本章简单阐述硬件电路设计的基础理论知识。

# 逻辑代数

**逻辑变量**是逻辑代数中参与逻辑运算的变量，一般用字母A、B、C……表示。逻辑变量只能取两种值：0或1。在硬件电路设计当中，分为正负逻辑的规定。一般采用正逻辑规定：高电平为逻辑1，低电平为逻辑0。负逻辑规定则相反。逻辑运算有三种基本的运算，分别为与、或、非，也可以称为合取、析取、否定，也可以直接记作AND、OR、NOT。

## 逻辑运算

下表列出的是逻辑运算的记法与运算规则，加粗的是基本运算。其他的逻辑运算符都可以通过基本的逻辑运算组合得到。

| 逻辑运算 | 记法 | 规则 |
| --- | --- | --- |
| **与** | $A \bigwedge B$ | 当且仅当A、B都为1时，结果为1；其他情况结果为0 |
| **或** | $A \bigvee B$ | 当且仅当A、B都为0时，结果为0；其他情况结果为1 |
| **非** | !A | 结果与A的取值相反 |
| 异或 | $A \bigoplus B$ | 当A、B取值相反时，结果为1；其他情况结果为0 |
| 同或 | $A \equiv B$ | 结果与异或相反（即对异或运算取反得到） |
| 实质蕴涵 | $A \rightarrow B$ | 逻辑函数表达为：$A \rightarrow B$ = $!A \bigvee B$ |

## 逻辑代数运算律

| 逻辑或 | 逻辑与 | 运算律 |
| --- | --- | --- |
| $A\bigvee(B\bigvee C)=(A\bigvee B)\bigvee C$ | $A\bigwedge(B\bigwedge C)=(A\bigwedge B)\bigwedge C$ | 结合律 |
| $A\bigvee B=B\bigvee A$ | $A\bigwedge B=B\bigwedge A$ | 交换律 |
| $A\bigvee(A\bigwedge B)=A$ | $A\bigwedge(A\bigvee B)=A$ | 吸收律 |
| $A\bigvee(B\bigwedge C)=(A\bigvee B)\bigwedge(A\bigvee C)$ | $A\bigwedge(B\bigvee C)=(A\bigwedge B)\bigvee(A\bigwedge C)$ | 分配律 |
| $A\bigvee(!A)=1$ | $A\bigwedge(!A)=0$ | 互补律 |
| $A\bigvee A=A$ | $A\bigwedge A=A$ | 幂等律 |
| $A\bigvee 0=A$ | $A\bigwedge 1=A$ | 有界律 |
| $A\bigvee 1=1$ | $A\bigwedge 0=0$ | 有界律 |
| | | |

| | | |
|---|---|---|
| !0=1 | !1=0 | 0与1互补 |
| !(A⋁B)=(!A)⋀(!B) | !(A⋀B)=(!A)⋁(!B) | 对偶律 |
| !(!A)=A | | 对合律 |

# 逻辑代数的基本规则

### 代入规则

任何一个含有变量X的等式，如果将所有出现X的位置都替换成一个逻辑函数F，此等式仍然成立。

### 对偶规则

设F是一个逻辑函数式，如果将F中的所有 `*` 变成 `+`， `+` 变成 `*`，0变成1，1变成0，而变量保持不变。那么就得到了一个逻辑函数式F'，F'称为F的对偶式。如果两个逻辑函数F和G相等，那么它们的对偶式F'和G'也相等。

### 反演规则

设F是一个逻辑函数式，如果把F中的所有 `*` 变成 `+`， `+` 变成 `*`，0变成1，1变成0，原变量变成反变量，反变量变成原变量。那么就得到一个逻辑函数式(!F)。

# 逻辑函数的标准形式

逻辑变量的**逻辑与运算**叫做**与项**，与项的逻辑或运算构成了逻辑函数的**与或式**，也称为**积之和式**（SP form）。

逻辑变量的**逻辑和运算**叫做**和项**，和项的逻辑与运算构成了逻辑函数的**或与式**，也称为**和之积式**（PS form）。

对于n个变量的逻辑函数而言，如果它的与项包含全部n个变量，那么这个与项就称为该逻辑函数的**最小项**。如果它的或项包含全部n个变量，那么这个或项就称为该逻辑函数的**最大项**。

# 逻辑函数的化简

通过运用逻辑代数的基本规则和运算律对逻辑函数进行变换，可以得到逻辑表达式的最简形式（最简与或式/最简或与式）。逻辑函数的化简最常使用的方法是**卡诺图**。而卡诺图就是真值表的另一种表示形式。

### 真值表

真值表用于计算逻辑表达式在每种逻辑变量取值的组合上的值，也就是说，真值表的行数与逻辑表达式的变量数目相关。逻辑表达式中有n个变量时，真值表的行数为2^n行。例如，一个半加器的真值表如下：

| | | | |
|---|---|---|---|

| A | B | Cout | S |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

从真值表中可以得到输出Cout和S的逻辑表达式：

$$Cout = A * B$$

$$S = A \bigoplus B$$

### 卡诺图

卡诺图是真值表的变形，它将有n个变量的逻辑函数的2^n个最小项组织在给定的长方形表格当中。需要注意的是，表格需要遵循最小项之间逻辑相邻性以及几何相邻的统一。简单来说，就是保证每个最小项格表示的最小项与四个（如果有）相邻的最小项格表示的最小项有且只有一个变量互补。例如，逻辑函数：

$$Z_{(A,B,C)} = \Sigma_m(3,5,6,7)$$

的卡诺图为：

| A/BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| **0** | | | 1 | |
| **1** | | 1 | 1 | 1 |

使用卡诺图来化简逻辑函数，是根据一定的原则“画圈”来合并相邻的最小项。原则如下：

- 从大圈开始画起，尽量画大圈，但每个圈内只能含有2^n个相邻项。圈需要注意对边相邻性和四角相邻性。
- 圈的个数尽量少。
- 卡诺图中所有取值为1的方格均要被圈过。
- 新画的圈内至少要有一个没有被圈过的方格。

画圈后，写出每个圈的最简与项，得到的就是最简的逻辑表达式。如上面的卡诺图得到的逻辑表达式为：

$$f = A * C + B * C + A * B$$

# 硬件电路设计概述

在Verilog以及VHDL等硬件描述语言（HDL）出现之前，硬件电路的设计人员主要依靠的是电路版图进行设计（74系列）。而在HDL出现后，直到现在，数字集成电路的设计可以分为以下基本步骤：系统定义、寄存器传输级设计、物理设计。而根据逻辑的抽象级别，设计又分为系统行为级、寄存器传输级、逻辑门级。在整个过程当中，设计人员需要使用数个甚至更多数量的工具、语言来设计、验证数字集成电路。而对于我们的实验以及课程来说，我们主要着眼于寄存器传输级的电路设计以及验证，也就是RTL级的电路设计。

## 寄存器传输级设计

寄存器传输级（Register-Transfer Level, RTL）实际上就是一种对电路的抽象模型，这种模型实际上就是根据数字信号在硬件寄存器、存储器、组合逻辑装置和总线等逻辑单元之间的流动，以及其逻辑代数运作的方式来确定的。在Verilog以及VHDL等的HDL语言中，被创建为对实际电路的高层次描述。

数字电路由两个主要的元素构成：寄存器和组合逻辑电路。寄存器通常由D触发器组成，可以按照时序脉冲来进行同步的时序操作，且能够保存逻辑状态信息。而组合逻辑电路则由逻辑门组成。使用HDL来设计数字集成电路，设计人员通常不需要在晶体管级别进行设计，而是在更高的抽象级别进行工程设计。设计人员只需要声明寄存器（和计算机编程语言声明变量类似），然后使用运算符以及条件语句等来表述组合逻辑的功能即可。上述的这种级别的设计就是RTL设计。

```
module PC(
  input         clock,
  input         reset,
  input  [31:0] io_pc_in,
  output [31:0] io_pc_out
);
  reg [31:0] pc;
  assign io_pc_out = pc;
  always @(posedge clock) begin
    if (reset) begin
      pc <= 32'h0;
    end else begin
      pc <= io_pc_in;
    end
  end
endmodule
```

上面的Verilog代码描述的是一个极其简单的电路结构。实际上，电路当中只定义了一个寄存器 `pc`，而 `pc` 将在每个时钟周期的上升沿（如果 `reset` 不为1）写入输入口 `io_pc_in` 的值，而输出口 `io_pc_out` 则输出 `pc` 的值。可以发现，在RTL级别的电路设计当中，设计人员不需要考虑寄存器的内部结构（门电路），而只需要关注信号在寄存器之间是如何流动的。如在上述例子当中，信号从输入口输入，写入到寄存器当中，然后经由输出口输出。同时，RTL级别的设计当中时序的信息是显式的，也

就是说设计人员必须定义时序信息。

在RTL级的电路设计当中，可以分为几种电路。如果寄存器的输出端和输入端存在环路，那么这样的电路可以称为“状态机”。如果寄存器之间有连接但是没有环路，则这样的电路称为“流水线”结构。

RTL设计的描述通常会通过EDA工具转换成逻辑门电路连接的网表描述，然后再经过布线等的步骤，可以得到物理的电路。同时，这一过程还会通过使用各种模拟仿真工具来验证RTL级的描述的功能是否正确。

# 高层次设计工具入门

现今集成电路的规模愈来愈大，传统的HDL设计语言和工具开始无法满足设计人员庞大的设计需求。这源自于传统HDL语言的局限性，它们都是基于过程式的编程语言。对于大规模的集成电路设计来说，模块性以及重构性式非常重要的，从另一个角度上来说，敏捷开发在软件工程领域已经很成熟，但是在硬件设计领域，它还没有被大规模使用。将硬件设计与高级程序编程语言有一个很突出的优势是，像Python、Java这类的语言具有面向对象的特性，这是传统的HDL所不具备的。Chisel是最先的尝试，UCB的计算机体系结构团队在2012年发布了Chisel的第一个版本，并一直延续到现在。Chisel的创举在于，它本质仍然是对硬件的描述，而不是基于类似于HLS（高层次综合）的方法。但是它与HDL也有些许的不同，它重在硬件的构筑而不是描述，因此Chisel并不是HDL，而是HCL（Hardware Construction Language）。Chisel通过将硬件设计嵌入在Scala语言当中来获得宿主语言的各种优势，包括面向对象的特性和函数式编程。但正如前一章节所阐述过的，Scala的使用门槛比较高，而且工具生态、社区生态不太活跃，学习曲线比较陡峭，因此它不利于广泛的推广。

# PyHCL

PyHCL是笔者团队所设计开发的基于Python的硬件设计工具，其思想借鉴于Chisel。在这一节当中，会介绍PyHCL最基础的使用方式。PyHCL的架构，或者说设计路线，是使用预先定义好的PyHCL原语（Python库）来构建硬件电路，然后经过PyHCL内核的综合后生成目标FIRRTL中间代码，再经过FIRRTL的编译器生成可综合的Verilog代码。下面，将会介绍PyHCL最简单的使用入门方法，在往后的使用章节中，会逐步的结合实验内容介绍PyHCL的使用方法。

### 模块

与Verilog相似，在PyHCL当中，一个具有功能的电路实体一般作为一个模块来看待。比如一个全加器，一个状态机或者一个硬件算法核。要声明一个模块，我们需要声明一个继承了 `Module` 基类的Python类。前面提到过了，通过嵌入在高级程序设计语言当中，可以充分利用宿主语言的特性。因此，PyHCL的模块具有面向对象的各种特性，并且还能够支持函数式编程。我们可以看看模块定义的例子：

```
class Mo(Module):
  # 模块内部逻辑
```

PyHCL的Module基类的设计比较特殊，其使用了Meta类来进行构建。因此在模块内的逻辑设计不是通过构造函数来创建的（也就是 `__init__()`）。有时候，为了充分利用参数化的模块功能，可以使用函数内联的方式来声明类，比如：

```
def mo(width):
  class Mo(Module):
    # 模块内部逻辑
  return Mo()
```

`width` 是可以下放到模块 `Mo` 当中的参数，或者在函数内部进行一定的操作。PyHCL的模块可以继承，可以组合，也就是在别的模块类当中实例化。

### 输入输出端口

模块是PyHCL的基本组成单位，而每个模块需要定义它的输入输出端口，来作为与外界沟通的方式。这同时也体现了模块的封装性。PyHCL规定模块必须定义IO口。可以通过实例化一个 `IO` 的类来定义IO口：

```
class Adder(Module):
  io = IO(
    a=Input(U.w(1)),
    b=Input(U.w(1)),
    cout=Output(U.w(1)),
    s=Output(U.w(1))
  )
```

模块 `Adder` 有4个IO口，分别是 `a` ， `b` ， `cout` 以及 `s` 。端口的方向通过实例化为 `Input()` 或者 `Output()` 类来决定，而参数则是端口的类型，在这里可以先简单的了解到 `U.w(1)` 就是指无符号的1位整数，在之后会逐步介绍类型相关的内容。

### 模块层次

模块定义后，它需要被使用。使用模块有两种方法，一种是作为顶层模块直接输出，另一种是被上层模块实例化调用。如果当前模块就是顶层模块，那么直接调用PyHCL的 `Emitter` 类来综合、编译当前模块即可：

```
# 综合并生成模块Adder的Verilog代码，文件名为Adder.fir.v
Emitter.dumpVerilog(Emitter.dump(Emitter.emit(Adder()), "Adder.fir"))
```

在其他模块中，可以实例化子模块。例如，在 `Tile` 模块中实例化一个 `Adder` 模块：

```
class Tile(Module):
  io = IO(
    # ...
  )
  adder = Adder()           # 实例化一个Adder模块
  io.output <<= adder.io.s  # Adder模块的输出s连接到Tile模块的输出口output
```

在模块内引用输入输出口，使用 `.` 运算符即可，如 `io.s`， `io.a` 。同理，引用实例化的子模块的输入输出口，则需要先引用模块的 `io` 成员，再引用对应的输入输出端口。 `<<=` 可以先简单理解为电路的连接符号，也就是把 `adder` 的输出口 `s` 连接到当前模块的输出口 `output` 当中。

# 实验内容与练习

## 全加器

全加器（Full Adder）是一个非常经典且简单的逻辑电路组件，它有3个输入端口：数据输入端口 `a`， `b`，以及进位输入端口 `cin`，以及2个输出端口：数据输出端口 `s` 以及进位输出端口 `cout` 。已知全加器的真值表如下：

| a | b | cin | s | cout |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

请画出输出结果 `s` 以及 `cout` 的卡诺图，并写出 `s` 以及 `cout` 的关于 `a`， `b`， `cin` 的逻辑表达式。

## 你的第一个模块

参考上述的半加器的真值表，用PyHCL尝试实现一个半加器模块。这里给出一个不完整的例子。在PyHCL中，按位与的运算符是 `&`，按位或的运算符是 `|`，按位异或的运算符是 `^`：

```
class HalfAdder(Module):
  io = IO(
    a=Input(U.w(1)),
    b=Input(U.w(1)),
    s=Output(U.w(1)),
    cout=Output(U.w(1))
  )
  # 填充完整输出端口s以及cout的逻辑，例如：
  # io.s <<= ?
  # io.cout <<= ?

# 生成Verilog文件
if __name__ == '__main__':
    Emitter.dumpVerilog(Emitter.dump(Emitter.emit(HalfAdder()),
"HalfAdder.fir"))
```