

计算机组织与结构 II: CPU 设计文档

李勃璘*

冯光宇†

张凯超‡

版本: 1.0

日期: 2025 年 5 月 8 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、用户输入输出设计和完整的硬件实现。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。

*东南大学吴健雄学院，学号：61522529

†东南大学吴健雄学院，学号：61522527

‡东南大学吴健雄学院，学号：61522532

目录

1 概述	1
1.1 中央处理单元 CPU 简介	1
1.2 NEXYS 4 DDR FPGA 简介	1
1.3 本文内容安排	2
2 CPU 结构设计	2
2.1 总体架构	2
2.2 指令集架构	3
2.2.1 位宽设计	3
2.2.2 指令集支持的指令	3
2.2.3 寻址方式	3
2.3 CPU 内部寄存器	4
2.4 算术逻辑单元 ALU	4
2.5 控制单元 CU	5
2.5.1 控制单元结构	5
2.5.2 微操作指令 (Micro-Operations)	5
2.5.3 CU 控制信号 (Control Signals)	7
2.6 CPU 内部总线和外部总线	9
2.7 数据内存	9
2.8 指令内存	9
3 用户交互设计	10
3.1 指令写入设计	10
3.1.1 用户端代码编译	10
3.1.2 UART 接收逻辑	11
3.1.3 FIFO 数据缓冲与存储结构	11
3.2 板载交互设备配置	11
4 核心模块设计	12
4.1 时钟、复位与停止信号	12
4.2 UART 传输与指令内存	12
4.2.1 UART 模块	14
4.2.2 FIFO 模块	15
4.2.3 指令内存模块	16
4.3 控制单元	16
4.3.1 Control Memory	17
4.3.2 CAR	17
4.3.3 CBR	18
4.4 ALU 和内部寄存器设计	19
4.4.1 ALU	20
4.4.2 MAR	21
4.4.3 MBR	22
4.4.4 PC	23

4.4.5 IR	24
4.4.6 ACC	25
4.5 外部总线设计	26
4.6 数据内存设计	27
4.7 用户面设计	28
4.7.1 按键消抖模块	28
4.7.2 七段显示器	29
4.7.3 LED 灯显示	30
4.8 时序优化	31
5 性能分析与功能验证	32
5.1 时序分析	32
5.2 资源分析	32
5.3 CPU 功能仿真	33
5.3.1 简单加法器	33
5.3.2 乘法与溢出验证	34
5.3.3 乘加运算验证	36
5.3.4 移位运算验证	37
5.3.5 逻辑运算与无条件跳转验证	38
5.4 FPGA 实物验证	39
5.4.1 开发环境与测试准备	39
5.4.2 简单加法器实物验证	39
5.4.3 乘法与溢出实物验证	40
5.4.4 乘加运算实物验证	40
5.4.5 移位运算实物验证	40
5.4.6 逻辑运算与无条件跳转实物验证	42
6 总结与展望	42
6.1 设计总结	42
6.2 已知问题	42
附录	45
A 完整设计代码	45
A.1 代码目录	45
A.2 汇编程序处理 Python 脚本	46
A.3 UART 接收与指令内存模块	49
A.4 控制单元设计	59
A.5 内部寄存器与 ALU 设计	69
A.6 数据内存设计	84
A.7 外部总线设计	85
A.8 用户面设计	87
A.9 顶层模块、测试与验证模块	98

插图目录

1	CPU 总体架构	2
2	控制单元结构示意图	6
3	控制信号示意图	7
4	CPU 模块结构图	13
5	FPGA 时序分析	32
6	FPGA 资源分析	33
7	简单加法器最后两次循环仿真结果	34
8	乘法与溢出测试例仿真结果	35
9	乘加运算验证仿真结果	36
10	移位运算验证仿真结果	37
11	逻辑运算与无条件跳转验证仿真结果	38
12	简单加法器验证	39
13	乘法与溢出验证	40
14	乘加运算验证	41
15	移位运算验证	41
16	逻辑运算与无条件跳转验证	42

表格目录

1	指令集包含指令及功能	3
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	4
4	状态寄存器列表	4
5	ALU_{op} 与执行运算的对应关系	5
6	CPU 微操作指令表	6
7	寄存器控制信号一览	8
8	CPU 控制信号表	8
9	指令内存模块外部接口	14
10	UART 模块外部接口	14
11	FIFO 模块外部接口	15
12	指令内存模块外部接口	16
13	CU_TOP 模块外部接口	16
14	Control Memory 模块外部接口	17
15	CAR 模块外部接口	18
16	CBR 模块外部接口	18
17	REG_TOP 模块外部接口	19
18	ALU 模块外部接口	20
19	MAR 模块外部接口	21
20	MBR 模块外部接口	22
21	PC 模块外部接口	23
22	IR 模块外部接口	24
23	ACC 模块外部接口	25
24	外部总线模块外部接口	27
25	数据内存模块外部接口	28
26	KEY_JITTER 模块外部接口	29
27	SEVEN_SEGMENT_DISPLAY 模块外部接口	29
28	LED_DISPLAY 模块外部接口	31

1 概述

1.1 中央处理单元 CPU 简介

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。

1.2 NEXYS 4 DDR FPGA 简介

NEXYS 4 DDR 是一款基于 Xilinx Artix-7 FPGA 的高性能开发板，为数字电路设计提供了完整的开发平台。该开发板采用 Xilinx XC7A100T-1CSG324C FPGA，拥有丰富的片上资源：15,850 个逻辑片（每片含四个 6 输入查找表和 8 个触发器）、4,860 Kb 的块 RAM、240 个 DSP 片以及内置的 ADC 等模块。系统主频最高可达 450MHz，满足各类设计的高速运算需求[1]。

在存储资源方面，NEXYS 4 DDR 配备了 128MiB 的 DDR2 SDRAM（16 位数据宽度）以及 16MB 的 Quad-SPI Flash 存储器，适合实现需要大量数据存储的应用。开发板支持通过 USB-JTAG 编程端口进行配置，也可以通过 Quad-SPI Flash 实现掉电后的程序保存。

在外部接口方面，NEXYS 4 DDR 提供了丰富的用户交互设备：

- 16 个用户可编程的 LED 灯
- 两个 4 位 7 段数码管显示器
- 5 个按钮开关和 16 个滑动开关
- 3 轴加速度计
- 温度传感器
- 12 位 1MSPS 模数转换器
- PWM 音频输出接口

通信接口包括：

- 10/100 以太网 PHY
- USB-UART 和 USB-HID 接口
- 支持 SD 卡的 Micro SD 插槽
- 多个 Pmod 接口，可扩展各类外设
- VGA 接口，支持 8 位颜色输出（512 种颜色）
- USB 主机接口，支持鼠标和键盘

此外，NEXYS 4 DDR 还配备了系统时钟发生器，提供 100MHz 的默认系统时钟。在电源管理方面，开发板具有自动监测和管理功能，能够通过 USB 或外部电源供电，并提供多种电压的稳压输出，确保系统稳定运行。

1.3 本文内容安排

本文通过设计一个基于 NEXYS 4 DDR 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：

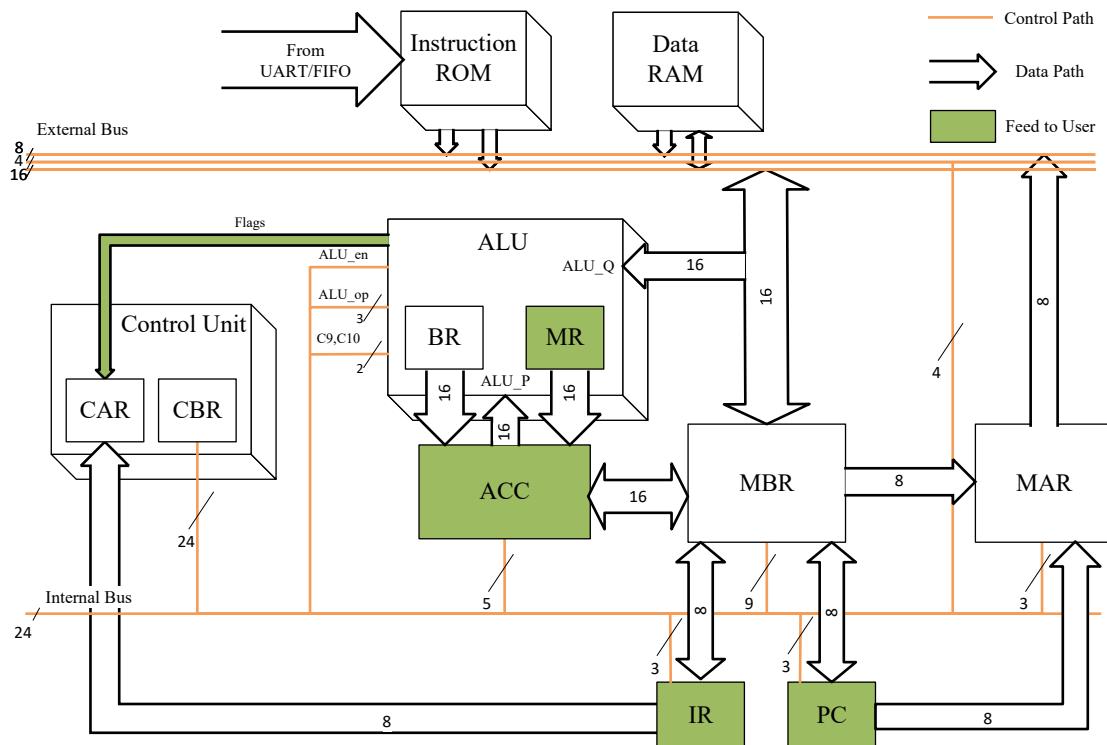
第二章将介绍 CPU 内部架构，即指令集、内部寄存器、ALU、内外总线以及控制单元设计，第三章将主要介绍 CPU 外部设备的设计，包括前端输入指令、指令传入内存、内存格式和结果显示功能，第四章是二、三章提出的设计架构的 Verilog HDL 实现，第五章是该设计的资源分析、时序分析、整体仿真结果和在 NEXYS 4 DDR FPGA 开发板上的测试结果。第六章对该设计进行了总结，并提出一些可改进的方向。另外，附录中还提供了设计的全部 Verilog 代码和项目地址。

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。

图 1: CPU 总体架构



CPU 由控制单元 (CU)，内部寄存器 (Registers) 算术逻辑单元 (ALU)，内外控制总线 (Internal/External Bus)、数据内存 (Data RAM) 和指令内存 (Instruction ROM) 组成，内部寄存器数据通路 (Data Path) 连接，同时被 CU 输出控制信号所驱动的控制通路 (Control Path) 所控制。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。另外，CPU 还通过 UART 接口接收用户输入的指令，并在用户发出请求时向 LED 和七段数码管输出图中标注绿色的寄存器的值。

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。

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（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
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¹。详细说明见第 3.1.1 节。

¹ STORE、JGZ、JMP 指令 IMMEDIATE 关键字无效，仅支持直接寻址。

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 3.2 节）。

表 4: 状态寄存器列表

寄存器	全称	行为
ZF	Zero Flag	ALU 运算结果为 0 时置 1
CF	Carry Flag	存储算术移位移出的比特
OF	Overflow Flag	16 位运算下 BR 溢出时置 1，32 位运算下 MR 溢出置 1
NF	Negative Flag	ALU 运算结果为负数时置 1
MF	Multiply Flag	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²。为简单起见，CPU 的计算全部为定点有符号数计算。

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 获得运算的两个数据 ALU_P 和 ALU_Q ，并将计算结果存入 16 位 BR 寄存器（若出现过乘法且结果大于 16 位，则可能存入 MR 寄存器），同时根据 ALU_P 、 ALU_Q 和运算结果更新 Flags 寄存器。

表 5 描述了 ALU_{op} 与执行运算的对应关系。

²自增与 PC 赋值在设计中不引入 ALU。

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，而是从 MR 借位。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。

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 (PC 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。

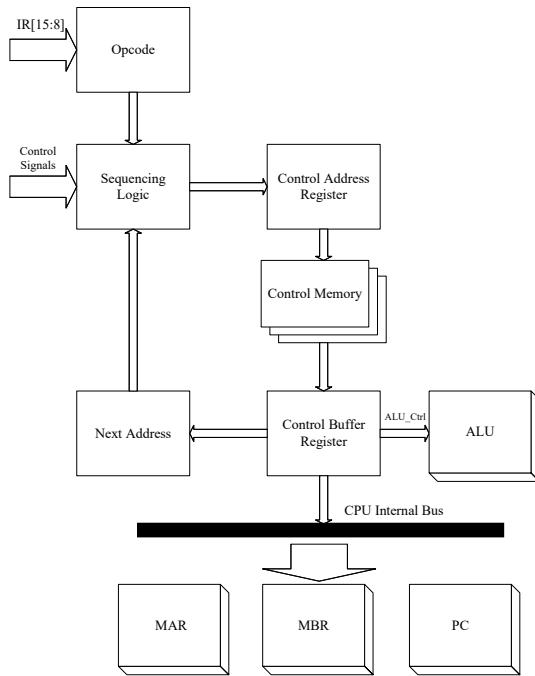


图 2: 控制单元结构示意图

- **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。

另外, 为了更好地支持用户面操作和 32 位算术运算, 在微操作指令层面添加了两条非指令集中的指令: NOP 和 STOREH。它们的作用分别是:

- NOP: 空指令, CPU 在执行该指令时不进行任何操作。该指令的作用是缓冲, 保证 CPU 在不需要执行指令时不进行取指, 不更改寄存器, 不读写内存。
- STOREH: 存储高位指令, 在 MR 为 1 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, 并在读取 MR 时清空 MR 的值。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。

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	$t_1: \text{IR} \leftarrow \text{MBR}; t_2: \text{CU} \leftarrow \text{IR}[15:8]$	
FO	阶段		$\text{MBR} \leftarrow \text{IR}[7:0]$
IND	阶段	$t_1: \text{MAR} \leftarrow \text{MBR}; t_2: \text{MBR} \leftarrow \text{Mem}[\text{MAR}]$	
NOP	/	无操作	无操作
STORE X	0001	$\text{MAR} \leftarrow \text{MBR};$ $\text{MBR} \leftarrow \text{ACC}$	$\text{Mem}[\text{MAR}] \leftarrow \text{MBR}$
STOREH X	$0001 + \text{MF} = 1$	$\text{MAR} \leftarrow \text{MBR};$ $\text{MBR} \leftarrow \text{ACC}$	$\text{Mem}[\text{MAR}] \leftarrow \text{MBR};$ $\text{ACC} \leftarrow \text{MR}, \text{MAR} + 1$

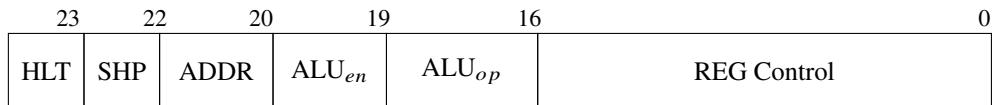
表 6: (续表) CPU 微操作指令表

指令	机器码	EX	WB
		存储高位	MBR \leftarrow MR
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): 全局停止控制字，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
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[15:0]	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	$t_1 : C_8, t_2 : C_0, C_5$	
NOP	/	无操作	无操作
STORE X	0001	C_8, C_{12}	C_0, C_{13}
STOREH X	0001 + MF 为 1	C_8, C_{12}	$C_0, C_{10}, C_{13}, \text{SHP}$
	存储高位	C_{12}	C_0, C_{13}
LOAD X	0010	无操作	C_{11}
ADD X	0011	$C_6, C_7, \text{ALU}_{en}, \text{ALU}_{op}$	C_9
SUB X	0100	$C_6, C_7, \text{ALU}_{en}, \text{ALU}_{op}$	C_9
MPY X	1000	$C_6, C_7, \text{ALU}_{en}, \text{ALU}_{op}$	C_9
JGZ X	0101	判断: ZF=0 且 NF=0?	若满足, C_3 否则 NOP

表 8: (续表) CPU 控制信号表

指令	机器码	EX	WB
JMP X	0110	无操作	C_3
HALT	0111	无操作	HLT
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，控制寄存器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。控制信号也作用于目标寄存器，防止“0”数据与“无输入”数据混淆。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 需要监测这三个控制信号的状态，并在它们为高电平时输出或在下一个周期更新自身寄存器的值。

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，他们构成了外部总线的 4 位控制总线。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。

2.7 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³。

CPU 与数据内存通过三条外部总线交互，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。

2.8 指令内存

传入 FPGA 的所有指令存于单独的指令内存 (ROM) 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，适合内存管理。指令内存的大小也为 512 Byte，每条存入内存的指令大小为 2byte，共能存入 256 条指令。指令内存初始为空，起始写入地址为 1。

³即高位存储于高地址，低位存储于低地址。

CPU 和指令内存也通过相同的三条外部总线交互，并由控制总线决定在这个周期中是否读指令内存。指令内存通过读取地址总线决定读取指令内存中的地址，通过读取数据总线决定读取指定地址中的数据。

在本设计中，指令内存的初始值通过板载 UART/FIFO 接口输入指令 ROM，详见第 3.1.1 节。

3 用户交互设计

3.1 指令写入设计

本设计采用用户编写汇编代码 → python 脚本转换为 16 位机器码 → UART/FIFO 的方式向指令内存输入指令。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。

3.1.1 用户端代码编译

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。

以下从 1 加到 100 的程序举例：

```
1 LOAD IMMEDIATE 0
2 STORE 1
3 LOAD IMMEDIATE 1
4 STORE 2
5 LOOP: LOAD 1
6 ADD 2
7 STORE 1
8 LOAD 2
9 ADD IMMEDIATE 1
10 STORE 2
11 LOAD IMMEDIATE 101
12 SUB 2
13 JGZ 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，或 STORE、JMP、JGZ 指令的 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；
- 一行内分号后的内容为注释，编译器会忽略它。

按照上述规则，示例代码将被编译为如下机器码：

```
1 8200 ; 8XXX = IMMEDIATE
2 8101
3 8201
4 8102
5 0201 ; LOOP Address: Line 05
6 0302
7 8101
```

```

8 0202
9 8301
10 8102
11 8265
12 0402
13 8505 ; Operand 05 = LOOP Address
14 0201
15 0700

```

3.1.2 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：

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

采用 UART 传输常见的 **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：

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

接收端在每位中间进行采样，保证采样数据稳定，并认为超过 0.5 秒 RX 端仍无新数据填入时（保持高电平），指令传输完成。CPU 会向用户发出提示，指示接收完成。

3.1.3 FIFO 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：

- 可确保 FIFO 中每帧数据均为 16 位数据，不会出现奇数字节的数据包；
- 指令内存地址从地址 1 开始顺序写入。
- 指令内存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）；

3.2 板载交互设备配置

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。

4 核心模块设计

本设计除 Testbench 以外的 Verilog 模块按照图 4 所示的层次结构进行组织。本章接下来将按照用户与 CPU 交互 →CPU 内部数据流 →CPU 输出反馈的逻辑路径, 也即层次结构图从下至上, 从右至左的顺序进行介绍。

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

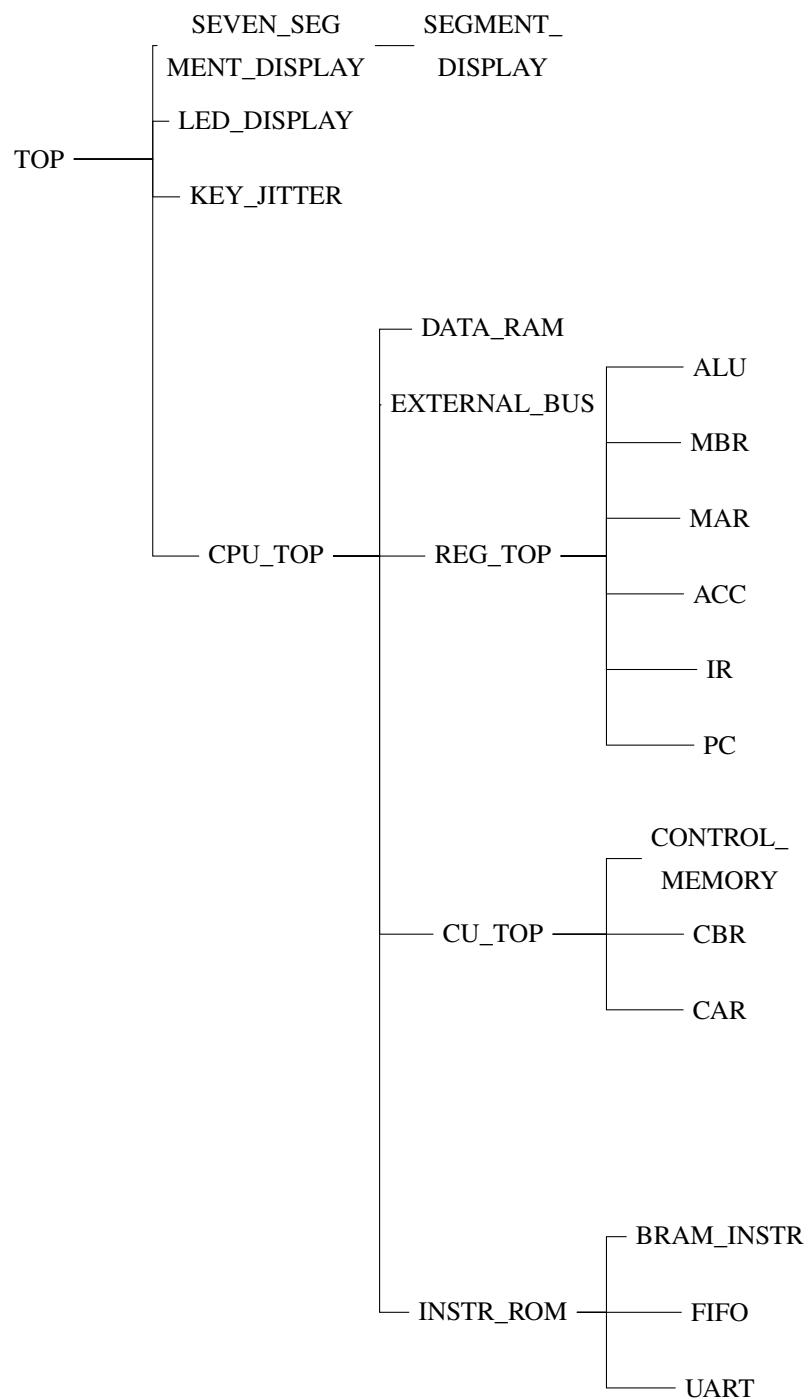
指令写入通过 Python 脚本 (附录 A.2) 完成, 其可以根据用户编写的 txt 格式汇编代码 (格式参照 3.1.1) 输出一串 UART 格式的比特流。用户可在 python 程序中设置连接 UART 端口的 USB 端口, 通过 PySerial 串口通信库进行传输, 或复制脚本的输出作为输入测试样例。

模块代码: 见附录 A.3。

功能块基本信息:

- 模块名: INSTR_ROM
- 最新更新日期: 5.6
- 是否经过测试: 是

图 4: CPU 模块结构图



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	读指令内存地址
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：5.6
- 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。

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 (分频时钟代码见 clk_divider.v)
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，否则 rx_no_data_counter 计数器在每个波特率时钟计数；
- 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，即 rx_no_data_counter 计数至 434，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 字节的指令输出给指令内存。

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，写入指令内存
o_addr_bram	输出	8	指令内存写入地址，从 0 开始自增
o_wr_en_bram	输出	1	指令内存写使能，高电平表示写入有效
o_fifo_empty	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[2]。

4.2.3 指令内存模块

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： 描述一指令内存，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，驱动读写操作
en_write	输入	1	写使能信号，高电平时允许将指令写入指令内存
en_read	输入	1	读使能信号，高电平时允许外部总线从指令内存中读取指令
i_addr_write	输入	8	要写入的指令地址
i_instr_write	输入	16	要写入的指令内容
i_addr_read	输入	8	要读取的指令地址
o_instr_read	输出	16	从指令内存中读取的指令内容

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 个时钟周期执行完成。

模块代码： 见附录 A.4。

功能块基本信息：

- 模块名：CU_TOP
- 最新更新日期：2025.5.7
- 是否经过测试：是

表 13: CU_TOP 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

(续表) CU_TOP 模块外部接口

信号名	方向	位宽	描述
ctrl_cpu_start	输入	1	CPU 启动控制信号
ctrl_step_execution	输入	1	单步执行控制信号
i_next_instr_stimulus	输入	1	下一条指令触发信号
i_ir_data	输入	8	当前指令操作码
i_flags	输入	5	状态寄存器标志位 (ZF, CF, OF, NF, MF)
o_alu_op	输出	4	ALU 运算类型控制信号
o_ctrl_mar_increment	输出	1	MAR 自增控制信号
o_ctrl_halt	输出	1	全局暂停信号
CO-C15	输出	1 × 16	16 位寄存器控制信号

4.3.1 Control Memory

模块基本信息：

- 模块名：CONTROL_MEMORY
- 最新更新日期：5.5
- 是否经过测试：是

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。

模块外部接口：

表 14: 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：5.7
- 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 15: 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、ZF、NF、MF 值综合决定具体执行指令, 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。

4.3.3 CBR

模块基本信息:

- 模块名: CBR
- 最新更新日期: 4.28
- 是否经过测试: 是

模块功能: CBR 寄存器用于存储控制信号, 接收来自 CAR 的微操作指令, 并将其输出到各个单元。CBR 寄存器在每个时钟周期内更新一次, 输出当前的控制信号。

模块外部接口:

表 16: CBR 模块外部接口

信号名	方向	位宽	描述
memory	输入	24	从 CM 传入的控制信号
ctrl_global_halt	输出	1	全局暂停信号

(续表) CBR 模块外部接口

信号名	方向	位宽	描述
<code>ctrl_mar_increment</code>	输出	1	MAR 自增信号
<code>next_addr</code>	输出	2	下一跳地址控制信号
<code>ALU_op</code>	输出	4	ALU 运算类型控制信号
<code>C0 至 C15</code>	输出	1×16	16 位寄存器控制信号

模块行为：

- 根据 CM 传入的 24 位控制信号，解析并输出以下控制信号：
 - `ctrl_global_halt`: 控制信号的第 23 位，即 HLT;
 - `ctrl_mar_increment`: 控制信号的第 22 位，即 SHP。
 - `next_addr`: 控制信号的第 21-20 位，即 ADDR;
 - `ALU_op`: 控制信号的第 19-16 位，即 ALU_{en} 和 ALU_{op} ;
 - `C(x)`: 控制信号的第 15-0 位。

4.4 ALU 和内部寄存器设计

模块代码：见附录 A.5。

功能块基本信息：

- 模块名: REG_TOP
- 最新更新日期: 2025.5.7
- 是否经过测试: 是

功能块外部接口：

表 17: REG_TOP 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>ctrl_cpu_start</code>	输入	1	CPU 启动控制信号
<code>i_ALU_op</code>	输入	4	ALU 运算类型控制信号
<code>i_ctrl_halt</code>	输入	1	全局暂停信号
<code>i_ctrl_mar_increment</code>	输入	1	MAR 自增控制信号
<code>i_memory_data</code>	输入	16	从数据总线传入的数据
<code>o_memory_addr</code>	输出	8	输出到地址总线的地址
<code>o_memory_data</code>	输出	16	输出到数据总线的数据
<code>o_ir_cu</code>	输出	8	输出到 CU 的操作码
<code>o_flags</code>	输出	5	状态寄存器标志位 (ZF, CF, OF, NF, MF)
<code>o_ACC_user</code>	输出	16	输出到用户接口的 ACC 寄存器数据
<code>o_MR_user</code>	输出	16	输出到用户接口的 MR 寄存器数据
<code>o_PC_user</code>	输出	8	输出到用户接口的 PC 寄存器数据

(续表) REG_TOP 模块外部接口

信号名	方向	位宽	描述
o_IR_user	输出	8	输出到用户接口的 IR 寄存器数据

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：2025.5.7
- 是否经过测试：是

模块功能：在控制信号 `ctrl_alu_en` 有效时，根据 `ctrl_alu_op` 执行指定的运算，运算结果存储在 BR (低 16 位) 和 MR (高 16 位) 寄存器中，并更新状态寄存器 (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 18: ALU 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_acc_alu_p	输入	16	ALU 的第一个操作数
i_acc_alu_q	输入	16	ALU 的第二个操作数
ctrl_alu_op	输入	3	运算类型控制信号
ctrl_alu_en	输入	1	ALU 使能信号
o_BR	输出	16	运算结果的低 16 位
o_MR	输出	16	运算结果的高 16 位 (仅乘法有效)
o_flags	输出	5	状态寄存器 (ZF, CF, OF, NF, MF)
i_user_sample	输入	1	用户采样信号
o_MR_user	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: 算术运算溢出时置 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`。
- 当 `C_10` 信号有效时, MR 在下一个周期清零, 代表高位结果已被读取;
- 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: 2025.4.29
- 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 19: 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
<code>C2</code>	输入	1	控制信号, 允许从 PC 读取地址
<code>C8</code>	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：2025.4.28
- 是否经过测试：是

模块功能： 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。MBR 模块支持以下功能：

- 在控制信号 C1 有效时，从 PC 寄存器读取地址；
- 在控制信号 C3 有效时，将数据传输到 PC 寄存器；
- 在控制信号 C4 有效时，将数据传输到 IR 寄存器；
- 在控制信号 C5 有效时，从数据总线读取数据；
- 在控制信号 C6 有效时，将数据传输到 ALU 的 ALU_Q 输入端；
- 在控制信号 C8 有效时，将数据传输到 MAR 寄存器；
- 在控制信号 C11 有效时，将数据传输到 ACC 寄存器；
- 在控制信号 C12 有效时，从 ACC 寄存器读取数据；
- 在控制信号 C15 有效时，从 IR 寄存器读取立即操作数；

模块外部接口：

表 20: 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_pc_mbr	输入	8	从 PC 传入的地址
i_ir_mbr	输入	8	从 IR 传入的操作数
i_data_bus_mbr	输入	16	从数据总线传入的数据
i_acc_mbr	输入	16	从 ACC 传入的数据
o_mbr_data_bus	输出	16	输出到数据总线的数据
o_mbr_pc	输出	8	输出到 PC 寄存器的地址
o_mbr_ir	输出	16	输出到 IR 寄存器的数据
o_mbr_mar	输出	8	输出到 MAR 寄存器的地址
o_mbr_acc	输出	16	输出到 ACC 寄存器的数据
o_mbr_alu_q	输出	16	输出到 ALU 的 ALU_Q 输入端的数据
C1	输入	1	控制信号，允许从 PC 读取地址
C3	输入	1	控制信号，允许将数据传输到 PC
C4	输入	1	控制信号，允许将数据传输到 IR
C5	输入	1	控制信号，允许从数据总线读取数据

(续表) MBR 模块外部接口

信号名	方向	位宽	描述
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
C12	输入	1	控制信号, 允许从 ACC 读取数据
C15	输入	1	控制信号, 允许从 IR 读取操作数

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_MBR 读取数据;
- 当 C15 信号有效时, MBR 从 i_IR_MBR 读取操作数;
- 当 C1 信号有效时, MBR 从 i_PC_MBR 读取地址;
- 当 C12 信号有效时, MBR 从 i_ACC_MBR 读取数据;
- 当 C11 信号有效时, MBR 将数据输出到 o_MBR_ACC;
- 当 C6 信号有效时, MBR 将数据输出到 o_MBR_ALU_Q;
- 当 C8 信号有效时, MBR 将数据输出到 o_MBR_MAR;
- 当 C3 信号有效时, MBR 将数据输出到 o_MBR_PC;
- 当 C4 信号有效时, MBR 将数据输出到 o_MBR_IR;
- 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: 2025.4.30
- 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 21: 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

(续表) PC 模块外部接口

信号名	方向	位宽	描述
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: 2025.4.30
- 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 22: 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

(续表) IR 模块外部接口

信号名	方向	位宽	描述
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：2025.4.30
- 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 23: 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

(续表) ACC 模块外部接口

信号名	方向	位宽	描述
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 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。

模块代码: 见附录 A.7。

模块基本信息:

- 模块名: EXTERNAL_BUS
- 最新更新日期: 2025.5.5
- 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)。⁴

模块外部接口:

⁴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 24: 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 的值;
 - 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 的值。
- 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。

4.6 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。

模块代码: 见附录 A.6。

模块基本信息:

- 模块名: DATA_RAM
- 最新更新日期: 2025.5.5
- 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 25: 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。

4.7 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。

模块代码： 见附录 A.8。

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 26: 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：2025.5.6
- 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：2025.5.5
- 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 seg_valid 用于更新七段显示器的显示（轮询周期为 2.4ms）；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 27: SEVEN_SEGMENT_DISPLAY 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

(续表) SEVEN_SEGMENT_DISPLAY 模块外部接口

信号名	方向	位宽	描述
i_RST_N	输入	1	全局复位信号
switch_start_cpu	输入	1	CPU 启动控制开关
button_check_instruction	输入	1	连向 BTNR
button_check_flags	输入	1	连向 BTND
button_check_result	输入	1	连向 BTNL
light_instr_transmit_done	输入	1	指令传输完成信号
segment_current_PC	输入	8	当前程序计数器值
segment_current_Opcode	输入	8	当前指令操作码
segment_flags	输入	5	标志位状态 (ZF,CF,OF,NF,MF)
segment_result_high	输入	16	运算结果高 16 位
segment_result_low	输入	16	运算结果低 16 位
segment_max_instr_addr	输入	8	指令内存最大地址
o_seg_valid	输出	8	七段显示器使能信号
o_seg_value	输出	8	七段显示器编码值

模块行为：

- 当 i_RST_N 为低电平时，所有输出信号复位为 0；
- 当 button_check_instruction 信号有效时，前四个数码管显示下一条指令的地址，后四个显示当前指令操作码；
- 当 button_check_flags 信号有效时，显示标志位状态（从高到低依次为 ZF、CF、OF、NF、MF）；
- 当 button_check_result 信号有效时，显示运算结果；
- 当指令传输完成信号有效，且 CPU 未启动时，更新目前的最大指令地址（传入指令数量）。

模块参数：

- SCAN_INTERVAL：轮询信号切换的时间间隔，单位为时钟周期。默认值为 30000，对应 2.4ms 的轮询周期。此参数可在最上层模块自定义。

4.7.3 LED 灯显示

模块基本信息：

- 模块名：LED_DISPLAY
- 最新更新日期：2025.5.5
- 是否经过测试：是

模块功能： LED 灯显示模块用于显示 CPU 的运行模式和运行状态。它通过控制板载两个 LED 灯的颜色来指示不同的状态。

模块外部接口：

表 28: LED_DISPLAY 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_instr_transmit_done	输入	1	指令传输完成信号
i_halt	输入	1	CPU 停止信号
i_step_execution	输入	1	连接 BTNC
i_start_cpu	输入	1	CPU 启动控制信号
RGB1_RED	输出	1	左侧 LED 红色控制信号
RGB1_BLUE	输出	1	左侧 LED 蓝色控制信号
RGB2_RED	输出	1	右侧 LED 红色控制信号
RGB2_BLUE	输出	1	右侧 LED 蓝色控制信号
RGB2_GREEN	输出	1	右侧 LED 绿色控制信号

模块行为：

- 当 i_RST_N 为低电平时，模块复位，所有 LED 灯熄灭，状态机进入 STATE_IDLE 状态；
- 在 STATE_IDLE 状态下：
 - 所有 LED 灯熄灭；
 - 若 i_instr_transmit_done 信号有效，状态机切换到 STATE_GREEN；
- 在 STATE_GREEN 状态下：
 - 右侧 LED 灯亮绿色；
 - 若 i_start_cpu 信号有效，状态机切换到 STATE_BLUE；
- 在 STATE_BLUE 状态下：
 - 右侧 LED 灯亮蓝色；
 - 若 i_halt 信号有效，状态机切换到 STATE_RED；
- 在 STATE_RED 状态下：
 - 右侧 LED 灯亮红色；
- 左侧 LED 灯的颜色根据 i_step_execution 信号决定：
 - 若 i_step_execution 信号有效，左侧 LED 灯亮红色；
 - 否则，左侧 LED 灯亮蓝色；

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 阶段的两个时钟沿来更新。

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 0.05ns 提升至 0.36ns，时序余量提升了 600%。

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 5 所示。

图 5: FPGA 时序分析

Design Timing Summary			
Setup	Hold	Pulse Width	
Worst Negative Slack (WNS): 0.364 ns	Worst Hold Slack (WHS): 0.061 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.

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。

5.2 资源分析

使用 Vivado 2024.2 进行 FPGA 资源分析，结果如图 6 所示。

在 FPGA 资源利用方面，核心模块主要集中在 `control_unit` (265 个 LUT 和 13 个寄存器) 和 `internal_registers` (220 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。

图 6: 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		1043	555	48	16	333	885	158	1	30	1
cpu (TOP_CPU)		740	303	44	16	256	582	158	1	0	0
control_unit (CU_TOP)		265	13	12	0	118	265	0	0	0	0
data_ram (DATA_RAM)		72	0	32	16	23	8	64	0	0	0
external_bus (EXTERNAL_BUS)		0	1	0	0	1	0	0	0	0	0
instruction_rom (INSTR_ROM)		184	135	0	0	76	90	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)		92	8	0	0	29	4	88	0	0	0
instr_load_fifo (FIFO)		30	64	0	0	23	24	6	0	0	0
instr_load_uart (UART)		32	51	0	0	19	32	0	0	0	0
internal_registers (REG_TOP)		220	101	0	0	117	220	0	1	0	0
reg_ACC (ACC)		121	16	0	0	62	121	0	0	0	0
reg_ALU (ALU)		55	37	0	0	42	55	0	1	0	0
reg_IR (IR)		6	16	0	0	8	6	0	0	0	0
reg_MAR (MAR)		11	8	0	0	5	11	0	0	0	0
reg_MBR (MBR)		41	16	0	0	27	41	0	0	0	0
reg_PC (PC)		10	8	0	0	3	10	0	0	0	0
ins_button_check_flags (KEY_JITTER_parameterized0)		27	25	0	0	10	27	0	0	0	0
ins_button_check_instruction (KEY_JITTER_parameterized0_0)		28	25	0	0	12	28	0	0	0	0
ins_button_check_result (KEY_JITTER_parameterized0_1)		27	25	0	0	8	27	0	0	0	0
ins_button_next_instr (KEY_JITTER_parameterized0_2)		27	25	0	0	9	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)		83	24	0	0	30	83	0	0	0	0
ins_switch_step_execution (KEY_JITTER_4)		28	24	0	0	10	28	0	0	0	0
led_display (LED_DISPLAY)		1	9	0	0	4	1	0	0	0	0
segment_display (SEVEN_SEGMENT_DISPLAY)		55	70	4	0	27	55	0	0	0	0
seven_segment (SEVEN_SEGMENT)		31	35	4	0	18	31	0	0	0	0

5.3 CPU 功能仿真

使用开源轻量化 HDL 仿真工具 [iverilog](#)[Github] 和开源轻量化 HDL 波形查看工具 [GTKWave](#)[Github] 进行 CPU 功能仿真。它具有以下特性，适用于简单 FPGA 设计的功能仿真。

- 轻量化，波形查看仅依赖命令行操作，且不需要安装额外的 GUI 工具；
- 全面的信号可视化功能，允许跟踪所有 CPU 寄存器；
- 层次化的信号分组，便于导航复杂的模块；
- 搜索功能，可以定位特定的信号变化；
- 测量工具，用于精确的时序分析；
- 多种信号格式（二进制、十六进制、十进制）便于数据检查。

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 1
LOAD IMMEDIATE 1
STORE 2
LOOP: LOAD 1
ADD 2
STORE 1
LOAD 2
ADD IMMEDIATE 1
STORE 2
LOAD IMMEDIATE 101 ; faster
SUB 2
JGZ LOOP
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.2。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。

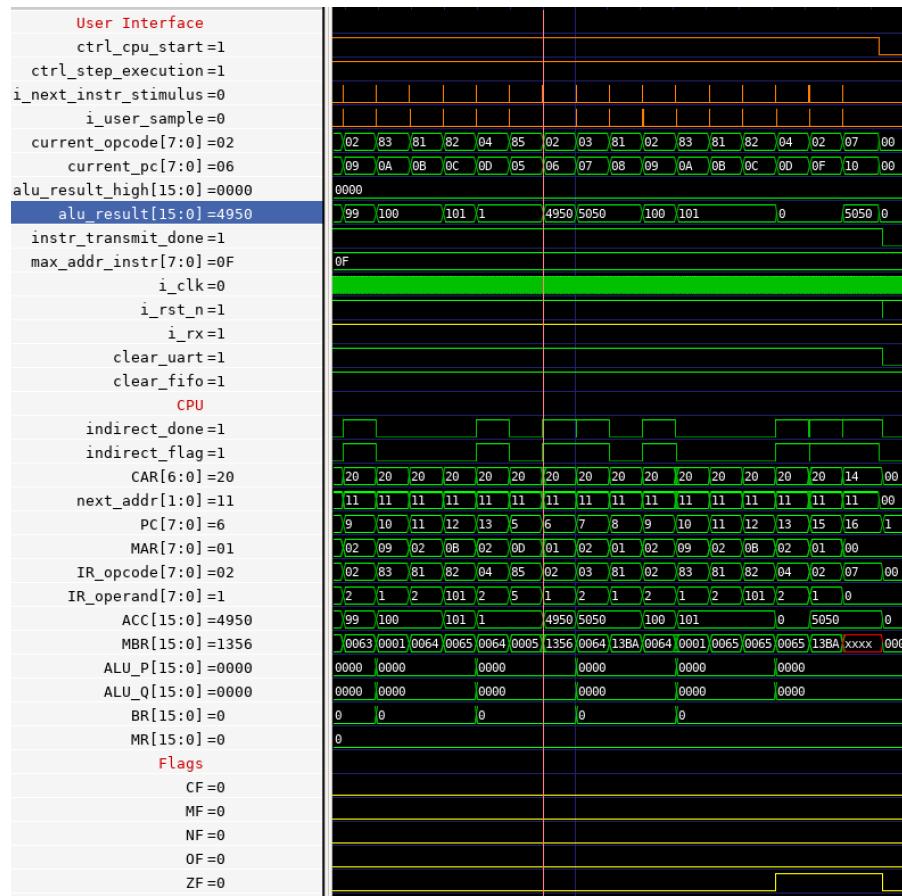
这个测试样例可以测试 CPU 的条件跳转指令功能、加减法功能和 ZF 功能的正确性。

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 7 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。

图 7：简单加法器最后两次循环仿真结果



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：

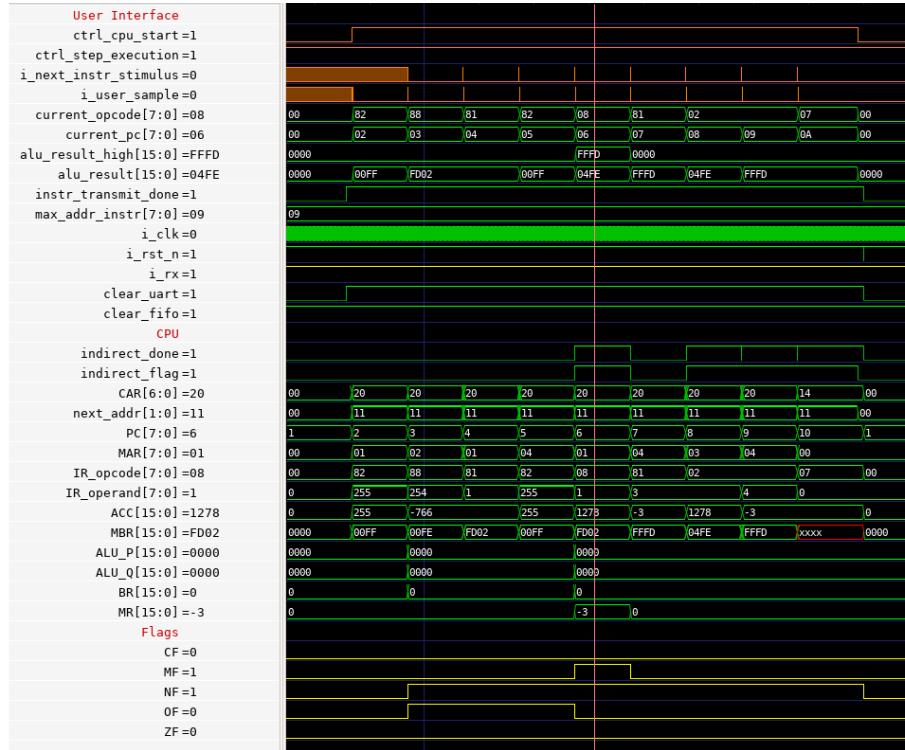
```

1 LOAD IMMEDIATE 255
2 MPY IMMEDIATE 254
3 STORE 1
4 LOAD IMMEDIATE 255
5 MPY 1
6 STORE 3
7 LOAD 3
8 LOAD 4
9 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 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 8: 乘法与溢出测试例仿真结果



仿真结果： 如图 8 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 乘加运算验证

仿真内容：计算 $255 \times 254 \times 255 - 1400$ 的值。

由第5.3.2节可知， $255 \times 254 \times 255 = -195330$ ，因此计算结果应为 $-195330 - 1400 = -196730$ ，转换为16进制补码可得结果为 FFFCFF86。

激励设置：用户首先编写源程序如下：

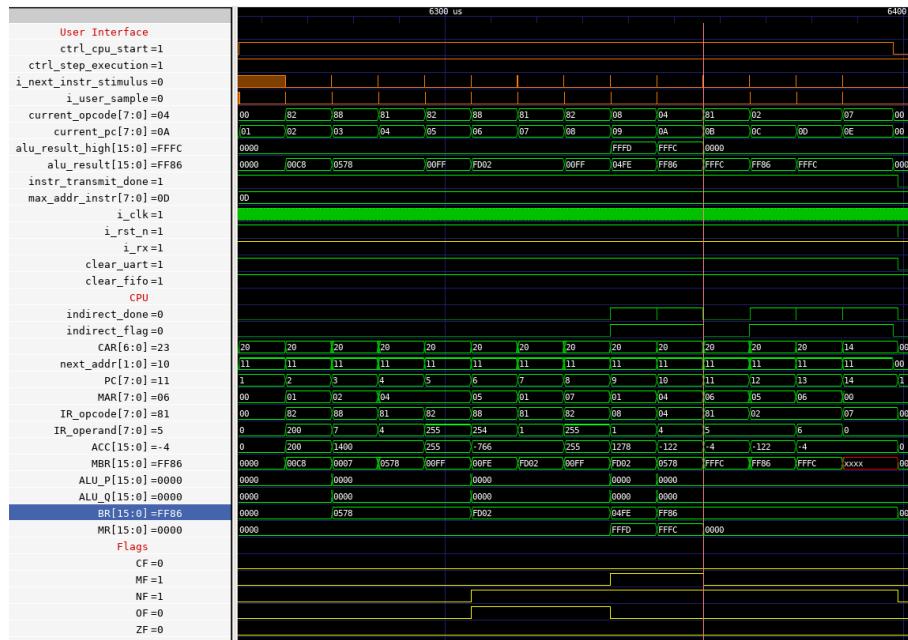
```

1 LOAD IMMEDIATE 200
2 MPY IMMEDIATE 7
3 STORE 4
4 LOAD IMMEDIATE 255
5 MPY IMMEDIATE 254
6 STORE 1
7 LOAD IMMEDIATE 255
8 MPY 1
9 SUB 4
10 STORE 5 ; this is 32-bit
11 LOAD 5
12 LOAD 6
13 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对32位 $m \times x + n$ 型运算的支持。

图9：乘加运算验证仿真结果



仿真结果：如图9所示，MR被正确借位，表明减法32位运算成功被触发；运算结果的低位、高位结果正确地存储在内存地址5和6，表明STOREH成功触发并正确执行。使用LOAD指令查看内存地址5、6的值，可知：

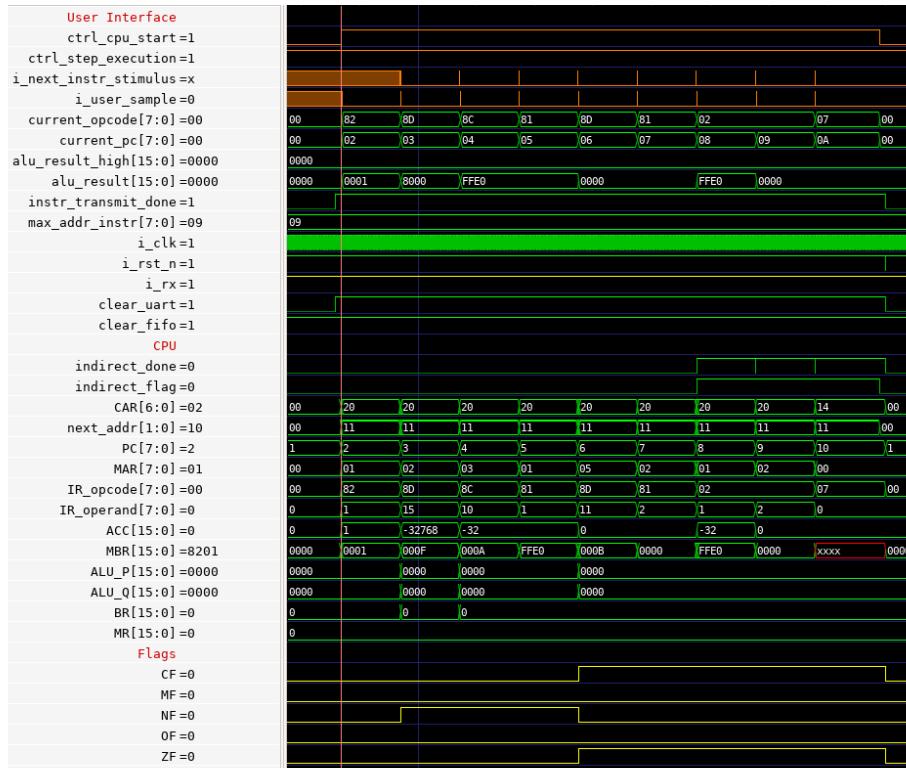
- 第一次乘法运算结果高 16 位为 FFFD，低 16 位为 04FE，MF、NF 置 1，表明乘法使用高 16 位且结果为负；
- 第二次乘法运算结果高 16 位为 FFFC，低 16 位为 FF86。由于 MR 最高位为 1，NF、MF 置 1；未发生 32 位溢出，故 OF 为 0；
- 存入内存后，MR 归零，MF 随之归零。

5.3.4 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;

图 10: 移位运算验证仿真结果



激励设置： 用户首先编写源程序如下：

```

1 LOAD IMMEDIATE 1
2 SHIFTL IMMEDIATE 15
3 SHIFTR IMMEDIATE 10
4 STORE 1
5 SHIFTL IMMEDIATE 11
6 STORE 2
7 LOAD 1
8 LOAD 2
9 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 10 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。

5.3.5 逻辑运算与无条件跳转验证

仿真内容：依次完成以下操作：

- $99 - 100 = -1(0xFFFF)$;
- $0xFFFF \& 0x000C = 0x000C$;
- $0x000C | 0x0003 = 0x000F$;
- $\neg 0x000F = 0xFFFF0$;
- 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：

```

1 LOAD IMMEDIATE 100
2 STORE 1
3 LOAD IMMEDIATE 99
4 SUB 1
5 AND IMMEDIATE 12 ; 1100
6 OR IMMEDIATE 3 ; 0011
7 STORE 2
8 NOT 2
9 JMP 11
10 ADD IMMEDIATE 1
11 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 11 所示，正确的逻辑运算的结果依次显示在用户接口中，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 11：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。

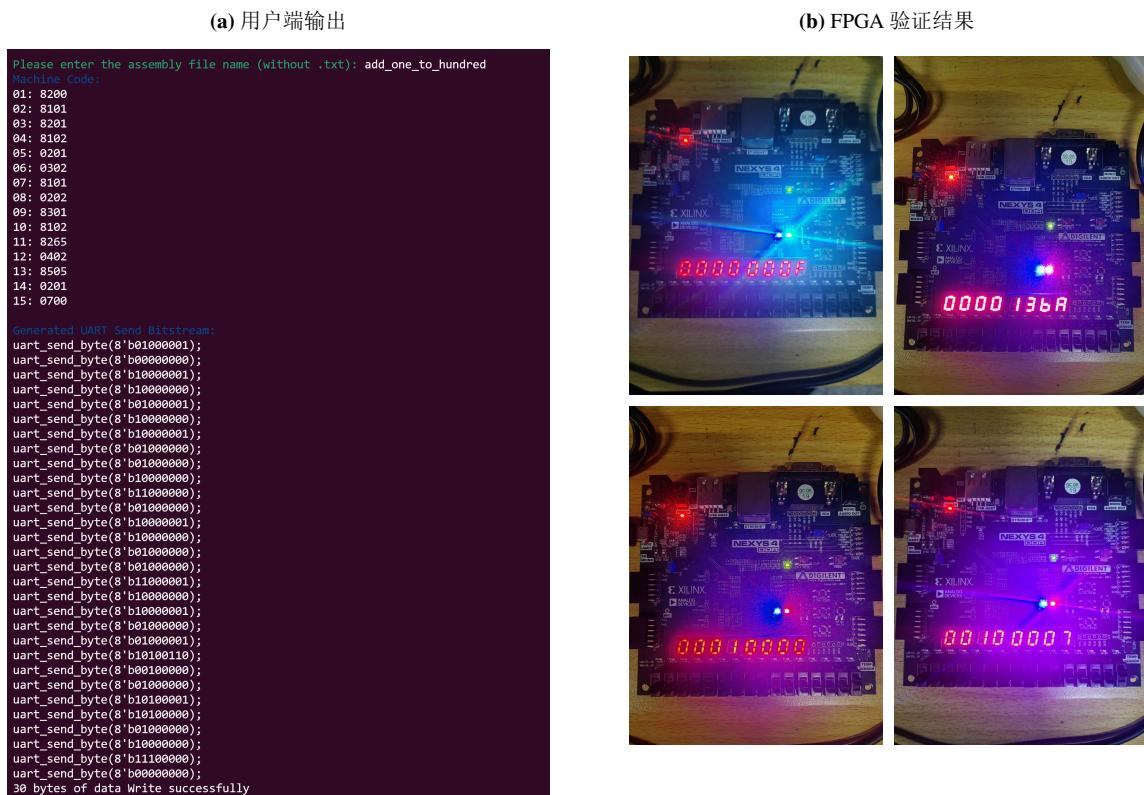
- 开发工具: Vivado 2024.2、Python 3.12.3、PySerial 3.5
 - 测试环境: Ubuntu 24.04

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

首先如图(a)输入待执行汇编代码，指令通过串口输入到FPGA中，用户端显示输出的指令机器码和Testbench模板（便于仿真测试）。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12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 12: 简单加法器验证

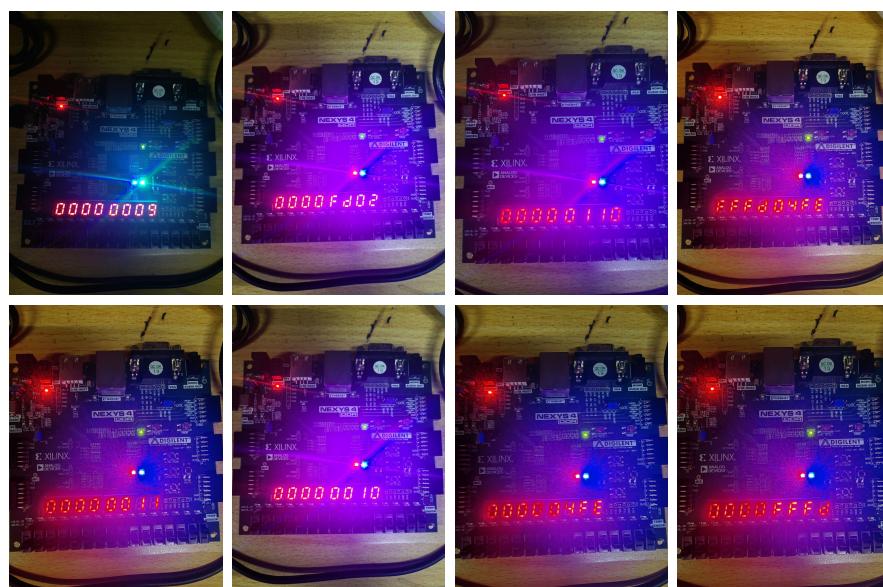


5.4.3 乘法与溢出实物验证

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 13 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。

图 13：乘法与溢出验证



5.4.4 乘加运算实物验证

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。此轮仿真共计 13 条指令，全流程如图 14 所示。加载指令完成后指令内存的最大地址为 0x0D，即共存入 13 条指令。将最右侧开关打开后，CPU 开始单步调试执行指令。

与乘法测试不同，本测试首先计算出减数 200×7 的结果为 0x0578（1400），再进行乘法测试得到被减数 0xFFFFD04FE，最后执行减法。从图中可以看到，减法结果与仿真结果一致并正确，显示为 0xFFFFCFF86，执行 LOAD 5 指令后，七段数码管成功显示乘法结果低位 0xFF86；执行 LOAD 6 指令后，显示高位 0xFFFF，验证了 CPU 正确运算存储 32 位乘加运算结果、处理溢出的能力。

5.4.5 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 15 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 14: 乘加运算验证

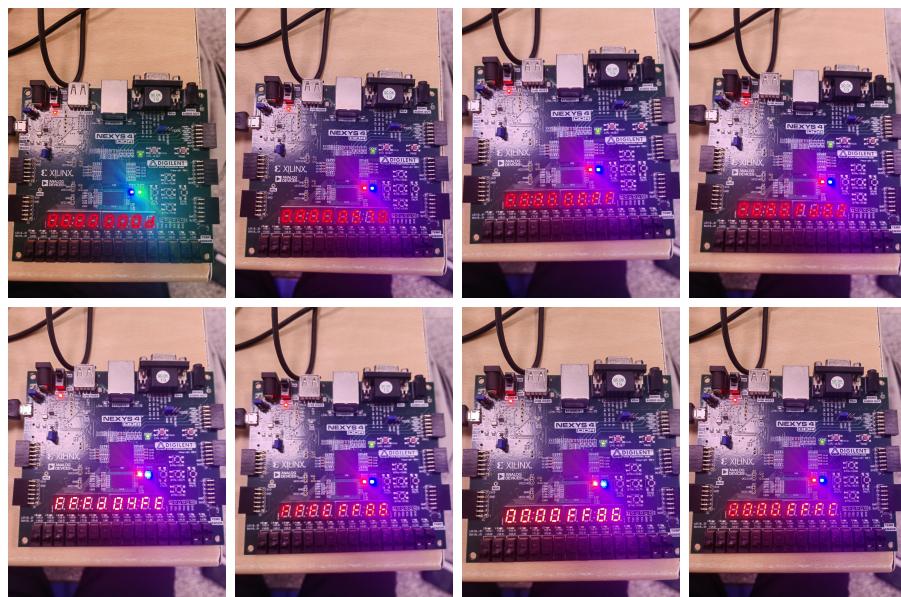
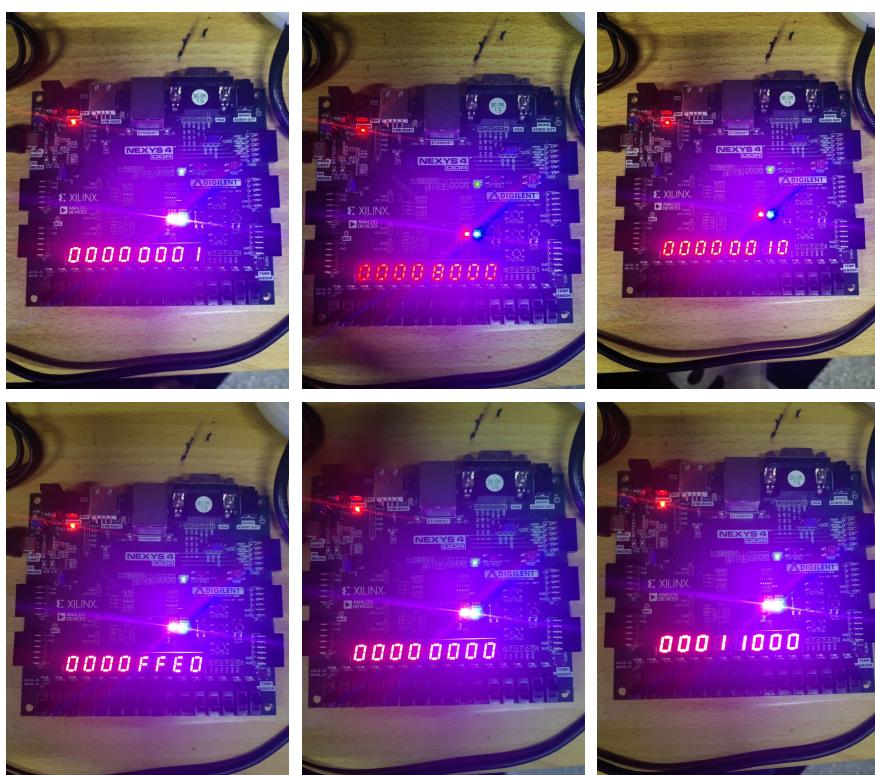


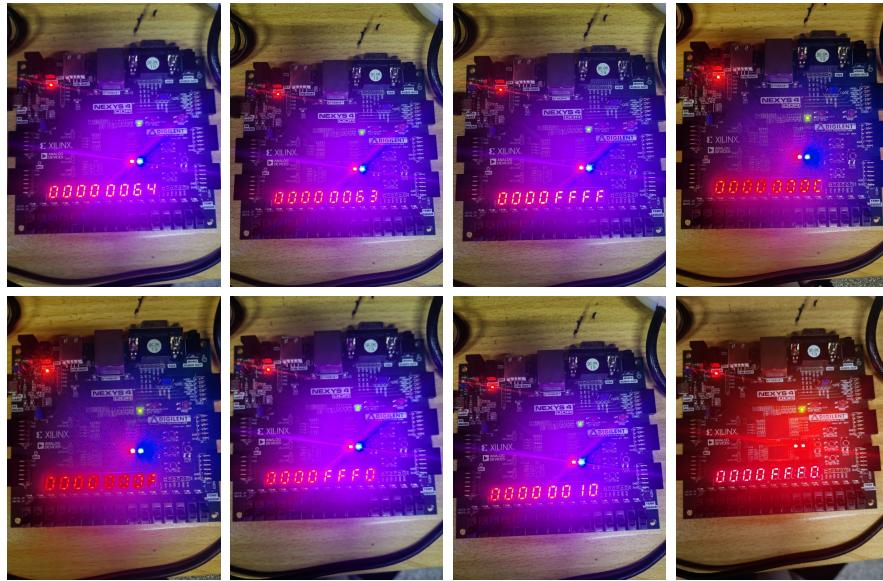
图 15: 移位运算验证



5.4.6 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 16 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 16：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：

1. 仅支持 32 位算术运算，不支持 32 位逻辑运算和移位运算，但可以通过算术运算间接实现这两种运算。
2. CPU 没有异常处理机制，若发生异常（如未传入汇编代码或汇编代码无 HALT 指令时启动 CPU），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。
3. 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 是基于单周期架构设计的，如果要实现多周期或流水线设计，需要对现有设计进行较大改动，故在本设计中不再实现。

4. 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。

参考文献

- [1] Digilent. Nexys 4 DDR FPGA Board Reference Manual[A/OL]. Digilent Inc. 2016. <https://digilent.com/reference/programmable-logic/nexys-4-ddr/reference-manual>.
- [2] 菜鸟教程. Verilog FIFO 设计[EB/OL]. 2020. <https://www.runoob.com/w3cnote/verilog2-fifo.html>.

A 完整设计代码

该部分以数据流向和从内向外的顺序，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 代码目录

1	write_bistream.py	46
2	uart.v	49
3	fifo.v	52
4	bram_instr.v	56
5	clk_divider.v	56
6	top_instr_rom.v	57
7	cu_control_memory.v	59
8	cu_control_address_register.v	62
9	cu_control_buffer_register.v	66
10	cu_top.v	67
11	alu.v	69
12	acc.v	73
13	mar.v	74
14	mbr.v	76
15	pc.v	78
16	ir.v	79
17	reg_top.v	80
18	top_data_ram.v	84
19	external_bus.v	85
20	key_jitter.v	87
21	top_seven_segment.v	89
22	seven_segment.v	92
23	led_display.v	95
24	cpu_top.v	98
25	top.v	103
26	CPU Testbench	107
27	含用户面 Testbench	113
28	开发板引脚约束	120

A.2 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     addr = 1
43     for line in lines:
44         line = line.split(';')[0].strip()
45         if not line:
```

```

        continue
47 if ':' in line:
48     parts = line.split(':', 1)
49     line = parts[1].strip()
50     if not line:
51         continue
52
53 tokens = line.split()
54 if not tokens:
55     continue
56
57 instr = tokens[0]
58 immediate = False
59
60 if instr in ["HALT"] : # No operand, fill with 0
61     opcode = MEMONICS[instr]
62     operand = 0x00
63 else:
64     if len(tokens) < 2:
65         raise ValueError(f"Missing operand in line: {line}")
66     # Force MSB=1 for STORE instruction
67     if instr in ["STORE", "JGZ", "JMP"]:
68         immediate = True
69         opcode = MEMONICS[instr] | 0x80 # MSB = 1
70     elif tokens[1] == "IMMEDIATE":
71         immediate = True
72         operand_str = tokens[2]
73         opcode = MEMONICS[instr] | 0x80 # MSB = 1
74     else:
75         operand_str = tokens[1]
76         opcode = MEMONICS[instr] # MSB = 0
77
78     if operand_str.isdigit():
79         operand = int(operand_str)
80     elif operand_str in labels:
81         operand = labels[operand_str]
82     else:
83         try:
84             operand = int(operand_str, 0) # Support 0x form operand
85         except:
86             raise ValueError(f"Unknown operand: {operand_str}")
87
88     if operand < 0 or operand > 255:
89         raise ValueError(f"Operand out of 8-bit range: {operand}")
90
91     machine_code.append((opcode << 8) | operand)
92     addr += 1

```

```

93
94     return machine_code
95
96
97 def assemble_to_bytes(code: list[int]) -> bytearray:
98     result = bytearray()
99     for word in code:
100         result.append((word >> 8) & 0xFF) # opcode
101         result.append(word & 0xFF)      # operand
102     return result
103
104
105 def send_to_serial(bitstream:bytearray) -> None:
106     # FPGA Config: Baud rate = 115200, 8N1 Transmission
107     write_port = '/dev/ttyUSB1' # Default port
108     ser = serial.Serial(
109         port= write_port,
110         baudrate= 115200,
111         timeout=1,
112         bytesize=8,
113         parity= "N",
114         stopbits=1
115     )
116
117     # 向 FPGA 发送数据
118     number_of_bytes = ser.write(bitstream) # 将字符串转换为字节并发送
119     print(f"{number_of_bytes} bytes of data Write successfully")
120
121     # 关闭串口
122     ser.close()
123
124
125 def main(filename:str):
126     os.chdir("./designs/input_src")
127     file_path = f"{filename}.txt"
128     if not os.path.exists(file_path):
129         raise ValueError("Cannot find such file.")
130     else:
131         with open(file_path, 'r') as file:
132             lines = file.readlines()
133
134
135     machine_words = parse_assembly(lines)
136     binary = assemble_to_bytes(machine_words)
137
138     # For Reference:
139     print("\033[1;34mMachine Code:\033[0m")
     for i, word in enumerate(machine_words):
         print(f"\033[1;32m{i+1:02}\033[0m: {word:04X}\033[0m")

```

```

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

```

Listing 1: write_bistream.py

A.3 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。

```

1 `timescale 1ns / 1ps
2 // 5.5 Update: adapt 8N1 format
3 // 5.6 Update: always module driven by i_clk rather than i_clk_uart
4 module UART (
5     i_clk,
6     i_clk_uart,
7     i_rst_n,
8     i_rx,
9     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 // i_clk_rd is the system clock
57 always @(posedge i_clk_rd) begin
58   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
102 reg [7:0] data_buffer;
103 reg      byte_flag; // flag of the first UART byte is read
104
105 always @ (posedge i_clk_rd or negedge i_rst_n) begin
106     if (!i_rst_n) begin
107         rd_ptr_bin <= 0;
108         rd_ptr_gray <= 0;
109         byte_flag  <= 0;
110         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

```

module CLK_DIVIDER #(
    parameter DIVIDE_BY = 2 // 2 or 868
)();
    input wire i_clk,

```

```

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

```

Listing 5: clk_divider.v

```

`timescale 1ns / 1ps

module INSTR_ROM (
    i_clk,
    i_RST_N,
    i_RX,
    en_read,
    i_addr_read,
    o_instr_read,
    o_instr_transmit_done,
    o_max_addr
);

```

```

1
2
3
4
5
6
7
8
9
10
11
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.4 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 * Critical path: STOREH with indirect for 10 clock cycles
10 */
11 `timescale 1ns / 1ps
12
13 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; // FO, 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
67      7'h0F:
68          control_word = 24'b00_10_1010_00000000_11000000; // EX, 6 7
69
70      7'h10:
71          control_word = 24'b00_11_0010_00000010_00000000; // WB, 9
72
73          // JGZ & JMP
74
75      7'h11:
76          control_word = 24'b00_10_0000_00000000_00000000; // EX
77
78      7'h12:
79          control_word = 24'b00_11_0000_00000000_00001000; // WB, 3
80
81
82          // HALT
83          // Stop and reset control word to IF
84
85      7'h13:
86          control_word = 24'b00_10_0000_00000000_00000000; // EX
87
88      7'h14:
89          control_word = 24'b10_11_0000_00000000_00000000; // WB, HALT
90
91
92          // AND
93
94      7'h15:
95          control_word = 24'b00_10_1011_00000000_11000000; // EX, 6 7
96
97      7'h16:
98          control_word = 24'b00_11_0011_00000010_00000000; // WB, 9
99
100
101          // OR
102
103      7'h17:
104          control_word = 24'b00_10_1100_00000000_11000000; // EX, 6 7
105
106      7'h18:
107          control_word = 24'b00_11_0100_00000010_00000000; // WB, 9
108
109
110          // NOT
111
112      7'h19:
113          control_word = 24'b00_10_1101_00000000_01000000; // EX, 6
114
115      7'h1A:
116          control_word = 24'b00_11_0101_00000010_00000000; // WB, 9
117
118
119          // SHIFTR
120
121      7'h1B:
122          control_word = 24'b00_10_1110_00000000_11000000; // EX, 6 7
123
124      7'h1C:
125          control_word = 24'b00_11_0110_00000010_00000000; // WB, 9
126
127
128          // SHIFTL

```

```

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

```

Listing 7: cu_control_memory.v

```

1 `timescale 1ns / 1ps
2
3 // Sequencing Logic & CAR
4 /* Sequencing Logic of CAR
5 * 10 self increment
6 * 11 back to 0
7 * 01 jump
8 * 00 nothing
9 */
10 module CAR (

```

```

11     ctrl_cpu_start,
12     ctrl_step_execution,
13     i_ctrl_halt,
14     i_next_instr_stimulus,
15     i_clk,
16     i_rst_n,
17     i_control_word_car,
18     i_ir_data,
19     i_ctrl_ZF,
20     i_ctrl_NF,
21     i_ctrl_MF,
22     o_car_data
23   );
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
39 reg ctrl_cpu_start_reg;
40
41 always @ (posedge i_clk) begin
42   ctrl_cpu_start_reg <= ctrl_cpu_start;
43 end
44
45
46 reg [4:0] ir_data;
47 reg [6:0] CAR;
48 reg indirect_done;
49 wire indirect_flag = ctrl_cpu_start ? (!ir_data[4] && (ir_data[3:0] != 4'b0)) : 1'b0;
50
51
52 always @ (posedge i_clk or negedge i_rst_n) begin
53   if (!i_rst_n) begin
54     ir_data <= 5'b0;
55   end
56   else begin
57     if (i_ir_data[3:0] != 3'b0) begin
58       ir_data <= i_ir_data[4:0];
59     end
60   end
61 end
62
63 end

```

```

58
59 // always @(*) begin
60 //     if(i_ir_data[3:0] != 3'b0) begin
61 //         ir_data = i_ir_data[4:0];
62 //     end
63 //     else begin
64 //         ir_data = ir_data;
65 //     end
66 // end
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
107           4'd7:
108               CAR <= 7'h13; // HALT
109
110           4'd8:
111               CAR <= 7'h0F; // MPY
112
113           4'd9:
114               CAR <= 7'h15; // AND
115
116           4'd10:
117               CAR <= 7'h17; // OR
118
119           4'd11:
120               CAR <= 7'h19; // NOT
121
122           4'd12:
123               CAR <= 7'h1B; // SHIFTR
124
125           4'd13:
126               CAR <= 7'h1D; // SHIFTL
127
128           default:
129               CAR <= 7'h00;
130
131           endcase
132
133       end
134
135   2'b10: begin
136       CAR <= CAR + 1; // Next Micro-instruction
137
138   end
139
140   2'b11: begin
141       if (i_ctrl_halt) begin
142           // Privilege HALT
143           CAR <= CAR;
144
145       end
146       else if (ctrl_step_execution) begin
147           // Step-by-step instruction fetch
148           if (i_next_instr_stimulus) begin
149               CAR <= 7'h00;
150               indirect_done <= 1'b0;
151
152           end
153           else begin
154               // NOP WB Stage
155               CAR <= 7'h20;
156
157           end
158
159       end
160       else begin
161           // Auto fetch
162           CAR <= 7'h00; // Fetch next instruction
163           indirect_done <= 1'b0; // Reset Indirect Flag
164
165       end
166
167   end
168
169   default:
170
171       CAR <= CAR; // Prevent latch

```

```

152     endcase
153
154   end
155
156 assign o_car_data = ctrl_cpu_start ? CAR : 7'h20;
157
158 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
28 input ctrl_cpu_start;
29 input [23:0] memory;
30 output ctrl_global_halt; // C23
31 output ctrl_mar_increment; // C22
32 output [1:0] next_addr; // C21-C20
33 output [3:0] ALU_op; // C19-C16
34
35 output C0, C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13, C14, C15;
36
37
38 assign C0 = ctrl_cpu_start & memory[0];
39 assign C1 = ctrl_cpu_start & memory[1];
40 assign C2 = ctrl_cpu_start & memory[2];
41 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 // Assignments
107
108 assign o_IF_stage = C2;
109
110 endmodule

```

Listing 10: cu_top.v

A.5 内部寄存器与 ALU 设计

```

1 /*
2 module ALU
3 Author: LiPtP

```

```

4   function:
5     1. update BR and MR registers on rising clock edge when `ctrl_alu_en` is open;
6     2. Bus control using C9, C10, the target port is o_br and o_mr;
7     3. Operation encoding is defined in doc.
8     4. Currently, this ALU is the optimal design within all tried methods.
9   */
10  module ALU (
11    i_clk,
12    i_rst_n,
13    i_acc_alu_p,
14    i_acc_alu_q,
15    ctrl_alu_op,
16    ctrl_alu_en,
17    C9,
18    C10,
19    o_mr,
20    o_br,
21    o_flags,
22    i_user_sample,
23    o_mr_user
24  );
25  input i_clk;
26  input i_rst_n;
27  input [15:0] i_acc_alu_p;
28  input [15:0] i_acc_alu_q;
29  input [2:0] ctrl_alu_op;
30  input ctrl_alu_en;
31  input C9;
32  input C10;
33  output [15:0] o_mr;
34  output [15:0] o_br;
35  output [4:0] o_flags;
36  input i_user_sample;
37  output [15:0] o_mr_user;
38
39 // Re-interpret input to signed values
40 wire signed [15:0] ALU_P;
41 wire signed [15:0] ALU_Q;
42
43 // Calculation result
44 reg signed [15:0] ALU_RES_LOW;
45 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'b110) begin
111       MR <= ALU_RES_HIGH;
112     end
113   end
114   // On write back, MR are set to 0
115   else if (C10 && !i_user_sample) begin
116     MR <= 16'b0;
117   end
118   else begin
119     BR <= BR;
120     MR <= MR;
121   end
122 end
123
124 // Sequential logic: Update Flags upon ctrl_alu_en
125 always @(posedge i_clk or negedge i_rst_n) begin
126   if (!i_rst_n) begin
127     ZF <= 1'b0;
128     CF <= 1'b0;
129     OF <= 1'b0;
130     NF <= 1'b0;
131     MF <= 1'b0;
132   end
133   else if (ctrl_alu_en) begin // EX
134     if (ctrl_alu_op == 3'b000) begin // ADD
135       OF <= MF ? ((MR[15] == ALU_P[15]) && (ALU_RES_HIGH[15] != MR[15])) :
136           ((ALU_P[15] == ALU_Q[15]) && (ALU_RES_LOW[15] != ALU_P[15]));
137     end
138     else if (ctrl_alu_op == 3'b001) begin // SUB
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'b010) begin // MPY
143       OF <= MF ? ((ALU_P[15] == ALU_Q[15]) && (ALU_RES_HIGH[15] != 16'b0)) :
144           ((ALU_P[15] == ALU_Q[15]) && (ALU_RES_LOW[15] != 16'b0));

```

```

145      end
146
147      else begin
148          OF <= 1'b0;
149      end
150
151      CF <= (ctrl_alu_op == 3'b110) ? ALU_P[15 - ALU_Q] : // SHIFTL highest shiftout bit
152          (ctrl_alu_op == 3'b111) ? ALU_P[ALU_Q] : 1'b0; // SHIFTR lowest shiftout bit
153
154  end
155
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
161  end
162
163  else begin
164      ZF <= ZF;
165      CF <= CF;
166      OF <= OF;
167      NF <= NF;
168      MF <= (MR != 16'b0); // preventing one cycle ctrl_en
169
170  end
171
172 end
173
174
175 // Input
176 assign ALU_P = i_acc_alu_p;
177 assign ALU_Q = i_acc_alu_q;
178
179 // Output
180 assign o_br = C9 ? BR : 16'b0;
181 assign o_mr = C10 ? MR : 16'b0;
182 assign o_flags = {ZF, CF, OF, NF, MF};
183 assign o_mr_user = i_user_sample ? MR : 16'b0;
184
185 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 input i_clk;
18 input i_rst_n;
19 input [15:0] i_br_acc;
20 input [15:0] i_mr_acc;
21 input [15:0] i_mbr_acc;
22 input C7;
23 input C9;
24 input C10;
25 input C11;
26 input C12;
27 input i_user_sample;
28 output [15:0] o_acc_alu_p;
29 output [15:0] o_acc_mbr;
30 output [15:0] o_acc_user;
31 reg [15:0] ACC;
32
33 always @(posedge i_clk or negedge i_rst_n) begin
34   if (!i_rst_n) begin
35     ACC <= 16'b0;
36   end
37   else begin
38     if (C9) begin
39       ACC <= i_br_acc;
40     end
41     else if (C10) begin
42       ACC <= i_mr_acc;
43     end
44     else if (C11) begin
45       ACC <= i_mbr_acc;
46     end
47     else begin
48       ACC <= ACC;
49     end
50   end
51 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

```

```

3 Author: LiPtP
4 function:
5 1. self increment upon STOREH implicit instruction
6 2. write value sequence: MBR > PC
7 */
8 `timescale 1ns / 1ps
9 module MAR (
10     i_clk,
11     i_rst_n,
12     i_mbr_mar,
13     i_pc_mar,
14     C2,
15     C8,
16     ctrl_mar_increment,
17     o_mar_address_bus
18 );
19 input i_clk;
20 input i_rst_n;
21 input ctrl_mar_increment;
22 input C2;
23 input C8;
24 input [7:0] i_mbr_mar;
25 input [7:0] i_pc_mar;
26 output [7:0] o_mar_address_bus;
27
28 reg [7:0] MAR;
29
30 always @ (posedge i_clk or negedge i_rst_n) begin
31     if (!i_rst_n) begin
32         MAR <= 8'b0;
33     end
34     else begin
35         if (ctrl_mar_increment) begin
36             MAR <= MAR + 1;
37         end
38         else begin
39             if (C8) begin
40                 MAR <= i_mbr_mar;
41             end
42             else if (C2) begin
43                 MAR <= i_pc_mar;
44             end
45             else begin
46                 MAR <= MAR;
47             end
48         end
49     end
50 end

```

```

50 end
51
52 // Address bus judgement logic at reg_top
53 assign o_mar_address_bus = MAR;
54 endmodule

```

Listing 13: mar.v

```

/*
1 module MBR
2 Author: LiPtP
3 function:
4 1. write value sequence: Bus > IR > PC > ACC
5 */
6 `timescale 1ns / 1ps
7 module MBR (
8     i_clk,
9     i_rst_n,
10    i_pc_mbr,
11    i_ir_mbr,
12    i_data_bus_mbr,
13    i_acc_mbr,
14    o_mbr_data_bus,
15    o_mbr_pc,
16    o_mbr_ir,
17    o_mbr_mar,
18    o_mbr_acc,
19    o_mbr_alu_q,
20    C1,
21    C3,
22    C4,
23    C5,
24    C6,
25    C8,
26    C11,
27    C12,
28    C15
29 );
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
   operand on immediate addressing.
50  output [15:0] o_mbr_ir;
51
52  output [7:0] o_mbr_mar;
53  output [15:0] o_mbr_acc;
54  output [15:0] o_mbr_alu_q;
55
56 reg [15:0] MBR;
57
58 always @ (posedge i_clk or negedge i_rst_n) begin
59   if (!i_rst_n) begin
60     MBR <= 16'b0;
61   end
62   else begin
63     if (C5) begin
64       MBR <= i_data_bus_mbr;
65     end
66     else if (C15) begin
67       MBR <= {8'b0, i_ir_mbr};
68     end
69     else if (C1) begin
70       MBR <= {8'b0, i_pc_mbr};
71     end
72     else if (C12) begin
73       MBR <= i_acc_mbr;
74     end
75     else begin
76       MBR <= MBR;
77     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 module PC
2 Author: LiPtP
3 function:
4 1. self increment upon C2
5 2. write value sequence: MBR
6 3. PC starts from 1 (0428 updated)
7 */
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 module IR
2 Author: LiPtP
3 function:
4 1. dump high 8 bits to CU
5 2. store immediate opcode and operand from MBR and push back operand on F0 stage
*/
6
7 module IR (
8   i_clk,
9   i_rst_n,
10  i_mbr_ir,
11  C4,
12  C14,
13  C15,
14  o_ir_cu,
15  o_ir_mbr,
16  i_user_sample,
17  o_ir_user
18 );
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 module REG_TOP(
11     ctrl_cpu_start,
12     i_user_sample,
13     o_ACC_user,
14     o_MR_user,
15     o_PC_user,
16     o_IR_user,
17     i_clk,
18     i_rst_n,
19     i_memory_data,
20     o_memory_addr,
21     o_memory_data,
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 MBR reg_MBR(
122     .i_clk(i_clk),
123     .i_rst_n(i_rst_n),
124     .i_pc_mbr(PC_MBR),

```

```

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

```

```

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

```

Listing 17: reg_top.v

A.6 数据内存设计

```

/*
1 module DATA_RAM
2 Author: LiPtP
3 function:
4 0. Write means write to RAM, READ means read from RAM
5 1. Write data to itself according to input address and data
6 2. Output data according to input address
*/
7 module DATA_RAM (
8   i_clk,
9   ctrl_write,
10  i_addr_write,
11  i_data_write,
12  ctrl_read,
13  i_addr_read,
14  o_data_read
15 );
16
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 // Read Operation
39 assign o_data_read = ctrl_read ? mem[i_addr_read] : 16'b0;
40
41
42 endmodule

```

Listing 18: top_data_ram.v

A.7 外部总线设计

```

1
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    end
58    else begin
59      if (C2) begin
60        memory_select <= 1'b1; // ROM
61      end
62      else begin
63        memory_select <= 1'b0; // RAM
64      end

```

```

65     end
66 end
67
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.8 用户面设计

```

`timescale 1ns / 1ps

module KEY_JITTER #(
    parameter CNT_MAX = 20'hf_ffff,
    parameter POSEdge = 1'b0
) (
    i_clk,
    key_in,
    key_out
);

```

```

11
12    input i_clk;
13    input key_in;
14    output key_out;
15
16
17    reg [1:0] key_in_r;
18    reg [19:0] cnt_base;
19    reg key_value_r;
20    reg key_value_rd;
21    reg key_posedge;
22
23 // Sample and stage key input
24 always @(posedge i_clk) begin
25     key_in_r <= {key_in_r[0], key_in};
26 end
27
28 // Counter starts if key changes
29 always @(posedge i_clk) begin
30     if (key_in_r[0] != key_in_r[1]) begin
31         cnt_base <= 20'b0;
32     end
33     else if(key_in_r[0] == key_in_r[1]) begin
34         if (cnt_base < CNT_MAX) begin
35             cnt_base <= cnt_base + 1'b1;
36         end
37         else if (cnt_base == CNT_MAX) begin
38             cnt_base <= 20'b0;
39         end
40     end
41     else begin
42         // reset logic
43         cnt_base <= 20'b0;
44     end
45 end
46
47 // Prevent Synthesis from optimizing out the register
48 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 input i_clk;
21 input i_rst_n;
22 input [7:0] segment_current_PC;
23 input [7:0] segment_current_Opcode;
24 input [4:0] segment_flags;
25 input [15:0] segment_result_high;
26 input [15:0] segment_result_low;
27 input [7:0] segment_max_instr_addr;
28 input button_check_instruction;
29 input button_check_flags;
30 input button_check_result;
31 input light_instr_transmit_done;
32 input switch_start_cpu;
33 output [7:0] o_seg_valid;
34 output [7:0] o_seg_value;
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 reg [2:0] current_state, next_state;
46
47 // 状态机: 状态切换逻辑
48 always @ (posedge i_clk or negedge i_rst_n) begin
49   if (!i_rst_n) begin
50     current_state <= STATE_DEFAULT;
51   end
52   else begin
53     current_state <= next_state;
54   end
55 end
56
57 // 状态机: 下一状态逻辑
58 always @(*) begin
59   case (current_state)
60     STATE_DEFAULT: begin
61       if (button_check_instruction) begin
62         next_state = STATE_INSTRUCTION;
63       end
64       else if (button_check_flags) begin
65         next_state = STATE_FLAGS;
66       end
67       else if (button_check_result) begin
68         next_state = STATE_RESULT;
69       end
70       else if (light_instr_transmit_done && !switch_start_cpu) begin
71         next_state = STATE_MAX_ADDR;
72       end
73     else begin
74       next_state = STATE_DEFAULT;
75     end
76   end
77   STATE_INSTRUCTION: begin
78     if (!button_check_instruction) begin
79       next_state = STATE_DEFAULT;
80     end
81     else begin
82       next_state = STATE_INSTRUCTION;
83     end
84   end
85   STATE_FLAGS: begin
86     if (!button_check_flags) begin
87       next_state = STATE_DEFAULT;
88     end

```

```

85         else begin
86             next_state = STATE_FLAGS;
87         end
88     end
89
90     STATE_RESULT: begin
91         if (!button_check_result) begin
92             next_state = STATE_DEFAULT;
93         end
94         else begin
95             next_state = STATE_RESULT;
96         end
97     end
98
99     STATE_MAX_ADDR: begin
100        if (!(light_instr_transmit_done && !switch_start_cpu)) begin
101            next_state = STATE_DEFAULT;
102        end
103        else begin
104            next_state = STATE_MAX_ADDR;
105        end
106    end
107
108    default: begin
109        next_state = STATE_DEFAULT;
110    end
111
112    endcase
113 end
114
115 // 状态机: 输出逻辑
116 always @ (posedge i_clk or negedge i_rst_n) begin
117     if (!i_rst_n) begin
118         segment_data <= 32'b0;
119     end
120     else begin
121         case (current_state)
122             STATE_DEFAULT: begin
123                 segment_data <= segment_data;
124             end
125             STATE_INSTRUCTION: begin
126                 segment_data <= {8'b0, segment_current_PC, 8'b0, segment_current_Opcode};
127             end
128             STATE_FLAGS: begin
129                 segment_data <= {15'b0,segment_flags[4],3'b0,segment_flags[3],3'b0,segment_flags
130                               [2],3'b0,segment_flags[1],3'b0,segment_flags[0]};
131             end
132             STATE_RESULT: begin
133                 segment_data <= {segment_result_high, segment_result_low};
134             end
135             STATE_MAX_ADDR: begin

```

```

135         segment_data <= {24'b0, segment_max_instr_addr};
136     end
137
138     default: begin
139         segment_data <= 32'b0;
140     end
141
142     endcase
143
144 end
145
146 SEVEN_SEGMENT #(
147     .SCAN_INTERVAL(SCAN_INTERVAL)
148 ) seven_segment (
149     .i_clk(i_clk),
150     .i_rst_n(i_rst_n),
151     .i_data(segment_data),
152     .o_seg_valid(o_seg_valid),
153     .o_seg_value(o_seg_value)
154 );
155
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;
output reg [7:0] o_seg_value;

// Clock Divider
reg [15:0] count_num;
always @(posedge i_clk or negedge i_rst_n) begin
    if (!i_rst_n) begin
        count_num <= 3'b0;
    end
    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
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:
65             display_value = {1'b0, i_data[27:24]};
66         3'd2:
67             display_value = {1'b0, i_data[23:20]};
68         3'd3:
69             display_value = {1'b0, i_data[19:16]};
70         3'd4:
71             display_value = {1'b0, i_data[15:12]};
72         3'd5:
73             display_value = {1'b0, i_data[11:8]};

```

```

74      3'd6:
75          display_value = {1'b0, i_data[7:4]};
76      3'd7:
77          display_value = {1'b0, i_data[3:0]};
78      default:
79          display_value = 5'b0;
80      endcase
81  end
82
83 // Encoder of 7-segment display
84 localparam SEG_0 = 8'b1100_0000, SEG_1 = 8'b1111_1001,
85     SEG_2 = 8'b1010_0100, SEG_3 = 8'b1011_0000,
86     SEG_4 = 8'b1001_1001, SEG_5 = 8'b1001_0010,
87     SEG_6 = 8'b1000_0010, SEG_7 = 8'b1111_1000,
88     SEG_8 = 8'b1000_0000, SEG_9 = 8'b1001_0000,
89     SEG_A = 8'b1000_1000, SEG_B = 8'b1000_0011,
90     SEG_C = 8'b1100_0110, SEG_D = 8'b1010_0001,
91     SEG_E = 8'b1000_0110, SEG_F = 8'b1000_1110;
92
93 // Something else to display
94 localparam SEG_S = 8'b1011_1111, SEG_r = 8'b1010_1111,
95     SEG_o = 8'b1010_0011, SEG_n = 8'b1111_1111,
96     SEG_ot = 8'b1001_1100, SEG_left = 8'b1111_1100,
97     SEG_right = 8'b1101_1110, SEG_happy = 8'b1110_0011,
98     SEG_sad = 8'b1010_1011;
99
100 always @(*) begin
101     case (display_value)
102         5'd0:
103             o_seg_value = SEG_0;
104         5'd1:
105             o_seg_value = SEG_1;
106         5'd2:
107             o_seg_value = SEG_2;
108         5'd3:
109             o_seg_value = SEG_3;
110         5'd4:
111             o_seg_value = SEG_4;
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;
5'd10:
124         o_seg_value = SEG_A;
5'd12:
125         o_seg_value = SEG_C;
5'd13:
126         o_seg_value = SEG_D;
5'd14:
127         o_seg_value = SEG_E;
5'd15:
128         o_seg_value = SEG_F;
5'd16:
129         o_seg_value = SEG_S;
5'd17:
130         o_seg_value = SEG_r;
5'd18:
131         o_seg_value = SEG_o;
5'd19:
132         o_seg_value = SEG_n;
5'd20:
133         o_seg_value = SEG_ot;
5'd21:
134         o_seg_value = SEG_left;
5'd22:
135         o_seg_value = SEG_right;
5'd23:
136         o_seg_value = SEG_happy;
5'd24:
137         o_seg_value = SEG_sad;
default:
138         o_seg_value = 8'b1;
endcase
139
end
140
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
29
30
31
32
33
34  reg [1:0] state;
35  localparam STATE_IDLE = 2'b00;
36  localparam STATE_GREEN = 2'b01;
37  localparam STATE_BLUE = 2'b10;
38  localparam STATE_RED = 2'b11;
39
40
41  always @ (posedge i_clk or negedge i_RST_N) begin
42    if (!i_RST_N) begin
43      state <= STATE_IDLE;
44      RGB2_RED_reg <= 1'b0;
45      RGB2_BLUE_reg <= 1'b0;
46      RGB2_GREEN_reg <= 1'b0;
47    end
48    else begin
49      case (state)
50        STATE_IDLE: begin
51          RGB2_GREEN_reg <= 1'b0;
52          RGB2_BLUE_reg <= 1'b0;
53          RGB2_RED_reg <= 1'b0;
54          if (i_instr_transmit_done) begin
55            state <= STATE_GREEN;
56          end

```

```

57     end
58
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
68     STATE_BLUE: begin
69         RGB2_GREEN_reg <= 1'b0;
70         RGB2_BLUE_reg <= 1'b1;
71         RGB2_RED_reg <= 1'b0;
72         if (i_halt) begin
73             state <= STATE_RED;
74         end
75     end
76
77     STATE_RED: begin
78         RGB2_GREEN_reg <= 1'b0;
79         RGB2_BLUE_reg <= 1'b0;
80         RGB2_RED_reg <= 1'b1;
81     end
82
83     default: begin
84         state <= STATE_IDLE;
85     end
86
87     endcase
88
89 end
90
91 always @ (posedge i_clk or negedge i_rst_n) begin
92     if (!i_rst_n) begin
93         RGB1_BLUE_reg <= 1'b0;
94         RGB1_RED_reg <= 1'b0;
95     end
96     else begin
97         if (i_step_execution) begin
98             RGB1_BLUE_reg <= 1'b0;
99             RGB1_RED_reg <= 1'b1;
100        end
101        else begin
102            RGB1_BLUE_reg <= 1'b1;
103            RGB1_RED_reg <= 1'b0;
104        end
105    end
106
107 end
108
109 // Assignments
110
111 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

A.9 顶层模块、测试与验证模块

```

1 /*
2 module CPU_TOP
3 Author: LiPtP
4
5 Should instantiate:
6 1. External Bus
7 2. Control Unit
8 3. Internal Registers
9 4. Memories
10 and they should be placed at the same hierarchy level as they explains the whole CPU.
11
12 The user interface should acknowledge part of CPU status and provide input to CPU, so it should
13      be at the same level as this module.
14 */
15 module TOP_CPU(
16     i_clk,
17     i_rst_n,
18     ctrl_step_execution,
19     i_user_sample,
20     i_rx,
21     i_start_cpu,
22     i_next_instr_stimulus,
23     o_instr_transmit_done,
24     o_max_addr,
25     o_halt,
26     o_alu_result_low,
27     o_alu_result_high,
28     o_flags,
29     o_current_Opcode,
30     o_current_PC
31 );
32
33 input i_clk;
34 input i_rst_n;
35

```

```

36 // from/to user interface
37 input ctrl_step_execution;
38 input i_next_instr_stimulus;
39 input i_user_sample;
40 input i_rx;
41 input i_start_cpu; // start CPU execution
42 output o_instr_transmit_done;
43 output [7:0] o_max_addr;
44 output o_halt; // CPU halt signal
45
46 output reg [4:0] o_flags; // ALU flags
47 output reg [7:0] o_current_Opcode;
48 output reg [7:0] o_current_PC;
49 output reg [15:0] o_alu_result_low;
50 output reg [15:0] o_alu_result_high;
51
52 // external bus
53 wire [15:0] MBR_DATA_BUS;
54 wire [15:0] DATA_BUS_MBR;
55 wire [15:0] DATA_BUS_MEMORY;
56 wire [15:0] INSTR_MEMORY_DATA_BUS;
57 wire [15:0] DATA_MEMORY_DATA_BUS;
58 wire [7:0] MAR_ADDR_BUS;
59 wire [7:0] ADDR_BUS_MEMORY;
60 wire en_write_to_instr;
61 wire en_write_to_data;
62 wire en_read_from_data;
63 wire en_read_from_instr;
64
65 // internal registers & CU
66 wire C0,C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,C12,C13,C14,C15;
67 wire [4:0] flags;
68 wire [7:0] opcode;
69 wire [3:0] alu_op;
70
71 wire halt;
72 wire mar_increment;
73
74
75 // To User Interface
76 wire [7:0] PC;
77 wire [7:0] IR;
78 wire [15:0] ALU_RES_LOW;
79 wire [15:0] ALU_RES_HIGH;
80
81 always @(posedge i_clk or negedge i_rst_n) begin
82     if(!i_rst_n) begin

```

```

83     o_current_Opcode <= 8'b0;
84     o_current_PC <= 8'b0;
85     o_alu_result_low <= 16'b0;
86     o_alu_result_high <= 16'b0;
87     o_flags <= 5'b0;
88   end
89   else begin
90     if(i_user_sample && i_start_cpu) begin
91       o_current_Opcode <= IR;
92       o_current_PC <= PC;
93       o_alu_result_low <= ALU_RES_LOW;
94       o_alu_result_high <= ALU_RES_HIGH;
95       o_flags <= flags;
96     end
97     else if(!i_start_cpu) begin
98       o_current_Opcode <= 8'b0;
99       o_current_PC <= 8'b0;
100      o_alu_result_low <= 16'b0;
101      o_alu_result_high <= 16'b0;
102      o_flags <= 5'b0;
103    end
104    else begin
105      o_current_Opcode <= o_current_Opcode;
106      o_current_PC <= o_current_PC;
107      o_alu_result_low <= o_alu_result_low;
108      o_alu_result_high <= o_alu_result_high;
109      o_flags <= o_flags;
110    end
111  end
112 end
113
114 // Assignments
115
116 assign o_halt = halt;
117
118
119
120 EXTERNAL_BUS external_bus(
121   .i_clk(i_clk),
122   .i_rst_n(i_rst_n),
123   .i_mbr_data_bus(MBR_DATA_BUS),
124   .i_mar_address_bus(MAR_ADDR_BUS),
125   .o_data_bus_memory(DATA_BUS_MEMORY),
126   .o_address_bus_memory(ADDR_BUS_MEMORY),
127   .o_data_ram_write(en_write_to_data),
128   .o_instr_rom_read(en_read_from_instr),
129   .o_data_ram_read(en_read_from_data),

```

```

130     .CO(CO),
131     .C2(C2),
132     .C5(C5),
133     .C13(C13),
134     .i_instr(INSTR_MEMORY_DATA_BUS),
135     .i_data(DATA_MEMORY_DATA_BUS),
136     .o_data_bus_mbr(DATA_BUS_MBR)
137
138   );
139
DATA_RAM data_ram(
140   .i_clk(i_clk),
141   .ctrl_write(en_write_to_data),
142   .i_addr_write(ADDR_BUS_MEMORY),
143   .i_data_write(DATA_BUS_MEMORY),
144   .ctrl_read(en_read_from_data),
145   .i_addr_read(ADDR_BUS_MEMORY),
146   .o_data_read(DATA_MEMORY_DATA_BUS)
147 );
148
149
150
INSTR_ROM instruction_rom(
151   .i_clk(i_clk),
152   .i_rst_n(i_rst_n),
153   .i_rx(i_rx),
154   .en_read(en_read_from_instr),
155   .i_addr_read(ADDR_BUS_MEMORY),
156   .o_instr_read(INSTR_MEMORY_DATA_BUS),
157   .o_instr_transmit_done(o_instr_transmit_done),
158   .o_max_addr(o_max_addr)
159 );
160
REG_TOP internal_registers(
161   .ctrl_cpu_start(i_start_cpu),
162   .i_user_sample(i_user_sample),
163   .o_ACC_user(ALU_RES_LOW),
164   .o_MR_user(ALU_RES_HIGH),
165   .o_PC_user(PC),
166   .o_IR_user(IR),
167   .i_clk(i_clk),
168   .i_rst_n(i_rst_n),
169   .i_memory_data(DATA_BUS_MBR),
170   .o_memory_addr(MAR_ADDR_BUS),
171   .o_memory_data(MBR_DATA_BUS),
172   .o_ir_cu(opcode),
173   .o_flags(flags),
174   .i_alu_op(alu_op),
175   .i_ctrl_halt(halt),
176   .i_ctrl_mar_increment(mar_increment),

```

```

177     .C1(C1),
178     .C2(C2),
179     .C3(C3),
180     .C4(C4),
181     .C5(C5),
182     .C6(C6),
183     .C7(C7),
184     .C8(C8),
185     .C9(C9),
186     .C10(C10),
187     .C11(C11),
188     .C12(C12),
189     .C14(C14),
190     .C15(C15)
191 );
192
193
194 CU_TOP control_unit(
195     .ctrl_cpu_start(i_start_cpu),
196     .ctrl_step_execution(ctrl_step_execution),
197     .i_next_instr_stimulus(i_next_instr_stimulus),
198     .i_clk(i_clk),
199     .i_rst_n(i_rst_n),
200     .i_flags(flags),
201     .i_ir_data(opcode),
202     .o_alu_op(alu_op),
203     .o_ctrl_halt(halt),
204     .o_IF_stage(),
205     .o_ctrl_mar_increment(mar_increment),
206     .CO(CO),
207     .C1(C1),
208     .C2(C2),
209     .C3(C3),
210     .C4(C4),
211     .C5(C5),
212     .C6(C6),
213     .C7(C7),
214     .C8(C8),
215     .C9(C9),
216     .C10(C10),
217     .C11(C11),
218     .C12(C12),
219     .C13(C13),
220     .C14(C14),
221     .C15(C15)
222 );
223

```

```
224  
225  
226 endmodule
```

Listing 24: cpu_top.v

```
1  /*  
2   module TOP  
3   Author: LiPtP  
4  
5   Should instantiate:  
6   1. CPU  
7   2. User Interface  
8  
9   This is the very top module of the CPU.  
10  * New ideas on 4.29 3.A.M  
11  1. When in STEP mode, instantiate a button to fetch specified Register Value. Use switches to  
12    decide the displayed register.  
13  2. When in AUTO mode, show ACC value and MR value on HALT (optional).  
14  */  
15  module TOP #(  
16      parameter MAX_DELAY_TOLERANCE = 20'hfffff,  
17      parameter SCAN_INTERVAL = 16'd30000  
18  )(  
19      // Should only declare signals from/to the board  
20      // and they are directly assigned to physical ports via constraint files.  
21      CLK_100MHz,  
22      START_CPU,  
23      STEP_EXECUTION,  
24      BTNC,  
25      BTNL,  
26      BTNR,  
27      BTNU,  
28      BTND,  
29      RXD,  
30      SEG_VALID,  
31      SEG_VALUE,  
32      RGB1_RED,  
33      RGB1_BLUE,  
34      RGB2_RED,  
35      RGB2_BLUE,  
36      RGB2_GREEN  
37  );  
38  
39  input CLK_100MHz;  
40  input START_CPU;  
41  input STEP_EXECUTION;  
42  input BTNL;
```

```

42    input BTND;
43    input BTNR;
44    input BTNU;
45    input BTNC;
46    input RXD;
47    output [7:0] SEG_VALID;
48    output [7:0] SEG_VALUE;
49    output RGB1_RED;
50    output RGB1_BLUE;
51    output RGB2_RED;
52    output RGB2_BLUE;
53    output RGB2_GREEN;

54
55 // ===== Wires =====
56 wire clk = CLK_100MHz;
57
58 wire switch_step_execution; // using a switch resource
59 wire switch_start_cpu;
60 wire button_rst;
61 wire button_rst_n = !button_rst;
62 wire button_next_instr;
63 wire button_check_result;
64 wire button_check_instruction;
65 wire button_check_flags;
66 wire user_sample = button_check_flags | button_check_instruction | button_check_result;
67
68 wire light_instr_transmit_done; // RGB2 Green
69 wire light_halt; // RGB2 Red
70 wire light_cpu_running; // RGB2 Blue
71 wire light_cpu_step; // RGB1 Red
72 wire light_cpu_auto; // RGB1 Blue

73
74 // This will show at cpu_start = 0
75 wire [7:0] segment_max_instr_addr;
76 wire [7:0] segment_current_PC;
77 wire [7:0] segment_current_Opcode;
78 wire [15:0] segment_result_low;
79 wire [15:0] segment_result_high;
80 wire [4:0] segment_flags;

81
82 wire rx = RXD;
83 // ===== Instantiations =====
84 TOP_CPU cpu (
85     .i_clk(clk),
86     .i_rst_n(button_rst_n),
87     .ctrl_step_execution(switch_step_execution),
88     .i_user_sample(user_sample),

```

```

85     .i_rx(rx),
86     .i_start_cpu(switch_start_cpu),
87     .i_next_instr_stimulus(button_next_instr),
88     .o_instr_transmit_done(light_instr_transmit_done),
89     .o_max_addr(segment_max_instr_addr),
90     .o_halt(light_halt),
91     .o_alu_result_low(segment_result_low),
92     .o_alu_result_high(segment_result_high),
93     .o_flags(segment_flags),
94     .o_current_Opcode(segment_current_Opcode),
95     .o_current_PC(segment_current_PC)
96   );
97 // ===== User Interfaces =====
98
99 // Signals prefixed "switch" should be instantiated here
100 KEY_JITTER #( .CNT_MAX(MAX_DELAY_TOLERANCE),
101   .POSEDGE(1'b0)
102   ) ins_switch_start_cpu(
103     .i_clk(clk),
104     .key_in(START_CPU),
105     .key_out(switch_start_cpu)
106   );
107 KEY_JITTER #( .CNT_MAX(MAX_DELAY_TOLERANCE),
108   .POSEDGE(1'b0)
109   ) ins_switch_step_execution(
110     .i_clk(clk),
111     .key_in(STEP_EXECUTION),
112     .key_out(switch_step_execution)
113   );
114
115 // Signals prefixed "button" should be instantiated here
116 KEY_JITTER #( .CNT_MAX(MAX_DELAY_TOLERANCE),
117   .POSEDGE(1'b1)
118   ) ins_button_reset (
119     .i_clk(clk),
120     .key_in(BTNU),
121     .key_out(button_rst)
122   );
123
124 KEY_JITTER #( .CNT_MAX(MAX_DELAY_TOLERANCE),
125   .POSEDGE(1'b1)
126   ) ins_button_next_instr(
127     .i_clk(clk),
128     .key_in(BTNC),
129     .key_out(button_next_instr)
130   );
131
132
133
134
135

```

```

136 KEY_JITTER #( .CNT_MAX(MAX_DELAY_TOLERANCE),
137                 .POSEDGE(1'b1)
138             ) ins_button_check_result(
139                 .i_clk(clk),
140                 .key_in(BTNL),
141                 .key_out(button_check_result)
142             );
143
144 KEY_JITTER #( .CNT_MAX(MAX_DELAY_TOLERANCE),
145                 .POSEDGE(1'b1)
146             ) ins_button_check_instruction(
147                 .i_clk(clk),
148                 .key_in(BTNR),
149                 .key_out(button_check_instruction)
150             );
151
152 KEY_JITTER #( .CNT_MAX(MAX_DELAY_TOLERANCE),
153                 .POSEDGE(1'b1)
154             ) ins_button_check_flags(
155                 .i_clk(clk),
156                 .key_in(BTND),
157                 .key_out(button_check_flags)
158             );
159
160 // Signals prefixed "segment" should be instantiated here
161 SEVEN_SEGMENT_DISPLAY #(
162     .SCAN_INTERVAL(SCAN_INTERVAL)
163 ) segment_display (
164     .i_clk(clk),
165     .i_rst_n(button_rst_n),
166     .switch_start_cpu(switch_start_cpu),
167     .button_check_instruction(button_check_instruction),
168     .button_check_flags(button_check_flags),
169     .button_check_result(button_check_result),
170     .light_instr_transmit_done(light_instr_transmit_done),
171     .segment_current_PC(segment_current_PC),
172     .segment_current_Opcode(segment_current_Opcode),
173     .segment_flags(segment_flags),
174     .segment_result_high(segment_result_high),
175     .segment_result_low(segment_result_low),
176     .segment_max_instr_addr(segment_max_instr_addr),
177     .o_seg_valid(SEG_VALID),
178     .o_seg_value(SEG_VALUE)
179 );
180
181 // Signals prefixed "light" should be instantiated here
182

```

```

183 LED_DISPLAY led_display(
184     .i_clk(clk),
185     .i_RST_N(button_RST_N),
186     .i_instr_transmit_done(light_instr_transmit_done),
187     .i_halt(light_halt),
188     .i_step_execution(switch_step_execution),
189     .i_start_cpu(switch_start_cpu),
190     .RGB1_RED(RGB1_RED),
191     .RGB1_BLUE(RGB1_BLUE),
192     .RGB2_RED(RGB2_RED),
193     .RGB2_BLUE(RGB2_BLUE),
194     .RGB2_GREEN(RGB2_GREEN)
195 );
196
197 // ===== End of Module =====
198 endmodule

```

Listing 25: top.v

```

`timescale 1ns/1ps
module tb_CPU;

// Parameters
parameter bit_period = 8680; // 8.68us, if timescale is 1ns

// Registers
reg i_rx;
reg cpu_start;
reg next_instr;
reg user_sample;
reg clk;
reg rst_n;

// Wires
wire instr_transmit_done;
wire [7:0] max_addr_instr;
wire halt;
wire [15:0] alu_result;
wire [15:0] alu_result_high;
wire [4:0] flags;
wire [7:0] current_opcode;
wire [7:0] current_pc;

// Task: UART send byte
task uart_send_byte(input [7:0] data);
    integer i;
    begin
        // Start bit

```

```

30     i_rx = 0;
31     #(bit_period);
32     // Data bits (LSB first)
33     for (i = 0; i < 8; i = i + 1) begin
34         i_rx = data[i];
35         #(bit_period);
36     end
37     // Stop bit
38     i_rx = 1;
39     #(bit_period);
40     i_rx = 1;
41     #(bit_period);
42   end
43 endtask
44
45 // Clock generation
46 initial begin
47   clk = 0;
48   forever
49     #5 clk = !clk;
50 end
51
52 // Asynchronous reset
53 initial begin
54   rst_n = 1;
55   #100 rst_n = 0;
56   #5 rst_n = 1;
57 end
58
59 // DumpFile
60 initial begin
61   $dumpfile("cpu_0428.vcd");
62   $dumpvars(0, tb_CPU);
63 end
64
65 // RXD Data
66 initial begin
67   i_rx = 1; // idle state high
68   #(bit_period);
69   // uart_send_byte(8'b01000001);
70   // uart_send_byte(8'b00000000);
71   // uart_send_byte(8'b10000001);
72   // uart_send_byte(8'b10000000);
73   // uart_send_byte(8'b01000001);
74   // uart_send_byte(8'b10000000);
75   // uart_send_byte(8'b10000001);
76   // uart_send_byte(8'b01000000);

```

```

71 // uart_send_byte(8'b01000000);
72 // uart_send_byte(8'b10000000);
73 // uart_send_byte(8'b11000000);
74 // uart_send_byte(8'b01000000);
75 // uart_send_byte(8'b10000001);
76 // uart_send_byte(8'b10000000);
77 // uart_send_byte(8'b01000000);
78 // uart_send_byte(8'b01000000);
79 // uart_send_byte(8'b11000001);
80 // uart_send_byte(8'b10000000);
81 // uart_send_byte(8'b10000001);
82 // uart_send_byte(8'b01000000);
83 // uart_send_byte(8'b01000000);
84 // uart_send_byte(8'b01000000);
85 // uart_send_byte(8'b11000001);
86 // uart_send_byte(8'b10000000);
87 // uart_send_byte(8'b10000001);
88 // uart_send_byte(8'b01000000);
89 // uart_send_byte(8'b01000001);
90 // uart_send_byte(8'b10100110);

91
92 // uart_send_byte(8'b00100000);
93 // uart_send_byte(8'b01000000);

94
95 // uart_send_byte(8'b10100001);
96 // uart_send_byte(8'b10100000);

97
98 // uart_send_byte(8'b01000000);
99 // uart_send_byte(8'b10000000);

100
101 // uart_send_byte(8'b11100000);
102 // uart_send_byte(8'b00000000);
103 // mul.txt
104 // uart_send_byte(8'b01000001);
105 // uart_send_byte(8'b11111111);
106 // uart_send_byte(8'b00010001);
107 // uart_send_byte(8'b01111111);
108 // uart_send_byte(8'b10000001);
109 // uart_send_byte(8'b10000000);
110 // uart_send_byte(8'b01000001);
111 // uart_send_byte(8'b11111111);
112 // uart_send_byte(8'b00010000);
113 // uart_send_byte(8'b10000000);
114 // uart_send_byte(8'b10000001);
115 // uart_send_byte(8'b11000000);
116 // uart_send_byte(8'b01000000);
117 // uart_send_byte(8'b11000000);
118 // uart_send_byte(8'b01000000);
119 // uart_send_byte(8'b00100000);
120 // uart_send_byte(8'b11100000);
121 // uart_send_byte(8'b00000000);

122
123 // shift.txt

```

```

124 // uart_send_byte(8'b01000001);
125 // uart_send_byte(8'b10000000);
126 // uart_send_byte(8'b10110001);
127 // uart_send_byte(8'b11110000);
128 // uart_send_byte(8'b00110001);
129 // uart_send_byte(8'b01010000);
130 // uart_send_byte(8'b10000001);
131 // uart_send_byte(8'b10000000);
132 // uart_send_byte(8'b10110001);
133 // uart_send_byte(8'b11010000);
134 // uart_send_byte(8'b10000001);
135 // uart_send_byte(8'b01000000);
136 // uart_send_byte(8'b01000000);
137 // uart_send_byte(8'b10000000);
138 // uart_send_byte(8'b01000000);
139 // uart_send_byte(8'b01000000);
140 // uart_send_byte(8'b11100000);
141 // uart_send_byte(8'b00000000);

142
143 // logic.txt
144 // uart_send_byte(8'b01000001);
145 // uart_send_byte(8'b00100110);
146 // uart_send_byte(8'b10000001);
147 // uart_send_byte(8'b10000000);
148 // uart_send_byte(8'b01000001);
149 // uart_send_byte(8'b11000110);
150 // uart_send_byte(8'b00100000);
151 // uart_send_byte(8'b10000000);
152 // uart_send_byte(8'b10010001);
153 // uart_send_byte(8'b00110000);
154 // uart_send_byte(8'b01010001);
155 // uart_send_byte(8'b11000000);
156 // uart_send_byte(8'b10000001);
157 // uart_send_byte(8'b01000000);
158 // uart_send_byte(8'b11010000);
159 // uart_send_byte(8'b01000000);
160 // uart_send_byte(8'b01100001);
161 // uart_send_byte(8'b11010000);
162 // uart_send_byte(8'b11000001);
163 // uart_send_byte(8'b10000000);
164 // uart_send_byte(8'b11100000);
165 // uart_send_byte(8'b00000000);

166
167 // mul_add.txt
168 uart_send_byte(8'b01000001);
169 uart_send_byte(8'b00010011);
170 uart_send_byte(8'b00010001);

```

```

171 uart_send_byte(8'b11100000);
172 uart_send_byte(8'b10000001);
173 uart_send_byte(8'b00100000);
174 uart_send_byte(8'b01000001);
175 uart_send_byte(8'b11111111);
176 uart_send_byte(8'b00010001);
177 uart_send_byte(8'b01111111);
178 uart_send_byte(8'b10000001);
179 uart_send_byte(8'b10000000);
180 uart_send_byte(8'b01000001);
181 uart_send_byte(8'b11111111);
182 uart_send_byte(8'b00010000);
183 uart_send_byte(8'b10000000);
184 uart_send_byte(8'b00100000);
185 uart_send_byte(8'b00100000);
186 uart_send_byte(8'b10000001);
187 uart_send_byte(8'b10100000);
188 uart_send_byte(8'b01000000);
189 uart_send_byte(8'b10100000);
190 uart_send_byte(8'b01000000);
191 uart_send_byte(8'b01100000);
192 uart_send_byte(8'b11100000);
193 uart_send_byte(8'b00000000);
194 end
195
196
197 // Task: Wait for CPU start
198 task cpu_start_task;
199 begin
200     cpu_start = 0;
201     wait(instr_transmit_done);
202     #1000 cpu_start = 1;
203 end
204 endtask
205
206 // Task: Execute instructions
207 task execute_instructions;
208 begin
209     user_sample = 0;
210     wait(cpu_start == 1);
211     repeat (10) @(posedge clk);
212     while (!halt) begin
213
214         #10 user_sample = 1;
215         #10 user_sample = 0;
216         #10000;
217         next_instr = 1;

```

```

218     #10 next_instr = 0;
219     repeat (10) @(posedge clk);
220 end
221 $display("CPU halted at %t", $time);
222 $display("ALU result: %h", alu_result);
223 $display("ALU result high: %h", alu_result_high);
224 #10 user_sample = 1;
225 #10 user_sample = 0;
226 #10000;
227 end
228 endtask
229
230 // Task: Reset and restart CPU
231 task reset_and_restart_cpu;
232 begin
233     #1000 cpu_start = 0;
234     #1000 rst_n = 0;
235     #10 rst_n = 1;
236     #(bit_period);
237     uart_send_byte(8'b01000001);
238     uart_send_byte(8'b00000000);
239 end
240 endtask
241
242 // Main initial block
243 initial begin
244     cpu_start_task();
245     execute_instructions();
246     reset_and_restart_cpu();
247     wait(instr_transmit_done);
248     #10000 $finish;
249 end
250
251
252
253
254 TOP_CPU uut_cpu(
255     .i_clk(clk),
256     .i_rst_n(rst_n),
257     .ctrl_step_execution(1'b1),
258     .i_user_sample(user_sample),
259     .i_rx(i_rx),
260     .i_start_cpu(cpu_start),
261     .i_next_instr_stimulus(next_instr),
262     .o_instr_transmit_done(instr_transmit_done),
263     .o_max_addr(max_addr_instr),
264     .o_halt(halt),

```

```

265     .o_alu_result_low(alu_result),
266     .o_alu_result_high(alu_result_high),
267     .o_flags(flags),
268     .o_current_Opcode(current_opcode),
269     .o_current_PC(current_pc)
270   );
271
272 endmodule

```

Listing 26: CPU Testbench

```

1 // Author: LiPtP
2 // Last modified: 2025.5.5
3 // This is a testbench to test user interface and CPU.
4 `timescale 1ns/1ps
5 module tb_TOP;
6
7 // Parameters
8 parameter bit_period = 8680; // 8.68us, if timescale is 1ns
9 parameter MAX_DELAY_TOLERANCE = 20'hffff;
10 parameter SCAN_INTERVAL = 16'd30000;
11
12 // Registers
13 reg clk;
14 reg rst_n;
15 reg i_rx;
16 reg cpu_start;
17 reg next_instr;
18 reg step_execution;
19 reg sample_result;
20 reg sample_instr;
21 reg sample_flags;
22
23 // Wires
24 // wire instr_transmit_done;
25 // wire [7:0] max_addr_instr;
26 // wire halt;
27
28 wire [7:0] seg_valid;
29 wire [7:0] seg_value;
30 wire RGB1_BLUE;
31 wire RGB1_RED;
32 wire RGB2_BLUE;
33 wire RGB2_RED;
34 wire RGB2_GREEN;
35
36 // Task: Insert random clock delay
37 task random_clk_delay;
38   integer random_time;

```

```

38     begin
39         random_time = 10 + $random % 1000;
40         repeat (random_time) @(posedge clk);
41     end
42 endtask
43
44 // Task: UART send byte
45 task uart_send_byte(input [7:0] data);
46     integer i;
47     begin
48         // Start bit
49         i_rx = 0;
50         #(bit_period);
51         // Data bits (LSB first)
52         for (i = 0; i < 8; i = i + 1) begin
53             i_rx = data[i];
54             #(bit_period);
55         end
56         // Stop bit
57         i_rx = 1;
58         #(bit_period);
59         // There will be no interval between bytes
60     end
61 endtask
62
63 // Clock generation
64 initial begin
65     clk = 0;
66     forever
67         #5 clk = !clk;
68 end
69
70 // Asynchronous reset
71 initial begin
72     rst_n = 1;
73     #100 rst_n = 0;
74     repeat (MAX_DELAY_TOLERANCE + 3) @(posedge clk);
75     #10
76     rst_n = 1;
77 end
78
79 // DumpFile
80 initial begin
81     $dumpfile("top_0504.vcd");
82     $dumpvars(0, tb_TOP);
83 end
84

```

```

85
86
87
88 // Task: Wait for CPU start
89 task cpu_start_task;
90     begin
91         cpu_start = 0;
92         step_execution = 0;
93         #10 step_execution = 1;      // Edit step execution switch here
94         wait(RGB2_GREEN);
95         #1000 cpu_start = 1;
96     end
97 endtask
98
99 // Task: Execute instructions step by step
100 task execute_instructions;
101     begin
102         sample_result = 0;
103         sample_instr = 0;
104         sample_flags = 0;
105         next_instr = 0;
106         // Wait for CPU to start
107         wait(RGB2_BLUE);
108         repeat (10) @(posedge clk);
109         while (!RGB2_RED) begin
110             #10 sample_result = 1;
111             repeat (MAX_DELAY_TOLERANCE + 3) @(posedge clk);
112             #10 sample_result = 0;
113             repeat (100) @(posedge clk);
114             #10 sample_instr = 1;
115             repeat (MAX_DELAY_TOLERANCE + 3) @(posedge clk);
116             #10 sample_instr = 0;
117             repeat (100) @(posedge clk);
118             #10 sample_flags = 1;
119             repeat (MAX_DELAY_TOLERANCE + 3) @(posedge clk);
120             #10 sample_flags = 0;
121             repeat (100) @(posedge clk);
122             next_instr = 1;
123             repeat (MAX_DELAY_TOLERANCE + 3) @(posedge clk);
124             #10 next_instr = 0;
125             repeat (100) @(posedge clk);
126         end
127         $display("CPU halted at %t", $time);
128         // $display("ALU result: %h", alu_result);
129         // $display("ALU result high: %h", alu_result_high);
130     end
131 endtask

```

```

132
133 // Task: Check result after auto run
134 task check_result;
135 begin
136     sample_flags = 0;
137     sample_instr = 0;
138     sample_result = 0;
139     wait(RGB2_RED);
140     $display("CPU halted at %t", $time);
141     #10 sample_result = 1;
142     repeat (MAX_DELAY_TOLERANCE + 3) @(posedge clk);
143     #10 sample_result = 0;
144     repeat (100) @(posedge clk);
145     #10 sample_instr = 1;
146     repeat (MAX_DELAY_TOLERANCE + 3) @(posedge clk);
147     #10 sample_instr = 0;
148     repeat (100) @(posedge clk);
149     #10 sample_flags = 1;
150     repeat (MAX_DELAY_TOLERANCE + 3) @(posedge clk);
151     #10 sample_flags = 0;
152     repeat (100) @(posedge clk);
153     next_instr = 1;
154     repeat (MAX_DELAY_TOLERANCE + 3) @(posedge clk);
155     #10 next_instr = 0;
156     repeat (100) @(posedge clk);
157     // $display("ALU result: %h", alu_result);
158     // $display("ALU result high: %h", alu_result_high);
159 end
160
161 endtask
162 // Task: Reset and restart CPU, try send some byte again to test robustness
163 task reset_and_restart_cpu;
164 begin
165     #1000 cpu_start = 0;
166     #1000 rst_n = 0;
167     repeat (MAX_DELAY_TOLERANCE + 3) @(posedge clk);
168     #10 rst_n = 1;
169     random_clk_delay();
170     uart_send_byte(8'b01000001);
171     uart_send_byte(8'b00000000);
172 end
173 endtask
174
175 // Main initial block
176 initial begin
177     cpu_start_task();
178     if(step_execution) begin

```

```

179     execute_instructions();
180 end else begin
181     check_result();
182 end
183 reset_and_restart_cpu();
184 wait(RGB2_GREEN);
185 #10000 $finish;
186 end
187
188
189
190 TOP #((
191     .MAX_DELAY_TOLERANCE(MAX_DELAY_TOLERANCE),
192     .SCAN_INTERVAL(SCAN_INTERVAL)
193 )uut_cpu_top (
194     .CLK_100MHz(clk),           // 时钟信号
195     .START_CPU(cpu_start),     // 启动 CPU 信号
196     .STEP_EXECUTION(step_execution), // 单步执行信号
197     .BTNC(next_instr),         // 中间按钮 (下一条指令)
198     .BTNL(sample_result),       // 左按钮 (未使用)
199     .BTNR(sample_instr),        // 右按钮 (未使用)
200     .BTNU(rst_n),              // 上按钮 (复位)
201     .BTND(sample_flags),        // 下按钮 (未使用)
202     .RXD(i_rx),                // UART 接收数据
203     .SEG_VALID(seg_valid),      // 数码管有效信号 (未连接)
204     .SEG_VALUE(seg_value),       // 数码管显示值 (未连接)
205     .RGB1_RED(RGB1_RED),        // RGB LED 1 红色 (未连接)
206     .RGB1_BLUE(RGB1_BLUE),       // RGB LED 1 蓝色 (未连接)
207     .RGB2_RED(RGB2_RED),        // RGB LED 2 红色 (未连接)
208     .RGB2_BLUE(RGB2_BLUE),       // RGB LED 2 蓝色 (未连接)
209     .RGB2_GREEN(RGB2_GREEN)      // RGB LED 2 绿色 (未连接)
210 );
211
212
213 // RXD Data
214 initial begin
215     i_rx = 1; // idle state high
216     wait(rst_n == 0);
217     wait(rst_n == 1);
218     repeat (2* MAX_DELAY_TOLERANCE ) @ (posedge clk);
219     #10
220     // // Addr 1: LOAD IMMEDIATE 0
221     // uart_send_byte(8'b01000001);
222     // uart_send_byte(8'b00000000);
223     // // STORE IMMEDIATE 1
224     // uart_send_byte(8'b10000001);
225     // uart_send_byte(8'b10000000);

```

```

226 // // LOAD IMMEDIATE 1
227 // uart_send_byte(8'b01000001);
228 // uart_send_byte(8'b10000000);
229 // // STORE IMMEDIATE 2
230 // uart_send_byte(8'b10000001);
231 // uart_send_byte(8'b01000000);
232 // // LOAD 1
233 // uart_send_byte(8'b01000000);
234 // uart_send_byte(8'b10000000);
235 // // ADD 2
236 // uart_send_byte(8'b11000000);
237 // uart_send_byte(8'b01000000);
238 // // STORE IMMEDIATE 1
239 // uart_send_byte(8'b10000001);
240 // uart_send_byte(8'b10000000);
241 // // LOAD 2
242 // uart_send_byte(8'b01000000);
243 // uart_send_byte(8'b01000000);
244 // // ADD IMMEDIATE 1
245 // uart_send_byte(8'b11000001);
246 // uart_send_byte(8'b10000000);
247 // // STORE IMMEDIATE 2
248 // uart_send_byte(8'b10000001);
249 // uart_send_byte(8'b01000000);
250 // // SUB IMMEDIATE 100
251 // uart_send_byte(8'b00100001);
252 // uart_send_byte(8'b00100110);
253 // // STORE 3
254 // uart_send_byte(8'b10000001);
255 // uart_send_byte(8'b11000000);
256 // // JGZ 5
257 // uart_send_byte(8'b10100001);
258 // uart_send_byte(8'b10100000);
259 // // LOAD IMMEDIATE 1
260 // uart_send_byte(8'b01000000);
261 // uart_send_byte(8'b10000000);
262 // // HALT
263 // uart_send_byte(8'b11100000);
264 // uart_send_byte(8'b00000000);

265 // uart_send_byte(8'b01000001);
266 // uart_send_byte(8'b11111111);
267 // uart_send_byte(8'b00010001);
268 // uart_send_byte(8'b01111111);
269 // uart_send_byte(8'b10000001);
270 // uart_send_byte(8'b10000000);
271 // uart_send_byte(8'b01000001);
272

```

```
273 // uart_send_byte(8'b11111111);  
274 // uart_send_byte(8'b00010000);  
275 // uart_send_byte(8'b10000000);  
276 // uart_send_byte(8'b10000001);  
277 // uart_send_byte(8'b11000000);  
278 // uart_send_byte(8'b01000000);  
279 // uart_send_byte(8'b11000000);  
280 // uart_send_byte(8'b01000000);  
281 // uart_send_byte(8'b00100000);  
282 // uart_send_byte(8'b11100000);  
283 // uart_send_byte(8'b00000000);  
284  
285 // uart_send_byte(8'b01000001);  
286 // uart_send_byte(8'b10000000);  
287 // uart_send_byte(8'b10110001);  
288 // uart_send_byte(8'b11110000);  
289 // uart_send_byte(8'b00110001);  
290 // uart_send_byte(8'b01010000);  
291 // uart_send_byte(8'b10000001);  
292 // uart_send_byte(8'b10000000);  
293 // uart_send_byte(8'b10110001);  
294 // uart_send_byte(8'b11010000);  
295 // uart_send_byte(8'b10000001);  
296 // uart_send_byte(8'b01000000);  
297 // uart_send_byte(8'b01000000);  
298 // uart_send_byte(8'b10000000);  
299 // uart_send_byte(8'b01000000);  
300 // uart_send_byte(8'b01000000);  
301 // uart_send_byte(8'b11100000);  
302 // uart_send_byte(8'b00000000);  
303  
304 uart_send_byte(8'b01000001);  
305 uart_send_byte(8'b00100110);  
306 uart_send_byte(8'b10000001);  
307 uart_send_byte(8'b10000000);  
308 uart_send_byte(8'b01000001);  
309 uart_send_byte(8'b11000110);  
310 uart_send_byte(8'b00100000);  
311 uart_send_byte(8'b10000000);  
312 uart_send_byte(8'b10010001);  
313 uart_send_byte(8'b00110000);  
314 uart_send_byte(8'b01010001);  
315 uart_send_byte(8'b11000000);  
316 uart_send_byte(8'b10000001);  
317 uart_send_byte(8'b01000000);  
318 uart_send_byte(8'b11010000);  
319 uart_send_byte(8'b01000000);
```

```
320     uart_send_byte(8'b01100001);  
321     uart_send_byte(8'b11010000);  
322     uart_send_byte(8'b11000001);  
323     uart_send_byte(8'b10000000);  
324     uart_send_byte(8'b11100000);  
325     uart_send_byte(8'b00000000);  
326 end  
327  
328 endmodule
```

Listing 27: 含用户面 Testbench

```

1 set_property -dict {PACKAGE_PIN C4 IOSTANDARD LVCMOS33} [get_ports RXD]
2 set_property -dict {PACKAGE_PIN E3 IOSTANDARD LVCMOS33} [get_ports CLK_100MHz]
3 set_property -dict {PACKAGE_PIN M18 IOSTANDARD LVCMOS33} [get_ports BTNU]
4 set_property -dict {PACKAGE_PIN P18 IOSTANDARD LVCMOS33} [get_ports BTND]
5 set_property -dict {PACKAGE_PIN P17 IOSTANDARD LVCMOS33} [get_ports BTNL]
6 set_property -dict {PACKAGE_PIN M17 IOSTANDARD LVCMOS33} [get_ports BTNR]
7 set_property -dict {PACKAGE_PIN N17 IOSTANDARD LVCMOS33} [get_ports BTNC]
8 set_property -dict {PACKAGE_PIN G14 IOSTANDARD LVCMOS33} [get_ports RGB1_BLUE]
9 set_property -dict {PACKAGE_PIN N16 IOSTANDARD LVCMOS33} [get_ports RGB1_RED]
10 set_property -dict {PACKAGE_PIN M16 IOSTANDARD LVCMOS33} [get_ports RGB2_GREEN]
11 set_property -dict {PACKAGE_PIN R12 IOSTANDARD LVCMOS33} [get_ports RGB2_BLUE]
12 set_property -dict {PACKAGE_PIN N15 IOSTANDARD LVCMOS33} [get_ports RGB2_RED]
13 set_property -dict {PACKAGE_PIN T10 IOSTANDARD LVCMOS33} [get_ports {SEG_VALUE[0]}]
14 set_property -dict {PACKAGE_PIN R10 IOSTANDARD LVCMOS33} [get_ports {SEG_VALUE[1]}]
15 set_property -dict {PACKAGE_PIN K16 IOSTANDARD LVCMOS33} [get_ports {SEG_VALUE[2]}]
16 set_property -dict {PACKAGE_PIN K13 IOSTANDARD LVCMOS33} [get_ports {SEG_VALUE[3]}]
17 set_property -dict {PACKAGE_PIN P15 IOSTANDARD LVCMOS33} [get_ports {SEG_VALUE[4]}]
18 set_property -dict {PACKAGE_PIN T11 IOSTANDARD LVCMOS33} [get_ports {SEG_VALUE[5]}]
19 set_property -dict {PACKAGE_PIN L18 IOSTANDARD LVCMOS33} [get_ports {SEG_VALUE[6]}]
20 set_property -dict {PACKAGE_PIN H15 IOSTANDARD LVCMOS33} [get_ports {SEG_VALUE[7]}]
21 set_property -dict {PACKAGE_PIN J17 IOSTANDARD LVCMOS33} [get_ports {SEG_VALID[0]}]
22 set_property -dict {PACKAGE_PIN J18 IOSTANDARD LVCMOS33} [get_ports {SEG_VALID[1]}]
23 set_property -dict {PACKAGE_PIN T9 IOSTANDARD LVCMOS33} [get_ports {SEG_VALID[2]}]
24 set_property -dict {PACKAGE_PIN J14 IOSTANDARD LVCMOS33} [get_ports {SEG_VALID[3]}]
25 set_property -dict {PACKAGE_PIN P14 IOSTANDARD LVCMOS33} [get_ports {SEG_VALID[4]}]
26 set_property -dict {PACKAGE_PIN T14 IOSTANDARD LVCMOS33} [get_ports {SEG_VALID[5]}]
27 set_property -dict {PACKAGE_PIN K2 IOSTANDARD LVCMOS33} [get_ports {SEG_VALID[6]}]
28 set_property -dict {PACKAGE_PIN U13 IOSTANDARD LVCMOS33} [get_ports {SEG_VALID[7]}]
29 set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33} [get_ports START_CPU]
30 set_property -dict {PACKAGE_PIN L16 IOSTANDARD LVCMOS33} [get_ports STEP_EXECUTION]
31 # UART input
32
33 # Clock 100MHz
34 create_clock -period 10.000 -name sys_clk_pin -waveform {0.000 5.000} -add [get_ports
    CLK_100MHz]
```

```

36 # set_clock_groups -name async_group -asynchronous # -group [get_clocks sys_clk_pin] # -group [
37   get_clocks cpu_clk_pin]
38
39 # Buttons
40
41
42 # RGB Lights
43
44
45 # 7-segment Lights
46
47
48
49 # Switches
50
51 # set_property -dict {PACKAGE_PIN M13 IOSTANDARD LVCMOS33} [get_ports {i_addr_read[2]}]
52 # set_property -dict {PACKAGE_PIN R15 IOSTANDARD LVCMOS33} [get_ports {i_addr_read[3]}]
53 # set_property -dict {PACKAGE_PIN R17 IOSTANDARD LVCMOS33} [get_ports {i_addr_read[4]}]
54 # set_property -dict {PACKAGE_PIN T18 IOSTANDARD LVCMOS33} [get_ports {i_addr_read[5]}]
55 # set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports {i_addr_read[6]}]
56 # set_property -dict {PACKAGE_PIN R13 IOSTANDARD LVCMOS33} [get_ports {i_addr_read[7]}]
57 # set_property -dict {PACKAGE_PIN T13 IOSTANDARD LVCMOS33} [get_ports {key_in_money[0]}]
58 # set_property -dict {PACKAGE_PIN H6 IOSTANDARD LVCMOS33} [get_ports {key_in_money[1]}]
59 # set_property -dict {PACKAGE_PIN U12 IOSTANDARD LVCMOS33} [get_ports {key_in_money[2]}]
60 # set_property -dict {PACKAGE_PIN U11 IOSTANDARD LVCMOS33} [get_ports {key_in_money[3]}]
61 # # set_property -dict {PACKAGE_PIN V10 IOSTANDARD LVCMOS33} [get_ports {key_in_money[4]}]
62
63 set_property BITSTREAM.CONFIG.SPI_BUSWIDTH 4 [current_design]

```

Listing 28: 开发板引脚约束