2.理解同步 RAM 和异步 RAM 的区别及其仿真行为。
3.初步掌握进行数字逻辑电路功能仿真时常见的错误及其调试方法。

二、实验内容说明

（1）实践任务一：寄存器堆仿真

1. 《CPU 设计实战》提供的寄存器堆源码为“两读一写”的结构，也就是有两个读端口(读端口没有使能位控制，表示永远使能)和一个写端口。接口信号如表 3-1 所示。

名称	宽度	方向	描述
时钟			
clk	1	input	时钟信号
读端口一			
raddr1	5	input	寄存器堆读地址 1
rdata1	32	output	寄存器堆读返回数据 1
读端口二			
raddr2	5	input	寄存器堆读地址 2
Rdata2	32	output	寄存器堆读返回数据 2
写端口			
we	1	input	寄存器堆写使能
waddr	5	input	寄存器堆写地址
wdata	32	input	寄存器堆写数据

表 3-1 寄存器堆接口信号列表

针对任务一寄存器堆实验，完成仿真，在感想收获中思考并回答问题：为什么寄存器堆要设计成“两读一写”？

（2）实践任务二：同步 RAM 和异步 RAM 仿真、综合与实现

本实践任务要求为同步 RAM、异步 RAM 各建立一个工程，调用 Xilinx 库 IP 实例化同步 RAM、异步 RAM，会提供一个设计的顶层文件，将它们封装成相同的模块名和接口。封装后的 RAM 接口信号如表 3-2 所示。

表 3-2 RAM 顶层接口信号列表

名称	宽度	方向	描述
clk	1	Input	时钟信号
ram_wen	1	Input	RAM 的写使能信号：为 1 表示写入操作，为 0 表示读取操作

ram_addr	16	Input	RAM 的地址信号，读和写的地址都由该信号指示
ram_wdata	32	Input	RAM 的写数据信号，表示写入的数据
ram_rdata	32	Output	RAM 的读数据信号，表示读出的数据

注意:封装后的 RAM 接口没有使能(或者称为片选)信号，表示永远使能(使能信号在 RAM 内部恒为 1)。

在实践过程中，应特别注意以下几点:
1)生成 IP 时，请将对应 IP 命名为 block ram 和 distributed ram，如命名错误，IP 将会报错。若遇到已生成 IP 无法改名的情况，可以删除该 P，重新生成。
2)生成 IP 时，可以点击窗口左侧的图查看接口信息。当参数正确时，端口名和宽度应与指定的顶层文件中的调用相对应。
3)有兴趣的读者可以自行调研、参考同步/异步 RAM 定制的资料，并根据仿真波形对比参数的作用。
4)对程序进行综合之前请确保已正确加载约束文件(ram.xdc)。
5)添加 testbench 时请注意选择 add simulation source，否则会导致顶层文件错误，综合结果不正确。
6)对程序进行综合时，所用的计算机不同，综合时间会有一定的差异。综合时会耗费大量时间，所以应提前计划，安排好时间。
7)时序报告和资源报告的生成需要查看综合、实现完成后的结果。

针对任务二同步 ram 和异步 ram 实验，可以参考实验指导手册中的存储器实验，注意同步和异步需要分开建工程,然后仿真,在感想收获中分析同步 ram 和异步 ram 各自的特点和区别。

（3）实践任务三：数字逻辑电路的设计与调试

本实践任务提供了一个有 5 个 bug 的数字逻辑电路设计源码。该设计的正确功能是:
1) 获取开发板最右侧 4 个拨码开关的状态(记为“拨上为 1，拨下为 0”，实际开发板上拨码开关的电平是“拨上为低电平，拨下为高电平”)，共有 16 个状态(数字编号是 0~15)。
2) 最左侧数码管实时显示 4 个拨码开关的状态。数码管只支持显示 0~9，如果拨码开关状态是 10~15，则数码管的显示状态不更改(显示上一次的显示值)。
3)最右侧的 4 个单色 LED 灯会显示上一次的拨码开关的状态，支持显示 0~ 15(拨码开关拨上，对应 LED 灯亮)。
比如，初始状态下，4 个拨码开关拨下，按复位键，则数码管显示 0，LED 灯都不亮;拨码开关拨为 1，则数码管显示 1，LED 灯还是都不亮;拨码开关再拨为 3，则数码管显示 3，LED 灯显示 1。
提供的设计源码中包含 5 个 bug，其中 4 个是 3.2.2 节中提到的波形异常的前 4 种情况:波形为“Z”、波形为“X”、波形停止和越沿采样，另外的 1 个 bug 是功能 bug。本任务提供的示例设计的顶层接口如表 3-3 所示。

表 3-3 示例设计的顶层信号列表

名称	宽度	方向	描述
clk	1	input	时钟信号
resetn	1	input	复位信号
switch	4	input	对应开发板上最右侧 4 个拨码开关
num_csn	8	output	数码管的片选信号
num_a_g	7	output	数码管的 7 段信号
led	4	output	对应开发板上最右侧 4 个单色 LED 灯

针对任务三，重点介绍清楚发现 bug、修改 bug 和验证的过程，在感想收获中总结使用 vivado 调试的经验步骤。

三、实验步骤

（一）实践任务一：寄存器堆仿真

·添加 regfile.v 文件和 rf_tb 文件

Project Summary | regfile.v | rf_tb.v

K:/jizu/regfile/regfile.srcs/sources_1/new/regfile.v

```
1  `timescale 1ns / 1ps
2  module regfile(
3    input         clk,
4    input  [ 4:0] raddr1,
5    output [31:0] rdata1,
6    input  [ 4:0] raddr2,
7    output [31:0] rdata2,
8    input         we,
9    input  [ 4:0] waddr,
10   input  [31:0] wdata
11 );
12 reg [31:0] rf[31:0];
13 // WRITE
14 always @(posedge clk) begin
15     if (we) rf[waddr]<= wdata;
16 end
17 // READ OUT 1
18 assign rdata1 = (raddr1==5'b0) ? 32'b0 : rf[raddr1];
19 // READ OUT 2
20 assign rdata2 = (raddr2==5'b0) ? 32'b0 : rf[raddr2];
21 endmodule
22
```

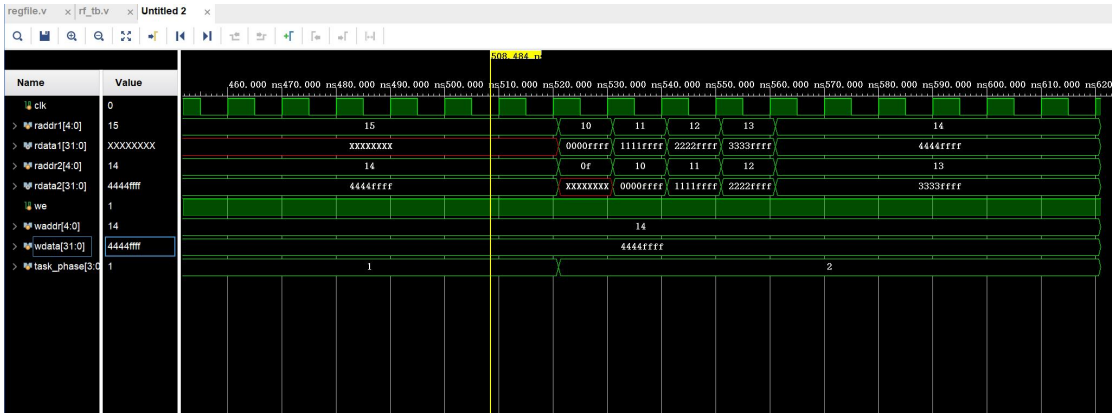
regfile.v | rf_tb.v *

K:/jizu/regfile/regfile.srcs/sim_1/new/rf_tb.v

```
34     initial
35     begin
36         raddr1 =  5'd0;
37         raddr2 =  5'd0;
38         waddr  =  5'd0;
39         wdata  = 32'd0;
40         we     =  1'd0;
41         task_phase =  4'd0;
42         #2000
43
44         $display("==============================");
45         $display("Test Begin");
46         #1;
47         // Part 0 Begin
48         #10;
49         task_phase = 4'd0;
50         we         = 1'b0;
51         waddr      = 5'd1;
52         wdata      = 32'hffffffff;
53         raddr1     = 5'd1;
54         #10;
55         we         = 1'b1;
56         waddr      = 5'd1;
57         wdata      = 32'h1111ffff;
```

将 rf_tb 文件中第四十二行的#2000 改短或者删除。

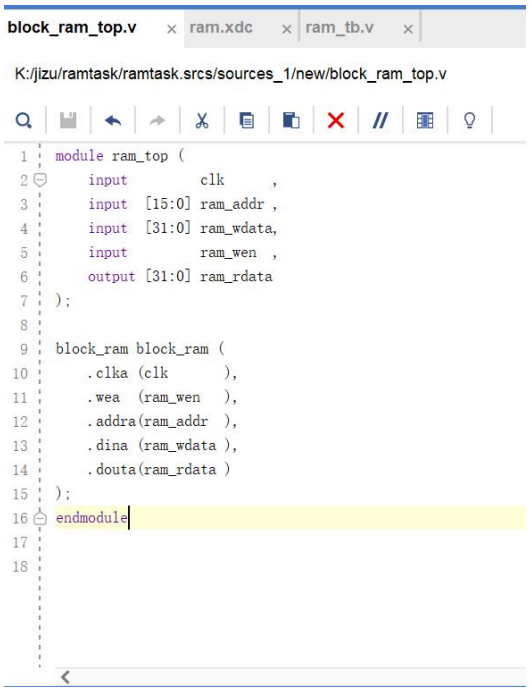
·进行仿真测试，结合波形观察寄存器堆的读写行为



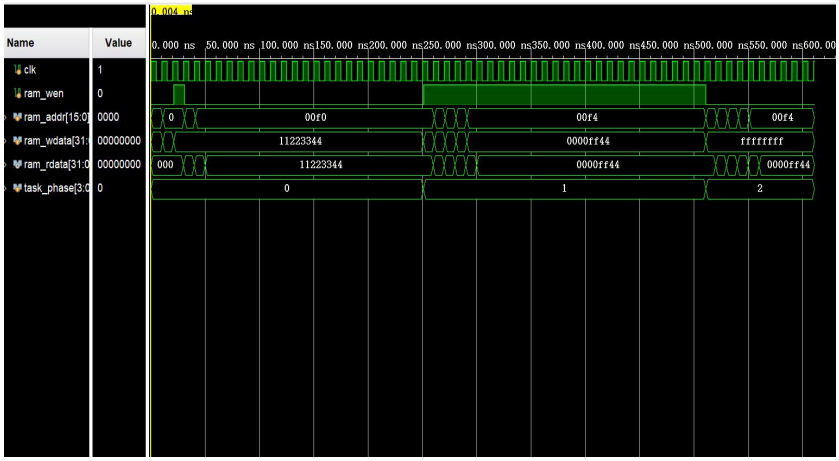
（二）实践任务二：同步 RAM 和异步 RAM 仿真、综合与实现

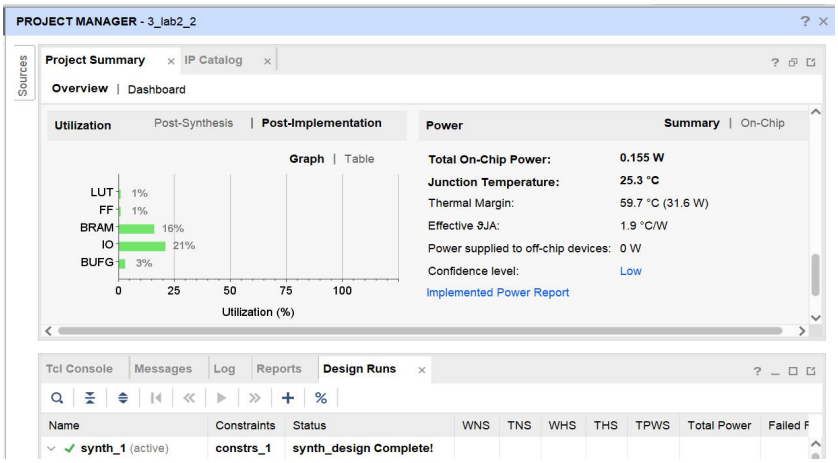
1. 同步：

・添加 block_ram_top.v 文件、ram_tb 文件和 ram.xdc 文件



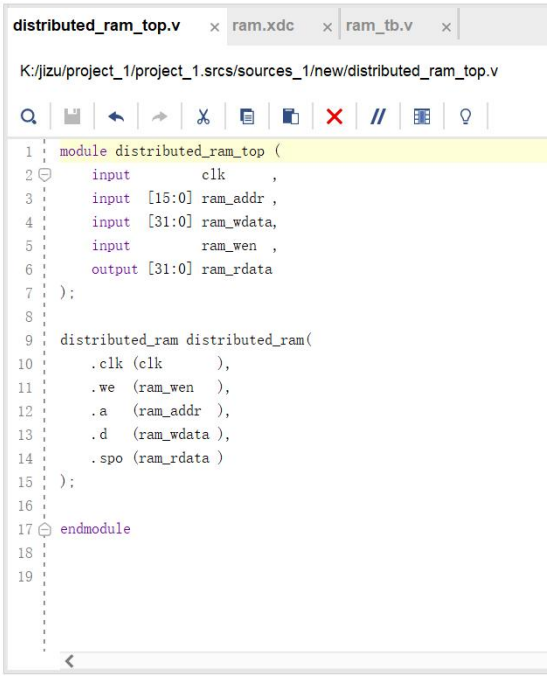
・进行仿真测试，生成同步 RAM，如下图所示



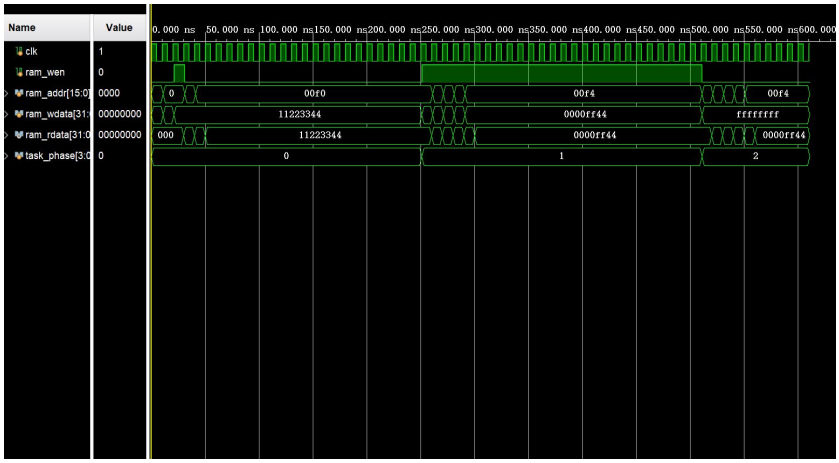


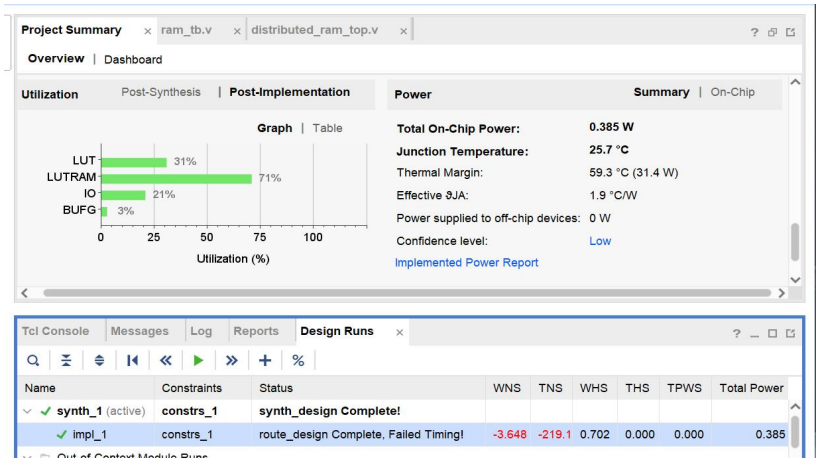
2. 异步：

・添加 distributed_ram_top.v 文件、ram_tb 文件和 ram.xdc 文件



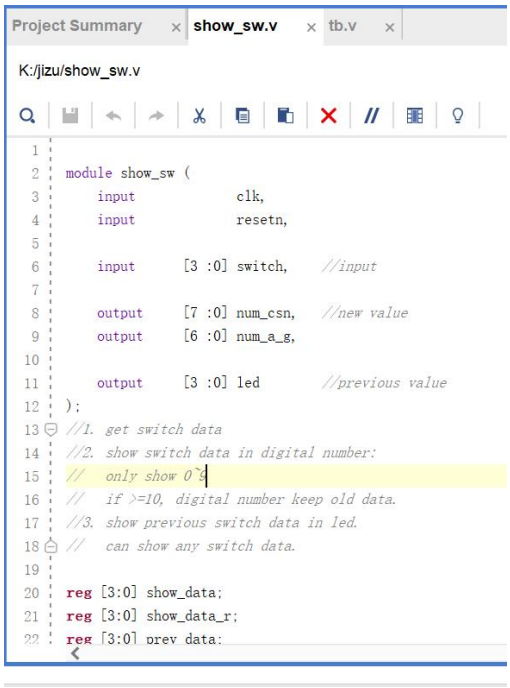
・进行仿真测试，生成异步 RAM，如下图所示



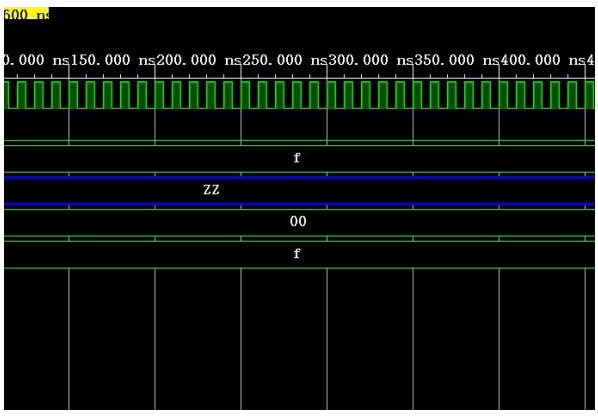


（三）实践任务三：数字逻辑电路的设计与调试

·添加 show_sw. v 文件、tb 文件和 show_sw. xdc 文件

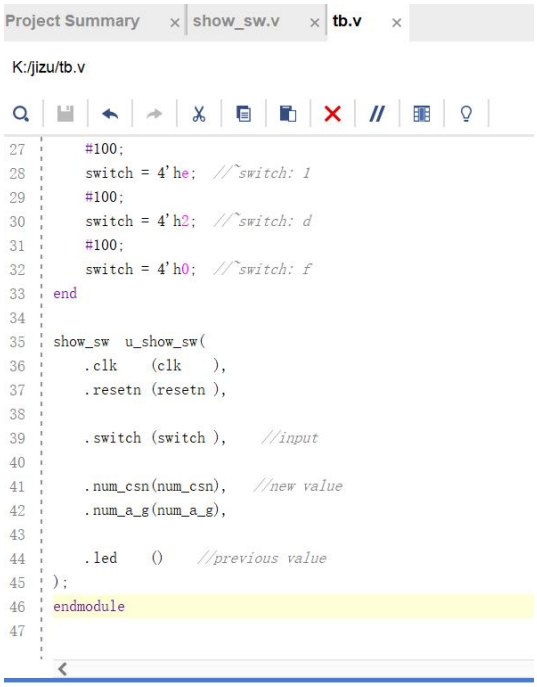


问题一：波形为“Z”



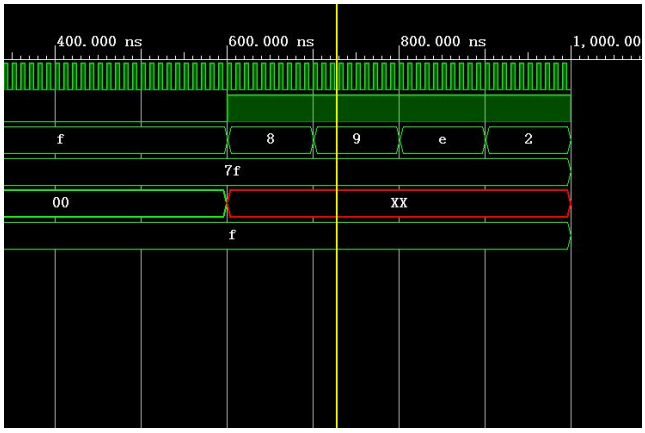
问题现象：在仿真波形中观察到"Z"（高阻态）错误。

查阅得知，此类问题通常由以下两种情况引发：
• 代码中声明为 wire 类型的信号未被任何驱动源赋值
• 模块实例化时存在信号连接错误导致端口悬空
当前问题定位：
通过代码审查发现，在 show_num 模块的实例化过程中存在连接错误：将输出信号 num_csn 误接到未定义的 num_scn 端口。
解决方法：应保持信号名称一致，连接到 num_csn 端口



修正后仿真波形中的高阻态消失，信号传输恢复正常。此案例表明模块接口连接的准确性对电路功能实现至关重要，信号名称的微小差异都可能导致功能异常。

· 进行仿真测试，如下图所示
问题二：波形出现不定态（X）



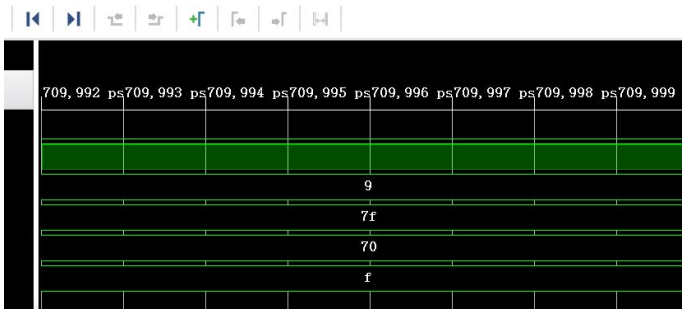
问题现象：仿真波形中出现 不定态（X），表明信号未被正确初始化或存在冲突驱动。
查阅得知，该问题通常由以下两种原因导致：
• reg 型变量未被赋值：声明为 reg 但未初始化或未在逻辑中赋值。

• 多驱动冲突：同一 reg 变量在多个 always 块或逻辑中被赋值，导致竞争条件。
当前问题定位：

```
22  reg [3:0] prev_data;
23
24  //new value
25  always @(posedge clk)
26  begin
27  //    show_data    <= `switch;
28  end
29
30  always @(posedge clk)
31  begin
32      show_data_r = show_data;
33  end
```

代码中 show_data 被声明为 reg 型，但第 27 行被注释掉，导致该信号未被驱动。
由于 reg 默认值为 X，仿真时会呈现不定态。
解决方法 ：取消注释并正确赋值

问题三：仿真卡死（波形停止）



问题现象：仿真在某个时刻（switch=9 时）完全停止，但仿真器仍在运行
查阅得知，RTL 中存在组合逻辑环路
当前问题定位：show_sw 模块对 switch=6 的输入未编码，导致输出悬空形成环路

```
//keep unchange if show_dtaa>=10
wire [6:0] keep_a_g;
assign     keep_a_g = num_a_g + nxt_a_g;

assign nxt_a_g = show_data==4'd0 ? 7'b1111110 :   //0
                 show_data==4'd1 ? 7'b0110000 :   //1
                 show_data==4'd2 ? 7'b1101101 :   //2
                 show_data==4'd3 ? 7'b1111001 :   //3
                 show_data==4'd4 ? 7'b0110011 :   //4
                 show_data==4'd5 ? 7'b1011011 :   //5
                 show_data==4'd7 ? 7'b1110000 :   //7
                 show_data==4'd8 ? 7'b1111111 :   //8
                 show_data==4'd9 ? 7'b1111011 :   //9
                                   keep_a_g   ;
endmodule
//-------------------------{digital number}end-----------------
```

补全代码逻辑：

```
wire [6:0] keep_a_g;
assign     keep_a_g = num_a_g + nxt_a_g;

assign nxt_a_g = show_data==4'd0 ? 7'b1111110 :   //0
                 show_data==4'd1 ? 7'b0110000 :   //1
                 show_data==4'd2 ? 7'b1101101 :   //2
                 show_data==4'd3 ? 7'b1111001 :   //3
                 show_data==4'd4 ? 7'b0110011 :   //4
                 show_data==4'd5 ? 7'b1011011 :   //5
                 show_data==4'd6 ? 7'b1011111 :   //6
                 show_data==4'd7 ? 7'b1110000 :   //7
                 show_data==4'd8 ? 7'b1111111 :   //8
                 show_data==4'd9 ? 7'b1111011 :   //9
                                   keep_a_g   ;
endmodule
//-------------------------{digital number}end----------------------
```

问题四：LED 状态保持异常（越沿采样）
问题现象：LED 显示未能保持前一次状态，出现随机跳变
根本原因：在时序逻辑中错误使用阻塞赋值(=)
当前问题定位：show_sw 模块中的寄存器赋值方式错误

```
always @(posedge clk)
begin
    show_data_r = show_data;
end
//previous value
always @(posedge clk)
begin
```

修改代码逻辑：

```
29
30  always @(posedge clk)
31  begin
32      show_data_r <= show_data;
33  end
34  //previous value
```

·生成比特流文件，上实验箱进行验证
问题五：数码管异常熄灭（功能 bug）
问题现象：当拨码开关输入 10~15（4'b1010~4'b1111）时，数码管异常熄灭
根本原因：段选信号错误叠加导致非法编码
当前问题定位：错误叠加

```
//keep unchange if show_dtaa>=10
wire [6:0] keep_a_g;
assign     keep_a_g = num_a_g + nxt_a_g;
```

修改代码逻辑：

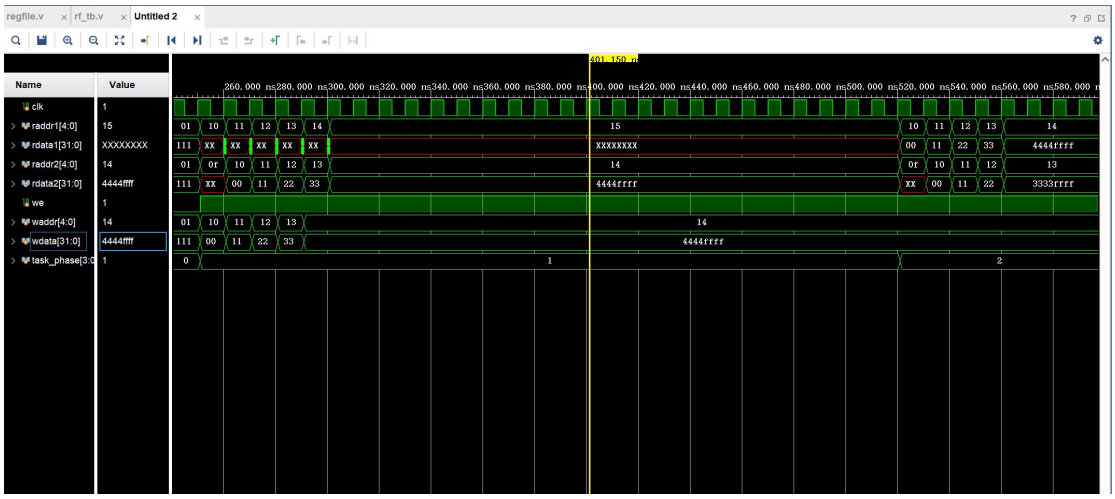
```
// 方案 1：完全禁用叠加
assign nxt_a_g = show_data==4'd0 ? 7'b1111110 :  // 0
                 // ...其他数字编码...
                 show_data==4'd9 ? 7'b1111011 :  // 9
                                   num_a_g;      // 保持原值

// 方案 2：安全叠加
assign keep_a_g = num_a_g | nxt_a_g;  // 按位或代替加法
```

四、实验结果分析

（一）实践任务一：寄存器堆仿真

• 仿真结果 simulation 波形图



1. 写入功能验证

在初始阶段，通过 we=1 能够成功向寄存器 1 写入数据 32'h1111ffff，并且在时钟上沿触发时可正确读取到新写入值，说明单个寄存器的写入和读取功能正常。

在连续写入阶段，从 t=250ns 开始，依次向寄存器 16 - 20 写入不同数据，每个寄存器都能正确接收并存储相应数据，表明连续写入功能正常。

2. 读取功能验证

在写入操作后，通过 raddr1 和 raddr2 能正确读取相邻寄存器的值，如写入 0x11 后，raddr1 能读取到新写入的 0x11 对应值，raddr2 能读取到前一个寄存器 0x10 的值，说明读取逻辑可按预期获取相邻寄存器数据。

在仅读取阶段（t=470ns 后），寄存器 16 - 20 存储的数据能被稳定读取，未出现数据丢失或错误读取的情况，表明寄存器在写入数据后，数据存储稳定，读取功能可靠。

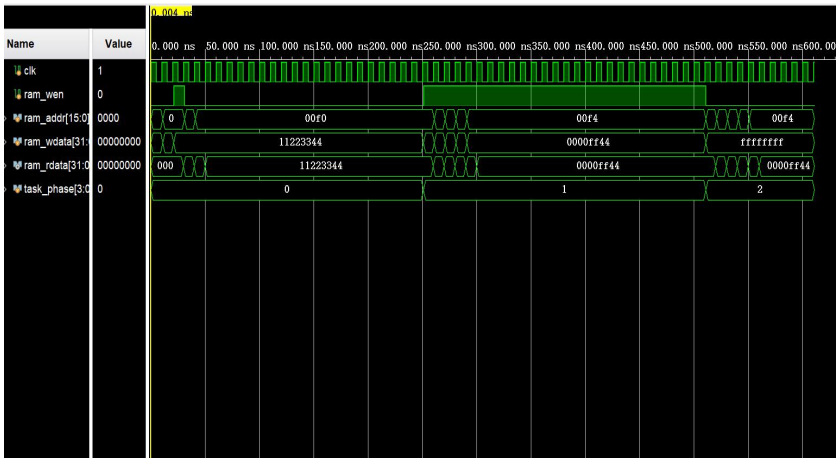
3. 总结

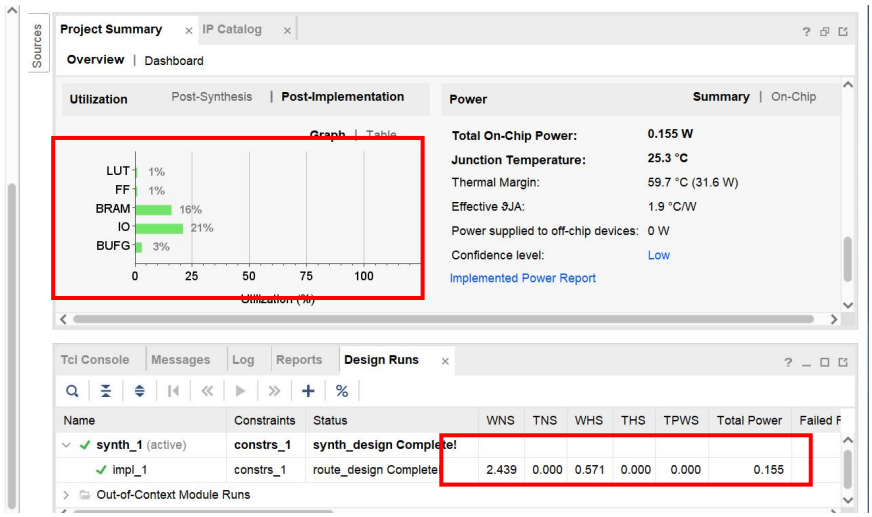
该波形图所涉及的寄存器模块，在写入和读取操作方面表现正常，能够按照预期的逻辑实现数据的写入、存储和读取，功能具备正确性和稳定性。

（二）实践任务二：同步 RAM 和异步 RAM 仿真、综合与实现

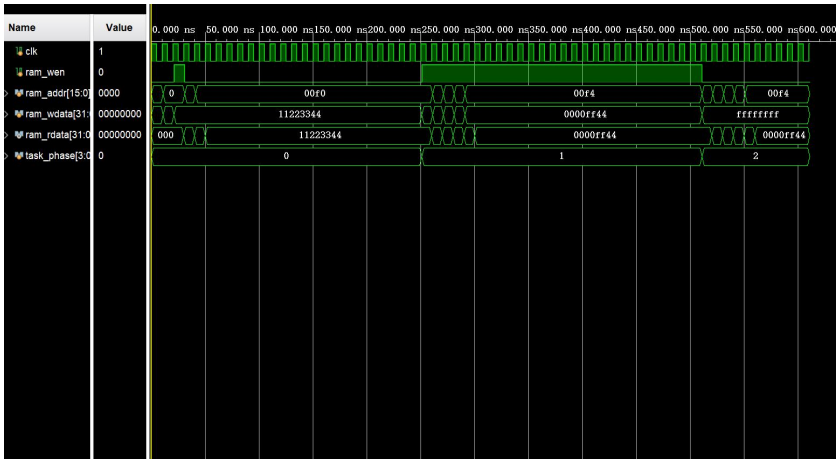
• 仿真结果 simulation 波形图

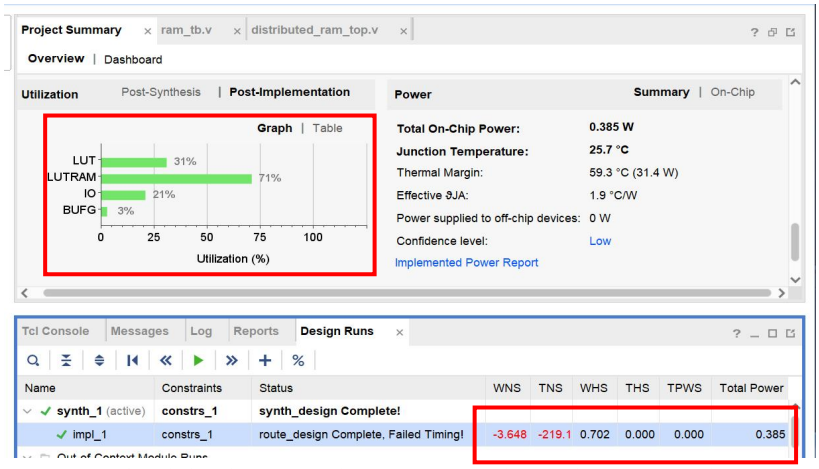
同步：


Name
Value
clk
ram_wen
ram_addr[15:0]
ram_wdata[31:
ram_rdata[31:0]
task_phase[3:0]
0.000 ns
50.000 ns
100.000 ns
150.000 ns
200.000 ns
250.000 ns
300.000 ns
350.000 ns
400.000 ns
450.000 ns
500.000 ns
550.000 ns
600.000 ns
00f0
00f4
11223344
0000ff44
ffffffff
0000ff44


Project Summary
IP Catalog
Overview | Dashboard
Utilization
Post-Synthesis | Post-Implementation
Graph | Table
LUT 1%
FF 1%
BRAM 16%
IO 21%
BUFG 3%
0
25
50
75
100
Power
Summary | On-Chip
Total On-Chip Power: 0.155 W
Junction Temperature: 25.3 °C
Thermal Margin: 59.7 °C (31.6 W)
Effective ϑJA: 1.9 °C/W
Power supplied to off-chip devices: 0 W
Confidence level: Low
Implemented Power Report
Tcl Console
Messages
Log
Reports
Design Runs
Name
Constraints
Status
WNS
TNS
WHS
THS
TPWS
Total Power
synth_1 (active)
constrs_1
synth_design Complete!
impl_1
constrs_1
route_design Complete
2.439
0.000
0.571
0.000
0.000
0.155
Out-of-Context Module Runs

异步：


Name
Value
clk
ram_wen
ram_addr[15:0]
ram_wdata[31:
ram_rdata[31:0]
task_phase[3:0]
0.000 ns
50.000 ns
100.000 ns
150.000 ns
200.000 ns
250.000 ns
300.000 ns
350.000 ns
400.000 ns
450.000 ns
500.000 ns
550.000 ns
600.000 ns
00f0
00f4
11223344
0000ff44
ffffffff
0000ff44



1. 写入行为:在初始写入和连续写入测试中,同步 RAM 和异步 RAM 的写入操作均依赖时钟沿 ，写入行为相似。

2. 读取行为：差异明显。同步 RAM 读取操作需等待时钟沿，存在一个时钟周期的读取延迟；而异 RAM 读取操作立即响应地址变化，无需等待时钟沿，无读取延迟。

3. 同步 RAM 主要依赖专用 BRAM 资源，对 LUT/FF 资源占用少；异步 RAM 使用分布式 RAM（LUTRAM），大量消耗 LUT 资源，不占用 BRAM 资源 。

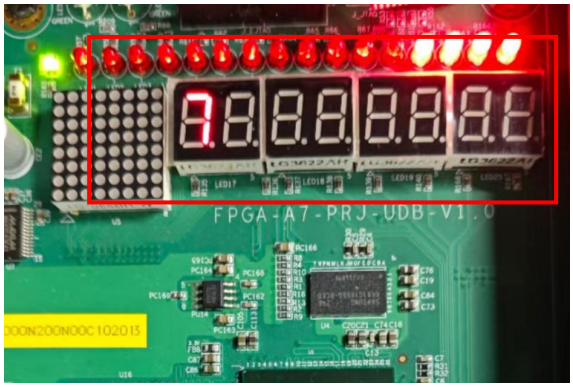
4. 同步 RAM 时序表现良好，能完全满足时序要求；异步 RAM 存在严重的建立时间违规问题，时序不收敛。

5. 若系统对时序要求严格，且有充足的 BRAM 资源，同步 RAM 更合适；若对读取响应速度要求极高，且 LUT 资源充裕，能接受一定时序问题，异步 RAM 可作为备选方案。 但总体而言，同步 RAM 在时序可靠性上优势显著。

（三）实践任务三：数字逻辑电路的设计与调试

- 实验箱结果

如图所示，开始时拨码开关处于 0 位，所有指示灯熄灭，数码管显示 0。当拨码开关切换至 1 时，指示灯保持前次状态（此时仍为全灭），数码管更新显示 1。拨码开关再拨为 3，则数码管显示 3，LED 灯显示 1。



其他情况经验证也均符合预期。

五、总结感想

任务一：寄存器堆仿真

思考：为什么寄存器堆要设计成“两读一写”？

寄存器堆采用“两读一写”结构是 CPU 设计中的常见方案，主要原因包括：

•支持多操作数指令：许多指令需要同时读取两个寄存器的值进行计算，因此需要两个独立的读端口。

•写后读冲突避免：单写端口可以确保在同一个时钟周期内不会发生多个写操作冲突，避免数据竞争问题。

•硬件资源优化：增加读端口的成本较低，而增加写端口需要更复杂的冲突检测和仲裁逻辑，影响时序和面积。

•符合 RISC 架构需求：RISC 指令集通常采用“两读一写”模式，这种设计能高效支持此类指令。

实验收获：通过仿真验证了寄存器堆的读写功能，理解了多端口设计对 CPU 数据通路的重要性。

任务二：同步 RAM 和异步 RAM 仿真

同步 RAM 与异步 RAM 的特点与区别：

特性	同步 RAM	异步 RAM
时钟依赖	读写操作均需时钟沿触发	读写操作直接响应地址/控制信号变化
读取延迟	需等待下一个时钟周期输出数据	立即输出数据（无时钟延迟）
时序可靠性	时序严格，易满足建立/保持时间	可能因路径延迟导致时序违规
资源占用	使用专用 BRAM,节省 LUT/FF	使用分布式 RAM（LUTRAM），消耗大量 LUT
适用场景	高频、时序敏感系统（如 CPU Cache）	低频率或对延迟敏感的小容量存储

实验收获：同步 RAM 更适合现代 FPGA 设计，因其时序可控且资源利用率高。异步 RAM 的即时读取特性在某些场景（如组合逻辑接口）有优势，但需谨慎处理时序问题。

任务三：数字逻辑电路调试
Bug 发现与修复过程：
1. Bug 1：波形高阻态（"Z"）
问题现象：仿真波形中 num_csn 信号显示为高阻态（"Z"）。
原因分析：检查发现 show_num 模块实例化时，误将输出端口 num_csn 连接到未定义的信号 num_scn，导致信号悬空。
修复方法：修正实例化连接，确保端口名称一致：

2. Bug 2：波形不定态（"X"）
问题现象：show_data 信号显示为不定态（"X"）。
原因分析：show_data 为 reg 型变量，但未在逻辑中赋值（代码中被注释）。
修复方法：取消注释并补充赋值逻辑（如 always @(*) show_data = switch;）。
验证结果：信号初始化为确定值，波形恢复正常。

3. Bug 3：仿真卡死（波形停止）
问题现象：仿真在 switch=9 时停止运行。
原因分析：组合逻辑中存在未处理的 switch=6 分支，导致输出悬空形成环路。
修复方法：补全所有输入状态的分支逻辑。
验证结果：仿真全程无卡顿，波形连续。

4. Bug 4：LED 状态异常（越沿采样）
问题现象：LED 显示随机跳变，未保持前次状态。
原因分析：时序逻辑中错误使用阻塞赋值（=），导致竞争条件。
修复方法：改用非阻塞赋值（<=）：
验证结果：LED 状态稳定更新，符合预期。

5. Bug 5：数码管异常熄灭（功能错误）
问题现象：输入 10~15 时数码管熄灭。
原因分析：段选信号错误叠加（如直接相加）导致非法编码。
修复方法：采用“安全叠加”策略（按位或操作）：
验证结果：数码管在 10~15 时保持前次显示值，无异常熄灭。

Vivado 调试经验总结：
1.先通过仿真定位波形异常（如"X"、"Z"、卡死），再上板验证，节省硬件调试时间。
2.输入→逻辑→输出逐级检查：确认输入信号（如 switch）是否正确传递；检查组合逻辑（如 always @(*)）是否覆盖所有分支；验证时序逻辑（如 always @(posedge clk)）是否使用非阻塞赋值。

3.使用 Vivado 的 Waveform 窗口观察信号跳变沿和稳态值，标记关键信号（如时钟、复位、使能）以快速定位问题，关注综合报告的“Critical Warnings”（如未连接端口、时序违规）。

4.初始化所有寄存器，避免 reg 型变量默认值为"X"；避免组合环路，确保所有 if/case 分支均有赋值。区分赋值方式，组合逻辑用阻塞赋值（=），时序逻辑用非阻塞赋值（<=）。

实验收获：

本次实验深入理解了数字逻辑电路的设计与调试方法，如寄存器堆的多端口设计是 CPU 高效执行指令的基础，同步/异步存储器的选择需权衡时序、资源和应用场景，Vivado 调试的核心是“分模块验证+波形分析”，良好的代码习惯（如初始化、避免组合环路）能显著减少错误，硬件设计需兼顾功能正确性和时序收敛性，仿真与实测结合是关键。

硬件调试需耐心与严谨，微小的代码差异可能导致功能失效，而规范的代码习惯能显著降低调试难度。