



# Digital Logic 2025 Fall Project Architecture Report(Final Version)

## Authors:

- 陈彦桥(Yanqiao Chen, 12412115), 负责顶层模块、输入、随机数生成、矩阵覆盖逻辑、随机选择、卷积模块的开发
- 蒋言(Yan Jiang, 12410337): 负责计算模块以及下属计算模块的开发
- 侯栋升(Dongsheng Hou, 12410421): 负责BRAM和Matrix Manager的开发

贡献比1: 1: 1

## 开发计划日程安排和实施情况，版本修改记录

### 开发计划日程安排

项目开发从初始规划开始，逐步实现各个模块。以下是主要阶段：

- **初始阶段**：项目立项，确定基本架构，使用DRAM存储。编写基本的储存模块，通过测试debug储存功能。
- **中期阶段**：实现UART通信、按键消抖、基本模式切换，实现主要模式控制的自动机以及数据多路选择器。
- **后期阶段**：实现矩阵输入、存储和计算功能，发现综合问题和时序违例，重构为BRAM存储。
- **最终阶段**：完善所有计算模式、设置模式、错误处理，初步开发了GUI但是尚未完善，因此不提交检查。

### 实施情况

项目基本完成所有要求，并实现了若干bonus功能，包括卷积操作、倒计时错误恢复、随机生成等。GUI有框架但未完全连接。

### 版本修改记录

GitHub仓库链接：<https://github.com/SUSTech-Digital-Logic-Project-Team/Digital-Logic->

提交记录（部分）：

- ae04f1c Merge pull request #28 from hotteano/main
- 7d797e5 Merge branch 'main' into main
- 31489e5 update
- 72e5d1f update
- 38d3476 update
- e5bf7ca update
- e6a417d update
- 15bd659 update
- 6817a1a update
- 679961f Okay this is the end
- ... (更多提交见仓库)，共108次commit， pull request共29个

## 项目架构设计说明

基于提交的架构设计文档，改进后的最终实现方案如下：

**原有架构内容和内部实现保持不变**

## 输出对齐和参数配置设计思路

### 输出对齐

在显示和UART输出中，实现输出对齐通过在数字间添加空格，并在行列分隔时发送换行符。计数器维护当前行数据，当一行显示完毕时发送换行符（ASCII码10）。

其实现主要通过若干子状态寄存器实现格式化输出，其核心在于通过读取Matrices Manager中的矩阵维度信息，通过维护计数器，检查边界来实现格式化输出。没有与更多其它模块有数据交换。

### 参数配置

我们在顶层设计了一系列config寄存器，用于存储设置模式所设置的参数。设置模式允许用户通

过UART自定义参数，包括最大矩阵维度、最大数值、同一维度最大矩阵数量、倒计时设置。FPGA复位后参数初始化为默认值，用户可在运行时修改。与周边模块的关系：设置模式与顶层模块交互，更新config寄存器；后续，顶层模块将这些参数传递给各子模块，如计算模式、生成模式等，用于限制输入范围和操作。

## 应用开发

项目实现了GUI应用（matrix\_calculator\_gui.py，位于github仓库，此部分代码由于未完成，故未提交）。

## 输入和输出接口

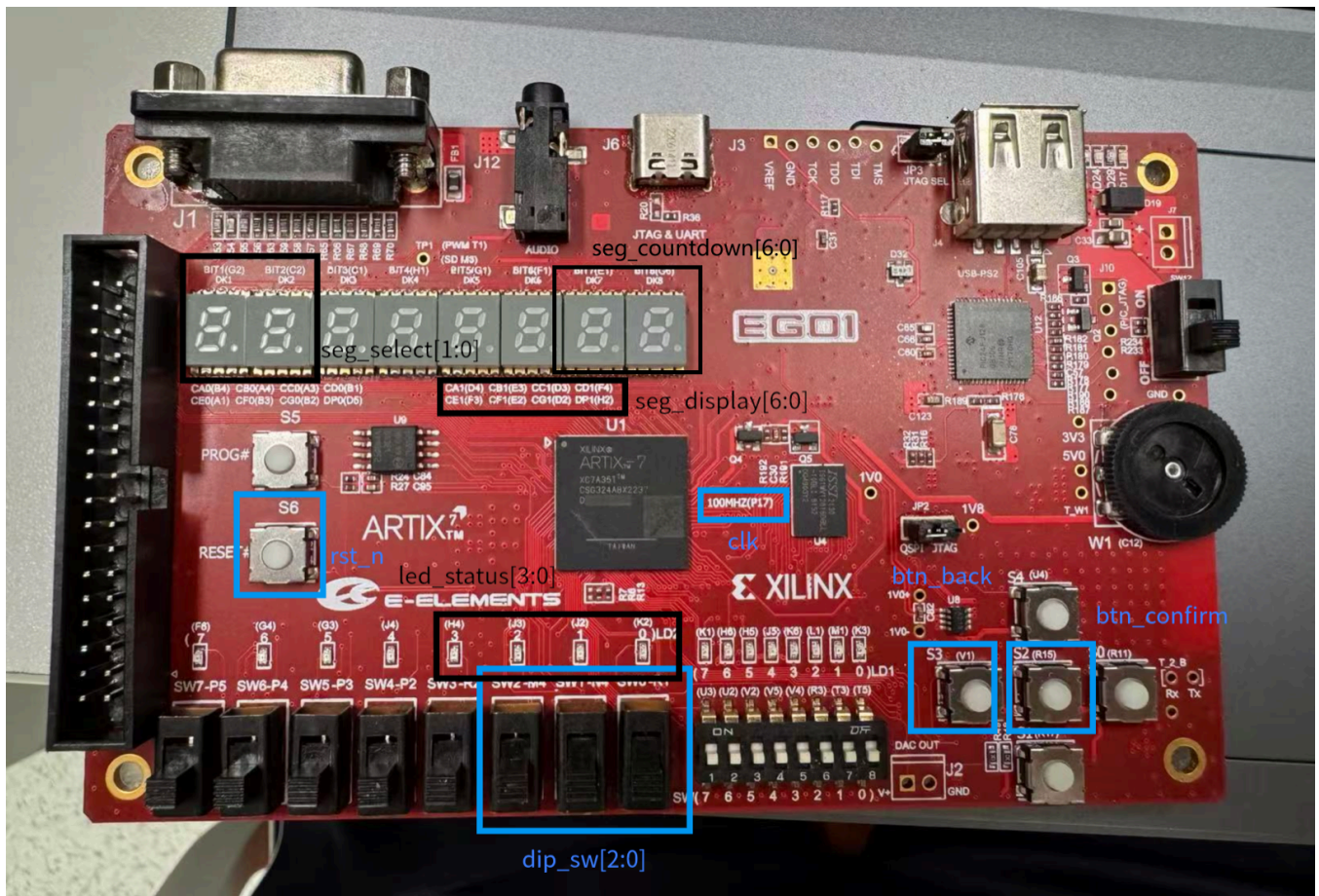
The input port and output port is listed as follows:

### INPUT:

- The clock: 1 bit width
- Reset button: 1 bit width
- DIP switch: 3 bits width, SW2 as MSB, SW0 as LSB, used for choosing different modes
- Confirm button: 1 bit width, used for confirm main mode selection
- Go back button: 1 bit width, used for go back to main menu
- UART Receiver: 1 bit width, used for receiving data from PC

### OUTPUT:

- UART Transmitter: 1 bit width, used for transmitting data to PC
- 7-Seg LED: 7 bits width, used for displaying information like mode, operation type, error code, counting time...
- LED: 4 bits width, LD3 as MSB, LD0 as LSB, used for indicating working, error type and so on.
- 7-Seg LED selection: 2 bits width, used for selecting 7-Seg LED



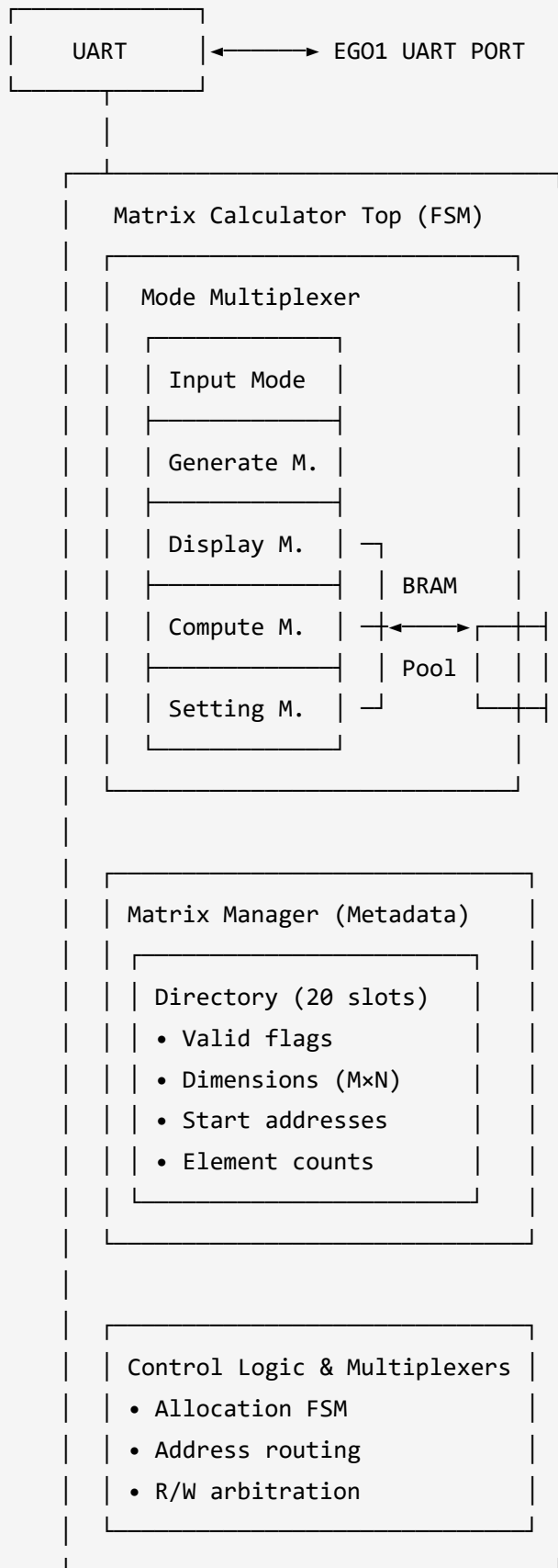
## 整体架构概览

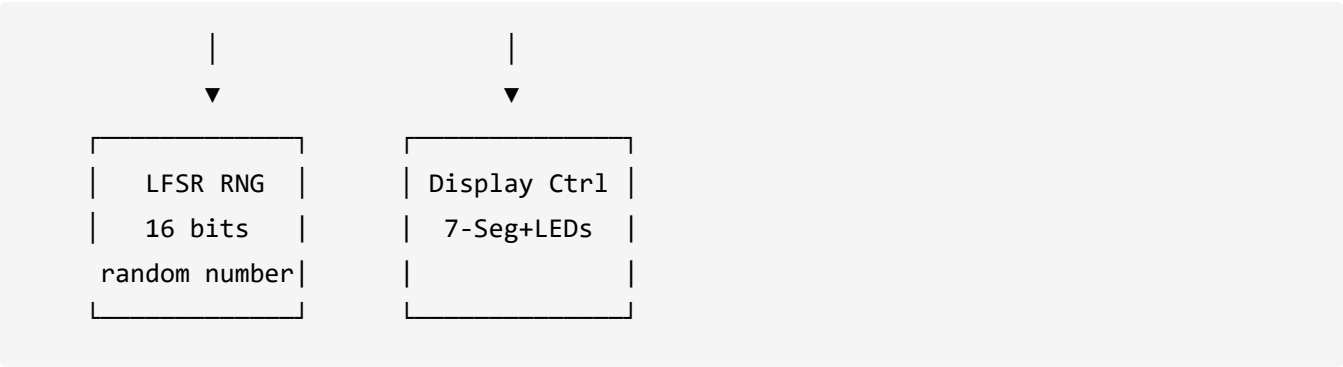
The Architecture of this project is purposed as follow:

### A Global View:

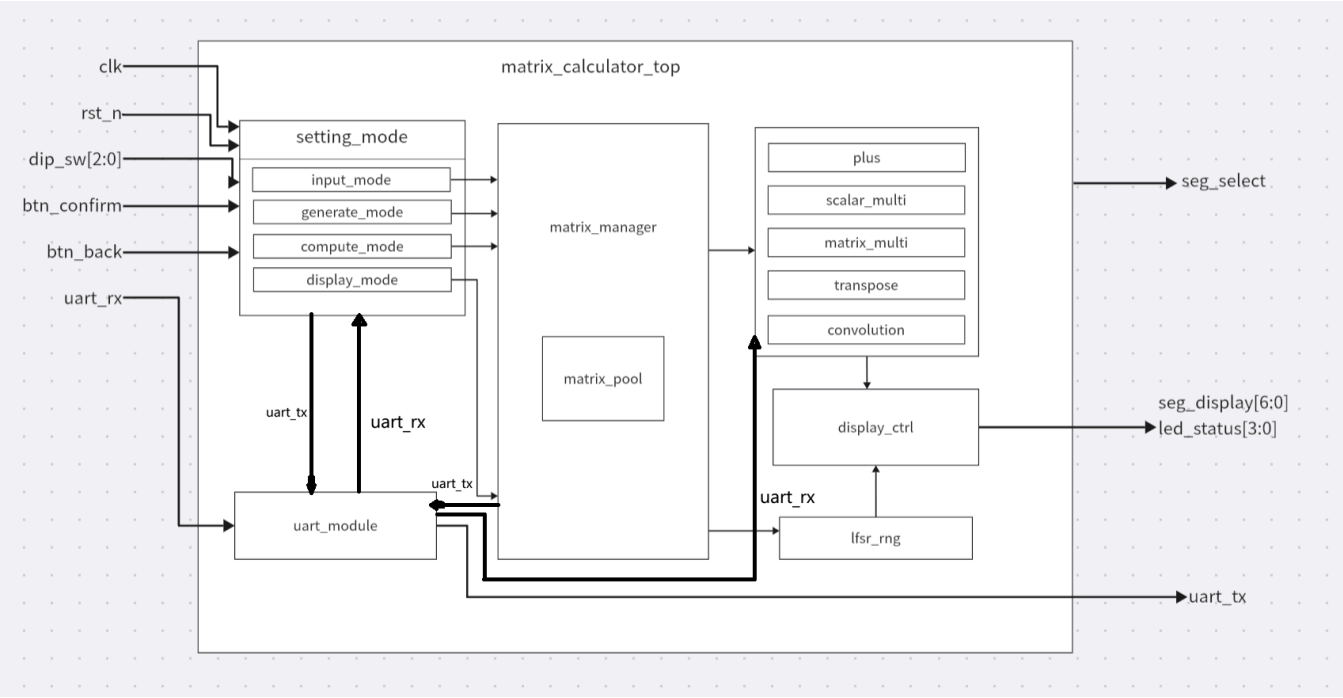
- The Top Module
- The Memory Controller
- Processing Modes
- Display Controller
- UART Module
- Tool Kit

### 架构图示意





架构和数据流示意图



The Top Module

顶层模块包含：

Name	Input	Output	Usage
Matrix Calc Top	clk, rst_n, dip_sw[2:0], btn_confirm, btn_back, uart_rx	uart_tx, seg_display[6:0], led_status[3:0], seg_select[1:0]	Top module

# The Memory Controller

Name	Input	Output	Usage
Memory Pool	clk, rst_n, a_n, a_we, a_addr[ADDR_WIDTH-1:0], a_din[DATA-1:0], b_en, b_addr[ADDR_WIDTH-1:0]	a_dout[ADDR_WIDTH-1:0], b_dout[DATA_WIDTH-1:0]	Storing Data
Matrix Manager	clk, rst_n, alloc_req, alloc_m[3:0], alloc_n[3:0], commit_req, commit_slot[3:0], commit_m[3:0], commit_n[3:0], commit_addr[11:0], query_clot[3:0]	alloc_slot[3:0], alloc_addr[11:0], alloc_valid, query_m[3:0], query_n[3:0], query_addr[11:0], query_element_count[7:0], total_matrix_count[7:0]	Managing Matrices

## Process Modes

Name	Input	Output	Usage
Compute Mode	clk (1 bit), rst_n (1 bit), mode_active (1 bit), config_max_dim [3:0], dip_sw [2:0], btn_confirm (1 bit), rx_data [7:0], rx_done (1 bit), tx_busy (1 bit), total_matrix_count [7:0], query_valid (1 bit), query_m [3:0], query_n [3:0], query_addr [11:0], query_element_count [7:0], mem_rd_data [15:0]	clear_rx_buffer (1 bit), tx_data [7:0], tx_start (1 bit), selected_op_type [3:0], query_slot [3:0], mem_rd_en (1 bit), mem_rd_addr [11:0], error_code [3:0], sub_state [3:0]	Performs matrix computations like addition, multiplication
Generate Mode	clk (1 bit), rst_n (1 bit), mode_active (1 bit), config_max_dim [3:0],	clear_rx_buffer (1 bit), tx_data [7:0], tx_start (1 bit), alloc_req (1 bit), commit_req (1 bit),	Generates matrices with random or

Name	Input	Output	Usage
	config_max_value [3:0], random_value [3:0], rx_data [7:0], rx_done (1 bit), tx_busy (1 bit), alloc_slot [3:0], alloc_addr [11:0], alloc_valid (1 bit)	commit_slot [3:0], commit_m [3:0], commit_n [3:0], commit_addr [11:0], mem_wr_en (1 bit), mem_wr_addr [11:0], mem_wr_data [15:0], error_code [3:0], sub_state [3:0]	predefined values
Input Mode	clk (1 bit), rst_n (1 bit), mode_active (1 bit), config_max_dim [3:0], config_max_value [3:0], rx_data [7:0], rx_done (1 bit), tx_busy (1 bit), alloc_slot [3:0], alloc_addr [11:0], alloc_valid (1 bit), mem_rd_data [15:0]	clear_rx_buffer (1 bit), tx_data [7:0], tx_start (1 bit), alloc_req (1 bit), alloc_m [3:0], alloc_n [3:0], commit_req (1 bit), commit_slot [3:0], commit_m [3:0], commit_n [3:0], commit_addr [11:0], mem_wr_en (1 bit), mem_wr_addr [11:0], mem_wr_data [15:0], mem_rd_en (1 bit), mem_rd_addr [11:0], error_code [3:0], sub_state [3:0]	Receives matrix data from UART and manages memory allocation
Setting Mode	clk (1 bit), rst_n (1 bit), mode_active (1 bit), rx_data [7:0], rx_done (1 bit), tx_busy (1 bit)	clear_rx_buffer (1 bit), tx_data [7:0], tx_start (1 bit), config_max_dim [3:0], config_max_value [3:0], config_matrices_per_size [3:0], error_code [3:0], sub_state [3:0]	Configures operational settings



# Display Controller

Name	Input	Output	Usage
Display Control	clk (1 bit), rst_n (1 bit), matrix_data [7:0], mode [2:0]	seg_display [6:0], seg_select [1:0], led_status [3:0]	Manages display of matrix data and status indicators

# UART Module

Name	Input	Output	Usage
UART Receiver	clk (1 bit), rst_n (1 bit), uart_rx (1 bit)	received_data [7:0]	Receives data from PC
UART Transmitter	clk (1 bit), rst_n (1 bit), data_to_send [7:0]	uart_tx (1 bit)	Sends data to PC
UART Module	clk (1 bit), rst_n (1 bit), uart_rx (1 bit), data_to_send [7:0]	uart_tx (1 bit), received_data [7:0]	Combines UART Receiver and Transmitter functionalities

# Tool kit

Name	Input	Output	Usage
matrix package	NO	NO	Some Macros settings, like clock frequency
LSFR Random number generator	clk, rst_n, max_value[3:0]	random_value[3:0]	For generating psuedorandom number, by using polynomial

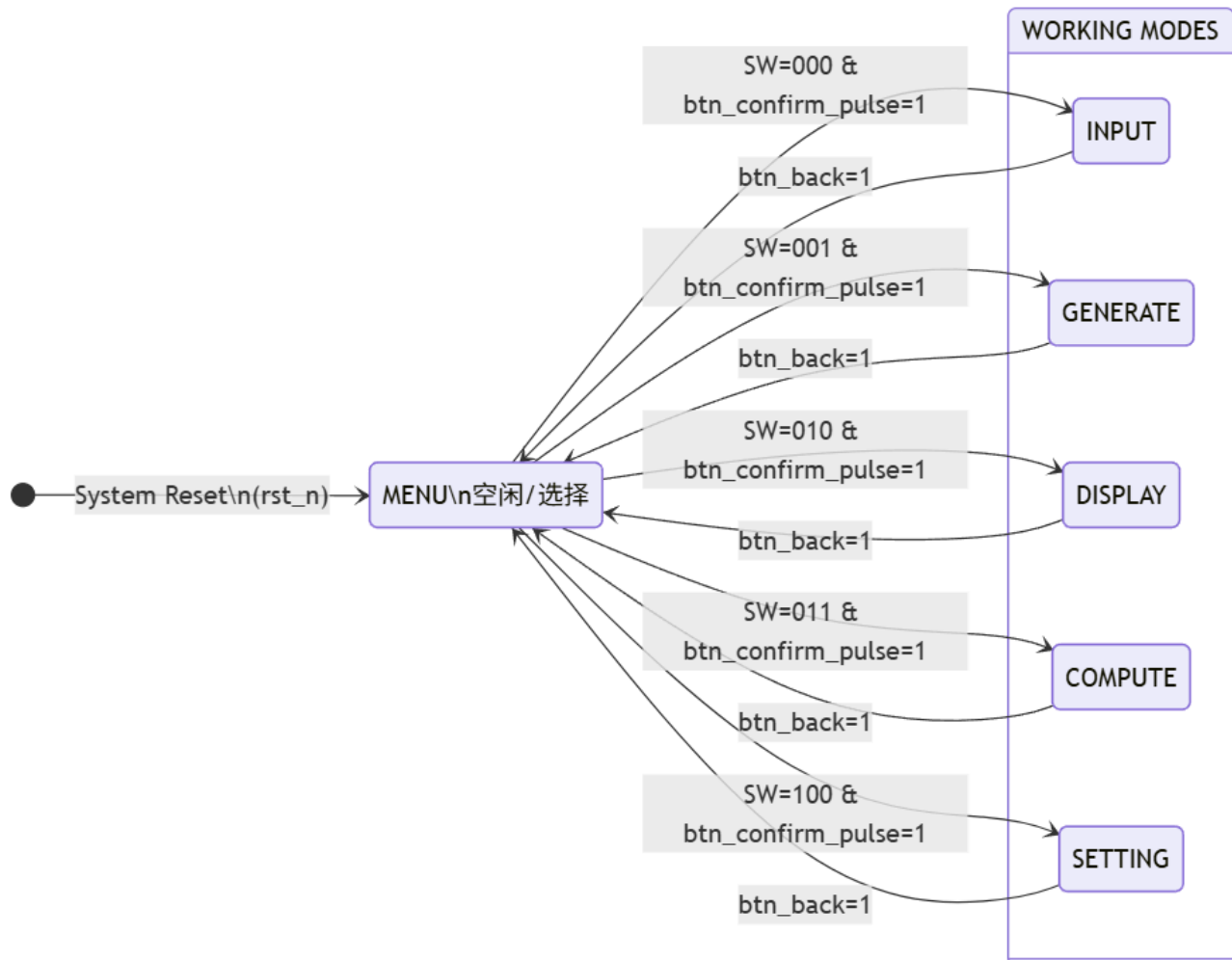
# The FSM

项目中主要的状态机包含：

- IDLE, MODE\_INPUT, MODE\_COMPUTE, MODE\_GENERATE, MODE\_SETTING, MODE\_DISPLAY
- Input Mode: IDLE, PARSE\_M, PARSE\_N, CHECK\_DIM, WAIT\_ALLOC, PARSE\_DATA, FILL\_ZEROS, COMMIT, DISPLAY\_MATRIX, DONE, ERROR
- Display Mode: IDLE, SHOW\_COUNT, WAIT\_SELECT, READ\_DATA, CONVERT\_DATA, SEND\_DIGITS, DONE
- Generate Mode: IDLE, WAIT\_M, WAIT\_N, ALLOC, GEN\_DATA, COMMIT, DONE
- Compute Mode: IDLE, SELECT\_OP, SELECT\_MATRIX, EXECUTE, SEND\_RESULT, DONE

## 状态机示意图

Top

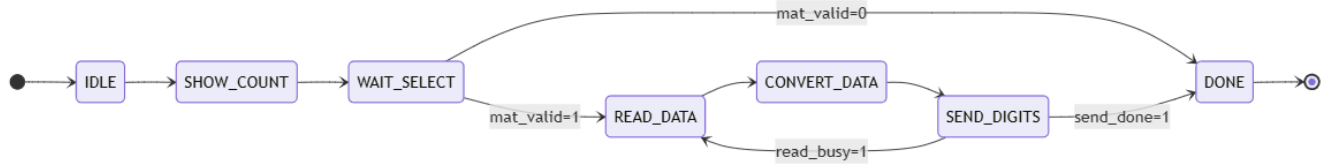


Modules

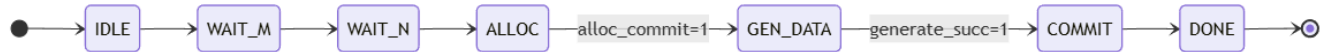
## INPUT



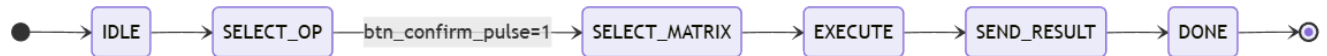
## DISPLAY



## GENERATE



## COMPUTE



# 开源和AI使用

## 开源声明

本项目使用MIT协议开源，所有代码和文档可在GitHub仓库获取。

## AI使用声明和开源代码使用声明

- AI提供BRAM优化建议，有效避免LUT综合爆炸。
- 在开发和测试的两个环节中，我们通过AI生成重复的功能代码以及生成测试文件来进行开发。
- AI局限性：在复杂Verilog中，AI常在无关处过度检查，输出需人工review和debug。
- 其中，input mode部分代码由上一版本（无BRAM版本）使用AI完全重构而来，generate mode由input mode复制和手动修改而来。内存管理（BRAM Pool和Matrices Manager由AI主要编写）主要参考AI编写，例如分配逻辑和覆盖逻辑。计算模块由Yan Jiang整体编写，然后通过AI进行细节调优，尤其是对一些编写错误、状态重复和寄存器冗余进行了调整，并且由完全整合在Compute Mode里面变成了模块化为单独模块。UART模块为课程提供的模块，我们加入了UART Module方便其他模块调用。
- 随机数生成器（线性反馈位移寄存器）由大模型提供，我们上网查找了资料，对其

有了一定的了解。

- Button Debounce主要由AI提供（此时实验课尚未讲授，因此我们查找资料并用AI生成了一份），并且在后续的实验课上进行了完善。
- Setting Mode设置思路主要由小组成员提供，并且交由AI搭建状态机框架，然后根据需要进行调整。
- 限制文件为手动编写，无AI生成内容。
- 本项目主要使用的开源代码即课程提供的UART模块，没有使用其它仓库代码。
- 本项目许多比较困难的地方由Claude Opus 4.5编写，其它部分包含了Gemini Pro 2.5, Gemini Pro 3, GPT-5.1-Codex-Max, GPT-4o, Grok Code Fast, Claude Sonnet 4.5, Claude Haiku 4.5。

## 思考与总结

本次项目，我们大量采用AI生成重复代码-人工调优debug模式，尤其在状态机设置、多路选择和模块连接中。这些模块编写起来相当枯燥单调，因此我们普遍使用了Prompt工程来生成这些代码。另外，尤其是在Compute Mode中，其中包含了大量格式化输出所需要的代码，我们大部分都使用AI来生成这些重复且冗余的代码。

在AI时代，我们往往会因为Vibe Coding所带来的方便而陷入其中。但是实际上，过度的Vibe Coding往往会导致一个项目变成完全的屎山的速度大大加快。AI往往会添加大量不需要的冗余功能，并且可能埋下若干在前期设计中完全意想不到的问题。尤其是在当前，AI尚不具备设计高度复杂系统的能力，因此需要工程师对于AI生成的代码更加谨慎和小心，必须要完整地理解AI所写的代码的功能、实现方法和其与其它部分的交互。一个可以借鉴的经验是，不要给AI提供太长的上下文，最好是让AI修改局部功能，或者生成重复的代码，而不要将整个工程交给AI来处理，这样往往会导致不必要的麻烦。

## AI辅助开发导致的问题：上下文理解混乱

例如，我们在开发过程中遇到了这样的问题：AI在前期设计的时候，使用了5位位宽的二进制数来表示状态，但是到了后期，由于模块功能更加复杂，状态迅速增加到了100+个，此时，AI仍然使用5位的位宽来标记状态（不知为何，可能受到了上下文干扰），因此导致了我们在开发过程中需要仔细地去查找到到底是哪个地方出了问题。最终我们将这个问题检查出来并进行了修复。

另一个例子是，我们通过AI来去除一些不需要的代码（例如，我们在前期编写了display模式，但是后续放弃了开发，需要删除），但是AI在匹配了所有应该删除项之后，继续删除了很多必要的功能，这导致我们不得不回滚代码以保证代码可以运行。此类情况在无需付费的免费模型尤其

常见（例如GPT-4o和Grok Code Fast）

这表明，AI辅助编程仍然无法很好地实现复杂功能，若要实现，则需要非常详细的Prompt。有时候，清晰地描述一个需求比自己直接编写还要困难，因为需要更加谨慎的检查AI的修改是否会影响到其它部分的正常工作。

## 一些比较好的交互例子

如下是在进行矩阵覆盖开发的对话。

---

**Human:** 可以保存一个指针，指向每一个维度优先替换的那个矩阵，matrix\_manager的分配逻辑增加：若超出顶层模块中的“同一维度矩阵限制”，那么直接返回指针指向的那个矩阵，然后将指针指向下一个同一维度的矩阵（由于矩阵在内存上先后被分配，因此只需要向后查找第一个同维度的矩阵），如果指针指向的同一维度矩阵数超出限制，则回到第一个。

**Agent:**

(此处省略若干修改，AI正确地修改了逻辑，并且使用了全局时间戳记录矩阵的录入顺序)

- Added per-dimension cap support in matrix\_manager\_optimized.v: tracks per-size counts and replacement pointers, returns the pointer slot for overwrite when count hits dim\_limit\_per\_size, advances pointer to the next same-dimension matrix, and keeps counts/pointers in sync on commit. New helper find\_next\_same\_dim wraps within slots.
- Introduced per-dimension tracking tables (1024 entries for 5-bit m/n) and reset them; kept existing address reuse so overwrites use the original start\_addr.
- Wired config\_matrices\_per\_size into the manager via new port dim\_limit\_per\_size in matrix\_calculator\_top\_optimized.v.

Next steps: build/sim to confirm synthesis passes and that allocation+commit flows from input/generate/compute still behave as expected under dimension caps.

---

如上，我们发现，如果需要Agent正确地修改文件，必须要提供明确的上下文（上文中，我们明确提供了Matrix Manager文件，并且要求其修改分配逻辑），并且指出你的实现思路，最好是分步叙述，并且要给出明确的对象描述，不能够模糊，否则AI可能根据其想法通过其它的方式进行

实现，这大大增加了理解成本和debug的成本。

**如下是在进行错误状态倒计时开发的对话。**

---

**Human:** 显然，如果我们要显示一个两位数倒计时，我们就不能只选中一个数码管，我们应该在两个数码管之间高频切换（例如使用一个两位独热码从而显示例如15之类的倒计时）

**Agent:**

(此处省略若干修改，AI正确地修改了逻辑，并且使用了全局时间戳记录矩阵的录入顺序)

明白了，您需要将 `count_down_select` 从单一的使能信号改为一个2位的独热码，用于在两个数码管之间高频切换，以显示两位数的倒计时（如15秒）。

让我先查看 `display_ctrl` 模块了解当前的显示控制逻辑：

`display_ctrl.v`

现在我理解了整体结构。当前 `countdown_val` 是4位的，只能显示0-9。需要：

将 `countdown_val` 扩展为支持两位数（5-15）

在 `display_ctrl` 中添加两个数码管的扫描逻辑用于倒计时显示

将 `count_down_select` 改为2位独热码输出

已编辑

`matrix_calculator_top_optimized.v`+25-24

现在修改 `display_ctrl` 模块，添加两位数倒计时显示的扫描逻辑：

现在需要调整扫描计数器的声明顺序，因为 `scan_cnt` 在倒计时逻辑中被引用，但是它的声明在后面。让我检查一下文件的当前状态：`display_ctrl.v`

需要将 `scan_cnt` 的声明移到倒计时逻辑之前：

现在删除重复的 `scan_cnt` 声明和 `always` 块：

`display_ctrl.v`

现在验证所有文件是否有错误：

`display_ctrl.v`  
`matrix_calculator_top_optimized.v`

修改完成！让我总结一下本次更改：

## 修改总结

1. matrix\_calculator\_top\_optimized.v
  - count\_down\_select 从 1 位改为 2 位输出（独热码）
  - 删除了固定的 assign count\_down\_select = 1'b1，改由 display\_ctrl 模块驱动
  - countdown\_val 从 4 位扩展为 5 位，支持 5-15 的两位数显示
  - countdown\_reg 同样扩展为 5 位
  - 在 display\_ctrl 实例化中添加了 .count\_down\_select(count\_down\_select) 端口连接
2. display\_ctrl.v
  - countdown\_val 输入从 4 位扩展为 5 位
  - 新增 count\_down\_select 2 位独热码输出端口
  - 添加了 countdown\_tens（十位）和 countdown\_ones（个位）的计算逻辑
  - 添加了 digit\_to\_seg 函数用于数字到七段显示的编码
  - 倒计时显示使用与主显示相同的 scan\_cnt[16] 进行高频切换：
    - scan\_cnt[16] == 0：选中左侧数码管（count\_down\_select = 2'b10），显示十位
    - scan\_cnt[16] == 1：选中右侧数码管（count\_down\_select = 2'b01），显示个位
    - 十位为 0 时不显示（避免显示 "05" 而是显示 " 5"）
  - 无错误时 count\_down\_select = 2'b00（关闭倒计时数码管）

---

如上，我们可以看到，我们的提示词并不复杂，但是Claude Opus 4.5很好的修改了代码，并且给出了详细的代码注释。这表明，Claude Opus 4.5的代码能力非常强。我们在实际体验中也感受到了这一模型的效率和准确率。这表明，在编写过程中，选择正确的代码模型也是很重要的。许多同学可能无法使用Claude系列模型，在项目开发中可能就会因为许多模型（尤其是GPT的代码模型）的胡乱输出导致整个项目不得不回滚。

## 总结

总而言之，我们在本次的项目开发中学到了如何搭建一个完整的小项目，并且从AI的代码中学习了Verilog相关的知识（例如，强制综合器使用BRAM，强制综合器不优化掉某一个变量），同时也学会了很多Debug方面的知识，例如编写测试文件，设置断点测试并上板通过串口进行调试，等等。最终我们也顺利完成了本次Project的开发，实现了绝大部分的功能，并且实现了参数设置、输出对齐和卷积等bonus功能。

# 对课程的建议

我认为，Project后续的设计应该聚焦于FPGA上计算本身的优化，而不是主要在输入输出、存储等细节上。首先，Memory在课程中比较后面才开始讲到，并且比较浅显，对于Project早期开发造成了一定困扰。其次，AI很容易能够编写出类似的功能，因此此功能很可能无法反映出学生对于数字逻辑课程本身的学习程度。

一个建议是，下次的Project中，应该将“功能性”的内容转向“性能”方向，例如，通过PE阵列加速运算、编写快速傅里叶卷积、快速矩阵乘法等算法，以及通过Vivado本身的各类优化器对FPGA生成的电路进行优化，甚至在FPGA上实现神经网络等等，这或许是在深度学习时代的一种探索方向。

## 附录：一些可能的Q&A

来自项目仓库中README的部分，此部分对项目中的大部分的代码细节进行了简要的解释，详见代码注释。

### 如何操作内存？

- 内存管理器只做一件事：读入地址，给出数据
- 矩阵管理器负责：储存矩阵的统计信息（行列维数、起始地址和结束地址），分配内存（找到一个空闲位置，就返回给模块，让模块去内存管理器存）
- 模块如果要存入数据，首先是启动Alloc阶段，挂起Alloc请求，模块等待内存分配；若分配成功，则将将要存储的地址返回，交给模块（这里是矩阵管理器和模块的交互），读入数据以后，进行一系列处理，然后模块用内存地址提交写入（这里是模块和内存管理器交互）

### 如何执行计算？

- 确定操作数矩阵的元素位置地址，访问然后寄存
- 确定内存写入段落，直接根据每一位的表达式计算结果，然后写入
- 计算逻辑大同小异，此处不再赘述

### 伪随机数？

- 使用线性反馈位移寄存器（LFSR）生成随机数（具体原理没了解）



- 通过一个随机种子起始进行迭代，然后取低四位作为输出（如果大于当前设定最大值，取模保证数值位置）

## 如何实现规格打印？

实际上非常简单，只需要打印一个数字，然后打印一个空格，通过计数器维护当前一行的数据，如果一行显示完毕，那么发送换行符（ASCII码），如是就可以实现数字之间有空格、行列分开的打印效果

## 如何实现倒计时？

当输入检测过程中，检测到数字超出范围，电路将error寄存器为非零数，顶层模块接收到报错之后，报错led点亮，顶层模块的计时器开始倒计时，如果直到倒计时没有正确输入，那么发送reset信号，将模块内部状态恢复到IDLE，清空暂存数据。

## 如何实现设置模式？

首先，顶层模块中包含了几个关键寄存器（矩阵最大维数、最大数值、同一维度最大矩阵数量、倒计时设置——config系列变量）。FPGA烧录之后，第一个时钟上升沿，rst\_n = 0，所有参数被置为为matrix\_pkg中的参数设置。后续进入setting之后，根据uart接收的数据，将寄存器中的值更改为用户定义的值。

## 如何实现多数码管显示？

通过两位数码管选择位，当选择位为“01”的时候，显示左边那一位；选择位为“10”的时候，显示左边那一位。我们以countdown为例，当处于“10”的时候，显示十位数，我们只需要计算出寄存器countdown\_second的十位数，然后输入display\_ctrl即可控制；“01”的时候同理，只需要使用一个选择器即可。

## 如何实现矩阵覆盖？

我们使用一个slot\_age来记录每一个槽位的“年龄”，年龄越小，越应该被覆盖。每一次分配内存的时候，先检测是否达到了上限，并且事先保存年龄最大的矩阵，如果超出上限，那么就将年龄最大的矩阵的起始地址返回，写入的时候会自然覆盖这个矩阵（因为维度相同，不需要再次分配内存）

# 线路和寄存器说明

## 顶层模块

- main\_state, main\_state\_next 主状态寄存器
- op\_type\_from\_compute 从模块内取出计算类型，方便退出其中一个计算模式
- active系列变量 模块激活信号
- 配置系列变量config\_setting 由setting模式控制，初始化配置为默认值，后续可自由配置
- 组合逻辑config：将setting中的信号寄存起来，传输给其他模块
- uart接口
- 内存读写寄存器：读写使能、写入使能、地址和输入输出（din表示数据写入线路，dout表示读出数据线路）
- 数据多路选择信号：分配申请、分配维度信息、分配槽位、分配地址、分配valid信号；查询信号，查询维度和查询槽号，查询valid信号；存入信号commit系列（表明写入完成，commit的全部内容记录进入matrix\_manager，若失败，则不会存入内存；未来，可以增加写入失败自动回滚的机制）
- 随机数寄存器（来自线性反馈位移寄存器）
- 来自各模式的错误码，错误led灯信号；倒计时寄存器，倒计时秒数控制，十位和个位分别存储
- 子状态信号寄存
- 多路选择逻辑：通过激活信号控制多模块信号，防止multidriver（后续的地址也是同理的）
- 主状态机，通过拨码开关控制主状态机；处于计算模式的时候，若内部状态处于1（SELECT\_OP），那么可以返回主菜单；如果不是，等内部归位以后，如果再次按下按钮再退出；方便连续在模式内部进行计算
- substate用于读取内部的状态细节，方便输出错误（实际上在本项目中没有太大用处）
- 计时器，用两个信号输入显示控制器，方便用两个数码管显示倒计时（如果进入错误模式，那么开始倒计时；结束倒计时之后，发送timeout信号重置模块内部状态）
- 若干实例化

## 输入模式和生成模式（唯一区别是数据来源）

- 输入数字：通过寄存器积累读入两位数字；并且通过读入顶层模块的最大维度设置，进行错误检验

- 等待分配：发起内存分配请求和矩阵数据，得到分配数据；若分配失败，可以考虑进入错误状态。因为没有要求所以没做。
- 解析数字，如有错误，那么将会进入错误状态
- 如果输入的数字不足，那么自动写入0直到计数器到达要求的元素数量
- 如果多余，自动截断
- Commit，将数据写入内存，地址信息写入matrix\_manager
- 展示模式：规格化打印字符，根据刚刚分配的地址读取内存中的数据，检查是否正确写入了数据
- 对于生成模式，增加了一个矩阵数量计数器，如果不足，回到ALLOC状态

## 消抖

- 一个10ms的倒计时，用来检测确实产生了一个上升电平，然后在下降沿产生脉冲

## BRAM POOL

这是一个很简单的写入和读取控制：使能为1、读入地址，写入数据；使能为1，读入地址，读取数据，传出数据

ramstyle用来控制综合的时候的风格，保证使用bram1以减少LUT使用。

## MATRIX MANAGER

考虑到地址数据并不多，因此使用DRAM储存，ramstyle为distributed

- matrix valid标记槽位是否可用
- rows存储行，对应槽位
- col存储列，对应槽位
- start\_addr 存储矩阵开始的地址，end\_addr记录矩阵结束的地址
- slot\_age用来存储矩阵的年龄，我们永远使用最年老的矩阵进行覆写
- global\_age\_counter存储全局年龄，方便给新来的矩阵分配年龄
- 5B最大矩阵限制配置，防止配置的最大矩阵数超过实际限制
- query通过地址读取manager内部数据

分配逻辑如下：

- 申请暂存变量
- 如果接收到了分配请求，那么计算分配的位置数量；
- 计数已经存储同维度矩阵的数量，防止超出限制

- 从小槽位开始向大的遍历，直到找到合法的
- 检查末端地址，然后返回上一个矩阵的末尾的后一个地址作为新的地址
- 如果统计到同维度矩阵过多，那么就根据前面统计到的最年老的矩阵开始替换（直接返回这个矩阵的地址，因为维度相同，不需要重新计算元素数量）
- 写入完毕，commit，将地址、valid、行列记录在manager中，以便下次利用

也就是说，我们只需要使用槽位就可以访问矩阵的所有内容，这提高了效率。

## COMPUTE MODE

- 按下按键，进入对应的计算模式；
- 计算valid槽位的数量，然后打印数量
- 遍历所有的矩阵维度组合，打印所有的统计信息（耗时比较久），然后计数
- 输入矩阵维度，然后遍历，打印符合要求的矩阵的信息
- 输入当前的索引，然后再次遍历，找到索引所引的槽号，然后记录到op1中；同样，重复一遍（如果需要第二个操作数）
- 打印操作数，按下确认，进行计算
- 如果是卷积，那么选择矩阵，然后给出所有的3乘3卷积核，然后选择，按下按钮进行计算

## 线性反馈位移寄存器

我们通过LFSR生成随机数序列，通过反馈多项式和位移来生成随机数字

## 计算方法

- 根据计算的性质，计算出需要分配的内存，然后将结果写入内存，对于每一个地址，我们在一个循环中计算出地址上应该写入的数据，然后直接写入
- 计算完毕以后，根据地址打印矩阵

## 计算实现

- 加法：读取对应坐标上的两个数字，然后相加，写入内存
- 标量乘法同理
- 转置：顺序读取，然后写入内存中对称位置的内存 ( $\text{start} + j * \text{dim}_m + i$ )
- 乘法：读取 ( $\text{addr\_op1} + (i * \text{dim}_p + k)$ ,  $\text{addr\_op2} + (k * \text{dim}) + j$ )，然后k++，累加到位置里面
- 卷积：同理，每一个i, j变化，我们通过ki和kj访问op2，计算偏置（例如，是左上

角，我们通过row\_idx, col\_idx存储出对应的地址偏移)，然后计算出大矩阵此时需要读取的地址，累加结果直到9个数字都被累加完毕