# 流水线处理器设计实验

## 实验六 实现R型运算类指令的理想流水线设计实验

### 实验目的

1. 掌握R型运算类指令的数据通路。
2. 掌握经典单发射五级流水线的设计方法。
3. 掌握流水线CPU设计的编程基本框架。

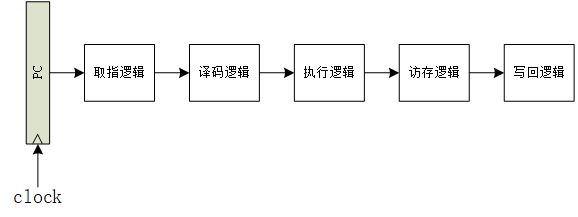
### 实验原理与实验内容

#### 单周期CPU与流水线CPU

##### 结构差异

RISC-V单周期CPU中所有指令的执行时间均为一个等长的时钟周期。为保证每条指令都来得及完成，只能以指令系统中最长的指令周期作为系统的时钟周期。对执行时间短的指令而言，时钟周期里剩余的时间都被浪费，而短指令在整个指令系统里占大部分，因此单周期CPU的效率很低。现代处理器设计不采用单周期方式，取而代之的是多周期设计方式。在多周期CPU设计中流水线CPU设计是目前的主流技术。

电路流水化设计的初衷是：**缩短时序器件之间组合逻辑关键路径的时延，在不降低电路处理吞吐率的情况下提升电路的时钟频率。**将一段组合逻辑电路按照功能划分为若干阶段，在各功能段的组合逻辑电路之间插入时序器件，通常是触发器。前一阶段的组合逻辑电路输出接入时序器件的输入，后一阶段的组合逻辑电路输入来自这些时序器件的输出。



单周期CPU逻辑划分

将电路流水化的难点在于分析单周期CPU中的组合逻辑电路应该划分为多少个功能段？各个功能段分别包含哪些功能？图7-1展示了单周期CPU执行指令的五级功能逻辑划分，图中每一段逻辑完成指令功能的一个部分，多个逻辑段连贯起来就完成了一条指令的功能。

事实上切分功能段的设计决策需要结合CPU产品的性能（含主频）、功耗、面积指标以及具体采用的工艺特性来考虑。为避免内容过多过细过深，本书实验采用经典的单发射五级流水线设计，划分的五级流水从前往后依次为：取指阶段（Fetch）、译码阶段（Decode）、执行阶段（Execute）、访存阶段（Memory）和写回阶段（WriteBack）。

1. 取指阶段的功能是：以PC为地址，从存储器中将指令取到CPU中。
2. 译码阶段的功能是：解析指令；生成控制信号；从通用寄存器堆中读出源操作数。
3. 执行阶段的功能是：对源操作数进行算术逻辑运算；对于访存指令则是计算访存地址。
4. 访存阶段的功能是：从存储器中读数，或将操作数写入存储器。
5. 写回阶段的功能是：将结果写入通用寄存器。

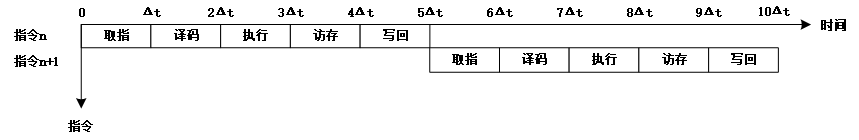
在图7-1所示各段之间加入触发器作为流水线缓存，图7-2展示了RISC-V流水线的逻辑结构。



五级流水线CPU逻辑结构

所有部件采用同一个系统时钟clock来进行同步。在每个时钟clock的上升沿，各段逻辑功能部件处理完毕的数据会被锁存到下一级的流水线缓存中，作为下一段的输入数据，指令执行进入到下一阶段。clock的频率取决于流水线缓存两级间的最大逻辑延迟。

##### 性能差异



单周期CPU时空图



五级流水线CPU时空图

图7-3给出了单周期CPU的时空图。图中表示系统时钟周期，每条指令执行需要5个时钟周期，即指令周期是。1个时钟周期时长1个，每5个提交1条指令，单周期CPU的IPC是1÷5=0.2。算出运行n条指令花费的总时间为。图7-4给出了RISC-V理想的五级流水线CPU时空图。在理想情况下，当流水线满载运行时，每个时钟周期提交1条指令，也就是CPU的IPC为1。流水线完成n条指令的总时间为。当n趋近于时，相比单周期CPU执行n条指令花费的时间，五级流水线的加速比，即理想的五级流水线CPU的执行效率是单周期CPU的5倍。

#### 运算类指令的数据通路的前端设计

CPU不但要完成运算，也要能维持自身的状态，所以CPU中既有组合逻辑电路又有时序逻辑电路。CPU输入、运算、存储和输出的数据都在组合逻辑电路和时序逻辑电路上流转，这些逻辑电路被称为**数据通路**。因此，设计CPU的数字逻辑电路，首要工作就是设计数据通路。因为数据通路中包括有多路选择器、时序逻辑器件，所以还应该有相应的控制信号，产生这些控制信号的逻辑被称为**控制逻辑**。从宏观的视角来看，设计一个CPU就是设计“数据通路+控制逻辑”。

根据指令系统规范中的定义设计“数据通路+控制逻辑”的基本方法是：对指令系统中定义的指令逐条进行功能分解，得到一系列操作和操作对象。这些操作和操作对象必然对应其各自的数据通路。又因为指令间存在一些相同或相近的操作和操作对象，所以可以只设计一套数据通路供多个指令公用。对于确实存在差异无法共享数据通路的情况，只能各自设计一套，再用多路选择器进行选择。加法指令是运算类指令的典型代表，以add指令为例，介绍如何遵循该一般性方法来设计运算类指令的数据通路。

首先，分析实现add指令需要有哪些数据通路部件？

执行指令的前提是从存储器中读出add指令的指令码送到CPU中。

①在支持虚拟存储器的系统中，指令的地址是**虚拟地址**，需要先进行虚实地址转换，得到**物理地址**才能访问指令存储器InstMEM从而取指令。

②在不支持虚拟存储器的系统中，指令地址就是访问指令存储器InstMEM的**物理地址**。

因此add指令的数据通路中应该具备的部件有：指令存储器InstMEM、取指单元和完成虚实地址转换的内存管理单元MMU（可选）。它们构成了流水线CPU的前端。

##### 指令存储器InstMEM

本书将片上的1片SRAM芯片拆分成两个独立的SRAM模块，分别用于实现本书模型机的指令存储器InstMEM和数据存储器DataMEM。

指令保存在指令存储器InstMEM中，指令存储器InstMEM按**字节编址**且是只读存储器。指令存储器InstMEM的地址线InstMEM\_addr宽32位；输出数据InstMEM\_rdata宽32位；输出使能信号InstMEM\_en和写使能信号InstMEM\_wen各占宽1位。指令存储器InstMEM输出的32位数据就是指令码。本书模型机的存储都采用**小端模式**，所以指令存储器InstMEM输出的32位指令码与指令系统规范中定义的字节顺序是一致的，不需要做任何字节序列调整。

从图7-5所展示的指令存储器InstMEM的结构可知。虽然指令存储器InstMEM是只读存储器，但还是保留了写端口，写控制信号InstMEM\_wen（高电平有效）接地，即写操作始终无效。这样的设计是为了和之后的AXI总线协议设计保持一致性。

##### 取指单元

本实验将要实现的是一个64位的RISC-V流水线处理器， 图7-5给出了该流水线处理器中取指单元的结构及与指令存储器InstMEM的连接关系。取指单元位于CPU内部，而指令存储器InstMEM位于CPU外部。clock是系统时钟，每个时钟周期从上跳沿开始，延续至下一个上跳沿结束；reset是同步系统复位信号，高电平有效。当reset=1时，系统复位，此时PC被强制置为7FFCH。当reset=0时，系统处于工作状态。**译码级缓存DecodeStage**用于保存取指阶段的结果，在下一个clock提供给流水线的下一级进行译码。

程序计数器PC由一组64位的触发器组成。pc\_next由地址加法器对PC加4得到，pc\_next总是等于PC+4（此处4代表4个字节，即一条指令的宽度）。在clock上升沿时，用pc\_next更新PC的值。访问指令存储器InstMEM的地址由pc\_next提供，以便取出的指令和地址PC相对应。这是因为对指令存储器InstMEM的一次读数操作需要跨越两个时钟周期。连接地址信号的时候，将pc\_next的低32位接入指令存储器的地址端InstMEM\_addr。如果测试程序不需要4GB空间大小的话，那么也可以只将pc\_next的16位（对应64KB空间）或低8位（对应256B空间）接到指令存储器的地址端InstMEM\_addr，而InstMEM\_addr多余的高位地址线接地即可。



取指单元及指令存储器InstMEM的结构

图7-6展示了以PC+4为地址的指令存储器InstMEM的访问时序波形图。由图7-6可知，**指令存储器InstMEM的访问需要2个时钟周期，其时序**是：

**①第1个时钟周期：向指令存储器InstMEM发送读使能信号和读地址。**

因为指令存储器InstMEM是只读存储器，因此读使能信号InstMEM\_en=~reset，只要复位信号reset无效，在每个clock都可以从InstMEM中读取指令。访存地址由pc\_next（即PC+4）提供。由于第1个时钟周期尚未读到I1指令，因此此时的valid值应该为0，表示此时读到的指令无效。从图7-6看到，在读出I1指令的clock周期里（第2个时钟周期），PC正好是I1指令的地址8000H；在读出I2指令的clock周期里（第3个时钟周期），PC正好是I2指令的地址8004H…这样，在将PC和指令码一起保存到图7-5中译码级缓存DecodeStage里的时候，PC与该指令的地址对应。

**②第2个时钟周期：在clock的上升沿，指令存储器InstMEM输出指令码。**

在指令存储器InstMEM输出指令码的同一个时钟周期clock结束时的上升沿，此时valid的值为1指令有效，需要将PC和指令码以及valid同时写入译码级缓存DecodeStage。



取指令访问时序波形图

##### 虚实地址转换

①在支持虚拟存储器的系统中，需要实现RISC-V的S模式来支持虚拟存储器的机制。任何时候CPU上运行的程序中使用的地址都是**虚拟地址**，而真正访问内存读和写的时候，以及访问I/O设备的地址都是**物理地址**，因此需要使用内存管理单元MMU来对CPU给出的虚拟地址进行转换，使用转换得到的物理地址进行访存或访问I/O设备。

②在不支持虚拟存储器的系统中，CPU给出的就是物理地址，直接可用于访存或访问I/O设备。

在本书模型机实现RISC-V的S模式之前，是不支持虚拟存储器的，所以内存管理单元MMU使用直接映射的方式，即物理地址的值等于虚拟地址的值。因此在实现S模式之前的实验项目中，内存管理单元MMU的结构设计可以先忽略。

##### 译码级缓存

取指单元和译码单元这两级间的缓存称为**译码级缓存**。译码级缓存的触发器中存储的内容供应给下一级使用。譬如，译码单元中运行的指令和数据实际上是译码级缓存中存储的指令和数据。

本书对前后端的定义为：译码级缓存之前的流水线阶段称为**前端**，译码级缓存之后的流水线阶段称为**后端**。



译码级缓存结构

图7-7展示了译码级缓存的结构。译码级缓存内部有一个用于保存上一级传来的data数据包（即，图7-7中的data1）的寄存器，寄存器由触发器组成，在每个clock的上跳沿更新寄存器的内容。寄存器保存的内容直接传往下一级。

用Chisel对图7-7中data数据包的定义如下所示，其中XLEN的值为64，这样定义可以避免硬编码，使CPU更具有拓展性。

class IfIdData extends Bundle {

val valid   = Bool()

  val inst       = UInt(XLEN.W)

  val pc         = UInt(XLEN.W)

}

#### 运算类指令的数据通路的后端设计

前端部分已经成功取得指令，接下来需要通过译码识别出这条指令是add指令，并产生add指令执行所需的控制信号。

译码单元完成指令译码和准备源操作数这两个操作，指令译码由译码器完成；源操作数通过访问通用寄存器堆获得。

##### 译码单元——译码器

RISC-V有6种指令格式，如表1-4所示。译码器根据指令的操作码opcode字段识别出指令的格式后，再对其他字段译码。

本实验仅实现R型格式的运算指令。表7-1展示了RV64中所有R型运算指令， R型非字指令（助记符末位不带w）的操作码opcode都是**0110011**；R型字指令（助记符末位是w）的操作码opcode都是**0111011**。

分析出是R型字/非字指令后，通过func3字段区分各指令的运算类型，其中add和sub指令的func3一致、srl和sra指令的func3一致。最后再由func7字段的第6位（即I30）进行区分。

R型运算指令列表

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 25 | 24 | 20 | 19 | 15 | 14 | 12 | 11 | 7 | 6 | 0 |  |
| **funct7** | | **rs2** | | **rs1** | | **funct3** | | **rd** | | **opcode** | | **指令** |
| 0000000 | | rs2 | | rs1 | | 000 | | rd | | 0110011 | | add |
| 0100000 | | rs2 | | rs1 | | 000 | | rd | | 0110011 | | sub |
| 0000000 | | rs2 | | rs1 | | 001 | | rd | | 0110011 | | sll |
| 0000000 | | rs2 | | rs1 | | 010 | | rd | | 0110011 | | slt |
| 0000000 | | rs2 | | rs1 | | 011 | | rd | | 0110011 | | sltu |
| 0000000 | | rs2 | | rs1 | | 100 | | rd | | 0110011 | | xor |
| 0000000 | | rs2 | | rs1 | | 101 | | rd | | 0110011 | | srl |
| 0100000 | | rs2 | | rs1 | | 101 | | rd | | 0110011 | | sra |
| 0000000 | | rs2 | | rs1 | | 110 | | rd | | 0110011 | | or |
| 0000000 | | rs2 | | rs1 | | 111 | | rd | | 0110011 | | and |
| 0000000 | | rs2 | | rs1 | | 000 | | rd | | 0111011 | | addw |
| 0100000 | | rs2 | | rs1 | | 000 | | rd | | 0111011 | | subw |
| 0000000 | | rs2 | | rs1 | | 001 | | rd | | 0111011 | | sllw |
| 0000000 | | rs2 | | rs1 | | 101 | | rd | | 0111011 | | srlw |
| 0100000 | | rs2 | | rs1 | | 101 | | rd | | 0111011 | | sraw |

R型运算指令都是三地址指令，每条指令包含2个源操作数和1个目的操作数，所有操作数都采用寄存器寻址方式。记源操作数1为src1，源操作数2为src2。译码器应产生的控制信号如表7-2所示。

译码器需要产生的控制信号

|  |  |  |  |
| --- | --- | --- | --- |
| **信号名** | **含义** | **信号名** | **含义** |
| src1\_raddr | src1的通用寄存器堆读地址 | src2\_raddr | src2的通用寄存器堆读地址 |
| reg\_wen | 写回通用寄存器堆使能信号 | reg\_waddr | 通用寄存器堆的写地址 |
| op | 指令的操作类型 | valid | 指令是否有效 |

译码单元传递给运算单元的info数据包的结构如下：

object FuOpType {

  def apply() = UInt(5.W)

}

class Info extends Bundle {

  val valid   = Bool()

  val src1\_raddr = UInt(REG\_ADDR\_WID.W)

  val src2\_raddr = UInt(REG\_ADDR\_WID.W)

  val op         = FuOpType()

  val reg\_wen    = Bool()

  val reg\_waddr  = UInt(REG\_ADDR\_WID.W)

}

在这段chisel编写的info数据包结构代码中，REG\_ADDR\_WID的值为5，FuOpType是一个object类的apply方法，其值对应指令的操作类型op的宽度。

add指令格式和功能

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **位数**  **指令** | **31 25** | **24 20** | **19 15** | **14 12** | **11 7** | **6 0** | **R型格式算术运算指令的功能** |
| **funct7** | **rs2** | **rs1** | **funct3** | **rd** | **opcode** |
| add rd,rs1,rs2 | 0000000 | rs2 | rs1 | 000 | rd | 0110011 | 加：rs1+rs2→rd |

表7-3给出了add指令的定义和功能。add指令的的源操作数都来自通用寄存器堆。寄存器堆采用2个读端口和1个写端口的结构。两个读端口的地址信号src1\_raddr对应指令码I[19:15]，src2\_raddr对应指令码I[24:20]。add指令需要将运算结果写回通用寄存器堆，因此写使能信号reg\_wen=1，reg\_waddr对应指令码I[11:7]。因为算术逻辑运算都是由执行单元的多功能算逻运算器ALU完成，因此只需要将指令操作类型op设置正确就能完成对指令的区分。以下介绍两种设置op的方法：

**①第一种方法：**

将所有指令从0开始按顺序编号，如add为1、sub为2、sll为3……

**②第二种方法：**

R型非字指令（助记符不带W）和字指令（助记符带W）,它们的操作码opcode在I[3]这一位不同。此外func3和func7的第6位有区别。在R型指令格式识别基础上，将这几个不同位进行拼接后二次译码识别，便可区分出不同的R型运算指令，如表7-4所示。譬如，设计op={I3|I30|I[14:12]}，那么对于R型加法指令有add而言，译码器对I[6:0]译码，发现是R型指令，则将{I3|I30|I[14:12]}赋值给op，编码00000对应add指令。再如，对addw指令，译码器对I[6:0]译码，发现是R型指令，则将{I3|I30|I[14:12]}赋值给op，编码10000对应addw指令。

第二种op设置方法下的R型指令译码

|  |  |  |
| --- | --- | --- |
| **指令** | **译码opcode=I[6:0]** | **赋值 op={I3|I30|I[14:12]}** |
| add | 0110011 | 00000 |
| sub | 0110011 | 01000 |
| sll | 0110011 | 00001 |
| slt | 0110011 | 00010 |
| sltu | 0110011 | 00011 |
| xor | 0110011 | 00100 |
| srl | 0110011 | 00101 |
| sra | 0110011 | 01101 |
| or | 0110011 | 00110 |
| and | 0110011 | 00111 |
| addw | 0111011 | 10000 |
| subw | 0111011 | 11000 |
| sllw | 0111011 | 10001 |
| srlw | 0111011 | 10101 |
| sraw | 0111011 | 11101 |

不同的op设计对应译码器的FU（FunctionUnit，功能部件）内部的解码逻辑也不一样，优秀的op编码设计方案有助于提升硬件性能。

在Chisel中新建一个object类来存放ALU中对指令操作类型op的定义，并且可以定义相应的函数方便后续使用：

object ALUOpType {

  def add  = "b00000".U

  def sraw = "b11101".U

  def isWordOp(func: UInt) = func(4)

}

valid表示当前指令是否有效，其值来自于译码级缓存DecodeStage的valid信号。

##### 译码单元——通用寄存器堆

完成了控制信号的生成，接下来需要准备源操作数，也就是访问通用寄存器堆。如前述，通用寄存器堆采用双读端口和单写端口的三端口结构，相比指令存储器的存储类型，通用寄存器堆的访问都是在当前时钟周期内完成。通用寄存器堆可以实现在译码单元内部，也可以直接实现在CPU内部作为一个独立模块，两者没有什么太大的区别。

图7-8展示了译码单元的结构，译码器将从译码级缓存获得的指令进行译码，产生了相关的控制信号，将所有的控制信号打包成info数据包与寄存器堆读回的源操作数组成的src\_info数据包以及译码级缓存获得的PC一起打包成一个data数据包发送至下一级流水线缓存。



译码单元结构

图中src\_info由src1\_rdata和 src2\_rdata组合而成，其定义如下：

class SrcInfo extends Bundle {

  val src1\_data = UInt(XLEN.W)

  val src2\_data = UInt(XLEN.W)

}

##### 执行级缓存和执行单元

译码单元和执行单元这两级间的缓存称为**执行级缓存**，其结构与图7-7所示的译码级缓存结构类似，也是由触发器构成的寄存器。

执行单元中使用的data数据包（即图7-9中的data1）的内容有：当前指令地址PC值、当前指令译码信号产生的信号info、从通用寄存器堆读出的两个源操作数src\_info。用Chisel编写本级data数据包定义如下：

class IdExeData extends Bundle {

  val pc       = UInt(XLEN.W)

  val info     = new Info()

  val src\_info = new SrcInfo()

}

指令在执行单元中完成运算，并获得计算结果。



执行单元结构

图7-9展示了执行单元的结构，R型运算指令只需要使用多功能算术逻辑运算器ALU这一个部件。将执行级缓存传来的data数据包中的info数据包和src\_info数据包发送给ALU。ALU根据info数据包内指令操作op编码进行相应的计算，源操作数从src\_info数据包而来。ALU将运算结果reg\_wdata打包到rd\_info数据包中，和PC值以及info数据包内的valid、reg\_wen、reg\_waddr信号一块打包成新的data数据包发送至访存级缓存。

图7-9中rd\_info数据包的定义如下：

class RdInfo extends Bundle {

  val wdata = UInt(XLEN.W)

}

##### 访存级缓存和访存单元

执行单元和访存单元这两级间的缓存称为**访存级缓存**，其结构与执行级缓存结构一样。

图7-10中访存级缓存的data1数据包定义如下：

class ExeMemData extends Bundle {

  val pc       = UInt(XLEN.W)

  val info     = new Info()

  val rd\_info  = new RdInfo()

}

可见执行级缓存输出的src\_info数据包没有继续往访存级缓存传递，在访存级缓存中增加了rd\_info的数据包。

图7-10展示了访存单元的结构。R型运算指令不需要访问内存，因此add指令在该流水级什么也不做，只需要将上一级缓存内的data数据包传到下一级缓存中即可，即在图7-10中，data1数据包与data2数据包完全相同。将数据存储器的读使能信号DataMEM\_en和写使能信号DataMEM\_wen都置为无效（即，清零），无需理会数据存储器的地址信号DataMEM\_addr、写入数据DataMEM\_wdata以及读出数据DataMEM\_rdata这三种信号。



访存单元结构

##### 写回级缓存和写回单元

访存单元和写回单元这两级间的缓存称为**写回级缓存**，写回级缓存的结构与访存级缓存结构一样，data数据包的内容也与访存级缓存一样。

R型运算指令需要写回通用寄存器堆，因此add指令需要在写回级访问通用寄存器堆。图7-11展示了写回单元的结构。由图可知，将写回级缓存输出的data数据包解包得到valid、reg\_wen、reg\_waddr和reg\_wdata信号。由于指令可能无效，因此还需对写使能进行处理，最终的写使能等于valid和reg\_wen的逻辑与。将最终的写使能、写地址和写数据信号发送至通用寄存器堆，同时这些信号还需要与PC一起发往CPU的外部作为debug调试信号。



写回单元

### 实验要求

1. 基于本实验提供的五级流水线编程框架开展编程设计，使得设计的MyCPU 能够完成表7-1中的所有指令的功能。
2. 通过本实验提供的所有仿真验证测试用例。
3. 通过本实验提供的所有板级验证测试用例。
4. 线上提交实验作品，获得系统评分。
5. 撰写实验报告。报告中应包含以下内容和至少3道本实验思考题。
6. 选择表7-1中的一条指令（非add），按照你自己的理解，逐步介绍其数据通路设计的思路以及实现过程。
7. 尝试自己绘制一幅MyCPU内部数据通路图，后续实验将在此基础上修改，使得该MyCPU能够执行更多格式的指令。

### 实验步骤

1. 阅读本书第2章，掌握Chisel基本语法。
2. 按照第3章的方法，安装并熟悉实验开发环境。
3. 套用所给的流水线CPU模板，编写代码，在MyCPU中实现表7-1中的所有指令功能。
4. 阅读第4章，了解差分测试调试方法，并用差分调试方法开展调试。
5. 在Vivado平台上生成本实验对应的.bit文件，并开展远程板级调试，远程板级调试网址是：fpga.hdu.edu.cn。
6. 撰写实验报告。

### 思考与探索

1. RISC-V 指令集是定长指令集吗？
2. RV64 和 RV32的 R 型运算指令是否有区别？
3. 设计比较指令slt 和 sltu 的目的是什么？
4. sll、srl 和 sra 这三条指令在 rs2[63:6]不全为 0 的时候，指令的执行结果是什么？
5. RISC-V 的运算指令有进行运算结果的溢出判断吗？为什么？
6. 为什么并不是所有的R型计算指令都有对应的字指令(助记符带W的指令)？
7. 请问差分测试框架只用图7-11中的4个debug信号够吗？假如有的指令不将结果写回通用寄存器，这时框架该如何发现问题？
8. 当前处理器采用的是哈佛结构还是冯诺依曼结构？
9. 谈谈你在实验中碰到了哪些问题？又是如何解决的？