MIPS 基础

本文介绍 MIPS 架构的相关知识。首先讨论一些基础背景知识，然后介绍指令，最后展示用于实现这种 ISA 的一种可行的数据通路设计。

简介

MIPS 全称 **M**icroprocessor without **I**nterlocked **P**ipeline **S**tage。

最早的 MIPS 架构为 32 位，最新的已经是 64 位。

现在龙芯用的就是 MIPS 架构。

存储设计

MIPS 采用的是按字节编址，按字访问的方式。使用的字长为 32 位，访问时以字为单位存/取。字有**对齐限制（Alignment Restriction）**，所有字的首地址必须为 4 的倍数。访问字时访问的是字的首地址，即字所包含的地址中最小的那个。

MIPS 采用的是大端模式。不过这个貌似不是很重要。

MIPS 有一片栈空间专门用来保存函数调用时调用者（Caller）的相关信息。

设计原则

记住一些设计原则有利于指令的记忆和理解。

1. 简单即规整（Simplicity favors regularity）。即指令比较简单，同时同一类型的指令格式统一。
2. 少就是快（Smaller is faster）。即要平衡一些设计，加的东西太多未必好。
3. 加快经常性事件（Make the common case fast）。例如寄存器溢出，以及使用零寄存器等。
4. 良好的设计就是合理的折衷（Good design demands good compromises）。例如 I 型指令，牺牲了立即数的取值范围，保证了指令长度的统一。

汇编基础规则

1. 一行一条指令。
2. 可以有注释，用 # 表示注释的开始，但是注释只能放在一行的最后面。
3. 寄存器号前加一个 $ 符号。

操作数

可以是通用寄存器，也可以是常数（一般会被称为立即数）。

通用寄存器

寄存器（Register） 是位于 CPU 内，距离运算器最近、具有最快访问速度的存储器。

> 似乎直接来自内存也是可以的，但是 RISC 下为了维护简单性没有这么做。x86 允许操作数直接来源于内存。

通用寄存器总共有 32 个，每一个都具有一定的功能。下面列举一下。

> 为什么说是通用寄存器？因为它们是对程序员可见的。还有一些寄存器并不对程序员可见，如 PC。在谈论操作数的时候，一般说的寄存器就是指通用寄存器。

- 零寄存器，记作 zero。特点是始终是 0，即使对其进行写入操作也保持是 0，原因在于其是**硬接线的（Hard-wired）**。也就是硬件层面上保证了其恒定为 0。
- at。其由汇编器控制，用于处理较大的常数。
- v0 和 v1。用于存储子程序返回值的寄存器。
- a0 到 a3。用于存储子程序要用到的参数。可以直接保存参数本身，或者保存参数在内存中的地址。
- t0 到 t9。没什么特定功能的寄存器。
- s0 到 s7。没什么特定功能的寄存器，但是会在子程序调用时被保存在栈中。
- k0 和 k1。操作系统专用。
- gp。用于编译器优化。
- sp。用作栈结构的栈顶指针。
- fp。（不知道干啥用的）
- ra。用于保存子程序执行完之后应当让 PC 返回到的地址。

既然寄存器这么快，为什么不多整几个？只保留 32 个？原因有几个方面。

- 大量的寄存器可能导致电信号的行进距离增长，导致时钟周期变长、时钟频率降低。
- 增加寄存器可能导致功耗上升。
- 受到指令格式的限制。如果寄存器变多就需要在指令中给寄存器编号分配更多的位数，这在 RISC 指令定长的限制下是相当难办的事情。

> 在事实层面，寄存器的数目增长也是非常缓慢的。因为寄存器的数目和指令集架构有关，指令集架构的发展存在惰性，因而寄存器数目的增长和指令集架构的发展近乎一样缓慢。

总之，平衡好程序员的工作难度和硬件的运行速度很重要。

立即数

立即数也就是常数，可以看作是被硬编码在了指令中的操作数。

使用立即数的好处在于其可以简化很多指令的设计。例如 move 指令就是一条其中一个操作数是 0 的加法指令。

立即数虽然好用，但经常受到定长指令的限制，导致其的取值范围往往达不到 32 位。

指令种类

由于 MIPS 属于 RISC 架构，因此其的指令长度全部固定为 32 位。

MIPS 指令按照指令格式可以分成三大类：R 型指令、I 型指令和 J 型指令。

先列出我们下面将会讨论到的指令：

```
add rd, rs, rt    # rd = rs + rt
sub rd, rs, rt    # rd = rs - rt
and rd, rs, rt    # rd = rs & rt
or rd, rs, rt     # rd = rs | rt
xor rd, rs, rt    # rd = rs ^ rt
sll rd, rt, sa    # rd = rt << sa
srl rd, rt, sa    # rd = rt >> sa (logically)
sra rd, rt, sa    # rd = rt >> sa (arithmetically)
jr rs             # PC = rs

addi rs, rt, imm  # rt = rs + (sign)imm
```

```
andi rs, rt, imm  # rt = rs & (zero)imm
ori rs, rt, imm   # rt = rs | (zero)imm
xori rs, rt, imm  # rt = rs ^ (zero)imm
lw rt, imm(rs)    # rt = *(rs + (sign)imm)
sw rt, imm(rs)    # *(rs + (sign)imm) = rt
beq rs, rt, imm   # if rs == rt then PC = PC + 4 + ((sign)imm << 2)
bne rs, rt, imm   # if rs != rt then PC = PC + 4 + ((sign)imm << 2)
lui rt, imm       # rt = imm << 16

j addr            # PC = {(PC + 4)[31: 28], addr << 2}
jal addr          # $31 = PC + 4, PC = {(PC + 4)[31: 28], addr << 2}
```

一般写汇编的时候分支和跳转都可以直接写标签名称，而不用写具体的 offset 或者地址。如：

```
main: addi $1, $0, 144
      addi $2, $0, 128
loop: lw   $5, 0($1)         # input in_port to $5
      sll  $6, $5, 16        # left shift by 16 bits
      ori  $7, $6, 21845     # or it with 0x00005555
      sw   $7, 0($2)         # output in_port to out_port
      j loop
```

写标签时需要注意冒号后应当至少有一个空格。

> 当然，标签的本质就是地址。

R 型指令

R 型指令和寄存器相关（取 Register 头字母）。其具有以下构成：

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- op 表示**操作码（Opcode）**，用于识别指令种类。R 型指令的 op 是全 0。I 型、J 型也是开头 6 位用作操作码。
- rs，rt 表示两个操作数。s 指 source，t 指 temporary。
- rd 表示目的操作数，即运算结果的保存位置。

> 上面三个可以记忆为 std。需要注意的是，汇编代码中的寄存器顺序和这里的寄存器顺序并不一致。例如 `add $1, $2, $3` 表示的是 `$1 = $2 + $3`，而 rs，rt 分别对应 2，3 号寄存器。

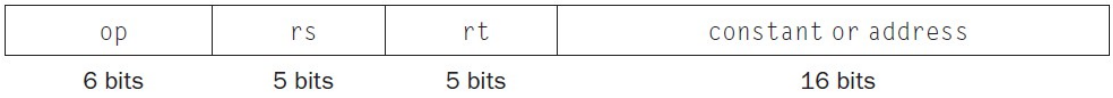
- shamt 表示偏移量（Shift Amount）。
- funct 表示指令功能。只有 R 型指令有这个位域，因为其操作码是全 0，只能靠 funct 指定功能。

R 型指令大部分都和算术运算相关，如 `add`、`sub` 等等。最特殊的一个是 `jr`，即跳转到某个寄存器保存的地址上。按照模式可以细分成下面几类：

- rs，rt 和 rd 都用上的类型，如 `add`、`sub`。
- 只用了 rt 和 rd，以及移位量，如 `sll`，`srl`。
- 只用了 rs，如 `jr`。

I 型指令

I 型指令和立即数相关（取 Immediate 头字母）。其具有以下构成：



- op 还是操作码，不是全 0，表示指令功能。
- rs、立即数是两个操作数，rt 是目标操作数。
- 后 16 位表示一个常数，这个常数对于不同的指令而言有着不同的意义。

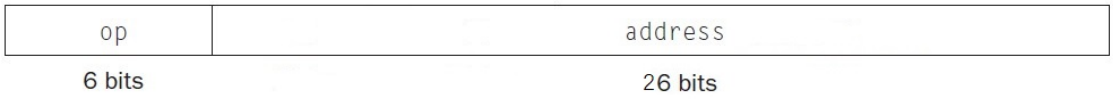
I 型指令的构成就比较丰富。既有算术指令的扩展版本（即一个运算数变成常数），也有存储器的读写指令（即 `lw`、`sw`），还有分支执行指令（即 `beq`、`bne`）。它们的功能差异较大。

按照模式可以分成下面几类：

- 用了 rs 和 rt 的。
  - 写回寄存器堆的都写入 rt 中。如 `addi`，`andi`，`lw`。
  - rt 作为写入数据来源的。如 `sw`。
- 只用了 rt 的，如 `lui`。其将立即数移到高位后写入 rt。

J 型指令

J 型指令和跳转相关（取 Jump 头字母）。其具有以下构成：



- op 还是操作码，不是全 0，表示指令功能。
- address 表示跳转位置。由于其只有 26 位，因此真实的跳转位置为 PC + 4 的高 4 位，再连上 address 左移两位后的结果。

J 型指令只包含两条指令：`j` 和 `jal`。`j` 是无条件跳转，方便构造出死循环，在波形仿真中经常使用；而 `jal` 表示跳转并链接，将 PC + 4 先保存在 31 号寄存器上（即 ra 寄存器）再进行跳转。它一般用在子程序调用中。

> 值得注意的是，`jal` 保存的地址可能是 PC + 8。这和*延迟槽*有关。会在 Lecture Notes 4 中控制冒险部分说明为什么这么做。在单周期中，我们还是使用 PC + 4。

NOP

很多 ISA，包括 MIPS，提供了一个空指令 `NOP`：它不做任何事。

指令功能

此前提到过，指令可以进行多种操作，但操作一般都可以分成 4 类：

- CPU 和存储器。
- CPU 和 I/O。
- 数据处理。
- 控制。

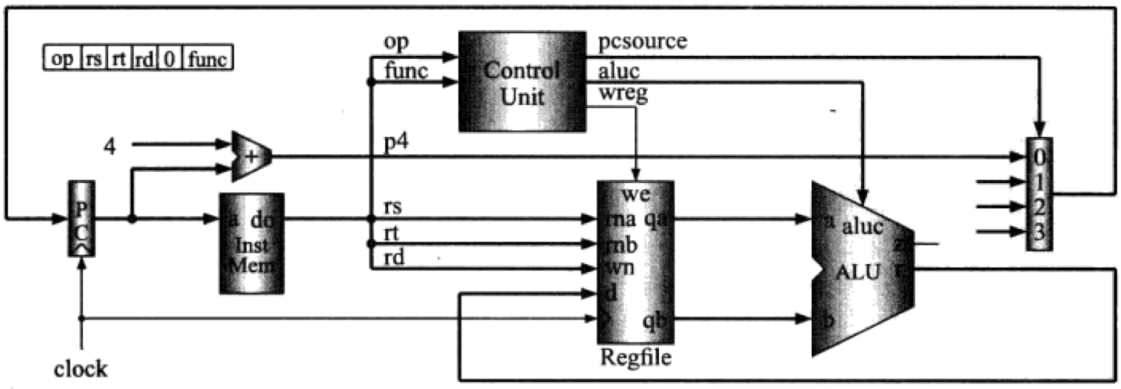
这里我们假设 I/O 采用存储器映射编址的方法（即在抽象层面把 I/O 端口地址视作存储器地址空间的一部分），那么前两类就合成了一类。

下面分别讨论这几类指令，并给出它们的数据通路。

数据处理

数据处理包含各种算术、逻辑运算，如 `add`、`sub`、`and`、`or` 等。它们的执行过程基本相同：通过操作码或者 funct 获取具体的 `aluc`，用于控制 ALU 执行相应的运算，然后将结果写入目标寄存器。

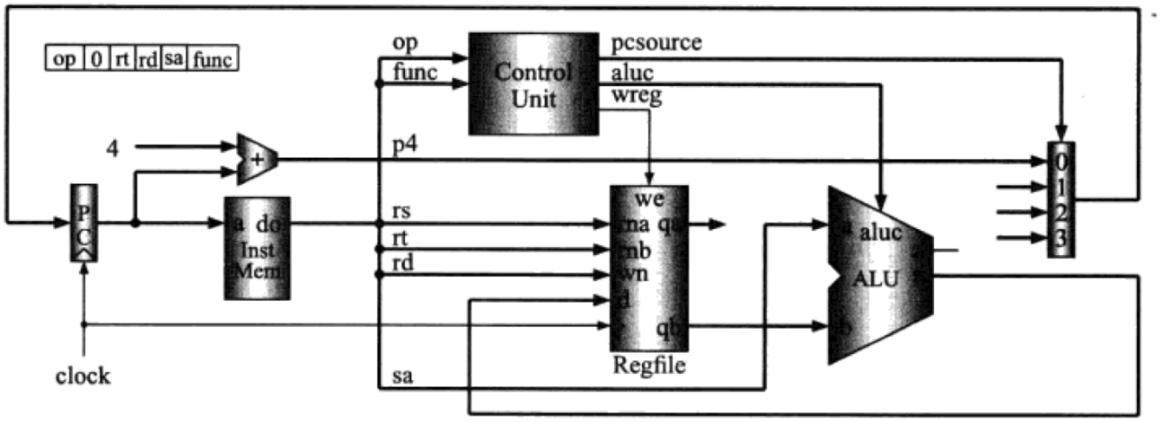
对于 R 型指令，一种数据通路如图所示。



其中 rs，rt 传入寄存器堆的 na，nb 两个口，对应的数据从 qa，qb 读出。

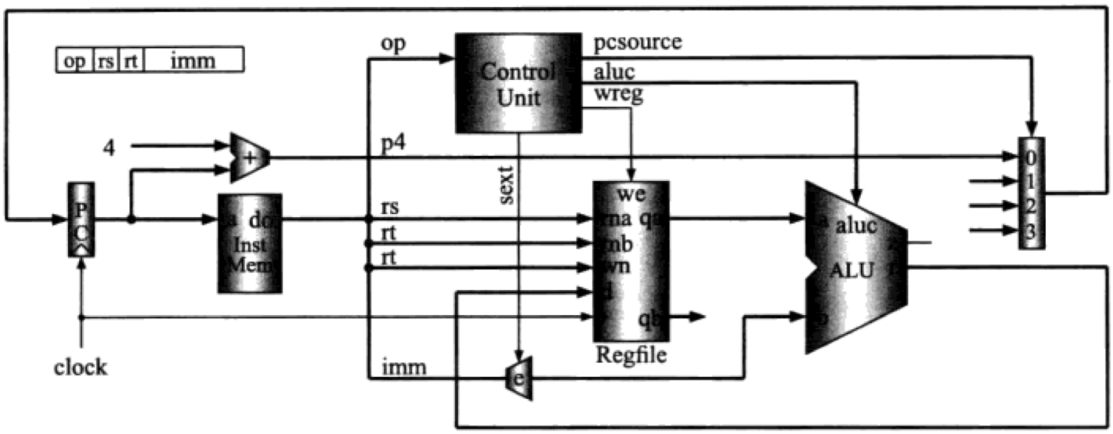
> na，nb 有时候也用 n1，n2 表示。本质没有不同。

另一种数据通路会在移位时用到。



此时不是由 qa 提供第一个操作数，而是由偏移量作为第一个操作数。这和移位时 rs 域为 0 相对应。

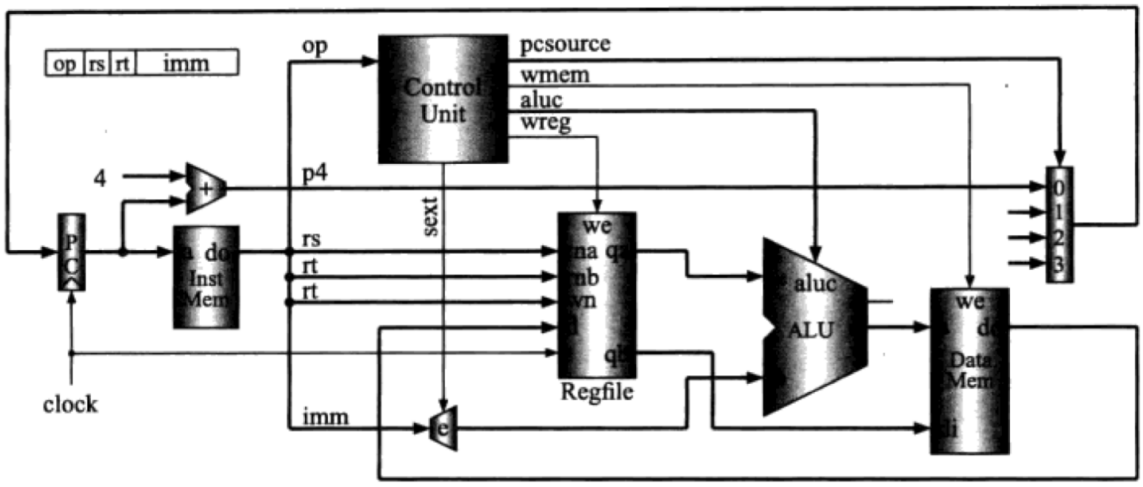
对于 I 型指令，数据通路如图所示。



其中一个（或者 0 个）操作数由寄存器堆提供，另一个操作数是可能经过扩展的立即数。运算结果写回寄存器堆。

CPU 与存储器

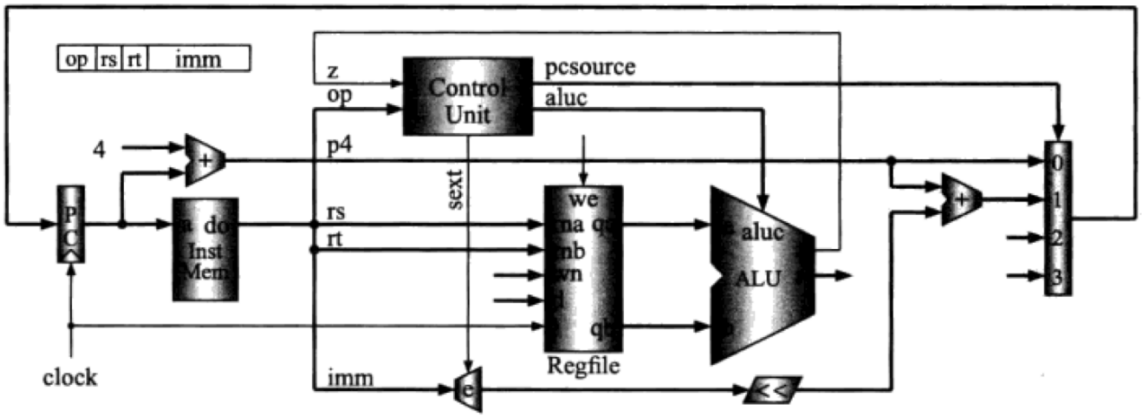
MIPS 指令中只有 `lw` 和 `sw` 会读写存储器。数据通路如下。



可以看到主要的区别就是增加了一个数据存储器。

控制：分支

分支指令包括 `beq` 和 `bne`。数据通路：



可以看出与上面的图相比，ALU 的 zero 信号启用了。当运算结果为 0 时该信号有效。利用该信号可判断是否分支。因此，可以将 `beq` 和 `bne` 对应的比较运算设为减法或者异或，并体现在 `aluc` 上。

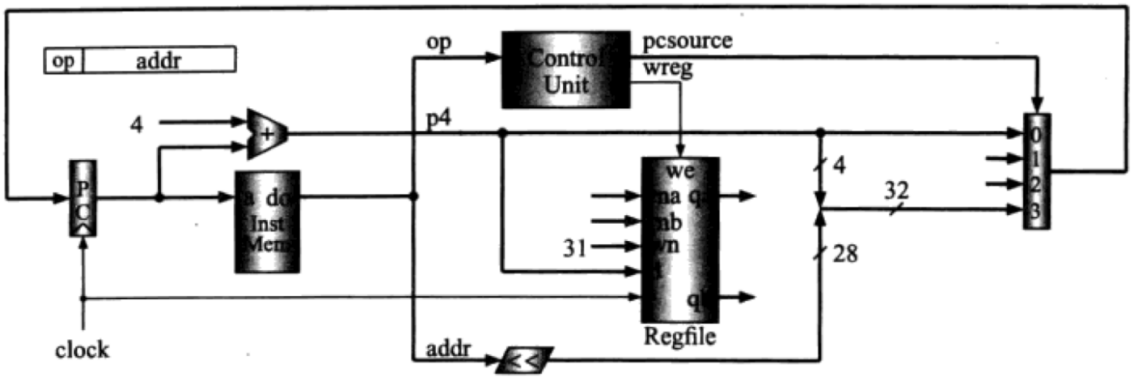
> 基本块（Basic Block）是除了末尾，别处不可能有分支，且除了开头，别处不可能有分支目标（标签）的指令序列。编译最初阶段的任务之一就是将程序分解为若干个基本块。

使用分支指令可以实现分支、循环等结构。还可以实现 `switch` 语句。

> 实现 `switch` 语句的一种方法是将其转化为一系列 `if-else`。另一种是将多个指令序列分支的地址编码为一张表，即**转移地址表（Jump Address Table）**或**转移表（Jump Table）**。它是一个由代码中标签对应地址构成的数组。程序需要跳转时可以索引该表，将表中适当的地址加载到寄存器中，然后利用 `jr` 跳转。

控制：跳转

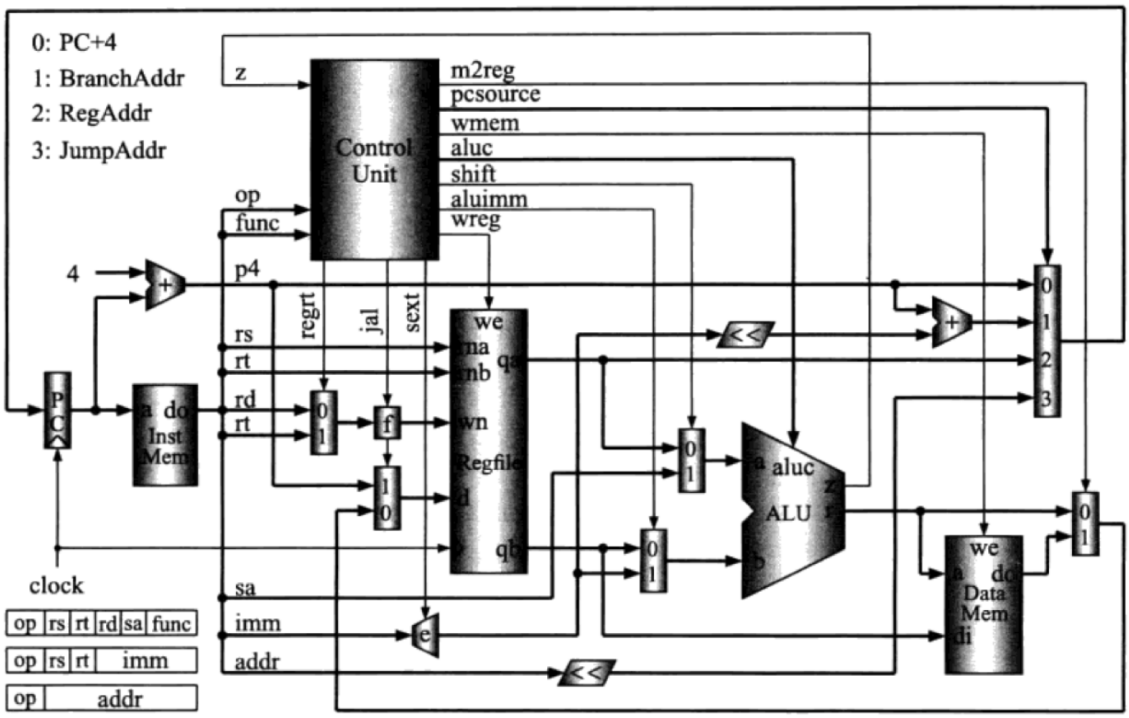
跳转指令包括 `j`，`jr` 和 `jal`。数据通路仅以 `jal` 为例：



利用这几条指令可实现**子过程（Procedure）**。更多的和子过程相关的信息参见另一篇文章。

控制信号

将上述指令的对应的数据通路加以整合，就得到了下面的整体结构。



> 这里是单周期 CPU 的结构，流水线的结构以及控制信号会稍有不同。具体见 Lecture Notes 4。

> 图中将指令存储器和数据存储器分开了。这可以通过现代的 CPU 一般都有分开的指令 cache 和数据 cache 来理解，即两个存储器实际上都是 cache。

在译码时需要产生正确的控制信号。将所需要用到的 10 个控制信号列出如下。对于单 bit 信号均为 1 有效。

- `pcsource`：决定下一条指令的地址。下一条指令地址有 4 个来源，每一来源对应一个 `pcsource` 的值：
  - PC + 4，记为 0。
  - 分支地址，记为 1。
  - `jr` 用的地址，记为 2。
  - `j`、`jal` 用的地址，记为 3。
- `aluc`：决定 ALU 运算类型。
- `shift`：决定 ALU 的第一个运算数是否为 sa。
- `aluimm`：决定 ALU 的第二个运算数是否为立即数。

- `sext`：是否对立即数进行符号扩展。
- `wmem`：是否写存储器。
- `wreg`：是否写寄存器堆。
- `m2reg`：是否用存储器读出的结果写寄存器堆。
- `regrt`：运算结果是否写入 rt 而非 rd。
- `jal`：跳转时是否要保存返回位置。有效时会在写寄存器时将写目标改为 31 号寄存器（认为是写到 rd）。

更多的指令

没有提到的一些指令：

- 乘除法。
- 比大小。如 `slt`、`slti`、`sltu`、`sltiu`，其接受两个操作数，如果前者小于后者则将目标寄存器置为 1，否则置 0。比较可以是无符号的或者有符号的，通过 `u` 指定。
- 读写字节。如 `lb`、`lbu`，可以读取一个字节；`lh`、`lhu`，可以读取半个字。`sb`、`sh` 类似。

总结

了解 MIPS 指令的构成对于后续扩展 MIPS 指令十分有帮助。

致谢

本文的主要内容参考自：

- 上海交通大学《计算机组成》（课程代号：EI332）一课的课程材料。
- 《Computer Organization and Design: Hardware/Software Interface (5th edition)》
- 《计算机原理与设计：Verilog HDL 版》

本文的图片都来自《计算机原理与设计：Verilog HDL 版》。