

计算机组织与结构 II: CPU 设计文档

李勃璘

吴健雄学院

版本: 1.0

日期: 2025 年 5 月 6 日

摘要

本文设计并实现了一个基于 FPGA 的 16 位单周期 CPU，包括指令集架构、流水线结构和完整的硬件实现。设计采用改进型冯·诺依曼结构，将指令内存和数据内存分离，使用统一的外部总线进行交互。CPU 支持 13 条指令，包括数据传输、算术逻辑运算、移位及跳转指令，并实现了直接寻址和立即数寻址方式。为支持用户交互，设计了基于 UART 的指令输入接口以及 LED 与七段显示器的输出接口。本文详细介绍了 CPU 的整体架构、内部模块设计和实现方法，并对算术、逻辑、跳转运算和溢出处理功能进行了仿真和实物验证。实验结果表明，所设计的 CPU 能够正确执行各类指令并完成复杂计算任务，验证了设计的有效性和可靠性。

目录

1 概述	1
2 CPU 结构设计	1
2.1 总体架构	1
2.2 指令集架构	1
2.2.1 位宽设计	1
2.2.2 指令集支持的指令	1
2.2.3 寻址方式	2
2.3 CPU 内部寄存器	3
2.4 算术逻辑单元 ALU	4
2.5 控制单元 CU	4
2.5.1 控制单元结构	4
2.5.2 微操作指令 (Micro-Operations)	5
2.5.3 CU 控制信号 (Control Signals)	6
2.6 CPU 内部总线和外部总线	8
3 外围设备	8
3.1 用户端代码解释	8
3.2 UART 接收与指令内存写入设计	9
3.2.1 UART 接收逻辑	9
3.2.2 数据缓冲与存储结构	9
3.3 数据内存	10
3.4 用户交互设计	10
4 核心模块设计	11
4.1 时钟、复位与停止信号	11
4.2 UART 传输与指令内存	11
4.2.1 UART 模块	11
4.2.2 FIFO 模块	12
4.2.3 指令 BRAM 模块	13
4.3 控制单元	14
4.3.1 Control Memory	14
4.3.2 CAR	15
4.4 内部寄存器和 ALU 设计	15
4.4.1 ALU	15
4.4.2 MAR	17
4.4.3 MBR	17
4.4.4 PC	18
4.4.5 IR	19
4.4.6 ACC	20
4.5 数据内存设计	21
4.6 外部总线设计	22
4.7 用户面设计	23

4.7.1 按键消抖模块	23
4.7.2 七段显示器	24
4.7.3 LED 灯显示	25
4.8 时序优化	25
5 性能分析与功能验证	25
5.1 时序分析	25
5.2 资源分析	26
5.3 CPU 功能仿真	26
5.3.1 简单加法器	26
5.3.2 乘法与溢出验证	27
5.3.3 移位运算验证	29
5.3.4 逻辑运算与无条件跳转验证	30
5.4 FPGA 实物验证	31
5.4.1 开发环境与测试准备	31
5.4.2 简单加法器实物验证	31
5.4.3 乘法与溢出实物验证	32
5.4.4 移位运算实物验证	32
5.4.5 逻辑运算与无条件跳转实物验证	34
6 总结与展望	34
6.1 设计总结	34
6.2 已知问题	34
附录	36
A 完整设计代码	36
A.1 汇编程序处理 Python 脚本	36
A.2 UART 接收与指令内存模块	39
A.3 控制单元设计	49
A.4 内部寄存器与 ALU 设计	60
A.5 数据内存设计	74
A.6 外部总线设计	75
A.7 用户面设计	77

表格目录

1	指令集包含指令及功能	2
2	指令集支持的寻址方式	3
3	CPU 内部寄存器的含义、总存储条数、单位位宽和数据解释格式	3
4	状态寄存器列表	3
5	ALU _{op} 与执行运算的对应关系	4
6	CPU 微操作指令表	5
7	寄存器控制信号一览	7
8	CPU 控制信号表	7
9	指令内存模块外部接口	11
10	UART 模块外部接口	12
11	FIFO 模块外部接口	12
12	指令 BRAM 模块外部接口	13
13	Control Memory 模块外部接口	14
14	CAR 模块外部接口	15
15	ALU 模块外部接口	16
16	MAR 模块外部接口	17
17	MBR 模块外部接口	18
18	PC 模块外部接口	19
19	IR 模块外部接口	20
20	ACC 模块外部接口	21
21	数据内存模块外部接口	22
22	外部总线模块外部接口	23
23	KEY_JITTER 模块外部接口	24
24	KEY_JITTER 模块外部接口	25

1 概述

中央处理单元（CPU）是计算机系统的核心组件，负责执行程序中的指令并处理数据。它由多个核心部件组成，包括算术逻辑单元（ALU）、控制单元（CU）、寄存器、缓存、总线以及与外部存储和外设的接口。CPU 的设计和实现是计算机体系结构的基础，决定了计算机的性能、效率以及可扩展性。随着现代计算机技术的不断发展，CPU 的设计已经经历了从单核到多核、从简单指令集到复杂指令集的转变，涉及到流水线、缓存管理、指令调度等多个高级设计问题。

在现代 CPU 中，指令集架构（ISA）定义了 CPU 能够识别并执行的指令类型，而 ALU 则负责执行这些指令中的算术和逻辑运算。控制单元（CU）则根据指令的操作码生成控制信号，协调 CPU 内部和外部的各个组件进行协作。此外，寄存器和缓存等存储单元在数据处理和存储中起着至关重要的作用。通过高效的设计和优化，CPU 能够实现高速的计算和响应能力，从而支持各种计算任务的执行。

本文通过设计一个基于 FPGA 的简化 CPU 架构，探索了 CPU 的基本组成与工作原理。整个项目的设计过程中，从指令集的定义到硬件实现，涵盖了计算机体系结构中的核心概念与技术，旨在帮助深入理解 CPU 设计的各个方面。

本文接下来的章节安排如下：

第二章将介绍 CPU 内部架构，即指令集、内部寄存器、ALU、内外总线以及控制单元设计，第三章将主要介绍 CPU 外部设备的设计，包括前端输入指令、指令传入内存、内存格式和结果显示功能，第四章是二、三章提出的设计方案的 Verilog 实现和分模块仿真结果，第五章是该设计的整体仿真结果和在 NEXYS 4 DDR FPGA 开发板上的测试结果。第六章对该设计进行了总结，并提出一些可改进的方向。另外，附录中还提供了设计的全部 Verilog 代码和项目地址。

2 CPU 结构设计

2.1 总体架构

CPU 的总架构（包括内存、外设等）示意图可见图 1。

CPU 由控制单元（CU），逻辑运算单元（ALU），数据内存（Data Memory）、指令内存（Instruction Memory）和寄存器组（Registers）组成，寄存器组通过被 CU 生成的控制信号控制的数据通路（Data Path）连接。另外，MAR 和 MBR 分别还和外部地址总线、数据总线相连接，用于与内存交互。

2.2 指令集架构

指令集是指 CPU 能够对数据进行的所有操作的集合。每一条指令都可以被解释为寄存器与寄存器、内存、I/O 端口之间的交互。交互方式由 CU 中的微指令（Micro-operation）给出，且每一条微指令都需要一个时钟执行（如不进行优化）。

2.2.1 位宽设计

地址段长为 **8** 位，指令码（Opcode）宽度为 **8** 位。因此，每一条指令的位宽为 **16** 位。

2.2.2 指令集支持的指令

指令集共支持 13 条不同的指令，列于表 1。每一条指令包含一个指令码，使用二进制格式存储。

图 1: CPU 总体架构

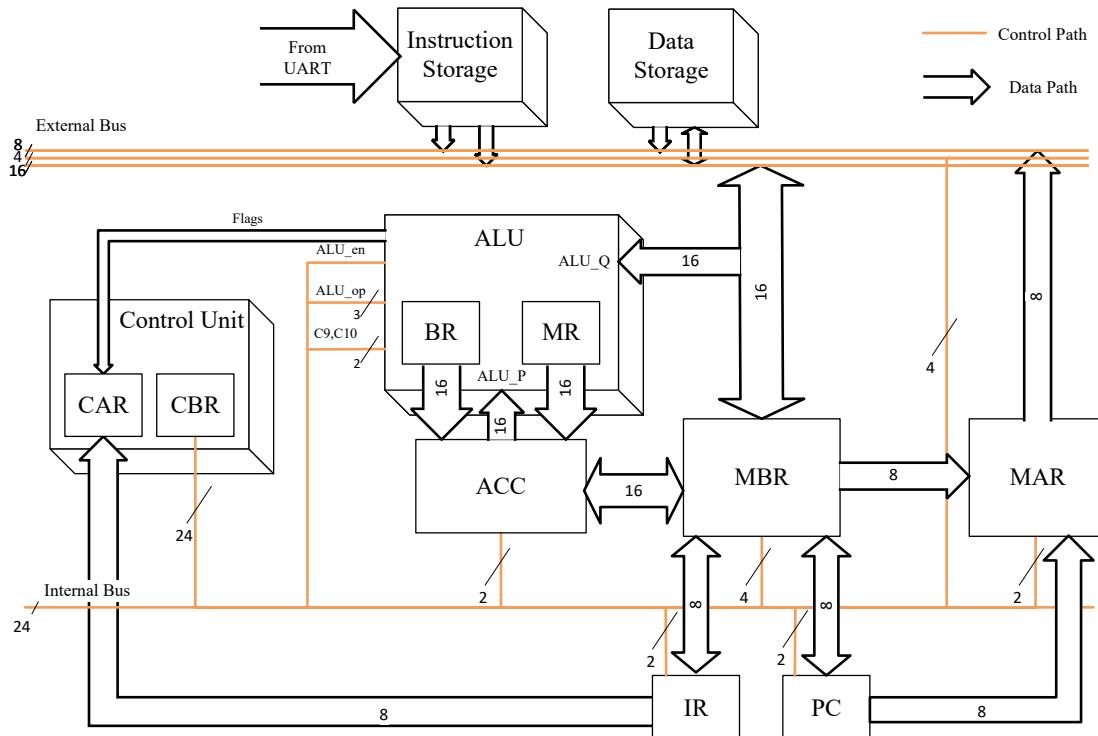


表 1: 指令集包含指令及功能

助记符	指令码（低四位）	描述
STORE X	0001	结果存入数据地址 X
LOAD X	0010	加载数据地址 X
ADD X	0011	定点数加法
SUB X	0100	定点数减法
JGZ X	0101	结果 > 0 时跳转至指令地址 X
^④ JMP X	0110	无条件跳转至指令地址 X
HALT	0111	暂停程序
MPY X	1000	定点数乘法
AND X	1001	按位与
OR X	1010	按位或
NOT X	1011	按位非
SHIFTR X	1100	算术右移 X 位
SHIFTL X	1101	算术左移 X 位

2.2.3 寻址方式

寻址方式指对地址段数据的解释方式。寻址方式由对应指令指定，支持表 2 中的全部寻址方式。目前设计中指令码的最高位为 1 时，寻址方式为立即数寻址；指令码的最高位为 0 时，寻址方式为直接寻址。

表 2: 指令集支持的寻址方式

寻址方式	描述	最高位
立即数寻址	地址字段是操作数本身，数据为补码格式	1
直接寻址	地址字段为存放操作数的地址	0

在实际编程中，默认所有指令为直接寻址，除非指令中通过标注“IMMEDIATE”明确指定为立即数寻址。详细说明见第 3.1 节。

2.3 CPU 内部寄存器

该部分描述 CPU 内部寄存器的含义、存储格式和数据被解释为的格式。这些寄存器通过 CPU 的内部数据通路相连接。寄存器操作是 CPU 快速操作的核心。

表 3: CPU 内部寄存器的含义、总存储条数、单位位宽和数据解释格式

寄存器	含义	条数	位宽	数据解释格式	归属模块
PC	程序计数器，存储当前指令地址	1	8	指令码 (Opcode)	/
MAR	内存地址寄存器，存储要访问的内存地址	1	8	地址码 (Address)	/
MBR	内存缓冲寄存器，存储从内存读取或写入的数据	1	16	二进制补码	/
IR	指令寄存器，存储当前正在执行的指令	1	8	指令码 (Opcode)	/
BR	ALU 内部寄存器，存储 ALU 计算结果	1	16	二进制补码	ALU
ACC	累加寄存器，存储 ALU 运算结果	1	16	二进制补码	/
MR	ALU 内部寄存器，存储 ALU 乘法高 16 位	1	16	二进制补码	ALU
CM	控制存储器，存储微指令控制信号	37	24	控制信号	CU
CAR	控制地址寄存器，指向当前执行的微指令	1	7	CM 中的条数下标	CU
CBR	控制缓冲寄存器，存储当前微指令的控制信号	1	24	控制信号	CU

除上述寄存器以外，ALU 进行运算时还会更改状态寄存器（Flags），用于 CU 进行条件判断。例如，JGZ 命令需要判断上一步的运算结果是否大于 0，CU 便可以直接通过状态寄存器中的 ZF（Zero Flag）和 NF（Negative Flag）寄存器进行判断。本设计中使用的所有状态寄存器见表 4，它们都直接连向 CU，通路不受控制信号的控制。Flags 对用户公开，配置详见用户交互部分（第 3.4 节）。

表 4: 状态寄存器列表

寄存器	全称	行为
ZF	Zero Flag	ALU 运算结果（通常为 ACC）为 0 时置 1
CF	Carry Flag	存储算术移位移出的比特（由于有符号数不存储进位）
OF	Overflow Flag	非乘法运算下 BR 溢出时置 1，乘法运算下 MR 溢出置 1
NF	Negative Flag	ALU 运算结果为负数时置 1

MF	Multiply Flag	乘法运算结果超过 16 位，且高位未被读取时置 1
----	---------------	---------------------------

2.4 算术逻辑单元 ALU

算术逻辑单元 ALU 负责进行大部分 CPU 内的计算¹。为简单起见，CPU 的计算全部为 **16 位定点有符号数计算**。

ALU 与外围寄存器的控制通路见第 2.5.3 节。ALU 受到来自控制单元的 ALU_{en} 和 ALU_{op} 控制，前者决定 ALU 能否进行运算，后者决定 ALU 执行什么运算。在 ALU_{en} 为 1 时，它通过 ACC 和 MBR 获取运算的两个数据 ALU_P 和 ALU_Q ，并将计算结果存入 16 位 BR 寄存器（若出现过乘法则可能存入 MR 寄存器），同时根据 LU_P 、 ALU_Q 和运算结果和更新 Flags 寄存器。

表 5 描述了 ALU_{op} 与执行运算的对应关系。

表 5: ALU_{op} 与执行运算的对应关系

ALU_{op}	运算类型	ALU_{op}	运算类型
000	加法 (ADD)	100	或 (OR)
001	减法 (SUB)	101	非 (NOT)
010	乘法 (MPY)	110	算术右移 (SHIFTR)
011	与 (AND)	111	算术左移 (SHIFTL)

由于在实际运算中，很可能会出现类似 $m \times x + n$ 的运算。如果加法和减法不支持 32 位运算，则很可能导致乘法的结果也受溢出影响。因此，设计中引入了 MF (Multiply Flag) 寄存器，表示乘法运算的结果是否超过 16 位且未被存储。在 MF 为 1 时，加减法被允许进行 32 位运算，且在 16 位溢出时不会更新溢出标志位 OF。而当出现存储指令 STORE 时，系统检测到此时 MR 和 BR 寄存器都不为 0，则会顺次存储 BR 和 MR 寄存器（即执行 STOREH 指令，该指令是一条内部隐藏指令，具体内容见第 4.3.1 节）。由于 MR 寄存器被读取，MF 会被清零，以避免后续的加减法运算受到影响。

2.5 控制单元 CU

控制单元 (Control Unit, CU) 负责协调和控制寄存器、ALU、内存等各个模块以实现指令的执行。它采用微操作指令模式设计，根据当前指令的操作码和状态寄存器的标志位生成相应的控制信号，指引数据通路中的各个寄存器、ALU、内存和外设进行正确的操作。2.5.1 节将介绍该控制单元的结构；2.5.2 节将具体描述本设计使用的微操作指令，并提供指令集的微操作指令表以供参考；2.5.3 节将介绍各个控制信号位的作用以及微操作指令表与控制信号的对应。

2.5.1 控制单元结构

控制单元由控制地址寄存器 (Control Address Register, CAR)、控制数据寄存器 (Control Buffer Register, CBR) 和控制单元内存 (Control Memory, CM) 组成，并受到寻址逻辑 (Sequencing Logic) 的控制。在一个微操作指令周期，控制单元通过完成以下操作执行一个微操作：

1. 根据 CAR 的地址，寻找 CM 对应地址存储的控制信号，并传输给 CBR；
2. CBR 将控制信号译码，传输到相应的接收单元，并将下一跳信息传输给 CAR；
3. 寻址逻辑通过下一跳信息、Flags 和 Opcode 确定下一跳地址，并写入 CAR。

控制单元示意图 (图 2) 体现了 CU 内部的关键单元，以及上述操作的数据流向。

¹自增与 PC 赋值在设计中不引入 ALU。

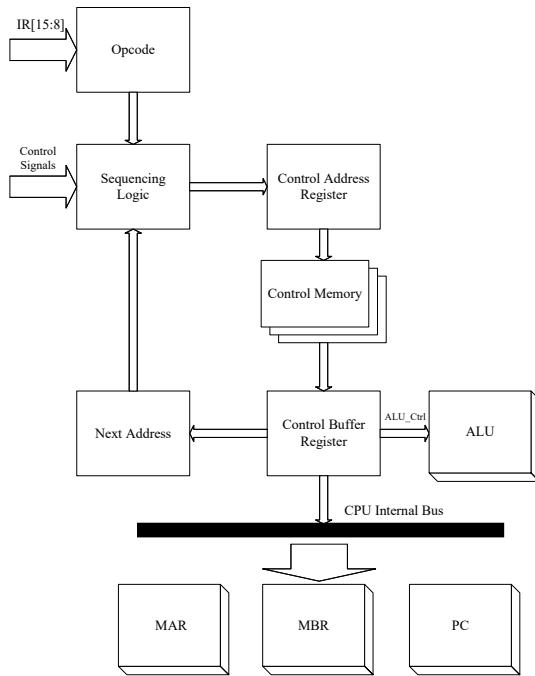


图 2: 控制单元结构示意图

2.5.2 微操作指令 (Micro-Operations)

指令集中所有指令都需要多个时钟周期完成，因此需要将指令集的指令分解为多步微操作指令。每步微操作指令通常为寄存器操作。按照寄存器操作的类型，可以将每条指令的执行整合为以下六个步骤，并按步骤顺序执行。

- **IF(Instruction Fetch):** 从指令存储器中取出指令，同时确定下一条指令地址（指针指向下一条指令）；
- **ID(Instruction Decode):** 翻译指令，同时让计算机得出要使用的运算，并得出寻址方式。
- **FO(Fetch Operands):** 取立即操作数到 MBR，即指令的低 8 位。
- **IND(Indirect):** 间接寻址周期，每插入一个 IND 周期则间接寻址深度 +1。不插入 IND 周期则为立即数寻址。在本设计中由于不考虑间接寻址，因此最多只有 1 个 IND 周期。立即数寻址的指令将跳过这一阶段。
- **EX(Execution):** 按照微操作指令指示打开数据通路。
- **WB(Write Back):** 将运算结果保存到目标寄存器。

注意到：对于所有的指令，前四个阶段的微操作指令是通用的，因此对每一条指令而言，只需要设计 EX 阶段和 WB 阶段的微操作指令即可，这大大缩小了 CM 所需空间。经设计，所有的微操作指令列举于表 6。

表 6: CPU 微操作指令表

指令	机器码	EX	WB
IF	阶段	$t_1: \text{MAR} \leftarrow \text{PC}; t_2: \text{MBR} \leftarrow \text{Mem}[\text{MAR}], \text{PC} \leftarrow \text{PC}+1$	
ID	阶段	$t_1: \text{IR} \leftarrow \text{MBR}; t_2: \text{CU} \leftarrow \text{IR}$	
FO	阶段	$\text{MBR} \leftarrow \text{IR}[7:0]$	
IND	阶段	$t_1: \text{MAR} \leftarrow \text{MBR}; t_2: \text{MBR} \leftarrow \text{Mem}[\text{MAR}]$	
STORE X	0001	$\text{MAR} \leftarrow \text{MBR};$	$\text{Mem}[\text{MAR}] \leftarrow \text{ACC}$

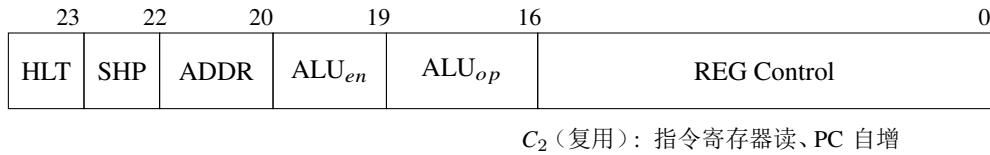
表 6: (续表) CPU 微操作指令表

指令	机器码	EX	WB
LOAD X	0010	无操作	ACC \leftarrow MBR
ADD X	0011	BR \leftarrow ACC + MBR	ACC \leftarrow BR
SUB X	0100	BR \leftarrow ACC - MBR	ACC \leftarrow BR
MPY X	1000	MR, BR \leftarrow ACC \times MBR	ACC \leftarrow BR
JGZ X	0101	判断: ZF=0 且 NF=0? 若满足, PC \leftarrow MBR, 否则 NOP)	
JMP X	0110	无操作	PC \leftarrow MBR
HALT	0111	无操作	停止程序
AND X	1001	BR \leftarrow ACC AND MBR	ACC \leftarrow BR
OR X	1010	BR \leftarrow ACC OR MBR	ACC \leftarrow BR
NOT X	1011	BR \leftarrow NOT MBR	ACC \leftarrow BR
SHIFTR X	1100	BR \leftarrow ACC \ggg X	ACC \leftarrow BR
SHIFTL X	1101	BR \leftarrow ACC \lll X	ACC \leftarrow BR

2.5.3 CU 控制信号 (Control Signals)

采用水平微指令 (Horizontal Micro-operation) 设计。水平微指令支持并行操作，执行效率高。每一个水平微指令携带所有控制信号位和下一个微操作指令地址的寻址方式。该 CPU 共有 24 位控制信号。其中低 16 位为寄存器控制信号，高 8 位为控制字。(图 3)

图 3: 控制信号示意图



各控制字的意义如下:

- HLT(HALT): 全局暂停控制字，所有 CPU 内部单元停止工作。
- SHP(Store High Part): 存储乘法寄存器高位结果到指定数据内存地址 +1。
- ADDR(Address): CU 内部控制字，共 2 位，指示下一步的地址为取指 (11) / 执行 (01) / 当前地址 +1 (10)。
- ALU_{en}: ALU 使能控制字，允许 ALU 进行运算操作。
- ALU_{op}: ALU 运算控制字 (3 位)，指示 ALU 执行的 8 种运算类型。运算类型编码可见 ALU 部分。
- REG_Control: 寄存器控制信号 (16 位)，每一位代表两个寄存器/总线之间的开关，对应关系见表 7。
- C₂: 复用控制字。除寄存器控制信号的功能外，还指示指令寄存器读、PC 自增。

关键存储单元之间通过数据通路进行连接。每条数据通路都由一位控制信号控制。控制信号为 1 时表示通路打开，数据沿指定流向进行传输。参考图 1 使用箭头标出的通路，分配控制信号位如表 7 所示。

表 7: 寄存器控制信号一览

控制信号位	源寄存器/单元	目的寄存器/单元
内部总线控制		
C_0	MAR	地址总线
C_1	PC	MBR
C_2	PC	MAR
C_3	MBR	PC
C_4	MBR	IR
C_5	数据总线	MBR
C_6	MBR	ALU_Q
C_7	ACC	ALU_P
C_8	MBR	MAR
C_9	BR	ACC
C_{10}	MR	ACC
C_{11}	MBR	ACC
C_{12}	ACC	MBR
C_{13}	MBR	数据总线
C_{14}	IR	CU
C_{15}	IR[7:0]	MBR

由上述的控制信号位设计，便可以将微操作指令一一对应，画出控制信号表（表 8）。控制信号表经过整合后写入 CM，结合 CU 的整体结构和合理的寻址设计，便能完成控制单元的设计。整合逻辑和寻址设计由于涉及到具体电路安排，详见模块设计部分，此处从略。

表 8: CPU 控制信号表

指令	机器码	EX	WB
IF	阶段	$t_1 : C_2, t_2 : C_0, C_5$	
ID	阶段	$t_1 : C_4, t_2 : C_{14}$	
FO	阶段	C_{15}	
IND	阶段	$t_1 : C_8, t_2 : C_0, C_5$	
STORE X	0001	C_8	C_0, C_{12}, C_{13}
LOAD X	0010	无操作	C_{11}
ADD X	0011	$C_6, C_7, ALU_{en}, ALU_{op}$	C_9
SUB X	0100	$C_6, C_7, ALU_{en}, ALU_{op}$	C_9
MPY X	1000	$C_6, C_7, ALU_{en}, ALU_{op}$	C_9
JGZ X	0101	判断: ZF=0 且 NF=0? 若满足, C_3 否则 NOP	
JMP X	0110	无操作	C_3
HALT	0111	无操作	HLT

表 8: (续表) CPU 控制信号表

指令	机器码	EX	WB
AND X	1001	$C_6, C_7, ALU_{en}, ALU_{op}$	C_9
OR X	1010	$C_6, C_7, ALU_{en}, ALU_{op}$	C_9
NOT X	1011	C_6, ALU_{en}, ALU_{op}	C_9
SHIFTR X	1100	$C_6, C_7, ALU_{en}, ALU_{op}$	C_9
SHIFTL X	1101	$C_6, C_7, ALU_{en}, ALU_{op}$	C_9

2.6 CPU 内部总线和外部总线

为了实现 CU 对 CPU 内部寄存器的控制，所有内部寄存器均连接到 CPU 内部总线。CU 可对 CPU 内部总线写控制信号，而所有内部寄存器通过读取内部总线中的某一位或几位控制信号，决定打开自身与某寄存器的数据通路。在本设计中，所有的控制信号作用于源寄存器，使得在控制信号关时，数据通路上没有来自源寄存器的数据，避免了可能的误读。例如：对于 PC 寄存器，其向 MAR、MBR 输出自身数据，并从 MBR 获取数据，三个行为分别由 C_1, C_2 和 C_3 控制，那么 PC 只需要读取 C_1, C_2 ，并在它们打开时输出自身寄存器的值。

MAR 和 MBR 寄存器是 CPU 与内存或外设的交互接口。由表 7 可知：他们连向了地址总线和数据总线，这两根总线合称外部总线。地址总线为 8 位单向总线，提供 CPU（即 MAR）到内存的地址传送通路。数据总线为 16 位双向总线，提供 CPU（即 MBR）与内存的双向数据通路。

外部总线还负责管理内存的读写以及选择读写内存设备，受到控制信号 C_0, C_2, C_5, C_{13} 的控制，他们被称为“控制总线”。由于指令和数据的物理存储空间不同，外部总线首先需要确定写入/读取的设备。在整个指令执行的流程中，仅 IF 阶段需要访问指令内存进行寻址，故该判决逻辑可通过复用控制信号的 C_2 完成。CPU 读内存时， C_0, C_5 开，故当且仅当两者同开时，总线可向选中的内存发出读信号，内存读地址总线，向数据总线输出相应地址的数据。CPU 写内存时， C_0, C_{13} 开，故当且仅当两者同开时，总线可向选中的内存发出写信号，内存读地址总线，读数据总线并存入对应地址。

3 外围设备

3.1 用户端代码解释

目前采用用户编写汇编代码 → 转换为 16 位机器码的方式输入指令。用户可在文本编辑器中编写类汇编代码，而编译器负责将其编译为机器码。

以从 1 加到 100 的程序举例：

```

1 LOAD IMMEDIATE 0
2 STORE IMMEDIATE 1
3 LOAD IMMEDIATE 1
4 STORE IMMEDIATE 2
5 LOOP: LOAD 1
6 ADD 2
7 STORE IMMEDIATE 1
8 LOAD 2
9 ADD IMMEDIATE 1
10 STORE IMMEDIATE 2

```

```

11 LOAD IMMEDIATE 101 ; faster
12 SUB 2
13 JGZ IMMEDIATE LOOP
14 LOAD 1
15 HALT          ; Expected 5050

```

代码最终将被解释为一串二进制比特流，解释服从：

- 地址占 1byte，Opcode 占 1byte；
- 含 IMMEDIATE 关键字的行，Opcode 的 MSB 为 1；
- 第一条指令的地址为 1，依次递增；
- 在汇编代码编译的过程中，标签（LOOP）应映射到相同行指令的地址，在这个程序里，它对应地址 5；
- 一行内分号后的内容为注释，编译器会忽略它。

3.2 UART 接收与指令内存写入设计

本设计基于 NEXYS 4 DDR 开发板，通过其板载的 UART 接口完成主机与 FPGA 之间的数据传输。该方案无需额外的数据线或 I/O 资源，直接复用 JTAG 烧录线即可实现对指令内存的程序写入，提升了系统的硬件集成度与使用便捷性。

3.2.1 UART 接收逻辑

开发板主系统时钟频率为 100 MHz，串口通信波特率设定为 115 200 bps。根据 UART 通信协议，每接收 1 位数据所需的时钟周期数为：

$$CLK_BAUD = \frac{CLK_FREQ}{BAUD_RATE} = \frac{100\,000\,000}{115200} \approx 868 \quad (1)$$

采用常见的 **8N1** 格式传输，即每帧包括：

- 1 位起始位（Start Bit）；
- 8 位数据位（Data Bits）；
- 1 位停止位（Stop Bit）。

因此，每帧共 10 位，总计需要约：

$$CLK_FRAME = 10 \times CLK_BAUD = 10 \times 868 = 8680 \text{ cycles} \quad (2)$$

接收端在每位开始时执行采样，认为超过 0.5 秒 RX 端仍无新数据填入时，指令传输完成。CPU 会向用户发出提示，指示接收完成。

3.2.2 数据缓冲与存储结构

为保证串口数据完整接收，接收模块首先将每帧数据写入异步 FIFO 缓冲区。FIFO 将两帧数据合并在一起，以完整指令格式由控制逻辑从 FIFO 中读取数据，并写入指令内存。

数据写入格式：

- RAM 的每个地址对应两个字节（16 位）数据；
- 高字节为操作码（Opcode），低字节为立即数或地址（Operand）；
- 若当前行含有 IMMEDIATE 关键字，则 Opcode 的最高位（MSB）为 1。

写入控制规则：

- 每一条指令都为 2byte 指令，对于没有操作数的情况（HALT、JMP），将操作数位置补零。
- RAM 地址从地址 1 开始顺序写入。
- 程序中如含有 LOOP 标签，将在 RAM 地址分配完毕后由 python 程序在解析阶段回填其地址位置。

另外，传入 FPGA 的所有指令将存于单独的指令内存中，与 CPU 数据内存隔离开来。CPU 内存仅存数据，这符合用户编写的直观感受。这种方法通过串口实现了简洁的二进制指令数据装载方式，降低了外设复杂性。

3.3 数据内存

数据内存（RAM）存储 CPU 保存的数据。内存的大小为 512 Byte，每条存入内存的数据位宽为 16，共能存入 256 条数据。数据内存初始为空，起始写入地址为 0，采用 Little Endian 写入方式²。

CPU 与数据内存通过三条总线交互，分别为控制总线、地址总线和数据总线。控制总线中的控制信号决定在这个周期中内存的读/写状态，是否向数据总线写入，同步时序等功能。内存通过读取地址总线决定写入内存中的地址，通过读取数据总线决定写入指定地址中的数据。关于总线的具体配置见 2.6。

3.4 用户交互设计

用户可以通过 FPGA 板上的按钮和开关与 CPU 进行交互，并通过 LED 灯和七段数码管查看 CPU 的运行状态和计算结果。具体设计如下：

全局复位 按下上侧按钮（BTNH）时，CPU 触发全局复位信号，所有寄存器和内存数据清空，控制信号复位为初始状态。

运行与停止 最右侧开关控制 CPU 的运行状态：

- 开启状态：CPU 开始执行指令。
- 关闭状态：CPU 停止运行并保持当前状态。

当 CPU 发出停止信号（HALT）时，右侧 LED 灯亮红色；当 CPU 处于运行状态时，右侧 LED 灯亮蓝色。

运行模式与单步调试模式 CPU 支持两种模式：

- **运行模式**：CPU 自动连续执行指令直至停止，适合快速验证结果。
- **单步调试模式**：用户通过按下最中间的按钮（BTNC）控制 CPU 逐条执行指令。每次执行完一条指令后，CPU 暂停，等待用户操作。

CPU 运行开关左侧相邻开关控制模式选择：

- 开启状态：单步调试模式，左侧 LED 灯亮红色。
- 关闭状态：运行模式，左侧 LED 灯亮蓝色。

结果与指令查看

- 按下左侧按钮（BTNL）：查看当前运算结果。
- 按下右侧按钮（BTNR）：查看当前指令码和指令地址。
- 按下下侧按钮（BTND）：查看当前 Flags 寄存器的值。

²即高位存储于高地址，低位存储于低地址。

4 核心模块设计

4.1 时钟、复位与停止信号

CPU 由全局同步时钟控制，时钟主频为 100MHz。除异步复位信号外，所有分频时钟、控制逻辑与计算逻辑全部在该时钟上升沿进行。

CPU 设有全局异步复位信号，低电平有效。当异步复位时，内存中除指令集数据以外所有数据清空，所有寄存器清空，控制信号全部归为断开（0）。

当 CPU 执行 07 号指令 HALT 时，CPU 处于停止状态，发出 HLT 信号。与复位不同的是，此时所有寄存器不清空，但所有通路断开。解除该状态的唯一方法是全局复位。

4.2 UART 传输与指令内存

指令写入通过 Python 脚本（附录 A.1）完成，其可以根据用户编写的 txt 格式汇编代码（格式参照 3.1）输出一串 UART 格式的比特流。用户可在 python 程序中设置连接 UART 端口的 USB 端口，通过 PySerial 串口通信库进行传输，或复制脚本的输出作为输入测试样例。

模块代码：见附录 A.2。

功能块基本信息：

- 模块名：INSTR_ROM
- 最新更新日期：4.14
- 是否经过测试：是

功能块外部接口：

表 9：指令内存模块外部接口

信号名	方向	位宽	描述
i_clk_uart	输入	1	时钟信号（100MHz）
i_RST_N	输入	1	全局复位信号
i_RX	输入	1	绑定至 UART 接收引脚
i_ADDR_READ	输入	8	读 RAM 地址
o_INSTR_READ	输出	16	指令输出信号
o_INSTR_TRANSMIT_DONE	输出	1	指令完成传入标志
o_MAX_ADDR	输出	8	最大地址输出

4.2.1 UART 模块

模块基本信息：

- 模块名：UART
- 最新更新日期：4.30
- 是否经过测试：是

模块功能：将用户输入代码比特流（8N1 格式）译码为 1 字节数据。分频时钟代码见 clk_divider.v。

模块外部接口：

表 10: UART 模块外部接口

信号名	方向	位宽	描述
i_clk	输入	1	分频后的时钟信号
i_RST_N	输入	1	全局复位信号
i_RX	输入	1	UART 接收引脚
o_data	输出	8	接收到的一帧数据
o_valid	输出	1	数据有效标志，高电平表示 o_data 已生成
o_clear_sign	输出	1	表示 UART 输入结束（第一次输入结束后 0.5 秒内无新的输入）

模块行为：

- 当 i_RST_N 为低电平时，模块复位，状态机进入 START 状态，清空接收寄存器和计数器；
- 在 START 状态下，若 i_RX 信号检测到起始位（低电平），状态机在下一波特率时钟进入 DATA 状态；
- 在 DATA 状态下，在波特率时钟下降沿采样数据，依次存入移位寄存器，使得其可在信号位中点采样数据，增加稳定性；
- 在接收完 8 位数据后，状态机进入 STOP 状态，检测停止位（高电平）；
- 若停止位有效，输出接收到的 8 位数据到 o_data，并将 o_valid 置高，表示数据有效；状态机在完成当前帧的接收后返回 START 状态，等待下一帧数据。
- 若在 START 状态下超过 0.5 秒未接收到新数据，o_clear_sign 置高，表示 UART 输入结束；

4.2.2 FIFO 模块

模块基本信息：

- 模块名：FIFO
- 最新更新日期：4.13
- 是否经过测试：是

模块功能： 异步 FIFO，缓存 UART 数据并处理 UART 模块和 CPU 主模块的跨时钟域问题。它将每两个读出的 UART 数据拼成 2 字节的指令输出给 BRAM。

模块外部接口：

表 11: FIFO 模块外部接口

信号名	方向	位宽	描述
i_RST_N	输入	1	异步复位信号，低有效
i_CLK_WR	输入	1	写时钟信号，UART 使用的波特率时钟
i_VALID_UART	输入	1	表示当前 UART 输入数据有效
i_DATA_UART	输入	8	UART 接收到的 8 位数据字节
i_CLK_RD	输入	1	读时钟信号，使用 100MHz 时钟
o_DATA_BRAM	输出	16	两个 UART 字节拼接后的数据，写入 BRAM
o_ADDR_BRAM	输出	8	BRAM 写入地址，从 0 开始自增

(续表) FIFO 模块外部接口

信号名	方向	位宽	描述
<code>o_wr_en_bram</code>	输出	1	BRAM 写使能，高电平表示写入有效
<code>o_fifo_empty</code>	输出	1	表示 FIFO 空（作为输入完成的判据）

模块行为：

- 当 `i_rst_n` 为低电平时，FIFO 复位：
 - 写指针 `wr_ptr_bin` 和读指针 `rd_ptr_bin` 清零；
 - FIFO 存储器 `fifo_mem` 清空；
 - 输出信号 `o_data_bram`、`o_addr_bram` 和 `o_wr_en_bram` 复位为 0。
- 当 `i_valid_uart` 信号有效时：
 - 若 FIFO 未满 (`!fifo_full`)，将 `i_data_uart` 写入 `fifo_mem` 中 `wr_ptr_bin` 指向的地址；
 - 写指针 `wr_ptr_bin` 递增，并更新为 Gray 码 `wr_ptr_gray`。
- 在读时钟域 (`i_clk_rd`)：
 - 若 FIFO 非空 (`!fifo_empty`)，从 `fifo_mem` 中 `rd_ptr_bin` 指向的地址读取数据；
 - 若当前为第一个字节 (`byte_flag` 为 0)，将数据存入缓冲区 `data_buffer`；
 - 若当前为第二个字节 (`byte_flag` 为 1)，将缓冲区 `data_buffer` 与当前读取的数据拼接为 16 位数据，输出到 `o_data_bram`；
 - 读指针 `rd_ptr_bin` 递增，并更新为 Gray 码 `rd_ptr_gray`；
 - 输出写使能信号 `o_wr_en_bram`，并将地址 `o_addr_bram` 递增。
- 当 FIFO 为空时，`o_fifo_empty` 置高，表示数据传输完成。

备注：设计思路参照文献[1]。

4.2.3 指令 BRAM 模块

模块基本信息：

- 模块名：BRAM_INSTR
- 最新更新日期：4.28
- 是否经过测试：否

模块功能： 描述一指令块 RAM，可存放 256 条 2byte 指令。读写双口，拥有写使能（FIFO 传入）。外部设备可通过开读使能信号并传入地址读取对应地址的 2byte 指令。

模块外部接口：

表 12：指令 BRAM 模块外部接口

信号名	方向	位宽	描述
<code>i_clk</code>	输入	1	系统时钟信号，驱动读写操作
<code>en_write</code>	输入	1	写使能信号，高电平时允许将指令写入 BRAM
<code>en_read</code>	输入	1	读使能信号，高电平时允许外部总线从 BRAM 中读取指令
<code>i_addr_write</code>	输入	8	要写入的指令地址
<code>i_instr_write</code>	输入	16	要写入的指令内容

(续表) 指令 BRAM 模块外部接口

信号名	方向	位宽	描述
i_addr_read	输入	8	要读取的指令地址
o_instr_read	输出	16	从 BRAM 中读取的指令内容

模块行为：

- 在 en_write 有效时，将指令内容写入指令地址；
- 在 en_read 有效时，将要读取的指令地址中的内容输出到连接外部总线的 o_instr_read。

4.3 控制单元

控制单元由 CAR、CBR 寄存器和 CM (Control Memory) 只读模块组成。控制单元通过将存储于 CM 的微操作指令和控制信号输出至 CBR 后，再通过内部控制总线输出到各个单元（寄存器、ALU、外部总线）控制整个系统。控制单元每个时钟周期执行一条微操作指令，由表 6 可知平均每条指令需要执行 8 条微操作指令，故每条指令约需要 8 个周期执行完成。

4.3.1 Control Memory

模块基本信息：

- 模块名：CONTROL_MEMORY
- 最新更新日期：4.23
- 是否经过测试：否

模块功能： 存储 CPU 的水平微操作指令，并根据输入的微操作指令地址写出控制信号到 CBR。微操作指令表参考表 8。另外，为了更好地支持乘法后存储高位、跳转补全周期等操作，CM 还存储了两条非指令集中的指令：NOP 和 STOREH。它们的作用分别是：

- NOP：无操作指令，CPU 在执行该指令时不进行任何操作。该指令的作用是占位，保证 JGZ 指令可以和其余指令执行时间相同。
- STOREH：存储高位指令，在上一条指令为乘法时，若本次指令为 STORE，则在存放 ACC 寄存器后，继续将 MR 寄存器的高位通过 ACC 寄存器存入地址 +1 位置的内存。该指令的作用是将乘法运算的高位低位结果都存储到内存中。

模块外部接口：

表 13: Control Memory 模块外部接口

信号名	方向	位宽	描述
car	输入	7	要读取的微操作指令地址
control_word	输出	24	从 CM 中读取的控制信号

模块行为： 输出 car 地址存储的 control_word。

4.3.2 CAR

模块基本信息：

- 模块名：CAR
- 最新更新日期：4.23
- 是否经过测试：是

模块功能： 根据控制信号中 ADDR 控制字、IR Opcode 和 Flags，综合判断出下一条指令所在微操作指令的地址。

模块外部接口：

表 14: CAR 模块外部接口

信号名	方向	位宽	描述
i_clk	输入	1	时钟信号
i_rst_n	输入	1	全局复位信号
ctrl_cpu_start	输入	1	用户面控制，控制 CPU 启动与停止
ctrl_step_execution	输入	1	用户面控制，控制 CPU 逐条执行指令
i_next_instr_stimulus	输入	1	用户面控制，控制 CPU 执行下一条指令
i_ctrl_halt	输入	1	HLT 控制字
i_control_word_car	输入	2	ADDR 控制字（参照 2.5.3）
i_ir_data	输入	5	IR 的最高位 + IR 的低四位
i_ctrl_ZF	输入	1	ZF 值
i_ctrl_NF	输入	1	NF 值
i_ctrl_MF	输入	1	MF 值
o_car_data	输出	7	待取指令在 CM 中的地址

模块行为：

- 时钟上升沿时，若 i_ir_data 有效，从外部更新 ir_data 寄存器的值；
- 根据 ir_data 最高位判断是否需要 IND 周期，若最高位为 0 则置 indirect_flag 为 1，标志需要 IND 周期；
- 根据 ADDR 控制字决定下一条微操作指令地址：
 - 跳转 (01): indirect_flag 为 1 时，将 IND 周期的微指令地址写入 CAR，否则进入 EX 阶段，根据 ir_data 低四位决定具体执行指令，并将微指令地址写入 CAR；
 - 自增 (10): CAR 自增；
 - 指令结束 (11): 若 HLT 为 1，CAR 保持不变；若在单步调试功能下，CAR 跳转到 NOP 的 WB 阶段，等待 i_next_instr_stimulus 信号将 CAR 归零，否则 CAR 归零，自动获取下一条指令。

4.4 内部寄存器和 ALU 设计

4.4.1 ALU

ALU (Arithmetic Logic Unit) 是算术逻辑单元，负责执行 CPU 中的算术和逻辑运算。

模块基本信息：

- 模块名：ALU
- 最新更新日期：2025.4.25
- 是否经过测试：是

模块功能： ALU 模块支持以下功能：

- 在控制信号 `ctrl_alu_en` 有效时，根据 `ctrl_alu_op` 执行指定的运算；
- 支持的运算包括加法、减法、乘法、按位与、按位或、按位非、算术左移和算术右移；
- 运算结果存储在 BR（低 16 位）和 MR（高 16 位）寄存器中；
- 更新状态寄存器（Flags），包括 ZF（零标志）、CF（进位标志）、OF（溢出标志）、NF（负数标志）和 MF（乘法标志）。

模块外部接口：

表 15: ALU 模块外部接口

信号名	方向	位宽	描述
<code>i_clk</code>	输入	1	系统时钟信号
<code>i_RST_N</code>	输入	1	全局复位信号
<code>i_acc_alu_p</code>	输入	16	ALU 的第一个操作数
<code>i_acc_alu_q</code>	输入	16	ALU 的第二个操作数
<code>ctrl_alu_op</code>	输入	3	运算类型控制信号
<code>ctrl_alu_en</code>	输入	1	ALU 使能信号
<code>o_BR</code>	输出	16	运算结果的低 16 位
<code>o_MR</code>	输出	16	运算结果的高 16 位（仅乘法有效）
<code>o_flags</code>	输出	5	状态寄存器（ZF, CF, OF, NF, MF）
<code>i_user_sample</code>	输入	1	用户采样信号
<code>o_MR_user</code>	输出	16	输出到用户接口的 MR 寄存器数据

模块行为：

- 当 `i_RST_N` 为低电平时，BR 和 MR 寄存器复位为 `16'b0`，Flags 寄存器复位为 `5'b0`；
- 当 `ctrl_alu_en` 信号有效时，根据 `ctrl_alu_op` 执行以下运算：
 - 000: 加法（ADD），结果存入 BR（若高 16 位存在，则存入 MR 和 BR）；
 - 001: 减法（SUB），结果存入 BR（若高 16 位存在，则存入 MR 和 BR）；
 - 010: 乘法（MPY），低 16 位存入 BR，高 16 位存入 MR；
 - 011: 按位与（AND），结果存入 BR；
 - 100: 按位或（OR），结果存入 BR；
 - 101: 按位非（NOT），结果存入 BR；
 - 110: 算术右移（SHIFTR），结果存入 BR；
 - 111: 算术左移（SHIFTL），结果存入 BR。
- 根据运算结果更新 Flags 寄存器：
 - ZF：结果为 0 时置 1；
 - CF：移位操作时存储移出的位；
 - OF：加法或减法溢出时置 1；

- NF: 结果为负数时置 1;
- MF: 乘法运算时置 1。
- 当 `i_user_sample` 信号有效时, 将 MR 寄存器的数据输出到 `o_mr_user`。
- 在其他情况下, BR 和 MR 寄存器保持当前值。

4.4.2 MAR

MAR (Memory Address Register) 是内存地址寄存器, 用于存储当前访问的内存地址。它通过地址总线与内存交互, 决定内存的读写地址。

模块基本信息:

- 模块名: MAR
- 最新更新日期: 2025.4.25
- 是否经过测试: 是

模块功能: MAR 模块支持以下功能:

- 在控制信号 C8 打开时, 从 MBR 寄存器读取地址;
- 在控制信号 C2 打开时, 从 PC 寄存器读取地址;
- 在 `ctrl_mar_increment` 信号有效时, 自增地址 (用于支持 STOREH 指令的高位存储操作)。

模块外部接口:

表 16: MAR 模块外部接口

信号名	方向	位宽	描述
<code>i_clk</code>	输入	1	系统时钟信号
<code>i_RST_N</code>	输入	1	全局复位信号
<code>i_mbr_mar</code>	输入	8	从 MBR 传入的地址
<code>i_pc_mar</code>	输入	8	从 PC 传入的地址
<code>ctrl_mar_increment</code>	输入	1	自增控制信号
C2	输入	1	控制信号, 允许从 PC 读取地址
C8	输入	1	控制信号, 允许从 MBR 读取地址
<code>o_mar_address_bus</code>	输出	8	输出到地址总线的地址

模块行为:

- 当 `i_RST_N` 为低电平时, MAR 寄存器复位为 8'b0;
- 当 `ctrl_mar_increment` 信号有效时, MAR 寄存器的值自增;
- 当 C8 信号有效时, MAR 从 `i_mbr_mar` 读取地址;
- 当 C2 信号有效时, MAR 从 `i_pc_mar` 读取地址;
- 在其他情况下, MAR 保持当前值。

4.4.3 MBR

MBR (Memory Buffer Register) 是内存缓冲寄存器, 用于存储从内存读取或写入的数据。它通过数据总线与内存交互, 是内存与 CPU 之间的数据桥梁。

模块基本信息：

- 模块名：MBR
- 最新更新日期：2025.4.25
- 是否经过测试：是

模块功能： MBR 模块支持以下功能：

- 在控制信号 C5 打开时，从数据总线读取数据；
- 在控制信号 C13 打开时，将数据写入数据总线；
- 在控制信号 C6 打开时，将数据传输到 ALU 的 ALU_Q 输入端；
- 在控制信号 C8 打开时，将数据传输到 MAR 寄存器；
- 在控制信号 C11 打开时，将数据传输到 ACC 寄存器。

模块外部接口：

表 17: MBR 模块外部接口

信号名	方向	位宽	描述
i_clk	输入	1	系统时钟信号
i_RST_N	输入	1	全局复位信号
i_data_bus	输入	16	从数据总线传入的数据
o_data_bus	输出	16	输出到数据总线的数据
o_alu_q	输出	16	输出到 ALU 的 ALU_Q 输入端的数据
o_mar	输出	8	输出到 MAR 寄存器的地址
o_acc	输出	16	输出到 ACC 寄存器的数据
C5	输入	1	控制信号，允许从数据总线读取数据
C13	输入	1	控制信号，允许将数据写入数据总线
C6	输入	1	控制信号，允许将数据传输到 ALU_Q
C8	输入	1	控制信号，允许将数据传输到 MAR
C11	输入	1	控制信号，允许将数据传输到 ACC

模块行为：

- 当 i_RST_N 为低电平时，MBR 寄存器复位为 16'b0；
- 当 C5 信号有效时，MBR 从 i_data_bus 读取数据；
- 当 C13 信号有效时，MBR 将数据输出到 o_data_bus；
- 当 C6 信号有效时，MBR 将数据输出到 o_alu_q；
- 当 C8 信号有效时，MBR 将数据输出到 o_mar；
- 当 C11 信号有效时，MBR 将数据输出到 o_acc；
- 在其他情况下，MBR 保持当前值。

4.4.4 PC

PC (Program Counter) 是程序计数器，用于存储当前指令的地址，并在指令执行后指向下一条指令的地址。

模块基本信息：

- 模块名：PC
- 最新更新日期：2025.4.25
- 是否经过测试：是

模块功能： PC 模块支持以下功能：

- 在控制信号 C1 打开时，将当前地址传输到 MBR 寄存器；
- 在控制信号 C2 打开时，自增地址（用于指令取址阶段）；
- 在控制信号 C3 打开时，从 MBR 寄存器读取地址；
- 在复位信号有效时，PC 寄存器复位为 8'd1。

模块外部接口：

表 18: PC 模块外部接口

信号名	方向	位宽	描述
i_clk	输入	1	系统时钟信号
i_RST_N	输入	1	全局复位信号
i_MBR_PC	输入	8	从 MBR 传入的地址
C1	输入	1	控制信号，允许将当前地址传输到 MBR
C2	输入	1	控制信号，允许 PC 自增
C3	输入	1	控制信号，允许从 MBR 读取地址
o_PC_MAR	输出	8	输出到 MAR 寄存器的地址
o_PC_MBR	输出	8	输出到 MBR 寄存器的地址
o_PC_USER	输出	8	输出到用户接口的地址

模块行为：

- 当 i_RST_N 为低电平时，PC 寄存器复位为 8'd1；
- 当 C2 信号有效时，PC 寄存器的值自增；
- 当 C3 信号有效时，PC 从 i_MBR_PC 读取地址；
- 当 C1 信号有效时，PC 将当前地址输出到 o_PC_MBR；
- 在其他情况下，PC 保持当前值。

4.4.5 IR

IR (Instruction Register) 是指令寄存器，用于存储当前正在执行的指令，并将指令的操作码和操作数分离，供控制单元和其他模块使用。

模块基本信息：

- 模块名：IR
- 最新更新日期：2025.4.25
- 是否经过测试：是

模块功能： IR 模块支持以下功能：

- 在控制信号 C4 打开时，从 MBR 寄存器读取指令；
- 在控制信号 C14 打开时，将操作码输出到控制单元（CU）；
- 在控制信号 C15 打开时，将操作数输出到 MBR 寄存器；
- 在用户采样信号有效时，将操作码输出到用户接口。

模块外部接口：

表 19: IR 模块外部接口

信号名	方向	位宽	描述
i_clk	输入	1	系统时钟信号
i_RST_N	输入	1	全局复位信号
i_MBR_IR	输入	16	从 MBR 传入的指令
C4	输入	1	控制信号，允许从 MBR 读取指令
C14	输入	1	控制信号，允许将操作码输出到 CU
C15	输入	1	控制信号，允许将操作数输出到 MBR
i_USER_SAMPLE	输入	1	用户采样信号
o_IR CU	输出	8	输出到 CU 的操作码
o_IR_MBR	输出	8	输出到 MBR 的操作数
o_IR_USER	输出	8	输出到用户接口的操作码

模块行为：

- 当 i_RST_N 为低电平时，IR 寄存器复位为 8'b0；
- 当 C4 信号有效时，从 i_MBR_IR 读取指令，并将高 8 位存储为操作码，低 8 位存储为操作数；
- 当 C14 信号有效时，将操作码输出到 o_IR CU；
- 当 C15 信号有效时，将操作数输出到 o_IR_MBR；
- 当 i_USER_SAMPLE 信号有效时，将操作码输出到 o_IR_USER；
- 在其他情况下，IR 保持当前值。

4.4.6 ACC

ACC (Accumulator) 是累加寄存器，用于存储 ALU 运算的结果低 16 位，并作为 ALU 的输入之一。

模块基本信息：

- 模块名：ACC
- 最新更新日期：2025.4.25
- 是否经过测试：是

模块功能： ACC 模块支持以下功能：

- 在控制信号 C9 打开时，从 BR 寄存器读取数据；
- 在控制信号 C10 打开时，从 MR 寄存器读取数据；
- 在控制信号 C11 打开时，从 MBR 寄存器读取数据；
- 在控制信号 C12 打开时，将数据传输到 MBR 寄存器；
- 在控制信号 C7 打开时，将数据传输到 ALU 的 ALU_P 输入端；

- 在用户采样信号有效时，将数据输出到用户接口。

模块外部接口：

表 20: ACC 模块外部接口

信号名	方向	位宽	描述
i_clk	输入	1	系统时钟信号
i_RST_N	输入	1	全局复位信号
i_BR_ACC	输入	16	从 BR 传入的数据
i_MR_ACC	输入	16	从 MR 传入的数据
i_MBR_ACC	输入	16	从 MBR 传入的数据
C7	输入	1	控制信号，允许将数据传输到 ALU_P
C9	输入	1	控制信号，允许从 BR 读取数据
C10	输入	1	控制信号，允许从 MR 读取数据
C11	输入	1	控制信号，允许从 MBR 读取数据
C12	输入	1	控制信号，允许将数据传输到 MBR
i_USER_SAMPLE	输入	1	用户采样信号
o_ACC_ALU_P	输出	16	输出到 ALU 的 ALU_P 输入端的数据
o_ACC_MBR	输出	16	输出到 MBR 的数据
o_ACC_USER	输出	16	输出到用户接口的数据

模块行为：

- 当 i_RST_N 为低电平时，ACC 寄存器复位为 16'b0;
- 当 C9 信号有效时，ACC 从 i_BR_ACC 读取数据;
- 当 C10 信号有效时，ACC 从 i_MR_ACC 读取数据;
- 当 C11 信号有效时，ACC 从 i_MBR_ACC 读取数据;
- 当 C12 信号有效时，ACC 将数据输出到 o_ACC_MBR;
- 当 C7 信号有效时，ACC 将数据输出到 o_ACC_ALU_P;
- 当 i_USER_SAMPLE 信号有效时，ACC 将数据输出到 o_ACC_USER;
- 在其他情况下，ACC 保持当前值。

4.5 数据内存设计

数据内存 (Data RAM) 是用于存储 CPU 运行过程中产生的数据的存储单元。它通过外部总线与 CPU 交互，支持数据的读写操作。

模块基本信息：

- 模块名：DATA_RAM
- 最新更新日期：2025.4.25
- 是否经过测试：是

模块功能： 数据内存模块支持以下功能：

- 根据输入的写地址和数据，在写使能信号有效时，将数据写入指定地址；
- 根据输入的读地址，在读使能信号有效时，从指定地址读取数据；

- 支持 256 条 16 位数据的存储。

模块外部接口：

表 21: 数据内存模块外部接口

信号名	方向	位宽	描述
i_clk	输入	1	系统时钟信号
i_RST_N	输入	1	全局复位信号
ctrl_write	输入	1	写使能信号，高电平表示写入有效
i_addr_write	输入	8	写入地址
i_data_write	输入	16	写入的数据
ctrl_read	输入	1	读使能信号，高电平表示读取有效
i_addr_read	输入	8	读取地址
o_data_read	输出	16	读取的数据

模块行为：

- 当 i_RST_N 为低电平时，数据内存保全部清空并置 0；
- 当 ctrl_write 信号有效时，在时钟上升沿将 i_data_write 写入 i_addr_write 指定的地址；
- 当 ctrl_read 信号有效时，从 i_addr_read 指定的地址读取数据，并输出到 o_data_read；
- 在其他情况下，数据内存保持当前值。

4.6 外部总线设计

外部总线（External Bus）是 CPU 与内存或外设之间的数据交互桥梁，负责管理地址传输、数据传输以及内存设备的选择。

模块基本信息：

- 模块名：EXTERNAL_BUS
- 最新更新日期：2025.4.25
- 是否经过测试：是

模块功能： 外部总线模块支持以下功能：

- 根据控制信号决定内存的读写操作；
- 通过地址总线传输内存地址；
- 通过数据总线传输数据；
- 根据控制信号选择访问指令内存（ROM）或数据内存（RAM）。³

模块外部接口：

³尽管指令内存是可写的，但由于指令内存初次写入后，内容不会在复位前发生变化，因此在此处将其视为只读存储器。

表 22: 外部总线模块外部接口

信号名	方向	位宽	描述
i_clk	输入	1	系统时钟信号
i_rst_n	输入	1	全局复位信号
i_mbr_data_bus	输入	16	从 MBR 传入的数据
i_mar_address_bus	输入	8	从 MAR 传入的地址
i_instr	输入	16	从指令内存读取的数据
i_data	输入	16	从数据内存读取的数据
o_data_bus_mbr	输出	16	输出到 MBR 的数据
o_data_bus_memory	输出	16	输出到内存的数据
o_address_bus_memory	输出	8	输出到内存的地址
o_instr_rom_read	输出	1	指令内存读使能信号
o_data_ram_read	输出	1	数据内存读使能信号
o_data_ram_write	输出	1	数据内存写使能信号
C0	输入	1	控制信号, 允许地址总线传输
C2	输入	1	控制信号, 选择指令内存
C5	输入	1	控制信号, 允许数据总线读取
C13	输入	1	控制信号, 允许数据总线写入

模块行为:

- 当 i_rst_n 为低电平时, 所有输出信号复位为 0;
- 当 C0 信号有效时, 地址总线传输 i_mar_address_bus 的值;
- 当 C5 和 C0 信号同时有效时:
 - 若 C2 信号有效, 读取指令内存, o_instr_rom_read 置 1, o_data_bus_mbr 输出 i_instr;
 - 若 C2 信号无效, 读取数据内存, o_data_ram_read 置 1, o_data_bus_mbr 输出 i_data.
- 当 C13 和 C0 信号同时有效时:
 - 数据总线传输 i_mbr_data_bus 的值;
 - o_data_ram_write 置 1, 写入数据内存。
- 在其他情况下, 所有输出信号保持为 0。

4.7 用户面设计

所有来自用户的输入信号（按钮、开关）在按键消抖后传递给 CPU 和外设控制逻辑，所有来自 CPU 的输出信号通过 FPGA 外设（LED 灯、七段显示器）控制逻辑传递给用户。

4.7.1 按键消抖模块

按键消抖模块（KEY_JITTER）用于对用户输入的按钮或开关信号进行消抖处理，确保输入信号的稳定性，避免因机械抖动导致的误触发。

模块基本信息:

- 模块名: KEY_JITTER
- 最新更新日期: 2025.4.25

- 是否经过测试: 是

模块功能: KEY_JITTER 模块支持以下功能:

- 对输入的按键信号进行采样和稳定性检测;
- 通过计数器延迟, 过滤掉机械抖动产生的短暂信号变化;
- 输出稳定的按键信号, 供其他模块使用。

模块外部接口:

表 23: KEY_JITTER 模块外部接口

信号名	方向	位宽	描述
i_clk	输入	1	系统时钟信号
key_in	输入	1	原始按键输入信号
key_out	输出	1	消抖后的稳定按键信号

模块行为:

- 按键输入信号 key_in 通过两级寄存器进行采样, 检测信号变化;
- 当检测到信号变化时, 计数器 cnt_base 清零并开始计数;
- 如果信号在计数器达到最大值 CNT_MAX 之前保持稳定, 则更新输出信号 key_out;
- 如果信号在计数器计满之前发生变化, 则重新开始计数;
- 输出信号 key_out 仅在信号稳定后更新, 根据 POSEDGE 参数决定是否仅在上升沿触发。

模块参数:

- CNT_MAX: 计数器的最大值, 用于设置消抖时间窗口。默认值为 20'hf_ffff, 对应较长的消抖时间。此参数可在最上层模块自定义。
- POSEDGE: 用于设置输出是否仅在按键上升沿触发 (冲激信号)。对于开关类型的信号, POSEDGE 应设置为 0, 以捕获持续的高电平信号; 对于按钮类型的信号, POSEDGE 应设置为 1, 以捕获按键按下的瞬时信号。

4.7.2 七段显示器

顶层模块基本信息:

- 模块名: SEVEN_SEGMENT_DISPLAY
- 最新更新日期: 2025.4.25
- 是否经过测试: 是

显示模块基本信息:

- 模块名: SEVEN_SEGMENT
- 最新更新日期: 2025.4.25
- 是否经过测试: 是

模块功能: 七段显示器使用两个模块实现, 一个为显示编码模块, 用于将数字转换为七段显示器的编码, 并输出轮询信号用于更新七段显示器的显示; 另外一个为顶层模块, 根据用户请求将待显示的数字传入显示模块。

模块外部接口：

表 24: KEY_JITTER 模块外部接口

信号名	方向	位宽	描述
i_clk	输入	1	系统时钟信号
key_in	输入	1	原始按键输入信号
key_out	输出	1	消抖后的稳定按键信号

4.7.3 LED 灯显示

4.8 时序优化

经由时序分析工具分析，CPU 的时序路径集中在 ALU 和 CAR 模块之间，尤其是 NF 和 ZF 上。如果发生时序违例，可能会导致数据错误或系统不稳定。为了解决这个问题，我们对时序路径进行了优化，确保在时钟周期内完成数据传输和处理。

时序路径分析： 整个系统由 CAR 模块控制，对于内存操作，CAR 只需向外部总线发送控制信号，然后由外部总线转发给内存即可完成读写操作。而 ALU 作为纯组合逻辑单元，其内部的 Flags 寄存器依赖于 ALU 的计算结果，同时 ALU 的输入又取决于 CAR 的控制信号和内部寄存器的值。因此，从 CAR 到 Flags 寄存器的组合逻辑路径比到内存模块的路径更长。此外，ZF 和 NF 寄存器还需要判断高 16 位是否为零，这引入了大量额外的组合逻辑，导致时序路径更为复杂。基于以上分析，我们需要对 Flags 寄存器的时序路径进行专门优化。

时序优化方法： 由于某条指令产生的 Flags 必然不会在本指令中使用，且更新 Flags 的指令必然使用 ALU，也就必然拥有使用 ALU 的 EX 和 WB 阶段，因此 Flags 可以使用本条指令的 EX 和 WB 阶段的时钟沿来更新。

按照判决 Flags 所需要的输入信号类型，可以将 Flags 分为两类：

- 需要 ALU_P 和 ALU_Q: OF、CF；
- 不需要 ALU_P 和 ALU_Q: ZF、NF、MF。

对于第一类 Flags，ALU 的输出在 EX 阶段更新到 Flags 寄存器；对于第二类 Flags，ALU 的输出在 WB 阶段更新到 Flags 寄存器，它们的判决不再依赖于 ALU 的输出，而是依赖于寄存了 ALU 输出的 MR、BR 寄存器。这样，Flags 寄存器的更新时序路径就可以分为两条，减轻了时序路径的压力。

- ALU 的输出在 EX 阶段更新到 Flags 寄存器，时序路径为：ALU → Flags (OF、CF)
- ALU 的输出在 WB 阶段更新到 Flags 寄存器，时序路径为：ALU → MR/BR → Flags (ZF、NF、MF)

时序优化结果： 经过上述时序优化，本设计的时序余量从原来的 0.1ns 提升至 0.5ns，时序余量提升了 400%。

5 性能分析与功能验证

5.1 时序分析

使用 Vivado 2024.2 进行 FPGA 时序分析，结果如图 4 所示。

设计的时序分析结果显示，在工作频率为 100MHz 时，所有路径均能在一个时钟周期内完成，这表明设计在 FPGA 上运行稳定，满足了设计要求。

图 4: FPGA 时序分析

Design Timing Summary											
Setup			Hold			Pulse Width					
Worst Negative Slack (WNS):	0.533 ns		Worst Hold Slack (WHS):	0.037 ns		Worst Pulse Width Slack (WPWS):	3.750 ns				
Total Negative Slack (TNS):	0.000 ns		Total Hold Slack (THS):	0.000 ns		Total Pulse Width Negative Slack (TPWS):	0.000 ns				
Number of Failing Endpoints:	0		Number of Failing Endpoints:	0		Number of Failing Endpoints:	0				
Total Number of Endpoints:	2781		Total Number of Endpoints:	2781		Total Number of Endpoints:	720				
All user specified timing constraints are met.											

5.2 资源分析

使用 Vivado 2024.2 进行 FPGA 资源分析，结果如图 5 所示。

图 5: FPGA 资源分析

Name	^_1	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Slice (15850)	LUT as Logic (63400)	LUT as Memory (19000)	DSPs (240)	Bonded IOB (210)	BUFGCTRL (32)
TOP		1021	555	53	16	336	863	158	1	30	1
cpu (TOP_CPU)		718	303	49	16	254	560	158	1	0	0
control_unit (CU_TOP)		299	13	17	0	127	299	0	0	0	0
control_CAR (CAR)		299	13	17	0	127	299	0	0	0	0
data_ram (DATA_RAM)		64	0	32	16	16	0	64	0	0	0
external_bus (EXTERNAL_BUS)		0	1	0	0	1	0	0	0	0	0
instruction_rom (INSTR_ROM)		182	135	0	0	78	88	94	0	0	0
instr_fetch_clk_divide (CLK_DIVIDER)		12	12	0	0	4	12	0	0	0	0
instr_load_bram (BRAM_INSTR)		90	8	0	0	28	2	88	0	0	0
instr_load_fifo (FIFO)		30	64	0	0	19	24	6	0	0	0
instr_load_uart (UART)		32	51	0	0	21	32	0	0	0	0
internal_registers (REG_TOP)		173	101	0	0	113	173	0	1	0	0
reg_ACC (ACC)		97	16	0	0	53	97	0	0	0	0
reg_ALU (ALU)		24	37	0	0	42	24	0	1	0	0
reg_IR (IR)		2	16	0	0	7	2	0	0	0	0
reg_MAR (MAR)		9	8	0	0	6	9	0	0	0	0
reg_MBR (MBR)		48	16	0	0	39	48	0	0	0	0
reg_PC (PC)		13	8	0	0	7	13	0	0	0	0
ins_button_check_flags (KEY_JITTER_parameterized0)		27	25	0	0	8	27	0	0	0	0
ins_button_check_instruction (KEY_JITTER_parameterized0_0)		28	25	0	0	10	28	0	0	0	0
ins_button_check_result (KEY_JITTER_parameterized0_1)		27	25	0	0	9	27	0	0	0	0
ins_button_next_instr (KEY_JITTER_parameterized0_2)		27	25	0	0	10	27	0	0	0	0
ins_button_reset (KEY_JITTER_parameterized0_3)		27	25	0	0	8	27	0	0	0	0
ins_switch_start_cpu (KEY_JITTER)		82	24	0	0	27	82	0	0	0	0
ins_switch_step_execution (KEY_JITTER_4)		28	24	0	0	9	28	0	0	0	0
led_display (LED_DISPLAY)		1	9	0	0	3	1	0	0	0	0
segment_display (SEVEN_SEGMENT_DISPLAY)		57	70	4	0	30	57	0	0	0	0

在 FPGA 资源利用方面，核心模块主要集中在 `control_unit` (299 个 LUT 和 13 个寄存器) 和 `internal_registers` (173 个 LUT 和 101 个寄存器) 中，反映出控制通路和数据通路寄存单元的硬件开销占比较高。数据内存 `data_ram` 使用了 64 个 LUT，指令内存 `bram_instr` 使用了 88 个 LUT，全部作为存储资源，这是因为内存容量小，无需动用更稀缺的 BRAM 资源。显示和控制模块如 `segment_display` 和按键消抖模块占用资源较少，基本在 30 个 LUT 和 25 个寄存器左右，属于轻量级设计。此外，设计仅使用了 1 个 DSP 资源和 30 个 IO 引脚，未出现资源瓶颈，说明在保证功能完整的同时保持了良好的资源控制，具有进一步扩展和优化的潜力。

5.3 CPU 功能仿真

5.3.1 简单加法器

仿真内容：计算 $1 + 2 + \dots + 99 + 100 = 5050$ 。

激励设置：用户首先编写源程序如下：

```
LOAD IMMEDIATE 0
STORE IMMEDIATE 1
LOAD IMMEDIATE 1
STORE IMMEDIATE 2
```

```

5  LOOP: LOAD 1
6  ADD 2
7  STORE IMMEDIATE 1
8  LOAD 2
9  ADD IMMEDIATE 1
10 STORE IMMEDIATE 2
11 LOAD IMMEDIATE 101 ; faster
12 SUB 2
13 JGZ IMMEDIATE LOOP
14 LOAD 1
15 HALT
16 ; Expected 5050

```

通过 python 脚本将其转换为 UART 接收模块可以识别的二进制流，具体转换过程见附录 A.1。采用 115200 波特率输入指令内存，当指令内存发出“传输完成”信号后，打开 CPU 的单步调试功能。每间隔 0.1ms，向 CPU 发出一次“下一条指令”信号，直到 CPU 发出“停止”信号。

这个测试样例可以测试 CPU 的条件跳转指令功能、加减法功能和 ZF 功能的正确性。

仿真结果： 取 100 轮循环中的最后两个周期，观察 ALU 运算结果的变化如图 6 所示。在第一轮循环中，被加数为累计的和，其值为 4851，加数为 99。程序首先执行 $99 + 4851 = 4950$ 。接着程序将结果存入内存地址 1（累计和存放于此），然后程序将加数自增 1 后，加载立即数 101 并减去此时的加数 100，得到结果大于 0，ZF、NF 均为 0，JGZ 跳转到循环开始。

第二轮循环，程序累加结果为 $4950 + 100 = 5050$ ，并存入内存地址 1。此时，自增后的加数也为 101，减法运算结果为 0，ZF 置 1，JGZ 跳转条件不满足，循环终止，程序运行到 HALT 后自动终止。

通过 LOAD 指令查看内存地址 1 的值，可知运算结果为 5050，结果正确。

5.3.2 乘法与溢出验证

仿真内容： 计算 $255 \times 254 \times 255$ 。

在 16 位有符号数乘法中，由于 $32768 < 255 \times 254 < 65536$ ，会导致符号位溢出，但不会使用高 16 位，计算结果应为 $255 \times 254 - 2^{16} = -766$ 。第二次乘 255 会导致使用高 16 位存储，计算结果应为 $-766 \times 255 = -195330$ ，映射回 32 位有符号数，则为 FFFD04FE。

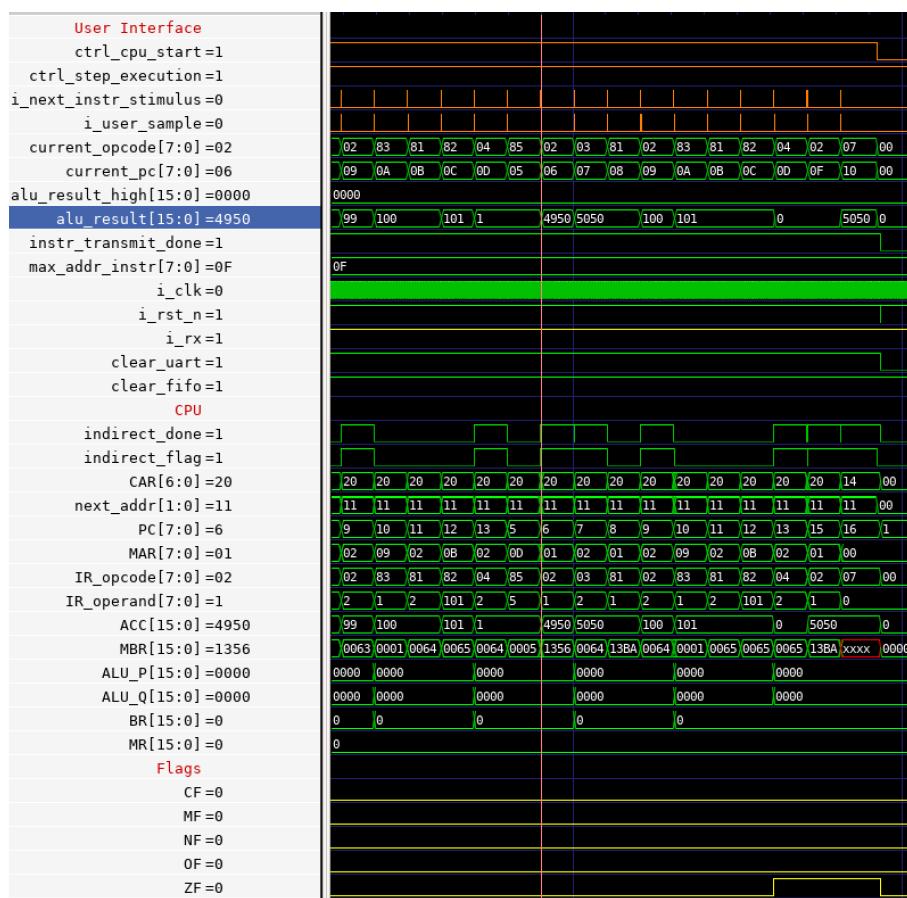
激励设置： 用户首先编写源程序如下：

```

1  LOAD IMMEDIATE 255
2  MPY IMMEDIATE 254
3  STORE IMMEDIATE 1
4  LOAD IMMEDIATE 255
5  MPY 1
6  STORE IMMEDIATE 3
7  LOAD 3
8  LOAD 4
9  HALT

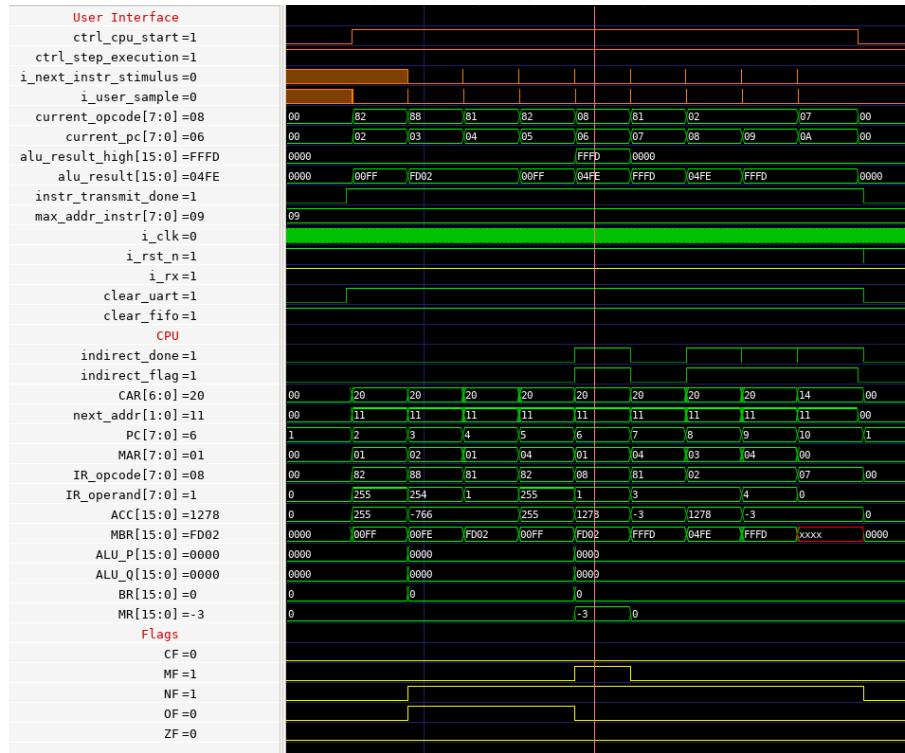
```

图 6: 简单加法器最后两次循环仿真结果



指令输入配置与激励设置和上测试例相同。这个测试样例可以测试 CPU 在乘法运算下的结果存储与置标志位行为。CPU 需要合理判断引入高 16 位前后 OF 标志位的判决规则改变，以及在存在高 16 位时调用 STOREH 指令存储高位。

图 7: 乘法与溢出测试例仿真结果



仿真结果： 如图 7 所示，乘法运算的低位、高位结果正确地存储在内存地址 3 和 4，表明 STOREH 成功触发并正确执行。使用 LOAD 指令查看内存地址 3、4 的值，可知：

- 第一次乘法运算结果高 16 位为 0，低 16 位为 FD02，OF、NF 置 1，表明乘法溢出且结果为负；
- 第二次乘法运算结果高 16 位为 FFFD，低 16 位为 04FE。由于乘数符号不同，乘法结果为负数，NF 置 1；未发生 32 位溢出，故 OF 为 0。

5.3.3 移位运算验证

仿真内容：

- 算术左右移测试： $1 <<< 15 = 32768$, $32768 >> 10 = 32$ ，存入内存地址 1；
- 移出标志位测试： $32 << 11 = 65536$ （溢出），存入内存地址 2；

激励设置： 用户首先编写源程序如下：

```

LOAD IMMEDIATE 1
SHIFTL IMMEDIATE 15
SHIFTR IMMEDIATE 10
STORE IMMEDIATE 1
SHIFTL IMMEDIATE 11
STORE IMMEDIATE 2
LOAD 1

```

```

LOAD 2
HALT

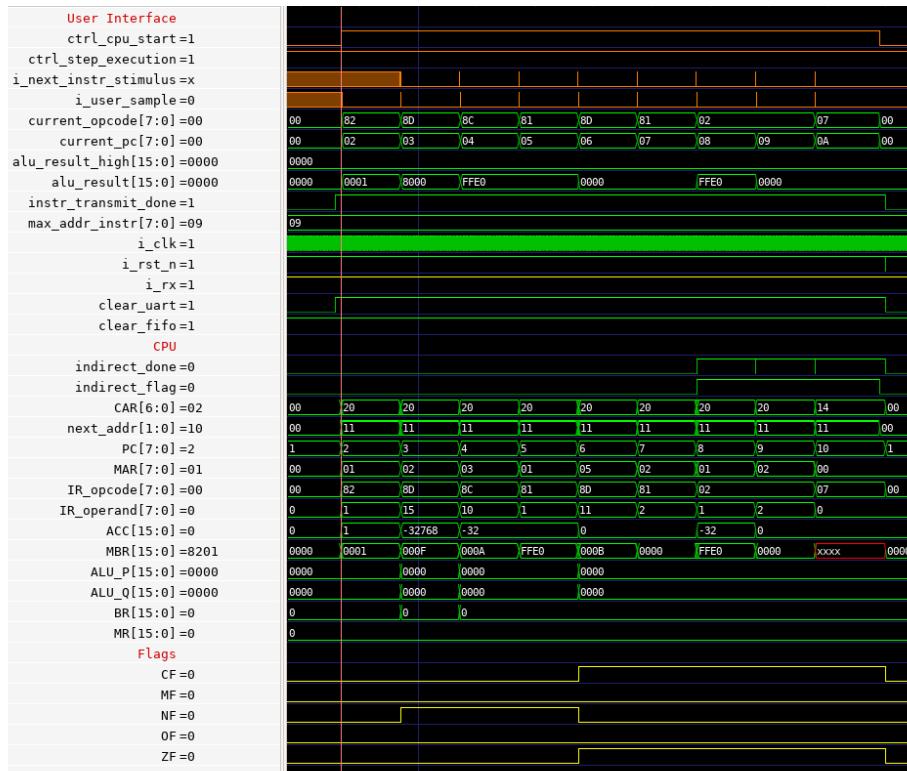
```

指令输入配置与激励设置和上测试例相同。这个测试样例可以测试 CPU 的算术左移 (SHIFTL) 和算术右移 (SHIFTR) 指令的正确性。

仿真结果： 如图 8 所示，移位运算的结果正确地存储在内存地址 1 和 2，表明移位指令执行正确。使用 LOAD 指令查看内存地址 1、2 的值，可知：

- 算术左移 15 位后，结果为 8000 (-32768)，NF 置 1，表明最高位正确移入符号位；
- 算术右移 10 位后，结果为 FFE0 (-32)，表明符号位成功填充到移入的空位；
- 再算术左移 11 位后，结果为 0000 (65536，最高位移出)，CF 置 1，表明移出标志位正确赋值。

图 8：移位运算验证仿真结果



5.3.4 逻辑运算与无条件跳转验证

仿真内容： 依次完成以下操作：

- $99 - 100 = -1(0xFFFF)$;
- $0xFFFF \& 0x000C = 0x000C$;
- $0x000C | 0x0003 = 0x000F$;
- $\neg 0x000F = 0xFFFF$;
- 跳过第 10 条指令，直接跳转到 HALT 指令，停止执行。

激励设置： 用户首先编写源程序如下：

```

LOAD IMMEDIATE 100
STORE IMMEDIATE 1

```

```

LOAD IMMEDIATE 99
SUB 1
AND IMMEDIATE 12 ; 1100
OR IMMEDIATE 3   ; 0011
STORE IMMEDIATE 2
NOT 2
JMP IMMEDIATE 11
ADD IMMEDIATE 1
HALT

```

指令输入配置与激励设置和上测试例相同。这个测试样例可以测试 CPU 的逻辑运算指令（AND、OR、NOT）和无条件跳转指令的正确性。

仿真结果：如图 9 所示，正确的逻辑运算的结果依次显示在用户面接口中，且 ADD 指令并未执行，可知无条件跳转指令和逻辑指令运行正确。

图 9：逻辑运算与无条件跳转验证仿真结果



5.4 FPGA 实物验证

5.4.1 开发环境与测试准备

基于 Nexys 4 DDR 开发板进行 FPGA 实物验证。该开发板使用 FT2232C 芯片接收来自 UART 引脚的串口数据，并通过同一引脚进行比特流烧录。由于本设计的 CPU 是基于 UART 接收模块接收指令，因此在测试时仅需要使用一根 USB-UART 转接线连接开发板和 PC 即可同时完成烧录和指令输入。

以下是实物测试所用的开发工具和开发环境。

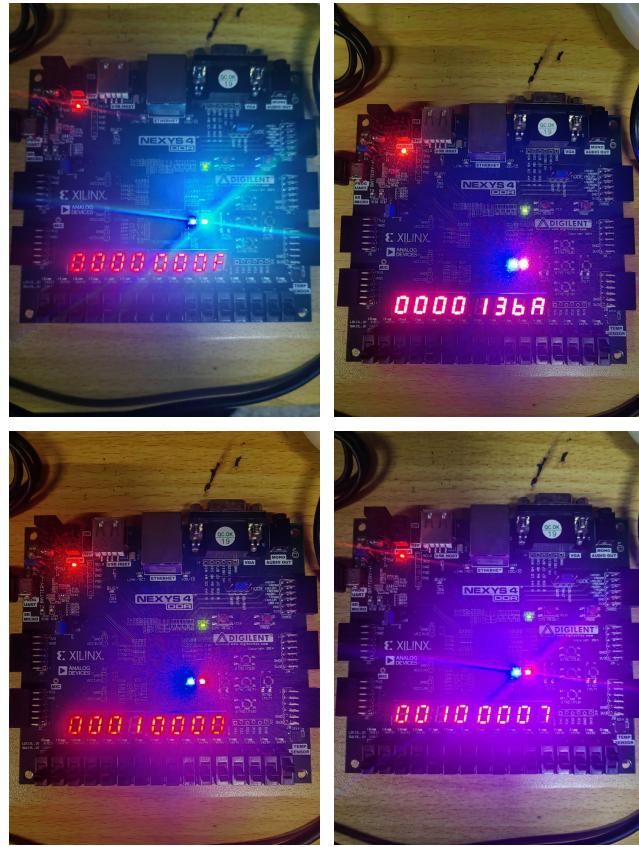
- 开发工具：Vivado 2024.2、Python 3.12.3、PySerial 3.5
- 测试环境：Ubuntu 24.04

开始测试前，首先将 SD/USB 跳线帽调整至 USB，用于接收串口数据，然后将烧录跳线帽调整至 JTAG，用于烧录比特流。完成配置后从 Vivado 烧录比特流，烧录完成后可看到左侧 LED 亮蓝灯、右侧 LED 亮绿色，七段数码管显示全零。这是系统的默认配置，LED 灯分别代表：自动执行指令、允许接收指令。

5.4.2 简单加法器实物验证

在这一轮测试中，CPU 采用自动模式执行指令，左侧 LED 亮起蓝灯，如图 10 所示。左上图显示：加载指令完成后指令内存的最大地址为 15，即共存入 15 条指令。将最右侧开关打开后，CPU 开始自动执行指令直到停止。CPU 停止运行后，右侧 LED 亮起红灯，按下左侧按钮，七段数码管显示 CPU 运算结果值为 0x13BA（5050），表明 CPU 成功执行了加法器测试程序。再按下右侧按钮，七段数码管显示 ZF 为 1，代表最后一轮减法的值为 0，ZF 置 1，JGZ 跳转条件不满足，CPU 停止执行。最后按下右侧按钮，从左至右第三、四个数码管显示下一条指令的 PC 值和当前指令的 Opcode，分别为 0x10 和 0x07，表明 CPU 成功停止在指令地址为 15 的 HALT 指令。

图 10: 简单加法器测试



5.4.3 乘法与溢出实物验证

在这一轮测试中，CPU 采用单步调试模式执行指令，左侧 LED 亮起红灯。此轮仿真共计 9 条指令，全流程如图 11 所示。加载指令完成后指令内存的最大地址为 9，即共存入 9 条指令。将最右侧开关打开后，CPU 开始单步调试执行指令。

执行完第一条乘法指令后，按下左侧按钮显示结果为 0xFD02，按下下侧按钮可见 NF 和 OF 均为 1，表明乘法结果为负且发生溢出。按下中间按钮执行第二次乘法后，查看结果显示为 0xFFFFD04FE，标志位显示 MF 和 NF 均为 1，这与仿真结果一致。继续按下中间按钮存储乘法结果，观察到 MF 标志位清零，证明乘法高位已被存入内存且 MR 寄存器已重置。执行 LOAD 3 指令后，七段数码管成功显示乘法结果低位 0x04FE；执行 LOAD 4 指令后，显示高位 0xFFFFD，验证了 CPU 正确运算存储 32 位乘法结果、处理溢出的能力。

5.4.4 移位运算实物验证

在这一轮测试中，CPU 采用单步调试模式执行指令，左侧 LED 亮起红灯，全流程如图 12 所示。第一条指令执行完成后，查看运算结果可知为 1。先对其左移 15 位，依次按下中间、左侧按钮查看结果，显示为 0x8000，NF 置 1，表明最高位正确移入符号位。接着按下中间按钮执行右移 10 位，查看运算结果可知为 0xFFE0，NF 置 1，表明符号位成功填充到移入的空位。最后按下中间按钮执行左移 11 位，查看运算结果可知为 0x0000（65536），CF、ZF 置 1，表明移出标志位正确赋值。

图 11: 乘法与溢出验证

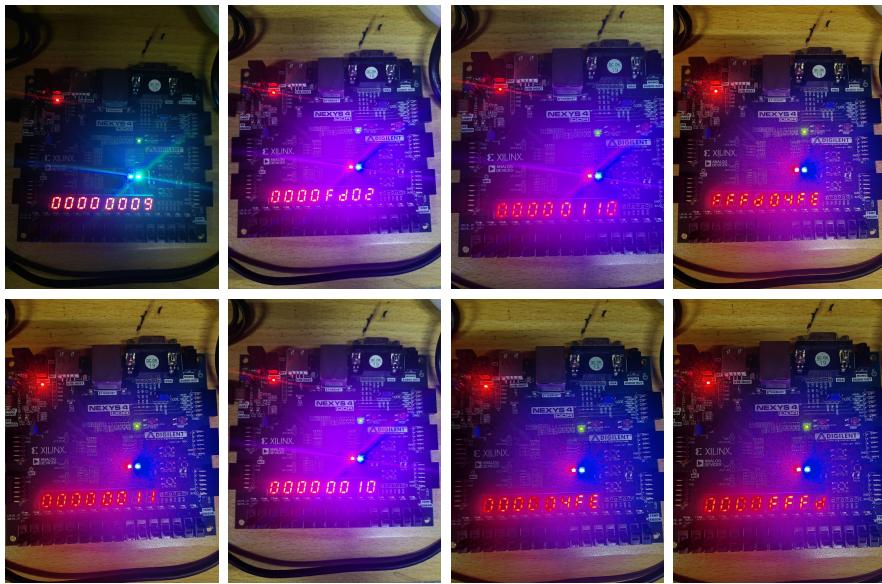
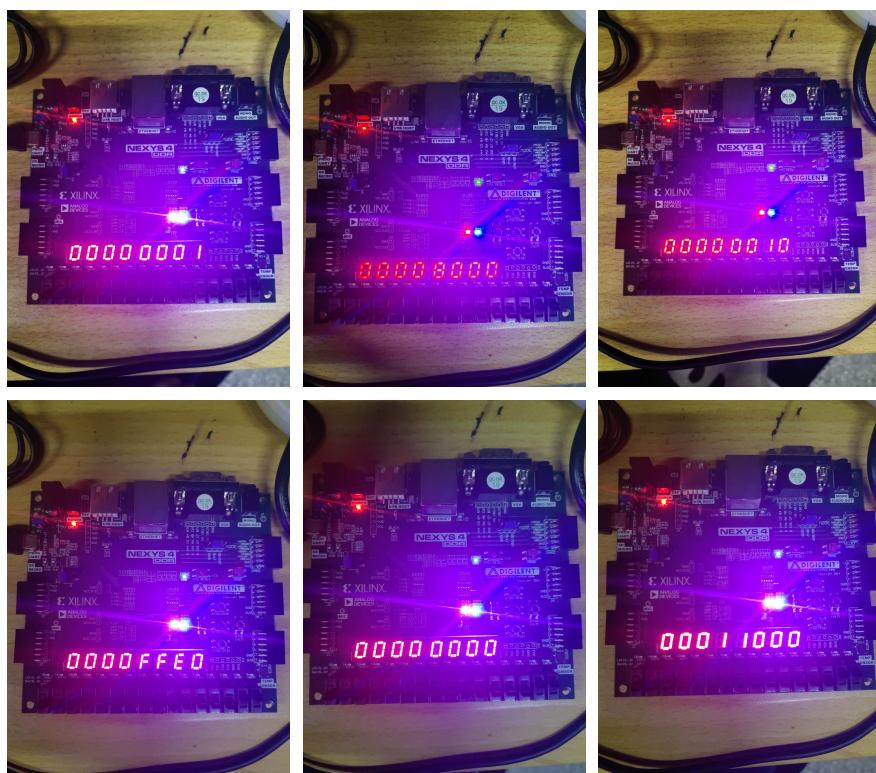


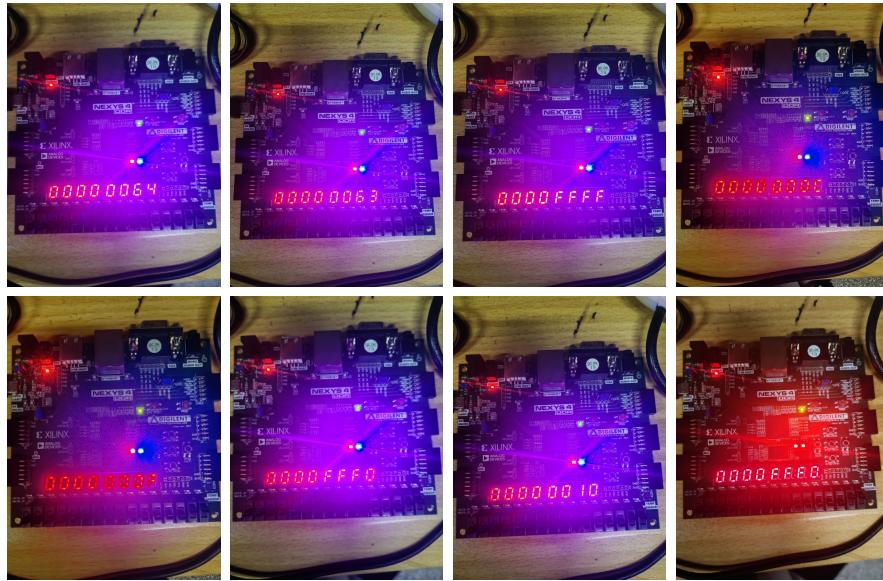
图 12: 移位运算验证



5.4.5 逻辑运算与无条件跳转实物验证

在这一轮测试中，CPU 采用单步调试模式执行指令，左侧 LED 亮起红灯，全流程如图 13 所示。第一条指令执行完成后，查看运算结果可知为 0xFFFF，按下左侧按钮查看 ZF 和 NF 均为 1，表明减法结果为负且发生溢出。接着按下中间按钮执行逻辑与、或、非操作，依次查看运算结果可知为 0x000C、0x000F、0xFFFF0，NF 置 1，表明逻辑运算结果正确。最后按下中间按钮执行无条件跳转指令，CPU 成功跳过 ADD 指令，停止在 HALT 指令（由于 CPU 停止时运算结果仍为 0xFFFF0）。

图 13：逻辑运算与无条件跳转验证



6 总结与展望

6.1 设计总结

本设计实现了一个基于 FPGA 的 16 位单周期 CPU，支持基本的算术、逻辑、移位和跳转指令，并通过功能仿真、时序分析和 FPGA 实物验证，验证了设计的正确性和稳定性，满足了课程的设计要求。在设计要求之上，本设计还完成了一个简单的汇编代码编译器，并通过 UART/FIFO 串口通信实时写入用户编写的程序，无需在每次更改程序时重新构建、烧录，极大地提升了调试的灵活性。

6.2 已知问题

本设计还存在一些问题，主要问题包括：

- 不适合固化处理，现有设计在烧录后会直接点亮数码管和 LED 灯，长期点亮会造成潜在的功耗和发热问题。后续可在设计中加入开关或按键，允许用户选择是否点亮数码管和 LED 灯。
- CPU 没有异常处理机制，若发生异常（如在汇编代码无 HALT 指令，或未传入汇编代码时），CPU 将无法正常工作。此时需要用户复位后重新传入修改过的指令。
- CPU 的指令集不够丰富，缺乏一些常用的指令，如除法、内存地址交换等指令。
- 现有设计下，CPU 只能单周期运行，无法实现多周期或流水线设计。由于本设计的 CPU 是基于单周期设计的，因此在指令执行时，CPU 只能完成一条指令的执行，无法同时执行多条指令。

参考文献

- [1] 菜鸟教程. Verilog FIFO 设计[EB/OL]. 2020. <https://www.runoob.com/w3cnote/verilog2-fifo.html>.
- [2] 赖兆磬. 基于 FPGA 流水线 CPU 的设计与实现[D]. 桂林电子科技大学, 2008.

A 完整设计代码

该部分以数据流向和 CPU 从内向外的顺序，给出设计的完整代码。顶层模块放在每节的最后，展示了各个模块的连接方式。测试时使用的汇编代码由于和本设计关联不大，且在正文中有所展现，故在附录部分不再单独列出。另外，该项目代码已开源于[Github](#)，欢迎提交项目相关的 issues 或 PR。

A.1 汇编程序处理 Python 脚本

```
1 import re
2 import serial
3 import os
4 import colorama
5 # mnemonics
6 MEMONICS = {
7     "STORE": 0x01,
8     "LOAD": 0x02,
9     "ADD": 0x03,
10    "SUB": 0x04,
11    "JGZ": 0x05,
12    "JMP": 0x06,
13    "HALT": 0x07,
14    "MPY": 0x08,
15    "AND": 0x09,
16    "OR": 0x0A,
17    "NOT": 0x0B,
18    "SHIFTR": 0x0C,
19    "SHIFTL": 0x0D
20 }
21
22 def parse_assembly(lines):
23     machine_code = []
24     labels = {}
25     pending = []
26
27     # First pass: find labels
28     addr = 1
29     for line in lines:
30         line = line.split(';')[0].strip() # clear comments
31         if not line:
32             continue
33         if ':' in line:
34             label, rest = map(str.strip, line.split(':', 1))
35             labels[label] = addr
36             if rest:
37                 addr += 1
38         else:
39             addr += 1
```

```

40
41     # Second pass: generate code
42
43     addr = 1
44
45     for line in lines:
46
47         line = line.split(';')[0].strip()
48
49         if not line:
50
51             continue
52
53         if ':' in line:
54
55             parts = line.split(':', 1)
56
57             line = parts[1].strip()
58
59             if not line:
60
61                 continue
62
63
64             tokens = line.split()
65
66             if not tokens:
67
68                 continue
69
70
71             instr = tokens[0]
72             immediate = False
73
74
75             if instr in ["HALT"] : # No operand, fill with 0
76
77                 opcode = MEMONICS[instr]
78
79                 operand = 0x00
80
81             else:
82
83                 if len(tokens) < 2:
84
85                     raise ValueError(f"Missing operand in line: {line}")
86
87
88                 if tokens[1] == "IMMEDIATE":
89
90                     immediate = True
91
92                     operand_str = tokens[2]
93
94                     opcode = MEMONICS[instr] | 0x80 # MSB = 1
95
96                 else:
97
98                     operand_str = tokens[1]
99
100                    opcode = MEMONICS[instr] # MSB = 0
101
102
103                 if operand_str.isdigit():
104
105                     operand = int(operand_str)
106
107                 elif operand_str in labels:
108
109                     operand = labels[operand_str]
110
111                 else:
112
113                     try:
114
115                         operand = int(operand_str, 0) # Support 0x form operand
116
117                     except:
118
119                         raise ValueError(f"Unknown operand: {operand_str}")
120
121
122                 if operand < 0 or operand > 255:
123
124                     raise ValueError(f"Operand out of 8-bit range: {operand}")

```

```

87
88     machine_code.append((opcode << 8) | operand)
89     addr += 1
90
91     return machine_code
92
93
94 def assemble_to_bytes(code: list[int]) -> bytearray:
95     result = bytearray()
96     for word in code:
97         result.append((word >> 8) & 0xFF) # opcode
98         result.append(word & 0xFF)      # operand
99     return result
100
101 def send_to_serial(bitstream:bytearray) -> None:
102     # must run on Linux system
103     # FPGA Config: Baud rate = 115200, 8N1 Transmission
104     write_port = '/dev/ttyUSB1'
105     ser = serial.Serial(
106         port= write_port,
107         baudrate= 115200,
108         timeout=1,
109         bytesize=8,
110         parity= "N",
111         stopbits=1
112     )
113
114     # 向 FPGA 发送数据
115     number_of_bytes = ser.write(bitstream) # 将字符串转换为字节并发送
116     # print(f"Write bit {bitstream} to serial port {write_port}\n")
117     print(f"{number_of_bytes} bytes of data Write successfully")
118     # # 读取来自 FPGA 的数据
119     # response = ser.readline() # 读取一行数据（假设 FPGA 发送数据是以换行符结尾）
120     # print(f"Received from FPGA: {response.decode().strip()}")
121
122     # 关闭串口
123     ser.close()
124
125 def main(filename:str):
126     os.chdir("./designs/input_src")
127     with open(f'{filename}.txt', 'r') as file:
128         lines = file.readlines()
129
130     machine_words = parse_assembly(lines)
131     binary = assemble_to_bytes(machine_words)
132
133     # For Reference:

```

```

134     print("033[1;34mMachine Code:\033[0m")
135     for i, word in enumerate(machine_words):
136         print(f"{i+1:02}: {word:04X}")
137
138     # For Simulation Testbench input drive:
139     print("\n\033[1;34mGenerated UART Send Bitstream:\033[0m")
140     for b in binary:
141         bits = f"{b:08b}"
142         reversed_bits = bits[::-1]
143         print(f"uart_send_byte(8'b{reversed_bits});")
144     # print(binary)
145     # For FPGA Verification:
146
147     # Bit reverse each byte in the binary array
148     bit_reversed_binary = bytearray()
149     for b in binary:
150         reversed_byte = 0
151         for i in range(8):
152             # Extract bit at position i and place it at position (7-i)
153             if b & (1 << i):
154                 reversed_byte |= (1 << (7-i))
155             bit_reversed_binary.append(reversed_byte)
156
157     # Use the bit-reversed binary for sending to serial
158     binary = bit_reversed_binary
159     send_to_serial(binary)
160
161
162 if __name__ == "__main__":
163     colorama.init()
164     filename = input(f"\033[1;32mPlease enter the assembly file name (without .txt): \033[0m")
165     main(filename)

```

Listing 1: write_bistream.py

A.2 UART 接收与指令内存模块

该部分包含了 UART 接收模块、FIFO 模块和指令内存模块的设计代码。

```

`timescale 1ns / 1ps
// 5.5 Update: adapt 8N1 format
// 5.6 Update: always module driven by i_clk rather than i_clk_uart
module UART (
    i_clk,
    i_clk_uart,
    i_rst_n,
    i_rx,
    o_data,

```

```

10         o_valid,
11         o_clear_sign
12     );
13     input wire      i_clk;
14     input wire      i_clk_uart;
15     input wire      i_rst_n;
16     input wire      i_rx;
17     output reg [7:0] o_data;
18     output reg      o_valid;      // on data is translated
19     output wire     o_clear_sign; // on no more data is received
20
21 // 0.5s no new data
22 parameter MAX_WAITING_CLK = 434;
23
24
25 // parameter IDLE = 3'b000;
26 parameter START = 2'b00;
27 parameter DATA = 2'b01;
28 parameter STOP = 2'b10;
29
30 reg [1:0] current_state, next_state;
31
32 reg [2:0] bit_counter;          // At most 8bit data
33 reg [25:0] rx_no_data_counter; // time-out counter
34 reg [7:0] rx_shift_reg;        // data storage
35
36 reg clear;
37 reg i_clk_uart_dly;
38 wire i_clk_uart_rising = (i_clk_uart && !i_clk_uart_dly);
39 wire i_clk_uart_falling = (!i_clk_uart && i_clk_uart_dly);
40
41 always @ (posedge i_clk or negedge i_rst_n) begin
42     if (!i_rst_n)
43         i_clk_uart_dly <= 0;
44     else
45         i_clk_uart_dly <= i_clk_uart;
46 end
47
48 // data storage update
49 always @ (posedge i_clk or negedge i_rst_n) begin
50     if (!i_rst_n) begin
51         current_state <= START;
52     end
53     else begin
54         if(i_clk_uart_falling) begin
55             current_state <= next_state;
56         end

```

```

57     end
58 end
59
60 // State Shift
61 always @(*) begin
62     case (current_state)
63         START:
64             next_state = (i_rx == 1'b0) ? DATA : START;
65         DATA:
66             next_state = (bit_counter == 7) ? STOP : DATA;
67         STOP:
68             next_state = START;
69         default:
70             next_state = START;
71     endcase
72 end
73
74 // 数据接收与控制逻辑
75 always @ (posedge i_clk or negedge i_rst_n) begin
76     if (!i_rst_n) begin
77         bit_counter <= 0;
78         rx_shift_reg <= 8'd0;
79         o_valid      <= 0;
80         o_data       <= 8'd0;
81     end
82     else begin
83         if(i_clk_uart_falling) begin
84             case (current_state)
85                 START: begin
86                     bit_counter <= 0;
87                     o_valid <= 0;
88                     rx_shift_reg <= 0;
89                 end
90                 DATA: begin
91                     // LSB first
92                     rx_shift_reg <= {rx_shift_reg[6:0], i_rx};
93                     bit_counter <= bit_counter + 1;
94                 end
95                 STOP: begin
96                     o_data <= rx_shift_reg;
97                     o_valid <= 1'b1;
98
99                 end
100                default: begin
101                    o_valid <= 0;
102                end
103            endcase

```

```

104     end
105   end
106 end
107
108 // Time-out detect
109 always @(posedge i_clk or negedge i_rst_n) begin
110   if (!i_rst_n) begin
111     clear <= 0;
112     rx_no_data_counter <= 0;
113   end
114   else begin
115     if(i_clk_uart_falling) begin
116       case (current_state)
117         START: begin
118           if (rx_no_data_counter == MAX_WAITING_CLK) begin
119             rx_no_data_counter <= 0;
120             clear <= 1;
121           end
122           else begin
123             rx_no_data_counter <= rx_no_data_counter + 1;
124           end
125         end
126         default: begin
127           clear <= 0;
128           rx_no_data_counter <= 0;
129         end
130       endcase
131     end
132   end
133 end
134
135 assign o_clear_sign = clear;
136
137 endmodule

```

Listing 2: uart.v

```

1 // Date: 25.4.13
2 // Author: LiPtP
3 `timescale 1ns / 1ps
4 module FIFO (
5   i_rst_n,
6   i_clk_wr,
7   i_valid_uart,
8   i_data_uart,
9   i_clk_rd,
10  o_data_bram,
11  o_addr_bram,

```

```

12     o_wr_en_bram,
13     o_fifo_empty
14 );
15 input i_rst_n;
16
17 // UART (100MHz)
18 input i_clk_wr;
19 input i_valid_uart;
20 input [7:0] i_data_uart;
21
22 // CPU (50MHz)
23 input i_clk_rd;
24 output reg [15:0] o_data_bram;
25 output reg [7:0] o_addr_bram;
26 output reg o_wr_en_bram;
27
28 // for judging completion
29 output o_fifo_empty;
30
31 localparam DEPTH = 16; // FIFO depth, for storing full commands
32 localparam ADDR_WIDTH = 4; // Address for FIFO
33
34 reg [7:0] fifo_mem[0:DEPTH-1];
35
36 reg [ADDR_WIDTH:0] wr_ptr_bin, rd_ptr_bin;
37 reg [ADDR_WIDTH:0] wr_ptr_gray, rd_ptr_gray;
38 reg [ADDR_WIDTH:0] wr_ptr_gray_sync1, wr_ptr_gray_sync2;
39 reg [ADDR_WIDTH:0] rd_ptr_gray_sync1, rd_ptr_gray_sync2;
40
41 // Read is faster than Write, so we use newer wr_sync pointer
42 wire fifo_empty = (wr_ptr_gray_sync2 == rd_ptr_gray);
43 wire fifo_full = ((rd_ptr_gray[ADDR_WIDTH] != wr_ptr_gray_sync2[ADDR_WIDTH]) &&
44                   (rd_ptr_gray[ADDR_WIDTH-1:0] == wr_ptr_gray_sync2[ADDR_WIDTH-1:0]));
45
46 reg clk_wr_d;
47 wire clk_wr_rising = (clk_wr_d && !i_clk_wr);
48 always @ (posedge i_clk_rd or negedge i_rst_n) begin
49   if (!i_rst_n) begin
50     clk_wr_d <= 0;
51   end
52   else begin
53     clk_wr_d <= i_clk_wr;
54   end
55 end
56
57 // i_clk_rd is the system clock
58 always @ (posedge i_clk_rd) begin
59   if (clk_wr_rising) begin

```

```

59         if (i_valid_uart && !fifo_full)
60             fifo_mem[wr_ptr_bin[ADDR_WIDTH-1:0]] <= i_data_uart;
61     end
62 end
63
64 // -----
65 // Write Time Zone (UART, 1.8432MHz)
66 // -----
67
68 always @ (posedge i_clk_rd or negedge i_rst_n) begin
69     if (!i_rst_n) begin
70         wr_ptr_bin <= 0;
71         wr_ptr_gray <= 0;
72     end
73     else begin
74         if(clk_wr_rising) begin
75             if (i_valid_uart && !fifo_full) begin
76                 wr_ptr_bin <= wr_ptr_bin + 1;
77                 wr_ptr_gray <= (wr_ptr_bin + 1) ^ ((wr_ptr_bin + 1) >> 1);
78             end
79         end
80     end
81 end
82
83
84 // 同步读指针 (Gray) 到写时钟域
85 always @ (posedge i_clk_rd or negedge i_rst_n) begin
86     if (!i_rst_n) begin
87         rd_ptr_gray_sync1 <= 0;
88         rd_ptr_gray_sync2 <= 0;
89     end
90     else begin
91         if(clk_wr_rising) begin
92             rd_ptr_gray_sync1 <= rd_ptr_gray;
93             rd_ptr_gray_sync2 <= rd_ptr_gray_sync1;
94         end
95     end
96 end
97
98 // -----
99 // Read Clock Zone (CPU, 100MHz)
100 // -----
101 reg [7:0] data_buffer;
102 reg      byte_flag; // flag of the first UART byte is read
103
104 always @ (posedge i_clk_rd or negedge i_rst_n) begin
105     if (!i_rst_n) begin

```

```

106     rd_ptr_bin <= 0;
107     rd_ptr_gray <= 0;
108     byte_flag <= 0;
109     o_data_bram <= 0;
110     o_addr_bram <= 0;
111     o_wr_en_bram <= 0;
112     data_buffer <= 0;
113   end
114   else begin
115     o_wr_en_bram <= 0;
116
117     // Read a byte to data_buffer if it's odd or write out
118     if (!fifo_empty) begin
119
120       rd_ptr_bin <= rd_ptr_bin + 1;
121       rd_ptr_gray <= (rd_ptr_bin + 1) ^ ((rd_ptr_bin + 1) >> 1);
122
123       if (!byte_flag) begin
124         data_buffer <= fifo_mem[rd_ptr_bin[ADDR_WIDTH-1:0]];
125         byte_flag <= 1;
126       end
127       else begin
128         o_data_bram <= {data_buffer, fifo_mem[rd_ptr_bin[ADDR_WIDTH-1:0]]}; // opcode +
129                     operand
130         o_addr_bram <= o_addr_bram + 1;
131         o_wr_en_bram <= 1;
132         byte_flag <= 0;
133       end
134     end
135
136     // end else if (byte_flag) begin
137     //   // if there are odd bytes from UART, fill zero
138     //   o_data_bram <= {data_buffer, 8'h00};
139     //   o_addr_bram <= o_addr_bram + 1;
140     //   o_wr_en_bram <= 1;
141     //   byte_flag <= 0;
142     // end
143   end
144
145 // 同步写指针 (Gray) 到读时钟域
146 always @ (posedge i_clk_rd or negedge i_rst_n) begin
147   if (!i_rst_n) begin
148     wr_ptr_gray_sync1 <= 0;
149     wr_ptr_gray_sync2 <= 0;
150   end
151   else begin

```

```

152     wr_ptr_gray_sync1 <= wr_ptr_gray;
153     wr_ptr_gray_sync2 <= wr_ptr_gray_sync1;
154   end
155 end
156
157 assign o_fifo_empty = fifo_empty;
158 endmodule

```

Listing 3: fifo.v

```

// 2025.4.28 Add read enable signal to this module
`timescale 1ns / 1ps

module BRAM_INSTR (
    i_clk,
    en_write,
    en_read,
    i_addr_write,
    i_addr_read,
    o_instr_read,
    i_instr_write,
    o_max_addr
);
input i_clk;
input en_write; // flag of write instructions.
input en_read; // flag of read instructions.
input [7:0] i_addr_write; // address of the upcoming instruction
input [15:0] i_instr_write; // content of the upcoming instruction
input [7:0] i_addr_read; // address of instruction to be read, starts from 1
output [15:0] o_instr_read; // content of instruction to be read
output [7:0] o_max_addr; // current max address of instr BRAM

reg [15:0] mem [0:255];
reg [7:0] current_addr;

always @(posedge i_clk) begin
    if (en_write) begin
        mem[i_addr_write] <= i_instr_write;
        current_addr <= i_addr_write;
    end
end

assign o_instr_read = en_read ? mem[i_addr_read] : 16'b0; // read fully combinational
assign o_max_addr = current_addr;
endmodule

```

Listing 4: bram_instr.v

```

1 module CLK_DIVIDER #((
2     parameter DIVIDE_BY = 2 // 2 or 868
3 )(
4     input wire i_clk,
5     input wire i_rst_n_sync,
6     output reg o_clk_div
7 );
8     reg rst_n_sync_reg;
9     reg [9:0] div_cnt;
10
11
12    always @(posedge i_clk or negedge i_rst_n_sync) begin
13        if (!i_rst_n_sync)
14            rst_n_sync_reg <= 1'b0;
15        else
16            rst_n_sync_reg <= 1'b1;
17    end
18
19
20    always @(posedge i_clk or negedge i_rst_n_sync) begin
21        if (!i_rst_n_sync) begin
22            div_cnt <= 0;
23            o_clk_div <= 1'b1;
24        end else if (!rst_n_sync_reg) begin
25            div_cnt <= 0;
26            o_clk_div <= 1'b1;
27        end else begin
28            if (div_cnt == DIVIDE_BY/2 - 1) begin
29                o_clk_div <= ~o_clk_div;
30                div_cnt <= 0;
31            end else begin
32                div_cnt <= div_cnt + 1'b1;
33            end
34        end
35    end
36
37 endmodule

```

Listing 5: clk_divider.v

```

`timescale 1ns / 1ps

module INSTR_ROM (
    i_clk,
    i_rst_n,
    i_rx,
    en_read,

```

```

8     i_addr_read,
9     o_instr_read,
10    o_instr_transmit_done,
11    o_max_addr
12  );
13
14 input i_clk; // Board Frequency: 100MHz
15
16 input i_rst_n; // Global Reset
17 input i_rx;
18 input en_read; // Read Enable Signal
19 input [7:0] i_addr_read;
20 output [15:0] o_instr_read;
21 output o_instr_transmit_done;
22 output [7:0] o_max_addr;
23
24 // Baud Rate Settings
25 parameter BAUD_RATE = 115200;
26 parameter CLK_FREQ = 100000000;
27
28 localparam CLK_DIV = CLK_FREQ / BAUD_RATE;
29
30 wire valid_uart;
31 wire [7:0] data_uart;
32 wire [15:0] data_bram;
33 wire [7:0] addr_bram;
34 wire enable_write_bram;
35 wire clear_uart;
36 wire clear_fifo;
37
38
39
40 CLK_DIVIDER #(
41     .DIVIDE_BY(CLK_DIV)
42 )
43     instr_fetch_clk_divide
44 (
45     .i_clk(i_clk),
46     .i_rst_n_sync(i_rst_n),
47     .o_clk_div(i_clk_uart)
48 );
49
50 UART instr_load_uart (
51     .i_clk(i_clk),
52     .i_clk_uart(i_clk_uart),
53     .i_rst_n(i_rst_n),
54     .i_rx(i_rx),

```

```

55     .o_data(data_uart),
56     .o_valid(valid_uart),
57     .o_clear_sign(clear_uart)
58 );
59
60 FIFO instr_load_fifo (
61     .i_rst_n(i_rst_n),
62     .i_clk_wr(i_clk_uart),
63     .i_valid_uart(valid_uart),
64     .i_data_uart(data_uart),
65     .i_clk_rd(i_clk),
66     .o_data_bram(data_bram),
67     .o_addr_bram(addr_bram),
68     .o_wr_en_bram(enable_write_bram),
69     .o_fifo_empty(clear_fifo)
70 );
71
72 BRAM_INSTR instr_load_bram (
73     .i_clk(i_clk),
74     .en_write(enable_write_bram),
75     .en_read(en_read),
76     .i_addr_write(addr_bram),
77     .i_addr_read(i_addr_read),
78     .o_instr_read(o_instr_read),
79     .i_instr_write(data_bram),
80     .o_max_addr(o_max_addr)
81 );
82
83 // needs fix
84 assign o_instr_transmit_done = clear_uart & clear_fifo;
85 endmodule

```

Listing 6: top_instr_rom.v

A.3 控制单元设计

该部分包含了控制单元的设计模块，包括控制存储器 CM、控制地址寄存器 CAR、控制缓冲寄存器 CBR 和顶层模块。

```

1 /*
2 * 1 global halt
3 * 1 MAR self increment
4 * 2 CAR
5 * 1 ALU_enable
6 * 3 ALU
7 * 16 internal bus
8 * C2: Control for PC+1
9 */

```

```

10 * Critical path: STOREH with indirect for 10 clock cycles
11 */
12 `timescale 1ns / 1ps
13
14 module CONTROL_MEMORY (
15     car,
16     control_word
17 );
18 input wire [6:0] car;           // From CAR
19 output reg [23:0] control_word; // Output Ctrl Signal
20
21 always @(*) begin
22     case (car)
23         // Instruction
24         7'h00:
25             control_word = 24'b00_10_0000_00000000_00000100; // IF1, 2 PC+1
26         7'h01:
27             control_word = 24'b00_10_0000_00000000_00100001; // IF2, 0 5
28         7'h02:
29             control_word = 24'b00_10_0000_00000000_00010000; // ID1, 4
30         7'h03:
31             control_word = 24'b00_10_0000_01000000_00000000; // ID2, 14
32
33         // Operand
34         7'h04:
35             control_word = 24'b00_01_0000_10000000_00000000; // F0, 15
36         7'h05:
37             control_word = 24'b00_10_0000_00000001_00000000; // IND1, 8
38         7'h06:
39             control_word = 24'b00_01_0000_00000000_00100001; // IND2, 0 5
40
41         // STORE
42         7'h07:
43             control_word = 24'b00_10_0000_00010001_00000000; // EX, 8 12
44         7'h08:
45             control_word = 24'b00_11_0000_00100000_00000001; // WB, 0 13
46
47         // LOAD
48         7'h09:
49             control_word = 24'b00_10_0000_00000000_00000000; // EX
50         7'h0A:
51             control_word = 24'b00_11_0000_00001000_00000000; // WB, 11
52
53         // ADD
54         7'h0B:
55             control_word = 24'b00_10_1000_00000000_11000000; // EX, 6 7
56         7'h0C:

```

```

57     control_word = 24'b00_11_0000_00000010_00000000; // WB, 9
58
59 // SUB
60 7'h0D:
61     control_word = 24'b00_10_1001_00000000_11000000; // EX, 6 7
62 7'h0E:
63     control_word = 24'b00_11_0001_00000010_00000000; // WB, 9
64
65 // MPY
66 7'h0F:
67     control_word = 24'b00_10_1010_00000000_11000000; // EX, 6 7
68 7'h10:
69     control_word = 24'b00_11_0010_00000010_00000000; // WB, 9
70
71 // JGZ & JMP
72 7'h11:
73     control_word = 24'b00_10_0000_00000000_00000000; // EX
74 7'h12:
75     control_word = 24'b00_11_0000_00000000_00001000; // WB, 3
76
77 // HALT
78 // Stop and reset control word to IF
79 7'h13:
80     control_word = 24'b00_10_0000_00000000_00000000; // EX
81 7'h14:
82     control_word = 24'b10_11_0000_00000000_00000000; // WB, HALT
83
84 // AND
85 7'h15:
86     control_word = 24'b00_10_1011_00000000_11000000; // EX, 6 7
87 7'h16:
88     control_word = 24'b00_11_0011_00000010_00000000; // WB, 9
89
90 // OR
91 7'h17:
92     control_word = 24'b00_10_1100_00000000_11000000; // EX, 6 7
93 7'h18:
94     control_word = 24'b00_11_0100_00000010_00000000; // WB, 9
95
96 // NOT
97 7'h19:
98     control_word = 24'b00_10_1101_00000000_01000000; // EX, 6
99 7'h1A:
100    control_word = 24'b00_11_0101_00000010_00000000; // WB, 9
101
102 // SHIFTR
103 7'h1B:

```

```

10      control_word = 24'b00_10_1110_00000000_11000000; // EX, 6 7
11'`h1C:
12      control_word = 24'b00_11_0110_00000010_00000000; // WB, 9
13
14      // SHIFTL
15'`h1D:
16      control_word = 24'b00_10_1111_00000000_11000000; // EX, 6 7
17'`h1E:
18      control_word = 24'b00_11_0111_00000010_00000000; // WB, 9
19
20      // Implicit Instructions
21
22      // NOP
23      // Used for completing the instruction cycle.
24      // Executed if JGZ is judged false.
25
26'`h1F:
27      control_word = 24'b00_10_0000_00000000_00000000; // EX
28'`h20:
29      control_word = 24'b00_11_0000_00000000_00000000; // WB
30
31      // STOREH (fixed on 25/5/3)
32      // Used for storage of high bytes of multiply results.
33      // Executed after STORE Operation on MF = 1.
34
35'`h21:
36      control_word = 24'b00_10_0000_00010001_00000000; // EX1, 8 12
37'`h22:
38      control_word = 24'b01_10_0000_00100100_00000001; // WB1, 0 10 13 MAR+1
39'`h23:
40      control_word = 24'b00_10_0000_00010000_00000000; // EX2, 12
41'`h24:
42      control_word = 24'b00_11_0000_00100000_00000001; // WB2, 0 13
43
44      default:
45          control_word = 24'b00_11_0000_00000000_00000000; // Back to zero addr
46      endcase
47  end
48
49 endmodule

```

Listing 7: cu_control_memory.v

```

`timescale 1ns / 1ps

// Sequencing Logic & CAR
/* Sequencing Logic of CAR
 * 10 self increment

```

```

6   * 11 back to 0
7   * 01 jump
8   * 00 nothing
9   */
10
11 module CAR (
12     ctrl_cpu_start,
13     ctrl_step_execution,
14     i_ctrl_halt,
15     i_next_instr_stimulus,
16     i_clk,
17     i_rst_n,
18     i_control_word_car,
19     i_ir_data,
20     i_ctrl_ZF,
21     i_ctrl_NF,
22     i_ctrl_MF,
23     o_car_data
24 );
25 input wire ctrl_cpu_start;
26 input wire ctrl_step_execution;
27 input wire i_clk;
28 input wire i_rst_n;
29 input wire i_next_instr_stimulus;
30 input wire [1:0] i_control_word_car;
31 input wire [4:0] i_ir_data; // MSB + IR[3:0]
32 input wire i_ctrl_ZF; // ZF Flag
33 input wire i_ctrl_NF; // NF Flag
34 input wire i_ctrl_MF; // MF Flag
35 input wire i_ctrl_halt; // C23
36 output wire [6:0] o_car_data;
37
38 reg ctrl_cpu_start_reg;
39
40 always @ (posedge i_clk) begin
41     ctrl_cpu_start_reg <= ctrl_cpu_start;
42 end
43
44 reg [4:0] ir_data;
45 reg [6:0] CAR;
46 reg indirect_done;
47 wire indirect_flag = ctrl_cpu_start ? (!ir_data[4] && (ir_data[3:0] != 4'b0)) : 1'b0;
48
49 always @ (posedge i_clk or negedge i_rst_n) begin
50     if (!i_rst_n) begin
51         ir_data <= 5'b0;
52     end
53     else begin

```

```

53     if (i_ir_data[3:0] != 3'b0) begin
54         ir_data <= i_ir_data[4:0];
55     end
56 end
57
58 // always @(*) begin
59 //     if(i_ir_data[3:0] != 3'b0) begin
60 //         ir_data = i_ir_data[4:0];
61 //     end
62 //     else begin
63 //         ir_data = ir_data;
64 //     end
65 // end
66
67
68 always @ (posedge i_clk or negedge i_rst_n) begin
69     if (!i_rst_n) begin
70         CAR <= 7'b0;
71         indirect_done <= 1'b0;
72     end
73     else begin
74         case (i_control_word_car)
75             2'b01: begin // Jump to execution or indirect
76                 // indirect at privilege
77                 if (indirect_flag && !indirect_done) begin
78                     CAR <= 7'h05;
79                     indirect_done <= 1'b1;
80                 end
81                 else begin
82                     case (ir_data[3:0])
83                         4'd1: begin
84                             if (i_ctrl_MF) begin
85                                 CAR <= 7'h21; // STORE & STOREH
86                             end
87                             else begin
88                                 CAR <= 7'h07; // STORE Only
89                             end
90                         end
91                         4'd2:
92                             CAR <= 7'h09; // LOAD
93                         4'd3:
94                             CAR <= 7'h0B; // ADD
95                         4'd4:
96                             CAR <= 7'h0D; // SUB
97
98                         4'd5: begin // JGZ
99                             if (i_ctrl_ZF || i_ctrl_NF)

```

```

100          CAR <= 7'h00;
101      else
102          CAR <= 7'h11;
103      end
104 4'd6:
105      CAR <= 7'h11; // JMP
106 4'd7:
107      CAR <= 7'h13; // HALT
108 4'd8:
109      CAR <= 7'h0F; // MPY
110 4'd9:
111      CAR <= 7'h15; // AND
112 4'd10:
113      CAR <= 7'h17; // OR
114 4'd11:
115      CAR <= 7'h19; // NOT
116 4'd12:
117      CAR <= 7'h1B; // SHIFTR
118 4'd13:
119      CAR <= 7'h1D; // SHIFTL
120 default:
121     CAR <= 7'h00;
122 endcase
123 end
124
125 2'b10: begin
126     CAR <= CAR + 1; // Next Micro-instruction
127 end
128 2'b11: begin
129     if (i_ctrl_halt) begin
130         // Privilege HALT
131         CAR <= CAR;
132     end
133     else if (ctrl_step_execution) begin
134         // Step-by-step instruction fetch
135         if (i_next_instr_stimulus) begin
136             CAR <= 7'h00;
137             indirect_done <= 1'b0;
138         end
139         else begin
140             // NOP WB Stage
141             CAR <= 7'h20;
142         end
143     end
144     else begin
145         // Auto fetch
146         CAR <= 7'h00; // Fetch next instruction

```

```

147         indirect_done <= 1'b0; // Reset Indirect Flag
148     end
149 end
150 default:
151     CAR <= CAR; // Prevent latch
152 endcase
153 end
154
155 assign o_car_data = ctrl_cpu_start ? CAR : 7'h20;
156
157 endmodule

```

Listing 8: cu_control_address_register.v

```

`timescale 1ns / 1ps

2
3 module CBR (
4     ctrl_cpu_start,
5     memory,
6     ctrl_global_halt,
7     ctrl_mar_increment,
8     next_addr,
9     ALU_op,
10    C0,
11    C1,
12    C2,
13    C3,
14    C4,
15    C5,
16    C6,
17    C7,
18    C8,
19    C9,
20    C10,
21    C11,
22    C12,
23    C13,
24    C14,
25    C15
26 );
27 input ctrl_cpu_start;
28 input [23:0] memory;
29 output ctrl_global_halt; // C23
30 output ctrl_mar_increment; // C22
31 output [1:0] next_addr; // C21-C20
32 output [3:0] ALU_op; // C19-C16
33 output C0, C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13, C14, C15;

```

```

34
35 assign C0 = ctrl_cpu_start & memory[0];
36 assign C1 = ctrl_cpu_start & memory[1];
37 assign C2 = ctrl_cpu_start & memory[2];
38 assign C3 = ctrl_cpu_start & memory[3];
39 assign C4 = ctrl_cpu_start & memory[4];
40 assign C5 = ctrl_cpu_start & memory[5];
41 assign C6 = ctrl_cpu_start & memory[6];
42 assign C7 = ctrl_cpu_start & memory[7];
43 assign C8 = ctrl_cpu_start & memory[8];
44 assign C9 = ctrl_cpu_start & memory[9];
45 assign C10 = ctrl_cpu_start & memory[10];
46 assign C11 = ctrl_cpu_start & memory[11];
47 assign C12 = ctrl_cpu_start & memory[12];
48 assign C13 = ctrl_cpu_start & memory[13];
49 assign C14 = ctrl_cpu_start & memory[14];
50 assign C15 = ctrl_cpu_start & memory[15];

51
52 assign ALU_op = ctrl_cpu_start ? memory[19:16] : 4'b0;
53
54 assign next_addr = ctrl_cpu_start ? memory[21:20] : 2'b0;
55 assign ctrl_mar_increment = ctrl_cpu_start & memory[22];
56 assign ctrl_global_halt = ctrl_cpu_start & memory[23];
57
58 endmodule

```

Listing 9: cu_control_buffer_register.v

```

1 // The top module of the CU
2 // Author: LiPtP
3 // Date: 2025.4.25
4 // Should be connected to:
5 // * All internal registers via Internal Bus
6 // * ALU
7 // * External Bus
8 // 4.28 Update: Add start_cpu signal to control the CPU execution
9 module CU_TOP (
10     ctrl_cpu_start,
11     ctrl_step_execution,
12     i_next_instr_stimulus,
13     i_clk,
14     i_rst_n,
15     i_flags,
16     i_ir_data,
17     o_alu_op,
18     o_ctrl_halt,
19     o_IF_stage,
20     o_ctrl_mar_increment,

```

```

21      C0,
22      C1,
23      C2,
24      C3,
25      C4,
26      C5,
27      C6,
28      C7,
29      C8,
30      C9,
31      C10,
32      C11,
33      C12,
34      C13,
35      C14,
36      C15
37  );
38
39 // External signals
40 input ctrl_cpu_start;
41 input ctrl_step_execution;
42 input i_next_instr_stimulus;
43 input i_clk;
44 input i_rst_n;
45 input [7:0] i_ir_data;
46 input [4:0] i_flags; // ZF, CF, OF, NF, MF
47
48 output [3:0] o_alu_op;
49 output o_ctrl_mar_increment; // C23
50 output o_IF_stage; // C2
51 output o_ctrl_halt; // C23
52 output C0, C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13, C14, C15;
53
54 // Internal signals
55
56 wire [ 1:0] next_addr;
57 wire [ 6:0] car_data;
58 wire [23:0] control_word;
59
60
61 CAR control_CAR (
62     .ctrl_cpu_start(ctrl_cpu_start),
63     .ctrl_step_execution(ctrl_step_execution),
64     .i_next_instr_stimulus(i_next_instr_stimulus),
65     .i_clk(i_clk),
66     .i_rst_n(i_rst_n),
67     .i_control_word_car(next_addr),

```

```

68     .i_ir_data({i_ir_data[7], i_ir_data[3:0]}),
69     .i_ctrl_ZF(i_flags[4]),
70     .i_ctrl_NF(i_flags[1]),
71     .i_ctrl_MF(i_flags[0]),
72     .i_ctrl_halt(o_ctrl_halt),
73     .o_car_data(car_data)
74   );
75
76 CONTROL_MEMORY control_memory (
77   .car(car_data),
78   .control_word(control_word)
79 );
80
81 CBR control_CBR (
82   .ctrl_cpu_start(ctrl_cpu_start),
83   .memory(control_word),
84   .ctrl_global_halt(o_ctrl_halt),
85   .ctrl_mar_increment(o_ctrl_mar_increment),
86   .next_addr(next_addr),
87   .ALU_op(o_alu_op),
88   .C0(C0),
89   .C1(C1),
90   .C2(C2),
91   .C3(C3),
92   .C4(C4),
93   .C5(C5),
94   .C6(C6),
95   .C7(C7),
96   .C8(C8),
97   .C9(C9),
98   .C10(C10),
99   .C11(C11),
100  .C12(C12),
101  .C13(C13),
102  .C14(C14),
103  .C15(C15)
104 );
105
106
107 // Assignments
108
109 assign o_IF_stage = C2;
110
111 endmodule

```

Listing 10: cu_top.v

A.4 内部寄存器与 ALU 设计

```
/*
module ALU
Author: LiPtP
function:
1. update BR and MR registers on rising clock edge when `ctrl_alu_en` is open;
2. Bus control using C9, C10, the target port is o_mr and o_br;
3. Operation encoding is defined in doc.
4. Currently, this ALU is the optimal design within all tried methods.
*/
module ALU (
    i_clk,
    i_rst_n,
    i_acc_alu_p,
    i_acc_alu_q,
    ctrl_alu_op,
    ctrl_alu_en,
    C9,
    C10,
    o_mr,
    o_br,
    o_flags,
    i_user_sample,
    o_mr_user
);
input i_clk;
input i_rst_n;
input [15:0] i_acc_alu_p;
input [15:0] i_acc_alu_q;
input [2:0] ctrl_alu_op;
input ctrl_alu_en;
input C9;
input C10;
output [15:0] o_mr;
output [15:0] o_br;
output [4:0] o_flags;
input i_user_sample;
output [15:0] o_mr_user;

// Re-interpret input to signed values
wire signed [15:0] ALU_P;
wire signed [15:0] ALU_Q;

// Calculation result
reg signed [15:0] ALU_RES_LOW;
reg signed [15:0] ALU_RES_HIGH;
```

```

46
47 // Output registers
48 reg [15:0] BR;
49 reg [15:0] MR;
50
51 // Flags
52 reg ZF, CF, OF, NF, MF;
53
54 // Combinational logic: ALU Operation
55 always @(*) begin
56     // Default
57     ALU_RES_LOW = 16'b0;
58     ALU_RES_HIGH = 16'b0;
59
60     case (ctrl_alu_op)
61         3'b000: begin // ADD
62             if(MF) begin
63                 {ALU_RES_HIGH, ALU_RES_LOW} = {MR, ALU_P} + {16'b0, ALU_Q};
64             end
65             else begin
66                 ALU_RES_LOW = ALU_P + ALU_Q;
67             end
68         end
69         3'b001: begin // SUB
70             if(MF) begin
71                 {ALU_RES_HIGH, ALU_RES_LOW} = {MR, ALU_P} - {16'b0, ALU_Q};
72             end
73             else begin
74                 ALU_RES_LOW = ALU_P - ALU_Q;
75             end
76         end
77         3'b010: begin // MPY
78             {ALU_RES_HIGH, ALU_RES_LOW} = ALU_P * ALU_Q;
79         end
80         3'b011: begin // AND
81             ALU_RES_LOW = ALU_P & ALU_Q;
82         end
83         3'b100: begin // OR
84             ALU_RES_LOW = ALU_P | ALU_Q;
85         end
86         3'b101: begin // NOT
87             ALU_RES_LOW = ~ALU_Q;
88         end
89         3'b110: begin // SHIFTR
90             ALU_RES_LOW = ALU_P >>> ALU_Q;
91         end
92         3'b111: begin // SHIFTL

```

```

93         ALU_RES_LOW = ALU_P <<< ALU_Q;
94     end
95     default: begin
96         ALU_RES_LOW = 16'b0;
97         ALU_RES_HIGH = 16'b0;
98     end
99 endcase
100 end
101
102 // Sequential logic: Update BR and MR upon ctrl_alu_en
103 always @(posedge i_clk or negedge i_rst_n) begin
104     if (!i_rst_n) begin
105         BR <= 16'b0;
106         MR <= 16'b0;
107     end
108     else if (ctrl_alu_en) begin
109         BR <= ALU_RES_LOW;
110         if(ctrl_alu_op == 3'b010) begin
111             MR <= ALU_RES_HIGH;
112         end
113         else begin
114             MR <= MR;
115         end
116     end
117
118     // On write back, MR are set to 0
119     else if (C10 && !i_user_sample) begin
120         MR <= 16'b0;
121     end
122     else begin
123         BR <= BR;
124         MR <= MR;
125     end
126 end
127
128 // Sequential logic: Update Flags upon ctrl_alu_en
129 always @(posedge i_clk or negedge i_rst_n) begin
130     if (!i_rst_n) begin
131         ZF <= 1'b0;
132         CF <= 1'b0;
133         OF <= 1'b0;
134         NF <= 1'b0;
135         MF <= 1'b0;
136     end
137     else if (ctrl_alu_en) begin // EX
138         if (ctrl_alu_op == 3'b000) begin // ADD
139             OF <= MF ? ((MR[15] == ALU_P[15]) && (ALU_RES_HIGH[15] != MR[15])) :

```

```

140          ((ALU_P[15] == ALU_Q[15]) && (ALU_RES_LOW[15] != ALU_P[15]));
141      end
142      else if (ctrl_alu_op == 3'b001) begin // SUB
143          OF <= MF ? ((MR[15] != ALU_P[15]) && (ALU_RES_HIGH[15] != MR[15])) :
144              ((ALU_P[15] != ALU_Q[15]) && (ALU_RES_LOW[15] != ALU_P[15]));
145      end
146      else if (ctrl_alu_op == 3'b010) begin // MPY
147          OF <= MF ? ((ALU_P[15] == ALU_Q[15]) && (ALU_RES_HIGH[15] != 16'b0)) :
148              ((ALU_P[15] == ALU_Q[15]) && (ALU_RES_LOW[15] != 16'b0));
149      end
150      else begin
151          OF <= 1'b0;
152      end
153      CF <= (ctrl_alu_op == 3'b110) ? ALU_P[15 - ALU_Q] : // SHIFTL highest shiftout bit
154          (ctrl_alu_op == 3'b111) ? ALU_P[ALU_Q] : 1'b0; // SHIFTR lowest shiftout bit
155  end
156  else if (C9) begin // WB
157      ZF <= ({MR, BR} == 32'b0);                                // Wait one cycle to update ZF
158      NF <= (MR != 16'b0) ? MR[15] : BR[15];
159      MF <= (MR != 16'b0); // only for STOREH
160  end
161  else begin
162      ZF <= ZF;
163      CF <= CF;
164      OF <= OF;
165      NF <= NF;
166      MF <= (MR != 16'b0); // preventing one cycle ctrl_en
167  end
168 end
169
170
171 // Input
172 assign ALU_P = i_acc_alu_p;
173 assign ALU_Q = i_acc_alu_q;
174
175 // Output
176 assign o_br = C9 ? BR : 16'b0;
177 assign o_mr = C10 ? MR : 16'b0;
178 assign o_flags = {ZF, CF, OF, NF, MF};
179 assign o_mr_user = i_user_sample ? MR : 16'b0;
180 endmodule

```

Listing 11: alu.v

```

1 module ACC (
2     i_clk,
3     i_rst_n,
4     i_br_acc,

```

```

5      i_mr_acc,
6      i_mbr_acc,
7      C7,
8      C9,
9      C10,
10     C11,
11     C12,
12     o_acc_alu_p,
13     o_acc_mbr,
14     i_user_sample,
15     o_acc_user
16   );
17
18 input i_clk;
19 input i_rst_n;
20 input [15:0] i_br_acc;
21 input [15:0] i_mr_acc;
22 input [15:0] i_mbr_acc;
23 input C7;
24 input C9;
25 input C10;
26 input C11;
27 input C12;
28 input i_user_sample;
29 output [15:0] o_acc_alu_p;
30 output [15:0] o_acc_mbr;
31 output [15:0] o_acc_user;
32 reg [15:0] ACC;
33
34 always @ (posedge i_clk or negedge i_rst_n) begin
35   if (!i_rst_n) begin
36     ACC <= 16'b0;
37   end
38   else begin
39     if (C9) begin
40       ACC <= i_br_acc;
41     end
42     else if (C10) begin
43       ACC <= i_mr_acc;
44     end
45     else if (C11) begin
46       ACC <= i_mbr_acc;
47     end
48     else begin
49       ACC <= ACC;
50     end
51   end
52 end

```

```

52
53 assign o_acc_alu_p = C7 ? ACC : 16'b0;
54 assign o_acc_mbr = C12 ? ACC : 16'b0;
55 assign o_acc_user = i_user_sample ? ACC : 16'b0;
56 endmodule

```

Listing 12: acc.v

```

/*
module MAR
Author: LiPtP
function:
1. self increment upon STOREH implicit instruction
2. write value sequence: MBR > PC
*/
`timescale 1ns / 1ps
module MAR (
    i_clk,
    i_rst_n,
    i_mbr_mar,
    i_pc_mar,
    C2,
    C8,
    ctrl_mar_increment,
    o_mar_address_bus
);
input i_clk;
input i_rst_n;
input ctrl_mar_increment;
input C2;
input C8;
input [7:0] i_mbr_mar;
input [7:0] i_pc_mar;
output [7:0] o_mar_address_bus;

reg [7:0] MAR;

always @ (posedge i_clk or negedge i_rst_n) begin
    if (!i_rst_n) begin
        MAR <= 8'b0;
    end
    else begin
        if (ctrl_mar_increment) begin
            MAR <= MAR + 1;
        end
        else begin
            if (C8) begin
                MAR <= i_mbr_mar;
            end
        end
    end
end

```

```

41         end
42
43     else if (C2) begin
44         MAR <= i_pc_mar;
45     end
46
47     else begin
48         MAR <= MAR;
49     end
50
51
52 // Address bus judgement logic at reg_top
53 assign o_mar_address_bus = MAR;
54
55 endmodule

```

Listing 13: mar.v

```

1 /*
2 module MBR
3 Author: LiPtP
4 function:
5 1. write value sequence: Bus > IR > PC > ACC
6 */
7 `timescale 1ns / 1ps
8 module MBR (
9     i_clk,
10    i_rst_n,
11    i_pc_mbr,
12    i_ir_mbr,
13    i_data_bus_mbr,
14    i_acc_mbr,
15    o_mbr_data_bus,
16    o_mbr_pc,
17    o_mbr_ir,
18    o_mbr_mar,
19    o_mbr_acc,
20    o_mbr_alu_q,
21    C1,
22    C3,
23    C4,
24    C5,
25    C6,
26    C8,
27    C11,
28    C12,
29    C15
30 );
31 input i_clk;

```

```

32    input i_RST_N;
33    input [7:0] i_PC_MBR;
34    input [7:0] i_IR_MBR;
35    input [15:0] i_Data_Bus_MBR;
36    input [15:0] i_ACC_MBR;
37    input C1;
38    input C3;
39    input C4;
40    input C5;
41    input C6;
42    input C8;
43    input C11;
44    input C12;
45    input C15;
46    output [15:0] o_MBR_Data_Bus;
47    output [7:0] o_MBR_pc;

48
49 // IR stages the storage of MBR on ID Stage, in order that MBR can directly receive immaculate
50 // operand on immediate addressing.
51
52    output [15:0] o_MBR_ir;
53
54    output [7:0] o_MBR_mar;
55    output [15:0] o_MBR_acc;
56    output [15:0] o_MBR_alu_q;

57
58    reg [15:0] MBR;
59
60
61    always @ (posedge i_clk or negedge i_RST_N) begin
62        if (!i_RST_N) begin
63            MBR <= 16'b0;
64        end
65        else begin
66            if (C5) begin
67                MBR <= i_Data_Bus_MBR;
68            end
69            else if (C15) begin
70                MBR <= {8'b0, i_IR_MBR};
71            end
72            else if (C1) begin
73                MBR <= {8'b0, i_PC_MBR};
74            end
75            else if (C12) begin
76                MBR <= i_ACC_MBR;
77            end
78            else begin
79                MBR <= MBR;
80            end
81        end
82    end

```

```

78     end
79 end
80
81 assign o_mbr_acc = C11 ? MBR : 16'b0;
82
83 assign o_mbr_alu_q = C6 ? MBR : 16'b0;
84 assign o_mbr_ir = C4 ? MBR : 16'b0;
85 assign o_mbr_mar = C8 ? MBR[7:0] : 8'b0;
86 assign o_mbr_pc = C3 ? MBR[7:0] : 8'b0;
87
88 // Data bus judgement logic at reg_top
89 assign o_mbr_data_bus = MBR;
90
91 endmodule

```

Listing 14: mbr.v

```

1 /*
2 module PC
3 Author: LiPtP
4 function:
5 1. self increment upon C2
6 2. write value sequence: MBR
7 3. PC starts from 1 (0428 updated)
8 */
9 module PC (
10     i_clk,
11     i_rst_n,
12     i_mbr_pc,
13     C1,
14     C2,
15     C3,
16     o_pc_mar,
17     o_pc_mbr,
18     i_user_sample,
19     o_pc_user
20 );
21 input i_clk;
22 input i_rst_n;
23 input i_user_sample;
24 input [7:0] i_mbr_pc;
25 input C1;
26 input C2;
27 input C3;
28 output [7:0] o_pc_mar;
29 output [7:0] o_pc_mbr;
30 output [7:0] o_pc_user;
31 reg [7:0] PC;

```

```

32
33    always @(posedge i_clk or negedge i_RST_N) begin
34        if (!i_RST_N) begin
35            PC <= 8'd1;
36        end
37        else begin
38            // when C2 is open, it must be fetch stage
39            if (C2) begin
40                PC <= PC + 1;
41            end
42            else begin
43                PC <= C3 ? i_MBR_pc : PC;
44            end
45        end
46    end
47
48    assign o_pc_mbr = C1 ? PC : 8'b0;
49    assign o_pc_mar = C2 ? PC : 8'b0;
50    assign o_pc_user = i_user_sample ? PC : 8'b0;
51
endmodule

```

Listing 15: pc.v

```

1  /*
2  module IR
3  Author: LiPtP
4  function:
5  1. dump high 8 bits to CU
6  2. store immediate opcode and operand from MBR and push back operand on F0 stage
7  */
8  module IR (
9      i_clk,
10     i_RST_N,
11     i_MBR_ir,
12     C4,
13     C14,
14     C15,
15     o_ir_cu,
16     o_ir_mbr,
17     i_user_sample,
18     o_ir_user
19 );
20
21 input i_clk;
22 input i_RST_N;
23 input [15:0] i_MBR_ir;
24 input C4;
25 input C14;

```

```

26    input C15;
27    input i_user_sample;
28    output [7:0] o_ir_cu;
29    output [7:0] o_ir_mbr;
30    output [7:0] o_ir_user;
31
32    reg [7:0] IR_opcode;
33    reg [7:0] IR_operand;
34
35    always @(posedge i_clk or negedge i_rst_n) begin
36        if (!i_rst_n) begin
37            IR_opcode <= 8'b0;
38            IR_operand <= 8'b0;
39        end
40        else begin
41            IR_operand <= C4 ? i_mbr_ir[7:0] : IR_operand;
42            IR_opcode <= C4 ? i_mbr_ir[15:8] : IR_opcode;
43        end
44    end
45
46    assign o_ir_cu = C14 ? IR_opcode : 8'b0;
47    assign o_ir_mbr = C15 ? IR_operand : 8'b0;
48    assign o_ir_user = i_user_sample ? IR_opcode : 8'b0;
49
endmodule

```

Listing 16: ir.v

```

1  /*
2   module REG_TOP
3   Author: LiPtP
4
5   Should be connected with:
6   1. External Bus
7   2. Control Unit
8   and they should be at the same hierarchy level.
9   */
10
11 module REG_TOP(
12     ctrl_cpu_start,
13     i_user_sample,
14     o_ACC_user,
15     o_MR_user,
16     o_PC_user,
17     o_IR_user,
18     i_clk,
19     i_rst_n,
20     i_memory_data,
21     o_memory_addr,
22     o_memory_data,
23

```

```

22     o_ir_cu,
23     o_flags,
24     i_alu_op,
25     i_ctrl_halt,
26     i_ctrl_mar_increment,
27     C1,
28     C2,
29     C3,
30     C4,
31     C5,
32     C6,
33     C7,
34     C8,
35     C9,
36     C10,
37     C11,
38     C12,
39     C14,
40     C15
41 );
42
43 input ctrl_cpu_start;
44 input i_user_sample;
45 input i_clk;
46 input i_rst_n;
47 // From External Bus
48 input [15:0] i_memory_data;
49
50 // From Control Unit
51 input [3:0] i_alu_op; // C19 - C16
52 input i_ctrl_halt; // C23
53 input i_ctrl_mar_increment; // C22
54 input C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C14, C15;
55
56 // To External Bus
57 output [7:0] o_memory_addr;
58 output [15:0] o_memory_data;
59
60 // To Control Unit
61 output [7:0] o_ir_cu;
62 output [4:0] o_flags;
63
64 // To User Interface
65 output [15:0] o_ACC_user;
66 output [15:0] o_MR_user;
67 output [7:0] o_PC_user;
68 output [7:0] o_IR_user;

```

```

69
70 // Internal signals (16 Data Path)
71
72 wire [7:0] MAR_ADDR_BUS; // C0
73 wire [7:0] PC_MBR; // C1
74 wire [7:0] PC_MAR; // C2
75 wire [7:0] MBR_PC; // C3
76 wire [15:0] MBR_IR; // C4
77 wire [15:0] DATA_BUS_MBR; // C5
78 wire [15:0] MBR_ALU_Q; // C6
79 wire [15:0] ACC_ALU_P; // C7
80 wire [7:0] MBR_MAR; // C8
81 wire [15:0] BR_ACC; // C9
82 wire [15:0] MR_ACC; // C10
83 wire [15:0] MBR_ACC; // C11
84 wire [15:0] ACC_MBR; // C12
85 wire [15:0] MBR_DATA_BUS; // C13
86 wire [7:0] IR CU; // C14
87 wire [7:0] IR_MBR; // C15
88
89 // Instantiate the registers
90
91 ACC reg_ACC(
92     .i_clk(i_clk),
93     .i_rst_n(i_rst_n),
94     .i_br_acc(BR_ACC),
95     .i_mr_acc(MR_ACC),
96     .i_mbr_acc(MBR_ACC),
97     .C7(C7),
98     .C9(C9),
99     .C10(C10),
100    .C11(C11),
101    .C12(C12),
102    .o_acc_alu_p(ACC_ALU_P),
103    .o_acc_mbr(ACC_MBR),
104    .i_user_sample(i_user_sample),
105    .o_acc_user(o_ACC_user)
106 );
107
108 // The first command in CM is open C2
109 PC reg_PC(
110     .i_clk(i_clk),
111     .i_rst_n(i_rst_n),
112     .i_mbr_pc(MBR_PC),
113     .C1(C1),
114     .C2(C2 & ctrl_cpu_start),
115     .C3(C3),

```

```

116     .o_pc_mar(PC_MAR),
117     .o_pc_mbr(PC_MBR),
118     .i_user_sample(i_user_sample),
119     .o_pc_user(o_PC_user)
120   );
121
122 MBR reg_MBR(
123   .i_clk(i_clk),
124   .i_rst_n(i_rst_n),
125   .i_pc_mbr(PC_MBR),
126   .i_ir_mbr(IR_MBR),
127   .i_data_bus_mbr(DATA_BUS_MBR),
128   .i_acc_mbr(ACC_MBR),
129   .o_mbr_data_bus(MBR_DATA_BUS),
130   .o_mbr_pc(MBR_PC),
131   .o_mbr_ir(MBR_IR),
132   .o_mbr_mar(MBR_MAR),
133   .o_mbr_acc(MBR_ACC),
134   .o_mbr_alu_q(MBR_ALU_Q),
135   .C1(C1),
136   .C3(C3),
137   .C4(C4),
138   .C5(C5),
139   .C6(C6),
140   .C8(C8),
141   .C11(C11),
142   .C12(C12),
143   .C15(C15)
144 );
145
146 MAR reg_MAR(
147   .i_clk(i_clk),
148   .i_rst_n(i_rst_n),
149   .i_mbr_mar(MBR_MAR),
150   .i_pc_mar(PC_MAR),
151   .ctrl_mar_increment(i_ctrl_mar_increment),
152   .o_mar_address_bus(MAR_ADDR_BUS),
153   .C2(C2),
154   .C8(C8)
155 );
156
157 ALU reg_ALU(
158   .i_clk(i_clk),
159   .i_rst_n(i_rst_n),
160   .i_acc_alu_p(ACC_ALU_P),
161   .i_acc_alu_q(MBR_ALU_Q),
162   .ctrl_alu_op(i_alu_op[2:0]),
163   .ctrl_alu_en(i_alu_op[3]),
164   .C9(C9),

```

```

163     .C10(C10),
164     .o_mr(MR_ACC),
165     .o_br(BR_ACC),
166     .o_flags(o_flags),
167     .i_user_sample(i_user_sample),
168     .o_mr_user(o_MR_user)
169   );
170
171 IR reg_IR(
172   .i_clk(i_clk),
173   .i_rst_n(i_rst_n),
174   .i_mbr_ir(MBR_IR),
175   .C4(C4),
176   .C14(C14),
177   .C15(C15),
178   .o_ir_cu(IR_CU),
179   .o_ir_mbr(IR_MBR),
180   .i_user_sample(i_user_sample),
181   .o_ir_user(o_IR_user)
182 );
183
184 // Assignments to external bus
185 // Logic are defined in external_bus module
186 assign o_memory_data = MBR_DATA_BUS;
187 assign o_memory_addr = MAR_ADDR_BUS;
188 assign DATA_BUS_MBR = i_memory_data;
189
190 // Assignments to CU
191 assign o_ir_cu = i_ctrl_halt ? 8'b0 : IR_CU;
192
193 // Assignments to User interface
194 // As they are existing for only one clock cycle, the signals should be stored at user
195 // interface.
196
197 endmodule

```

Listing 17: reg_top.v

A.5 数据内存设计

```

/*
module DATA_RAM
Author: LiPtP
function:
0. Write means write to RAM, READ means read from RAM
1. Write data to itself according to input address and data
2. Output data according to input address
*/

```

```

9  module DATA_RAM (
10    i_clk,
11    ctrl_write,
12    i_addr_write,
13    i_data_write,
14    ctrl_read,
15    i_addr_read,
16    o_data_read
17  );
18  input      i_clk;
19  input      ctrl_write;
20  input [7:0] i_addr_write;
21  input [15:0] i_data_write;
22  input      ctrl_read;
23  input [7:0] i_addr_read;
24  output [15:0] o_data_read;
25
26 // 256 x 16 RAM storage
27 reg [15:0] mem [0:255];
28
29 // Write Operation, no initialization of data RAM
30 integer i;
31 always @ (posedge i_clk) begin
32
33   if (ctrl_write) begin
34     mem[i_addr_write] <= i_data_write;
35   end
36 end
37
38
39 // Read Operation
40 assign o_data_read = ctrl_read ? mem[i_addr_read] : 16'b0;
41
42 endmodule

```

Listing 18: top_data_ram.v

A.6 外部总线设计

```

2  module EXTERNAL_BUS (
3    i_clk,
4    i_rst_n,
5    i_mbr_data_bus,
6    i_mar_address_bus,
7    i_instr,
8    i_data,

```

```

9      o_data_bus_mbr,
10     o_data_bus_memory,
11     o_address_bus_memory,
12     o_instr_rom_read,
13     o_data_ram_read,
14     o_data_ram_write,
15     C0,
16     C2,
17     C5,
18     C13
19   );
20
21 input i_clk;
22 input i_rst_n;
23
24 input C0;
25 input C2;
26 input C5;
27 input C13;
28
29 // reg <-> bus
30
31 input [15:0] i_mbr_data_bus;
32 input [7:0] i_mar_address_bus;
33 output [15:0] o_data_bus_mbr;
34
35 // memory <-> bus
36
37 input [15:0] i_instr; // Instruction from Instr ROM
38 input [15:0] i_data; // Data from Data RAM
39 output o_instr_rom_read;
40 output o_data_ram_read;
41 output o_data_ram_write;
42 output [15:0] o_data_bus_memory;
43 output [7:0] o_address_bus_memory;
44
45 wire memory_read_en = C0 & C5; // Memory read enable,
46 wire memory_write_en = C0 & C13; // Memory write enable
47
48 reg [15:0] DATA_BUS;
49 wire [7:0] ADDRESS_BUS = i_mar_address_bus;
50
51 reg memory_select;
52
53 // Memory Select logic on t1
54 always @ (posedge i_clk or negedge i_rst_n) begin
55   if (!i_rst_n) begin

```

```

56     memory_select <= 1'b0; // Default to RAM
57
58 end
59
60 else begin
61     if(C2) begin
62         memory_select <= 1'b1; // ROM
63     end
64     else begin
65         memory_select <= 1'b0; // RAM
66     end
67 end
68
69
70 // Data Bus
71 always @(*) begin
72     if(memory_read_en) begin
73         DATA_BUS = memory_select ? i_instr : i_data;
74     end
75     else if (memory_write_en) begin
76         DATA_BUS = i_mbr_data_bus;
77     end
78     else begin
79         DATA_BUS = 16'b0;
80     end
81 end
82
83 // Control Bus
84
85 assign o_instr_rom_read = memory_select & memory_read_en;
86 assign o_data_ram_read = ~memory_select & memory_read_en;
87 assign o_data_ram_write = ~memory_select & memory_write_en;
88
89 // Data Connections
90
91 assign o_data_bus_mbr = memory_read_en ? DATA_BUS : 16'b0;
92 assign o_data_bus_memory = memory_write_en ? DATA_BUS : 16'b0;
93
94 // memory r/w both need address bus
95 assign o_address_bus_memory = (memory_write_en || memory_read_en) ? ADDRESS_BUS : 8'b0;
96 endmodule

```

Listing 19: external_bus.v

A.7 用户面设计

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

```

5   module KEY_JITTER #((
6     parameter CNT_MAX = 20'hf_ffff,
7     parameter POSEDGE = 1'b0
8   )((
9     i_clk,
10    key_in,
11    key_out
12  );
13
14  input i_clk;
15  input key_in;
16  output key_out;
17
18
19  reg [1:0] key_in_r;
20  reg [19:0] cnt_base;
21  reg key_value_r;
22  reg key_value_rd;
23  reg key_posedge;
24
25
26 // Sample and stage key input
27 always @(posedge i_clk) begin
28   key_in_r <= {key_in_r[0], key_in};
29 end
30
31
32 // Counter starts if key changes
33 always @(posedge i_clk) begin
34   if (key_in_r[0] != key_in_r[1]) begin
35     cnt_base <= 20'b0;
36   end
37   else if(key_in_r[0] == key_in_r[1]) begin
38     if (cnt_base < CNT_MAX) begin
39       cnt_base <= cnt_base + 1'b1;
40     end
41     else if (cnt_base == CNT_MAX) begin
42       cnt_base <= 20'b0;
43     end
44   end
45   else begin
46     // reset logic
47     cnt_base <= 20'b0;
48   end
49 end
50
51
52 // Prevent Synthesis from optimizing out the register
53 always @(posedge i_clk) begin

```

```

49     if (cnt_base == CNT_MAX) begin
50         key_value_r <= key_in_r[0];
51     end
52 end
53
54 always @ (posedge i_clk) begin
55     key_value_rd <= key_value_r;
56     key_posedge <= (key_value_r & ~key_value_rd);
57 end
58
59 assign key_out = POSEDGE ? key_posedge : key_value_rd;
60
61 endmodule

```

Listing 20: key_jitter.v

```

1 module SEVEN_SEGMENT_DISPLAY #(
2     parameter SCAN_INTERVAL = 16'd49999
3 ) (
4     i_clk,
5     i_RST_N,
6     switch_start_cpu,
7     button_check_instruction,
8     button_check_flags,
9     button_check_result,
10    light_instr_transmit_done,
11    segment_current_PC,
12    segment_current_Opcode,
13    segment_flags,
14    segment_result_high,
15    segment_result_low,
16    segment_max_instr_addr,
17    o_seg_valid,
18    o_seg_value
19 );
20
21 input i_clk;
22 input i_RST_N;
23 input [7:0] segment_current_PC;
24 input [7:0] segment_current_Opcode;
25 input [4:0] segment_flags;
26 input [15:0] segment_result_high;
27 input [15:0] segment_result_low;
28 input [7:0] segment_max_instr_addr;
29 input button_check_instruction;
30 input button_check_flags;
31 input button_check_result;
32 input light_instr_transmit_done;
33 input switch_start_cpu;

```

```

32     output [7:0] o_seg_valid;
33     output [7:0] o_seg_value;
34
35
36     reg [31:0] segment_data;
37
38 // 状态定义
39     localparam STATE_DEFAULT = 3'b000;
40     localparam STATE_INSTRUCTION = 3'b001;
41     localparam STATE_FLAGS = 3'b010;
42     localparam STATE_RESULT = 3'b011;
43     localparam STATE_MAX_ADDR = 3'b100;
44
45
46     reg [2:0] current_state, next_state;
47
48 // 状态机: 状态切换逻辑
49     always @(posedge i_clk or negedge i_rst_n) begin
50         if (!i_rst_n) begin
51             current_state <= STATE_DEFAULT;
52         end
53         else begin
54             current_state <= next_state;
55         end
56     end
57
58 // 状态机: 下一状态逻辑
59     always @(*) begin
60         case (current_state)
61             STATE_DEFAULT: begin
62                 if (button_check_instruction) begin
63                     next_state = STATE_INSTRUCTION;
64                 end
65                 else if (button_check_flags) begin
66                     next_state = STATE_FLAGS;
67                 end
68                 else if (button_check_result) begin
69                     next_state = STATE_RESULT;
70                 end
71                 else if (light_instr_transmit_done && !switch_start_cpu) begin
72                     next_state = STATE_MAX_ADDR;
73                 end
74                 else begin
75                     next_state = STATE_DEFAULT;
76                 end
77             end
78             STATE_INSTRUCTION: begin
79                 if (!button_check_instruction) begin
80                     next_state = STATE_DEFAULT;
81                 end
82             end
83         endcase
84     end

```

```

80
81         end
82     else begin
83         next_state = STATE_INSTRUCTION;
84     end
85 end
86
87 STATE_FLAGS: begin
88     if (!button_check_flags) begin
89         next_state = STATE_DEFAULT;
90     end
91     else begin
92         next_state = STATE_FLAGS;
93     end
94 end
95
96 STATE_RESULT: begin
97     if (!button_check_result) begin
98         next_state = STATE_DEFAULT;
99     end
100    else begin
101        next_state = STATE_RESULT;
102    end
103 end
104
105 STATE_MAX_ADDR: begin
106     if (!(light_instr_transmit_done && !switch_start_cpu)) begin
107         next_state = STATE_DEFAULT;
108     end
109     else begin
110         next_state = STATE_MAX_ADDR;
111     end
112 end
113 default: begin
114     next_state = STATE_DEFAULT;
115 end
116 endcase
117
118 end
119
120 // 状态机: 输出逻辑
121
122 always @ (posedge i_clk or negedge i_rst_n) begin
123     if (!i_rst_n) begin
124         segment_data <= 32'b0;
125     end
126     else begin
127         case (current_state)
128             STATE_DEFAULT: begin
129                 segment_data <= segment_data;
130             end
131             STATE_INSTRUCTION: begin
132                 segment_data <= {8'b0, segment_current_PC, 8'b0, segment_current_0opcode};
133             end
134         endcase
135     end
136 end

```

```

127     end
128
129     STATE_FLAGS: begin
130         segment_data <= {15'b0,segment_flags[4],3'b0,segment_flags[3],3'b0,segment_flags
131             [2],3'b0,segment_flags[1],3'b0,segment_flags[0]};
132     end
133
134     STATE_RESULT: begin
135         segment_data <= {segment_result_high, segment_result_low};
136     end
137
138     STATE_MAX_ADDR: begin
139         segment_data <= {24'b0, segment_max_instr_addr};
140     end
141
142     default: begin
143         segment_data <= 32'b0;
144     end
145
146     endcase
147
148 end
149
150
151 SEVEN_SEGMENT #(
152     .SCAN_INTERVAL(SCAN_INTERVAL)
153 )
154     seven_segment (
155         .i_clk(i_clk),
156         .i_rst_n(i_rst_n),
157         .i_data(segment_data),
158         .o_seg_valid(o_seg_valid),
159         .o_seg_value(o_seg_value)
160     );
161
162 endmodule

```

Listing 21: top_seven_segment.v

```

`timescale 1ns / 1ps
module SEVEN_SEGMENT #(
    parameter SCAN_INTERVAL = 16'd49999
) (
    i_clk,
    i_rst_n,
    i_data,
    o_seg_valid,
    o_seg_value
);
    input i_clk;
    input i_rst_n;
    input [31:0] i_data;
    output reg [7:0] o_seg_valid;

```

```

18 output reg [7:0] o_seg_value;
19
20 // Clock Divider
21 reg [15:0] count_num;
22 always @(posedge i_clk or negedge i_rst_n) begin
23   if (!i_rst_n) begin
24     count_num <= 3'b0;
25   end
26   else begin
27     if (count_num == SCAN_INTERVAL) begin
28       count_num <= 16'd0;
29     end
30     else begin
31       count_num <= count_num + 1'd1;
32     end
33   end
34 end
35
36 // Segment Select & Polling the segments
37 reg [2:0] seg_num;
38 always @(posedge i_clk or negedge i_rst_n) begin
39   if (!i_rst_n) begin
40     seg_num <= 3'b0;
41     o_seg_valid <= 8'b0111_1111;
42   end
43   else begin
44     if (count_num == SCAN_INTERVAL) begin
45       if (seg_num == 3'd7) begin
46         seg_num <= 3'd0;
47         o_seg_valid <= 8'b0111_1111;
48       end
49       else begin
50         seg_num <= seg_num + 1'd1;
51         o_seg_valid <= {o_seg_valid[0], o_seg_valid[7:1]};
52       end
53     end
54   end
55 end
56
57 // Display Control
58 reg [4:0] display_value;
59 always @(*) begin
60   // Add Control Logic here to display other things
61   case (seg_num)
62     3'd0:
63       display_value = {1'b0, i_data[31:28]};
64     3'd1:

```

```

      display_value = {1'b0, i_data[27:24]};
5'd2:
      display_value = {1'b0, i_data[23:20]};
5'd3:
      display_value = {1'b0, i_data[19:16]};
5'd4:
      display_value = {1'b0, i_data[15:12]};
5'd5:
      display_value = {1'b0, i_data[11:8]};
5'd6:
      display_value = {1'b0, i_data[7:4]};
5'd7:
      display_value = {1'b0, i_data[3:0]};
default:
      display_value = 5'b0;
endcase
end

// Encoder of 7-segment display
localparam SEG_0 = 8'b1100_0000, SEG_1 = 8'b1111_1001,
SEG_2 = 8'b1010_0100, SEG_3 = 8'b1011_0000,
SEG_4 = 8'b1001_1001, SEG_5 = 8'b1001_0010,
SEG_6 = 8'b1000_0010, SEG_7 = 8'b1111_1000,
SEG_8 = 8'b1000_0000, SEG_9 = 8'b1001_0000,
SEG_A = 8'b1000_1000, SEG_B = 8'b1000_0011,
SEG_C = 8'b1100_0110, SEG_D = 8'b1010_0001,
SEG_E = 8'b1000_0110, SEG_F = 8'b1000_1110;

// Something else to display
localparam SEG_S = 8'b1011_1111, SEG_r = 8'b1010_1111,
SEG_o = 8'b1010_0011, SEG_n = 8'b1111_1111,
SEG_ot = 8'b1001_1100, SEG_left = 8'b1111_1100,
SEG_right = 8'b1101_1110, SEG_happy = 8'b1110_0011,
SEG_sad = 8'b1010_1011;

always @(*) begin
  case (display_value)
    5'd0:
      o_seg_value = SEG_0;
    5'd1:
      o_seg_value = SEG_1;
    5'd2:
      o_seg_value = SEG_2;
    5'd3:
      o_seg_value = SEG_3;
    5'd4:
      o_seg_value = SEG_4;
  endcase
end

```

```

112      5'd5:
113          o_seg_value = SEG_5;
114      5'd6:
115          o_seg_value = SEG_6;
116      5'd7:
117          o_seg_value = SEG_7;
118      5'd8:
119          o_seg_value = SEG_8;
120      5'd9:
121          o_seg_value = SEG_9;
122      5'd11:
123          o_seg_value = SEG_B;
124      5'd10:
125          o_seg_value = SEG_A;
126      5'd12:
127          o_seg_value = SEG_C;
128      5'd13:
129          o_seg_value = SEG_D;
130      5'd14:
131          o_seg_value = SEG_E;
132      5'd15:
133          o_seg_value = SEG_F;
134      5'd16:
135          o_seg_value = SEG_S;
136      5'd17:
137          o_seg_value = SEG_r;
138      5'd18:
139          o_seg_value = SEG_o;
140      5'd19:
141          o_seg_value = SEG_n;
142      5'd20:
143          o_seg_value = SEG_ot;
144      5'd21:
145          o_seg_value = SEG_left;
146      5'd22:
147          o_seg_value = SEG_right;
148      5'd23:
149          o_seg_value = SEG_happy;
150      5'd24:
151          o_seg_value = SEG_sad;
152      default:
153          o_seg_value = 8'b1;
154      endcase
155  end
156 endmodule

```

Listing 22: seven_segment.v

```

1 module LED_DISPLAY (
2     i_clk,
3     i_rst_n,
4     i_instr_transmit_done,
5     i_halt,
6     i_step_execution,
7     i_start_cpu,
8     RGB1_RED,
9     RGB1_BLUE,
10    RGB2_RED,
11    RGB2_BLUE,
12    RGB2_GREEN
13 );
14
15 input i_clk;
16 input i_rst_n;
17 input i_instr_transmit_done;
18 input i_halt;
19 input i_step_execution;
20 input i_start_cpu;
21 output RGB1_RED;
22 output RGB1_BLUE;
23 output RGB2_RED;
24 output RGB2_BLUE;
25 output RGB2_GREEN;
26
27
28 reg RGB1_RED_reg;
29 reg RGB1_BLUE_reg;
30 reg RGB2_RED_reg;
31 reg RGB2_BLUE_reg;
32 reg RGB2_GREEN_reg;
33
34
35 reg [1:0] state;
36 localparam STATE_IDLE = 2'b00;
37 localparam STATE_GREEN = 2'b01;
38 localparam STATE_BLUE = 2'b10;
39 localparam STATE_RED = 2'b11;
40
41
42 always @(posedge i_clk or negedge i_rst_n) begin
43     if (!i_rst_n) begin
44         state <= STATE_IDLE;
45         RGB2_RED_reg <= 1'b0;
46         RGB2_BLUE_reg <= 1'b0;
47         RGB2_GREEN_reg <= 1'b0;

```

```

47    end
48
49  else begin
50    case (state)
51      STATE_IDLE: begin
52        RGB2_GREEN_reg <= 1'b0;
53        RGB2_BLUE_reg <= 1'b0;
54        RGB2_RED_reg <= 1'b0;
55        if (i_instr_transmit_done) begin
56          state <= STATE_GREEN;
57        end
58      end
59      STATE_GREEN: begin
60        RGB2_GREEN_reg <= 1'b1;
61        RGB2_BLUE_reg <= 1'b0;
62        RGB2_RED_reg <= 1'b0;
63        if (i_start_cpu) begin
64          state <= STATE_BLUE;
65        end
66      end
67      STATE_BLUE: begin
68        RGB2_GREEN_reg <= 1'b0;
69        RGB2_BLUE_reg <= 1'b1;
70        RGB2_RED_reg <= 1'b0;
71        if (i_halt) begin
72          state <= STATE_RED;
73        end
74      end
75      STATE_RED: begin
76        RGB2_GREEN_reg <= 1'b0;
77        RGB2_BLUE_reg <= 1'b0;
78        RGB2_RED_reg <= 1'b1;
79      end
80      default: begin
81        state <= STATE_IDLE;
82      end
83    endcase
84  end
85
86 always @(posedge i_clk or negedge i_rst_n) begin
87   if(!i_rst_n) begin
88     RGB1_BLUE_reg <= 1'b0;
89     RGB1_RED_reg <= 1'b0;
90   end
91   else begin
92     if (i_step_execution) begin
93       RGB1_BLUE_reg <= 1'b0;
94       RGB1_RED_reg <= 1'b1;

```

```
94     end
95 else begin
96     RGB1_BLUE_reg <= 1'b1;
97     RGB1_RED_reg <= 1'b0;
98 end
99 end
100
101
102 // Assignments
103 assign RGB1_RED = RGB1_RED_reg;
104 assign RGB1_BLUE = RGB1_BLUE_reg;
105 assign RGB2_RED = RGB2_RED_reg;
106 assign RGB2_BLUE = RGB2_BLUE_reg;
107 assign RGB2_GREEN = RGB2_GREEN_reg;
108
109 endmodule
```

Listing 23: led_display.v