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 11 日

摘　要

本设计文档详细阐述了一款基于五级流水线的 CPU 体系结构及其 Verilog 实现。CPU 采用取指（IF）、译码（ID）、执行（EX）、访存（MEM）、写回（WB）五级流水线，以提高指令执行效率。文档首先介绍了 CPU 的总体架构，包括时钟与复位机制、关键存储单元、内部数据通路与控制信号，并详细说明了指令集架构及其编码格式。随后，针对流水线执行过程中可能出现的数据冒险、控制冒险、流水线暂停等问题，提出了旁路（Forwarding）、分支预测（Branch Prediction）、冒险检测（Hazard Detection）等优化方案，并给出了相应的 Verilog 设计。最后，文档分析了 CPU 与内存、总线及外部控制信号的交互方式，并探讨了 Verilog 在 FPGA 上的实现方案。该设计通过流水线优化与高效控制信号管理，提升了 CPU 的吞吐率，为后续硬件优化和扩展提供了良好的基础。

目录

表格目录

1 概述

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.[1]

2 CPU 结构设计

2.1 总体架构

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**。

这里需要一张图!!!

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。

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

[1] 手搓 CPU 是人类文明的伟大工程——沃兹基硕德

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 16 进制格式存储。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。[2]

表 2: 指令集包含指令及功能（直接寻址下）

助记符	指令码（低四位）	描述
*STORE X	0001	[X] ← ACC
*LOAD X	0010	ACC ← [X]
ADD X	0011	ACC ← ACC + [X]
SUB X	0100	ACC ← ACC - [X]
JGZ X	0101	ZF 和 NF 均为 0 ? PC ← X : PC ← PC + 1
JMP X	0110	PC ← X
®HALT	0111	Stop
MPY X	1000	MR, ACC ← ACC * [X]，MR 用于存储高 16 位
AND X	1001	ACC ← ACC & [X]
OR X	1010	ACC ← ACC | [X]
NOT X	1011	ACC ← ~[X]
® SHIFTR	1100	ACC ← ACC >>> 1，算术右移
® SHIFTL	1101	ACC ← ACC <<< 1，算术左移

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	寄存器，存储 ALU 计算结果	1	32	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	ALU
MR	乘法寄存器，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	未定	14	控制信号	CU
CAR	控制地址寄存器，指向当前执行的微指令	1	4	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	14	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是 0，CU 可以直接通过状态寄存器中的 ZF（Zero Flag）寄存器直接进行判断。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。

[2]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 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	ALU 运算结果发生溢出时置 1
NF	Negative Flag	ALU 运算结果为负数时置 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[3]。它在 EX 阶段通过 MBR 获取运算的第二个数据，计算方式受状态寄存器影响，计算结果将存入 BR 寄存器，等待 WB 阶段写回 ACC 寄存器中（若有乘法则可能存入 MR 寄存器）。

ALU 与外围寄存器的控制通路见第 2.5.1 节。为了让 ALU 内部获取要执行的运算操作，ALU 受到 `ALU_Ctrl` 端口中传入的操作码低四位控制。这四位决定 ALU 执行什么运算。ALU 参与执行的所有运算都列于表 5。如表所见，ALU 还会在算术的同时更新 Flag 值，该值对用户和 Control Unit 都公开。用户部分的配置，详见用户交互部分（第 3.3 节）。

表 5: 各指令对标志位的影响

操作码低四位	助记符	ZF	CF	OF	NF
0001	ADD	为 0 时置 1	0	溢出置 1	负数置 1
0010	SUB	为 0 时置 1	0	溢出置 1	负数置 1
0011	MPY	为 0 时置 1	0	0	负数置 1
0100	AND	为 0 时置 1	0	0	负数置 1
0101	OR	为 0 时置 1	0	0	负数置 1
0110	NOT	为 0 时置 1	0	0	负数置 1
0111	SHIFTR	为 0 时置 1	最低位	0	负数置 1
1000	SHIFTL	为 0 时置 1	最高位	0	负数置 1

2.5 CPU 内部数据通路、控制信号与微操作指令（Micro-Operations）

CPU 中总线、寄存器、内存和 ALU 等关键器件需要以合理的时序执行不同的指令，而这需要数据通路和控制通路共同实现。数据通路负责在各个器件中传递数据，而控制通路负责控制数据通路的开关。因此，在本节中，先定义 CPU 需要的数据通路，再进一步定义控制数据通路的控制信号，最后描述承载控制信号的微指令。

2.5.1 数据通路与控制信号

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。该 CPU 共有 **17** 位控制信号。[4]

[3] PC 自增与 PC 赋值在设计中不引入 ALU。

[4] CU 内部的数据通路不需要受到微指令的控制，因此不需要在此列出。请参照模块设计部分。

表 6: 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
基本控制		
0	MAR	地址总线
1	PC	MBR
2	PC	MAR
3	MBR	PC
4	MBR	IR
5	数据总线	MBR
6	MBR	ALU_Q
7	ACC	ALU_P
8	MBR	MAR
9	BR	ACC
10	BR	MR
11	MBR	ACC
12	ACC	MBR
13	MBR	数据总线
14	IR	CU
15	IR[7:0]	MBR
16	IR[11:8]	ALU_Ctrl
流水线控制		
17		

2.5.2 微操作指令（Micro-Operations）

为了实现指令集中所有指令，需要将指令集的指令分解为多步微操作指令。为了和后续流水线部分对齐，在微操作指令的设计上，也分为和六级流水线相同的步骤。

在本设计中，采用水平微指令（Horizontal Micro-operation）设计。这意味着每一个微操作指令携带所有控制信号位的开关和下一个微操作指令的地址。

所有用逗号隔开的指令均**从左到右顺序执行**，不能并行执行。所有的微操作指令见表 7。

表 7: CPU 微操作指令表

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

表 7:（续表）CPU 微操作指令表

指令	机器码	FO	EX	WB
MPY X	1000	MBR ← IR[7:0];	BR ← ACC × MBR	ACC ← BR[15:0]; MR ← BR[31:16]
JGZ X	0101	读取条件标志寄存器 MBR ← IR[7:0];	判断：ZF=0 且 NF=0?	若满足，PC ← MBR， 否则 PC ← PC（不变）
JMP X	0110	MBR ← IR[7:0];	无操作	PC ← MBR
HALT	0111	无操作	无操作	enable ← 0，停止流水线
AND X	1001	MBR ← IR[7:0];	BR ← ACC AND MBR	ACC ← BR
OR X	1010	MBR ← IR[7:0];	BR ← ACC OR MBR	ACC ← BR
NOT X	1011	MBR ← IR[7:0];	BR ← NOT MBR	ACC ← BR
SHIFTR	1100	无操作	BR ← ACC ⋙ 1	ACC ← BR
SHIFTL	1101	无操作	BR ← ACC ⋘ 1	ACC ← BR

表 8: CPU 控制信号表

指令	机器码	FO	EX	WB
IF	阶段		C_2;C_0,C_5	
ID	阶段		C_4, C_{14}	
IND	阶段		C_0,C_5,C_8	
STORE X	0001	C_{15}	C_8	C_0,C_{12},C_{13}
LOAD X	0010	C_{15}	无操作	C_{11}
ADD X	0011	C_{15}	C_6,C_7,C_{16}	C_9
SUB X	0100	C_{15}	C_6,C_7,C_{16}	C_9
MPY X	1000	C_{15}	C_6,C_7,C_{16}	C_9,C_{10}
JGZ X	0101	读取条件标志寄存器 C_{15}	判断：ZF=0 且 NF=0?	若满足，C_3 否则 PC ← PC+1
JMP X	0110	C_{15}	无操作	C_3
HALT	0111	无操作	无操作	enable ← 0，停止流水线
AND X	1001	C_{15}	C_6,C_7,C_{16}	C_9
OR X	1010	C_{15}	C_6,C_7,C_{16}	C_9
NOT X	1011	C_{15}	C_6,C_{16}	C_9
SHIFTR	1100	无操作	C_7,C_{16}	C_9
SHIFTL	1101	无操作	C_7,C_{16}	C_9

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 RAM 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。

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（通过 Vivado IP 核生成）。

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，即使其可能没有操作数。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 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。

3.4 数据 RAM

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。

内存支持总线读写，并受三条总线控制：地址总线、数据总线和控制总线。**控制总线**中的外部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。CPU 与内存（RAM）通过两条总线交互，分别为**地址总线**和**数据总线**。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 3.5。

采用 Harvard 结构，内存中不存放待执行指令。待执行指令存放于另一块内存中，防止数据通路和指令通路发生冲突。

3.5 总线与外部控制信号

对于指令 RAM 和数据 RAM，各自包含一条与 CPU 的地址总线和数据总线。前者的数据总线是单向通往 CPU 的，而后者则为双向总线。

3.5.1 地址总线

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。由于其为单向总线，仅需内存侧读使能信号与 CPU 侧 MAR 的控制信号控制即可，无需复用。

3.5.2 数据总线

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。数据总线采用分时复用的方式进行设计。如果 MEM 和 WB 在同一时钟下，会出现流水线冲突，需要考虑是否引入旁路。

3.5.3 控制总线与外部控制信号（需要等待流水线设计好）

外部控制信号是一组单比特信号，通过控制总线控制 CU 和 RAM 的行为，它们受流水线周期的控制置 0 或置 1。CU 和 RAM 通过内部映射决定监视哪些位的信号。外部控制信号主要包括以下功能：

- RAM 读写控制；
- 流水线控制信号。

所有外部控制信号列于表 9。

表 9: 外部控制信号一览

控制信号位/类型	别名	有效模块	高电平时作用	低电平时作用
RAM 读写控制				
0	MemoryWrite	RAM	RAM 写数据总线	无
1	MemoryRead	RAM	RAM 读数据总线和地址总线	无
分支预测				
2	BranchTaken	CU	执行 Branch	顺序执行
3	Jump	CU	执行 Jump	顺序执行
流水线控制				
4	PipelineStall	CU	流水线暂停（如数据冒险、存储器访问延迟）	无
5	PipelineFlush	CU	流水线清空（如错误预测、异常发生）	无

4 流水线架构与优化策略

4.1 总体架构

为了加速 CPU 的指令执行速度，采用 **6 级同步流水线**完成 CPU 执行指令的全流程。分别为：取指令（IF），指令译码（ID），取操作数（FO），间址（IND），指令执行（EX）和写回寄存器（WB）六个阶段。流水线的六个阶段如下所示。

$$\text{IF} \rightarrow \text{ID} \rightarrow \text{FO} \rightarrow \text{IND} \rightarrow \text{EX} \rightarrow \text{WB}$$

流水线各阶段的主要工作如下：

- **IF(Instruction Fetch):** 从指令存储器中取出指令，同时确定下一条指令地址（指针指向下一条指令）；
- **ID(Instruction Decode):** 翻译指令，同时让计算机得出要使用的寄存器，得出寻址方式，亦或者（转移指令）是给出转移目的寄存器与转移条件；
- **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。

因此，对于每一条指令，都会经历上述六个指令周期中的一部分，而由表 8 可知，其中 IF 和 ID 部分所有指令公用相同的微操作指令。

4.2 流水线寄存器

为了使多条指令各阶段的数据在转换时可以保存，避免出现输出数据被覆盖的情况，需要设立 5 个流水线寄存器贮存上一阶段的输出。为方便起见，这些寄存器命名统一为“阶段 1_ 阶段 2_SHIFT_REG”的方式。

每一个寄存器所贮存的内容应当至少支撑起下一阶段的输入，基于此原则，可以给出每个转移寄存器所需贮存的内容如表 10。

表 10: 流水线寄存器功能一览

转移寄存器	贮存内容（高位到低位）	位宽
IF_ID_SHIFT_REG	PC/IR	24
ID_FO_SHIFT_REG	PC	MBR
FO_IND_SHIFT_REG	PC	MAR
IND_EX_SHIFT_REG	MBR	16
EX_WB_SHIFT_REG	MBR	IR
WB_OUT_SHIFT_REG	指令的输出结果：ACC/MR/Flags	36

4.3 流水线冒险

包含三种冒险情况：结构冒险、数据冒险、控制冒险（分支冒险）。[1]

结构冒险，即**某指令引用了前一次的运算结果，但其结果值还未产生**。对于这种情况，采用旁路（Forwarding）技术解决。

4.4 流水线优化：分支预测

在本设计中，为了减少分支指令（JMP，JGZ）带来的流水线 Flush 和延迟，采用 1 比特分支预测进行流水线优化。

分支预测发生在分支指令执行至 **ID** 阶段时。CU 在译码后发现其为跳转类指令，就会向 Branch Predictor 发送一个信号，要求进行分支预测。分支预测的结果决定接下来流水线中 IF 阶段读取指令的地址，但若在 EX 阶段发现预测结果和实际结果不一致，则对已抓取的所有指令清空（Flush）。分支预测的具体步骤如算法 1。

Algorithm 1 1-bit 分支预测

Note: 使用 1-bit 预测器预测分支是否跳转
Input: $PC, BranchTaken$
Output: $NextPC$
Internal Registers: $PredictorBit, BranchTarget$
procedure BRANCH PREDICTION
    if $PredictorBit == 1$ **then**
        $PredictedTaken \leftarrow$ True
        $NextPC \leftarrow BranchTarget$
    else
        $PredictedTaken \leftarrow$ False
        $NextPC \leftarrow PC + 1$
    end if
end procedure
procedure BRANCH RESOLUTION
    if $BranchTaken \neq PredictedTaken$ **then** ▷ 预测错误
        $Flush \leftarrow 1$ ▷ 清空错误指令
        $NextPC \leftarrow$ **if** $BranchTaken$ **then** $BranchTarget$ **else** $PC + 1$
    end if
    Update Predictor:
    $PredictorBit \leftarrow BranchTaken$ ▷ 更新 1-bit 预测器
end procedure

5 模块设计

5.1 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。

5.2 UART 传输与指令写入

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。

5.2.1 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：否

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 11: 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）

5.2.2 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：否

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 12: 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）

设计思路： 参照文献 [2]。

5.2.3 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：否

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

模块外部接口：

表 13: 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

5.3 基础模块设计

5.3.1 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。

5.3.2 MAR

5.3.3 MBR

5.3.4 PC

5.3.5 IR

5.3.6 ACC

5.3.7 数据 RAM

5.3.8 Control Unit

参考表 6，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。

5.4 流水线支持

6 仿真验证

6.1 时延分析

6.2 并行计算加速比（Speed-up Factor）分析

6.3 激励设置

7 FPGA 实现

7.1 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

```

```
17      HALT
```

References

[1] Ansis11. 基于 *RISC-V* 架构-五级流水线 *CPU*》知乎专栏，访问于 2025 年 4 月 10 日. 2022. URL: https://zhuanlan.zhihu.com/p/453232311.

[2] 菜鸟教程. *Verilog FIFO* 设计. 访问于 2025 年 4 月 13 日. 2020. URL: https://www.runoob.com/w3cnote/verilog2-fifo.html.

[3] 赖兆磬. “基于 FPGA 流水线 CPU 的设计与实现”. MA thesis. 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

# 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


if __name__ == "__main__":
    # 示例程序：从1加到100
    asm_code = """
      LOAD IMMEDIATE 0
      STORE 1
      LOAD IMMEDIATE 1
      STORE 2
    LOOP: LOAD 1
      ADD 2
      STORE 1
      LOAD 2
      ADD IMMEDIATE 1
      STORE 2
      SUB IMMEDIATE 100
      JGZ LOOP
      HALT
    """

    lines = asm_code.strip().split('\n')
    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("机器码:")
    for i, word in enumerate(machine_words):
        print(f"{i:02}: {word:04X}")

    print("\n二进制比特流:")
    print(" ".join(f"{b:08b}" for b in binary))
    print("\nUART传输流:")
    print("".join(f"0{b:08b}1" for b in binary))
```

Listing 1: convert.py