# 2 | 组合逻辑实验


跨越阿尔卑斯山圣伯纳隘道的拿破仑（雅克·路易·大卫）

就像一只回旋镖，你已经忘记曾经把他掷出，却在意想不到的时候飞回你手中。

~ · ~

## 2.1 | 实验目的

1. 学习用 verilog 设计较复杂的组合逻辑电路
2. 进一步熟悉 vivado 工具

## 2.2 | 实验内容

| 英文 | 中文 |
|---|---|
| fs(fetch stage) | 取值级 |
| ds(instruction decode stage) | 译码级 |
| es(execute stage) | 执行级 |
| bus | 总线 |
| fs_to_ds_bus | 取指级到译码级总线 |
| ds_to_es_bus | 译码级到执行级总线 |
| rx(register x) | 寄存器 x 地址 |
| ry(register y) | 寄存器 y 地址 |
| rx_value | 寄存器 x 的值 |
| ry_value | 寄存器 y 的值 |
| op(operation code) | 操作码 |
| operand | 操作数 |

表 2.1: 术语表

### 2.2.1 | CPU 译码逻辑模块

译码逻辑模块的主要工作是解析指令（由下文的 RX、RY、OP 组成）的操作码（即 OP）和操作数（由 RX_VALUE、RY_VALUE 组成），将指令转化为控制信号并读取通用寄存器堆生成操作数（RX_VALUE、RY_VALUE）。本模块的输入输出定义如下：

```
1  module id(
2      input  wire [15:0] fs_to_ds_bus  ,  // from other modules
3      output wire [27:0] ds_to_es_bus  ,  // to other modules
4      output wire [ 1:0] rx            ,  // 读入rx寄存器（other modules）的值
5      output wire [ 1:0] ry            ,  // 读入ry寄存器（other modules）的值
6      input  wire [ 7:0] rx_value      ,  // rx寄存器（other modules）返回的值
7      input  wire [ 7:0] ry_value         // ry寄存器（other modules）返回的值
8  );
```

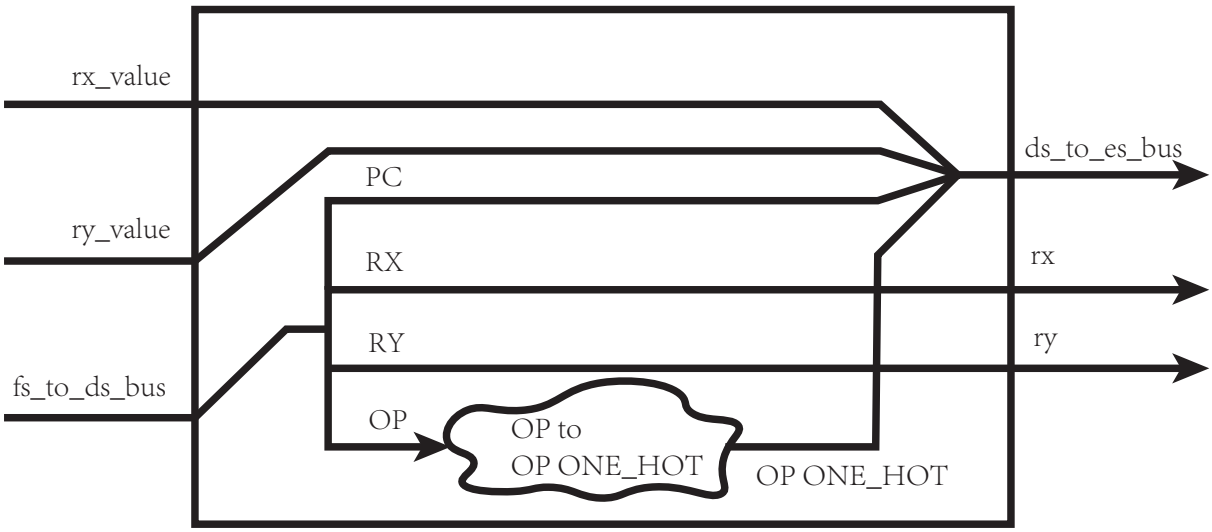


图 2.1: 内部结构

其中，`fs_to_ds_bus` 为从 `fs` 到 `ds` 的总线（bus），包含了从其他模块到译码模块的信息（即本模块输入）。`ds_to_es_bus` 为从 `ds` 到 `es` 的总线（bus），包含了从译码模块到其他模块的信息（即本模块输出）。

模块与其他模块的交互关系如下：

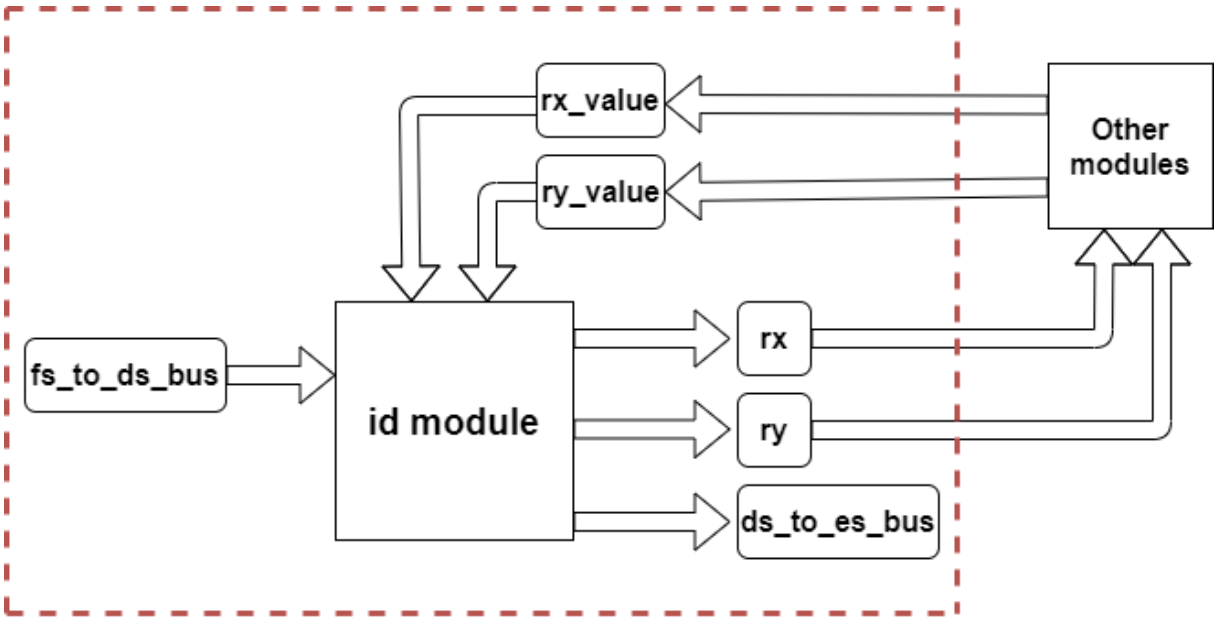


图 2.2: 顶层结构

端口中总线的编码格式如下：

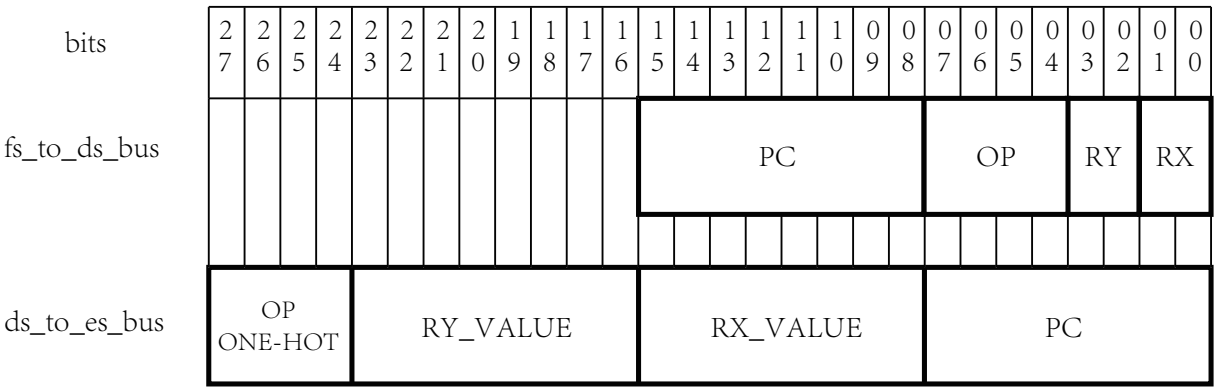


图 2.3: 端口编码格式，其中 bits 代表第 x 位，最右侧为第 00 位，最左侧为第 27 位。例：`fs_to_ds_bus` 的 PC 域表示为 `fs_to_ds_bus[15:8]`

`fs_to_ds_bus` 总线中，PC 代表一个立即数，**即一个数字，我们无需关心 PC 的含义**；OP 代表指令操作码；RX 与 RY 代表两个寄存器地址，译码模块需要将这两个域输出到对应的 `rx` 与 `ry` 端口，从而得到寄存器返回的值 `rx_value` 与 `ry_value`。

`ds_to_es_bus` 总线中，PC 应与输入的 `fs_to_ds_bus` 总线中 PC 相同；`RX_VALUE` 与 `RY_VALUE` 应与模块输入的 `rx_value` 与 `ry_value` 相同；`OP ONE_HOT` 为指令操作码的独热码形式，需要由译码模块将输入的 OP 转为独热码形式后输出。

其中，`fs_to_ds_bus` 中 PC 域代表一个八位立即数，在译码时无需关注，只需要将其赋值到 `ds_to_es_bus` 中 PC 域即可。

`fs_to_ds_bus` 中存在四位操作码域，译码模块需要将这四位的操作码转为独热码的形式输出到 `ds_to_es_bus` 中对应域：`ds_to_es_bus` 的 27 位到 24 位分别代表 move、add、sub 与 mul 指令。

`fs_to_ds_bus` 的 RY 域与 RX 域代表两个地址，输出到通过译码模块的 `rx` 与 `ry` 端口即可。此步骤是为了模拟读取寄存器堆的过程，`rx_value` 与 `ry_value` 的值将会取决于译码模块给出的 `rx` 与 `ry`。`ds_to_es_bus` 中 `RX_VALUE` 与 `RY_VALUE` 域代表 `rx_value` 与 `ry_value`，将译码模块的输入赋值到这两个域即可。

即译码模块需要完成以下操作：

1. 指令解码：将原始指令拆解为不同字段，如操作码，源寄存器，目标寄存器，立即数等
2. 识别操作码：确定这条指令要执行什么操作
3. 访问寄存器：根据指令要求，从指定的寄存器中读取源寄存器的值
4. 生成控制信号：根据操作码和其他字段生成控制信号，这些信号将在后续的执行，访存和写回阶段发挥作用

**以上是基于译码模块原理的分析，并且需要具有相关背景知识。下面，将会以更简单的语言描述本实验需要的工作：**

1. 新建工程，完成 CPU 译码逻辑模块的 Verilog 描述
2. 将 `fs_to_ds_bus` 的 PC 域(下简写为 `fs_to_ds_bus.PC`)赋值给 `ds_to_es_bus.PC`

3. 将 `fs_to_ds_bus.OP` 转换为独热码,`fs_to_ds_bus.OP == 0001` 时,`ds_to_es_bus.OP_ONE_HOT` 赋值为 `1000`，以此类推
4. 将输入的 `rx_value` 赋值给 `ds_to_es_bus.RX_VALUE`
5. 将输入的 `ry_value` 赋值给 `ds_to_es_bus.RY_VALUE`
6. 将 `fs_to_ds_bus.RY` 赋值给 `ry` 端口
7. 将 `fs_to_ds_bus.RX` 赋值给 `rx` 端口

涉及的指令编码格式如下

表 2.2: 指令格式

| 指令名称 | 指令格式 | | | |
|---|---|---|---|---|
| | 指令操作码 | 操作数 1 | 操作数 2 | 附加数据段 |
| Move | 0001 | Rx | Ry | 无 |
| Add | 0010 | Rx | Ry | 无 |
| Sub | 0011 | Rx | Ry | 无 |
| Mul | 0100 | Rx | Ry | 无 |

### 2.2.2 | ALU 设计

ALU（Arithmetic Logic Unit，算术逻辑单元）是计算机中的一种重要组件，它可以实现算术运算和逻辑运算。CPU 的绝大部分运算任务都会在 ALU 中完成。ALU 将会根据译码模块提供给 ALU 的信息：源操作数 1、源操作数 2 和操作码进行运算，并将运算结果正确输出。

注：在 ALU 单元中，要注意区分运算类型（操作码），同时保证操作数完成正确的算术逻辑运算。

在本实验中，需要实现一个 8 位 ALU，能实现下面给出的 12 种运算操作：

表 2.3: ALU 操作码

| **操作码** | **符号** | **介绍** |
|---|---|---|
| 12'h001 | $+$ | 加 |
| 12'h002 | $-$ | 减 |
| 12'h004 | & | 按位与 |
| 12'h008 | \|\| | 逻辑或 |
| 12'h010 | << | 逻辑左移 $a << b[1:0]$ |
| 12'h020 | >>> | 算术右移 $a >>> b[1:0]$ |
| 12'h040 | >>>> | 循环右移 $a >>>> b[1:0]$ |
| 12'h080 | $<_s$ | 小于比较 |
| 12'h100 | $<_u$ | 无符号小于比较 |
| 12'h200 | +& | 保留最高位进位加法，如有进位，舍弃最后一位 |
| 12'h400 | ^ | 异或 |
| 12'h800 | $\iota$ | 见下面操作定义 |

其中，$\iota$ 操作定义如下：
根据操作数 2 的高四位中最低位的 1 的位置对操作数 1 进行不同的拼接操作：

表 2.4: *itoa* 操作

| 1 的位置 | 拼接方式 |
|---|---|
| 4'bxxx1 | {oprand1[1:0], oprand1[7:6]} |
| 4'bxx10 | {oprand1[5:4], oprand1[3:2]} |
| 4'bx100 | {oprand1[7:6], oprand1[3:2]} |
| 4'b1000 | {oprand1[5:4], oprand1[1:0]} |

然后与操作数 2 的低四位进行拼接（操作数 2 的低四位位于拼接的高位）后进行循环位移，操作数 1 中未用来进行上述拼接操作的剩余四位按照操作数 1 中原来顺序进行拼接，最低位作为位移方向判断，0 为右移，1 为左移，高三位作为位移量。

例：oprand1 = 8'b1011_0110 oprand2 = 8'b1100_1110
进行位移的数为：8'b1110_1001 操作数 1 中剩下 4 位为：4'b1110，位移方向：右移，位移量：7
最后结果：8'b1101_0011

**注：本实验不会出现操作数 2 的高四位为零的情况，因此无需考虑。**

### 2.2.3 | 加法器实现

实现 32 位逐位进位加法器、32 位选择进位加法器。

1. 学习课件中逐位进位加法器、选择进位加法器的原理；
2. 新建工程，完成两种加法器的 Verilog 描述，两种加法器请都使用一位加法器作为基本模块开始搭建；一位加法器请根据实验一自行设计，模块名称及端口定义如下：

```
1    module csadd32 (a,b,cin,s,cout);    //选择进位加法器
2    module rcadd32 (a,b,cin,s,cout);    //逐位进位加法器
```

3. 编写测试激励：可更改实验 1 中的激励产生代码，按照自己的思路产生测试数据，通过仿真验证加法器功能。
4. 逐位进位加法器高层加法器需要等待低层加法器的计算结果，如果使用不带延时的一位加法器进行综合前仿真，不带延时的加法器将在“一瞬间”计算完毕，这样体现不出来逐位进位加法器的传递过程。故我们在此处手动添加延时来模拟实际情况，感兴趣的同学可以搜索有关综合后仿真的资料，自己撰写仿真代码。

   带延时的一位加法器参考代码如下：

```
1  module add1(
2          input a,
3          input b,
4          input cin,
5          output sum,
6          output cout);
7          assign #4 sum  = a ^ b ^ cin;
```

```
        assign #2 cout = (cin==1) | (cin==0) ? (a & cin) | (b & cin)| (a & b) :
            1'bx;
endmodule
```

5. 参考仿真代码如下:

```
`timescale 1ns / 1ps
module add32_tb();
    reg [31:0]a;
    reg [31:0]b;
    reg cin;
    reg clk;
    wire [31:0]s0,s1,s2;
    wire cout0,cout1,cout2;
    initial begin
        a = 4'bxxxx;
        b = 4'bxxxx;
        cin = 1'bx;
        clk = 0;
    end
    always #100 clk = ~clk;
    always@ (posedge clk) begin
        a   = {$random} % 2**30;
        b   = {$random} % 2**30;
        cin = {$random} % 2;

           // 请同学们根据波形图以及两种加法器的区别思考，这里延时 150ns 将信号
               cin 赋值为 X 的作用是什么？如果删除这两句，波形图会发生什么变化
        // 也请同学们仔细思考，选择加法器到底 “快” 在哪里
        #150;
        cin = 1'bx;
    end

    csadd32 A(a,b,cin,s0,cout0);//选择加法器
    rcadd32 B(a,b,cin,s1,cout1);//逐位加法器

endmodule
```

## 2.3 | 实验要求

1. 在实验报告中提交 Verilog 代码、RTL 分析图、测试激励代码、必要的仿真波形图;
2. 对仿真波形图，需要给出适当的文字说明;
3. 提交实验报告;
4. 本实验无需上板，但需验收。

   管脚约束见 Ego1 手册。

# B | 数的比较

## B.1 | 无符号数比较

对于无符号数比较（SLTU），我们可以通过减法来实现。无符号数减法 $A-B$ 如果产生借位（即进位位 `adder_cout` 为 0），则表明 $A<B$；如果没有借位（`adder_cout` 为 1），则表明 $A\geq B$。

```
// 无符号数比较（SLTU）
assign adder_a   = alu_src1;  // 被减数 A
assign adder_b   = ~alu_src2; // 减数 B 的按位取反，进行减法
assign adder_cin = 1'b1;      // 补码减法时加上 1
assign {adder_cout adder_result} = adder_a + adder_b + adder_cin; // 执行 A - B

// 结果判断：通过 adder_cout 确定是否有借位
assign sltu_result[0] = ~adder_cout; // 借位时结果为 1，表示 A < B
```

`adder_a` 和 `adder_b` 是加法器的两个输入。`adder_b` 被设置为 `~alu_src2`，即 $B$ 的按位取反加 1，以实现减法。

- 进位位判断：
  - `adder_cout` 是加法器的进位位。
  - 如果 `adder_cout = 0`，说明发生了借位，表示 $A<B$。
  - 如果 `adder_cout = 1`，则 $A\geq B$。
- 比较结果判断：
  - `sltu_result[0] = ~adder_cout` 是最终的比较结果。
  - 如果 `~adder_cout = 1`，表示 $A<B$。

## B.2 | 有符号数比较

对于有符号数比较（SLT），符号位（最高位）决定了数的正负。因此，我们可以通过符号位和减法结果的符号来判断两个有符号数的大小。

如果两个数的符号不同，负数一定小于正数；如果两个数符号相同，使用补码减法的结果符号来判断。

```
// 有符号数比较（SLT）
assign adder_a   = alu_src1;  // 被减数 A
assign adder_b   = ~alu_src2; // 减数 B 的按位取反，进行减法
assign adder_cin = 1'b1;      // 补码减法时加上 1
assign {adder_cout adder_result} = adder_a + adder_b + adder_cin; // 执行 A - B
// 结果判断：通过符号位判断是否 A < B
assign slt_result[0] = (alu_src1[31] & ~alu_src2[31]) // A负B正 => A < B
| (~(alu_src1[31] ^ alu_src2[31]) & adder_result[31]); // 符号相同，看减法结果符号
```

- 符号位判断：
  - `alu_src1[31]`：$A$ 的符号位，表示 $A$ 是正数还是负数。

- alu_src2[31]：$B$ 的符号位。
- 如果 $A$ 是负数，$B$ 是正数，则 $A < B$，设置 slt_result[0] = 1。

- 减法结果判断：
  - 如果 $A$ 和 $B$ 符号相同，通过 $A - B$ 的符号位 adder_result[31] 来判断。如果 adder_result[31] = 1，则 $A < B$。

## B.3 | 使用内置函数判断

```
$signed(expression)   // 将表达式转换为有符号数
例：$signed(a) < $signed(b)
$unsigned(expression) // 将表达式转换为无符号数
例：$unsigned(a) < $unsigned(b)
```

**注：verilog 中默认为无符号比较，且参与比较的任意一个数为无符号的表达式均为无符号。**