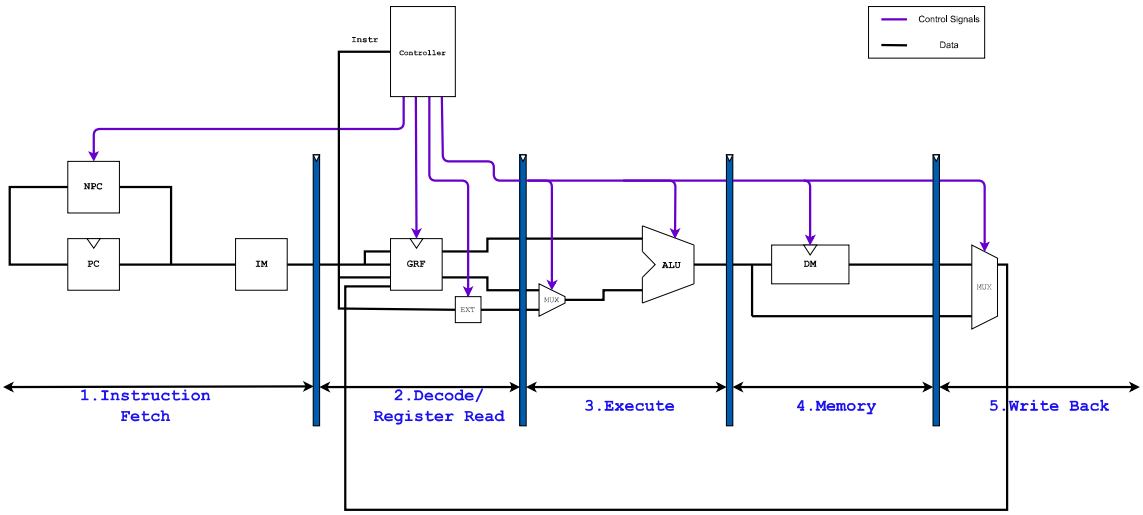
译码器

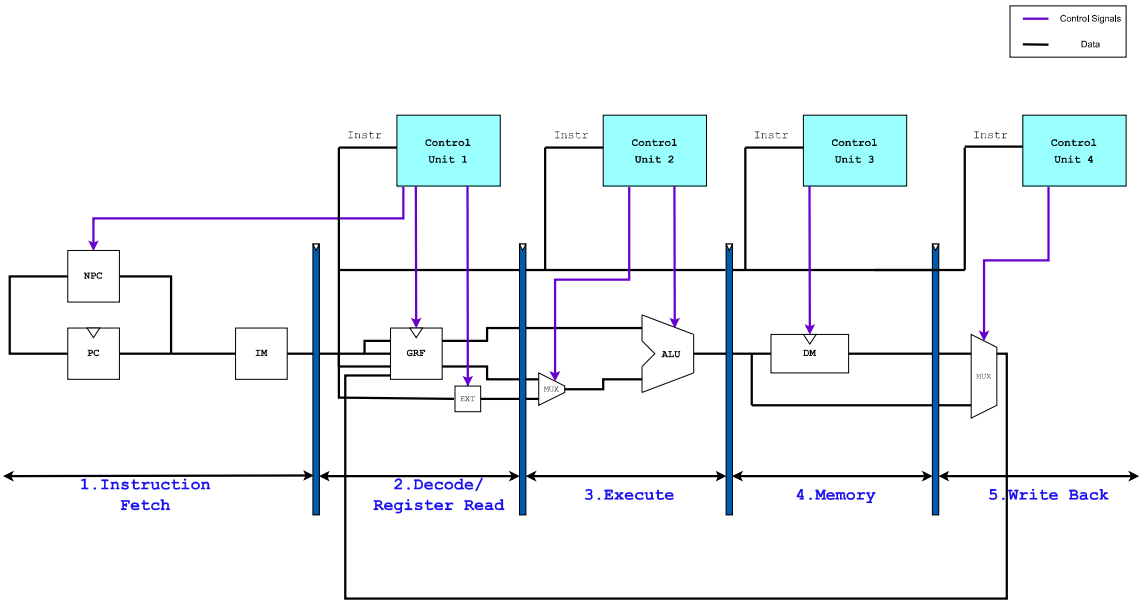
面对复杂的工程，我们会有多种方式对指令进行译码，下面将介绍主流的两种译码方式与两种译码风格，引导大家对其进行思考并作出适合自己的设计。当然，下面的所介绍的方法与风格都各有千秋，本文仅抛砖引玉，希望同学们能够斟酌后再进行设计编码。如果对控制器的设计有很好的想法或观点，欢迎在讨论区与大家进行分享。

译码方式

集中式译码：在取指令（F 级）时或者读取寄存器阵列信息（D 级）前，将所有的控制信号全部解析出，然后让其随着流水往后逐级传递。使用这种方法，只需要在初始对指令进行一次译码，减少了后续流水级的逻辑复杂度，但流水级之间需要传递的信号数量很大。



分布式译码：每一级都部署一个控制器，负责译出当前级所需控制信号。这种方法较为灵活，“现译现用”有效降低了流水级间传递的信号量，但是需要实例化多个控制器，增加了后续流水级的逻辑复杂度。



译码风格

指令驱动型：整体在一个 case 语句之下，通过判断指令的类型，对所有的控制信号一一赋值。这种方法便于指令的添加，不易遗漏控制信号，但是整体代码量会随指令数量增多而显著增大。

```
case (Instr[31:26])
    R:  begin
        case (Instr[5:0])
            add: begin
                grf_en  = 1;
                dm_en   = 0;
                alu_op  = 0;
                npc_sel = 0;
                // ...
            end
            // ...
        endcase
    // ...
endcase
```

如果重复部分太多，非常不推荐复制粘贴的做法，可以尝试用宏将其抽象出来。

控制信号驱动型：为每个指令定义一个 wire 型变量，使用或运算描述组合逻辑，对每个控制信号进行单独处理。这种方法在指令数量较多时适用，且代码量易于压缩，缺陷是如错添或漏添了某条指令，很难锁定出现错误的位置。

```
wire    R       = (op == 6'b000000);
wire    add     = R & (func == 6'b100001);
wire    sub     = R & (func == 6'b100011);
// wire ...
```

```
assign grf_en = (add | sub | /*...*/) ? 1'b1 : 1'b0;

// assign ...
```

思考题

简要描述你的译码器架构，并思考该架构的优势以及不足。