华为杯

第八届中国研究生创“芯”大赛

验证自动化流程设计说明

作品名称：混沌逻辑-大模型多智能体自动数字 IC 前端设计器

团队名称：混沌逻辑

参赛队员：孙林涵，诸人豪，张煜

2025 年 6 月 19 日

目录

摘要

本软件通过设置项目管理、设计工程师和验证工程师三个 Agent，模仿一般的 IC 设计流程，使通用大语言模型（LLM）生成可靠的 RTL 代码。本文档将详细介绍软件各个 Agent 的设计，以及软件的运行流程。

1 介绍

近年来，随着大语言模型（Large Language Model, LLM）技术逐渐进入应用场景，越来越多的领域开始尝试使用 LLM 来辅助工作。数字前端设计作为一个复杂的工程领域，也开始探索如何利用 LLM 来提高设计效率和准确性。然而，与其他领域相比，本领域的 RTL 代码开源少、且开源代码缺乏高质量设计；HDL 语言与 LLM 大量学习的软件语言看似相似，实则在基础思路上存在着巨大差距。此外，和 LLM 已取得广泛应用的软件工程相比，IC 设计的容错率极低，对代码正确率要求高。因此，直接调用已有的通用 LLM 到数字前端设计中是极不合理的。

为了解决这些问题，本项目设计了一套由通用 LLM 驱动的 RTL 代码的生成与验证自动化软件，旨在提高 IC 设计的效率和准确性。

2 自动化系统设计

本项目的设计目标是，在使用市场上现有的通用 LLM 的同时，最大化地减少 LLM 的幻觉，按照用户预期快速、自动地完成初版设计。为此，我们仿照现有的 IC 设计流程设计了多个智能体（Agent），逐步展开用户的需求；并在 LLM 运行的各个阶段尽可能地使用 Synopsys 工具链对大模型的结果提出反馈，避免设计错误在工作流中传导。

2.1 Agent 设计

考虑到前面提到的问题，期望 LLM 在单轮对话中直接给出正确的 RTL 设计是不现实的。要确保大模型的输出正确且符合用户需求，则需要为 LLM 提供反馈，在多轮迭代后取得期望的输出。但是，LLM 的输出较慢（如本项目使用的 Deepseek API 仅能达到 40 Token/s），若依赖人类产生反馈信息，则将占用用户大量时间，背离了自动化的设计初衷。因此，本项目模仿一般的 IC 设计流程，设置了项目管理、设计工程师和验证工程师三个 Agent，每个 Agent 均具有单独的短期记忆、长期记忆，并根据其职责设计了对应的工具供其使用。LLM 在使用工具时，间接调用了 Synopsys 的 VCS 等工具进行语法检查、仿真等，输出的结果将自动地反馈给大模型，实现了一定程度的自动化。下面详细介绍各个 Agent 的设计。

2.1.1 项目工程师 Agent

项目工程师解读用户输入的需求，生成项目的技术规范（下文简称 Spec），并将其存储在文件中。当设计完成时，项目工程师还会根据验证工程师的验证报告，检查设计是否满足规范要求。

系统提示词：

```
你是混沌逻辑公司 IC 设计部门的项目管理员，你的团队中包括设计工程师和验证工程师。

你需要将客户的需求转换为规范的 Spec 以协助设计。当收到验证工程师的验证报告时，你需要检查其是否符合客户需求
不准定义验证工作的内容
不准定义工艺节点和时钟频率
```

用户提示词，根据用户输入生成完整 spec 时：

```
# 项目当前状态

等待项目管理员的 Spec

# 客户需求

{user_requirements}

# 当前任务

根据客户需求，用中文完成 Spec，其中应当包括该模块的 Verilog IO 定义。使用 submit_spec 工具以
↪ 提交该 Spec。
```

用户提示词，验收项目时：

```
# 项目当前状态

等待项目管理员审阅验证报告

# Spec

{spec}

# 验证报告

{verification_report}

# 当前任务

根据 Spec，审阅验证报告，决定：

1. 批准当前验证报告，可用 accept_report 提交。

或者，

2. 否决当前验证报告，并通过 submit_spec 提交新的 spec ，使其他员工纠正现有版本的错误
```

工具：

```
submit_spec = {
    "type": "function",
    "function": {
        "name": "submit_spec",
        "description": "提交 SPEC 文档。",
        "strict": True,
```

```
        "parameters": {
            "type": "object",
            "properties": {
                "spec": {"type": "string", "description": "SPEC 文档内容"},
                "overwrite": {"type": "boolean", "description": "true 表示覆盖现有的 SPEC,
                ↪  false 表示将其追加到现有 SPEC 中"}
            },
            "required": ["spec", "overwrite"],
            "additionalProperties": False
        }
    }
}

accept_report = {
    "type": "function",
    "function": {
        "name": "accept_report",
        "description": "批准当前报告。",
        "strict": True
    }
}
```

2.1.2 设计工程师 Agent

设计工程师根据项目工程师提供的 Spec，生成 RTL 代码。设计工程师会将生成的代码存储在文件中，并在必要时进行修改。为避免 LLM 生成的代码存在语法错误，每次设计工程师提交代码时，都会使用 VCS 的 vlogan 工具进行语法检查。若检查出语法错误，设计工程师会根据错误信息进行修改，直到消除所有报错为止。

另外，设计工程师还会根据验证工程师的反馈，修改 RTL 代码以满足验证需求。

系统提示词：

```
你是混沌逻辑公司 IC 设计部门的设计工程师，你的团队中包括项目管理员和验证工程师。

你需要根据项目管理员提供的 Spec，设计对应的 Verilog RTL IP 块。

代码仅可通过外部工具提交，不能生成 markdown 代码块。
```

用户提示词，生成 RTL 代码时：

```
# 项目当前状态

项目管理员的 Spec 已完成

等待设计工程师的 RTL 代码
```

```
# 项目管理员的 Spec

{spec}

# 当前任务

根据 Spec 设计 Verilog RTL 代码
```

工具：

```
submit_design = {
    "type": "function",
    "function": {
        "name": "submit_design",
        "description": "提交你的 Verilog 设计代码。设计代码将保存在一个 .v 文件中。你的设计
        ↪  代码在提交后会自动进行语法检查。",
        "strict": True,
        "parameters": {
            "type": "object",
            "properties": {
                "code": {"type": "string", "description": "Verilog 设计代码"}
            },
            "required": ["code"],
            "additionalProperties": False
        }
    }
}
```

2.1.3 验证工程师 Agent

验证工程师根据项目工程师提供的 Spec 与用户提供的验证方案，生成验证计划，并编写测试用例。验证工程师会使用 VCS 编译仿真程序，并运行测试用例，生成验证报告。同样，当编译仿真程序时，若出现编译错误，验证工程师也会根据错误信息进行修改，直至成功编译得到 simv 仿真程序为止。接下来，验证工程师会检查 simv 仿真程序的输出，判断 RTL 代码是否存在问题。当 RTL 代码存在问题时，验证工程师会将问题反馈给设计工程师，并要求其修改 RTL 代码。当验证工程师认为 RTL 代码满足验证需求时，会将验证报告提交给项目工程师。

系统提示词：

```
你是混沌逻辑公司 IC 设计部门的验证工程师，你的团队中包括项目管理员和设计工程师。

你需要根据项目管理员提供的 Spec ，严格按照验证计划设计 Testbench，以验证设计工程师提供的
↪  Verilog RTL 代码，找出任何可能存在的问题。

# 注意！
```

```
testbench timescale 固定为 1ns/100ps

代码仅可通过外部工具提交，不要生成 markdown 代码块

参考模型已提供，模块名为 ref_model，端口定义与 dut 一致，构建 testbench 时应当例化参考模型

不要对 dut 的任何输出进行直接检查，所有检查都应当将 dut 的输出与参考模型相比较，报错时打印 dut
↪  与参考模型的对比

使用 ref_inst 作为实例名实例化 ref_model

不要在时钟的上升沿改变输入信号或检查输出信号

SystemVerilog 中，必须在 Testbench* 开头处 * 声明新变量
```

用户提示词，生成 testbench 时：

```
# 项目当前状态

项目管理员的 Spec 已完成

设计工程师的 RTL 代码已完成

等待验证工程师的报告

# 项目管理员的 Spec

{spec}

# 验证计划

{veri_plan}

# 当前任务

1. 按照验证计划，使用 submit_testbench 提交 testbench ，testbench 将自动执行，并返回测试结果

2. 当测试完成后，使用 write_feedback ，将测试中的问题反馈给设计工程师

3. 当测试无误后，使用 write_verification_report 提交验证报告
```

工具：

```
submit_testbench = {
    "type": "function",
    "function": {
        "name": "submit_testbench",
```

```
        "description": "提交你的 Testbench 代码。Testbench 将保存在一个 tb.v 文件中",
        "strict": True,
        "parameters": {
            "type": "object",
            "properties": {
                "code": {"type": "string", "description": "Testbench 代码"}
            },
            "required": ["code"],
            "additionalProperties": False
        }
    }
}
write_feedback = {
    "type": "function",
    "function": {
        "name": "write_feedback",
        "description": "撰写验证反馈。",
        "strict": True,
        "parameters": {
            "type": "object",
            "properties": {
                "text": {"type": "string", "description": "验证反馈"}
            },
            "required": ["text"],
            "additionalProperties": False
        }
    }
}
write_verification_report = {
    "type": "function",
    "function": {
        "name": "write_verification_report",
        "description": "撰写验证报告。",
        "strict": True,
        "parameters": {
            "type": "object",
            "properties": {
                "report": {"type": "string", "description": "验证报告"}
            },
            "required": ["report"],
            "additionalProperties": False
        }
    }
}
```

2.2 运行流程设计

接下来，介绍本软件是如何调用各个 Agent 以完成 RTL 代码的生成与验证的。

图 1展示了本软件的运行流程。首先，项目工程师 Agent 会解读用户需求，生成 Spec 并存储在文件中。接着，设计工程师 Agent 会根据 Spec 生成 RTL 代码，并进行语法检查。若存在语法错误，设计工程师会进行修改。然后，验证工程师 Agent 会根据 Spec 与验证方案生成验证计划，并编写测试用例。验证工程师会编译仿真程序并运行测试用例，生成验证报告。当 RTL 代码存在问题时，验证工程师会将问题反馈给设计工程师，设计工程师会修改 RTL 代码并重新提交。最终，当验证工程师认为 RTL 代码满足验证需求时，会将验证报告提交给项目工程师。项目工程师确认 RTL 代码满足规范要求后，整个流程结束。

为了避免设计工程师的错误误导验证工程师，设计工程师和验证工程师将分别根据项目工程师提供的 Spec 与用户提供的验证方案生成各自的代码和测试用例。整个流程中多次出现的语法检查和编译步骤，确保了生成的代码和测试用例都是正确的。

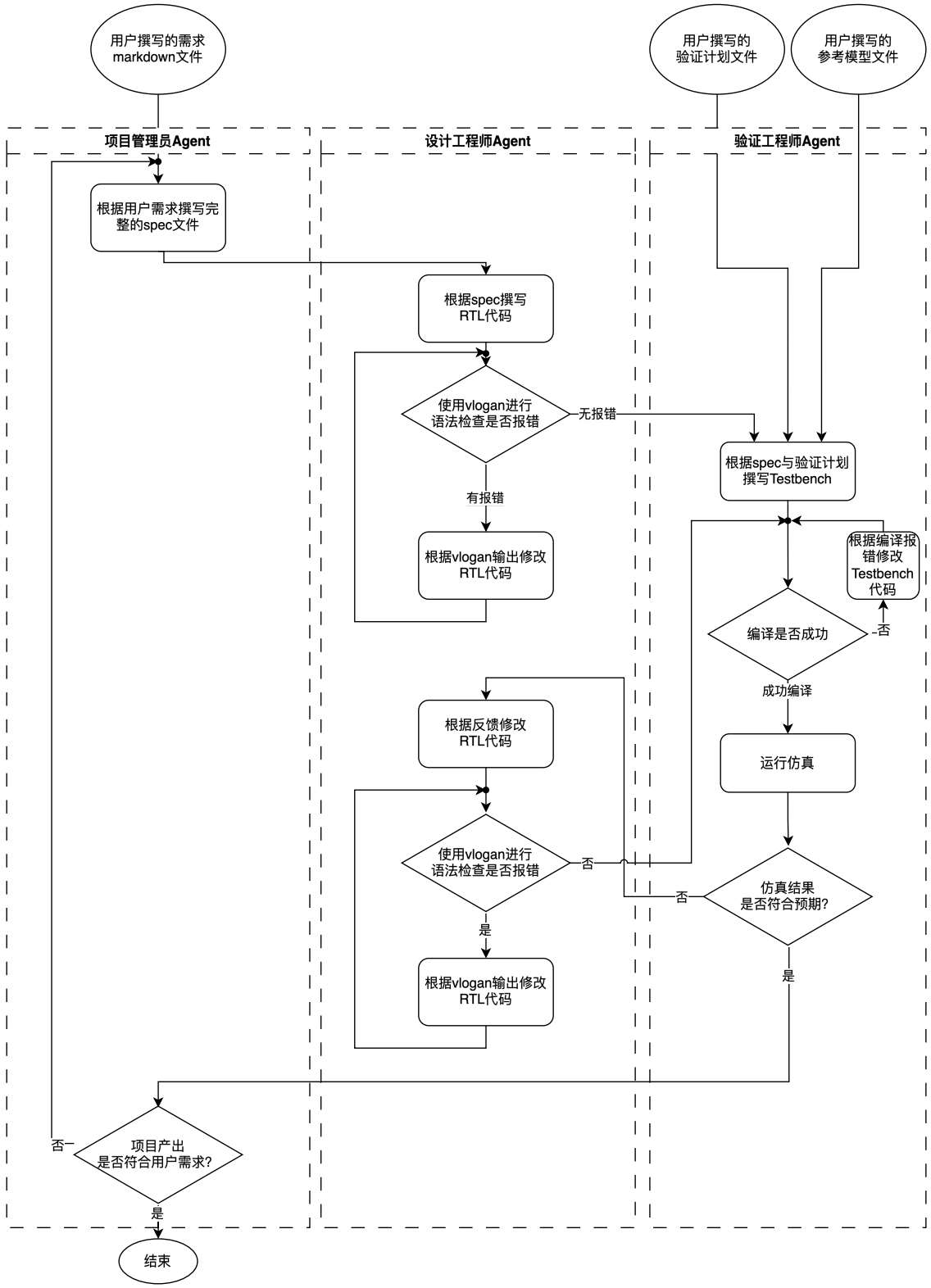

用户撰写的需求
markdown文件
用户撰写的
验证计划文件
用户撰写的
参考模型文件
项目管理员Agent
设计工程师Agent
验证工程师Agent
根据用户需求撰写完
整的spec文件
根据spec撰写
RTL代码
使用vlogan进行
语法检查是否报错
无报错
有报错
根据vlogan输出修改
RTL代码
根据spec与验证计划
撰写Testbench
根据编译报
错修改
Testbench
代码
否
编译是否成功
成功编译
根据反馈修改
RTL代码
运行仿真
使用vlogan进行
语法检查是否报错
否
否
仿真结果
是否符合预期?
是
是
根据vlogan输出修改
RTL代码
否
项目产出
是否符合用户需求?
是
结束

图 1: 程序流程图

3 运行过程展示与分析

在本章中，我们展示本系统是如何协助我们完成赛题要求的设计的。根据赛题要求，我们首先将需求的设计拆分成多个模块，其结构如图 2所示。经我们考虑，此步骤过于复杂，不可能交由 LLM 实现，因此此部分由人工进行。

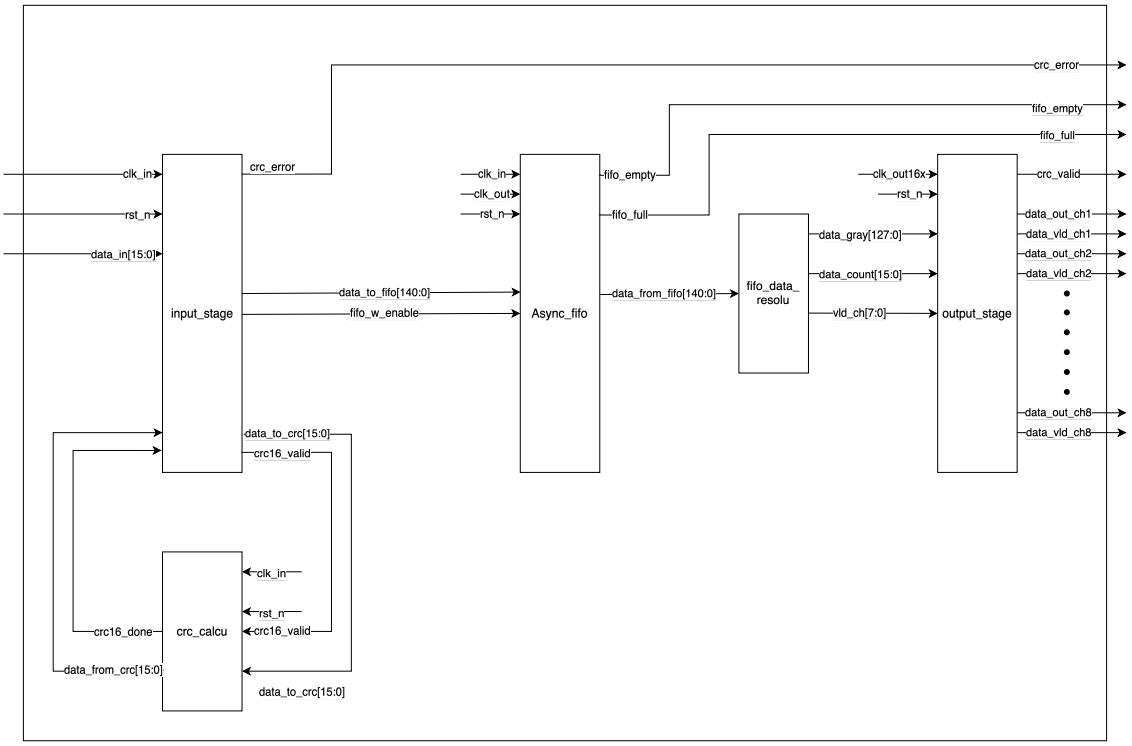


图 2: 硬件框图

在拆分子模块后，我们将使用自然语言描述各个模块的 IO 信号、功能等，并设计验证时的参考模型。下面以 fifo_data_resolu 为例子，介绍本系统的运行情况。

requirements.md 中描述了用户对该模块的需求，如下所示。

challenge/module3/requirements.md

```
# fifo 输出数据解析

命名：`fifo_data_resolu`

整体说明：纯组合逻辑模块，将从 fifo 读到的 140 位数据分离为 128 位数据位 +8 位通道选择位 +4
↪  位数据长度表示位。
其中高 128 位位数据位，中间 8 位位通道选择位，低 4 位为数据长度表示位。
4 位数据长度表示位生成 16 位具体数据长度位`data_count`，例如 0-8 分别输出 0 16 32 48 64 80
↪  96 112 128
数据位高位在前，长度由`data_count`给出，不足 128 位的数据将在低位补 0
数据位转为格雷码后，在低位补 0 补齐 128 位，输出信号`data_gray`也是高位在前。
8 位通道选择位不做处理，直接输出 8 位信号`vld_ch`。
```

```
## 顶层 IO

| 信号 | 位宽 |I/O|
|-----|-----|-----|
|data_from_fifo|140|I|
|data_gray|128|O|
|vld_ch|8|O|
|data_count|16|O|

## 信号说明

`data_from_fifo`: fifo 输出数据
`data_gray`: 输出数据的格雷码表示
`vld_ch`: 通道选择数据
`data_count`: 具体数据长度位
```

veri_plan.md 中简要提出了用户对验证的要求，如下所示。

challenge/module3/veri_plan.md

```
## 1. 基本功能测试

测试模块正确解析 140 位输入数据的功能

- 使用固定测试向量验证解析功能
- 检查 128 位数据位是否正确转换为格雷码输出 (data_gray)
- 验证低 8 位通道选择位是否正确输出 (vld_ch)
- 检查高 4 位数据长度表示位是否正确转换为 16 位 data_count
- 比较输出与预期结果，确保数据一致性

## 2. 数据长度转换测试

验证 4 位数据长度到 16 位 data_count 的转换

- 测试所有有效输入值 (0-8) 的转换:
  - 0 → 0
  - 1 → 16
  - 2 → 32
  - 3 → 48
  - 4 → 64
  - 5 → 80
  - 6 → 96
  - 7 → 112
  - 8 → 128
- 测试无效输入值 (9-15) 时的行为
- 验证转换功能为纯组合逻辑，无寄存器延迟

## 3. 二进制转格雷码测试
```

```
验证 128 位二进制到格雷码转换的正确性

- 测试全 0 输入时输出全 0 格雷码
- 测试全 1 输入时输出正确格雷码
- 测试边界值: 0x5555... 和 0xAAAA... 模式
- 测试单个位跳变时的格雷码输出
- 使用随机数据验证转换功能
- 实现软件参考模型进行实时比较

## 4. 通道选择信号测试

验证通道选择信号的处理

- 测试单个通道有效时的输出
- 测试多个通道同时有效时的输出
- 测试所有通道有效时的输出
- 验证通道选择信号直接传递无修改
- 检查信号传递为纯组合逻辑路径

## 5. 边界条件测试

测试极端输入情况下的模块行为

- 测试全 0 输入时的输出
- 测试全 1 输入时的输出
- 测试数据长度位为 0 时的行为
- 测试数据长度位为 15(最大值) 时的行为
- 测试输入数据瞬时变化时的输出响应

## 6. 实时响应测试

验证纯组合逻辑的实时响应特性

- 测试输入变化后输出是否立即跟随变化
- 验证模块无时钟延迟特性
- 测试输入信号建立/保持时间要求
- 检查输出信号是否有毛刺
- 测量输入到输出的最大组合路径延迟

# 验证环境要求

- 使用 SystemVerilog 搭建验证平台
- 实时比较 DUT 输出与参考模型结果
- 使用 $dumpfile("wave.vcd") 记录完整波形
- 不要对 dut 的任何输出进行直接检查，所有检查都应当将 dut 的输出与参考模型相比较，报错时打印
↪  dut 与参考模型的对比
```

为了确保设计符合预期，我们还为系统提供了参考模型，如下所示。

challenge/module3/ref_model/ref_model.sv

```
module ref_model (
  input  [139:0] data_from_fifo,  // FIFO 输入的 140 位数据
  output [127:0] data_gray,       // 格雷码转换后的 128 位数据
  output [7:0]   vld_ch,          // 8 位通道选择位
  output [15:0]  data_count       // 16 位数据长度位
);

  // 直接输出通道选择信号
  assign vld_ch = data_from_fifo[11:4];

  // 计算数据长度位
  assign data_count = (data_from_fifo[3:0] <= 8) ?
                      (data_from_fifo[3:0] << 4) : 16'b0;

  // 格雷码转换处理
  reg [127:0] gray_conv;
  always @(*) begin
    case(data_count)
      16'd0:    gray_conv = 128'b0;
      16'd16:   gray_conv = {data_from_fifo[139:124] ^ {1'b0, data_from_fifo[139:125]},
      ↪  112'b0};
      16'd32:   gray_conv = {data_from_fifo[139:108] ^ {1'b0, data_from_fifo[139:109]},
      ↪  96'b0};
      16'd48:   gray_conv = {data_from_fifo[139:92] ^ {1'b0, data_from_fifo[139:93]},
      ↪  80'b0};
      16'd64:   gray_conv = {data_from_fifo[139:76] ^ {1'b0, data_from_fifo[139:77]},
      ↪  64'b0};
      16'd80:   gray_conv = {data_from_fifo[139:60] ^ {1'b0, data_from_fifo[139:61]},
      ↪  48'b0};
      16'd96:   gray_conv = {data_from_fifo[139:44] ^ {1'b0, data_from_fifo[139:45]},
      ↪  32'b0};
      16'd112:  gray_conv = {data_from_fifo[139:28] ^ {1'b0, data_from_fifo[139:29]},
      ↪  16'b0};
      16'd128:  gray_conv = data_from_fifo[139:12] ^ {1'b0, data_from_fifo[139:13]};
      default:  gray_conv = 128'b0;
    endcase
  end

  assign data_gray = gray_conv;

endmodule
```

项目管理员首先根据输入生成了完整的 Spec:

fifo_data_resolu/spec.md

```
# FIFO 输出数据解析模块 Spec

## 模块名称
fifo_data_resolu（纯组合逻辑模块）

## 功能描述
将从 FIFO 读取的 140 位输入数据解析为三部分：
1. 128 位数据转换为格雷码（高位有效，低位补零）
2. 8 位通道选择位（直接输出）
3. 16 位数据长度位（由 4 位指示位计算获得）

## Verilog IO 定义
```verilog
module fifo_data_resolu (
  input  [139:0] data_from_fifo,  // FIFO 输入的 140 位数据
  output [127:0] data_gray,       // 格雷码转换后的 128 位数据
  output [7:0]   vld_ch,          // 8 位通道选择位
  output [15:0]  data_count       // 16 位数据长度位
);
```

数据格式规范
```
|  数据位 (128b)  | 通道位 (8b) | 长度位 (4b) |
|:---------------:|:----------:|:-----------:|
|    139 : 12     |  11 : 4    |   3 : 0     |
```

功能实现细节

1. 通道选择信号（vld_ch）
- 直接提取输入数据 [11:4] 位
- 无转换处理
- 计算公式：vld_ch = data_from_fifo[11:4]

2. 数据长度信号（data_count）
- 根据 4 位长度指示位生成
- 转换规则：
  0000 (0) → 0
  0001 (1) → 16
  0010 (2) → 32
  ...
  1000 (8) → 128
- 当长度指示位大于 8 时生成 0
- 计算公式：data_count = {8'b0, data_from_fifo[3:0], 4'b0}

3. 数据格雷码转换（data_gray）
```

```
- 处理步骤:
  1. 获取有效数据: data_high = data_from_fifo[139:12]
  2. 根据 data_count 获取有效高位
  3. 将有效数据转格雷码: gray = data_high ^ (data_high >> 1)
  4. 低位补零至 128 位
- 示例:
  当 data_count=64 时:
  data_gray = {gray[127:64], 64'b0}

## 设计约束
- 纯组合逻辑实现
- 无时钟/复位信号
- 处理延迟: 单周期组合逻辑延迟
- 长度指示位默认范围: 0-8 (其他值未定义)
```

根据完整的 Spec，设计工程师进行了多轮迭代，其详细过程过长，在提交文档中的'3-LLM运行'文件夹中给出。在几次迭代后，成功的设计如下：

fifo_data_resolu/design/dut.v

```
module fifo_data_resolu (
  input  [139:0] data_from_fifo,
  output [127:0] data_gray,
  output [7:0]   vld_ch,
  output [15:0]  data_count
);

  // 1. 直接提取通道选择信号
  assign vld_ch = data_from_fifo[11:4];

  // 2. 计算数据长度 (当长度指示位 >8 时输出 0)
  assign data_count = (data_from_fifo[3:0] > 4'd8) ? 16'b0 :
                      {8'b0, data_from_fifo[3:0], 4'b0};

  // 3. 数据转换处理
  wire [127:0] data_high = data_from_fifo[139:12];  // 提取 128 位原始数据
  wire [127:0] gray_full = data_high ^ (data_high >> 1);  // 完整格雷码转换

  // 根据长度指示位选择有效数据段
  reg [127:0] gray_out;
  always @(*) begin
    case (data_from_fifo[3:0])
      4'd0: gray_out = 128'd0;          // 0000 -> 全 0
      4'd1: gray_out = {gray_full[127:112], 112'd0};  // 高 16 位有效
      4'd2: gray_out = {gray_full[127:96], 96'd0};    // 高 32 位有效
      4'd3: gray_out = {gray_full[127:80], 80'd0};    // 高 48 位有效
      4'd4: gray_out = {gray_full[127:64], 64'd0};    // 高 64 位有效
```

```
      4'd5: gray_out = {gray_full[127:48], 48'd0};     // 高 80 位有效
      4'd6: gray_out = {gray_full[127:32], 32'd0};     // 高 96 位有效
      4'd7: gray_out = {gray_full[127:16], 16'd0};     // 高 112 位有效
      4'd8: gray_out = gray_full;         // 全部 128 位有效
      default: gray_out = 128'd0;         // 其他情况输出全 0
    endcase
  end

  assign data_gray = gray_out;

endmodule
```

同上，验证工程师的迭代过程也不再此处赘述。根据完整的 Spec 和用户生成的参考模型，验证工程师提供的最终 Testbench 如下：

fifo_data_resolu/verification/tb.sv

```
module tb;
  reg [139:0] data_from_fifo;
  wire [127:0] dut_data_gray, ref_data_gray;
  wire [7:0] dut_vld_ch, ref_vld_ch;
  wire [15:0] dut_data_count, ref_data_count;

  fifo_data_resolu dut (
    .data_from_fifo(data_from_fifo),
    .data_gray(dut_data_gray),
    .vld_ch(dut_vld_ch),
    .data_count(dut_data_count)
  );

  ref_model ref_inst (
    .data_from_fifo(data_from_fifo),
    .data_gray(ref_data_gray),
    .vld_ch(ref_vld_ch),
    .data_count(ref_data_count)
  );

  initial begin
    $dumpfile("wave.vcd");
    $dumpvars(0, tb);
  end

  always @* begin
    #10;

    if (dut_vld_ch !== ref_vld_ch) begin
      $display("[ERROR] vld_ch mismatch at time %t", $time);
      $display("  Input: %h", data_from_fifo);
```

```
    $display("  DUT vld_ch = %h", dut_vld_ch);
    $display("  REF vld_ch = %h", ref_vld_ch);
    $finish;
  end

  if (dut_data_count !== ref_data_count) begin
    $display("[ERROR] data_count mismatch at time %t", $time);
    $display("  Input: %h", data_from_fifo);
    $display("  DUT data_count = %h", dut_data_count);
    $display("  REF data_count = %h", ref_data_count);
    $finish;
  end

  if (dut_data_gray !== ref_data_gray) begin
    $display("[ERROR] data_gray mismatch at time %t", $time);
    $display("  Input: %h", data_from_fifo);
    $display("  DUT data_gray = %h", dut_data_gray);
    $display("  REF data_gray = %h", ref_data_gray);
    $finish;
  end
end

integer i;
reg [3:0] test_length;
reg [127:0] test_data;

initial begin
  test_data = 128'hA5A5A5A5A5A5A5A5A5A5A5A5A5A5A5A5;
  test_length = 4'd8;
  data_from_fifo = {test_data, 8'hFF, test_length};
  #100;

  test_data = 128'h5A5A5A5A5A5A5A5A5A5A5A5A5A5A5A5A;
  test_length = 4'd4;
  data_from_fifo = {test_data, 8'hAA, test_length};
  #100;

  for (i = 0; i <= 15; i = i + 1) begin
    test_length = i[3:0];
    test_data = $random;
    data_from_fifo = {test_data, $random, test_length};
    #100;
  end

  data_from_fifo = 140'd0;
  #100;

  data_from_fifo = {128'hFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF, 8'hFF, 4'h8};
```

```
    #100;

    test_data = 128'h55555555555555555555555555555555;
    data_from_fifo = {test_data, $random, 4'h8};
    #100;

    test_data = 128'hAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA;
    data_from_fifo = {test_data, $random, 4'h8};
    #100;

    for (i = 0; i < 50; i = i + 1) begin
      data_from_fifo = {$random, $random, $random} &
      ↪  140'hFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF;
      #100;
    end

    for (i = 0; i < 8; i = i + 1) begin
      data_from_fifo[11:4] = (1 << i);
      #100;
    end

    data_from_fifo[11:4] = 8'hFF;
    #100;

    data_from_fifo = 140'd0;
    #100;

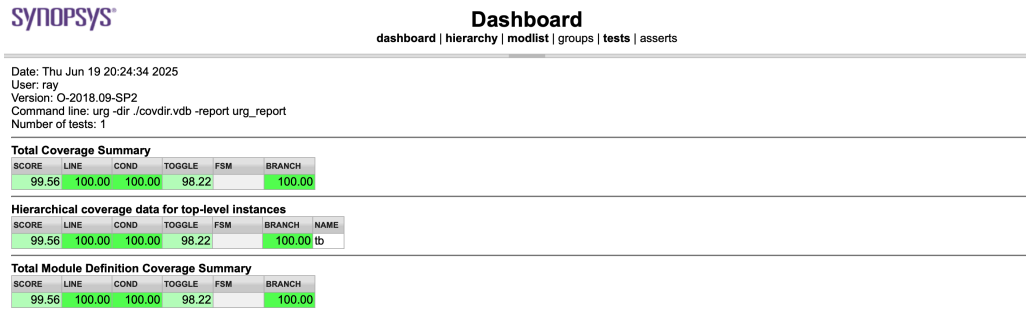
    data_from_fifo = 140'hFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF;
    #100;

    data_from_fifo[3:0] = 4'hF;
    #100;

    repeat (20) begin
      data_from_fifo = {$random, $random, $random} &
      ↪  140'hFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF;
      #1;
    end

    $display("All tests passed successfully!");
    $finish;
  end
endmodule
```

运行该 Testbench，可以看到全部设计通过，覆盖率如图 3所示。



SYNOPSYS

Dashboard

dashboard | **hierarchy** | **modlist** | groups | **tests** | asserts

Date: Thu Jun 19 20:24:34 2025
User: ray
Version: O-2018.09-SP2
Command line: urg -dir ./covdir.vdb -report urg_report
Number of tests: 1

Total Coverage Summary

SCORE	LINE	COND	TOGGLE	FSM	BRANCH
99.56	100.00	100.00	98.22		100.00

Hierarchical coverage data for top-level instances

SCORE	LINE	COND	TOGGLE	FSM	BRANCH	NAME
99.56	100.00	100.00	98.22		100.00	tb

Total Module Definition Coverage Summary

SCORE	LINE	COND	TOGGLE	FSM	BRANCH
99.56	100.00	100.00	98.22		100.00

图 3: 覆盖率

可以看到，LLM 设计的 Testbench 成功覆盖了本模块的绝大部分内容，整体的代码覆盖率达到了 99.56%。

4 总结

本项目设计了一套由 LLM 驱动的 RTL 代码的生成与验证自动化软件，实现了 IC 设计流程的自动化，能够快速产出原型设计。然而，在设计中，我们发现，一般的通用大模型在数字电路相关任务上、尤其是验证任务上，存在相当大的局限性。要进一步减少本项目的迭代次数，提升设计效率，必须对大模型进行针对性的训练，以使其能够更好地理解数字电路的设计与验证流程。本项目产出的原型设计，也应由专业的设计工程师和验证工程师进行进一步的修改与验证，以确保其满足实际的设计需求。