**实验19**

**简单单周期处理器**

# **实验介绍**

现在，我们终于可以开始搭建自己的第一个 CPU 了。当然，这只是一个十分基础的单周期 CPU，但这将是「图灵完备」在 COD 课程中的直接体现。

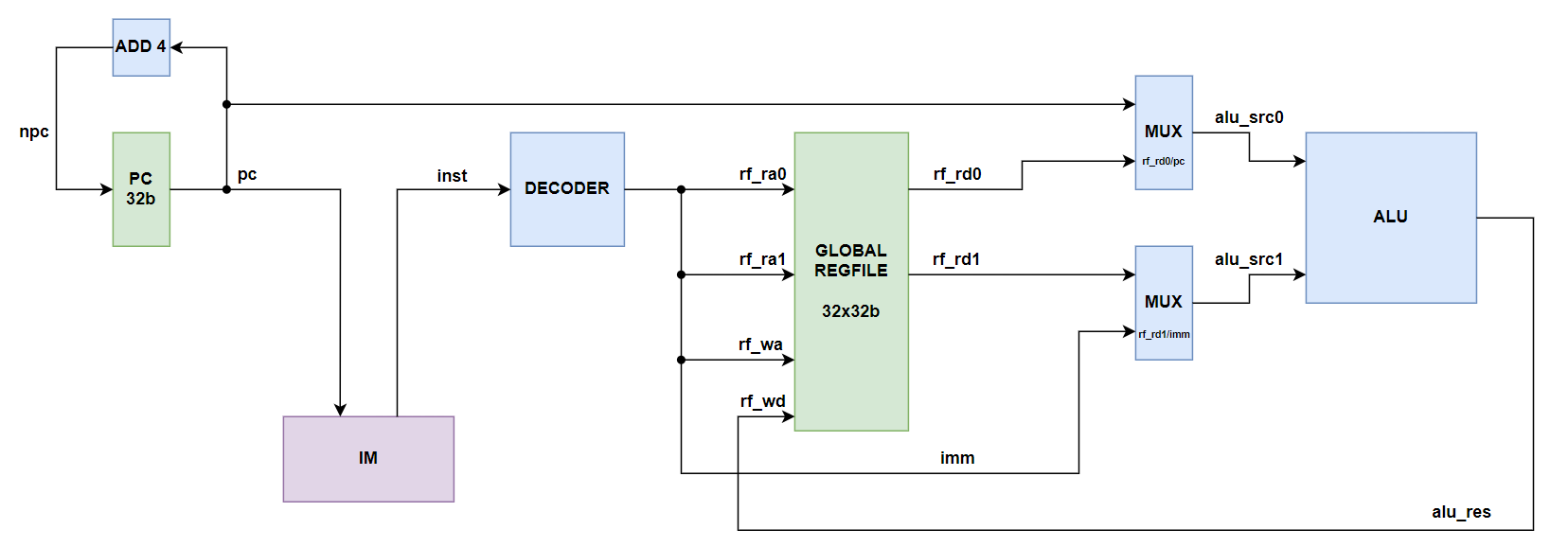
单周期 CPU（Single Cycle Processor）是指一条指令在一个时钟周期内完成，并在下一个时钟周期开始下一条指令的执行的 CPU。单周期 CPU 由时钟的上升沿或下降沿控制相关操作，两个相邻的上升沿或下降沿之间的时间间隔就是 CPU 的时钟周期。

需要注意的是，由于没有额外的暂存寄存器，单周期通路中的关键路径对应的延迟很高，以致于上板时的单条指令运行时长大于 10 纳秒。所以，我们不能直接使用开发板上的 100MHz 时钟作为 CPU 的运行时钟。除此之外，上板时我们也无法了解当前 CPU 运行到哪条指令，以及相应的结果如何。为此，我们不得不请出 PDU 来帮助我们完成这些工作。

本次实验我们将实现一个最为基础的单周期 CPU。这个 CPU 没有分支功能，也没有数据访存功能，只能进行基本的数据运算。但我们仍将深刻领悟到 CPU、PDU 以及 MEM 之间是如何相协同工作的。

# **实验内容**

## 1 整体框架

[](https://soc.ustc.edu.cn/COD/lab3/img/cpu/CPU.png)

上图展示了本次实验中 CPU 的数据通路。该数据通路可以兼容 RV32I 版本的 CPU 和 LA32R 版本的 CPU，二者仅在译码器 Decoder 的内部结构上不同。此外，通路中省略了控制信号，例如 alu\_op、rf\_we 等。

## 2 模块设计

### ****2.1 PC寄存器****

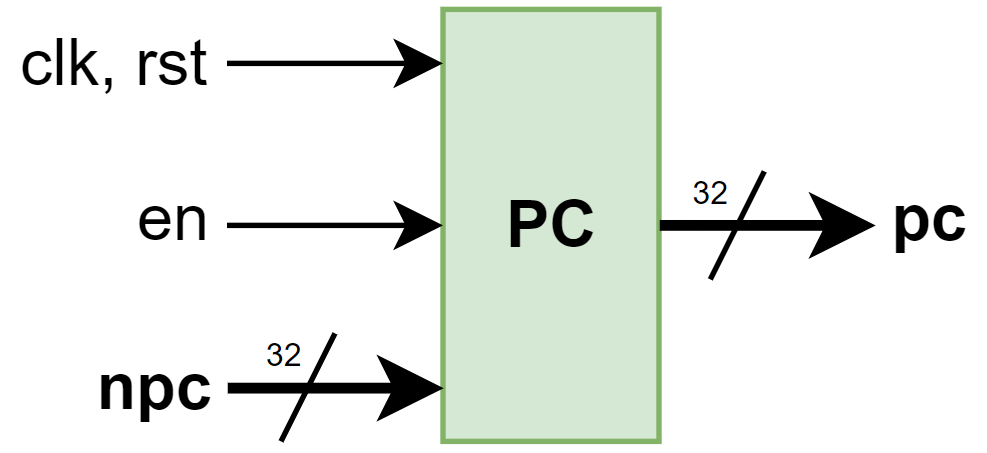
寄存器是我们最先接触的时序元件。一般而言，我们可以用如下的 Verilog 程序描述寄存器的行为：



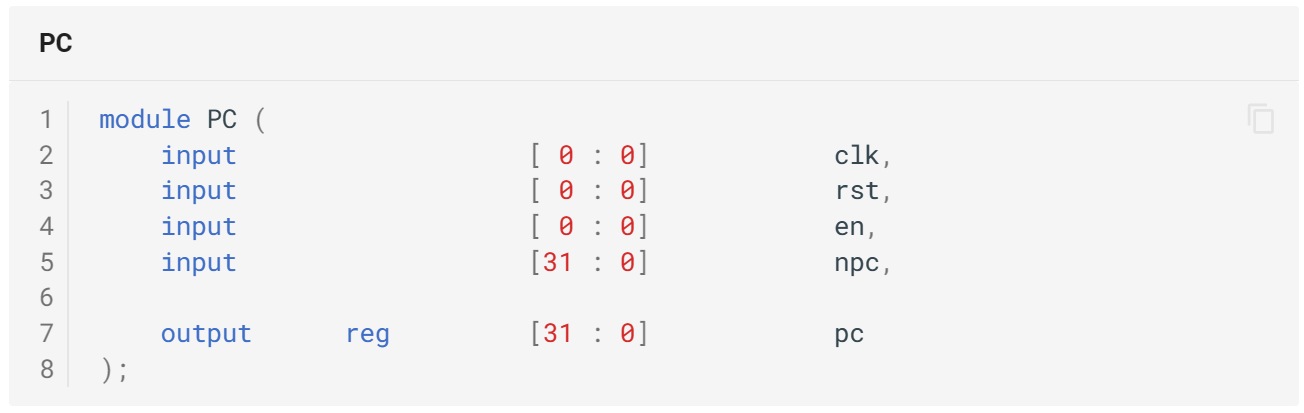
上面的代码给出了带有同步复位信号的寄存器单元。其中，q 为当前寄存器中保存的数据，d 为即将写入寄存器的数据。

在 CPU 中，PC 寄存器时刻存储了正在执行的指令的地址。它的功能是将当前指令的地址传递给指令存储器，从而读出此时正确的指令内容。同时，它也需要能够接受下一条指令地址的输入，并在时钟上升沿到来时更新自己的值，从而实现了指令的连续运行。

PC 寄存器的模块示意如下所示：

[](https://soc.ustc.edu.cn/COD/lab3/img/cpu/pc_struc.png)

其端口代码为：

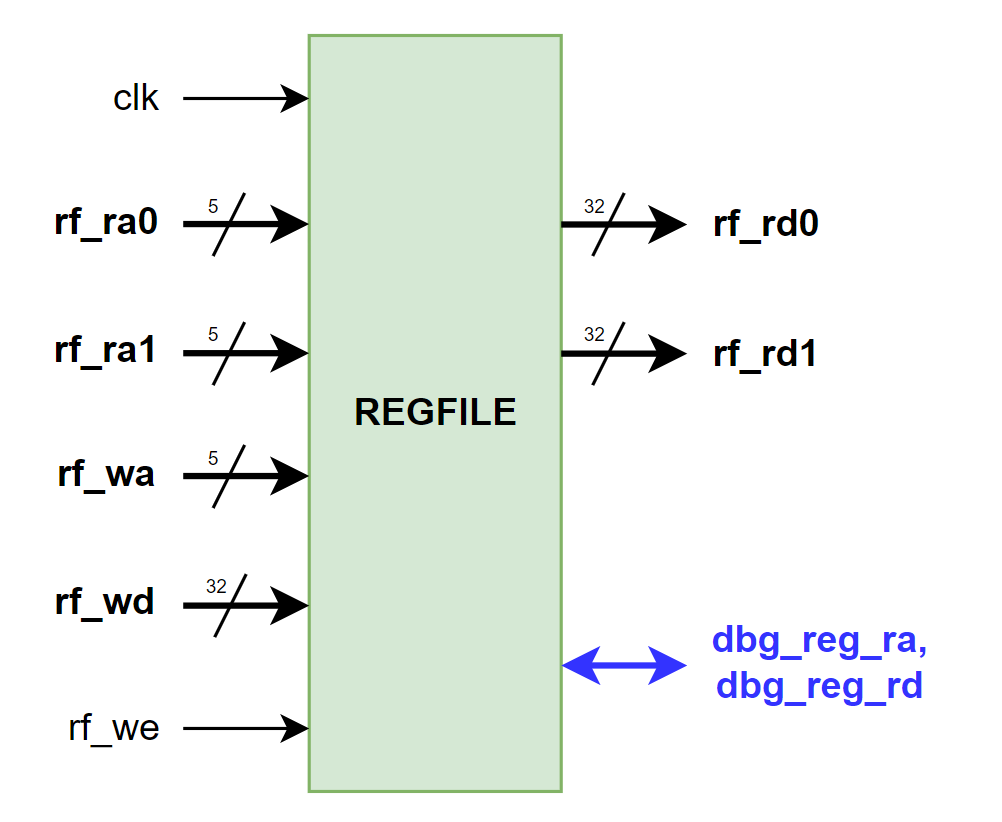


值得一提的是，从代码上看，PC 寄存器的行为可以被一个 always 块完全实现。那么为什么我们还是将其封装成了一个模块呢？

* 一方面，高度模块化的设计可以使我们的电路结构更加清晰，减少杂乱的门电路的出现（看不见就是没有.jpg）。
* 另一方面，我们可以更为直观地看到各个模块之间数据的传输关系，便于和提供的单周期数据通路进行比对。
* 此外，为了代码的可读性，我们一般不会在一个模块中保留过多的代码。

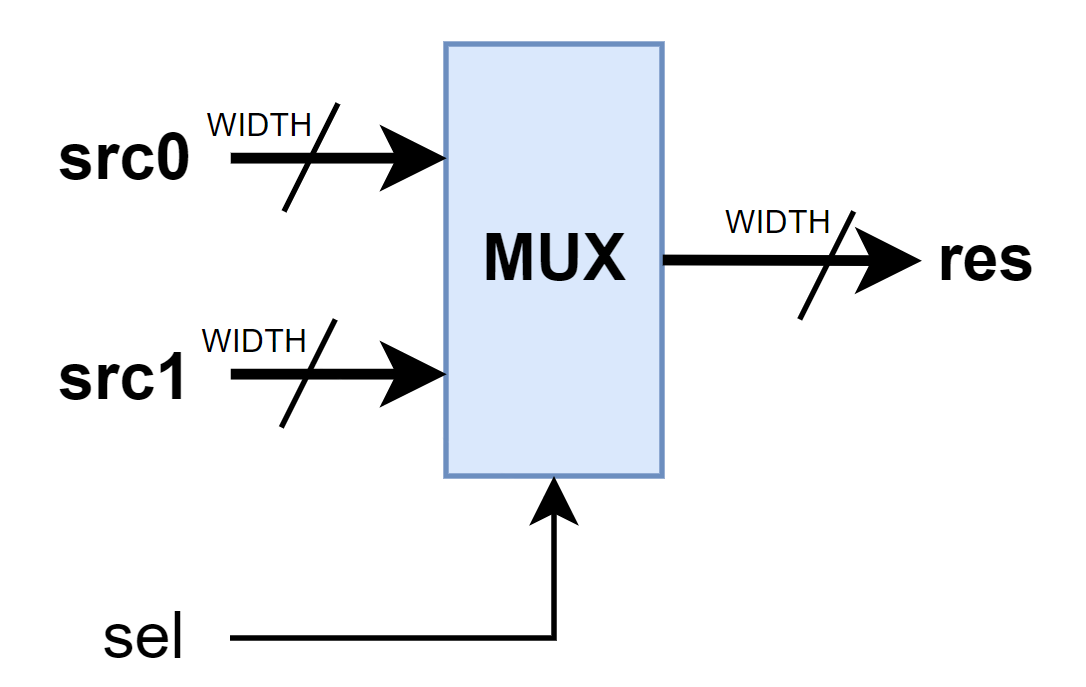
综上所述，对于部分基础模块的封装是很有必要的。

### ****2.2 寄存器堆****

[](https://soc.ustc.edu.cn/COD/lab3/img/cpu/rf_struc.png)

本次实验我们需要在先前实验的寄存器堆基础上，为其添加一组 debug 接口 debug\_reg\_ra、debug\_reg\_rd。这一对端口与数据读端口功能一致，只用于仿真与上板时的调试。CPU 在正常运行时并不会用到这两个端口。

### ****2.3 数据选择器****

[](https://soc.ustc.edu.cn/COD/lab3/img/cpu/sel.png)

ALU 的两个源操作数可能是寄存器堆的输出，也可能是立即数，还可能是当前 PC 的值。为此，我们需要为源操作数添加数据选择器，用于控制 ALU 的数据来源。一个简单的参数化选择器实现如下：

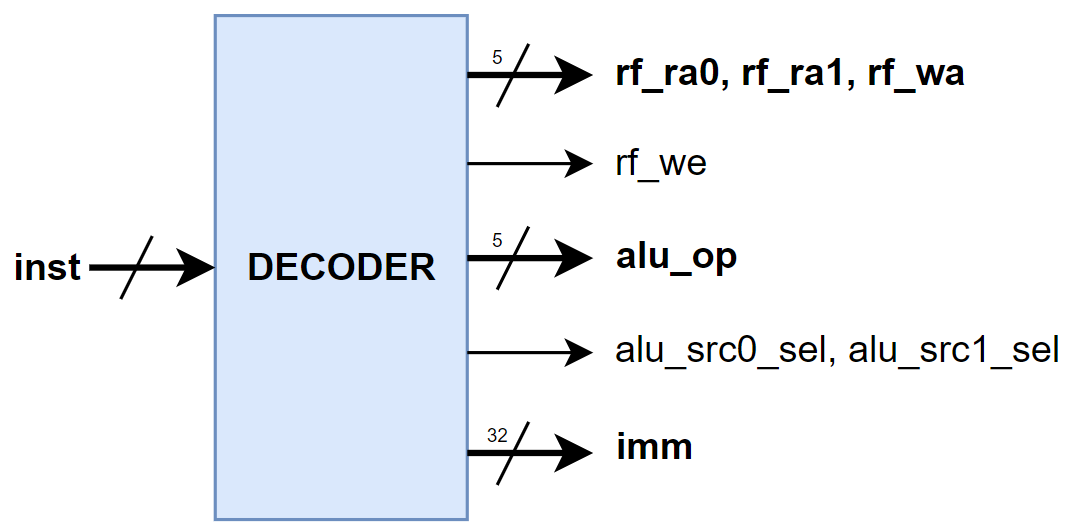


对于 LA32R 和 RV32I 指令集来说，ALU 的数据来源为：

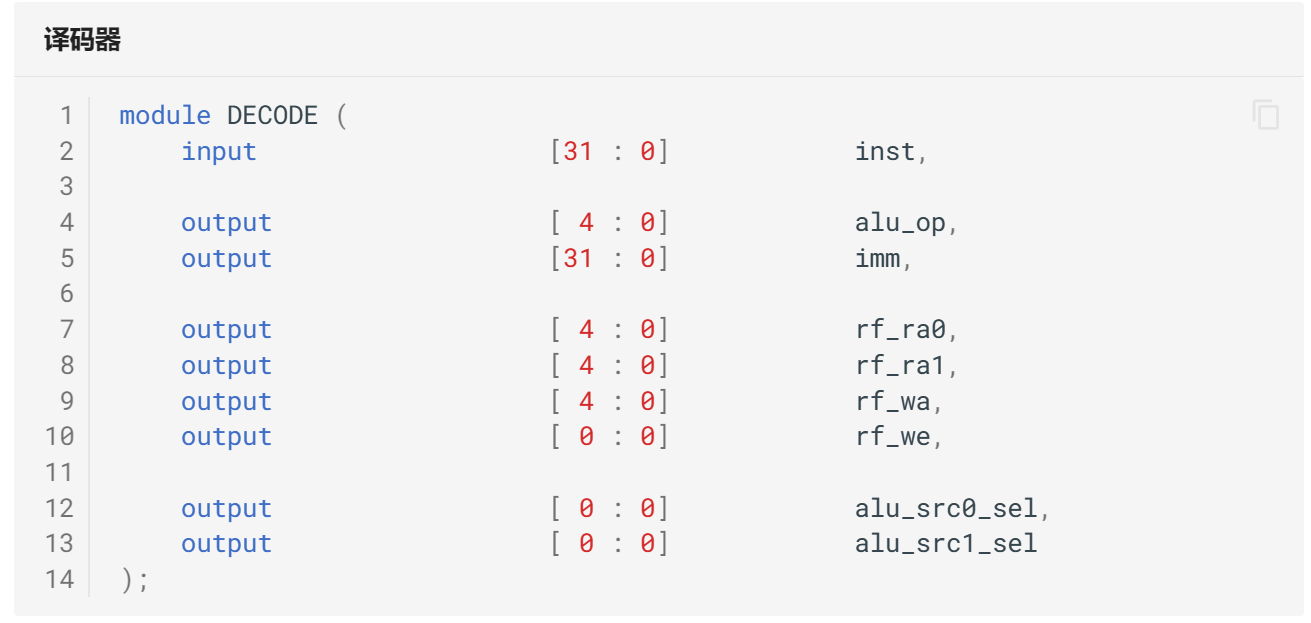
* alu\_src0：可能来自 PC，可能来自寄存器堆；
* alu\_src1：可能来自立即数，可能来自寄存器堆。

### ****2.4 译码器****

译码器是 CPU 的心脏，它负责根据输入的指令生成相应的控制与数据信号。下面是本次实验中译码器的结构示意图。

[](https://soc.ustc.edu.cn/COD/lab3/img/cpu/decode_struc.png)

其端口代码为：



各个输出端口的作用是：

* alu\_op：ALU 的运算模式码；
* imm：经过扩展的立即数；
* rf\_ra0、rf\_ra1：寄存器堆的读地址；
* rf\_wa：寄存器堆的写地址；
* rf\_we：寄存器堆的写使能；
* alu\_src0\_sel、alu\_src1\_sel：ALU 的源操作数选择信号。

译码器模块为一个巨大的组合逻辑单元，所有的信号都根据 inst 的数值，结合对应指令集的译码规则得出。需要指出的是：rf\_ra0、rf\_ra1 和 rf\_wa 三条信号较为特殊，因为其在指令中的位置是固定的，可以直接通过截取得到；其他信号则需要判断指令的类型后，通过更为复杂的逻辑得到。

一条指令没有这个信号该怎么办？这是一个很好的问题。以下面两条 LA32R 指令为例：

* 算数立即数右移：srai.w rd, rj, imm（RISC-V 中为 srai rd, rs1, shamt）
* 算数右移：sra.w rd, rj, rk（RISC-V 中为 sra rd, rs1, rs2）

srai.w 指令并没有 rk 字段，因此也就不需要译码器的 rf\_ra1 端口。我们自然可以让此时 rf\_ra1 的输出为 0，也可以按照相同的逻辑读取指令对应部分的值（尽管此时该字段为立即数），只需要保证后续送入 ALU 的源操作数是正确的即可。

同理，对于 sra.w 指令，译码器的 imm 端口可以是任意的数值，只需要保证后续送入 ALU 的源操作数是正确的即可。

# 思考与练习

1. 根据上面的内容，实现RV32I或LA32R指令集下的简单单周期CPU。