# 实验九 实现R型运算类指令的理想流水线设计实验

## 实验目的

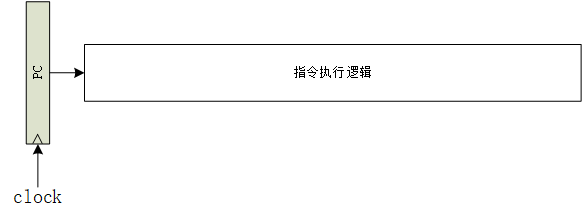
1. 掌握R型运算类指令的数据通路。
2. 掌握经典单发射五级流水线的设计方法。
3. 掌握流水线CPU设计的编程基本框架。

## 实验原理与实验内容

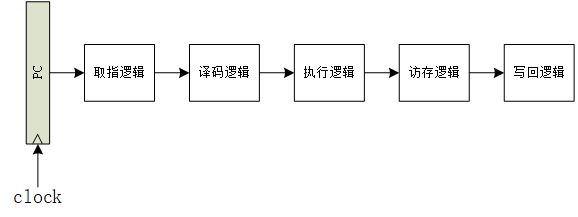
### 单周期与流水线

#### 结构差异

RISC-V单周期CPU设计实现简单，控制器部分是纯组合逻辑电路，但该CPU所有指令执行时间均是一个相同的周期，即以速度最慢的指令作为设计其时钟周期的依据。如图1-1所示，单周期CPU的时钟频率取决于数据通路中的关键路径（最长路径），所以单周期CPU效率较低，性能不佳，现代处理器中已不再采用单周期方式，取而代之的是多周期设计方式。而多周期CPU设计中流水线CPU设计是目前的主流技术。



* + - 1. 单周期CPU



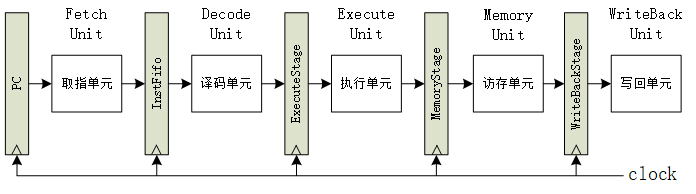
* + - 1. 单周期CPU逻辑划分

将电路流水线化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。从电路设计最终的实现形式来看，是将一段组合逻辑按照功能划分为若干阶段，在各功能阶段的组合逻辑之间插入时序器件（通常是触发器），前一阶段的组合逻辑输出接入时序器件的输入，后一阶段的组合逻辑输入来自这些时序器件的输出。

而将电路流水化最困难的地方是决定将单周期CPU中的组合逻辑划分为多少个阶段以及各个阶段包含哪些功能。这个设计决策需要结合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完成。对于教学而言，这部分设计涉及的内容过多、过细、过深，因此本实验将直接采用经典的单发射五级流水线划分。所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. 取指阶段的主要功能是将指令取回。
2. 译码阶段的主要功能是解析指令生成控制信号并读取通用寄存器堆生成源操作数。
3. 执行阶段的主要功能是对源操作数进行算术逻辑类指令的运算或者访存指令的地址计算。
4. 访存阶段的主要功能是取回访存的结果。
5. 写回阶段的主要功能是将结果写入通用寄存器堆。

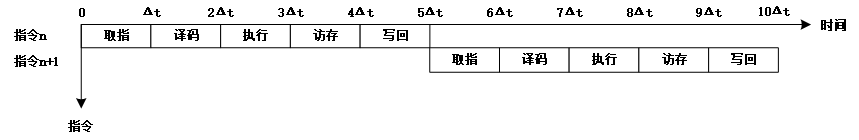
结合以上流水线阶段的划分方案，将单周期CPU的数据通路拆分为五段（如图1-2所示），并在各段之间加入触发器作为流水线缓存，图1-3展示了RISC-V流水线的逻辑结构。



* + - 1. 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，每到来一个时钟clock，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



* + - 1. 单周期CPU时空图



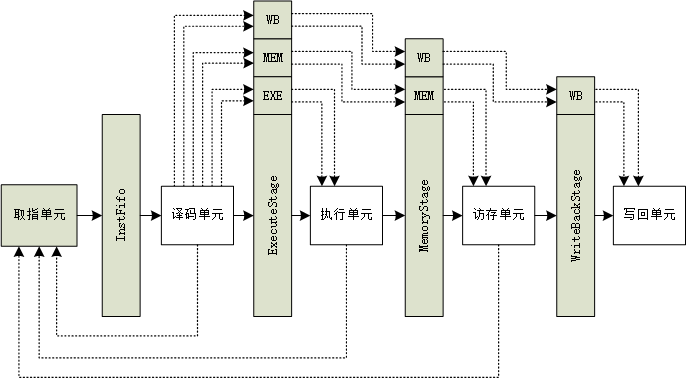
* + - 1. 五级流水线CPU时空图

图1-4给出了RISC-V单周期CPU的时空图，可以看到，每条指令执行需要5个时钟周期，即。1个时钟周期是1个，也就是每5个可以提交1条指令，单周期CPU的IPC是0.2。算出运行n条指令花费的总时间为。图1-5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，每个时钟周期流水线可以提交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。

### 数据通路设计

#### 设计的基本方法

在计算机结构设计实验中，需要设计实现的CPU也是一个数字逻辑电路，其设计应该遵循数字逻辑电路设计的一般性方法。CPU不但要完成运算，也要维持自身的状态，所以CPU一定是既有组合逻辑电路又有时序逻辑电路的。CPU输入的、运算的、存储的、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（Datapath）。因此，要设计CPU这个数字逻辑电路，首要的工作就是设计数据通路。同时，因为数据通路中会有多路选择器、时序逻辑器件，所以还要有相应的控制信号，产生这些控制信号的逻辑被称为控制逻辑。所以，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”。



* + - 1. 理想的五级流水线CPU数据与控制信号传递图

根据指令系统规范中的定义设计出“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的对象。显然，这些操作和操作的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，再用多路选择器从中选择出所需的结果。遵循这个一般性方法，下面具体介绍如何分析指令的功能以及如何设计出数据通路。图1-6展示了RISC-V理想的五级流水线CPU数据与控制信号传递图。

#### 以ADD指令为例

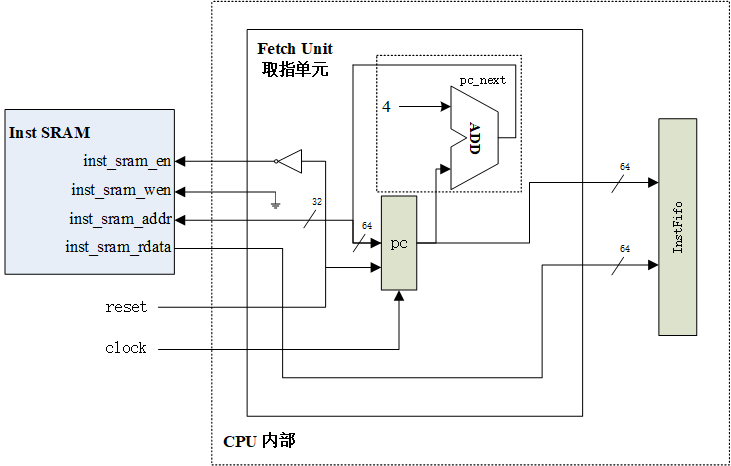
以ADD指令为例，分析一下该指令需要哪些数据通路部件。

首先，执行指令的前提是需要得到ADD这条指令，需要使用这条指令对应的PC作为虚拟地址进行虚实地址转换，得到访问指令SRAM的物理地址。这意味着需要的数据通路部件有：取指单元、虚实地址转换部件和指令SRAM。下面先对这部分的数据通路进行实现。

前端

##### 取指单元

因为实现的是一个64位的处理器，所以PC的指令宽度是64比特。用一组64位的触发器来存放PC。（后面为了行文简洁，在不会导致混淆的情况下，用pc代表这组用于存放PC的64位触发器。）由于处理器使用到了SRAM进行数据的存取，而SRAM的特性是一次读数操作需要跨越两个时钟周期，第一个时钟周期向RAM发出读使能和读地址，第二个时钟周期RAM才能返回读结果。因此发送给指令SRAM的地址应该是下一条指令的PC，也就是pc\_next，而目前的实验设计pc\_next的大小将一直等于pc+4（这里的4代表寻址4个字节，即一条指令的宽度），因此pc\_next和pc之间只是组合逻辑的关系。



* + - 1. 取指单元及指令RAM

pc的输出将送到指令SRAM中用于获取指令，由于指令SRAM的地址宽度只有32位，因此只有pc的低32会被使用。目前来看，PC的输入有两个，一个是复位值0x80000000（由于发送给指令SRAM的是pc\_next，所以pc的真正复位值其实是0x80000000-0x4），一个是复位撤销之后pc\_next的值。

因为取指单元只会对内存进行读操作，因此inst\_sram\_en只要在reset无效时使能即可，而inst\_sram\_wen应该恒为低电平。图1-7展示了取指单元的结构。

##### 虚实地址转换

任何时候CPU上运行的程序中出现的地址都是虚地址，而CPU本身访问内存、I/O所用的地址都是物理地址，因此需要对CPU发出的虚拟地址进行转换，使用物理地址进行访存。在实现RISC-V的S模式之前，目前实现的CPU的虚拟地址与物理地址之间使用直接映射的方式，即物理地址的值等于虚拟地址的值。因此虚实地址转换部件目前可以先省略。

##### 指令RAM

得到取指所需的物理地址后，接下来需要将该地址送往内存。实验采用片上的RAM作为内存，并且将RAM进一步分拆为指令RAM和数据RAM两块物理上独立的RAM以简化设计。

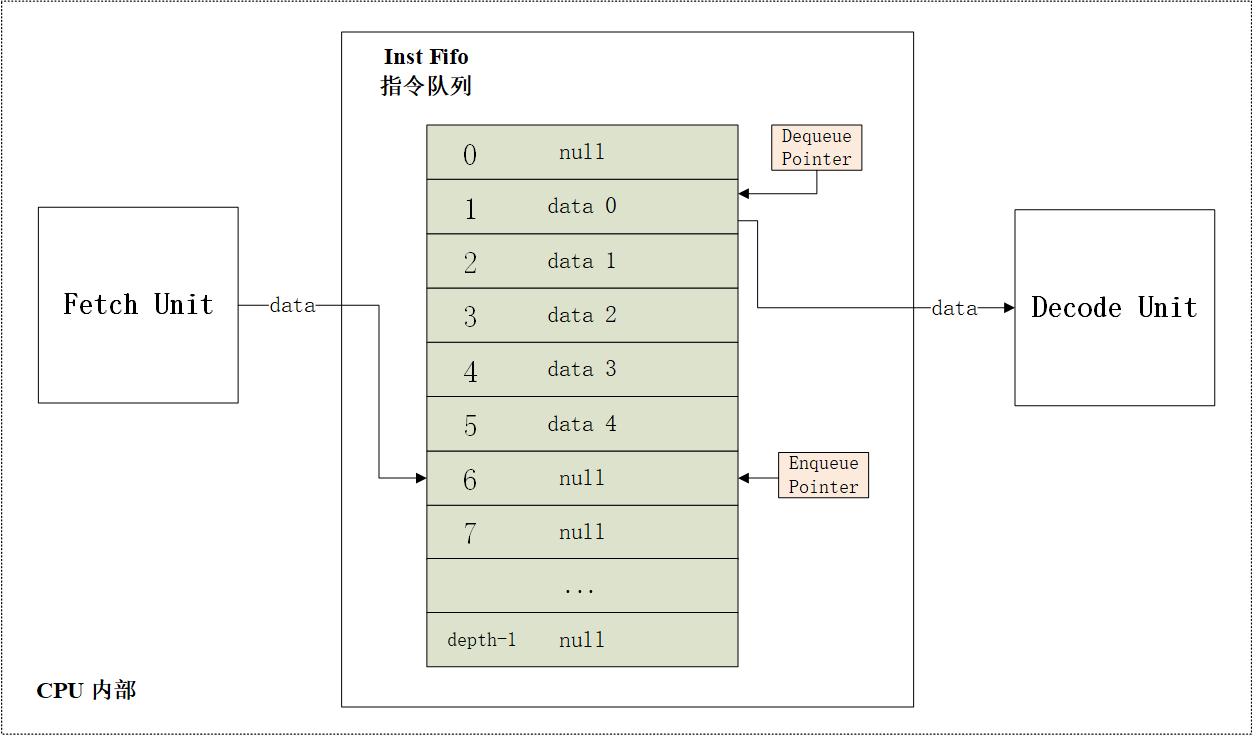
指令RAM输出的32位数据就是指令码。实验实现的CPU采用小尾端的寻址，所以指令RAM输出的32位数据与指令系统规范中的定义的字节顺序是一致的，不需要做任何字节序调整。

图1-7展示了指令RAM的结构。指令RAM保留了写接口，这样的接口设计，是为了和之后的AXI的设计保持一致性。

##### 指令队列

取指单元/译码单元之间的流水线缓存称为指令队列。指令队列之前的阶段称为前端，指令队列之后的阶段称为后端。

当取指单元一次取指的数量大于译码单元可以解码的数量时，又或是后端流水线发生暂停时，取指单元可以继续取指，多余的指令可以在指令队列中排队等待，而不用暂停取指。因此指令队列部件的实现可以解耦前后端。



* + - 1. 指令队列

如图1-8所示，指令队列的实现是一个深度为depth的寄存器组，每个寄存器中保存一个叫做data的数据包[[1]](#footnote-1)（目前需要保存指令的内容以及指令的PC这两个数据），宽度应该和data的宽度一致。出队指针和入队指针都是一个宽度为的寄存器。使用出队指针指示队列的头部，入队指针指示队列的尾部。由取指单元发送的数据存入入队指针指示的寄存器；出队指针指示的寄存器保存的数据发送到译码单元中。目前实现的是理想流水线，因此每一个clock的上跳沿来临时入队指针和出队指针都应该加1，发生reset时，两个指针都应该置为0。

这里的data数据包的定义如下，其中XLEN的值为64，这样子定义是为了避免硬编码，使CPU更具有拓展性。

class IfIdData extends Bundle {

  val inst         = UInt(XLEN.W)

  val pc           = UInt(XLEN.W)

}

后端

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为ADD指令，并产生相应的控制信号。

##### 译码单元

译码单元要完成指令译码和源操作数的准备这两个操作，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。

（a）译码器

首先需要明白译码器是如何解码不同指令的。RISC-V有6种指令格式，如表1-1所示。译码器根据指令的opcode段识别出指令的格式，再进行下一步的译码。

* + - * 1. RISC-V指令格式

| 31 | 25 | 24 | 20 | 19 | 15 | 14 | 12 | 11 | 7 | 6 | 0 | Type |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| funct7 | | rs2 | | rs1 | | funct3 | | rd | | opcode | | R |
| imm[11:0] | | | | rs1 | | funct3 | | rd | | opcode | | I |
| imm[11:5] | | rs2 | | rs1 | | funct3 | | imm[4:0] | | opcode | | S |
| imm[12|10:5] | | rs2 | | rs1 | | funct3 | | imm[4:1|11] | | opcode | | B |
| imm[31:12] | | | | | | | | rd | | opcode | | U |
| imm[20|10:1|11|19:12] | | | | | | | | rd | | opcode | | J |

本实验只需要实现R型的运算指令。表1-2展示了RV64中所有R型运算指令。先分析非字指令，不难发现，R型运算指令的opcode是0110011，再通过func3区别各指令的运算类型，其中ADD和SUB、SRL和SRA的func3一致，再由func7的第6位进行区分。而字指令的分析也和非字指令的分析一致。

* + - * 1. R型运算指令

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 25 | 24 | 20 | 19 | 15 | 14 | 12 | 11 | 7 | 6 | 0 |  |
| 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，每条指令都有两个源操作数以及一个目的操作数。记源操作数1为src1，源操作数2为src2。因此译码器中要产生的控制信号如表1-3所示。

* + - * 1. 译码器需要产生的控制信号

|  |  |
| --- | --- |
| 信号名 | 含义 |
| src1\_ren  src1\_raddr  src2\_ren  src2\_raddr  op  reg\_wen  reg\_waddr | src1是否需要读通用寄存器堆  src1的通用寄存器堆读地址  src2是否需要读通用寄存器堆  src2的通用寄存器堆读地址  指令的操作类型  是否需要写回通用寄存器堆  通用寄存器堆的写地址 |

因此可以得到info数据包的结构如下，其中REG\_ADDR\_WID的值为5，FuOpType是一个object类的apply方法，其值是op的宽度，关于op的定义将在下文介绍。

object FuOpType {

  def apply() = UInt(5.W)

}

class Info extends Bundle {

  val src1\_ren   = Bool()

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

  val src2\_ren   = Bool()

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

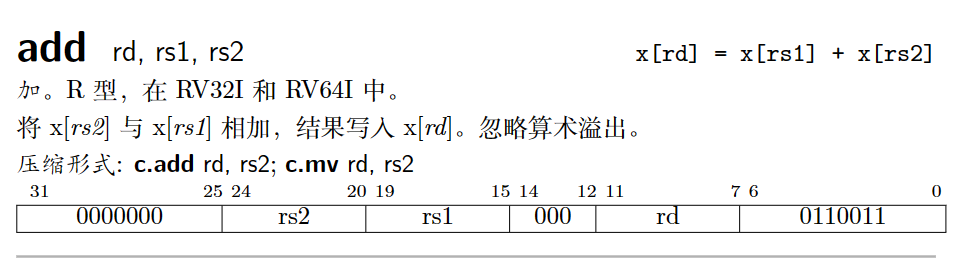
  val op         = FuOpType()

  val reg\_wen    = Bool()

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

}

接下来阅读手册，根据手册对指令功能的定义对各控制信号进行赋值。图1-9展示了ADD指令的定义。



* + - 1. ADD指令定义

ADD指令的源操作数都来自通用寄存器堆，因此src1\_ren和src2\_ren都为1，src1\_raddr对应指令的19至15位，src2\_raddr对应指令的24至20位。ADD指令需要写回通用寄存器堆，因此reg\_wen为1，reg\_waddr对应指令的11至7位。因为目前要实现的指令都在执行单元的ALU中进行运算，因此只需要将op设置正确就能完成指令的区分。op的设置有多种方法，下面介绍两种：

1. 简单的方法：

直接将指令从0开始按顺序编号，如ADD为1、SUB为2、SLL为3……

1. 稍微复杂一些的方法：

观察指令格式进行op的设计，比如非字指令的opcode一致，仅func3以及func7的第6位有区别，那么可以将这几位进行拼接，由于ADD又有ADD和ADDW的区别，因此可以再加一位进行字指令的区分，因此ADD的op可以设计为00000。按这种思路，SRAW的op就是11101。

不同的op设计对于FU（FunctionUnit，功能部件）内部的解码会有一定的影响，好的编码设计可以大大提升硬件性能。

在chisel中可以新建一个object类专门存放MDU中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

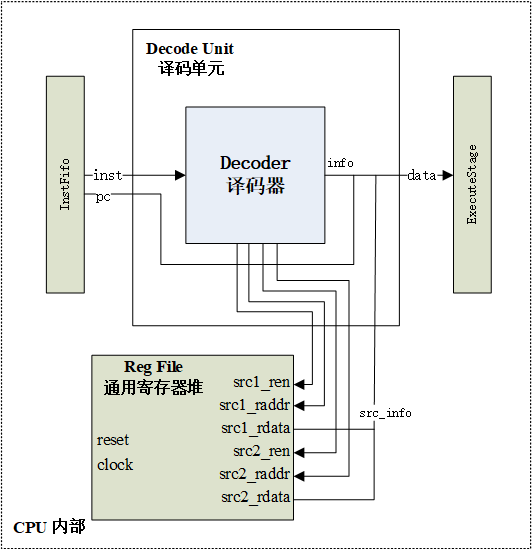
  def isWordOp(func: UInt) = func(4)

}

（b）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，相比SRAM这种存储类型，通用寄存器堆的访问都是当拍完成。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1-10展示了译码单元的结构，译码器将从指令队列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数据包以及指令队列获得的pc一起打包成一个data数据包发送至下一级流水线缓存。



* + - 1. 译码单元

其中src\_info的定义如下：

class SrcInfo extends Bundle {

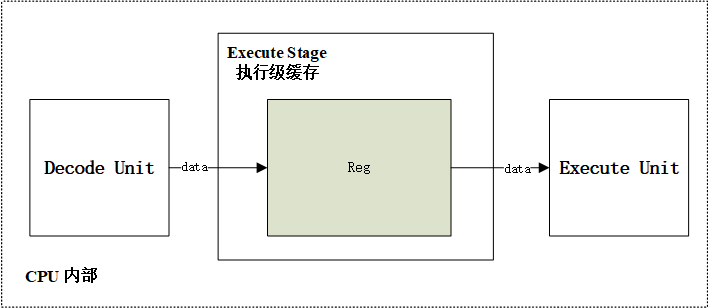
  val src1\_data = UInt(XLEN.W)

  val src2\_data = UInt(XLEN.W)

}

##### 执行级缓存

译码单元/执行单元这两级间的缓存称为执行级缓存，将缓存的名称归到它的输出对应的那一个流水阶段。这样子命名可以与仿真波形调试时的观察习惯保持一致，即除了触发器时钟采样边沿附近那段建立保持时间（当采用零延迟仿真时，这段时间可以看作瞬时、无穷小）外，触发器中存储的内容都是与它输出那一级流水的组合逻辑相关联的，如执行单元中运行的指令实际上是译码单元/执行单元这两级缓存中存储的数据。后面流水线级间缓存也将统一采用这种标识的方式。



* + - 1. 执行级缓存

图1-11展示了执行级缓存的结构。执行级缓存内部有一个用于保存上一级传来的data数据包的寄存器，在每个clock的上跳沿更新寄存器内容。寄存器保存的内容直接传往下一级。

在该级中对于data数据包的定义如下：

class IdExeData extends Bundle {

  val pc       = UInt(XLEN.W)

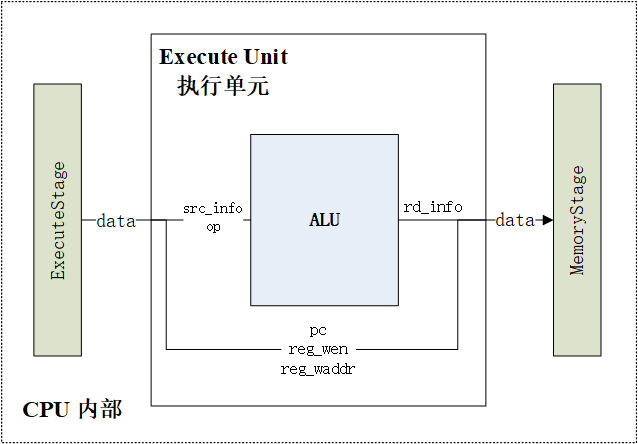
  val info     = new Info()

  val src\_info = new SrcInfo()

}

##### 执行单元

在执行单元中，指令需要在这里完成运算指令结果的计算。



* + - 1. 执行单元

图1-12展示了执行单元的结构，R型运算指令只需要使用ALU这一个部件即可（ALU部件已经在数字电路实验中学习过）。将执行级缓存传来的data数据包（包内包括info数据包和src\_info数据包）发送至ALU中。ALU可以通过info数据包内op的区别进行不同的计算操作，而源操作数在src\_info数据包中。ALU将运算结果reg\_wdata打包到rd\_info数据包中和pc以及info数据包内的reg\_wen、reg\_waddr一块打包成新的data数据包发送至访存级缓存。

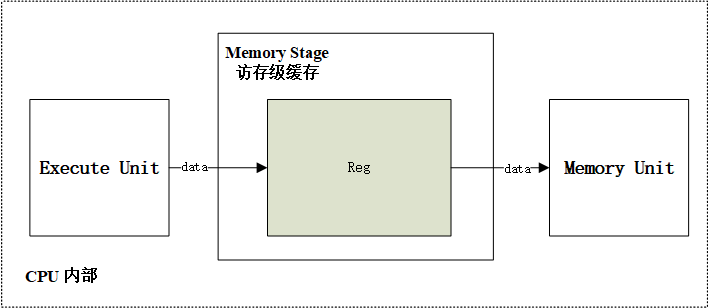
其中rd\_info的定义如下：

class RdInfo extends Bundle {

  val wdata = UInt(XLEN.W)

}

##### 访存级缓存



* + - 1. 访存级缓存

图1-13展示了访存级缓存的结构。访存级缓存的结构与执行级缓存结构一致。

data数据包的定义如下：

class ExeMemData extends Bundle {

  val pc       = UInt(XLEN.W)

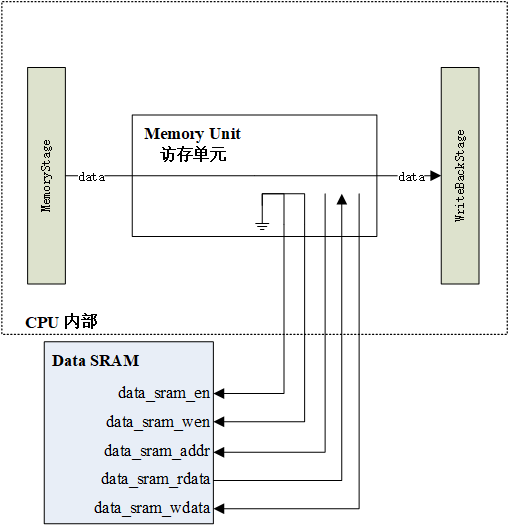
  val info     = new Info()

  val rd\_info  = new RdInfo()

}

可以发现src\_info数据包已经没有继续往下传递了，此处增加了rd\_info的数据包。

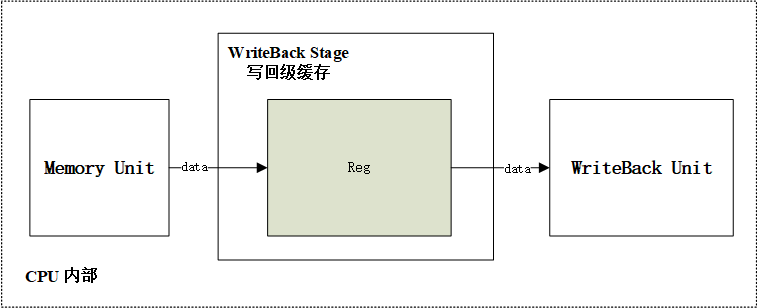
##### 访存单元



* + - 1. 访存单元

ADD指令并不需要访问内存，因此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。图1-14展示了访存单元的结构。将data\_sram\_en和data\_sram\_wen置为0，对于data\_sram\_addr、data\_sram\_wdata以及data\_sram\_rdata这三个信号就不需要理会了（在chisel语言中可以使用DontCare对data\_sram\_addr和data\_sram\_wdata进行赋值）。

##### 写回级缓存



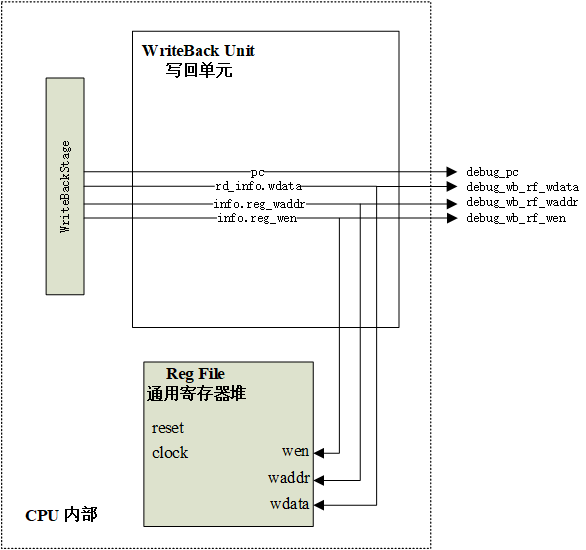
* + - 1. 写回级缓存

图1-15展示了写回级缓存的结构。写回级缓存的结构与访存级缓存结构一致。

data数据包的内容与访存级缓存一致。

##### 写回单元

ADD指令需要写回通用寄存器堆，因此需要在写回级访问通用寄存器堆。将data数据包解包，将info数据包内的reg\_wen、reg\_waddr和reg\_wdata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信号。图1-16展示了写回单元的结构。



* + - 1. 写回单元

### 开发环境的组织与结构

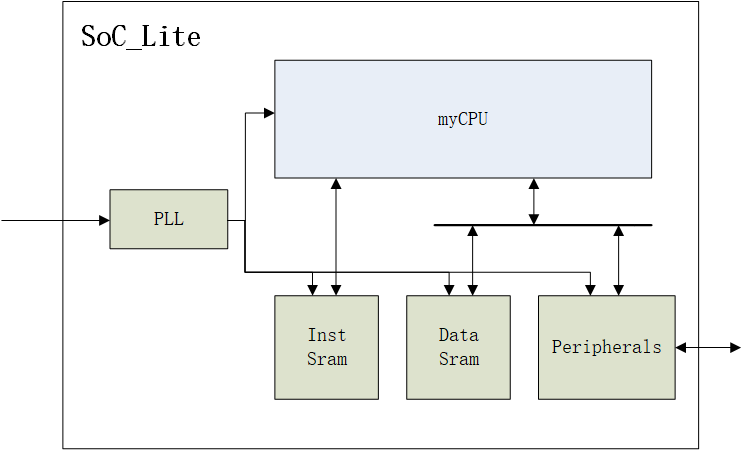
整个CPU设计开发环境（TODO：目录名称）的目录结构及各主要部分的功能如（TODO）所示。其中只有标黑色的部分是需要自行开发的，其余部分均为开发环境框架代码已经设计完毕。

TODO：图片

#### 验证所用的计算机硬件系统

单纯实现一个CPU没有什么实用价值，通常需要基于CPU搭建一个计算机硬件系统。本实验也是基于CPU搭建一个计算机硬件系统，然后通过在这个计算机硬件系统上运行测试程序来完成CPU的功能验证。

在引入AXI总线接口设计之前，实验将采用一个简单的计算机硬件系统。这个硬件系统将通过FPGA开发板实现。其核心是在FPGA芯片上实现的一个片上系统（System On Chip，SoC）。这个SoC芯片通过引脚连接电路板上的时钟晶振、复位电路，以及LED灯、数码管、按键等外设接口设备。SoC芯片内部也是一个小系统，其顶层为SoC\_Lite，内部结构如图1-17所示，对应的RTL代码均位于（TODO）目录下。只需重点关注SoC\_Lite这个小系统。可以看到，SoC\_Lite的核心是实验将要实现的CPU——myCPU。这个CPU与指令RAM（Inst Sram）和数据RAM（Data Sram）进行交互，完成取指和访存的功能。除此之外，这个小系统中还包含PLL、Peripherals等模块。



* + - 1. 基于myCPU的简单Soc结构

在数据通路分析中已经阐述了指令RAM和数据RAM与CPU之间的关系。这里简单解释一下PLL、Peripherals以及myCPU与Data Sram、Peripherals之间的二选一功能。

开发板上给FPGA芯片提供的时钟（来自时钟晶振）主频是100MHz。如果直接使用这个时钟作为SoC\_Lite中各个模块的时钟，则意味着myCPU的主频至少要能达到100MHz。对于初学者来说，这可能是个比较严格的要求，因此实验添加了一个PLL IP，将其输出时钟作为myCPU的时钟输入。这个PLL以100MHz 输入时钟作为参考时钟，输出时钟频率可以配置为低于100MHz。

myCPU通过访问Peripherals部件来驱动板上的LED灯、数码管，接收外部按键的输入。其操控的原理如下：外部的LED灯、数码管以及按键都是通过导线直接连接到FPGA的引脚上的，通过控制FPGA输出引脚上的电平的高、低就可以控制LED灯和数码管。同样，也可以通过观察FPGA输入引脚上电平的变化来判断一个按键是否按下。这些FPGA引脚又进一步连接到Peripherals部件中某些寄存器的某些位上，所以myCPU可以通过写Peripherals部件寄存器控制输出引脚的电平来控制LED灯和数码管，也可以通过读Peripherals部件寄存器来知晓连接到按键的引脚是高电平还是低电平。

myCPU和dram、Peripherals之间有一个“一分二”部件。这是因为在RISC-V指令系统架构下，所有I/O设备的寄存器都是采用Memory Mapped方式访问的，这里实现的Peripherals也不例外。MemoryMapped访问方式意味I/O设备中的寄存器各自有一个唯一内存编址，所以CPU可以通过load、store 指令对其进行访问。不过，dram作为内存也是通过load、store指令进行访问的。对于一条load或store指令来说，如何知晓它访问的是Peripherals还是dram？在设计SoC的时候可以用地址对其进行区分（相当于存在一个数据选择器，其根据地址选择数据）。因此在设计SoC的数据通路时就需要在这里引入一个“一分二”部件，它的选择控制信号是通过对访存的地址范围进行判断而得到的。

这里要提醒是，因为整个SoC\_Lite的设计都要实现到FPGA芯片中，所以在进行综合实现的时候，所需选择的顶层应该是SoC\_Lite，而不是设计实现的myCPU。

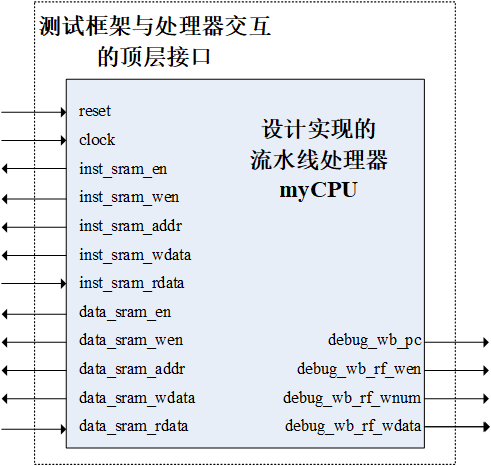
#### 验证所用的计算机仿真系统

由于上板的限制条件很多，这里再介绍软件的仿真方法，其运行效果与上板仿真几乎无异，但效率更高且更加方便。

TODO：仿真结构

#### myCPU的顶层接口

为了使设计的CPU能够直接集成到本平台所提供的CPU实验环境中，这里要对CPU的顶层接口做出明确的规定。myCPU顶层接口信号的详细定义如图1-18以及表1-4所示。只要设计的CPU符合这样的顶层接口就可以接入的差分测试框架中使用。

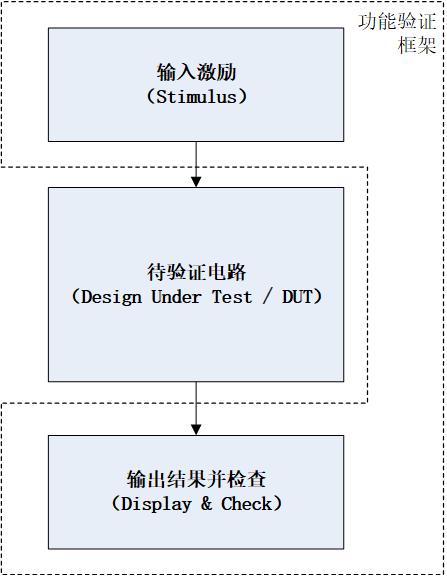


* + - 1. 差分测试框架接口
         1. myCPU顶层接口信号的描述

| 名称 | 宽度 | 方向 | 含义 |
| --- | --- | --- | --- |
|  |  |  | 时钟与复位 |
| clock | 1 | input | 时钟信号，来自PLL部件的输出 |
| reset | 1 | input | 复位信号，高电平同步复位 |
|  |  |  | 指令端访存接口 |
| inst\_sram\_en | 1 | output | RAM使能信号，高电平有效 |
| inst\_sram\_wen | 4 | output | RAM字节写使能信号，高电平有效 |
| inst\_sram\_addr | 32 | output | RAM读写地址，字节寻址 |
| inst\_sram\_wdata | 32 | output | RAM写数据 |
| inst\_sram\_rdata | 32 | input | 读数据 |
|  |  |  | 数据端访存接口 |
| data\_sram\_en | 1 | output | RAM使能信号，高电平有效 |
| data\_sram\_wen | 8 | output | RAM字节写使能信号，高电平有效 |
| data\_sram\_addr | 32 | output | RAM读写地址，字节寻址 |
| data\_sram\_wdata | 64 | output | RAM写数据 |
| data\_sram\_rdata | 64 | input | 读数据 |
|  |  |  | debug信号，供验证平台使用 |
| debug\_wb\_pc | 64 | output | 写回级PC，需要将PC从取指级逐级传至写回级 |
| debug\_wb\_rf\_wen | 1 | output | 写回级写通用寄存器堆的写使能 |
| debug\_wb\_rf\_wnum | 5 | output | 写回级写通用寄存器堆的寄存器号 |
| debug\_wb\_rf\_wdata | 64 | output | 写回级写通用寄存器堆的写数据 |

### 差分测试

在介绍如何对CPU进行差分测试前，先了解下如何对数字逻辑电路的进行功能验证。



* + - 1. 功能验证框架

数字逻辑电路的功能验证的作用是检查所设计的数字逻辑电路在功能上是否符合设计目标。简单来说，就是检查设计的电路功能是否正确。这里所说的功能验证与软件开发里的功能测试的意图是一样的。但是，要注意，这里使用了“验证”（Verification）这个词，这是为了避免和本领域另一个概念“测试”（Test）相混淆。在集成电路设计领域，测试通常指检查生产出的电路没有物理上的缺陷和偏差，能够正常体现设计所期望的电路行为和电气特性。

所谓数字电路的功能仿真验证，就是用（软件模拟）仿真的方式而非电路实测的方式进行电路的功能验证。图1-19给出了数字电路功能仿真验证的基本框架。

在这个基本框架中，给待验证电路（DUT）一些特定的输入激励，然后观察DUT的输出结果是否和预期一致。这个过程很类似与程序编程里的OJ（Online Judge）测试，通过输入不同的测试数据得到不同的运行结果，比对运行结果进行判断程序的正确性。

对CPU设计进行功能仿真验证时，沿用的依然是上面的思路，但是在输入激励和输出结果检查方面的具体处理方式与简单的数字逻辑电路设计有区别。对简单数字逻辑电路进行功能仿真验证时，通常是产生一系列变化的激励信号，输入到被验证电路的输入端口上，然后观察电路输出端口的信号，以判断结果是否符合预期。对于CPU来说，其输入/输出端口只有时钟、复位和I/O，采用这种直接驱动和观察输入/输出端口的方式，验证效率太低。

框架采用测试程序作为CPU功能验证的激励，即输入激励是一段测试指令序列，这个指令序列通常是用汇编语言或C语言编写、用编译器编译出来的机器代码。通过观察测试程序的执行结果是否符合预期来判断CPU功能是否正确。这样做可以大幅度提高验证的效率，但是验证过程中出错后定位出错点的调试难度也相应提升了。为了解决这个问题，框架在myCPU每条指令写寄存器的时候，将myCPU中的PC和写寄存器的信息同实现之前的模拟器的PC以及写寄存器信号进行比对，如果不一样，那么立刻报错并停止仿真。

## 实验要求

1. 根据本实验提供的五级流水线编程框架，在流水线 CPU 中添加以下指令：ADD、SUB、SLL、SLT、SLTU、XOR、SRL、SRA、OR、AND、ADDW、SUBW、SLLW、SRLW、SRLW、SRAW。
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. 撰写实验报告。撰写报告时要求叙述以下内容，以及你对本实验的思考与探索。
   1. 选择需要实现的指令中的一条，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
   2. 尝试自己绘制一幅属于自己的数据通路图。（注意：以后数据通路的添加都需要在该图上继续增加，因此打一个好的地基很重要，现在偷懒之后还是需要补的！）
   3. TODO：增加更多内容

## 实验步骤

1. TODO： 实验平台的使用
2. TODO： 如何上板
3. TODO： 如何打开工程文件进行编程
4. TODO： 如何使用模拟器进行仿真
5. TODO： 如何提交测评

## 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. SLT 和 SLTU 这类比较指令的实现是为了什么目的，比如是为了实现什么样的目标才有了这类指令？
4. SLL、SRL 和 SRA 这三条指令在 src2 高 63 至 6 位不全为 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，为什么要这样设计？可以对比 MIPS 指令集进行说明。
6. 为什么并不是所有的R型计算指令都有对应的字指令？
7. 请问差分测试框架只有这些debug信号就够了吗？假如有的指令不写回通用寄存器堆呢，这时框架又该如何发现问题？
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？

1. 这里的数据包指chisel中由Bundle实现的一个数据结构 [↑](#footnote-ref-1)