计算机组织与结构 II：CPU 设计文档

李勃璘
吴健雄学院

版本：1.1
日期：2025 年 4 月 25 日

摘　要

目录

表格目录

1 概述

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

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

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

本文接下来的章节安排如下：

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

2 CPU 结构设计

2.1 总体架构

CPU 的总架构（包括内存、外设等）示意图可见图 1。

CPU 由控制单元（CU），逻辑运算单元（ALU），内存（Memory）和寄存器组（Registers）组成，除内存以外，其余单元由被 CU 生成的控制信号控制的数据通路（Data Path）连接。另外，MAR 和 MBR 分别还和地址总线、数据总线相连接，用于与内存交互。控制单元和内存都和控制总线相连接，用于与外部控制信号交互。为简单起见，CPU 的计算全部为 **16 位定点有符号数计算**。

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 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 1 中的全部寻址方式。由于给定的指令集高四位均空闲，使用最高位存储支持的寻址方式。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。

图 1: CPU 总体架构

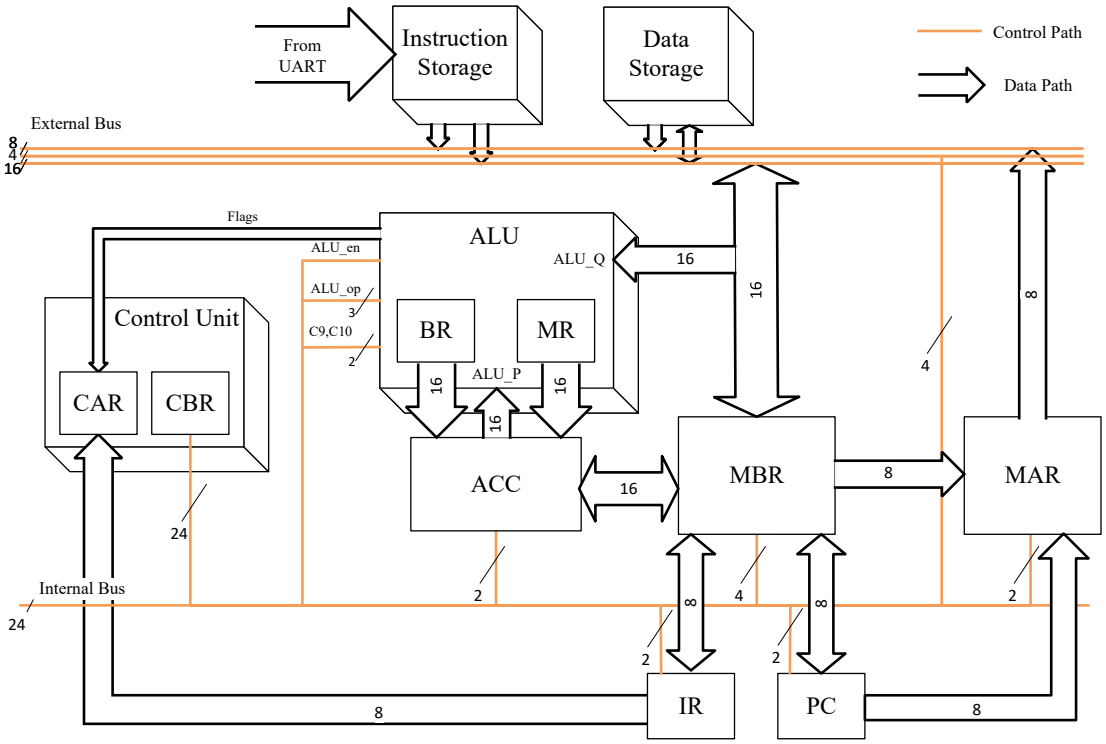


表 1: 指令集支持的寻址方式

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

2.2.3 指令集支持的指令

指令集共支持 13 条不同的指令，列于表 2。每一条指令包含一个指令码，使用二进制格式存储。[1]

表 2: 指令集包含指令及功能

助记符	指令码（低四位）	描述
*STORE X	0001	结果存入**数据地址** X
*LOAD X	0010	加载**数据地址** X
ADD X	0011	定点数加法
SUB X	0100	定点数减法
®JGZ X	0101	结果 > 0 时跳转至**指令地址** X
®JMP X	0110	无条件跳转至**指令地址** X
HALT	0111	暂停程序

续下页

[1]指令中含 * 的仅支持直接寻址，因为立即数寻址对这些指令无意义。含 ® 的仅支持立即数寻址。

表 2:（续表）指令集包含指令及功能

助记符	指令码（低四位）	描述
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

2.3 CPU 内部寄存器

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

表 3: CPU 内部寄存器的含义、总存储条数、单位位宽和数据解释格式

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

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

表 4: 状态寄存器列表

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

MF	Multiply Flag	ALU 此轮运算为乘法并溢出时置 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]。

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

表 5 描述了 ALU_{op} 与执行运算的对应关系。

表 5: ALU_{op} 与执行运算的对应关系

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

2.5 控制单元 CU

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

2.5.1 控制单元结构

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

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

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

2.5.2 微操作指令（Micro-Operations）

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

- **IF(Instruction Fetch):** 从指令存储器中取出指令，同时确定下一条指令地址（指针指向下一条指令）；

[2] 自增与 PC 赋值在设计中不引入 ALU。

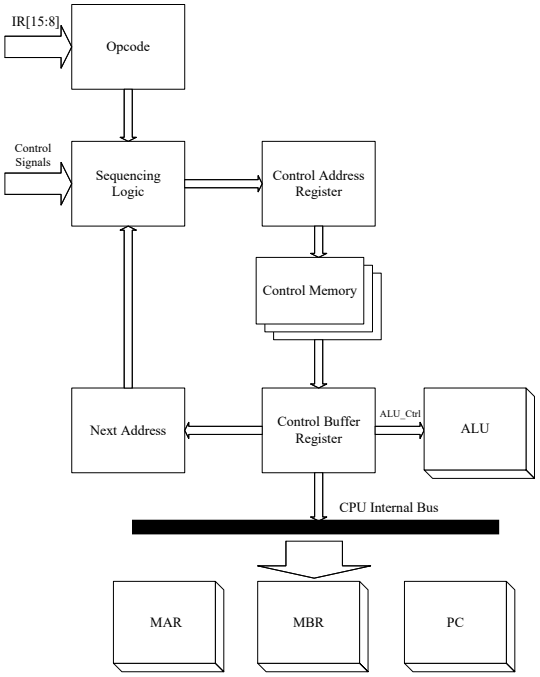


图 2: 控制单元结构示意图

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

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

表 6: CPU 微操作指令表

指令	机器码	EX	WB
IF	阶段	t_1:MAR ← PC; t_2: MBR ← Mem[MAR], PC ← PC+1	
ID	阶段	t_1:IR ← MBR; t_2: CU ← IR	
FO	阶段	MBR ← IR[7:0]	
IND	阶段	t_1:MAR ← MBR; t_2:MBR ← Mem[MAR]	
STORE X	0001	MAR ← MBR;	Mem[MAR] ← ACC
LOAD X	0010	无操作	ACC ← MBR
ADD X	0011	BR ← ACC + MBR	ACC ← BR
SUB X	0100	BR ← ACC - MBR	ACC ← BR
MPY X	1000	MR, BR ← ACC × MBR	ACC ← BR

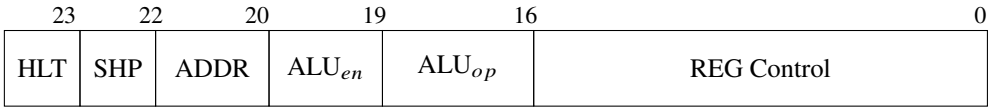
表 6:（续表）CPU 微操作指令表

指令	机器码	EX	WB
JGZ X	0101	判断：ZF=0 且 NF=0？	若满足，PC ← MBR，否则 NOP）
JMP X	0110	无操作	PC ← MBR
HALT	0111	无操作	暂停程序
AND X	1001	BR ← ACC AND MBR	ACC ← BR
OR X	1010	BR ← ACC OR MBR	ACC ← BR
NOT X	1011	BR ← NOT MBR	ACC ← BR
SHIFTR X	1100	BR ← ACC ⋙ X	ACC ← BR
SHIFTL X	1101	BR ← ACC ⋘ X	ACC ← 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



C_2（复用）：指令寄存器读、PC 自增

各控制字的意义如下：

- HLT(HALT)：全局暂停控制字，所有 CPU 内部单元停止工作。
- SHP(Store High Part)：存储乘法寄存器高位结果到指定数据内存地址 +1。
- ADDR(Address)：CU 内部控制字，共 2 位，指示下一步的地址为取指（11）/执行（01）/当前地址 +1（10）。
- ALU$_{en}$：ALU 使能控制字，允许 ALU 进行运算操作。
- ALU$_{op}$：ALU 运算控制字（3 位），指示 ALU 执行的 8 种运算类型。运算类型编码可见 ALU 部分。
- REG_Control：寄存器控制信号（16 位），每一位代表两个寄存器/总线之间的开关，对应关系见表 7。
- C_2：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。

表 7: 寄存器控制信号一览

控制信号位	源寄存器/单元	目的寄存器/单元
内部总线控制		
C_0	MAR	地址总线
		续下页

表 7:（续表）数据通路与控制信号一览

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

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

表 8: CPU 控制信号表

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

表 8: （续表）CPU 控制信号表			
指令	机器码	EX	WB
SHIFTR X	1100	C_6,C_7,ALU$_{en}$,ALU$_{op}$	C_9
SHIFTL X	1101	C_6,C_7,ALU$_{en}$,ALU$_{op}$	C_9

2.6 CPU 内部总线和外部总线

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

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

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

3 外围设备

3.1 用户端代码解释

目前采用**用户编写汇编代码 → 转换为 16 位机器码**的方式输入指令。用户可在文本编辑器中编写类汇编代码，而解释器负责将其解释为机器码。

以从 1 加到 100 的程序举例：

```
  LOAD IMMEDIATE 0 ; 初始化累加器为0 → ACC=0
  STORE 1          ; 存储到地址1（SUM变量）
  LOAD IMMEDIATE 1 ; 初始化计数器为1 → ACC=1
  STORE 2          ; 存储到地址2（i计数器）
LOOP: LOAD 0         ; 读取当前累加值 → ACC=SUM
  ADD 1            ; 加上当前计数器值 → ACC=SUM+i
  STORE 1          ; 更新累加值 → SUM=SUM+i
  LOAD 2           ; 读取计数器 → ACC=i
  ADD IMMEDIATE 1  ; 计数器自增 → ACC=i+1
  STORE 2          ; 更新计数器 → i=i+1
  SUB IMMEDIATE 100 ; 比较是否达到100 → ACC=i-100
  JGZ LOOP         ; 如果i<=100（即ACC<=0），继续循环
  HALT
```

代码最终将被解释为一串二进制比特流，解释服从：

- 地址占 1byte，Opcode 占 1byte；
- 含 IMMEDIATE 关键字的行，Opcode 的 MSB 为 1；
- 在代码解释的过程中，`LOOP` 应映射到相同行指令的地址。

3.2 UART 接收与指令内存写入设计

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

3.2.1 UART 接收逻辑

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

$$\text{CLK\_BAUD} = \frac{\text{CLK\_FREQ}}{\text{BAUD\_RATE}} = \frac{100\,000\,000}{115200} \approx 868 \tag{1}$$

采用常见的 **8N1** 格式传输，即每帧包括：

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

因此，每帧共 10 位，总计需要约：

$$\text{CLK\_FRAME} = 10 \times \text{CLK\_BAUD} = 10 \times 868 = 8680 \text{ cycles} \tag{2}$$

接收端采用中点采样策略，即在每位传输中间时刻（约第 434 个时钟周期）对数据位进行采样，以提升抗干扰能力。认为超过 300μsRX 端仍无新数据填入时，指令传输完成。

3.2.2 数据缓冲与存储结构

为保证串口数据完整接收，接收模块首先将每帧数据写入异步 FIFO 缓冲区。随后由控制逻辑从 FIFO 中读取数据，并写入开发板内部的块 RAM。

数据写入格式：

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

写入控制规则：

- 每一条指令都为 2byte 指令，对于没有操作数的情况，将操作数位置补零。
- RAM 地址从地址 0 开始顺序写入。
- 程序中如含有 `LOOP` 标签，将在 RAM 地址分配完毕后由软件在解析阶段回填其地址位置。

另外，传入 FPGA 的所有指令将存于单独的指令内存中，与 CPU 数据内存隔离开来。CPU 内存仅存数据，这符合用户编写的直观感受。每条存入内存的数据位宽为 16，即每个地址按顺序存放一条指令。地址从 1 开始依次递增，防止复位时地址位初始化为 0 导致出错。指令写入在 CPU 开机之前，写入成功后开发板亮蓝灯。若在 CPU 运行时没有指令，则开发板亮红灯。

本模块通过串口实现了简洁的二进制指令数据装载方式，降低了外设复杂性，为后续的控制单元译码与执行单元操作提供了明确的数据支持。

3.3 数据内存

数据内存（RAM）存储 CPU 保存的数据。内存的大小为 512 Byte，每条存入内存的数据位宽为 16，共能存入 256 条数据。数据内存初始为空，起始写入地址为 0，采用 Little Endian 写入方式[3]。

CPU 与内存（RAM）通过三条总线交互，分别为**控制总线**、**地址总线**和**数据总线**。控制总线中的控制信号决定在这个周期中内存的读/写状态，是否向数据总线写入，同步时序等功能。内存通过读取地址总线决定写入内存中的地址，通过读取数据总线决定写入指定地址中的数据。关于总线的具体配置见 2.6。数据内存中不存放待执行指令，防止数据通路和指令通路发生冲突。

3.4 用户交互设计

该部分描述用户与 FPGA 的交互接口（按钮、按键等）以及看到的运行状态信息与结果显示设计。

4 核心模块设计

4.1 时钟、复位与停止信号

CPU 由**全局同步时钟**控制，时钟主频为 50MHz。除复位信号外，所有控制逻辑与计算逻辑全部在时钟上升沿进行。UART 传输部分使用 100MHz 的时钟主频。

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**暂停**状态。与复位不同的是，此时所有寄存器不清空，但所有通路断开。在模块中使用 `enable` 信号标识（低电平有效）。恢复程序运行的方法是全局复位或继续运行信号（绑定 FPGA 的按键）。当该按键被按下时，`enable` 信号恢复为 1。

4.2 UART 传输与指令内存

指令写入通过 Python 脚本（附录 A.1 ）完成，其可以根据用户输出一串 UART 格式的比特流。用户可通过 PC 上的串口调试设备连接 UART 端口进行传输。

功能块基本信息：

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

功能块外部接口：

表 9: 指令内存模块外部接口

信号名	方向	位宽	描述
`i_clk_uart`	输入	1	时钟信号（100MHz）

[3]即高位存储于高地址，低位存储于低地址。

（续表）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（50MHz）
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 ROM 地址
o_instr_read	输出	16	指令输出信号
o_instr_transmit_done	输出	1	（安全的）指令完成传入标志
o_max_addr	输出	8	最大地址输出

4.2.1 UART 模块

模块基本信息：

- 模块名：UART
- 最新更新日期：4.13
- 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	系统时钟信号
i_rst_n	输入	1	全局复位信号
i_rx	输入	1	UART 接收引脚
o_data	输出	8	接收到的一帧数据
o_valid	输出	1	数据有效标志，高电平表示 o_data 已生成
o_clear_sign	输出	1	表示 UART 输入结束（第一次输入结束后 0.5 秒内无新的输入）

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。将每两个读出的 UART 数据拼成 2 字节的指令输出给 BRAM。

模块外部接口：

表 11: FIFO 模块外部接口

信号名	方向	位宽	描述
i_rst_n	输入	1	异步复位信号，低有效
i_clk_wr	输入	1	写时钟信号，UART 使用的 100MHz 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，主系统使用的 50MHz 时钟
o_data_bram	输出	16	两个 UART 字节拼接后的数据，写入 BRAM
o_addr_bram	输出	8	BRAM 写入地址，从 0 开始自增
o_wr_en_bram	输出	1	BRAM 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）

备注： 设计思路参照文献[1]。

4.2.3 指令 BRAM 模块

模块基本信息：

- 模块名：BRAM_INSTR
- 最新更新日期：4.13
- 是否经过测试：是

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

模块外部接口：

表 12: 指令 BRAM 模块外部接口

信号名	方向	位宽	描述
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 BRAM
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 BRAM 中读取的指令内容

4.2.4 仿真测试

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。控制单元通过微操作指令和控制信号控制整个系统。整个控制单元的结构如图 2 所示。

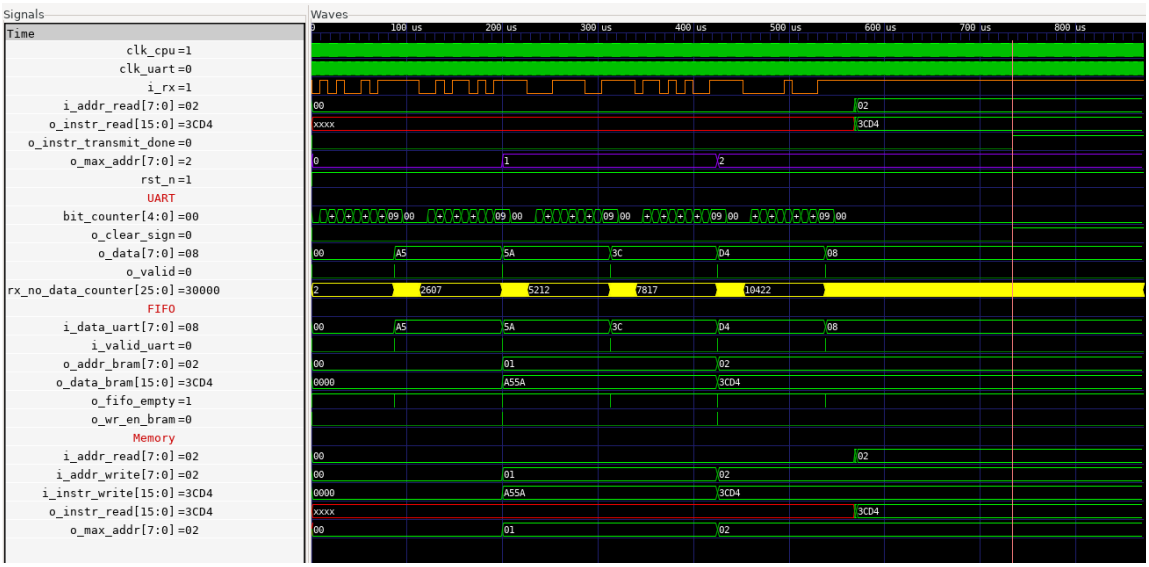


图 4: 指令内存部分仿真

4.3.1 Control Memory

模块基本信息：

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

模块功能： 全局复位时存储 CPU 的水平微操作指令。微操作指令表参考表 8 。

模块外部接口：

表 13: Control Memory 模块外部接口

信号名	方向	位宽	描述
`i_clk`	输入	1	系统时钟信号，驱动读写操作
`i_rst_n`	输入	1	全局复位，此时重新写入全部指令
`i_addr_read`	输入	8	要读取的指令地址
`o_signal_read`	输出	16	从 BRAM 中读取的指令内容

4.3.2 Sequencing Logic

模块基本信息：

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

模块功能： 根据 CBR 反馈的“下一条地址”逻辑以及控制信号，综合判断出下一条指令。

模块外部接口：

表 14: Sequencing Logic 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
i_rst_n	输入	1	全局复位
i_ctrl_CBR	输入	8	来自 CBR 的下一跳地址
o_signal_read	输出	16	从 BRAM 中读取的指令内容

4.4 基础模块设计

4.4.1 ALU

ALU 运算结果存放于 MR 寄存器和 ACC 寄存器中。其中 MR 寄存器存放乘法运算的高位结果，ACC 寄存器存放乘法运算的低位结果。

4.4.2 MAR

4.4.3 MBR

4.4.4 PC

4.4.5 IR

4.4.6 ACC

4.4.7 数据 RAM

4.4.8 Control Unit

参考表 7，确定指令集中的每一条指令对应的控制信号，并存储到 CU 的内部寄存器中，即可完成 CU 的主要功能设计。

5 仿真验证

5.1 时延分析

5.2 激励设置

6 FPGA 实现

6.1 用户输入端

采用第一个测试样例进行测试。

$$1 + 2 + \cdots + 99 + 100 = 5050$$

编写源程序如下：

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

```
    LOAD IMMEDIATE 100
    STORE 3
LOOP: LOAD 1
    ADD 2
    STORE 1
    LOAD 2
    ADD IMMEDIATE 1
    STORE 2
    LOAD 2
    SUB 3
    JGZ LOOP

    HALT
```

参考文献

[1] 菜鸟教程. Verilog FIFO 设计[EB/OL]. 2020. https://www.runoob.com/w3cnote/verilog2-fifo.html.

[2] 赖兆磬. 基于 FPGA 流水线 CPU 的设计与实现[D]. 桂林电子科技大学, 2008.

A 完整设计代码

A.1 汇编程序处理 Python 脚本

```python
import re
import serial
import os
# memonics
MEMONICS = {
    "STORE": 0x01,
    "LOAD": 0x02,
    "ADD":  0x03,
    "SUB":  0x04,
    "JGZ":  0x05,
    "JMP": 0x06,
    "HALT": 0x07,
    "MPY": 0x08,
    "AND": 0x09,
    "OR":  0x10,
    "NOT": 0x11,
    "SHIFTR": 0x12,
    "SHIFTL": 0x13
}

def parse_assembly(lines):
    machine_code = []
    labels = {}
    pending = []

    # First pass: find labels
    addr = 0
    for line in lines:
        line = line.split(';')[0].strip() # clear comments
        if not line:
            continue
        if ':' in line:
            label, rest = map(str.strip, line.split(':', 1))
            labels[label] = addr
            if rest:
                addr += 1
        else:
            addr += 1

    # Second pass: generate code
    addr = 0
    for line in lines:
        line = line.split(';')[0].strip()
```

```
        if not line:
            continue
        if ':' in line:
            parts = line.split(':', 1)
            line = parts[1].strip()
            if not line:
                continue

        tokens = line.split()
        if not tokens:
            continue

        instr = tokens[0]
        immediate = False

        if instr in ["HALT","SHIFTR","SHIFTL"] : # No operand, fill with 0
            opcode = MEMONICS[instr]
            operand = 0x00
        else:
            if len(tokens) < 2:
                raise ValueError(f"Missing operand in line: {line}")

            if tokens[1] == "IMMEDIATE":
                immediate = True
                operand_str = tokens[2]
                opcode = MEMONICS[instr] | 0x80 # MSB = 1
            else:
                operand_str = tokens[1]
                opcode = MEMONICS[instr] # MSB = 0

            if operand_str.isdigit():
                operand = int(operand_str)
            elif operand_str in labels:
                operand = labels[operand_str]
            else:
                try:
                    operand = int(operand_str, 0) # Support 0x form operand
                except:
                    raise ValueError(f"Unknown operand: {operand_str}")

        if operand < 0 or operand > 255:
            raise ValueError(f"Operand out of 8-bit range: {operand}")

        machine_code.append((opcode << 8) | operand)
        addr += 1

    return machine_code
```

```python


def assemble_to_bytes(code: list[int]) -> bytearray:
    result = bytearray()
    for word in code:
        result.append((word >> 8) & 0xFF) # opcode
        result.append(word & 0xFF)    # operand
    return result

def send_to_serial(bitstream:str) -> None:
    # must run on Linux system
    # FPGA Config: Baud rate = 115200, 8N1 Transmission
    write_port = '/dev/ttyUSB1'
    ser = serial.Serial(
    port= write_port,
    baudrate= 115200,
    timeout=1,
    bytesize=8,
    parity= "N",
    stopbits=1
    )

    # 向 FPGA 发送数据
    for item in bitstream:
        ser.write(item.encode()) # 将字符串转换为字节并发送
        print(f"Write bit {item} to serial port {write_port}\n")
    print("Write successfully")
    # # 读取来自 FPGA 的数据
    # response = ser.readline() # 读取一行数据（假设 FPGA 发送数据是以换行符结尾）
    # print(f"Received from FPGA: {response.decode().strip()}")

    # 关闭串口
    ser.close()

def main():
    os.chdir("./designs/input_src")
    with open('add_one_to_hundred.txt', 'r') as file:
        lines = file.readlines()

    machine_words = parse_assembly(lines)
    binary = assemble_to_bytes(machine_words)

    # 打印每条机器码（16位）和最终二进制流
    print("Machine Code:")
    for i, word in enumerate(machine_words):
        print(f"{i:02}: {word:04X}")

```

```
    print("\nGenerated Binary Bitstream:")
    print(" ".join(f"{b:08b}" for b in binary))
    bitstream = [f"{b:08b}" for b in binary]

    send_to_serial(bitstream)

main()
```

Listing 1: write_bistream.py