Lecture 8 处理器

1. 引言

CPU 性能

回顾 CPU 性能的公式

$$CPU\ Time = Instruction\ Count \times CPI \times Clock\ Cycle\ Time$$

- **编译器和指令集**决定了**一个程序所需要的指令数目**
- **处理器（CPU）**决定了**时钟周期长度和 CPI**

基本的 MIPS 实现

实现指令

我们要实现的方式包含了 MIPS 指令集的一个核心子集

- 存储器访问指令：lw、sw
- 算术逻辑指令：add、sub、and、or、slt（小于则设置）
- 分支指令：beq、j

这部分指令虽然不包括所有的整数指令、浮点数指令，但是该子集可以说明建立数据通路 data path和控制单元 control 的关键原理

实现方式概述

指令的实现过程大致相同，与具体的指令类型无关，每条指令实现的前两步为

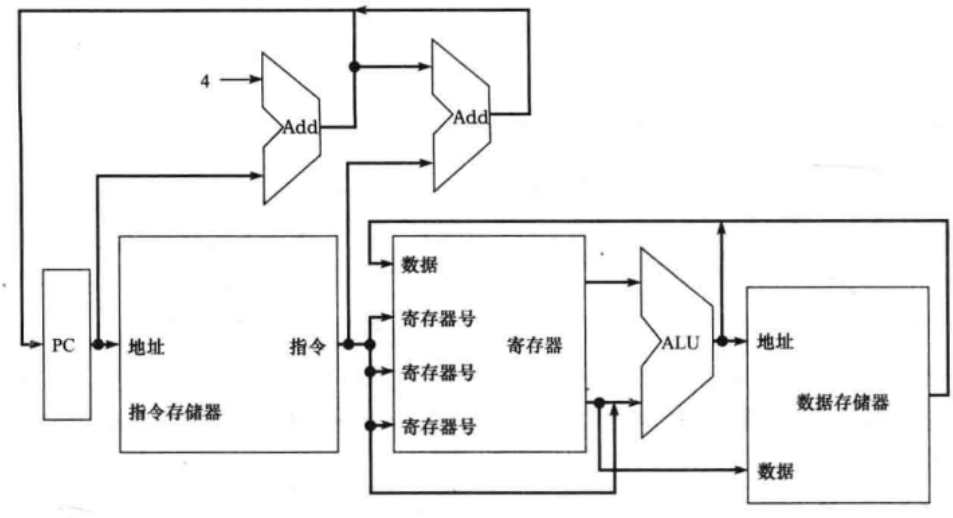
1. 程序计数器 PC 指向指令所在的存储单元，并从中取出指令
2. 通过指令字段内容，选择读取一个/两个寄存器
3. **除了跳转指令**外，所有指令读取寄存器后，都要用算数逻辑单元 ALU 进行计算判断

a. 存储访问指令：用 ALU 计算地址
b. 算数逻辑指令：用 ALU 进行运算
c. 分支指令：用 ALU 进行比较

4. 而后，不同指令的动作不一样了

a. 存储访问指令：访问内存以便读写数据
b. 算数逻辑指令/取数指令：将 ALU 或 寄存器的数据写入寄存器
c. 分支指令：选择下一条指令的地址

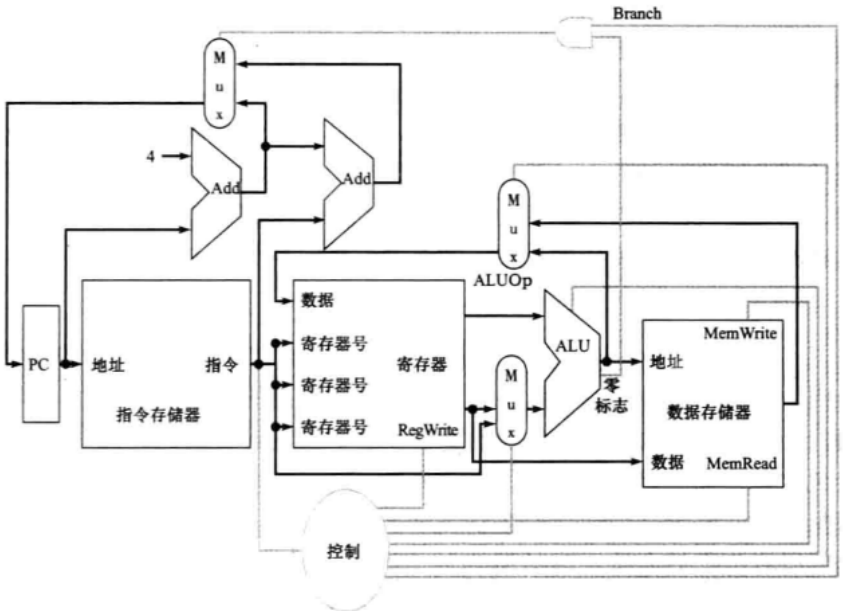
下面是关于是上述步骤的基本概念图：



它还有以下几个不足

- 缺少**多选器 multiplexor** 控制从不同数据来源中选择一个送给目标单元
- 许多单元的控制依赖于当前指令的类型，ALU 根据不同的指令执行不同的操作，这需要**控制信号**判断

所以我们进一步丰富设计成：



这是一个 MIPS 子集的基本实现，其中包含了必要的**多选器**和**控制信号**

- 最上面的多选器**控制写入 PC 的值**（PC+4或分支目的地址）
- 中间输出到寄存器堆的多选器，用来**选择将被写入寄存器堆中的是 ALU 的输出（算术逻辑指令时）或者数据存储器的输出（取数指令时）**
- 最后，最下面的多选器决定 **ALU 的第二个输入是来自寄存器堆（算术逻辑指令或分支指令时）还是指令的偏移量字段（存取指令时）**
- 新增的控制信号**直接控制 ALU 的操作**、数据存储器的读写和寄存器堆的写入等，控制信号在图中用灰色线标识出来

2. 逻辑设计一般方法

MIPS 在实现数据通路 data path 功能部件包括两种不同逻辑单元

二进制编码信息

- 低压= 0，高压= 1
- 一个 bit 对应一个 wire
- 多个 bit 的数据会编码在多个 wire 上

组合单元 combinational element

- 处理数据值
- 输出是输入的函数，它们的输出只取决于当前的输入
- 操作单元，如**门或 ALU**

状态单元 state element

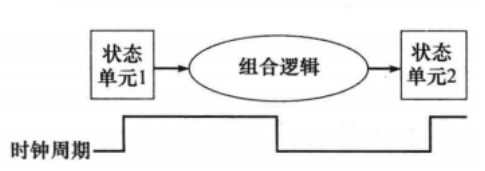
- 它包含状态
- 输出不仅依赖输入，还依赖当前的状态
- 存储单元，如**寄存器或存储器**

一个状态单元至少有两个输入和一个输出

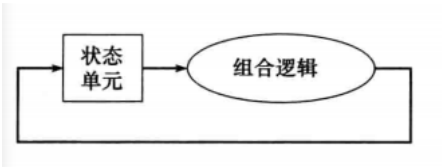
- 输入1：要写入单元的数据值
- 输入2：时钟信号
- 输出：提供在前一个时钟信号写入单元的数据值

时钟

时钟规定了信号可以读出和写入的时间，我们假定采用弄边沿触发时钟 edge-triggered clocking 方法，即在时序逻辑单元中存储的所有值都只允许在时钟跳变的边沿时改变



上图描述了一个组合逻辑单元与其相连的两个状态单元，组合逻辑单元在一个时钟周期完成：所有信号在一个时钟周期内从状态单元 1 经组合逻辑到达状态单元 2，信号到达状态单元 2 所需的时间决定了时钟周期的长度



上图描述了一种边沿触发方法，可以在一个时钟周期内读出一个寄存器的值，然后使用一些组合逻辑，同时将新值写入寄存器，必须保证时钟周期足够长，以使得当有效的时钟边沿到来时输入已经稳定

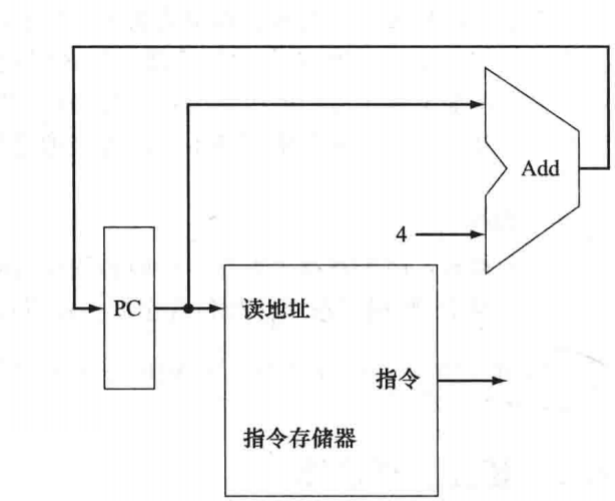
3. 建立数据通路

先来看执行 MIPS 指令需要哪些部件

指令存储器

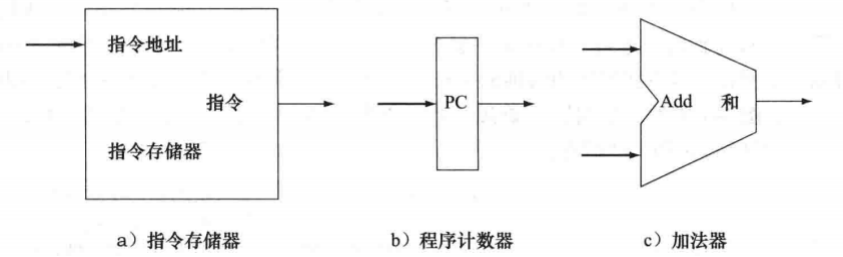
要执行任何一条指令，首先要从存储单元将指令取出

为了准备执行下一条指令，也必须增加**程序计数器指向下一条指令**，即向后移动 4 字节，此时数据通路如图所示



用于取指和程序计数器自增的数据通路部分。取出的指令被数据通路的其他部分使用

使用了下面三个部件



a）指令存储器　　b）程序计数器　　c）加法器

- 组合单元
  加法器：只采用加法的 ALU，将输入的两个 32-bit 数相加，输出结果
- 状态单元
  指令存储器：存放指令的存储器
  程序计数器：32-bit 寄存器，在每个时钟周期末被写入

R 型指令（算数逻辑指令）

这类指令读两个寄存器，对它们的内容进行 ALU 操作，再写回结果

寄存器堆

处理器的 32 个通用寄存器位于一个叫做**寄存器堆 register file** 的结构中，其中的寄存器都可以通过相应的寄存器号来进行读写

寄存器堆也包含了计算机的寄存器状态

R 型指令的处理流程

写入数据字（1个）

- 输入：要写的寄存器编号 register number 和要写的数据 data
- 写操作：由控制信号控制

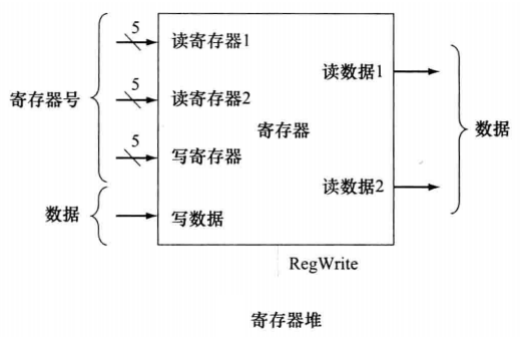
读出数据字（2个）

- 输出：两个数据

R 型指令需要的硬件

为了实现 R 型指令的 ALU 操作，我们需要两个单元：

寄存器堆



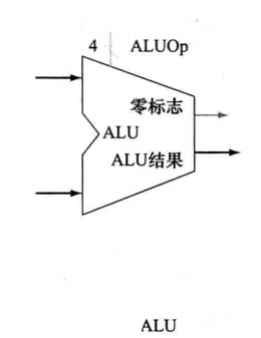
寄存器堆

- 为了读出一个数据字，寄存器堆需要输入一个要读的寄存器号和输出一个从寄存器堆读出的结果
- 为了写入一个数据字，寄存器堆需要提供要写的寄存器号和要写的数据

所以，一共有

- 4 个输入：3 个寄存器号和 1 个数据
- 2 个输出：2 个数据

以及一个 ALU



ALU

- 2 个输入：32-bit 数据
- 1 个输出：32-bit 数据
- 输出指示：指示其结果是否为 0

可能会有疑惑，读出后进行计算才写入，为什么寄存器提供这样的输入和输出呢，其实是因为**寄存器的写入操作都是沿边沿触发的**，故可以在同一时钟周期内读出和写入同一寄存器，读操作将读出以前写入的内容，而写入的内容在下一时钟周期才可读，也就是说可以先读出，在 ALU 内计算好（小于一个时钟周期），再写入

I 型指令

存取指令

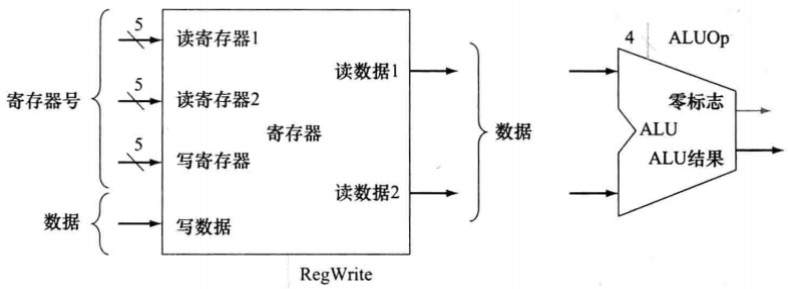
```
1  lw $t1, offset_value($t2)
```

这类指令通过基址寄存器 $t2 的内容与 **16-bit 带符号**的偏移地址相加，得到存储器地址

- 存储指令 lw：从 $t1 中读出需要存储的数据
- 取数指令 sw：从存储器中读出的数据存入指定的寄存器 $t1 中

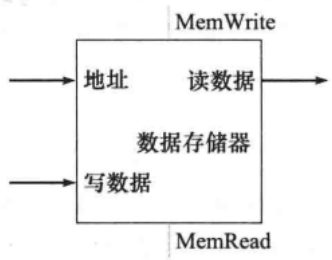
存取指令需要的硬件

所以，存取指令同样需要**寄存器堆和 ALU**

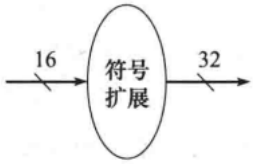


注意，偏移地址是 16-bit 的，与 $t2 相加之前需要先**符号拓展**到 32-bit

所以我们还需要**数据存储器**单元和**符号拓展单元**



a）数据存储器单元



b）符号扩展单元

数据存储单元是一个状态单元，两个输入为地址和所写的数据，一个输出为读出的数据

分支指令

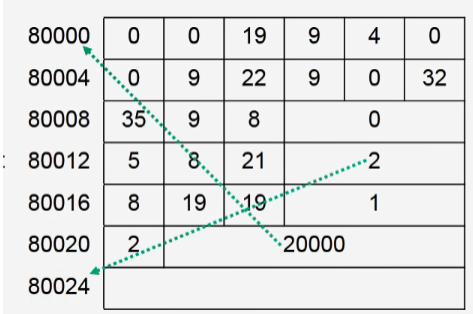
```
1  beq $t1, $t2, offset
```

beq 指令有 3 个操作数，其中两个为寄存器，用于比较是否相等，另一个是 16-bit 的偏移量，用于计算相对于分支指令所在地址的**分支目标地址 branch target address**

为了实现指令

1. 先将 16-bit 偏移量**符号拓展**到 32-bit
2. 系统结构规定偏移量左移动 2-bit 以指示以字 word 为单位的偏移量，这样偏移量的有效范围就扩大了 4 倍，所以为了处理这种情况，我们还需要把偏移量左移 2-bit（扩大 4 倍）
3. 规定指令集计算分支地址时使用基地址，是分支指令的下一条指令的地址，所以我们一般会计算 **PC + 4（下一条指令的地址）**
4. 除了计算分支目标地址，还需要确定是顺序执行下一条 指令，还是去执行分支目标地址的指令，当分支条件为真时，分支目标地址成为新的 PC，若操作数不等，则继续 PC + 4

回顾一下之前章节讲的示例



从80012行开始，此时PC = 80016， address = 2

所以下一条指令的地址为 80016 + 2 × 4 = 80024

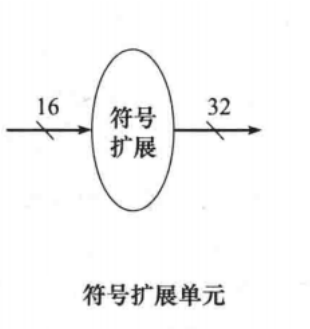
所以，分支数据通路需要执行两个操作

- 计算分支目标地址
- 比较操作数

分支指令需要的硬件

所以分支指令需要：

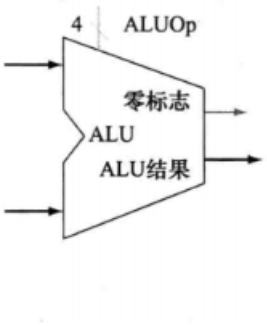
一个符号拓展单元



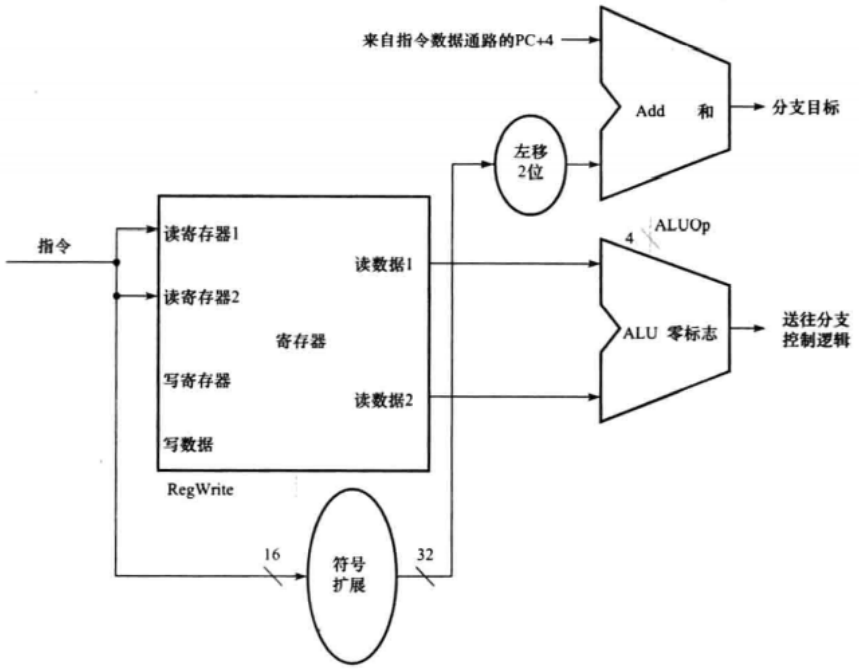
符号扩展单元

扩展 16-bit 偏移量至 32-bit 地址

一个加法器



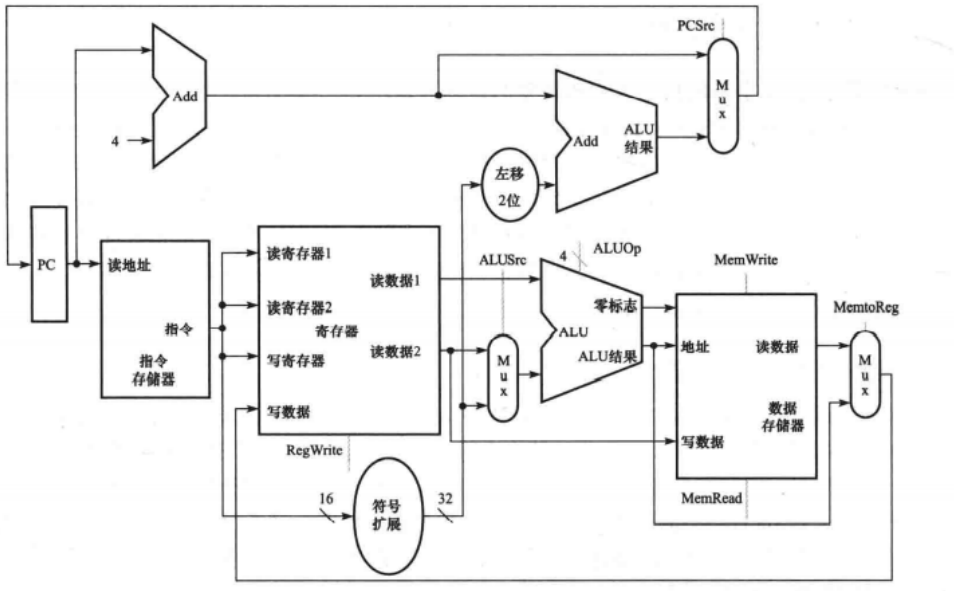
将偏移量右移并且与 PC 相加，得到分支目标地址



最终我们需要用 ALU 计算条件是否成立，以确定是否满足跳转条件

创建数据通路

下面是一个可以实现 R 型指令、存取指令和分支指令的概念图



它们合并在一起，构建了一个简单的 MIPS 体系结构的数据通路，这个数据通路可以在一个时钟周期内完成基本的指令

4. 一个简单的实现机制

来构建一个简单的 MIPS 体系结构，实现了lw、sw、beq、add、sub、AND、OR 和小于则置位

单周期实现

单周期 single-cycle implementation，指在一个时钟周期内执行一条指令的实现机制，在现实中，由于它太慢而不实用

ALU 控制单元的设计

MIPS ALU 在 4-bit 控制信号中定义了 6 中有效的输入组合

ALU 控制信号	功能	ALU 控制信号	功能
0000	与	0110	减
0001	或	0111	小于则置位
0010	加	1100	或非

除了或非指令在目前实现的 MIPS 子集中暂时没有外，其它的都可以实现

生成 ALU 控制信号

指令操作码	ALUOp	指令操作	funct 字段	ALU 动作	ALU 控制信号
取字	00	取字	XXXXXX	加	0010
存字	00	存储字	XXXXXX	加	0010
相等则分支	01	相等分支	XXXXXX	减	0110
R 类型	10	加	100000	加	0010
R 类型	10	减	100010	减	0110
R 类型	10	与	100100	与	0000
R 类型	10	或	100101	或	0001
R 类型	10	小于则置位	101010	小于则置位	0111

输入

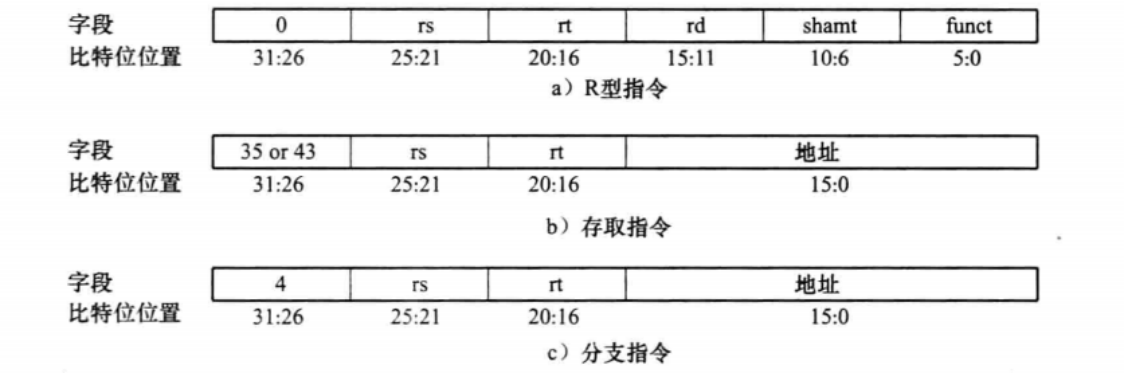
- 2-bit ALUOp 字段
    - 00：加法
    - 01：减法
    - 10：由 funct 字段决定
- funct 字段

输出

- ALU 控制信号

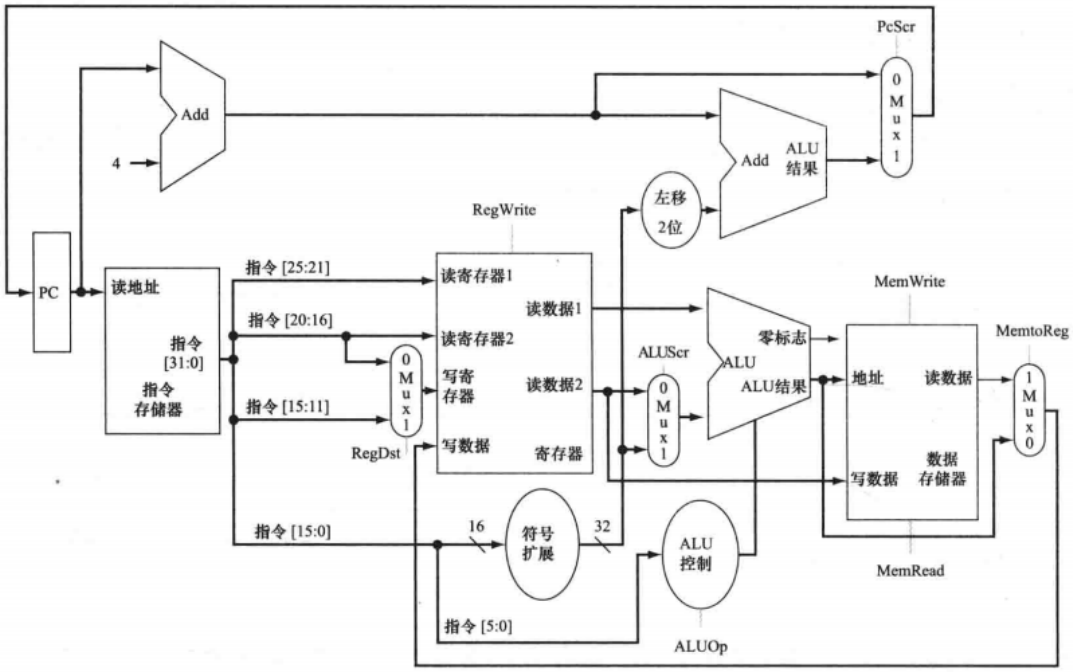
主控制单元的设计

回顾之前 3 种指令的划分



- **OP字段（操作码 opcode）**：总是 [31:26] 位，我们用 OP[5:0]表示
- 对于 R 型指令、分支指令和存取指令，要读取**两个寄存器为 rs 和 rt 字段**，分别为 [25:21] 位和 [20:16] 位
- 存取指令的基址寄存器在 [25:21] 位的 rs 中
- 相等则分支指令、取数指令的 16-bit 偏移量在 [15:0] 位中
- 存放目标寄存器的位置，对于取数指令是 [20:16] 位的 rt中，对于 R 型指令为 [15:11] rd 中，所以我们需要一个**多选器**，来指示要写的寄存器号在哪个字段中

那么设计如下所示

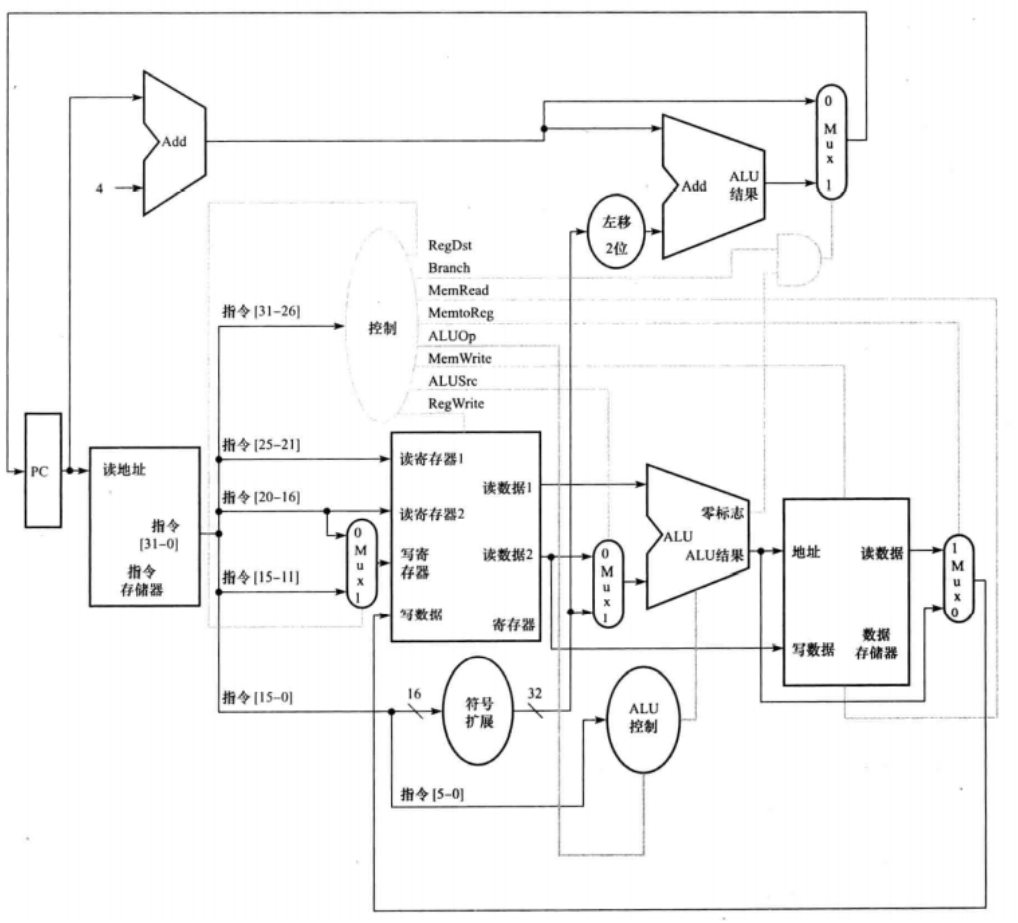


它有 7 个控制信号

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

除了 PCSrc 控制信号以外，所有的控制信号都可以由控制单元根据指令的操作码来确定，而 PCSrc 信号有效的条件是指令为相等则分支

现在，这 **9-bit 控制信号（上面的 7-bit + 2-bit ALUOp）**状态可根据控制单元的 6-bit 输入信号 [31:26] 来设置，我们细化如下图所示：



指令	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R 型	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

控制单元 OPcode 的实现

输入或输出	信号名	R 型	lw	sw	beq
输入	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0

输入或输出	信号名	R 型	lw	sw	beq
输出	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

J 指令扩展

跳转指令的高 4-bit 来自于跳转指令的 PC + 4，低 28-bit 来自于指令的 26-bit 立即数左移 2-bit，下图增加了对跳转指令的支持，此时多选器增加了一个控制信号 Jump，只有当操作码 opcode 为 00010，该控制信号才有效