

复杂性控制

我们设计的 CPU 所支持的指令有了大幅增长，对于部分模块也需要增加对应的功能。但好消息是，如果大家在 P5 已经保持了良好的工程设计风格，P6 的复杂性控制也不会带来太大挑战。

相信大家已经有了课上加指令的经验，但我们这次所做的指令添加工作有着较大的不同。我们可以针对某一条指令做我们任何想做的调整，但是面对数十条指令，我们不可能去针对每一条指令再去做特殊的改变。所以，面对增加的指令，我们需要有条理地将他们加入，对复杂性进行控制。

对复杂性的控制主要是凭借**规范和抽象**的力量。规范在之前工程设计风格中谈及了很多，简单来讲目的就是提高代码的可读性，同时保持代码的可扩展性，可以轻松地读懂代码，并且在增加模块功能的同时不能影响之前的设计。

而抽象简单地说就是提取共性，忽略特性、提取主要特征，而屏蔽次要信息。例如我们新增加的乘除模块，在高层模块的排布中就屏蔽了实现的细节，只保留了此模块的端口，以及端口的功能。

例如指令 `lb`, `lh` 都与指令 `lw` 有类似的功能，我们可以利用他们之间的相似性去一次性增加这一类指令，来减少我们的工作量，降低模块的复杂度及调试难度。

高内聚低耦合

这是大家理论课学过的设计模块需要遵守的原则。在这里我们最重要的是尽可能简化**每一个**模块的复杂度，尽量使他们彼此独立。我们使用独立译码器的目的就是将指令分析和模块的功能分离，让模块**只受**译码器给出的信号的控制。



译码器

大家可以先回顾 P5 工程化方法教程中关于控制器解码的教程部分及讨论区相关内容。

如果采用了**控制信号驱动型**的译码器，对于每一个控制信号赋值时不可能将所有指令全部进行枚举。可以先对指令进行简单的分类，比如先行判断指令是分支指令、进行移位操作的指令等，用一个信号去表示是否为某一类指令。如此在之后的控制信号的赋值时就可以先从大类进行出发，一步步进行细分。也就是先对指令进行抽象，从大的层次入手，完成赋值。

而如果采用**指令驱动型**的译码器，也可以进行抽象，例如完成计算功能的指令对跳转都有着相似的需求，可以统一进行赋值，对于重复度过高的赋值部分，为避免出现 Bug 可以使用宏定义。但是如果对指令进行抽象也许会让赋值变得更加繁琐难以检查，那这个时候的抽象就是无用之举了。

模块的功能

在 P6，ALU 等模块的功能将会极大程度地增加，体现在选择信号上则是有更多的取值。选择信号一般是在**译码器**和**功能模块**中使用到，这个时候无论用 `parameter` 还是 `define`，替代判断选择信号的常量，都可以让我们的代码可读性更高。同时也应选好适合自己的命名方式，通过命名可以读出是在哪一级、什么模块、有何功能的信号。

冲突和冒险

冲突冒险是过程中相对比较复杂的问题。我们还是回到抽象的力量中去，依旧利用指令之间的相似性，将在 P5 中对一个指令的处理扩展到一类指令的处理，可以大幅地降低复杂度。对于较好的冒险处理方案，在添加 P6 的新指令时对冒险处理模块只需做一定的微调甚至可以不做任何改动。

思考题

为了对抗复杂性你采取了哪些抽象和规范手段？这些手段在译码和处理数据冲突的时候有什么样的特点与帮助？