打地鼠

专用集成电路设计方法期末 *PJ*

课程: INFO130094.01
汇报人：王俊崴（22302016002）
学院 · 专业：软件学院 · 软件工程专业

2025 年 6 月 9 日

目录

1 设计计划

本项目的目标是在 ASIC（或 FPGA 平台）上实现一个**打地鼠游戏（Whac-A-Mole）**。该研究不仅为数字电路设计提供了实践案例，更为硬件开发学习者搭建了一个深入理解 ASIC/FPGA 设计方法与游戏逻辑硬件实现的平台。研究采用**自顶向下方法（Top-down Design Approach）**，从系统级规范出发，逐步分解为可实现的硬件模块，最终完成电路实现。对于非硬件专业背景的学习者而言，本项目通过完整的硬件开发生命周期实践，帮助掌握 FPGA 在数字系统中的典型应用，并熟练运用 Verilog 等硬件描述语言实现复杂控制逻辑。

通过打地鼠游戏的硬件实现，本研究重点探讨 FPGA 在交互式系统中的设计方法，特别是基于 Verilog 的实时控制逻辑与用户交互机制实现。项目特别注重**软硬件协同设计（Hardware-Software Co-design）**理念，既支持通过虚拟接口进行 FPGA 功能验证，也提供实体按键、LED 等外设的直接交互方案，从而全面提升硬件系统的设计灵活性与实践价值。

在实现过程中，项目将依托现代 EDA（电子设计自动化，**E**lectronic **D**esign **A**utomation）工具链，包括 Vivado 等商用工具及 Icarus Verilog、GTKWave 等开源工具，完成从**硬件设计（design）**到功能**仿真（simulation）**的全流程开发。为确保设计质量，项目将采用专业方法进行逻辑综合、时序约束与布局布线优化，切实提升 ASIC 设计的工程实践能力。通过本项目的完整实践，开发者不仅能深入理解数字系统的工作原理，更能系统掌握 FPGA 编程、RTL 设计、时序分析等核心技能，为后续复杂硬件系统开发奠定坚实基础。

1.1 设计要求

根据 PJ 说明文档的设计要求总结了一些面对**硬件（hardware）**和**逻辑（logic）**的**基础要求**：

1.1.1 硬件要求

- **8 个 LED 灯**：代表游戏中的“地鼠”。每次会随机亮起一个 LED，玩家需按下对应按键进行击打。
- **三位数数码管显示器**：用于实时显示玩家的**得分**。
- **8 个按键**：分别对应 8 个 LED 灯。玩家需要按下与点亮的 LED 灯相对应的按键才能得分。
- **开始按键**：用于开始或停止游戏。按下“开始”键后游戏开始，再次按下可暂停或重置游戏。应该也可以有一个“复位”按键。

1.1.2 基础游戏逻辑要求

- **游戏开始**：当按下“开始”键时，所有 LED 熄灭，分数清零，游戏**开始**。
- **地鼠出现**：每一轮会随机点亮一个 LED，玩家有 6 秒时间按下对应的按键。也就是说每一轮间隔最多六秒钟，看玩家能多快按下相应的按钮才能继续到下一轮。
  - 如果玩家在 **6 秒内**按下正确的按键，则得 1 分。
  - 如果玩家**未在 6 秒内**按下，得分为 0。

- **回合完成**：当 8 个 LED 灯都以随机顺序被点亮一次后，该轮结束。进入下一轮时，地鼠出现的**速度**和**频率**将加快。
- **游戏重置**：游戏过程中再次按下“开始”键将重置分数并开始**新一轮**游戏。

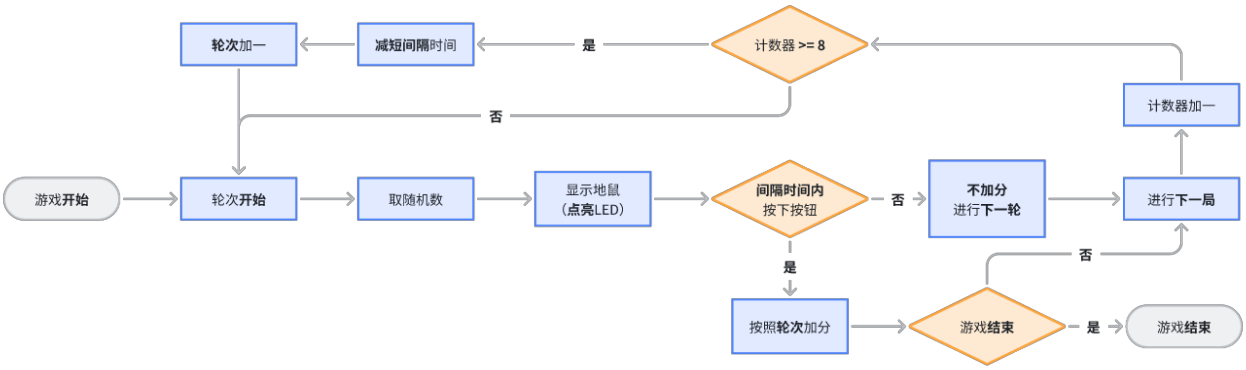


图 1: 基础游戏流程示意图

打地鼠游戏的核心逻辑主要基于三个关键计数器：**间隔时间计数器**、**轮次计数器**以及**得分计数器**，并辅以一个辅助计数器用于监测 LED 点亮次数。游戏设定每轮包含 8 次 LED 随机点亮机会，通过精确的计时控制实现难度递进。

游戏初始设置 6 秒的 LED 点亮间隔时间，并在每轮结束后递减 2 秒，以此实现游戏节奏的逐步加快。同时，得分倍率与轮次正相关，每完成一轮得分倍率增加 1 分。这种双重难度提升机制（时间间隔缩短和得分倍率增加）有效增强了游戏的挑战性和可玩性。

由于间隔时间从 6 秒起始并按每轮 2 秒递减，系统最多支持 3 轮游戏流程——当进入第 4 轮时，间隔时间将归零导致游戏无法继续。在此规则框架下，若玩家在每轮都能准确击中全部 8 个 LED 目标，理论最高得分可达 48 分（$8 \times (1 + 2 + 3) = 48$）。该设计在保证游戏趣味性的同时，也为玩家成绩评估提供了量化标准。

1.1.3 扩展实现（未来）

- **4x4 LED 矩阵与 4x4 按键矩阵**：在基础设计的 8 个 LED 与按键的基础上，本项目进一步扩展为使用 4x4 的 LED 阵列和对应的 4x4 按键阵列。这种设计不仅增加了游戏的复杂度和趣味性，也更贴近真实的“打地鼠”体验。每次会在 16 个 LED 中随机点亮一个或多个地鼠位置，玩家需迅速按下对应位置的按键完成击打。
- **软硬件协同设计**：本项目支持两种交互方式——通过 USB 虚拟组件接口进行软件端操作，或直接通过实体 FPGA 硬件设备操作。这种设计体现了软硬件协同的理念。在开发过程中，逻辑部分可在软件仿真环境中快速测试与迭代，而在物理部署阶段则可以实现真实按键与 LED 的交互体验，从而提高整体开发效率与系统灵活性。

1.2 设计思路

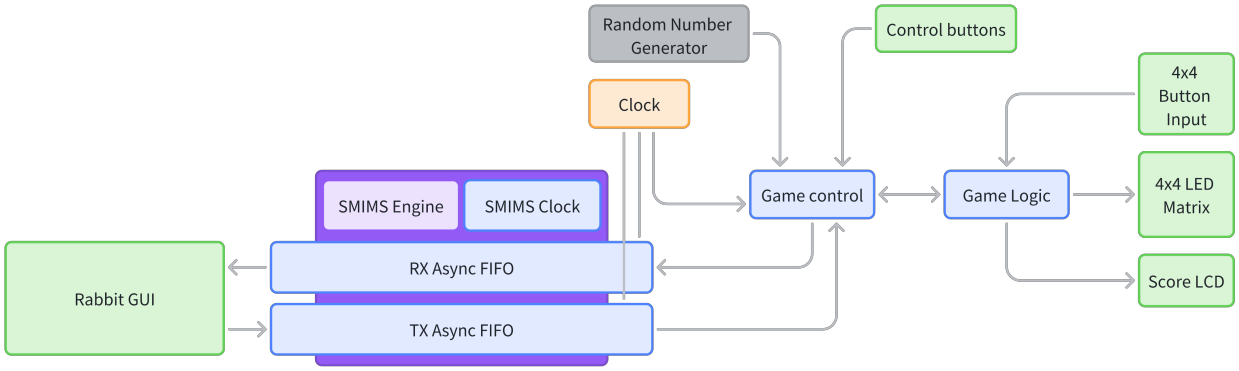
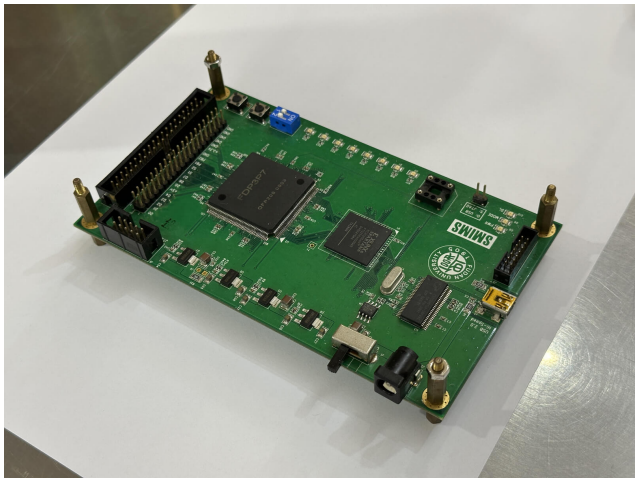
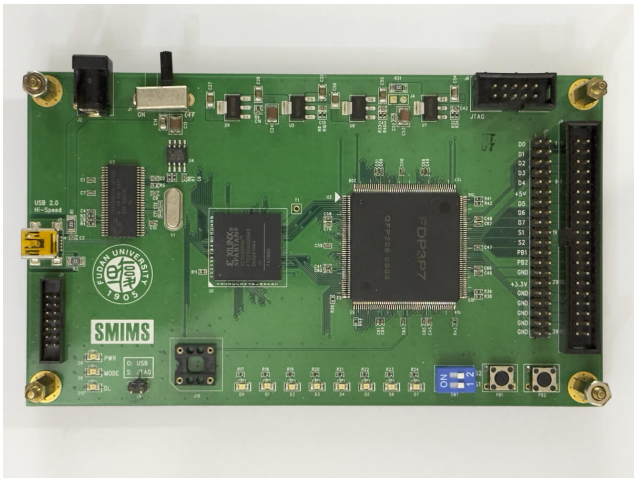


图 2: 基于 FDP37P 板子设计的顶层架构图。

1.3 设计方案与工具环境



图（一）



图（二）

图 3: FDP3P7 平台硬件示意图

讨论设计方案与工具环境前我们先介绍一下试验的**硬件平台**。本试验使用了**软硬件协同**的设计方法，允许通过 USB 接口或物理设备进行**交互**，从而增强了开发的灵活性与可用性。FDE 板子是复旦大学自研的一款 FPGA 教学与实验开发板，搭载 FDP3P7 FPGA 和 Xilinx Spartan XC3S200AN 芯片。该开发版上其实没有非常丰富的外设因为它可以通过一个 USB 链接与一台电脑通讯来实现虚拟部件（Virtual Components）。在外设方面，该 FDP 提供了丰富的 GPIO 管脚，LED 指示灯、按键和 USB 控制器，便于进行各类硬件实验与功能拓展。同时，它支持 SMIMS 驱动提供的虚拟组件接口，可通过 USB 与主机软件进行通信，实现无需物理接线的虚拟交互。这一特性使开发者可以在软件界面中模拟真实外设，显著提升开发效率和灵活性并降低调试和硬件部署的门槛。

1.3.1 设计平台

本项目在设计与验证阶段主要依赖**开源工具链**，而未使用 Vivado 等商业软件。具体流程如下：

首先，我们在 **VSCode** 中编写 Verilog 设计文件，并使用 **Icarus Verilog**（命令行工具 `iverilog`）对源代码进行编译。随后，通过编写 Verilog Testbench，将编译生成的可执行文件交由 **VVP** 模拟器运行，完成功能验证。整个仿真过程中，使用 **GTKWave** 对输出波形进行查看与调试，以确保设计在逻辑和时序层面均符合预期。

在完成 RTL 级功能验证后，进入综合阶段。首先，使用 **Synopsys Design Compiler（DC 综合）**将已验证的 Verilog 源代码综合成门级网表（RTL netlist）。接着，将 DC 输出的 RTL 网表输入到定制化综合框架 **UFDE+** 中，依次执行导入（import）、技术映射（map）、打包（pack）、布局布线（route）以及生成位流（generate bitstream）等操作，从而获得最终可用于 FPGA 或 ASIC 流片的比特流文件。

此外，为了实现更加完整的 ASIC 流程验证和可视化分析，本项目还使用了 **FDE2021** 工具。FDE2021 同样能够完成从 RTL 网表到版图的综合流程，但额外集成了静态时序分析（STA）模块以及芯片视图浏览功能，便于我们在设计后期进行时序收敛验证和版图检查。

综上所述，本项目在不同阶段分别采用了以下平台与工具：

- 设计与仿真阶段：**VSCode** + **Icarus Verilog**（iverilog）+ **VVP** 模拟器 + **GTKWave** 波形查看。
- RTL 综合阶段：**Synopsys Design Compiler（DC 综合）**将 Verilog 代码综合成 RTL 网表。
- 综合后端阶段：**UFDE+** 框架完成导入（import）、技术映射（map）、打包（pack）、布局布线（route）和位流生成（generate bitstream）。
- 版图与时序分析阶段：**FDE2021** 工具提供与 UFDE+ 类似的综合流程，并集成静态时序分析（STA）及芯片视图查看功能。

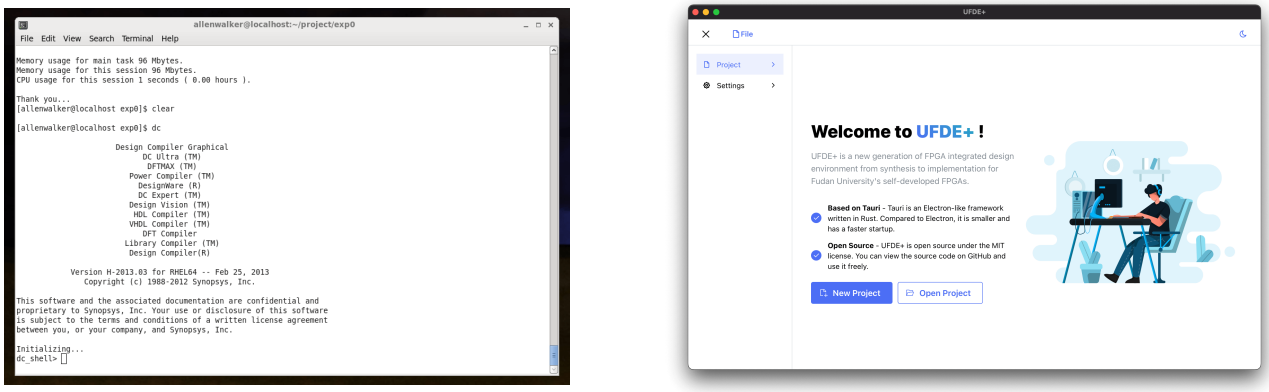


图 4: 左：Synopsys DC 编译器终端　右：UFDE+ 界面

这一设计平台的构建不仅提升了开发效率，也使得项目在不同层级上具备良好的可验证性和可移植性。

1.3.2 开发板和 EDA 工具使用说明

在本项目中，FDP3P7 开发板作为核心硬件平台，其功能通过 Xilinx Spartan FPGA 进行编程。为了完成设计，从编写 RTL 代码到数字综合，再到比特流文件的生成，整个流程都在 Xilinx 工具中完成。首先，使用 Xilinx ISE 进行 RTL 设计和合成，生成门级代码，接着使用 Xilinx Vivado 进行比特流文件的生成，最后将生成的比特流文件下载到 FDP3P7 开发板。开发板的接口，包括 GPIO、LED、按钮和 PID 开关，均通过 VeriComm 映射与 FPGA 进行交互，从而实现硬件逻辑的控制和数据交换。

1.3.3 设计流程图

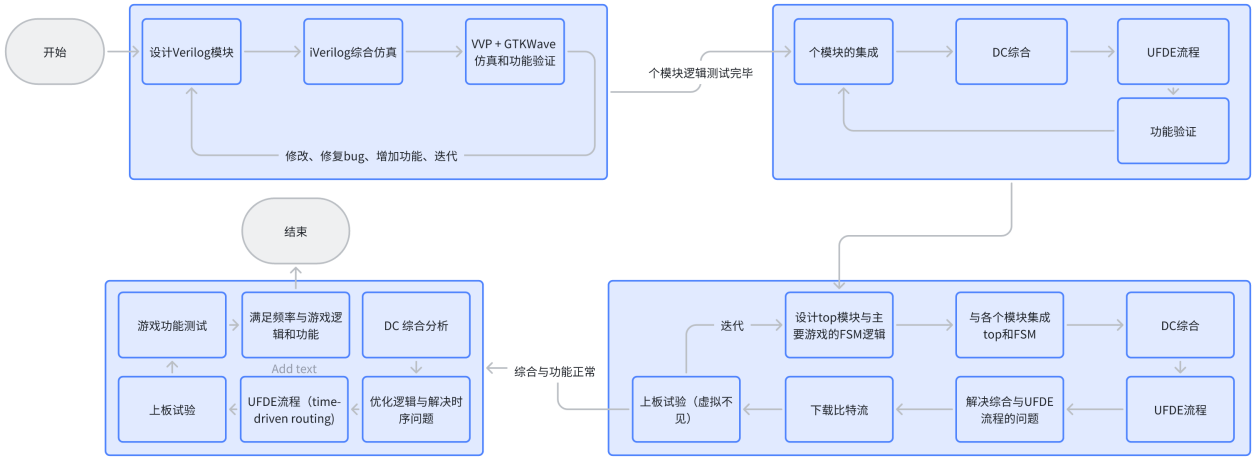


图 5: 打地鼠设计流程图

如图 5 所示，整个“打地鼠”FPGA 设计流程可以划分为四个阶段，并在每个阶段内部进行多次迭代，确保功能正确与时序收敛：

1. **单模块设计与仿真**
   - 从“设计 Verilog 模块”开始，编写每个子功能（如 PRNG、去重选择、定时计数、消抖、数码管解码等）的 RTL 代码；
   - 使用 iVerilog 做综合仿真，再通过 VVP+GTKWave 波形观察进行功能验证；
   - 根据仿真结果“修改、修复 bug、增加功能、迭代”，直至单模块逻辑测试通过，然后进入下一阶段。
2. **模块集成与综合验证**
   - 将已验证的各子模块集成起来，调用 Synopsys DC 进行逻辑综合；
   - 运行 UFDE（time-driven routing）流程完成布线；
   - 在功能验证平台上检查模块间的协同工作情况；
   - 如出现综合不通过或功能异常，返回“模块集成”阶段继续迭代。
3. **顶层 FSM 设计与板级验证**
   - 在所有子模块集成稳定后，编写 top 模块实现游戏主 FSM 逻辑；
   - 与各子模块一起做 DC 综合、UFDE 路由，并解决综合或布线过程中出现的问题；
   - 下载比特流到仿真平台（虚拟 FPGA）或试验板上，进行“上板试验（虚拟或真实）”；
   - 根据板级测试结果迭代完善 FSM 状态转移与接口逻辑。

4. **系统功能与时序优化**
   - 在真实 FPGA 板卡上进行完整的“游戏功能测试”，验证得分统计、倒计时、关卡升级等核心逻辑；
   - 检查系统运行频率及静态时序裕量，若不满足设计目标，则通过综合报告进行逻辑优化、约束调整，或再次 UFDE 路由；
   - 循环上述测试与优化，直至所有功能和时序指标均满足要求，流程结束。

通过“单模块 → 模块集成 → 顶层集成 → 系统验证”的多层次闭环迭代，既保证了各功能模块的可靠性，又最终实现了整体系统的高效、稳定运行。

2 设计实现

2.1 框图介绍

本节通过系统级框图，展示“打地鼠”游戏各模块之间的关系及数据流向。整个系统由按键消抖、伪随机数生成、去重选择、定时计数、解码驱动和顶层 FSM 六大功能模块组成，如图 6 所示。

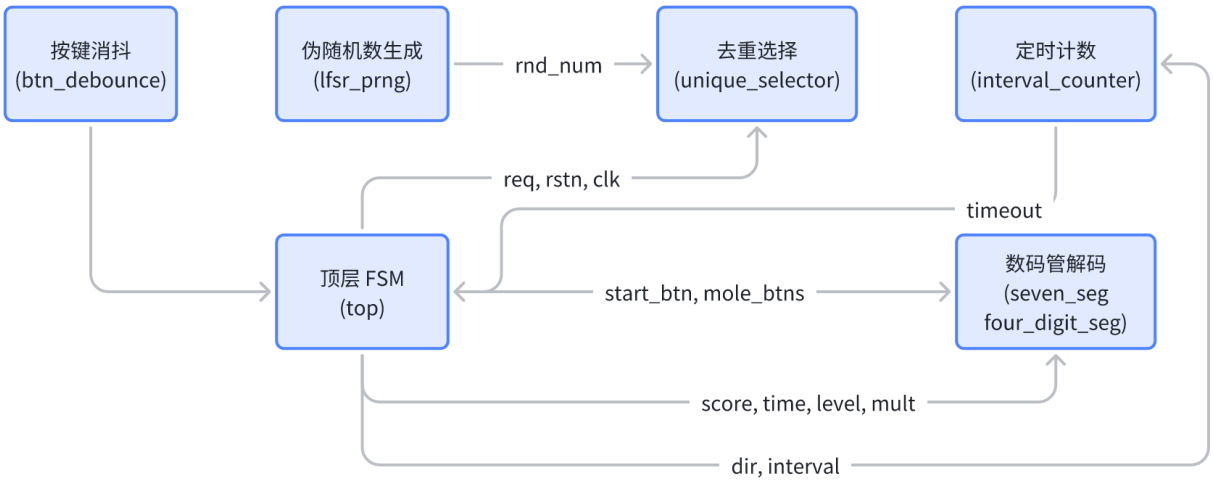


图 6: “打地鼠”系统总体框图

图中各模块功能简介如下：

- **按键消抖（`btn_debounce`）**：对异步按键输入进行同步和滤抖，输出稳定的“开始”以及“地鼠”按键信号，保证后续逻辑不会因颤动产生误触发。
- **伪随机数生成（`lfsr_prng`）**：基于线性反馈移位寄存器（LFSR）算法，以固定或可配置种子生成伪随机数序列，为地鼠位置选择提供随机源。
- **去重选择（`unique_selector`）**：利用伪随机数生成器输出，并结合内部掩码 FSM，确保每轮游戏中每个 LED 仅被点亮一次，输出当前选中位置及完成标志。

- **定时计数（`interval_counter`）**：根据游戏设定的倒计时或正计时要求，生成秒级定时脉冲（`timeout`），并在到达终点后锁定状态，供 FSM 判断超时。
- **数码管解码（`seven_seg` & `four_digit_seg`）**：将游戏得分、剩余时间、当前关卡和倍率等数值译码并多路扫描，驱动七段数码管显示。
- **顶层 FSM（`top`）**：集成上述所有模块，按游戏流程管理状态转移：IDLE、REQ、WAIT_MOLE、WAIT_HIT、LEVEL_DONE 与 GAME_OVER，并根据玩家输入、定时与选灯模块信号更新分数、关卡与显示内容，完成游戏主逻辑控制。

2.2 各模块设计和验证

注意：部分模块的测试代码及测试演示录屏可在 `experiments/*` 目录下找到。若存在多个版本，通常选择数字最大的版本。在各模块说明部分的末尾，也会标注相应的验证程序、视频或截图位置。下面开始介绍游戏主要实现模块的设计、使用说明及功能验证工作：

2.2.1 Makefile（小模块测试脚本）

在本项目中，`Makefile` 用于自动化小模块的测试和编译过程。它帮助简化了从源代码到可执行文件的构建流程，尤其是在模块的测试和验证阶段。通过执行简单的命令，用户可以轻松编译模块、运行测试并生成必要的报告和结果。具体而言，`Makefile` 中定义了各个模块的编译规则，包括源文件依赖关系、编译器设置以及清理工作（如删除临时文件）。此外，`Makefile` 还管理了测试流程，通过执行测试脚本，确保每个小模块按预期功能工作。通过使用该工具，开发人员能够高效地进行模块级别的验证和调试，确保最终实现符合预期的功能要求。

2.2.2 伪随机生成器 (PRNG, `lfsr_prng` 模块)

伪随机数生成器（**PRNG**，**P**seudo-**R**andom **N**umber Generator）是一种通过确定性算法生成数字序列的机制，这些数字看似随机，但实际上是由一个初始的**种子值（seed）**决定的。由于其生成过程完全确定，因此相同的种子会生成相同的序列。

伪随机数生成器广泛应用于多个领域，如模拟、加密、游戏开发、统计学等。在游戏领域，PRNG 也扮演着重要角色。例如，在本研究中的打地鼠游戏中，我们希望地鼠（或 LED 灯）能够随机出现或亮起。因此，必须实现一个具有随机性的数字生成器，用来决定每一轮哪个 LED 会亮起，从而为玩家提供更好的游戏体验。

尽管这些应用中的随机数看似随机，实际生成过程是完全**可预测的**，只要知道初始的种子值。例如，基于线性反馈移位寄存器（LFSR）的伪随机数生成器能够通过固定的种子值生成一系列的随机数。伪随机数生成器的特点如下：

- **确定性**：给定相同的种子，PRNG 每次生成的序列是**相同的**。
- **周期性**：PRNG 的输出序列是有限的，最终会回到起始位置，**形成一个循环**。
- **速度快**：PRNG 通常不需要依赖外部的随机物理过程，因此生成**速度非常快**，适用于需要大量随机数的场景。

PRNG 的实现通常包括以下几种方法：

- **线性反馈移位寄存器（LFSR）[1]**：通过移位寄存器和反馈机制生成伪随机数，适用于

硬件实现，速度快且简单。

- **异或移位算法**：通过异或和移位操作快速生成随机数。

为了简化项目的复杂性，我们认为 LFSR 方法是最简单且直接的设计方案，因为它只需要一个种子（可以是固定值）作为输入，便能直接生成输出。这样的设计不仅减轻了在设计、测试和集成过程中的工作量，还提高了项目的可靠性。以下展示了两个实验，用以验证伪随机数的正确性，以及改变种子的效果与正确性：第一个实验使用默认的种子（16'hABCD）生成随机数，第二个实验将种子参数修改为 16'hACE1，并观察生成器的效果。

```
5     // Testbench signals
6     reg clk;
7     reg rst;
8     wire [15:0] random;
9
10    // Instantiate the LFSR PRNG module
11    lfsr_prng uut (
12        .clk(clk),
13        .rst(rst),
14        .use_ext_seed(1'b0),
15        .random(random)
16    );
17    // lfsr_prng #(.INIT_SEED(16'hACE1)) uut (
18    //     .clk(clk),
19    //     .rst(rst),
20    //     .use_ext_seed(1'b0),
21    //     .random(random)
22    // );
23
```

图 7: PRNG 模块的 testbench 代码（默认种植）

```
junweiwang@Juns-MacBook-Pro:/Volumes/Jimmy1/S6/workspace/asic/...
make TEST_MODULE=tb_lfsr_prng
iverilog -g2012 -o obj/tb_lfsr_prng.vvp \
                v_src/lfsr_prng.v test/tb_lfsr_prng.v
Running simulation for tb_lfsr_prng...
vvp obj/tb_lfsr_prng.vvp
VCD info: dumpfile tb_lfsr_prng.vcd opened for output.
At time                    0, random = abcd
At time                15000, random = 579a
At time                25000, random = af34
At time                35000, random = 5e69
At time                45000, random = bcd2
At time                55000, random = 79a4
At time                65000, random = f348
At time                75000, random = e691
At time                85000, random = cd23
At time                95000, random = 9a46
At time               105000, random = 348c
test/tb_lfsr_prng.v:49: $finish called at 110000 (1ps)
/Volumes/J/S6/w/a/project                    base 11:02:41
```

图 8: LFSR PRNG 测试仿真结果（实验一）

```
10    // Instantiate the LFSR PRNG module
11    // lfsr_prng uut (
12    //     .clk(clk),
13    //     .rst(rst),
14    //     .use_ext_seed(1'b0),
15    //     .random(random)
16    // );
17    lfsr_prng #(.INIT_SEED(16'hACE1)) uut (
18        .clk(clk),
19        .rst(rst),
20        .use_ext_seed(1'b0),
21        .random(random)
22    );
23
24    // Generate clock signal
25    always begin
26        #5 clk = ~clk; // Toggle clock every 5ns
27    end
28
```

图 9: PRNG 模块的 testbench 代码（参数种植）

```
junweiwang@Juns-MacBook-Pro:/Volumes/Jimmy1/S6/workspace/asic/...
make TEST_MODULE=tb_lfsr_prng
iverilog -g2012 -o obj/tb_lfsr_prng.vvp \
                v_src/lfsr_prng.v test/tb_lfsr_prng.v
Running simulation for tb_lfsr_prng...
vvp obj/tb_lfsr_prng.vvp
VCD info: dumpfile tb_lfsr_prng.vcd opened for output.
At time                    0, random = ace1
At time                15000, random = 59c3
At time                25000, random = b387
At time                35000, random = 670f
At time                45000, random = ce1e
At time                55000, random = 9c3c
At time                65000, random = 3879
At time                75000, random = 70f2
At time                85000, random = e1e4
At time                95000, random = c3c8
At time               105000, random = 8791
test/tb_lfsr_prng.v:49: $finish called at 110000 (1ps)
/Volumes/J/S6/w/a/project                    base 11:46:37
```

图 10: LFSR PRNG 测试仿真结果（实验二）

通过该实验可以观察到种植决定了最初始生成的随机数，然后后面的输出都具有一定的随机性，因此验证了我们随机生成器的正确性。此实验也验证了通过参数来修改种植。

2.2.3 地鼠产卵器 (Mole Spawner, unique_selector 模块)

为实现游戏规则要求—每轮游戏中每个 LED 灯必须且仅亮起一**次**（例如 16 个 LED 对应 16 个回合，并且全部回合进行完以后每个 LED 亮过一**次**）—该系统称为**地鼠产卵器 (Mole Spawner)** 机制，配合状态跟踪逻辑：由于基本伪随机数生成器无法避免重复取值，该机制通过实时记录已亮 LED 的状态信息（位掩码），动态调整后续点亮选择，从而严格保证每个 LED 在当轮游戏中仅被激活一次，同时维持游戏过程的随机性和公平性。为了满足“每个 LED 灯在一轮游戏中只亮起一次”的需求，这部分提出一个 unique_selector 模块来实现地鼠产卵器，并采用了以下**设计思路**：

- **参数化接口**：通过参数 n 指定可选槽数为 2^n（对应 LED 数量），通过参数 `WIDTH` 指定外部伪**随机数宽度**；
  - 因为我们的伪随机数生成器的输出数值宽度是 16 位，和该模块需要的不一致，我们需要对他进行一下特殊处理（把多余的比特省略掉）；
  - 一般来说参数 n 是小于 `WIDTH` 但会导致在搜索中会浪费许多时钟周期；
- **输入信号**：外部模块在需要选灯时拉高 `req`（request）信号；由上层模块驱动的伪随机数生成器输出 `rnd_num` 给该模块提供一个有效的随机值；
- **状态跟踪掩码**：使用 $selected\_mask[2^n-1:0]$ 寄存器记录每个编号是否已被选用，“0”表示未选、“1”表示已选；
- **地鼠产卵器的有限状态机（FSM）**：
  - 在时钟上升沿或复位信号作用下**初始化**所有寄存器，包括清零 `selected_mask` 和控制标志 `searching`、`done`、`selected` 等；
  - 当 `req` 上升沿到来时，进入“搜索”状态（`searching=1`），拉低 `done`；
  - 在搜索状态下，每个时钟周期从 `rnd_num[n-1:0]` 中提取低 n 位作为候选编号 `potential_number`；
  - 检查该编号在**掩码（mask）**中对应位是否为“0”：
    - 若为“0”，则将其位置设成“1”，并将编号写入 `selected_number`，拉高 `selected` 和 `done`，退出搜索；
    - 否则保持搜索，继续下一时钟周期**重新采样**；
  - 搜索完成后，复位 `selected` 和 `done`，待下一次 `req`（转入休闲状态或 waiting）；
- 通过监测 `selected_mask`，当所有位均为“1”时，拉高 `all_selected`，指示当前回合已完成对所有 LED 的点亮（这个信号也可以给系统提示此轮次结束）；
- 以下是模块的参数和输入输出信号的说明：
  - **参数**：
    - n: 此参数决定了用于选择的位数。可以选择的唯一数字数量为 2^n，其中 n 指定了所选数字的位数。默认情况下，$n = 4$，允许从 16 个唯一数字中选择；
    - `WIDTH`: 此参数指定输入随机数的位宽，预计为外部伪随机数生成器（PRNG）输出。默认情况下，`WIDTH = 16`；
  - **模块输入**：
    - `clk`: 时钟信号。该信号用于同步模块的操作；
    - `rst`: 重置或复位信号。当该信号为高时，会重置模块的内部状态；
    - `req`: 请求信号，用于启动选择新唯一数字的过程。当该信号为高时，模块开始选择数字；
    - `rnd_num`: 一个 16 位输入，提供外部生成的伪随机数。该数字的低 n 位用于确定下一个潜在的唯一数字；
  - **模块输出**：

  - `selected_number`: 一个寄存器输出，保存所选的唯一数字，值的范围为 $[0, 2^n - 1]$；
  - `selected`: 一个脉冲信号，当选择一个新的唯一数字时，该信号在一个时钟周期内为高电平；
  - `done`: 一个信号，当选择过程完成时，该信号在一个时钟周期内为高电平。它表示一个唯一数字已经被选择并存储；
  - `all_selected`: 一个信号，当所有可能的数字至少被选择一次时，该信号变为高电平；
- **内部状态和功能**:
  - 该模块作为一个有限状态机（FSM）运行，其主要功能可以描述如下：
  - `selected_mask`: 这是一个位掩码，用于跟踪哪些数字已经被选择。掩码中第 i 位为高电平时，表示数字 i 已经被选择过。
  - `potential_number`: 这是一个寄存器，存储候选数字（从输入的 `rnd_num` 的低 n 位提取）。
  - `searching`: 这是一个标志，指示模块是否正在选择数字，或者处于空闲状态，等待下一个请求。

- **重置行为（reset）**：
  - 当 `rst` 信号为高时，所有内部寄存器都会被重置，包括 `selected_mask`、`selected_number` 和控制信号。FSM 返回到空闲状态。
- **选择过程**:
  1. 在接收到高 `req` 信号后，FSM 进入“searching”状态。
  2. 在“searching”状态中，模块读取输入 `rnd_num` 的低 n 位，并将其存储在 `potential_number` 中。
  3. 模块通过检查 `selected_mask` 来判断该数字是否已经被选择过。如果该数字未被选择，则接受它：
     - 在 `selected_mask` 中对应的位被设置为高电平。
     - `selected_number` 被更新为新选择的数字。
     - `selected` 信号在一个周期内为高电平。
     - `done` 信号在一个周期内为高电平。
     - FSM 退出“searching”状态。
  4. 如果该数字已经被选择过（即 `selected_mask` 中对应的位为高电平），FSM 继续搜索一个唯一数字。
  5. 一旦所有可能的数字都被选择（即 `selected_mask` 中的所有位都被设置为高电平），`all_selected` 信号被设置为高电平。
- **空闲行为（idle）**：
  - 如果没有 `req` 信号，模块将保持在空闲状态，且不会选择任何数字。

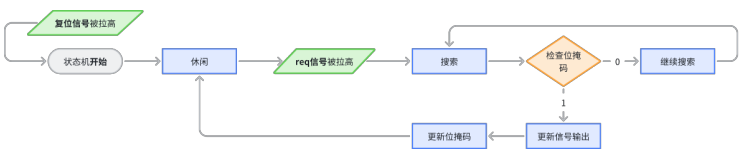


图 11: `unique_selector` 的流程图（flow chart）

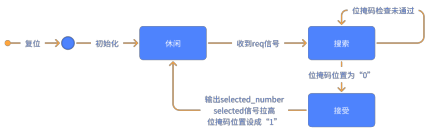


图 12: `unique_selector` 的状态机（FSM）

上述内容给出了 `unique_selector` 模块的流程图及其有限状

态机（FSM）。该设计借助掩码机制，在充分利用外部随机数的同时，严格避免重复选择，从而保证每轮游戏中每个 LED 仅被点亮一次。下面将介绍模块的功能验证与仿真方案。

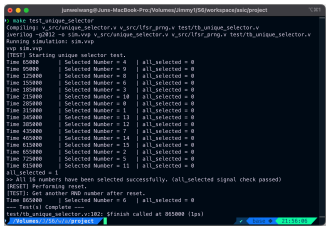


图 13: 地鼠产卵器测试仿真结果

验证重点如下（设模块参数为 n）：在一次完整运行周期内，模块应输出 2^n 个互不重复的随机编号 x，且满足 $x \in [0, 2^n - 1]$。因此，测试平台（以 $n = 4$ 为例）的思路为：

1. **连续激励**：驱动生成器运行 16 次（可能不仅 16 个时钟周期），依次产生全部 2^4 个候选编号；
2. **复位与请求信号处理**：在仿真中检查 `rst` 和 `req` 信号的响应是否符合设计预期——模块应在复位后加载种子并清空掩码，同时仅在 `req` 为高时输出新编号；
3. **唯一性与计数**：确认这 16 个编号互不相同，且总数确为 16；
4. **状态信号检查**：验证指示信号 `selected`（或 `done`）与 `all_selected` 是否在相应时刻正确拉高 / 拉低，反映出编号选取及全集完成的状态。

- **Reset handling** –仿真开始后立即将 `rst` 置高一个时钟周期。在此期间，`unique_selector` 应清空内部掩码并重新加载种子，使 `selected`、`done` 与 `all\_selected` 全部为 0；当 `rst` 重新拉低时，设计应处于干净的初始状态。
- **Picking all 16 unique numbers** –设 `n = 4`，测试平台连续产生 16 个单周期请求脉冲 (`req`)。每次脉冲后等待 `done` 拉高并打印当前 `selected\_number`；目标是观察 0–15 共 16 个唯一值各出现一次，且 `selected`、`done` 仅在新编号就绪时有效；最后一次选取时，`all\_selected` 必须置高。
- **Checking the `all_selected` flag** –第 16 次选取完成后立即读取 `all\_selected`。若其为 1，则输出 “all 16 numbers selected” 通过；否则报告错误，提示存在遗漏或重复。
- **Recovery after a second reset** –为进一步验证，可再次拉高 `rst` 清除历史掩码。复位结束后发出一次新的 `req` 脉冲：`all\_selected` 应复位为 0，模块需重新开始编号序列并输出一个新的数值（仍落在 0–15），证明其可在下一轮游戏前干净复位并正常工作。

通过以上步骤以及名为 `testbench` 的脚本，我们可以全面验证 `unique_selector` 模块在随机性、去重机制和状态指示等方面的正确性。验证结果如图 13 所示，相关实现见主目录下的 `tests/tb_unique_selector.tb` 文件。从图中可以看出，在包含 16 个可能位置的设置下，该模块能够随机选择一个位置，并在同一运行周期内保证不重复，只有在复位后才会重新选择。同时，该设计还提供了额外的信号，用于指示所有位置已被选完，从而通知游戏当前回合已结束。

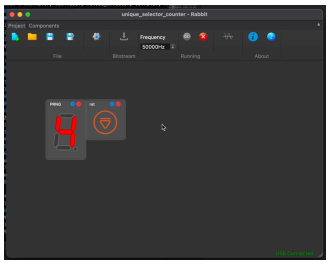


图 14: 地鼠产卵器试验截图

2.2.4 定时计数器（`interval_counter` 模块）

本模块用于在给定的秒数区间内，上 / 下计数并在到达终点时产生一个宽度为一个时钟周期的脉冲 `timeout`，同时停止后续计数，直到复位信号重新拉低。其主要设计思路和验证方案如下。

模块接口

- **输入**：
  - clk：系统时钟；
  - rst_n：低有效同步复位，拉低时清零所有状态并重新开始；
  - interval（[2:0]）：目标秒数，取值范围 0…7；
  - dir：计数方向，1 = 向上计数，0 = 向下计数。
- **输出**：
  - count（[2:0]）：当前计数值；
  - timeout：到达终点时的一个时钟周期脉冲，高电平表示计时完成。

内部实现

- 使用本地参数 DIV_W = $clog2(‘CLK_FREQ) 计算分频计数器位宽；
- 寄存器 div_cnt 实现时钟分频，当其计数到 ‘CLK_FREQ-1 时产生一拍 tick 并复位自身；
- 寄存器 done 用于锁存“已到终点”状态，避免重复触发；
- 在同步复位期间（!rst_n）：
  - div_cnt、timeout 清零；
  - 若 dir=1，则 count 置 0，否则置为 interval；
  - done 复位为 0。
- 在正常时钟下，每个时钟周期先更新 div_cnt，当 tick 为高时：
  - 若 done=0，比较当前 count 与终点值（interval 或 0）：
    - 达到终点则拉高 timeout，并置 done=1；
    - 未到终点则根据 dir 增 / 减 count；
  - 若 done=1，则保持 timeout=1，不再更新 count。

功能验证 使用名为 tb_interval_counter 的 testbench 驱动如下场景：

1. **复位验证**：初始化时拉低 rst_n，检查 count、timeout、done 均清零且 count 处于正确初值；
2. **向上计数**（dir=1）：设 interval=5，观察 count 每秒加 1，至 5 时产生 timeout 脉冲且停止；
3. **向下计数**（dir=0）：设 interval=3，count 初值为 3，每秒减 1，至 0 时产生 timeout 脉冲且停止；
4. **再复位恢复**：在 timeout 后拉低再拉高 rst_n，检查模块可重新计数并再次触发 timeout。

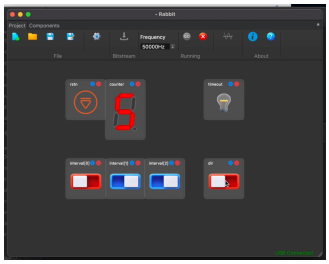


图 15: 定时计数器 interval_counter 的仿真波形

验证结果表明，模块在各种配置下均能按设计产生精确的秒级脉冲，并在到达终点后保持 timeout，直到复位后重新工作，满足游戏计时需求。演示视频与模块测试代码可在 experiments/counter_3 目录下找到。

2.2.5 七段显示解码器（seven_seg）

本模块将 4 位十六进制数字（0…F）译码为七段显示的 7 位控制信号，接口及功能如下：

模块接口

- **输入**：
  - hex（[3:0]）：要显示的十六进制数字；
- **输出**：
  - seg（[6:0]）：对应七段的 a–g 段，**高电平点亮**。

内部实现 使用组合逻辑 always @(*) 和 case 语句，根据输入 hex 选择对应的 7 位编码：

- 4’h0…4’h9：显示“0”到“9”；
- 4’hA…4’hF：显示“A”、“b”、“C”、“d”、“E”、“F”；
- default：所有段灭（7’b0000000）。

功能验证 在 tb_seven_seg 测试平台中依次驱动 hex 从 0 到 F，观察 seg 输出与预期编码一致。验证波形如图 16 所示。

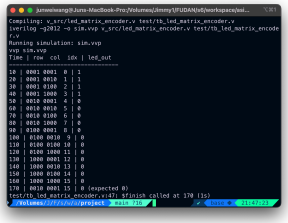


图 16: seven_seg 的仿真波形

2.2.6 四位数码管驱动（four_digit_seg）

本模块将 16 位二进制数值（0…9999）转换为四位十进制显示，通过多路扫描方式轮流驱动四位数码管。接口及功能如下：

模块接口

- **输入**：
  - clk：系统时钟（如 50 MHz）；
  - value（[15:0]）：要显示的十进制数值，范围 0…9999。
- **输出**：
  - seg（[6:0]）：当前位的七段编码，高电平点亮；
  - digit_en（[3:0]）：位选信号，**低电平有效**，指示哪一位数码管点亮。

内部实现

1. **定时分频**：计算扫描总频率 SCAN_FREQ = REFRESH_HZ × 4（本例 400 Hz），使用 scan_cnt 与常量 SCAN_MAX = $\frac{\text{CLK\_FREQ}}{\text{SCAN\_FREQ}}$ 分频产生 ∼400 Hz 的 digit_idx 更新脉冲。
2. **十进制提取**：组合 always @(*) 逻辑依次计算千、百、十、个位：
   - 通过比较减法与 case 语句，依次减去 1000、100、10 得到各位数字 d3–d0。
3. **多路扫描**：
   - 每次 digit_idx 递增，选择当前要显示的 current_nibble；
   - 将该半字节送入 seven_seg 模块译码，得到 decoded；
   - 根据 digit_idx 产生对应的 digit_en（低电平有效）。

设计迭代过程 在实现 four_digit_seg 模块的过程中，我们先后经历了三次主要方案迭代：

1. **初版：双重循环法**
   使用”移位加 3”双重循环（double-dabble）算法，在函数中对输入逐位移位并在必要时加法修正，以提取各十

进制位。但此法生成的大量组合逻辑不被 DC 编译器识别，导致映射 (mapping) 失败。

```
function [15:0] bin2bcd;
    input [15:0] binary_in;
    integer i;
    reg [27:0] shift_reg;
    begin
        shift_reg = {12'd0, binary_in};
        for (i = 0; i < 16; i = i + 1) begin
            if (shift_reg[27:24] > 4) shift_reg[27:24] =
                shift_reg[27:24] + 3;
            if (shift_reg[23:20] > 4) shift_reg[23:20] =
                shift_reg[23:20] + 3;
            // … 同理对其他 BCD 位
            shift_reg = shift_reg << 1;
        end
        bin2bcd = shift_reg[27:12];
    end
endfunction
```

Listing 1: 双重循环法提取十进制位

2. **第二版：除法与取模法**
   直接用除法（/）和取模（%）运算计算各位数字，例如 `digit0 = value%10`。该方法在仿真中正确，但 DC 编译器因不支持硬件除法器而拒绝综合。

```
wire [3:0] digit0 =  value      % 10;        // ones
wire [3:0] digit1 = (value /  10) % 10;       // tens
wire [3:0] digit2 = (value / 100) % 10;       // hundreds
wire [3:0] digit3 = (value /1000) % 10;       // thousands
```

Listing 2: 除法与取模法提取十进制位

3. **最终版：组合比较与减法法**
   放弃循环与除法，仅使用一系列比较与减法，依次判断并扣除千、百、十位权值，最后直接将余数的低 4 位作为个位。该方法经过 DC 编译器综合验证，且性能满足 400 Hz 扫描需求。

图 17: 四位数 LCD 的实现演示

```
always @(*) begin
    // thousands
    if      (value >= 9000) d3 = 4'd9;
    else if (value >= 8000) d3 = 4'd8;
    // … 依次判断至 1000
    else                    d3 = 4'd0;
    case (d3)
        4'd1: rem1 = value - 1000;
        // … 依次处理至 9000
        default: rem1 = value;
    endcase
    // hundreds、tens 同理
    d0 = rem3[3:0];  // ones digit
end
```

Listing 3: 组合比较与减法法提取十进制位

2.2.7 按键消抖模块（btn_debounce）

本模块用于对异步的按键输入信号进行同步与抖动滤波，输出稳定的同步按键信号，接口及功能如下：

模块接口

- **参数**：
  - WIDTH：移位寄存器长度（采样深度），默认为 16，表示需要连续 WIDTH 个时钟周期的稳定输入才切换输出。
- **输入**：
  - clk：系统时钟；
  - rst_n：低有效复位（0 有效），复位时所有寄存器清零；
  - button_in：原始异步按键输入，可能含颤动噪声。
- **输出**：
  - button_out：消抖后的同步按键信号，高电平表示按键按下，低电平表示松开。

内部实现

1. **两级同步器**：使用两个触发器 (sync_ff1, sync_ff2) 级联对异步输入进行寄存器同步，消除亚稳态风险。
2. **移位寄存器滤波**：在每个时钟上升沿将同步后的信号移入 shift_reg，当寄存器所有位均为 1 时，判定按键稳定按下；当所有位均为 0 时，判定按键稳定松开；否则保持上一次输出状态。
3. **复位行为**：在 rst_n=0 时，sync_ff*、shift_reg 全部清零，并将 button_out 置 0。

功能验证 在 tb_btn_debounce 测试平台中：

- 生成不同频率和宽度的按键颤动脉冲序列，观察 sync_ff2 与 shift_reg 波形；
- 检查 button_out 仅在输入稳定高或稳定低达到 WIDTH 周期后才翻转；
- 在复位动作前后验证 button_out 的正确初始化与恢复。

2.2.8 顶层模块与有限状态机（top）

本模块集成了按钮消抖、伪随机数生成、去重选择、定时计数、数码管显示以及游戏主逻辑，通过有限状态机（FSM）实现“打地鼠”游戏流程。主要接口如下：

模块接口

- **输入**：系统时钟 clk，复位 rstn，开始按钮 start_btn，以及 8 个地鼠按钮 mole1_btn …mole8_btn（均带消抖）；
- **输出**：8 个地鼠 LED mole1_led…mole8_led，得分数码管（4 位）、秒表数码管、关卡和倍率数码管的段选与位选信号，以及游戏结束指示灯 game_over_led 与击中指示灯 hit_led。

FSM 状态定义

- IDLE：等待开始，按下“开始”按钮进入 REQ；

- REQ：向 unique_selector 发出选灯请求，进入 WAIT_MOLE；
- WAIT_MOLE：等待新的地鼠编号就绪，完成后进入 WAIT_HIT；
- WAIT_HIT：等待玩家击中或超时；若击中且未全部选完，回到 REQ；若击中且本轮结束，进入 LEVEL_DONE；若超时，同样决定下一状态；
- LEVEL_DONE：完成一轮后，根据剩余时间决定继续下一轮 (REQ) 或游戏结束 (GAME_OVER)；
- GAME_OVER：游戏结束状态，保持指示，直到再次按“开始”复位回 IDLE。

FSM 实现示例：

```
always @* begin
  next_state = state;
  us_req      = 1'b0;
  case (state)
    IDLE: if (start_btn_release) next_state = REQ;
    REQ: begin
      if (start_btn_release) next_state = IDLE;
      else begin us_req = 1'b1; next_state = WAIT_MOLE; end
    end
    WAIT_MOLE:
      if (start_btn_release) next_state = IDLE;
      else if (us_done) next_state = WAIT_HIT;
    WAIT_HIT:
      if (start_btn_release) next_state = IDLE;
      else if (|mole_btns_release || ic_timeout)
        next_state = us_all_selected ? LEVEL_DONE : REQ;
    LEVEL_DONE:
      if (start_btn_release) next_state = IDLE;
      else next_state = (time_interval>3'd2) ? REQ : GAME_OVER;
    GAME_OVER:
      if (start_btn_release) next_state = IDLE;
      else next_state = GAME_OVER;
  endcase
end
```

Listing 4: top 模块中 FSM 状态转移逻辑

输出与内部寄存器更新 在每个时钟上升沿，根据当前 state 更新：

- IDLE：重置得分、关卡、倍率和时间间隔；
- WAIT_HIT：若检测到正确击中，则累加 score_reg；
- LEVEL_DONE：若剩余时间允许，增加关卡和倍率，并减少时间；否则直接过渡到 GAME_OVER；
- GAME_OVER：保持当前数据显示，直至复位或重启。

功能验证 在 tb_top 测试平台中，按以下步骤验证：

1. **开始与复位**：模拟多次按下/松开开始按钮，检查是否正确进入/退出游戏；
2. **地鼠选择与显示**：确保每轮唯一地鼠按 LED 照亮并可被击中；
3. **计时与超时**：调整定时器，验证超时分支和得分逻辑；
4. **关卡与游戏结束**：连续完成多轮，检查关卡升级与最终游戏结束指示。

2.3 综合与实现

2.3.1 综合条件设定和结果分析

在本项目中，综合工具使用的是最新版本的 UFDE+，具体使用的工艺库为 `fdp3000k`，约束文件位于 `whackamole_fde2021/whackamole_ufde_cons.xml`。以下是该设计的详细资源统计数据和分析：

- **设计名称：** top
- **资源统计：**
  - GCLK 数量：1（占 GCLK 总资源的 25%）
  - GCLKIOB 数量：1（占 GCLKIOB 总资源的 25%）
  - IOB 数量：36（占 IOB 总资源的 25.35%）
  - SLICE 数量：314（占 SLICE 总资源的 10.03%）
  - Net 数量：672
- **使用的硬件平台：** fdp3000k

本次设计使用了基于时序驱动的布局模式（Timing Driven Mode），并通过多次放置迭代优化了成本，最终得到了较为理想的资源占用与性能平衡。综合过程中的资源使用情况如下：

(参考文件：`whackamole_ufde/place.log`)

- **资源使用比例：**
  - GCLK 使用比例：25%
  - GCLKIOB 使用比例：25%
  - IOB 使用比例：25.35%
  - SLICE(LUT0) 使用比例：10.03%

综合工具通过进行多个阶段的迭代优化，最终达成了低成本的布局，使得设计能够有效地利用硬件资源，并满足时序要求。测试和验证阶段通过这些设置，确保了设计的稳定性和高效性。

2.3.2 静态时序结果分析 （STA）

静态时序结果分析 → **S**tatic **T**iming Results **A**nalysis (STA)

以下是 FDE2021 软件提供的 STA 分析结果的一个小结，详细的报告可以去改目录查看 (`whackamole_fde2021/whackamole_fde_dc_sta.rpt`)。本节对设计文件 top 在 FDP3000K 器件上的静态时序分析结果进行总结，并提出优化建议。

时钟及频率概况

- 最小时钟周期：126.16 ns
- 最大可达频率：7.93 MHz

关键路径分析

- **最差上升沿路径（Setup）**

  - 源：全局时钟缓冲 gclk_buf
  - 目的：Slice252 中的触发器 FFX
  - 到达时间：126.16 ns
  - 要求时间：126.16 ns
  - 时序裕量：0.00 ns
- 其它上升沿路径裕量（部分）：
  - Path 1（FFY，Slice252）到达 126.12 ns，裕量 +0.04 ns
  - Path 2（FFX，Slice250）到达 126.09 ns，裕量 +0.06 ns
  - Path 4（FFX，Slice152）到达 126.07 ns，裕量 +0.09 ns
  - Path 9（FFY，Slice415）到达 126.02 ns，裕量 +0.14 ns
- 最差下降沿路径（Setup）
  - 源：组合逻辑输出 g
  - 目的：Slice308 中的触发器 FFY
  - 到达时间：9.08 ns
  - 要求时间：9.08 ns
  - 时序裕量：0.00 ns
- 其它下降沿路径裕量最高达 +1.93 ns

时序裕量分布

- 零裕量路径：最差上升沿路径、最差下降沿路径
- 正裕量路径：其余上升沿和下降沿路径（裕量范围 +0.04 ns 至 +1.93 ns）

优化建议

1. 精简全局时钟网络：减少缓冲链或使用器件专用低抖动时钟资源。
2. 缩短关键路径逻辑：将简单组合逻辑前移或增加流水级。
3. 寄存器重定位与重定时：引入额外寄存器，平衡逻辑深度。
4. 约束优化：对非关键路径添加假路径或多周期路径约束。
5. 物理布局指导：将关键逻辑单元靠近时钟源放置，减少布线延迟。

小结 当前设计在 7.93 MHz 下满足功能需求，但最差路径裕量为零，抗工艺 / 电压 / 温度变化能力有限。后续应按上述措施逐步优化，并在每次调整后重新进行 STA 验证。

3 FPGA 系统介绍、下载实现与调试测试流程

本节介绍 FPGA 系统的整体架构、比特流下载流程以及在 FPGA 板上进行功能调试和游戏测试的方法。

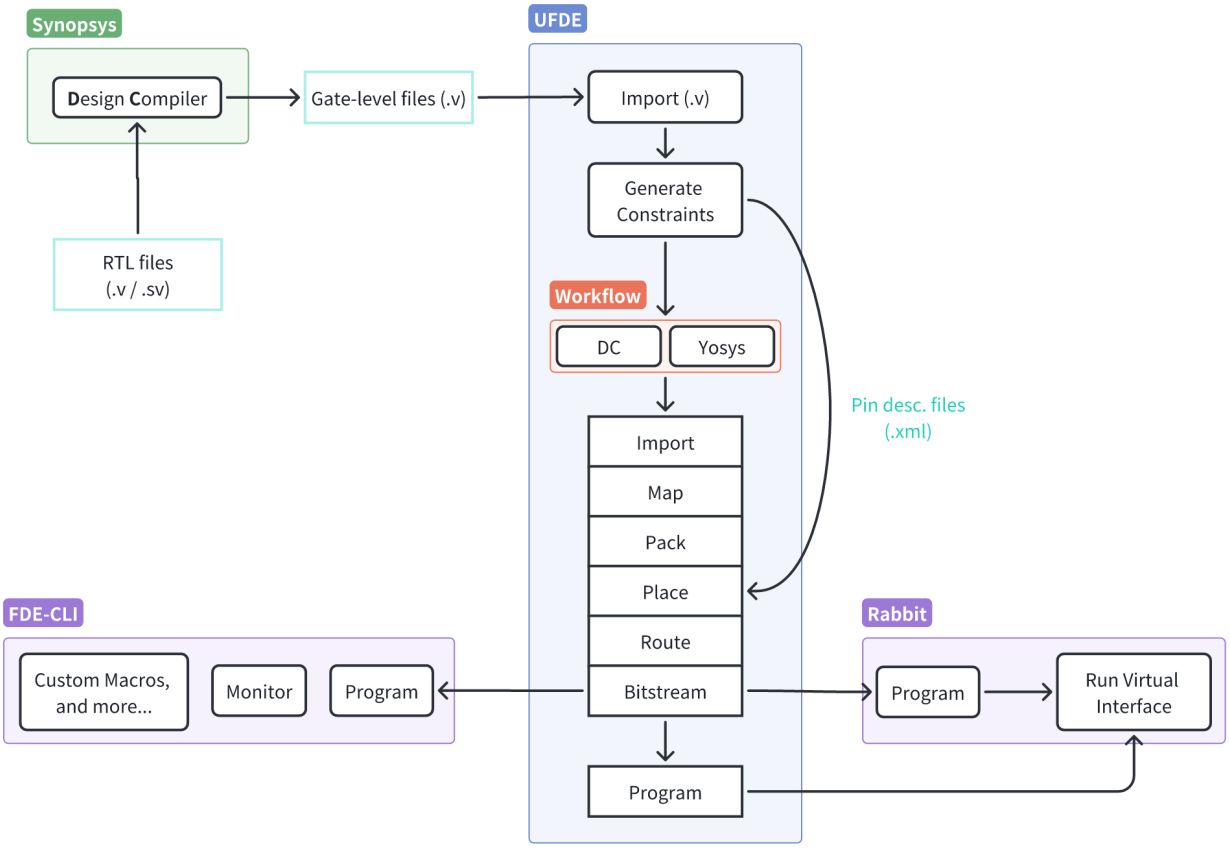


图 18: 综合与现在实现流程图

3.1 系统架构

本设计基于某型号 FPGA 开发板（如 FDP3P7），其主要硬件资源包括：

- 时钟管理单元：提供系统时钟（30 MHz）及各功能模块所需的分频时钟（最大虚拟时钟频率为 50 kHz，用于游戏逻辑、计时器等功能）。
- I/O 接口：按钮输入（Start、Mole1–Mole8）、LED 输出（Mole1–Mole8 指示灯）及四位七段数码管。
- USB 下载接口：用于下载比特流并进行在线调试，通过 UFDE、Rabbit 或 FDE2021 进行 USB 编程。

3.2 比特流生成与下载

1. **RTL 综合**：使用 Synopsys Design Compiler，将顶层 Verilog RTL 代码综合为网表文件（.v）。与 Vivado 的工作流程不同，FDP3P7 开发板使用 Synopsys 的 Design Compiler 进行 RTL 综合，不依赖于 Vivado Synthesis 工具。
2. **比特流生成**：通过 UFDE 或 FDE2021 工具进行后续的布局布线与比特流生成。此过程不依赖于 Vivado，而是利用这些工具对生成的网表进行优化，并产生适用于 FDP3P7 开发板的比特流文件。

3. **下载至 FPGA**：生成的比特流文件可以通过 USB 连接，使用 UFDE、Rabbit 或 FDE2021 进行下载，配置 FPGA 内部逻辑。此过程直接通过这些工具传输比特流，而不涉及 Vivado Programmer 或 Impact 工具。

3.3 在线调试与功能测试

下载完成后，采用以下步骤进行调试与测试：

1. **按键功能验证**：依次按下 Start 及各 Mole 按钮，观察对应 LED 是否正确响应，验证去抖模块和状态机逻辑。
2. **计时器与数码管显示**：检查数码管能否按照预期刷新显示剩余时间与得分，验证四位数码管驱动和计数器模块。
3. **完整游戏流程测试**：运行若干轮游戏，记录点击正确与错误、超时未点击等不同情况，确认游戏逻辑、得分计算与状态转换是否正确。
4. **性能分析**：在高按键频率和连续重试场景下，测试系统响应速度与时序稳定性，确保无死机或功能丢失。

本节内容为 FPGA 板级调试与验证流程的规范说明，可据此编写实验报告或测试规范文档。

3.4 游戏测试

以下图像记录了游戏基本功能的测试与验证：

- 游戏空闲状态
- 开始游戏
- 游戏结束
- 击中地鼠
- 进入下一关
- 获得最高分
- 未获得最高分（由于超时或按错按钮）
- 计时器到期示例

完整的功能验证演示视频可在 `demos/whackamole_final_demo.mp4` 文件中查看。

图 19: 游戏初始化界面

图 20: 游戏进行中按键响应演示

图 21: 游戏结束 LED 验证

图 22: 地鼠集中 LED 提示验证

图 23: 非满分的游戏结局验证

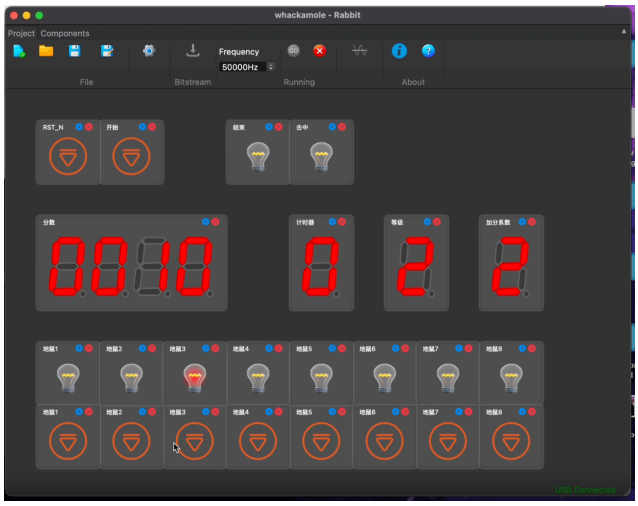


图 24: 时钟消耗截图

4 设计总结

4.1 自顶向下设计方法概述

自顶向下（Top-Down）设计方法是一种从系统整体出发，逐级分解到子系统和模块的设计思路。其核心思想是首先在系统级定义功能需求和架构，然后将系统划分为若干子模块，再分别对各子模块进行详细设计与实现，最后进行集成与验证。该方法具有以下优势：

- **明确总体目标**：在设计初期就确定系统功能和性能目标，保证设计方向一致。
- **分层次逐步细化**：通过逐级分解，将复杂系统分解为易于管理的小模块，提高设计可维护性。
- **模块化开发与复用**：各模块接口在设计初期定义，可并行开发并有利于后续复用。
- **设计可追溯性**：从需求到实现的每一级都清晰可追踪，便于验证与调试。
- **便于验证与优化**：可在不同抽象层次进行仿真与验证，并在高层和低层分别进行性能优化。

4.2 项目报告自顶向下应用评估

本报告在“设计计划”章节明确了系统功能需求和硬件平台要求，体现了在系统级别进行需求分析的思路；在“设计思路”与“设计方案与工具环境”中，将整体架构逐步分解为 PRNG、unique_selector、interval_counter、seven_seg、four_digit_seg、btn_debounce、top FSM 等子模块，符合自顶向下中先定义接口、再实现功能的流程；在“设计实现”部分，每个模块均给出了接口说明、实现原理及仿真验证结果，保证了设计的一致性和可追溯性。

在综合与实现阶段，报告使用多种 EDA 工具完成了从 RTL 综合到物理布局、布线和 STA 验证的全过程，体现了从高层逻辑设计到低层物理实现的层次化设计流程。此外，报告对综合条件和时序约束进行了针对性优化建议，符合自顶向下设计中在高层决策后，针对低层细节进行优化的特点。

尽管整体流程较为完整，但仍可在以下方面加强自顶向下设计方法的应用：

- 在需求分析中引入 UML 或 SysML 建模，增强需求与设计的可追溯性；
- 在模块划分阶段结合总线协议（如 AXI-lite）进行接口规范，提高模块移植性；
- 在验证阶段采用功能覆盖率度量，确保仿真测试的全面性；
- 提前编写 SDC 脚本并结合增量综合策略，优化多次综合与布局迭代；
- 引入形式化验证工具对关键控制逻辑进行等效性证明，提高设计可靠性。

4.3 演示截图

在本小节中展示项目在 FPGA 板上运行时的关键功能演示截图，包括游戏初始化、按键响应、计时器显示和胜利/失败提示等。

图 25: 游戏初始化界面

图 26: 游戏进行中按键响应演示

图 27: FDP3P7 平台硬件示意图

5 个人体会

通过本次项目的实践，我系统地学习了如何使用 Verilog 语言进行模块化设计开发。作为我第一次设计 FPGA 或 ASIC 系统，这对我来说是一次全新的挑战。在设计过程中，我首先将系统划分为多个功能独立的子模块，诸如状态机控制模块、按键去抖动模块、随机数生成模块、分数与时间显示模块等。每完成一个模块，我都使用仿真工具或开发板进行功能验证，确保其能够独立稳定地运行后再与其他模块进行集成。这种“自下而上”的设计与验证流程，不仅增强了我的逻辑思维能力，也让我在调试中积累了大量的实践经验。

作为一名非微电子专业的学生，我在初期对硬件电路的构建、调试流程感到十分陌生。例如，如何确定触发器类型、怎样避免时序竞争、模块之间如何传递控制信号等问题，起初都让我感到棘手。此外，由于缺乏配套的 Xilinx FPGA 实验板，我无法使用课程推荐的工具链进行开发与烧录，只能依托于王老师《FPGA 概念与应用》课程提供的 FDP3P7 FPGA 开发板来完成设计。这块开发板功能较为基础，部分资源受限，但也因此锻炼了我在受限条件下进行系统设计的能力。最终，我顺利完成了各模块的功能实现与集成，构建出了一个完整、可运行的“打地鼠”游戏系统。

在此项目中，我收获最深的是对软硬件编程逻辑差异的认知。软件编程以顺序执行为主，程序的执行流程清晰线性；而硬件描述语言（如 Verilog）则体现为“并行建模”：组合逻辑电路在同一个时钟周期内同时响应输入，而时序逻辑部分则需依赖时钟沿触发更新状态。因此，硬件电路的行为往往是“并发”的，多个信号变化可在同一时间内共同发生，这与传统的软件思维方式存在本质差异。

这种差异带来的挑战在于：硬件设计者不仅要关注功能实现本身，还必须深入理解时序、信号延迟、触发机制等电路运行细节。很多初学者容易陷入“程序代码等于执行顺序”的思维误区，而在 Verilog 中，代码的书写顺序并不代表执行顺序。若不能正确区分组合逻辑与时序逻辑，常会导致功能错误或不稳定行为。因此，我认为理解硬件编程需要从根本上转变思维方式，这对于习惯了软件开发的我而言，既是一次挑战，也是一种成长。

总的来说，此项目极大提升了我对数字电路系统的理解能力与工程实现能力。从 Verilog 编码、模块设计，到硬件验证与功能集成，每一步都充满了收获与反思，为我日后深入学习硬

件系统设计打下了坚实基础。

附：主要文件说明

```
Project_Root/
├── demos/
│   ├── whackamole_mid_demo0.mp4 (中期演示视频)
│   └── whackamole_final_demo.mp4 (最终实现演示视
│       频)
├── experiments (小模块集成试验与验证录屏)/
│   ├── counter/
│   ├── counter_2/
│   ├── counter_3/
│   ├── lcd_counter/
│   ├── prng/
│   ├── simple_couter/
│   ├── unique_selector_counter/
│   └── README.md
├── final_src (最终实现代码)/
│   ├── alib-52/
│   ├── result/
│   ├── v_src (最终设计文件)/
│   │   ├── btn_debounce.v
│   │   ├── defines.vh
│   │   ├── four_digit_seg.v
│   │   ├── interval_counter.v
│   │   ├── lfsr_prng.v
│   │   ├── seven_seg.v
│   │   ├── top.v
│   │   └── unique_selector.v
│   ├── work/
│   ├── command.log
│   ├── compile.tcl (综合脚本)
│   ├── default.svf
│   ├── DOCS.md
│   ├── filenames.log
│   └── output.log
├── images/
├── report (试验报告文档)/
└── test (小模块测试)/
    ├── tb_afifo.v
    ├── tb_button_matrix_decoder.v
    ├── tb_clock_generator.v
    ├── tb_led_matrix_encoder.v
    └── tb_lfsr_prng.v
```

```
    ├── tb_timer.v
    ├── tb_unique_selector.v
    └── tb_xor_shift_prng.v
├── v_src (小模块实现)/
    ├── afifo.v
    ├── button_matrix_decoder.v
    ├── clock_generator.v
    ├── defines.vh
    ├── four_digit_display.v
    ├── gopher_game.v
    ├── led_matrix_encoder.v
    ├── lfsr_prng.v
    ├── timer.v
    ├── top.v
    ├── unique_selector.v
    └── xor_shift_prng.v
├── whackamole_fde2021/
├── whackamole_rabbit/
├── whackamole_ufde/
├── .gitignore
├── 2025project 打地鼠.pdf (试验说明文档)
├── Makefile (小模块测试脚本)
└── README.md
```

参考文献

[1] Dan Gisselquist. *Generating pseudo-random numbers on an FPGA_2017*. Oct. 2017. URL: https://zipcpu.com/dsp/2017/10/27/lfsr.html.