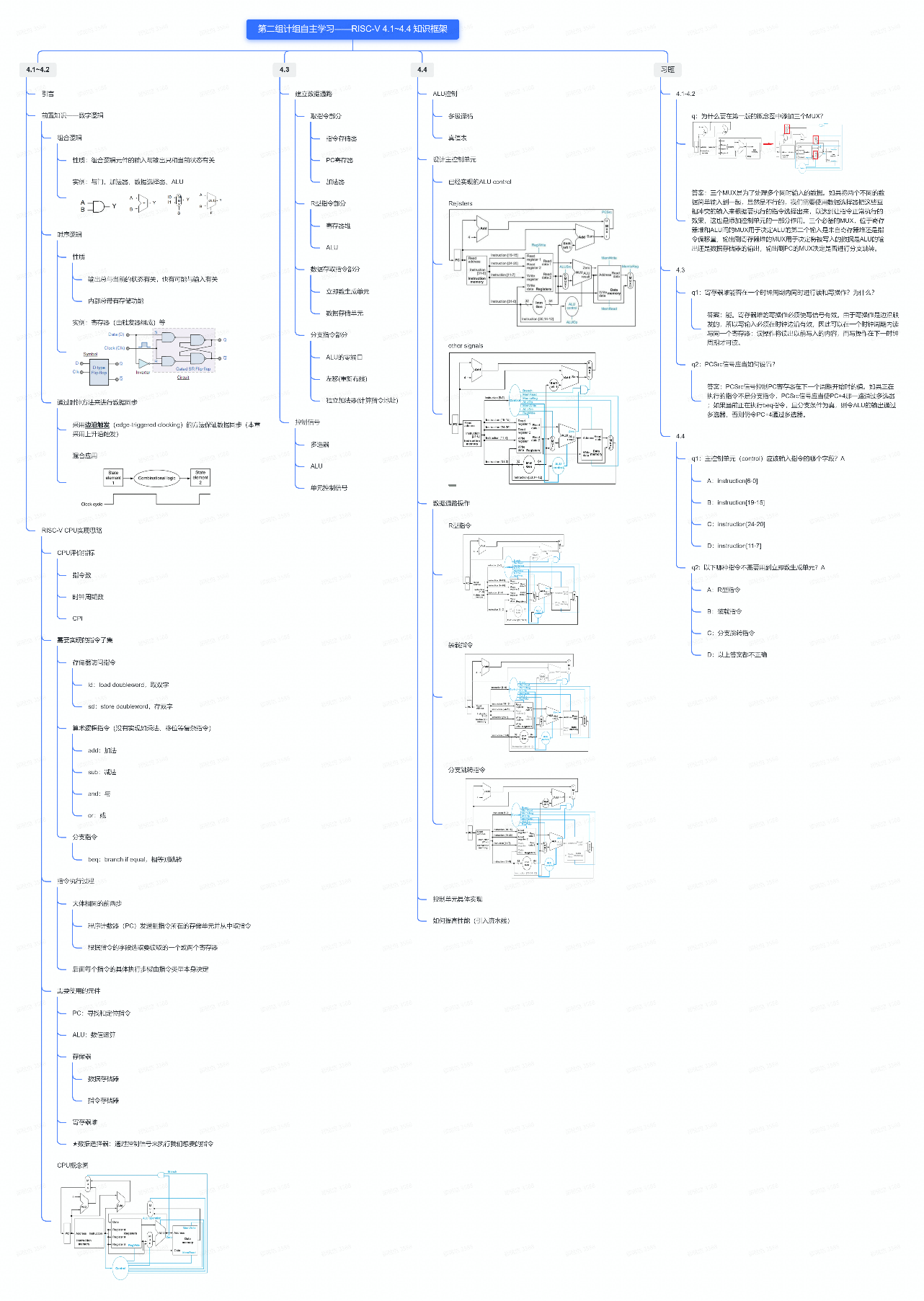
**计算机组成原理自主学习讲义——RISC-V 4.1~4.4**

**单周期CPU设计**

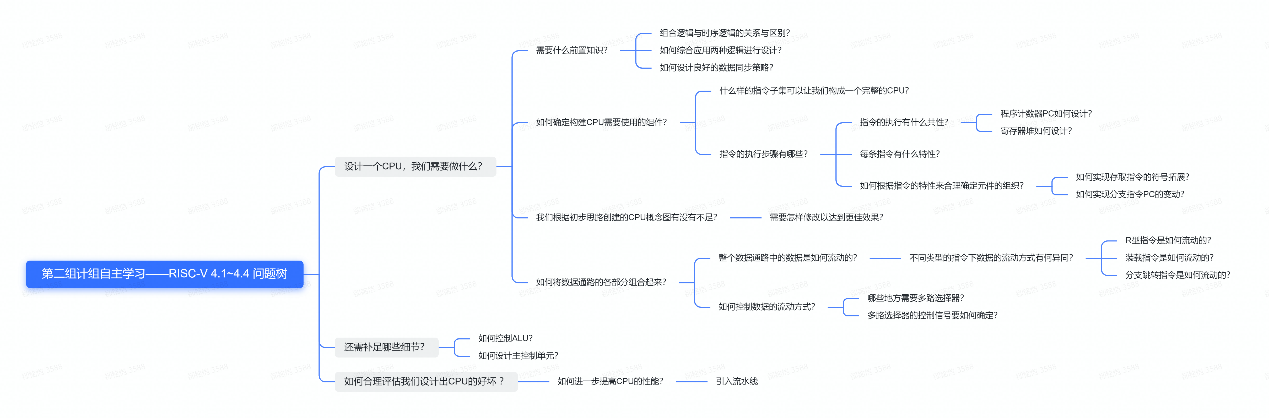
1. **知识框架图及问题树**

（一）知识框架图



Word中图片质量较低，若想访问原图请进入压缩包下文件或点击下面链接：

https://nankai.feishu.cn/mindnotes/bmncntOh6evQSMi3BbF9EqGqBRb（二）问题树



Word中图片质量较低，若想访问原图请进入压缩包下文件或点击下面链接：

https://nankai.feishu.cn/mindnotes/bmncnaidldeW5KzjNpZeEA27Gxc

**二、单周期CPU基本的RISC-V实现及逻辑设计的一般方法**

1. 引言

本章的内容是CPU，即中央处理器的有关内容。在本章中，我们可以学到关于CPU的一切知识，包括简单的单周期实现部分、流水线控制、指令间并行以及异常等等。本节中将从抽象的CPU模型出发，揭示逻辑设计的一般方法以及CPU内部的设计细节。

1. 前置知识——数字逻辑

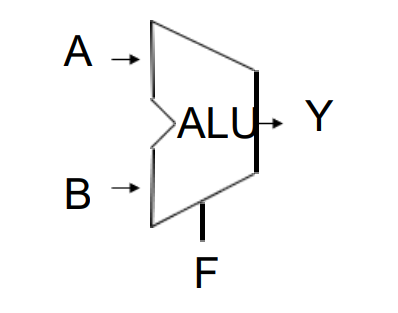
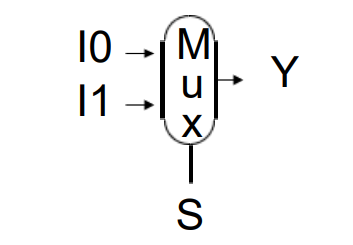
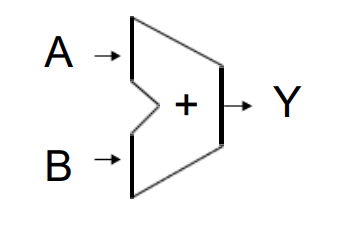
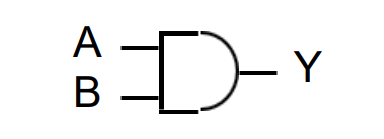
在CPU的设计之前，我们先对他的前置知识——数字逻辑以及逻辑设计的一般方法进行阐述。

众所周知，现代的计算机只能识别数字逻辑信号，即以电平的高低作为识别。一般来说，在正逻辑下，高电平记为1，低电平记为0，高低电平信号只分别对应一个比特位。如果需要多值输入的信号，我们就使用总线进行编码，之后进行传输。

1. 组合逻辑

我们需要通过对逻辑的设计来针对输入得到我们想要的输出，这个逻辑的载体就是我们所说的**组合逻辑元件**。组合逻辑元件的输入与输出只和当前状态有关，在CPU中，具备此特性的单元我们称其为**组合单元**。

典型的组合逻辑单元有：与或非门，加法减法器，数据选择器（MUX），以及**ALU**（arithmetic and logic unit，算术逻辑单元），这在后文中会提到。



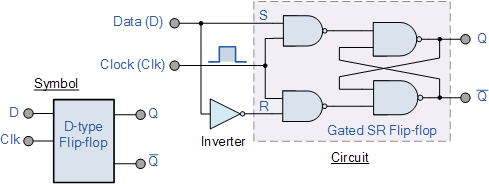
（从左到右分别是与门、加法器、两路数据选择器、ALU）

1. 时序逻辑

除了组合逻辑元件以外，还有一种**时序逻辑元件**，它的输出总与当前的状态有关，也有可能与输入有关，这种元件带有内部存储功能，会存储着状态。CPU中，具备时序逻辑元件特性的单元称为状态单元。可以说，CPU中的大多数组成部分都是状态单元，包括指令存储器、数据存储器与寄存器等。

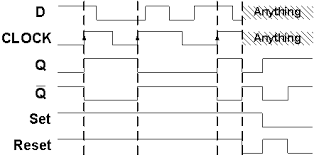
状态单元的一个典型样例是**寄存器**（register）。寄存器的功能是存储二进制代码，由多个触发器构成，一个触发器可以存储1位二进制代码，n个触发器可构成能存储n个bit的寄存器。

触发器是一个具有记忆功能的，具有两个稳定状态的信息存储器件，是构成多种时序电路的最基本逻辑单元，也是数字逻辑电路中一种重要的单元电路。



D触发器概览图

这里简单展示一下D-触发器。D-触发器会在时钟上升沿将输出替换为当前输入。我们在一个连续的时钟周期内，使用边沿触发的方式，使用由多个D触发器组成的寄存器，每次仅在时钟上升沿对数据进行更新，这样就实现了寄存器中对数据存储的效果。

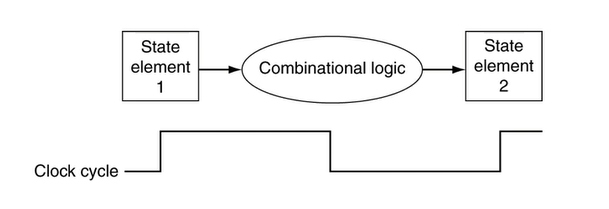


D触发器随时钟信号和输入的输出结果

1. 通过时钟方法来进行数据同步

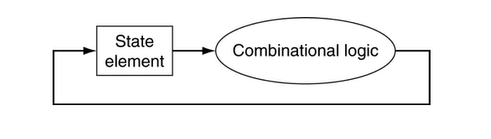
针对状态单元而言，我们一般使用“**时钟方法**”（clocking methodology）对其进行同步。时钟方法规定了信号可以读和写的时间，用于读与写之间的同步，以保证CPU可以正常工作。在本章中，我们采用**边沿触发**（edge-triggered clocking）的方法保证数据同步，即在每个时钟信号的跳变沿（可以是上升沿，也可以是下降沿，视情况而定）改变时序逻辑单元中存储的值，即写入。并不代表只存在边沿触发的方法，还有电平触发，即在特定电平时放开写入或读出的权限，为简单起见，本章中采用边沿触发的上升沿触发方式。

将以上两种元件和边沿触发方法结合应用，我们设计了这样一种结构，让两个状态单元错开一个时钟周期，使得在一个时钟周期的时间内，先在一个时钟上升沿读取前一个状态单元的结果，然后让中间的组合逻辑部分有足够多的时间来执行操作，最后在下一个时钟上升沿将处理好的数据传递给下一个状态单元。也就是说，在这个实例中，组合逻辑电路执行的最长时间决定了我们的时钟周期。



一种将组合逻辑与时序逻辑合并应用的思路

另一种实例是，可以支持状态单元在同一个时钟周期内同时读写而不会因竞争而出现中间数据。它的时钟周期必须够长，以确保时钟上升沿到来之前输入值稳定。我们在本章和下一章的设计中都采用类似本图片所示的结构。



另外一种实例

1. RISC-V CPU实现思路

计算机的性能取决于三个因素，分别为：指令数，时钟周期以及每条指令的时钟周期数（CPI）。处理器的实现方式决定了计算机的性能，也就是这三个因素。CPU一定由完成其功能的相应指令集的多条汇编指令而组成。我们尝试设计RISC-V的一个核心指令子集，作为我们的CPU，实现的指令如下：

* 存储器访问指令：load doubleword （ld）和load doubleword（sd）
* 算术逻辑指令：add、sub、and和or
* 分支指令：branch if equal（beq）

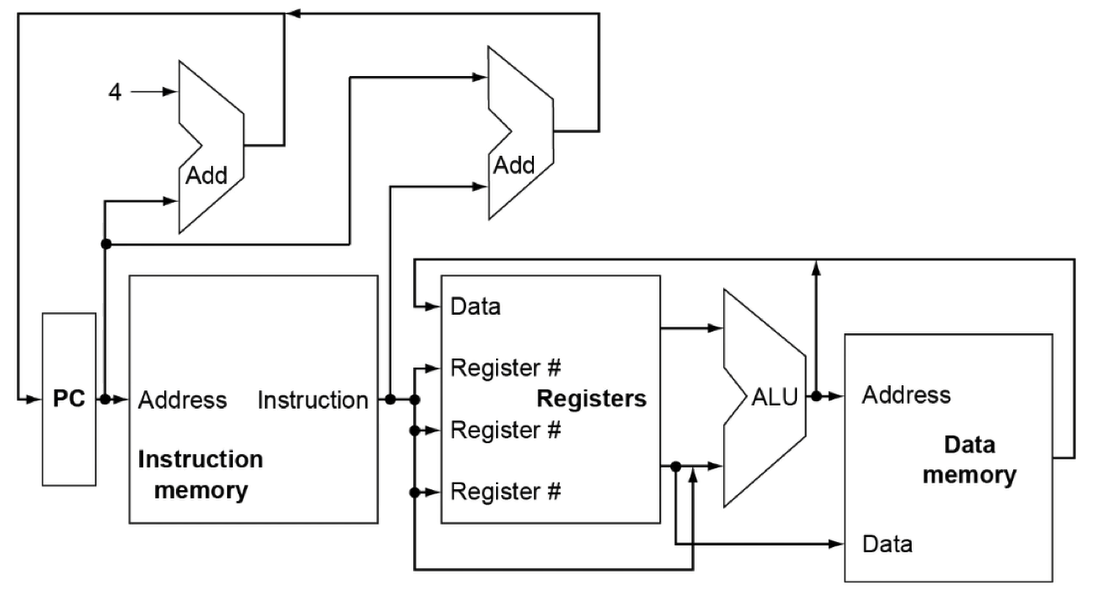
指令的大概执行过程，依照第二章中所学的知识，我们可以了解到，在RISC-V中，上述计算机指令的执行过程的前两步总是相同的，分别为：

* 程序计数器（PC）发送到指令所在的存储单元并从中取指令。
* 根据指令的字段选取要读取的一个或两个寄存器。对于ld指令而言，只需要读取一个寄存器，但是其他大多数指令都需要读取两个寄存器。

以上两步完成后，每个指令的具体执行步骤由指令类型本身决定，而对于以上三种指令而言，我们的剩余操作基本是相同的。RISC-V指令集的简单性与规整性使得不同类别的属性具有类似的执行过程，从而简化了实现。

所有类型的指令在读取寄存器后都使用了ALU，比如访问存储器的时候需要使用ALU计算地址，算术逻辑指令使用ALU计算结果，条件分支指令需要使用ALU进行条件的比较，甚至每次指令执行后，程序计数器自增4的地方也需要ALU辅助。

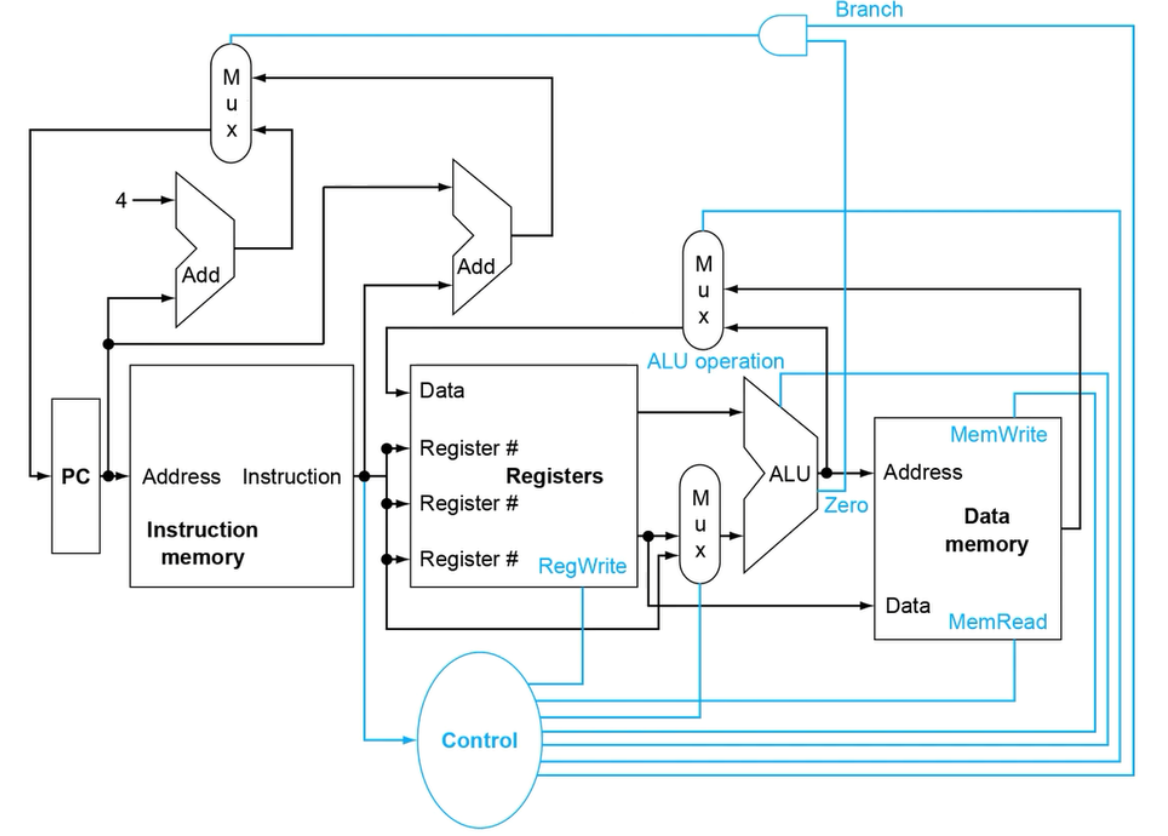
明确要实现上述指令的思路以后，我们需要选取能够完成这些任务的逻辑组件，共同构成一个简单的单周期CPU。对于我们需要采用的元件，首先是我们上文中提到的**程序计数器（PC）**，用于寻找和定位指令以及保障程序正常运行；然后是存储器，按照用途被分为指令存储器和数据存储器；我们还需要一个寄存器堆，来容纳寄存器；还需要一个ALU处理数值运算。基于此以及上文中描述的指令的执行步骤，我们可以设计一个RISC-V的CPU实现的抽象图：



RISC-V的CPU实现的抽象图

图中所有的粗线都是总线，这里不过多深究元件内部的细节，后面的章节会详细讲述它们的搭建。这张图中其实存在缺陷。我们观察到，这张图中的结构中会出现多个数据同时输入到一个单元中的情况，如果仅是简单地将它们连接到一起的话，显然是不合理的。因为在分支跳转语句beq中，我们无法确定什么时候将跳转后的地址还是PC+4存入PC中；我们无法确定取数语句lw在数据存储器中获得的结果，和通过ALU进行计算的指令在ALU中获得的结果中，哪一个结果应该被写入寄存器；我们无法确定

我们在这里引入**数据选择器（MUX）**以通过控制信号来执行我们想要的指令，并且为我们的抽象图添加一个控制单元，得到更新后的CPU概念图：



添加控制单元并使用MUX处理控制信号

图中添加了三个必备的MUX，位于寄存器堆和ALU间的MUX用于决定ALU的第二个输入是来自寄存器堆还是指令偏移量，输出到寄存器堆的MUX用于决定将被写入的数据是ALU的输出还是数据存储器的输出，输出到PC的MUX决定是否进行分支跳转。这样就基本完成了我们对CPU的顶层设计。本章的后续章节中将会在本张图的基础上继续改进。

**三、建立数据通路**

（一）引言

数据通路单元是一个用来操作或保存处理器中数据的单元。我们要设计的数据通路包括 寄存器、ALU（也就是算术逻辑单元）、多路选择器、内存。接下来我们进入设计的细节。

（二）取指部分

要执行任意一条指令，我们首先要从存储器中取出指令，然后增加程序计数器的值，使它指向下一条指令。由于RISCV的所有指令的长度都是4个字节，所以我们要让程序计数器增加4。

现在屏幕上显示的是取指令环节的硬件实现。它由三个部分组成，PC是程序计数器，最大的方块是指令存储器，最后还有一个加法器。

这一部分数据通路只进行读操作而不写入指令，因此指令存储器只提供读访问。每当一个时钟周期开始，PC向指令存储器提供指令地址，然后指令存储器把相应指令读取出来提供给数据通路的其他部分。每当时钟周期结束，它将加法器的结果写入到程序计数器中。由于这个写操作每周期都会发生，我们不需要一个额外的写信号控制它。

（三）R型指令部分

下一部分数据通路负责实现R型指令，也就是算术操作和逻辑操作。它包括寄存器堆和算术逻辑单元这两个部分。寄存器堆负责管理所有寄存器的读写操作，算术逻辑单元负责进行算术计算或者逻辑计算。

R型指令接收三个寄存器编号，包括两个读寄存器和一个写寄存器。它从两个读寄存器读取数据进行一些计算，然后把计算结果存入写寄存器中。

相应地，寄存器堆有三个端口，包括两个读端口和一个写端口。注意这几个箭头上标的数字5，它表示这些数据线传输5个二进制信号，5个信号刚好可以表示32个寄存器的编号。

每当一个时钟周期开始，寄存器堆从之前介绍的取指令部分获取3个寄存器编号，然后把两个读端口收到的编号的寄存器的数据读出来，从右边的read data输出到ALU。当这个时钟周期结束，它从ALU收到计算结果并将其写入写寄存器。

请注意寄存器堆的写操作由write data信号控制，仅在这个信号为真时执行。

（四）存取指令部分

接下来考虑RISCV的存取指令。存取指令包含两个寄存器x1, x2和一个偏移量。如果是取指令，它会从x2加上偏移量这个地址读取数据并把它存到x1中。如果是存指令，它则把x1中的数据存储到x2加偏移量这个地址处。

除了之前介绍的寄存器堆和ALU外，实现存储指令和载入指令还需要 数据存储单元 和 立即数生成单元。数据存储单元是一个状态单元，它有地址输入和写数据输入以及读取结果的单个输出。读信号和写信号是相互独立的。

立即数生成单元负责从指令中生成读写操作的偏移量，它以32为指令为输入，输出一个64位的有符号立即数。由于指令中仅仅编码了一个12位偏移量，立即数生成单元需要把它符号扩展到64位。

（五）分支指令部分

接下来考虑分支指令。Beq指令有三个操作数，其中两个寄存器用于比较是否相等，另一个是12位偏移量，用于计算相对于分支指令所在地址的分支目标地址。它利用之前设计的寄存器堆读取两个寄存器操作数，用立即数生成器从指令中读取立即数。

所以分支指令需要执行两个操作：计算分支目标地址和检测分支条件。

为了计算分支地址，数据通路额外包含一个加法器，它把立即数和程序计数器加在一起。

为了执行比较，它用ALU把两个寄存器相减，观察所得结果是否为零。

一个需要注意的地方是，立即数必须左移一位；这是因为RISCV以半字为单位进行跳转，偏移量必须是偶数。左移一位的操作在硬件上非常容易实现，在数据线最右侧添加一个0信号即可。

（六）完整的数据通路

我们已经讨论了几类不同指令的数据通路，现在我们要将其组合为一个完整的数据通路，并添加控制信号。其每周期执行一条指令，这意味着每周期，各单元只能被使用一次，如果需要使用多次同一单元，则需将其复制，因此我们将指令存储器和数据存储器分离设计。

R型指令和存储类指令的数据通路相似，区别在：

1）R型指令使用ALU时第二个操作数是寄存器，而存储类指令用于计算地址的操作数是一个12位偏移量经过符号扩展后的值。

2）存入目标寄存器的值来自于ALU(R型指令)或内存(载入指令)。

我们添加必要的多选器来建立存储类和算数逻辑类指令的数据通路。

多选器由控制单元控制，左侧的多选器决定操作数是来自寄存器还是扩展后的立即数。右侧的多选器决定写入寄存器的数据是来自存储器还是ALU的计算结果。

接下来我们将取指令数据通路，R型指令和存储类指令数据通路、分支指令数据通路合并，得到指令集的简单数据通路。由于分支指令使用主ALU来比较两个操作数是否相等，如前所述，我们需要复制单元来满足一个周期内的多次使用。我们设立一个专门的加法器来计算分支指令的目标地址，并增加一个多选器，用于决定下一条指令究竟是顺序执行，PC自增4还是分支跳转，转到目标地址。

立即数生成逻辑选择指令中要进行符号扩展的12位字段在不同指令中位于不同的字段。由于输入是完整的23位指令，可以使用指令的操作码选择合适的字段。RISC-V操作码的第6位在数据传输指令中为0，在分支指令中为1。第五位在载入指令中为0，在存储指令中为1.这样，操作码的第5,6位可以直接控制一个3:1选择器的输出，来选择不同指令立即数生成的合适字段。

以上是一个完整的单周期数据通路的实现。我们需要在此之上设计控制单元和控制信号，以完成CPU的设计。控制单元必须接受输入，生成每个单元的写信号，每个多选器的选择信号和ALU的控制信号。

**四、一个简单的实现机制**

（一）引言

上一小节中，我们已经得到了这样一个完整的数据通路，但还有一些细节没有解决。比如蓝色的这些控制信号应该如何从指令中得到？寄存器堆和符号扩展单元应该输入哪些字段？其中很重要的一点是，对于不同的指令，我们应该如何决定这个ALU（红色框标明）具体应该做什么。

（二）ALU控制

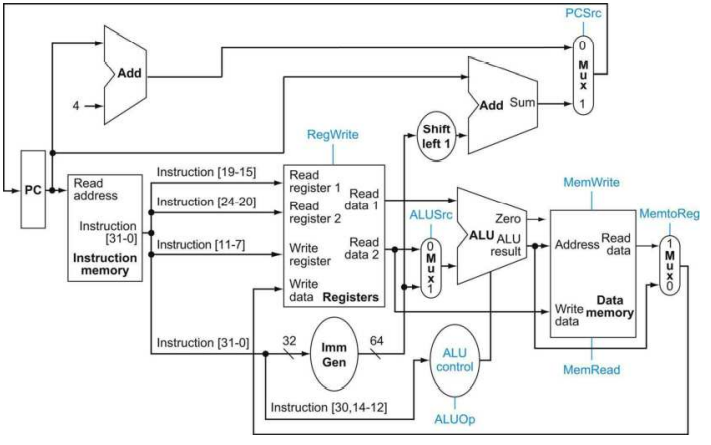
我们先来分析一下，对于RISC-Ⅴ的四种指令，ALU要做什么？对于载入和存储指令，需要add操作，对于跳转指令需要subtract操作，对于R型算术指令，需要根据funct7和funct3字段来具体确定做什么运算。对此RISC-Ⅴ ALU定义了四种控制线和对应的四种功能，如ppt的真值表所示。

这里采用了多级译码的方式，先用一个小的控制字段ALUop，指明要执行load/store(00)指令要做的加法，还是beq(01)要做的减法，还是R型指令(10)。如果是R型指令，再去看funct7和funct3字段设置的具体运算控制信号。这样一来可以减小主控制单元规模，减小延迟。

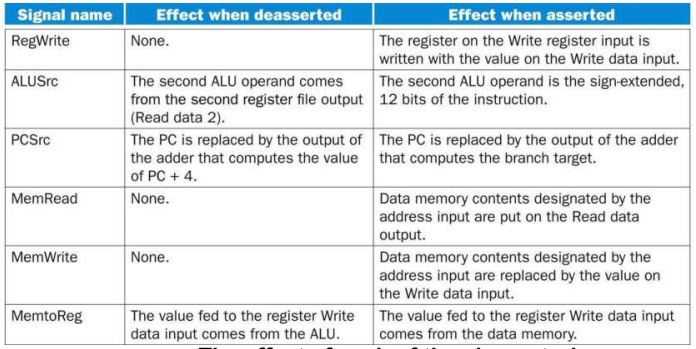
最终可以得到这个真值表，接下来只需要根据这个表机械化地生成门电路就可以了。

（三）设计主控制单元

截止目前，我们已经实现了较为复杂的ALU控制信号，关于寄存器堆和符号扩展的读入字段也非常容易看出。对于R型指令和分支指令，第一个寄存器操作数始终在15-19位，第二个寄存器操作数始终在20-24位，对于R型指令和载入指令，目标寄存器始终在7-11位。因此我们可以得到这样一个数据通路。



比较难以解决的是剩下的几个控制信号。我们对照数据通路一个一个来看：



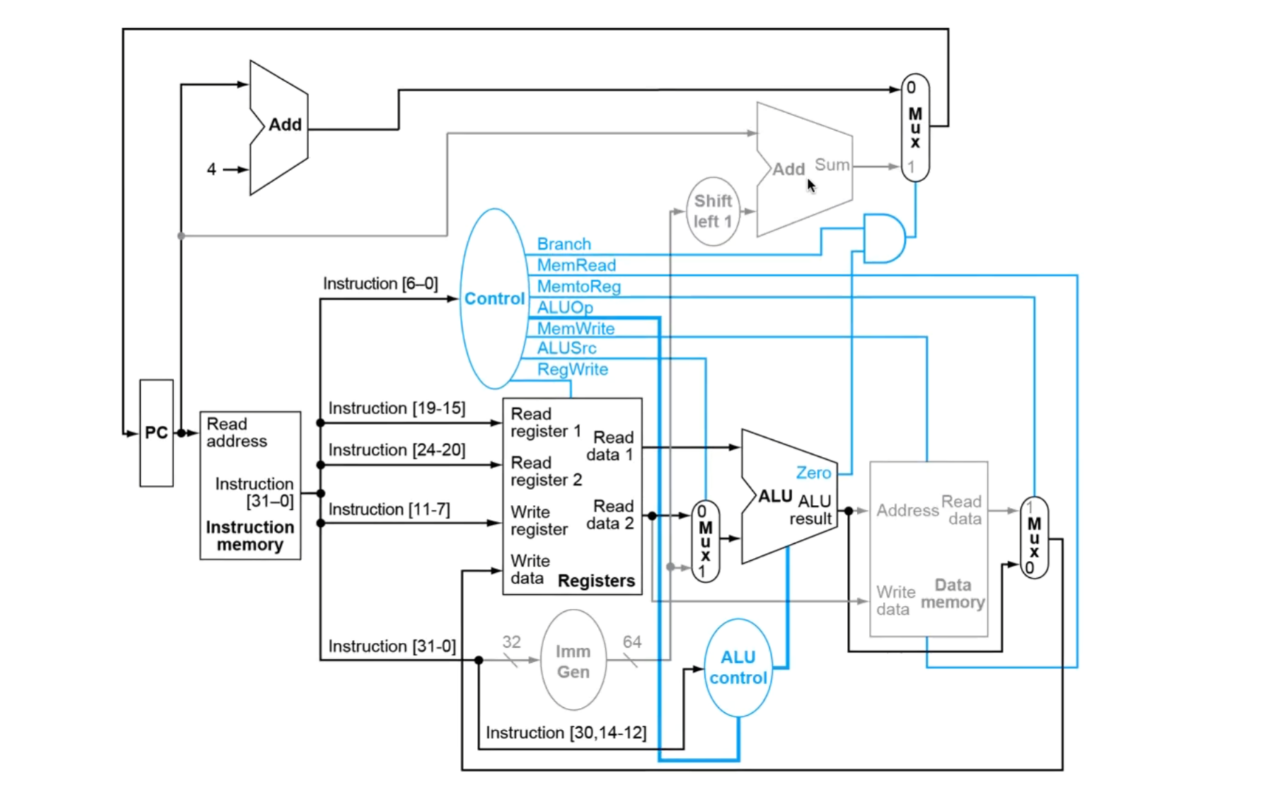
最终可以得到这样一个完整的简单数据通路

（四）数据通路操作

a）R型指令部分

上一小节中，我们已经得到了带有控制单元的简单数据通路。现在，我们来梳理一下各类型指令的数据通路操作。

R型指令的数据通路操作如下图所示，



虽然这些操作都发生在一个时钟周期内，但我们认为该指令该执行总共分为四个步骤。

1. 取出指令，PC自增。

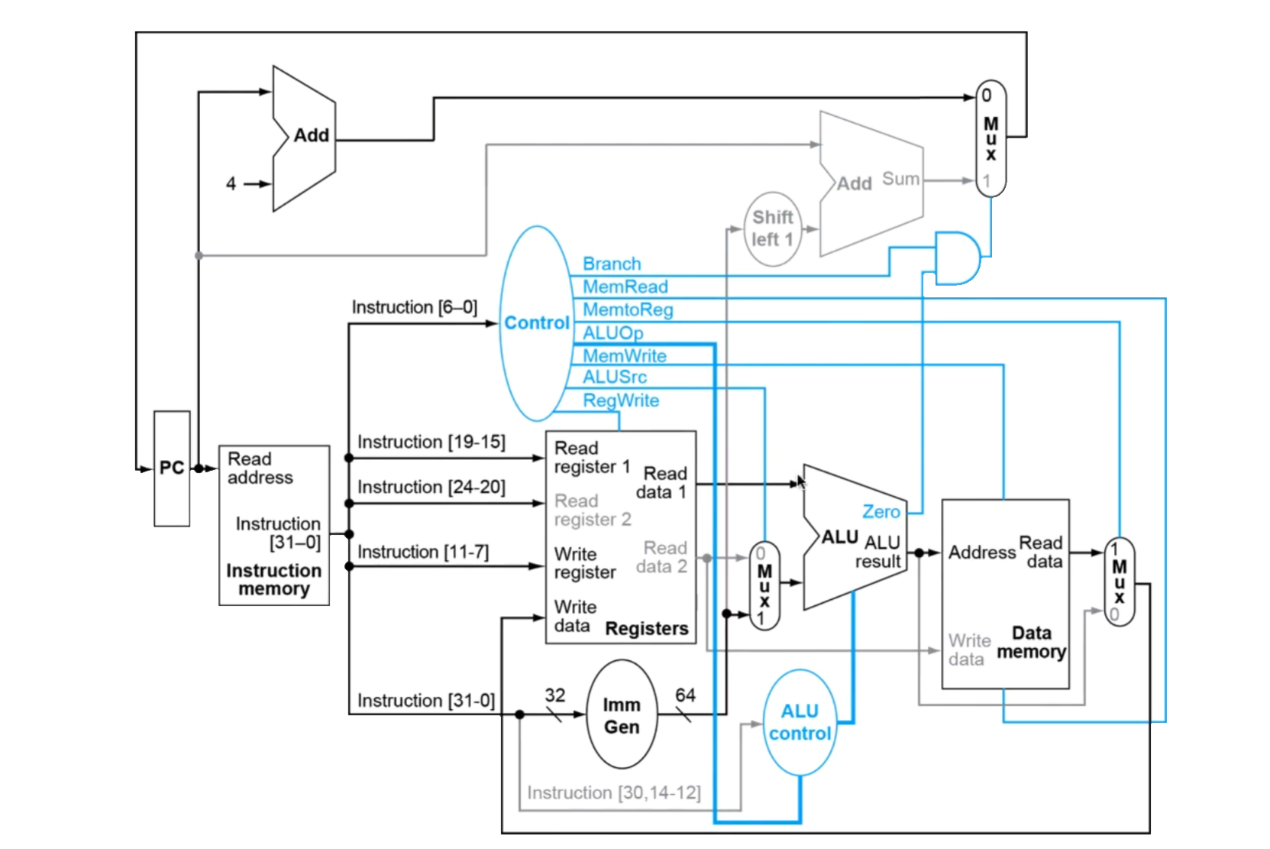
2. 从寄存器堆读出两个寄存器x2和x3，同时主控制单元在此步骤计算控制信号

3. 根据部分操作码确定 ALU 的功能，对从寄存器堆读出的数据进行操作。

4. 将ALU的结果写人寄存器堆中的目标寄存器(x1)。

b）装载指令部分

装载指令的数据通路操作如下图所示。



其具体执行步骤如下：

1. 从指令存储器中取出指令，PC自增。

2. 从寄存器堆读出寄存器(x2)的值。

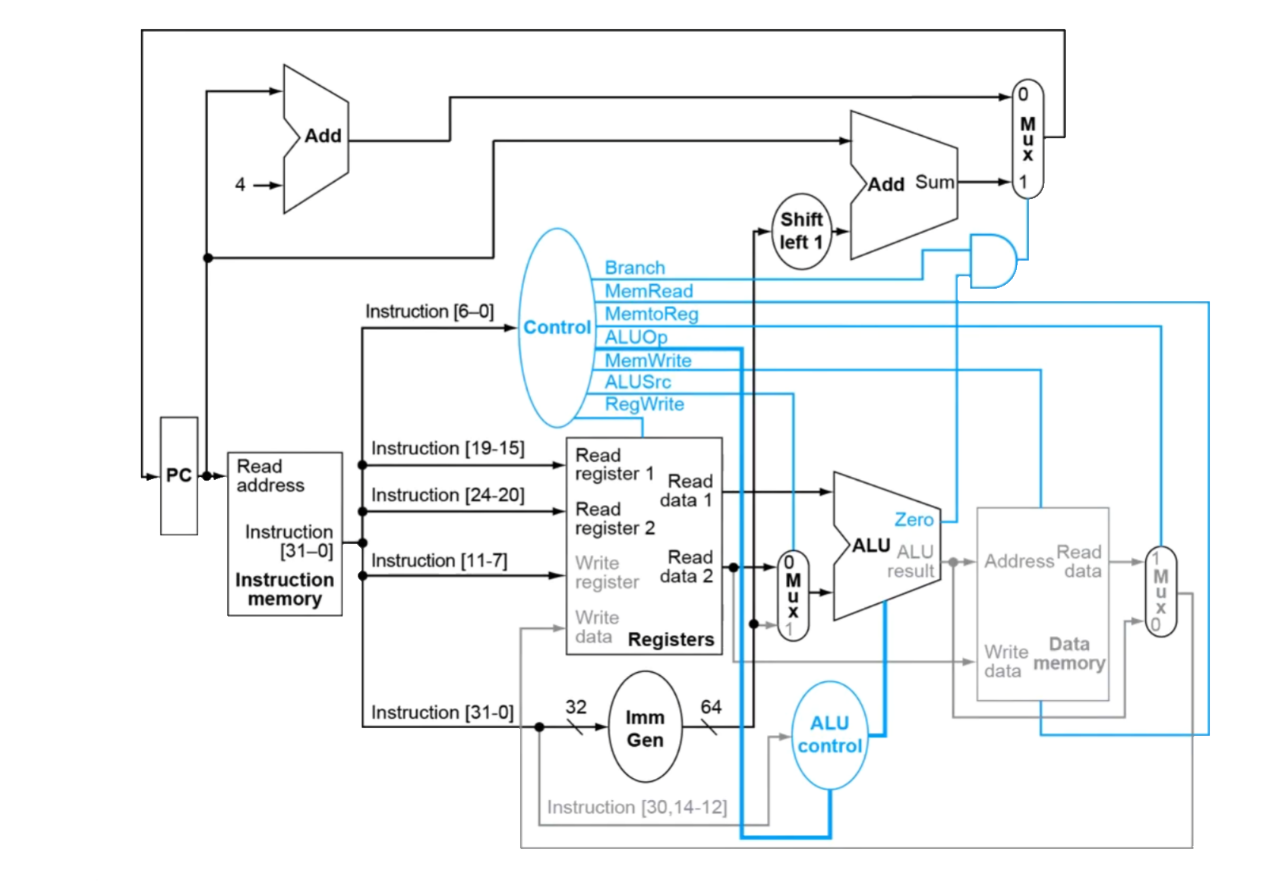
3. ALU将从寄存器堆中读出的值和符号扩展后的指令中的12位(偏移量)相加。

4. 将ALU的结果用作数据存储器的地址。

5. 将从存储器读出的数据写入寄存器堆(x1)。

c）分支跳转指令部分

分支跳转指令部分的数据通路操作如下图所示。



其具体执行步骤如下：

1. 从指令存储器中取出指令，PC 自增。

2. 从寄存器堆中读出两个寄存器x1和x2。

3. ALU将从寄存器堆读出的两数相减。PC与左移一位、符号扩展的指令中的12位(偏移)相加，结果是分支目标地址。

4. ALU的零输出决定将哪个加法器的结果写入PC。

（五）控制单元具体实现

那么，我们已经了解了各类指令的数据通路操作。那么，控制单元又是如何实现的呢？

如下图所示，控制单元的输入是opcode，输出是各种控制信号，其真值表如右图所示。根据这个真值表，我们可以设计对应的组合逻辑电路，完成控制单元的具体实现。



（六）如何提高性能

在前面的内容中，我们已经理解了单周期cpu的设计思路。但是，这样设计出的处理器性能是不足的，那么如何提升性能呢？我们知道，cpu的性能很大程度取决于单位时间内执行的指令数。对于单周期cpu而言，一个周期执行一条指令，每个周期需要是等长的，因此单个周期的时长取决于执行最慢的指令。一个典型的关键路径是load指令。

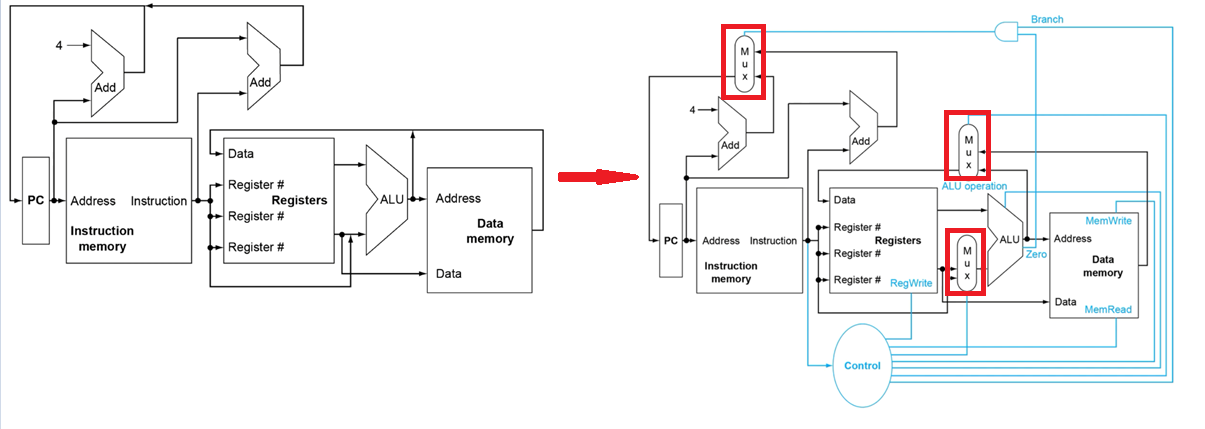
其执行步骤为：指令存储器→寄存器堆→ALU→内存→寄存器堆。根据我们的设计原则：加速大概率事件，这一事件是值得加速的。因此，我们可以引入流水线这一概念，设计出多周期cpu，最终提升cpu的性能。

1. **实验：指令译码器**

详见实验——指令译码器文档。

1. **习题**

（一）为什么要在第一版的概念图中添加三个MUX？

****

答案：三个MUX是为了处理多个同时输入的数据。如果将两个不同的数据简单输入到一起，显然是不行的，我们需要使用数据选择器把这些互相冲突的输入来根据要执行的指令选择出来，以达到让指令正常执行的效果，这也是添加控制单元的一部分作用。三个必备的MUX，位于寄存器堆和ALU间的MUX用于决定ALU的第二个输入是来自寄存器堆还是指令偏移量，输出到寄存器堆的MUX用于决定将被写入的数据是ALU的输出还是数据存储器的输出，输出到PC的MUX决定是否进行分支跳转。

（二）寄存器堆能否在一个时钟周期内同时进行读和写操作？为什么？

答案：能。寄存器堆的写操作必须使写信号有效。由于写操作是边沿触发的，所以写输入必须在时钟边沿有效，因此可以在一个时钟周期内读写同一个寄存器：读操作将读出以前写入的内容，而写操作在下一时钟周期才可读。

（三）PCSrc信号应当如何设置？

答案：PCSrc信号控制PC寄存器在下一个周期开始时的值。如果正在执行的指令不是分支指令，PCSrc信号应当使PC+4那一路通过多选器；如果当前正在执行beq指令，且分支条件为真，则令ALU的输出通过多选器，否则仍令PC+4通过多选器。

（四）主控制单元（control）应该输入指令的哪个字段？

A：instruction[6-0]

B：instruction[19-15]

C：instruction[24-20]

D：instruction[11-7]

答案：A

（五） 以下哪种指令不需要用到立即数生成单元？

A：R型指令

B：装载指令

C：分支跳转指令

D：以上答案都不正确

答案：A