第四章 处理器

1. 基本的MIPS实现

(1) MIPS 指令集：

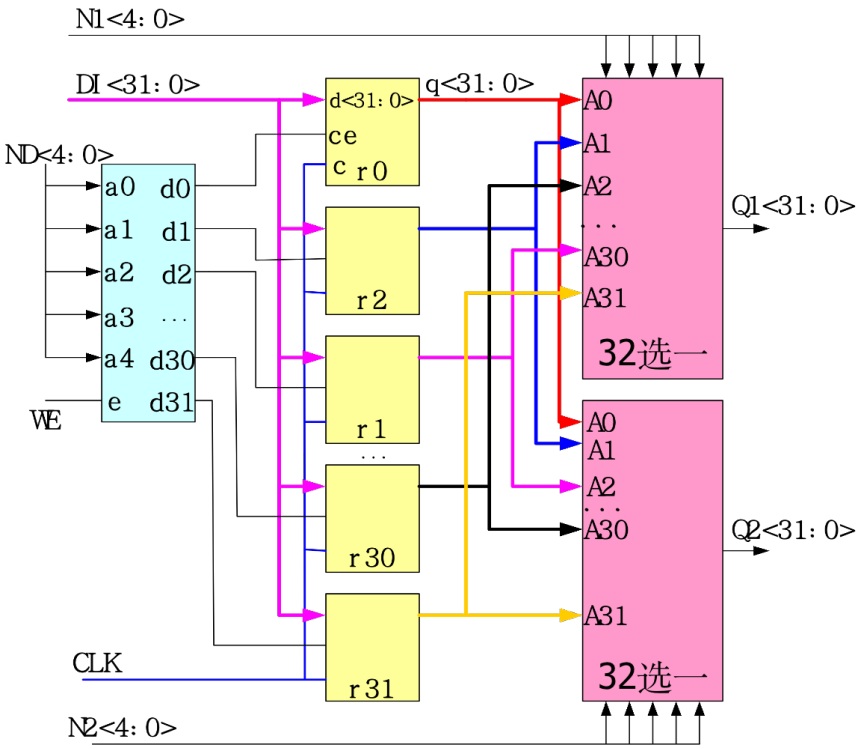
指令类型	指令	作用
存储器访问指令	lw, sw	取字、存字
算术逻辑指令	add, sub, and, or, slt	加减与或，小于则设置
分支指令	beq, j	相等则跳，无条件跳转

此为MIPS核心子集，用他们可以实现其他的指令。在第一章已经了解到了，指令集的实现会影响计算机的时钟速度和CPI（后续可看到）。

(2) 基本器件：

- 最基本的运算器件：与或非门，ALU运算器以及多路选择器。
- 寄存器（边沿触发，带读写使能端）
- 寄存器堆：多个寄存器放在一起，就构成了寄存器堆。寄存器堆是由地址译码电路和多路选择器来构成，要写哪个寄存器,数据送数据总线DI, 将编号送入地址译码电路中；要读哪个寄存器就将地址送多路选择器，将对应的寄存器送出到数据总线上。

  如下图所示，就是32位寄存器堆，有两个读端口和一个写端口的寄存器堆。实现时的逻辑如下面代码片所示：



```
always@(negedge CLK)     //在下降沿是改写寄存器
     begin
        // 如果寄存器不为0，并且RegWre为真，写入数据
        if (RegWre && WriteReg != 0)  register[WriteReg] = WriteData;
     end
```

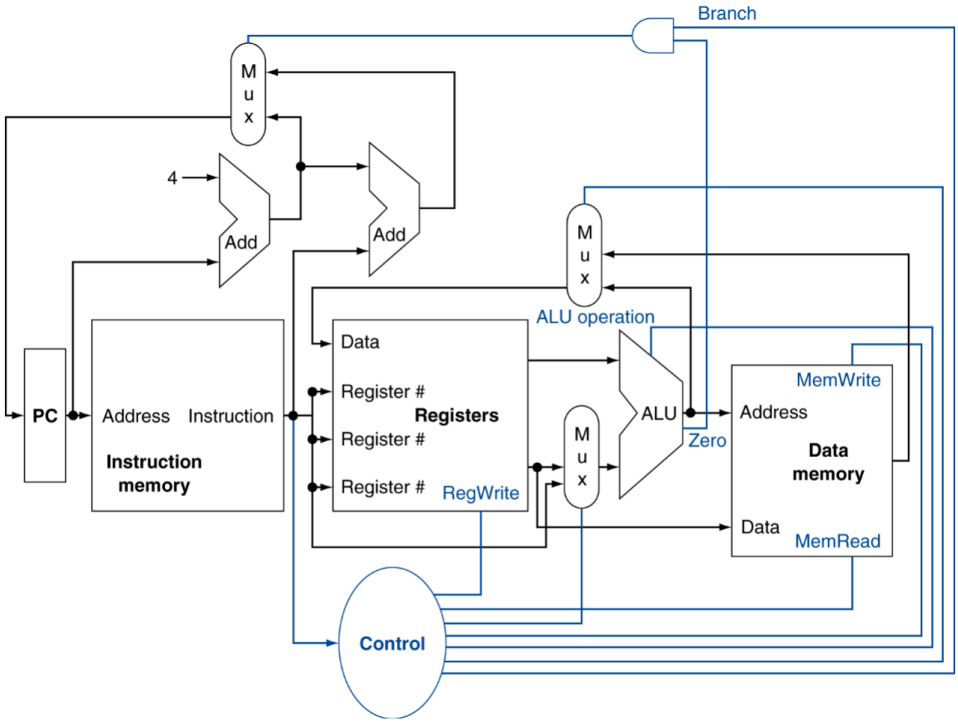
(3) 实现方法概述：

对三种指令的类型（存储器访问、算术逻辑、分支）动作都是大致相同的，MIPS格式相同，规整，使得每类指令之间的差别并不会太大，并且所有指令的执行，最前面两步一定是一样的：

> 1. Program Counter 指向指令所在的存储单元，从中读取出命令
> 2. 通过指令内容，选择读取寄存器中的数据。

书中图4-1只是一个对处理器的抽象图，只考虑各期间之间的关系，没有考虑数据的来源选择，这类选择通过多路选择器即可完成。

下图中增加了必要的多选器（MUX），此外还有一个控制单元（Control Unit）。可以发现，控制单元的输入是指令，它可以对指令进行解码，并且将相关的控制信号设置为有效。



三个多路选择器的作用（从图片的上到下编号为1~3）：

> 1. 控制对PC写入PC+4的结果或是跳转指令要求的结果，由branch门控制，branch是从Control中引出来的，是对指令做解码知道指令含义以后做出的操作；
> 2. 控制选择被写入寄存器堆中的是ALU的输出（算术指令）还是Data Memory的结果（取数指令）；
> 3. 控制ALU加法的第二个字段是来自寄存器还是指令的偏移量字段。这里应该还有立即数要考虑在内。

指令控制Control输出部分的依据：MIPS指令集架构。通过分片段解析指令，知道要做什么指令，从而控制不同的信号值的改变。

2. 逻辑设计的一般方法

(1) 实现所用的部件：

$$\text{分类：}\begin{cases}\text{组合单元：只取决于当前的输入(没有内部存储功能)}\\ \text{状态单元：带有内部存储功能（掉电不易失）}\end{cases}$$

状态单元需要输入有*写入的数据值*和*时钟信号*（决定何时写入），最简单就是由D触发器实现。此处复习一下D触发器：

```
module dff(
  input wire d,
  input wire clk,
  output reg q
  );
  always@(posedge clk) begin
    q <= d;
  end
endmodule
```

对于PC，可以不写输入的值，内部initial为9'b0，在PC内使用加法器，每次指令传进来以后就做+4的操作并在下一个上升沿的时候赋给q。

(2) 时钟控制：本书采用边沿触发（上升沿）

> Q1: 为什么要使用时钟周期?
> A1: 若有一个信号同时被读写，则所读的信号有可能是以前的值，产生冒险。为了避免这种事情发生，用边沿触发。如果不是每个周期都变的话，记得用控制信号做控制。

3. 建立数据通路（单周期）

数据通路是指CPU中的各种原件和部件的连接方式。数据通路部件是数据在处理器中流动需要经过的部件，他们操作或者保存处理器中的数据。

(1) 数据通路部件

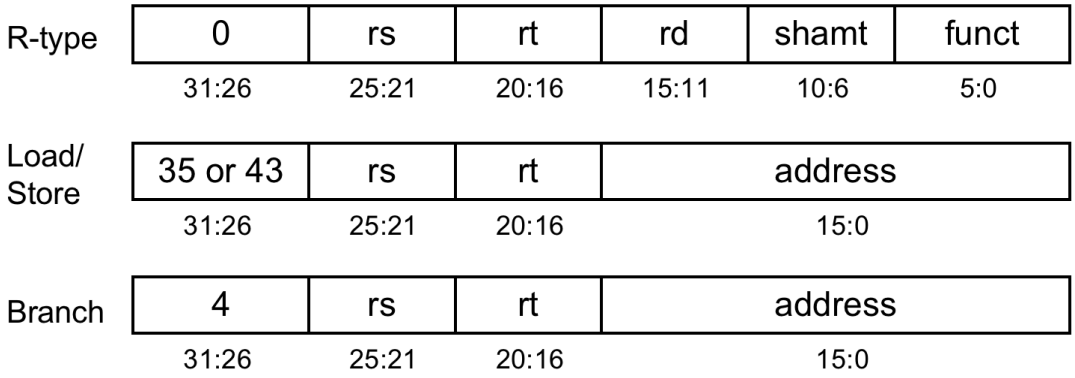
在MIPS实现中，数据通路部件包括指令存储器，数据存储器，寄存器堆，alu和加法器。

程序计数器PC：PC用来保存当前指令的地址，在内部，我们用一个加法器对PC的值进行加，并且传给PC的输入，使得下一次执行的PC值发生变化，IR取出下一地址中指令的内容。

> Q2: 为什么PC总是+4，而不是+1?
> A2: 这和处理器的寻址方式有关，+4的操作是适用于32位mips处理器的，MIPS32位CPU，一条指令32bit=4Byte，寻道址方式为按字节寻址（每个字节加1），结果就是PC+4.

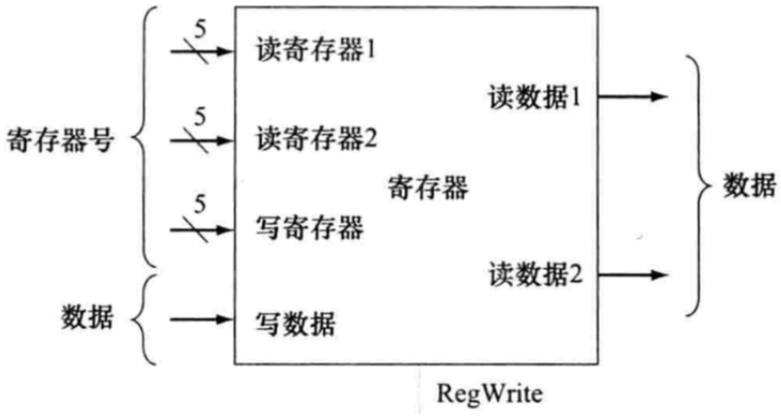
寄存器堆(register file)：包含了一系列寄存器的状态单元，可以通过提供寄存器号进行读写。这里面放有计算机的寄存器状态。寄存器堆的构造可以参考本章节的第一张图。

(2) 三类指令的执行过程



a. R-Format Instructions: 读两个寄存器，对内容进行操作（在ALU中执行）再写回到寄存器中。读两个寄存器，需要被读的寄存器号和输出指示；写入寄存器的时候，需要被写入的数据和对应寄存器号。被读取的数据放在寄存器堆regfile里面，regfile的工作如下：

从指令中获取rs，rt 进行读，输出rs 和 rt的数据到输出，两个输出的数据进入alu中进行了一系列操作后要写回到regfile里，已经从指令中知道被写的地址是rd，那么写入的数据就是从alu的运算结果接过来。regfile的控制信号是regwrite。结构如下：



b. Load/Store Instructions: 通常是将寄存器的内容与指令中的16位带符号偏移地址相加，得到目标寄存器的地址。存储指令从寄存器中读出要存储的数据，取数指令将从**存储器**中读取数据存入寄存器中。

注意到前面提到的偏移地址为16位带符号偏移地址，它需要做符号扩展(sign -extend)为32位带符号值，以及一个保存读出或写入数据的存储单元。符号扩展用verilog语言实现如下：

```
module signed_extend(
  input wire[15:0] a,
  output wire[31:0] y
    );
  assign y = {{16{a[15]}},a};
endmodule
```

c. Branch Instructions：也要对后面的指令偏移量做符号扩展再加到PC值上。

- 指令及规定计算分支地址的时候使用的基地址是分支指令的下一条指令的地址，意思是：假设当前执行的指令地址为PC，那么在执行beq指令的时候，实际上是对PC+4来计算指令要求的偏移量得

到的目标地址。

- ”系统结构还规定偏移量左移2位以指示以**字**为单位的偏移量，这样偏移量的有效范围就扩大了4倍。”（按字节寻址，指的是存储空间的最小编址单位是字节，按字编址，是指存储空间的最小编址单位是字。） 从下表我们可以看出，mips因为是采用了字节编号，所以读取第1个数(byte4)的时候，实际上是要跳转1 * 4 个字，也就是偏移(1<<2)个字才能到达。因此是要对偏移量做2位的左移的。这一点可以复习第二章。

	byte 0	**byte 1**	**byte 2**	**byte 3**	**byte 4**
地址(字节)编号：	0	1	2	3	4
字编号：	0	x	x	x	1

对于这样的指令，当分支条件为真时，由(PC + 4)和”offset先做符号位扩展再左移2位“共同形成的地址将成为下条指令，否则，PC+4为下一条指令。所以，分支数据通路要做两件事：

$$\begin{cases} \text{计算分支目标地址：由符号扩展单元和加法器完成} \\ \text{比较操作数：寄存器堆提供两个寄存器操作数作比较（不需要对寄存器堆写，只需要读）} \end{cases}$$

比较操作数，我们可以用到alu的减法操作，两个操作数做减法，如果结果为0，说明两个equal。我们在alu中添加了一个输出信号zero来输出判断的结果。

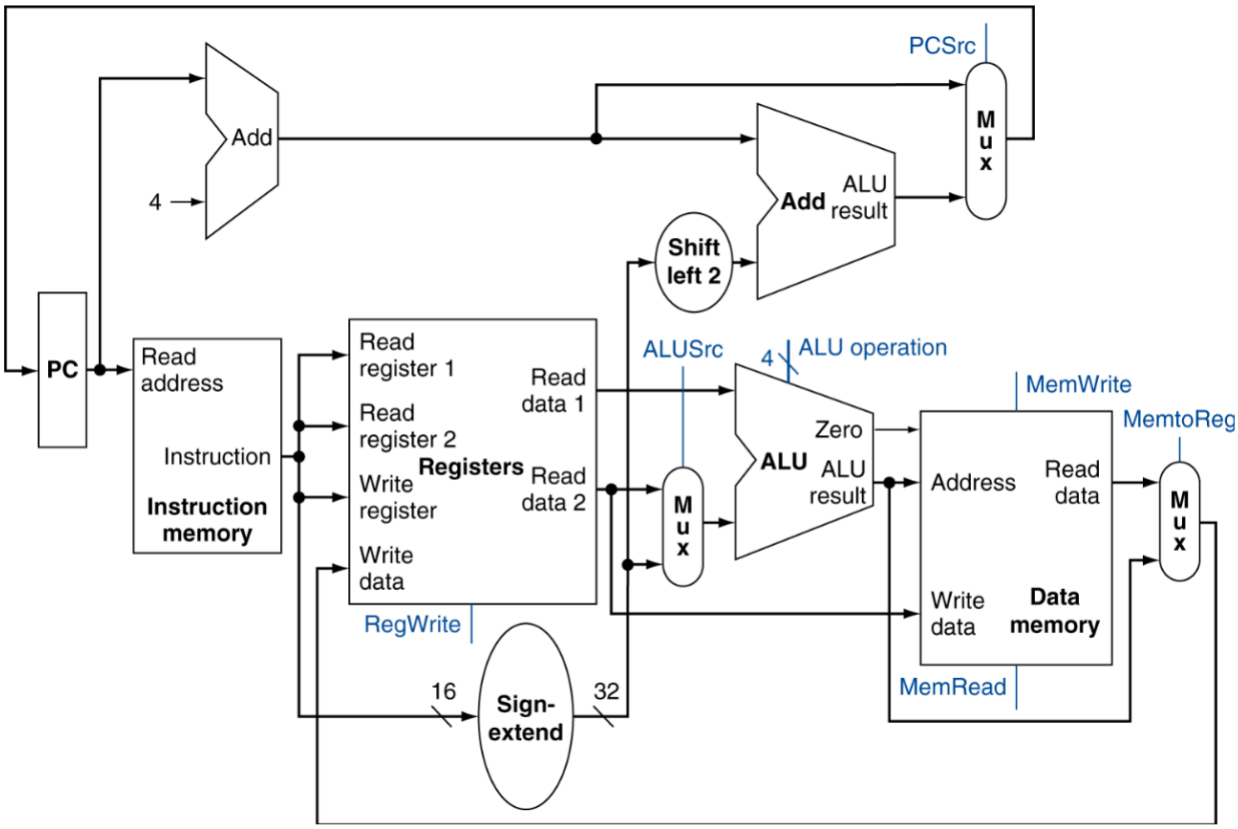
注意：对于数据存储器，读、写的控制信号都是独立的，尽管任何时钟只能激活其中一个。数据存储器是用来存放计算时用的一些数据。

分支指令的延迟：无论分支条件是否满足，他之后的那条指令总要被执行。

(3) 信号有效的含义

控制信号名	无效时的含义	有效时的含义
RegDst	写寄存器的目标寄存器号来自 rt 字段（位 20：16）	写寄存器的目标寄存器号来自 rd 字段（位 15:11）
RegWrite	无	寄存器堆写使能有效
ALUSrc	第二个 ALU 操作数来自寄存器堆的第二个输出（读数据 2）	第二个 ALU 操作数为指令低 16 位的符号扩展
PCSrc	PC 由 PC +4 取代	PC 由分支目标地址取代
MemRead	无	数据存储器读使能有效
MemWrite	无	将写入数据输入端的数据写入到用地址指定存储器单元中取
MemtoReg	写入寄存器的数据来自 ALU	写入寄存器的数据来自数据存储器

简单的单周期cpu数据通路图：



注意，这个数据通路图中，每一个时钟周期执行完一条完整的指令。

总结四种指令在单周期cpu中的执行流程

对于R-type的指令，流程：

> 1. 读pc， pc + 4
> 2. 解码instruction，读Reg里面对应的操作数的值
> 3. 结果给alu和alu_control
> 4. alu中计算，结果直接写回Reg

对于Load指令，流程：

> 1. 读pc， pc + 4
> 2. 通过指令内容，选择读取寄存器中的数据
> 3. 对offset偏移量做符号扩展
> 4. 用alu计算offset偏移量和pc+4的结果，为偏移地址
> 5. 输入到data memory，读取数据，写回到reg
>
> （Store指令5就是把Reg里面的内容写到存储器）

Branch-on-Equal （条件跳转）指令，流程：

> 1. 读pc， pc + 4
> 2. 通过指令内容，选择读取寄存器中的数据
> 3. 对offset偏移量做符号扩展
> 4. 在alu中做比较，判断是否跳转
> 5. (跳转控制为真)? (下条指令为PC+4+offset_extend<<2)：PC+4;

Jump（无条件跳转）指令，流程：

1. 读pc， pc + 4
2. 通过指令内容，选择读取寄存器中的数据
3. 下条指令为pc高四位与offset <<2之间拼接

4. 流水线CPU

4.1 Pipelined MIPS Processor

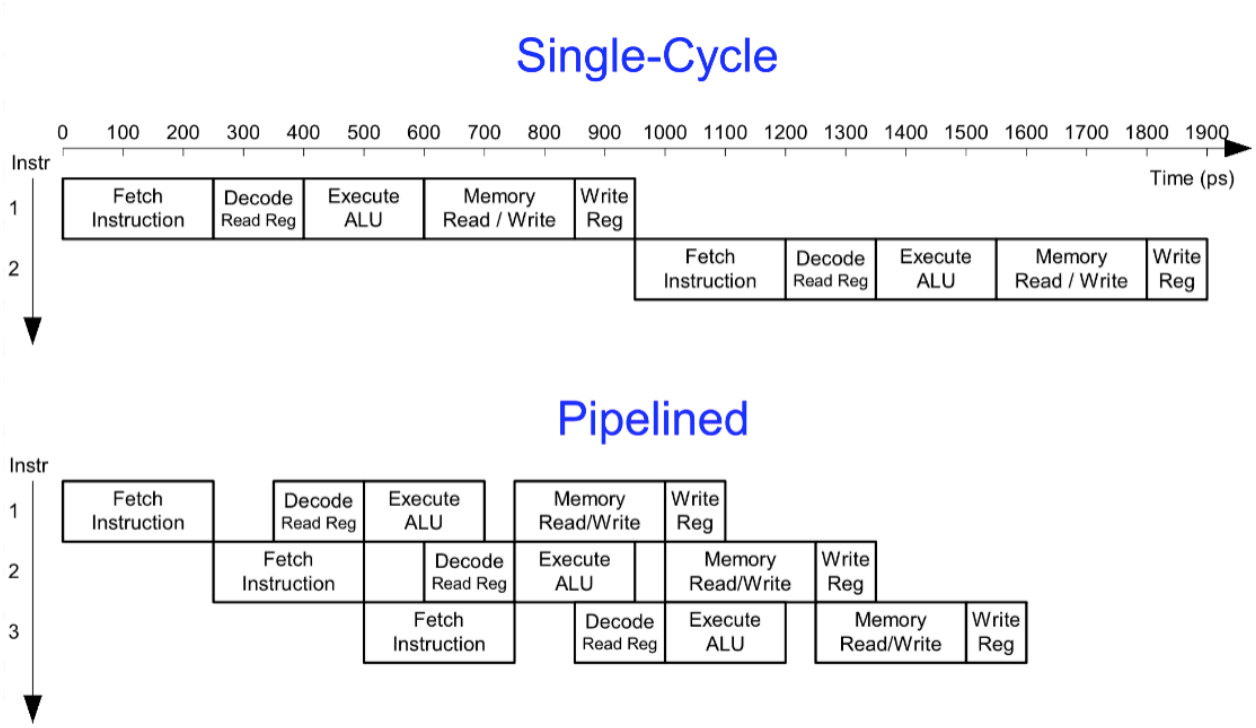
Temporal Parallelism 五个阶段：Fetch, Decode, Execute, Memory, Writeback.通过在流水线中加入寄存器来保存状态。用流水线可以提升操作的效率。

在理想情况下，流水线机器上的指令执行时间为：

$$指令执行时间_{流水线} = 指令执行时间_{非流水线} / 流水线级数$$

4.2 Single-Cycle vs. Pipelined

使得平均每条指令的周期数为1，从而cpu速度提升。（理想状态）



单周期流水线，就如实验2正在做的内容，是一线到底的数据通路，其每一阶段操作的内容如上图的single-cycled所示。对于流水线处理器，每一阶段都用一个寄存器来保存数据，指令寄存器通过译码得到中间结果（比如，opcode，funct等）读取也放入到寄存器。

对于流水线，以第一周期为例，当他读好了pc以后，就可以把数据传给下一级，同时开始下一次读取pc，就会快。由此，指令之间的间隔变成了单周期cpu的$\frac{1}{5}$.

时钟的上升沿进行写（从buffer里面写入），后半个周期，在下降沿的时候进行寄存器中内容的读取。

4.3 Pipelined Processor Control

每个阶段用到的控制信号相对来说会少一些。

4.4 Pipeline Hazards

- 结构冒险：某两个/多个指令要用到同一个部件，比如regfile。可以用硬件结构分成两半。
- 数据冒险：寄存器存放的时候读写矛盾. 解决数据冒险可以使用前推动的方法，即提前取出已经得到了的数据。

在计算机流水线中，数据冒险是由于一条指令依赖于更早的一条还在流水线中的指令造成的（这是一种在洗衣店例子中不存在的情况）。例如，假设有一条加法指令，它之后紧跟着一条减法指令，而减法指令要使用加法指令的和（$s0）：

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

在不做任何干涉的情况下，这一数据冒险会严重地阻碍流水线。加法指令直到第五步才能写回它的结果，这就意味着在流水线中浪费了三个时钟周期。

- 控制冒险：下一条指令的决策取决于上一条指令执行的结果。

4.5 handling data hazards

法1: 加入nops指令，等待几个周期，直到数据已经更新写入，再读入。(效率比较低)
法2: 编译器调整代码顺序。
法3: **数据前推**: 感觉是把上面的运算结果提前给到下一个周期用。

> 通过连线把数据从data memory里面取出来，然后通过多路选择器来决定它进入到alu的哪一端。
>
> 如果处于执行阶段，并且后面一条指令需要的寄存器编号和当前指令正在写的寄存器编号相同，就把当前的数据推过去。用多路选择器。
>
> however，读内存这种指令是没办法前推的。因为第四个周期结束快要进入第五个周期的时候才能得到内存里新写入的数据，而下一条指令第一周期就要用，因此，不能用数据前推。这时候就用stall（挂起）或者用nops指令,这里注意因为挂起了，所以先读了的一些指令不要了，做flush操作清掉。
>
> 由load word指令，可以引起暂停。

法4: 把processor挂起，暂停（如果发现要用到没有更新的数据）相当于是动态地插入了nops指令。

4.6 handling control hazards

beq的指令很多，但是它也比较复杂因为“branch not determined until 4th stage of pipeline”. beq在进入第四个周期之间才能知道是否要跳转，但是他后面已经有三条指令进入流水线了，如果要跳转的话，那就要把下面三条指令都给清空。

在译码阶段就判断是否要跳转。可以预测跳转（概论统计：每隔7条左右就有一个跳转指令）J（无条件跳转）第二周期就跳了，没有问题，所有的jump指令都要flush掉他的下一个指令。

在译码阶段就需要得到是否跳转的结果，如果要跳转，已经在流水线里开始的

～前推～

分支预测：一种解决分支冒险的方法。它预测分支结果并立即沿预测方向执行，而不是等真正的分支结果确定后才开始执行。

5. 多周期CPU

5.1 多周期cpu的优点

- 周期时间比较短
- 不同的指令可以使用不同的周期数来完成

- 装入指令 LW 需要5个周期
- 跳转指令仅仅需要3个周期
- 允许每条指令多次使用同一个功能部件

小测验答案

4.1 控制器、数据通路、存储器。少了输入和输出。
4.2 错。边沿触发状态单元可以同时进行读写。
4.3 I. a；II. c。
4.4 是，Branch 与 ALUOp0 是相同的。而且，MemtoReg 和 RegDst 是相反的，不需要额外的反相器。仅使用另外一个信号，并翻转多路选择器的输入即可。
4.5 Ⅰ. 因为 `lw` 的结果而阻塞；Ⅱ. 旁路第一个 `add` 的结果写入 `$t1`；Ⅲ. 不需要阻塞或旁路。
4.6 2 和 4 正确，其余错误。
4.8 1. 预测不发生；2. 预测发生；3. 动态预测。
4.9 第一条指令，因为在逻辑上它最先执行。
4.10 1. 都有；2. 都有；3. 软件；4. 硬件；5. 硬件；6. 硬件；7. 都有；8. 硬件；9. 都有。
4.11 前两个错误，后两个正确。