

# SoC设计概论☆

## SOC:

是指在单一芯片上集成了数字电路、模拟电路、信号采集和转换电路、存储器、MPU、MCU、DSP、MPEG等，实现了一个系统的功能。

## 优点：

- 微型化
- 工作速度提升
- 功耗降低
- 可靠性提升

## soc特点：

### 优点

体积小、功耗低、可靠性高、成本低以及更完善的功能和更高的性能指标。

### 缺点

复杂性上升、设计成本高、开发时间长，完全改变了先前整机系统的总体设计方案。

## soc基本构成

### 五个基本构成

- 嵌入式处理器核，如MPU(微处理器)，MCU(微控制器)，DSP
- 存储器，如SRAM，SDRAM，Flash rom
- 专用功能模块
- IO接口模块，如USB，UART等多种功能模块
- 片内总线

## SOC类型

- 计算控制型——微处理CPU、数字信号处理器DSP☆
- 通信网络型——如移动手机
- 信号处理型

## soc应用领域

- 消费电子
- 通信设备
- 控制类设备

## soc系统级研究内容

- 软硬件协同设计技术
- 设计重用技术
- 与底层相结合设计技术

## soc设计关键技术

- 设计重用技术
- 低功耗设计技术
- 软硬件协同设计
- 总线架构
- 可测试性设计
- 设计验证
- 物理综合

## 可重用技术

- 基于IP核的模块级重用 ✓
- 基于平台的系统级重用 ✓

## IP核

IP核是指经过反复验证过的、具有特定功能的，可重复利用的逻辑块或数据块，用于专用集成电路（ASIC）或者可编辑逻辑器件（FPGA）

## IP核分类

- 软核
- 固核
- 硬核

类型	定义	使用灵活性	时序性能 可预测性
Soft Cores (软核)	行为级或 RTL 级 HDL 源代码	灵活度高，可修改设计 代码，与实现工艺无关。	IP 很难保护，时序性能无法保证，由使用者确定。
Firm cores (固核)	完成软核所有的设计外，还完成了门电路级综合和时序仿真等设计环节，一般以门电路级网表形式提交用户使用。	部分功能可修改，采用指定的实现技术，与实现工艺相关。	关键路径时序可控制
Hard cores (硬核)	基于某种半导体工艺的物理设计，已有固定的拓扑布局和具体工艺，并已经过工艺验证，具有可保证的性能。提供电路物理结构掩模版图和全套工艺文件。	不能修改设计，必须采用指定实现技术。	包含工艺相关的布局和时序信息，IP 很容易保护，多数的处理器和存储器。

## 低功耗设计技术

### 工艺级

- 降低电源供电电压，减少跳变功耗
- 多阈值工艺 MTCMOS (Multi-Threshold VT CMOS)
- 变阈值工艺VTCMOS (Variable Threshold VT CMOS)

### 电路级

- 减摆幅
- 电荷再循环总线结构

### 门级

- 低电压实现低功耗技术，主要采用互补CMOS实现来实现

### 寄存器传输级 (RTL) ❤

RTL 描述的是数字电路中逻辑操作和数据传输的行为，通常以寄存器之间的传输作为基本操作。

- 降低的方法主要是消除其产生的条件，如延迟路径平衡、用时钟信号同步减少故障、结构重构。

### 体系结构级低功耗技术

- 并行技术
- 流水线技术
- 预算算技术

## 算法级

- 总线翻转译码技术
- 编码技术

## 系统级

- 门控时钟技术
- 异步电路技术

---

## 软硬件协同技术

软硬件协同设计方法强调软件和硬件设计开发的并行性和相互反馈，克服了传统方法中把软件和硬件分开设计带来的种种弊端，能协调软件和硬件之间的制约关系，达到系统高效工作的目的。

### 关键技术

- 系统建模
- 软硬件划分技术
- 软硬件协同综合
- 软硬件协同仿真与验证

---

## 系统建模

### 系统描述模型

- 离散事件模型
- 有限状态机模型
- 通信进程网络模型
- Petri网模型
- 任务流图模型
- 控制数据流图模型

### 软硬件划分

设计软硬件划分的最优化算法

### 软硬件协同综合

- 通信体系结构综合——软硬件接口
- 软件综合——软件构建
- 硬件综合——硬件IP

## 软硬间协同仿真验证

- 模块/IP核级验证
- 软硬件协同仿真验证
- FPGA验证

包括：

- 黑盒验证
- 白盒验证
- 灰盒验证

## 总线结构

对SOC上芯核和电路模块等的互连

单总线、多总线和片上网络（NoP）的方式

### 总线结构特点

- 设计尽可能简单
- 较大灵活性
- 降低功耗

## SOC系统设计方法

- 自顶向下
- 自底向上
- 上下结合

## 典型片上总线结构

- AMBA
  - AHB先进高性能总线
  - ASB先进系统总线
  - APB先进外设总线
  - AXI
- Core Connect
- Avalon
- Wishbone
- OPC

其中OPC、WishBone是真正可以免费使用的，其它三种都是声称免费但是需要授权协议的

# 复杂总线结构

- 双总线结构
- 多总线结构

## 硬件描述语言VHDL设计★

### IEEE标准下

- VHDL
- Verilog HDL

优点：功能强大、通用性强，语法严谨

缺点：语法结构复杂

## 电子系统设计描述等级

- 行为级
- 寄存器传输级(RTL)
- 逻辑门级
- 版图级

## 程序结构

- 实体——描述外部接口信号
- 结构体——描述内部电路
- 配置——从库中选取实体
- 包/集合——存放数据类型、常数、子程序
- 库——存放已经编译的实体、结构体、包集合和配置

### 2输入与门逻辑描述

```
LIBRARY ieee;                                --库说明语句
USE ieee.std_logic_1164.all;                  --程序包说明语句
ENTITY and2 IS
  PORT(a,b    : IN  STD_LOGIC;
        y       : OUT STD_LOGIC);
END and2;
ARCHITECTURE and2x OF and2 IS
BEGIN
  y<=a AND b;
END and2x;
```

## 类属

确定实体或组件中定义的局部常数。模块化设计时多用于不同层次模块之间信息的传递。可从外部改变内部电路

结构和规模

类属说明必须放在端口说明之前。

类属常用于定义：

- 实体端口的大小、✓
- 设计实体的物理特性、✓
- 总线宽度、✓
- 元件例化的数量等。✓

-- 例：2输入与门的实体描述

```
entity and2 is
    generic(risewidth: time:= 1 ns;
             fallwidth: time:= 1 ns);
    port(a1: in std_logic;
          a0: in std_logic;
          z0: out std_logic);
end entity and2;
```

-- 注：数据类型 `time` 用于仿真模块的设计。  
-- 综合器仅支持数据类型为整数的类属值。

## 端口模式

`in`: 输入型，此端口为只读型。(只能被引用，不能被赋值)

`out`: 输出型，只能在实体内部对其赋值。

`inout`: 输入输出型，既可读也可赋值。

`buffer`: 缓冲型，与 `out` 相似，但可读。

## 数据类型

端口上流动的数据的表达格式。为预先定义好的数据类型。

如：`bit`、`bit_vector`、`integer`、  
`std_logic`、`std_logic_vector` 等。

## 构造体（结构体）architecture

注：同一实体的结构体不能同名。定义语句中的 常数、信号不能与实体中的端口同名。



## 调用模块

```

architecture bhv of entityname is
    --元件的声明
    component component_name is
        port(
            port_name: signal_mode signal_type;
            port_name: signal_mode signal_type;
            ...
        );
    end component;

    --元件实例化
    label1: component_name port map (port_list);

end bhv;

```

## 2选1选择器

```

library ieee;
use ieee.std_logic_1164.all;

entity mux21 is
    port(a, b: in std_logic;
         s: in std_logic;
         y: out std_logic);
end mux21;

architecture bhv of mux21 is
begin
    -- 直接是并行语句，因此直接放
    y <=
        a when s = '0' else
        b when s = '1';
end bhv;

```

## 程序包说明

```
-- 独立于architecture的
package example is
    type byte is range 0 to 255;
    subtype nibble is byte range 0 to 15;
    constant byte_ff: byte := 255;
    signal addend: nibble;

    component byte_adder
        port(
            a, b: in byte;
            c: out byte;
            overflow: out boolean
        );
    end component;

    function my_function(a: in byte)
        return byte;

end example;
```

## 例子数码管译码器

```
package seven is
    subtype segments is bit_vector(0 to 6);
    type bcd is range 0 to 9;
end seven;
library work;
use work.seven.all;
entity decoder is
    port(input: in bcd;
          drive: out segments);
end decoder;
architecture art of decoder is
begin
    -- 根据input的值赋值给drive值
    with input select
        drive<=B“1111110” when 0,
        B“0110000” when 1,
        B“1101101” when 2,
        B“1111001” when 3,
        B“0110011” when 4,
        B“1011011” when 5,
        B“1011111” when 6,
        B“1110000” when 7,
        B“1111111” when 8,
        B“1111011” when 9,
        B“0000000” when others;
end architecture art;
```

## configuration配置

### 格式

```
configuration 配置名 of 实体名 is
    for 选配结构体名
    end for ;
end 配置名;
```

例子：一个与非门不同实现方式的配置如下

```
library ieee;
use ieee.std_logic_1164.all;
entity nand is
    port(a: in std_logic;
         b: in std_logic;
         c: out std_logic);
end entity nand;

-- 结构体一
architecture art1 of nand is
begin
    c<=not (a and b);
end architecture art1;

-- 结构体二
architecture art2 of nand is
begin
    c<='1' when (a='0') and (b='0') else
        '1' when (a='0') and (b='1') else
        '1' when (a='1') and (b='0') else
        '0' when (a='1') and (b='1') else
        '0';
end architecture art2;

-- 配置
configuration first of nand is
    for art1;
    end for;
end first;
configuration second of nand is
    for art2
    end for;
end second;
```

---

## VHDL的语言元素

### 包括

- VHDL数据对象
- VHDL数据类型
- VHDL操作符

# vhdl数据对象

## 常量

const

不可被赋值

## 变量

variable 在结构体

被赋值立刻改变值, ":="赋值

variable只能定义在process和subprogram (包括function和procedure) 中, 不可定以在其外部

## 信号

signal

被赋值后不立即更新, 当进程被挂起后才更新, "<="赋值

signal不能定义在process和subprogram (包括function和procedure) 中, 只可定以在其外部, 即  
结构体Architecture开始部分

## 文件

file

实现对文件的写入和读出

## 信号和变量区别

①信号是全局量, 可以进行进程之间的通信;  
而变量是局部量, 只能用于进程或子程序(即  
函数和过程)中;

②对变量的赋值是立即发生的, 对信号的赋值  
须经一段时间延迟才会发生;

③对变量的赋值用“:=”, 对信号的赋值用“<=”;

④信号可以比拟为硬件端口之间的连接, 而变  
量则与硬件之间没有对应关系;

## 信号对象的常用属性

delayed[(时延值)]: 使信号产生固定时间的延时并返回  
stable[(时延值)]: 返回boolean, 信号在规定时间内没有变化返回true  
transaction: 返回bit类型, 信号每发生一次变化, 返回值翻转一次  
event  
last\_value

-----  
属性使用""访问, 如

```
if(clk'event and clk = '0' and clk'last_value = '1') then -- 从1到0的下降沿
```

## 初始化赋值

对信号、变量的赋初值是不可综合的, 仅能用在仿真中

## 数据类型

### 标准数据类型

integer  
real  
bit  
bit\_vector  
等等

STD\_LOGIC类型的数据可以具有九种取值 (均为大写字母)

'U': 初始值  
'X': 不定态  
'0': 强制0  
'1': 强制1  
'Z': 高阻态  
'W': 弱信号不定态  
'L': 弱信号0  
'H': 弱信号1  
'\_': 不可能情况 (可忽略值)

其中, 'X'方便了系统仿真, 'Z'方便了双向总线的描述。

STD\_LOGIC\_VECTOR类型定义如下:

```
TYPE STD_LOGIC_VECTOR IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC;
```

**注意: 赋值语句的方向应该和声明的方向一致**

### 用户自定义数据类型

```
type <数据类型名> is <数据类型> <约束范围>  
-- 1) 整数、实数类型  
type myreal is real range 1e-3 to 1e3;  
-- 2) 枚举类型(特别是在状态机中的定义)  
type color is (red, green, yellow);  
-- 3) 数组类型、记录类型
```

```

type matrix is array (0 to 63) of std_logic_vector(7 downto 0);
type matrix is array(integer range<>) of std_logic_vector(7 downto 0);
signal tmp: matrix(0 to 2**M - 1);

-- 4) 子类型
subtype my_in is integer range 0 to 65535;
-- 5) 时间类型
type time is range -1e18 to 1e18;

```

## VHDL操作符

/=是不等于  
&连接符

## vhdl基本逻辑语句

### 并行处理

- architecture语句以及其子模块之间
- 子模块block中的语句

block语句同样在architecture中使用, block内部的语句是并行的

### 顺序处理

- 子模块process中的语句
- 子模块subprogram中的function和procedure中的语句

## 2选1数据选择器

```

-- 用BLOCK语句描述2选1电路的程序。

ENTITY mux2_1 IS
PORT(d0, d1, sel      : IN      STD_LOGIC;
      q        : OUT      STD_LOGIC);
END mux2_1;

ARCHITECTURE amux OF mux2_1 IS
SIGNAL tmp1,tmp2,tmp3 : STD_LOGIC;
BEGIN

cale: BLOCK
BEGIN
tmp1<=d0 AND sel;
tmp2<=d1 AND (not sel);
tmp3<=tmp1 OR tmp2;
q<=tmp3;
END BLOCK cale;
END amux;

```

-- 上述结构体中只有一个 BLOCK块，若电路复杂时可由几个BLOCK块组成

用带保护条件的BLOCK语句描述一个锁存器的结构。

```
ENTITY latch IS
    PORT(d, clk : IN STD_LOGIC;
          q, qn : OUT STD_LOGIC);
END latch;
ARCHITECTURE latch_a OF latch IS
BEGIN
    g1:BLOCK(clk='1')
        BEGIN
            q<=guarded d after 5ns;
            qn<=guarded not(d) after 7ns;
        END BLOCK g1;
END latch_a;
```

在process中，其值被引用的信号应当出现在敏感信号列表中 ()

```
process(<敏感信号列表>)
```

### 进程 process 小结

- 敏感信号表所标明的信号是用来启动进程的
- PROCESS内部各语句之间是顺序关系。
- 各进程之间的关系是并行关系；进程之间的通信则一边通过接口由信号传递，一边并行地同步执行。

## 子程序

一般定义在在程序包、结构体和进程中定义，一般就定义在 architecture中的begin之前

### 过程 procedure

-- 设计一个从两个整数中求取最大值的过程。

```
PROCEDURE max(a, b:      IN INTEGER;
              y:      OUT INTEGER)  IS
BEGIN
    IF (a<b) THEN
        y<=b;
    ELSE
        y<=a;
    END IF;
END max;

-- 过程的调用
max ( x,  y,  maxout ) ;
```

## 函数 function

应该封装在包中

```
FUNCTION max(a:std_logic_vector; b:std_logic_vector) RETURN std_logic_vector IS
    VARIABLE tmp :std_logic_vector(a'range);
BEGIN
    IF (a>b) THEN
        tmp:=a;
    ELSE
        tmp:=b;
    END IF;
    RETURN tmp;
END max;
```

例子：在结构体中调用求最大值的函数。

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY dpeak IS
    PORT(clk, set : IN STD_LOGIC;
          date      : IN STD_LOGIC_VECTOR(5 downto 0);
          dout      : OUT STD_LOGIC_VECTOR(5 downto 0));
END dpeak;

ARCHITECTURE rtl OF dpeak IS
    SIGNAL peak : STD_LOGIC_VECTOR(5 downto 0);
BEGIN
    dout<=peak;
    PROCESS (clk)
    BEGIN
        IF (clk'event and clk='1') THEN
            IF (set='1') THEN
                peak<=date;
            ELSE
                -- 调用函数max
                peak<= max(date,peak);
            END IF;
        END IF;
    END PROCESS;
END rtl;
```

## 顺序执行语句

Wait语句  
If 语句  
case语句  
for loop语句  
while 语句

- wait

如果process中已有敏感信号进程中不能使用wait语句

- if

```
process (A, B, C, X)
begin
  if(x = "0000") then
    Z <= A;
  elsif (x <= "0101") then
    Z <= B
  else
    Z <= C;
  end if;
end process;
```

- case

- for loop

• For loop 语句格式

• For loop 语句例子

**For** 循环变量 **in** 范围 **loop**  
【顺序处理语句】  
**end loop**

```
For i in 1 to 10
loop
  sum=sum+1;
end loop
```

注意：循环变量不需要定义(声明);例子中 i 不需要定义

```
For i in 1 to 10 loop
  sum=sum+1;
end loop
-- i不需要定义(重点)
```

**next** 跳出本次循环  
**exit** 结束整个循环

- while loop

## 并行处理语句

- 信号赋值操作
- 带条件的信号赋值语句

```

<= when
else

block begin
sel<=b & a;
q<=ain when sel="00" else
bin when sel="01" else
cin when sel="10" else
din when sel='11' else
xx; -- 这个else不可省略

end block

```

- 带选择的信号赋值语句

```

with ... select
q <=
when ...;

block begin
with sel select
q<=ain when sel="00",
bin when sel="01",
cin when sel="10",
din when sel='11'
xx; when others;
end block

```

## vhdl描述方法

### 行为（级）描述

属于高级描述，只描述功能，不直接指明硬件结构

例：一位全加器设计

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY full_adder IS
  GENERIC (tpd : TIME := 10 ns);
  PORT (x, y, c_in : IN STD_LOGIC;
        Sum, c_out : OUT STD_LOGIC);
END full_adder;
ARCHITECTURE behav OF full_adder IS
BEGIN
  PROCESS (x, y, c_in)
  VARIABLE n: INTEGER;
  CONSTANT sum_vector: STD_LOGIC_VECTOR (0 TO 3) := "0101";
  CONSTANT carry_vector: STD_LOGIC_VECTOR (0 TO 3) := "0011";

  BEGIN
    n := 0;
    IF x = '1' THEN
      n := n+1;
    END IF;
    IF y = '1' THEN
      n := n+1;
    END IF;
    IF c_in = '1' THEN
      n := n+1;
    END IF;
    c_out := n&3;
    sum := n&2;
  END PROCESS;
END behav;

```

x y c<sub>in</sub> s c<sub>out</sub>

```

    END IF;
    IF y = '1' THEN
        n:=n+1;
    END IF;
    IF c_in = '1' THEN
        n:=n+1;
    END IF;
    sum <= sum_vector (n);      -- sum_vector初值为“0101”
    c_out <= carry_vector (n);  -- carry_vector初值为“0011”
END PROCESS;                      -- (0 TO 3)

END behav;

```

只能进行行为层次的仿真，不能进行逻辑仿真

## 数据流描述/寄存器传输 (RTL) 描述

真正可以进行逻辑综合的描述方式。

例：基于全加器布尔方程的数据流描述

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY ADDER1B IS
PORT(A, B, C_IN: IN BIT;
      SUM, C_OUT: OUT BIT);
END ADDER1B ;
ARCHITECTURE ART OF ADDER1B IS
SUM<= A XOR B XOR C_IN;
C_OUT<=(A AND B)OR (A AND C_IN) OR (B AND C_IN); ✓
END ART;

```

### 数据流描述注意问题

- 禁止在一个进程中存在两个边沿检测的寄存器描述；(即使是同一个寄存器也尽量不要出现，两个边沿检测)
- 禁止使用IF语句中的ELSE项；
- 寄存器描述中必须代入信号值

## 元件调用

- 元件说明

```

architecture bhv of entity_name is
component name is
    port();
end component;
begin
end bhv;

```

- 元件例化

### 端口映射

- 位置映射 (位置对应)

- 名称映射 (使用"=>"对应)

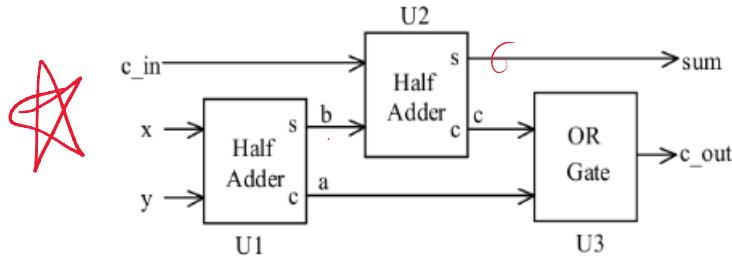
```
architecture bhv of entity_name is
begin
u1:name generic map() port map();
end bhv;
```

## 结构描述

结构描述方式通常采用元件例化语句和生成语句编写程序。

例：一位全加器的结构描述

- 全加器由两个半加器和一个或门组成的。



-- 半加器设计(RTL级描述)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY half_adder IS
  GENERIC (tpd: TIME:=10 ns);
  PORT (in1, in2: IN STD_LOGIC;
        sum, carry: OUT STD_LOGIC);
END half_adder;
ARCHITECTURE behavioral OF half_adder IS
BEGIN
  PROSESS (in1, in2)
  BEGIN
    sum <= in1 XOR in2;
    carry <= in1 AND in2;
  END PROSESS;
END behavioral;                                     -- 半加器设计完毕
```

-- 或门设计

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY or_gate IS
  GENERIC (tpd: TIME:=10 ns);
  PORT (in1, in2: IN STD_LOGIC;
        out1: OUT STD_LOGIC);
END or_gate;
ARCHITECTURE structural OF or_gate IS
BEGIN
  out1 <= in1 OR in2 AFTER tpd;
END structural;                                     -- 或门设计完毕
```

```

-- 1位全加器的结构描述程序
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY full_adder IS
  GENERIC (tpd: TIME:= 10 ns);
  PORT (x, y, c_in: IN STD_LOGIC;
        Sum, c_out: OUT STD_LOGIC);
END full_adder;

ARCHITECTURE structural OF full_adder IS

COMPONENT half_adder
  PORT (in1, in2: IN STD_LOGIC;
        sum, carry: OUT STD_LOGIC);
END COMPONENT;

COMPONENT or_gate
  PORT (in1, in2: IN STD_LOGIC;
        out1: OUT STD_LOGIC);
END COMPONENT;

SIGNAL a, b, c:STD_LOGIC;

BEGIN
  u1: half_adder PORT MAP (x, y, b, a);
  u2: half_adder PORT MAP (c_in, b, sum, c);
  u3: or_gate PORT MAP (c, a, c_out);
END structural;

```

描述方式	优点	缺点	适用场合
行为描述	电路特性清楚	综合效率低	大型复杂电路设计
数据流描述	布尔函数定义清楚	不易描述复杂电路, 修改困难	少量门数模块设计
结构描述	连接关系清晰, 电路模块化清晰	电路繁琐、复杂, 不易理解	电路层次化设计

## 逻辑电路设计

### 组合逻辑电路

输出仅仅与当前输入有关

- 无反馈延迟通路
- 无记忆元件

### 设计方法

- 基于数字电路的设计方法：利用真值表，画逻辑电路
  - 基于硬件描述语言的设计方法：vhdl实现
- 

## 时序逻辑电路(具有记忆元件、具有反馈通道)

结构：

- 组合电路
- 存储电路

任何一个时刻的输出状态不仅取决于当时的输入信号，还与电路的原状态有关。

---

## 分频电路(时钟)设计

分类

- 偶数分频器
  - 奇数分配器
  - 半整数分频器
  - 小数分频器
  - 分数分配器
  - 锁相环电路(PLL、 DLL)
- 

## 有限状态机设计 (FSM)

### FSM的程序结构 (由哪些三部分组成)

- 主控时序逻辑部分

负责状态机运转和在外部时钟驱动下实现内部状态转换的过程

- 主控组合逻辑部分

根据状态机外部输入的状态控制信号和当前状态确定下一个状态的取值内容，以及外部或内部其它进程输出控制信号的内容

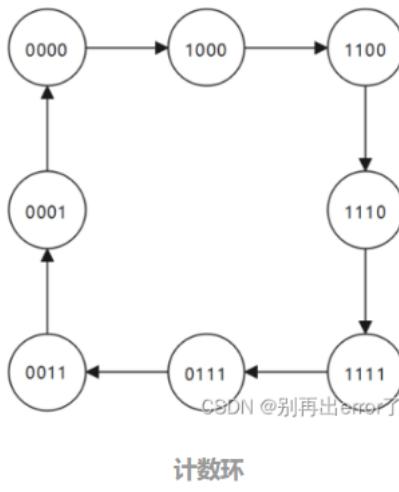
① ② ③

- 辅助逻辑部分

配合状态机主控时序逻辑和主控组合逻辑进行工作，完善和提高系统性能

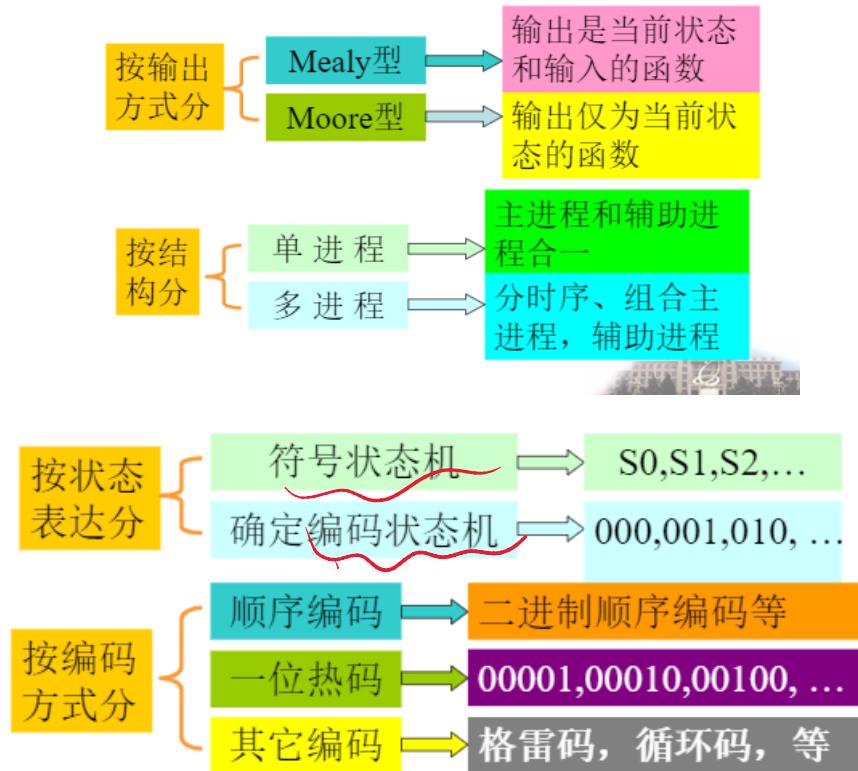
## 状态机编码

- 二进制
- 独热码：每一位对应一个触发器，用N位编码表示N个状态
- 格雷码：相邻两编码仅一位不同，用N位编码表示 $2^N$ 个状态（可以镜像生成）
- 约翰逊码：相邻两编码仅一位不同，用N位编码表示 $2N$ 个状态



## [状态机编码举例](#)

### 状态机分类



例：有限状态机

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY two_process_state_machine IS
    PORT (clk, reset : IN STD_LOGIC;
          state_inputs : IN STD_LOGIC;
          comb_outputs : OUT STD_LOGIC_VECTOR(0 TO 1));
END ENTITY two_process_state_machine;
ARCHITECTURE behv OF two_process_state_machine IS
    TYPE states IS (st0,st1,st2,st3); --定义 states 为枚举型数据类型，构造符号化状态机
    SIGNAL current_state, next_state: states;
BEGIN
    REG: PROCESS (reset, clk)      --时序逻辑进程
    BEGIN
        IF reset = '1' THEN      --异步复位
            current_state <= st0;
        ELSIF clk = '1' AND clk'EVENT THEN--出现时钟上升沿时进行状态转换
            current_state <= next_state;
        END IF;
    END PROCESS;

```

```

COM: PROCESS(current_state, state_inputs) --组合逻辑进程
    BEGIN
        CASE current_state IS
            WHEN st0 => comb_outputs <= "00"; --系统输出及其初始化
                IF state_inputs = '0' THEN --根据外部输入条件决定状态转换方向
                    next_state <= st0;
                ELSE next_state <= st1;
                END IF;
            WHEN st1=> comb_outputs <= "01";
                IF state_inputs = '0' THEN      next_state <= st1;
                ELSE next_state <= st2;
                END IF;
            WHEN st2=> comb_outputs <= "10";
                IF state_inputs = '0' THEN      next_state <= st2;
                ELSE next_state <= st3;
                END IF;
            WHEN st3=>comb_outputs <= "11";
                IF state_inputs = '0' THEN      next_state <= st3;
                ELSE next_state <= st0;
                END IF;
        END CASE;
    END PROCESS;

```

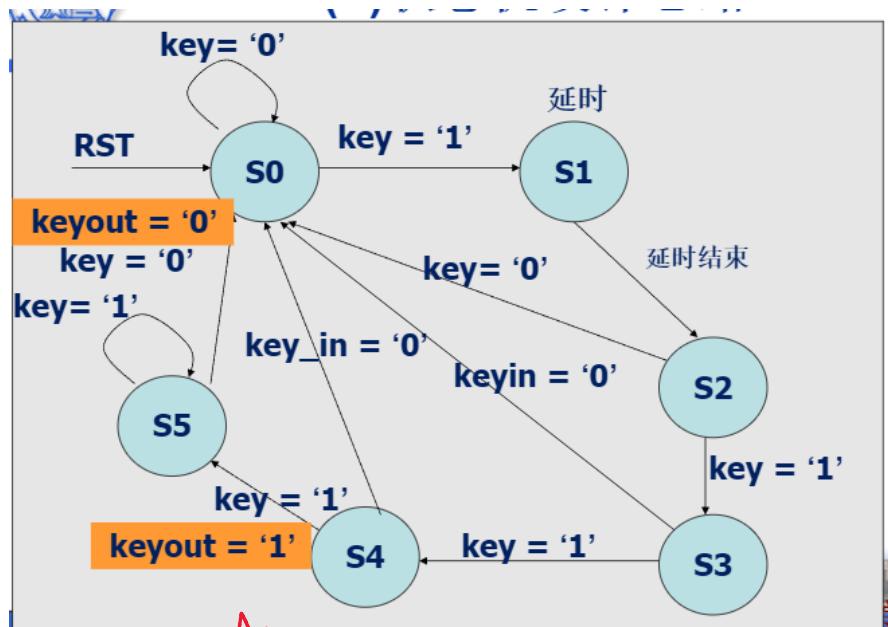
## moore型

输出只与当前状态有关，而与输入信号的当前值无关，是严格的现态函数

## mealy型

输出是现在状态和所有输入的函数，输出随输入变化而随时发生变化，属于异步输出的状态机。

## 消抖



注意是检测到变化 + 延时20ms + 连续三次检测确认，并且改图只能处理0到1的消抖

## 加法器设计★

### 定点加法器

### 串行进位加法器

进位信号采用串联结构，所用元件最少，逻辑电路比较简单，但运算时间比较长。

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity FAdder1 is
    Port ( p : in STD_LOGIC;
           q : in STD_LOGIC;
           c_in : in STD_LOGIC;
           s : out STD_LOGIC;
           c_out : out STD_LOGIC);
end FAdder1;

architecture Behavioral of FAdder1 is

begin
    s<=(p xor q) xor c_in;
    c_out<=(p and q) or (c_in and p) or (c_in and q);
end Behavioral;
```

```

        u0:FAdder1 port map(a(0),b(0),cin,sum(0),c(1));
        u1:FAdder1 port map(a(1),b(1),c(1),sum(1),c(2));
        u2:FAdder1 port map(a(2),b(2),c(2),sum(2),c(3));
        u3:FAdder1 port map(a(3),b(3),c(3),sum(3),c(4));
        u4:FAdder1 port map(a(4),b(4),c(4),sum(4),c(5));
        u5:FAdder1 port map(a(5),b(5),c(5),sum(5),c(6));
        u6:FAdder1 port map(a(6),b(6),c(6),sum(6),c(7));
        u7:FAdder1 port map(a(7),b(7),c(7),sum(7),cout);

```

## 并行进位加法器（超前进位加法器）

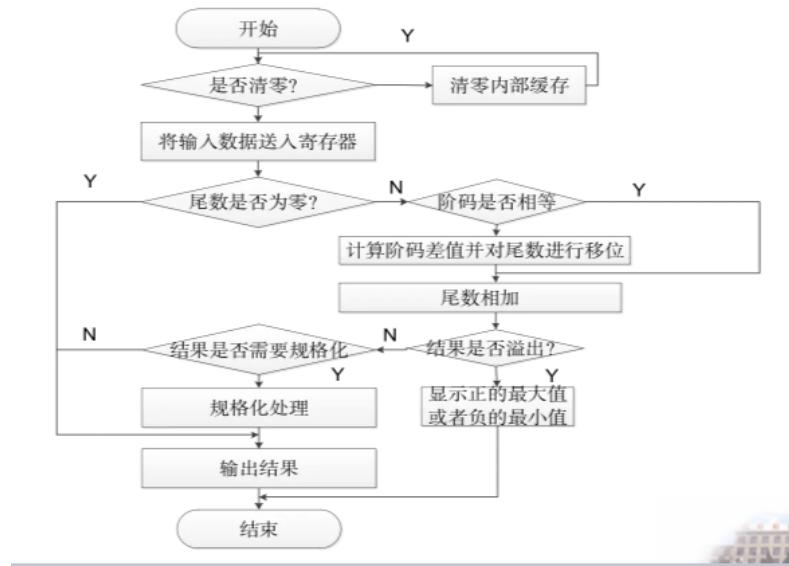
```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity ParAdder is
    Port ( a : in STD_LOGIC_VECTOR(3 DOWNTO 0);
            b : in STD_LOGIC_VECTOR(3 DOWNTO 0);
            cin : in STD_LOGIC;
            s : out STD_LOGIC_VECTOR(3 DOWNTO 0);
            cout : out STD_LOGIC;
            pp : out STD_LOGIC;
            gg : out STD_LOGIC);
end ParAdder;
architecture Behavioral of ParAdder is
signal c: std_logic_vector(3 downto 1);
signal p,g:std_logic_vector(3 downto 0);
begin
--绝对进位
    g(0)<=a(0) and b(0);
    g(1)<=a(1) and b(1);
    g(2)<=a(2) and b(2);
    g(3)<=a(3) and b(3);
--进位传递条件
    p(0)<=a(0) xor b(0);
    p(1)<=a(1) xor b(1);
    p(2)<=a(2) xor b(2);
    p(3)<=a(3) xor b(3);
--产生并行进位
    c(1)<=g(0) or (p(0) and cin);
    c(2)<=g(1) or (p(1) and g(0)) or (p(1)and p(0) and cin);
    c(3)<=g(2) or (p(2) and g(1)) or (p(2)and p(1) and g(0)) or (p(2)and p(1)and p(0)and cin);
    cout<=g(3)or(p(3) and g(2))or(p(3)and p(2) and g(1))or(p(3)and p(2)and p(1) and g(0));
                and p(1) and g(0)) or(p(3)and p(2)and p(1)and p(0) and cin );
--本组进位传递条件
    pp<=p(0) and p(1) and p(2) and p(3);
    gg<=g(3) or (p(3) and g(2)) or (p(3)and p(2) and g(1)) or (p(3)and p(2) and p(1) and g(0));
--和输出
    s(0)<=p(0) xor cin;
    s(1)<=p(1) xor c(1);
    s(2)<=p(2) xor c(2);
    s(3)<=p(3) xor c(3);
end Behavioral;

```

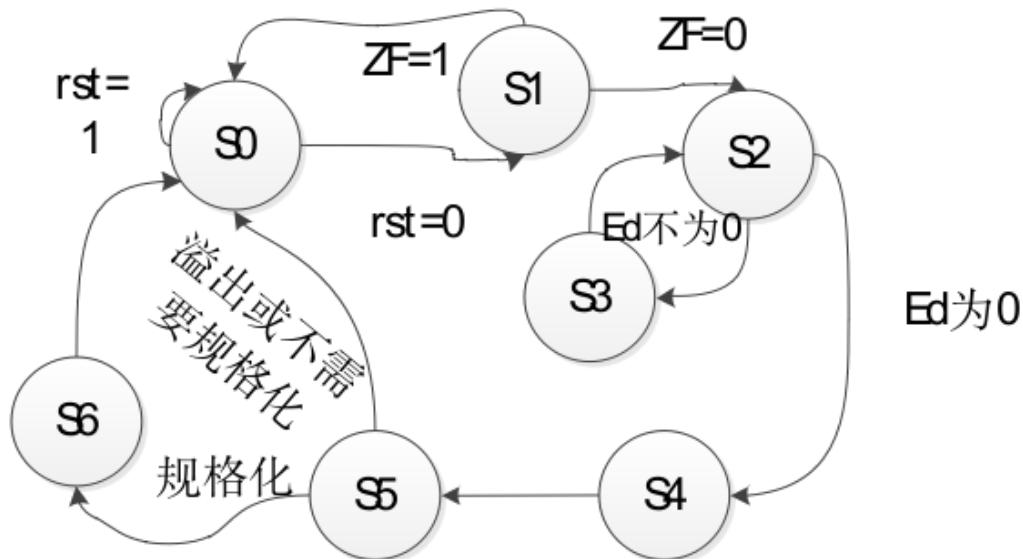
# 浮点加法器

## 工作流程



(1) 对阶 (2) 尾数进行加(减)运算 (3) 规格化 (4) 舍入处理

## 状态机描述工作流程



可以用FSM实现

## 乘除法器设计 (定点) ★

常用机器数编码格式 (含符号位的)

- 原码——实现乘除法容易，实现加减法困难
- 反码——
- 补码——负数（不含符号）将原码按位取反，末位加一

定点数

- 纯整数
- 纯小数

## 机器数编码格式

- 原码
- 反码
- 补码

## 定点乘法器

实现

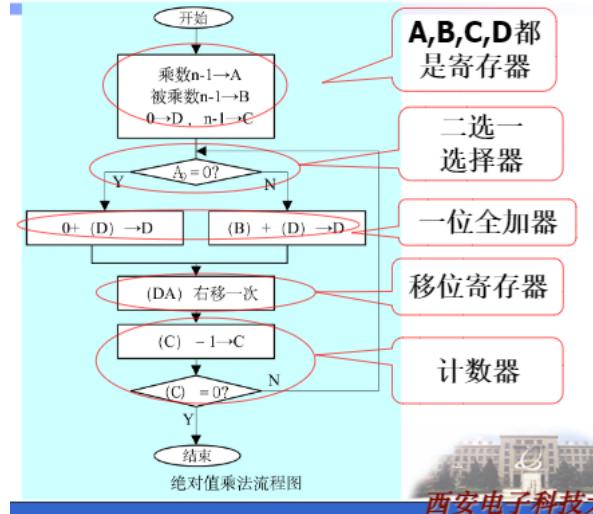
- 软件实现乘法
- 加法器基础上增加硬件实现乘法
- 设置专用硬件乘法器

## 原码一位乘法器

- 符号异或
- 乘积绝对值位绝对值之积

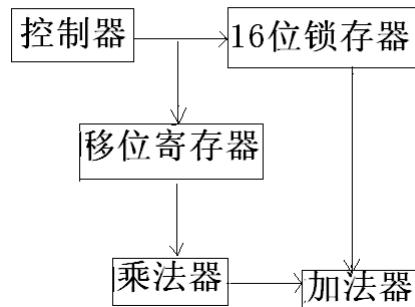
框图中符号位不参与

## 原码一位乘法原理（流程图）及实现



## 原码一位乘法器功能模块（模拟手工乘法过程）

- 控制器
- 16位锁存器
- 移位寄存器
- 乘法器
- 加法器



## 控制器设计

-- 控制16位锁存器和移位寄存器

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY ARICTL IS
    PORT (CLK, START : IN STD_LOGIC;
          CLKOUT,RSTALL,DONE: OUT STD_LOGIC );
END ARICTL;
ARCHITECTURE behav OF ARICTL IS
    SIGNAL CNT4B : STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
    PROCESS(CLK, START)
    BEGIN
        RSTALL <= START;
        IF START = '1' THEN CNT4B <= "0000";
        ELSIF CLK'EVENT AND CLK ='1' THEN
            IF CNT4B < 8 THEN CNT4B <= CNT4B + 1; END IF;
        END IF;
    END PROCESS;
    PROCESS(CLK, CNT4B, START)
    BEGIN
        IF START = '0' THEN
            IF CNT4B < 8 THEN CLKOUT <= CLK;
            ELSE CLKOUT <= '0'; DONE<='1'; END IF;
        ELSE CLKOUT <= CLK;  DONE<='0';END IF;
    END PROCESS;
END behav;

```

## 16位锁存器

存储部分积以及部分积移位

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY REG16B IS
    PORT (
        CLK : IN STD_LOGIC;
        CLR : IN STD_LOGIC;
        D : IN STD_LOGIC_VECTOR(8 DOWNTO 0);
        Q : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
    );
END REG16B;
ARCHITECTURE behav OF REG16B IS
    SIGNAL R16S : STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
    PROCESS(CLK, CLR)
    BEGIN
        IF CLR = '1' THEN                                -- 清零信号
            R16S <= "0000000000000000";-- 时钟到来时，锁存输入值，并右移低8位
        ELSIF CLK'EVENT AND CLK = '1' THEN
            R16S(6 DOWNTO 0) <= R16S(7 DOWNTO 1); -- 右移低8位
            R16S(15 DOWNTO 7) <= D;                  -- 将输入锁到高8位
        END IF;
    END PROCESS;
    Q <= R16S;
END behav;

```

## 移位寄存器

功能就是右移一位，同时输出最低数据位用于乘法运算

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY SREG8B IS
    PORT ( CLK : IN STD_LOGIC;    LOAD : IN STD_LOGIC;
            DIN : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
            QB : OUT STD_LOGIC  );
END SREG8B;
ARCHITECTURE behav OF SREG8B IS
    SIGNAL REG8 : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
    PROCESS (CLK, LOAD)
    BEGIN
        IF CLK'EVENT AND CLK = '1' THEN
            IF LOAD = '1' THEN                                -- 装载新数据
                REG8 <= DIN;
            ELSE
                REG8(6 DOWNTO 0) <= REG8(7 DOWNTO 1);      -- 数据右移
            END IF;
        END IF;
    END PROCESS;
    QB <= REG8(0);                                      -- 输出最低位
END behav;

```

## 1位乘法器

当前数据位与另一个操作数进行与运算

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY ANDARITH IS
    PORT ( ABIN : IN STD_LOGIC;
           DIN : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
           DOUT : OUT STD_LOGIC_VECTOR(7 DOWNTO 0) );
END ANDARITH;
ARCHITECTURE behav OF ANDARITH IS
BEGIN
    PROCESS(ABIN, DIN)
    BEGIN
        FOR I IN 0 TO 7 LOOP          -- 循环, 完成8位与1位运算
            DOUT(I) <= DIN(I) AND ABIN;
        END LOOP;
    END PROCESS;
END behav;

```

## 加法器设计

如八位并行加法器

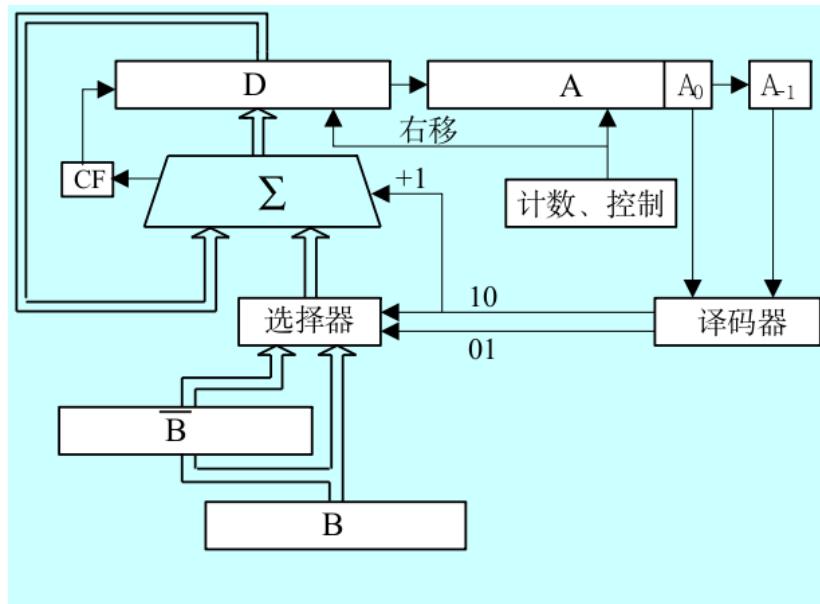
## 原码二位乘法

### 速度提高一倍

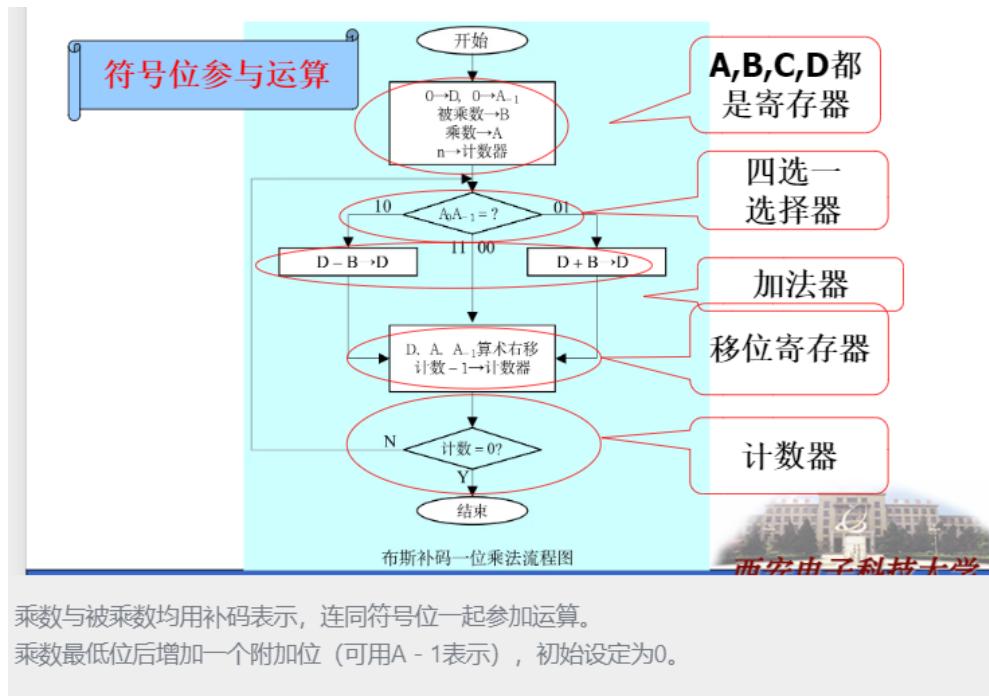
- $Y_{i+1} = Y_i = C$ 时——+0, 右移2次,  $C=0$
- $Y_{i+1} = 0 \&& Y_i \oplus C = 1$ ——+|X|, 右移2位,  $C=0$
- $Y_{i+1} \oplus Y_i = 1 \&& Y_i = C$ —— $-2|X|$ , 右移2位,  $C=0$
- $Y_{i+1} = 1 \&& Y_i \oplus C = 1$ —— $-|X|$ , 右移2位,  $C=0$

## 补码一位乘法 (布斯乘法)

符号位参与运算, 并且用补码表示纯小数



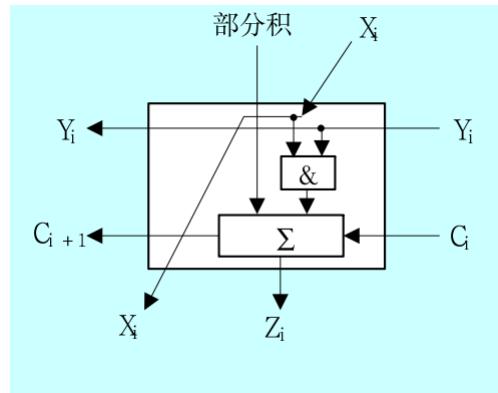
## 布斯补码一位乘法运算



## 阵列乘法器

### 基本乘加单元

- 二输入与门（乘法）
- 一位全加器



### 定点无符号阵列乘法器

TOP\_ROW(只有乘，没有加)

MID\_ROW(乘加单元)

LOW\_ROW(只有加，没有乘)

## 定点除法器

### 分类

- 原码除法
- 补码除法

# 原码恢复余数除法

## 符号位不参与运算

先减后判，如果减后发现不够减，则上商0，并加上除数

# 原码加减交替除法器（不恢复余数法）

## 符号位不参与运算，采用双符号位

### 余数符号应该和商一致

# 补码除法器

## 符号位参与运算，双符号位

- ①符号判断。被除数和除数同号，被除数减除数；若异号则加除数。
- ②余数与除数同号，上商为1，余数左移1位，下次用余数减除数操作求商。若异号，上商为0，余数左移1位，下次用余数加除数操作求商。
- ③重复②直至除尽或达到精度要求。
- ④商修正。在除不尽时，最低位恒置1修正

```
architecture Behavioral of div8_c is
constant itera: integer :=7; --累加移位次数
begin
    process(clk,rst)
        variable Q_temp:std_logic_vector(7 downto 0);--商
        variable count :integer range 0 to 7;--累加和移位计数器
        variable R_temp,B:std_logic_vector(8 downto 0);--余数和除数，双符号位，运算开始时余数即为被除数
    begin
        if(rst='1') then
            Q_temp(7 downto 0):="00000000";
            R_temp(8 downto 0):=oper_a(7)&oper_a(7 downto 0);
            B(8 downto 0):=oper_b(7)&oper_b(7 downto 0);
            count:=itera;
            done<='0';
        elsif(clk='1' and clk'event) then
            if(count>0) then
                if(R_temp(8)=B(8))then Q_temp(0):='1';
                else Q_temp(0):='0';
                end if;
                Q_temp(7 downto 0):=Q_temp(6 downto 0)&'0';
                R_temp(8 downto 0):=R_temp(7 downto 0)&'0';
                if(R_temp(8)=B(8))then R_temp:=R_temp-B;
                else R_temp:=R_temp+B;
                end if;
                count:=count-1;
            end if;
            elsif(count=0) then
                Q_temp(7 downto 0):=(not Q_temp(7)) &Q_temp(6 downto 1)&'1';
                done<='1';
                count:=itera;
            end if;
            Q<=Q_temp;
            R<=R_temp(7 downto 0);
        end if;
    end process;
end Behavioral;
```

复位赋初值

判断余数与除数  
符号是否相同

运算结束

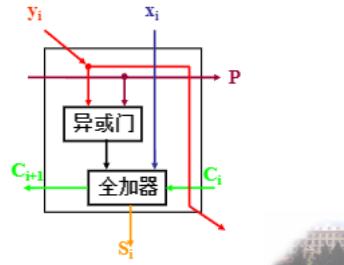
如何知道是除法，看复位赋初值

## 阵列除法器

### 可控加/减法单元

- 异或门（除法）
- 一位全加器

当  $P = 0$  时，做加法；  
当  $P = 1$  时，做减法，变  $+Y^*$  为  $+[ -Y^*]$  补



## 存储器设计

### SRAM设计

#### 功能分析

- 数据存储功能
- 地址控制功能
- 写入和读出功能

```
port(
    address: in std_logic_vector(3 downto 0);
    cs: in std_logic;
    oe, we: in std_logic;
    data: inout std_logic_vector(7 downto 0) -- 双向端口
    -- 注意三态（高阻态"ZZZZZZ"）
);
```

字扩展（宽度），位扩展（深度）

### ROM设计

只需要设置数据输出端口和地址输入端口

#### 电路结构：

- 存储矩阵
- 地址译码器
- 输出缓冲器

见实验

# 双端口RAM

允许两个独立的系统同时对其进行随机性访问的存储器（共享式多端口存储器）  
同时，随机访问

双端口RAM最大的特点是存储数据共享，并且必须具有访问仲裁控制。

## 方案

- 通用集成电路组成的双端口——容量大，经济
- 专用双端口RAM芯片——通信实时性要求高

# 先进先出存储器(FIFO)设计

## 特点

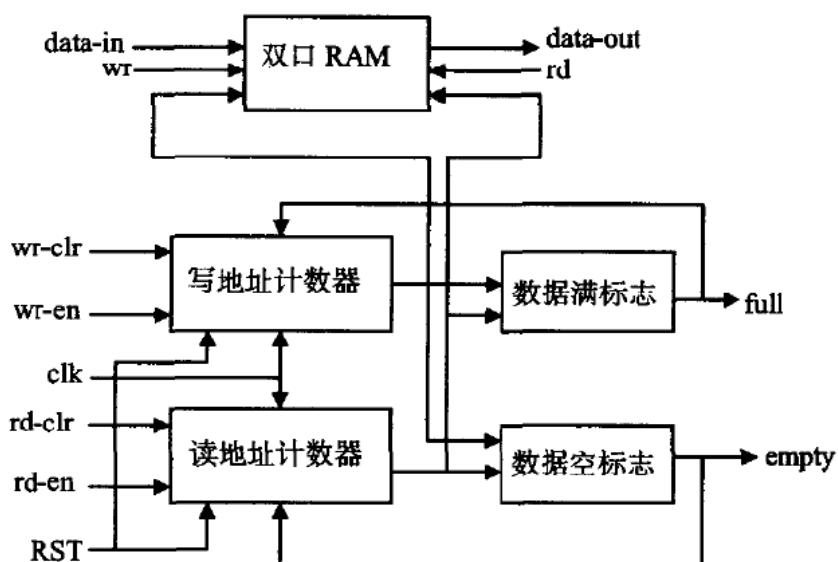
- 没有外部读写地址线，其数据地址由内部读写指针自动加减1完成

## 类型

- 同步控制的FIFO——FIFO的读写时钟相同
- 异步控制的FIFO——FIFO的读写时钟不同，用于跨时钟域的数据交换，读写时钟之间不一定存在相位、周期方面的约束

## 同步FIFO结构

- 双端口RAM
- 写地址计数器
- 读地址计数器
- 数据空标志**
- 数据满标志**



FIFO由5个模块构成，它们是双口RAM、写地址计数器、读地址计数器、数据满标志、数据空标志。

## 同步FIFO设计（环形）

FIFO存储器是一个环形结构，读地址计数器和写地址计数器分别代表下一次读数据操作时的读指针和下一次写数据操作时的写指针。

### 如何判断/产生空满标志位

M为FIFO存储器的容量，也即数据深度

当 $wr\_ptr=rd\_ptr$ 时，FIFO数据为空；

当 $wr\_ptr-rd\_ptr=M-1$ 或 $rd\_ptr-wr\_ptr=1$ 时，FIFO数据为满；

当 $wr\_ptr \geq rd\_ptr$ 时， $wr\_ptr-rd\_ptr$ 为FIFO内数据个数；

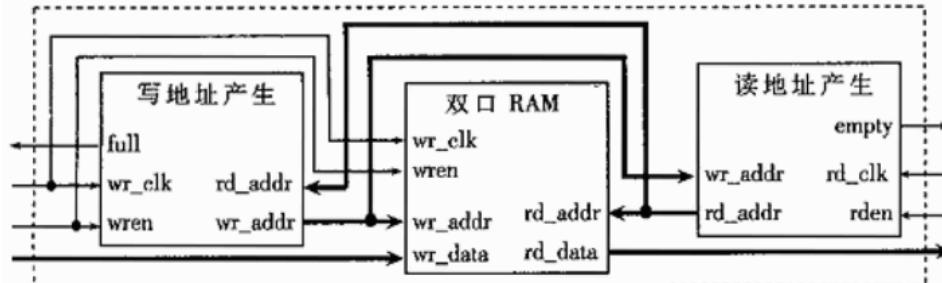
当 $wr\_ptr \leq rd\_ptr$ 时， $M-(rd\_ptr-wr\_ptr)$ 为FIFO内数据个数。

## 异步FIFO设计

- 读写时钟信号不相同
- 通过地址编码方式解决读写地址变化不同步而引起的空满标志错误的问题。

### 模块结构

- 双端口存储器RAM
- 写地址产生模块
- 读地址产生模块



### 如何判断/产生空满标志位 ❤️

异步FIFO存储是如何判断/产生空满标志位的？

ChatGPT

异步FIFO存储器通常通过监视其读取和写入指针的状态来确定存储器是否为空或已满。这些指针指示了存储器中的数据位置。以下是一些常见的方法：

空标志位（Empty Flag）：当读取指针（或称为取数据指针）等于写入指针（或称为放数据指针）时，FIFO存储器为空。因此，当两个指针相等时，空标志位被设置为逻辑“1”。

满标志位（Full Flag）：当写入指针比读取指针早一个位置时，FIFO存储器是满的。这意味着在下一次写入数据之前，必须先读取数据。因此，当写入指针比读取指针早一个位置时，满标志位被设置为逻辑“1”。

这些标志位通常由FIFO存储器的控制逻辑生成，并且可以由外部电路或微处理器读取以进行适当的操作，如等待或中断处理。

标志的产生是通过对读写地址的比较产生的，如果读取指针等于写入指针，FIFO为空；如果读取指针比写入指针小一个位置或者写入指针比读取指针小一个位置，FIFO为满。

当读写时钟完全异步时，对读写地址进行比较时，可能得出错误的结果，因此需要通过地址编码方式解决因读写地址变化不同步而引起的空满标志错误问题

## 校验电路设计

### 数据检测技术

- 奇偶校验
- 和校验
- 循环冗余码校验CRC

### CRC校验检测程序

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
entity crcn is
    port(clk,hrecv:in std_logic; --hrecv 握手信号-->hrecv检测模块接受,clk 时钟触发信号
          datacrci:in std_logic_vector(16 downto 0);--crc校验生成模块==>data(12位数据)+crc(5位校验码)
          rdata:out std_logic_vector(11 downto 0);--提取12位有效数据
          datafini:out std_logic;--flag数据接受校验完成状态显示位-->LED
          ERROR0:out std_logic);--ERROR0 误码警告信号-->beep;--hsend 握手信号(crc校验生成模块发送使用---->hrecv)
end crcn;
architecture comm of crcn is
    constant multi_coef: std_logic_vector(5 downto 0):="110101";--多项式系数，MSB一定为'1'
    signal cnt,rcnt:std_logic_vector(4 downto 0);           --用于计数
    signal dtemp,sdatam,rdtemp:std_logic_vector(11 downto 0);
    signal rdatacrc:std_logic_vector(16 downto 0);
    signal st,rt:std_logic; --st,rt状态转移标志位
begin
process(hrecv,clk)
    variable rcrcvar: std_logic_vector(5 downto 0);

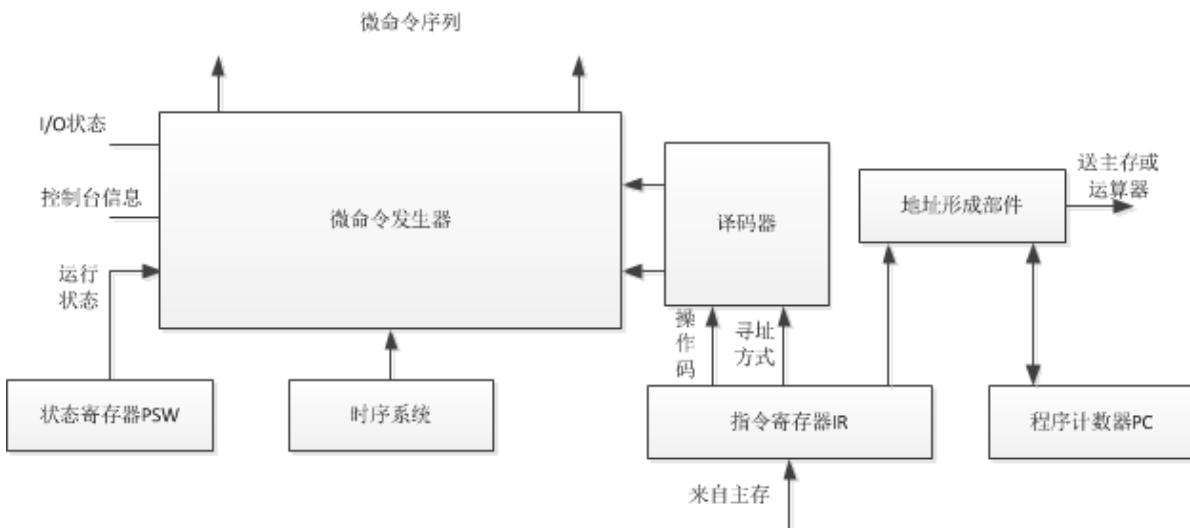
begin
process(hrecv,clk)
    variable rcrcvar: std_logic_vector(5 downto 0);
begin
    if(clk'event and clk='1') then
        if(rt='0' and hrecv='1') then
            if(rdttemp<=datacrci(16 downto 5));
            rdatacrc<=datacrci;
            rcnt<=(others=>'0');
            ERROR0<='0';
            rt<='1';
        elsif(rt='1' and rcnt<=7) then
            datafini<='0';
            rcnt<=rcnt+1;
            rcrcvar:= rdttemp(11 downto 6)xor multi_coef;
            if(rdttemp(11)='1') then
                rdttemp<=rcrcvar(4 downto 0) & rdtemp(5 downto 0) & '0';
            else rdttemp<=rdtemp(10 downto 0)&'0';
            end if;
        elsif(rt='1' and rcnt=7) then datafini<='1';
            rdata<=rdatacrc(16 downto 5);
            rt<='0';
            if(rdatacrc(4 downto 0) /=rdtemp(11 downto 7)) then
                ERROR0<='1';
            end if;
        end if;
    end if;
end process;
end comm;
```

# SoC系统设计实现 ☆

## CPU组成结构

控制部件和运算部件两部分

### 控制部件



类型：

- 硬布线控制器

控制部件看做产生专门固定时序控制信号的逻辑电路，以使用最少的元件和取得最高操作速度作为设计目标。

- 微程序控制器

将机器指令的操作（从取指令到执行）分解为若干更基本的微操作序列，并将有关的控制信息（微命令）以微码形式编成微指令输入控制存储器中。基于微程序控制原理

### 运算部件

其最重要的功能是执行算术和逻辑运算

#### 结构 (三部分)

- 输入逻辑
- 算术/逻辑运算单元
- 输出逻辑

### 寄存器组

五种类型

- 指令寄存器 IR
- 程序计数器 PC
- 存储器数据缓冲寄存器 MBR
- 存储器地址寄存器 MAR
- 程序状态字寄存器 PSW

# BMOSCPU分析

基于硬布线控制器设计

18个月

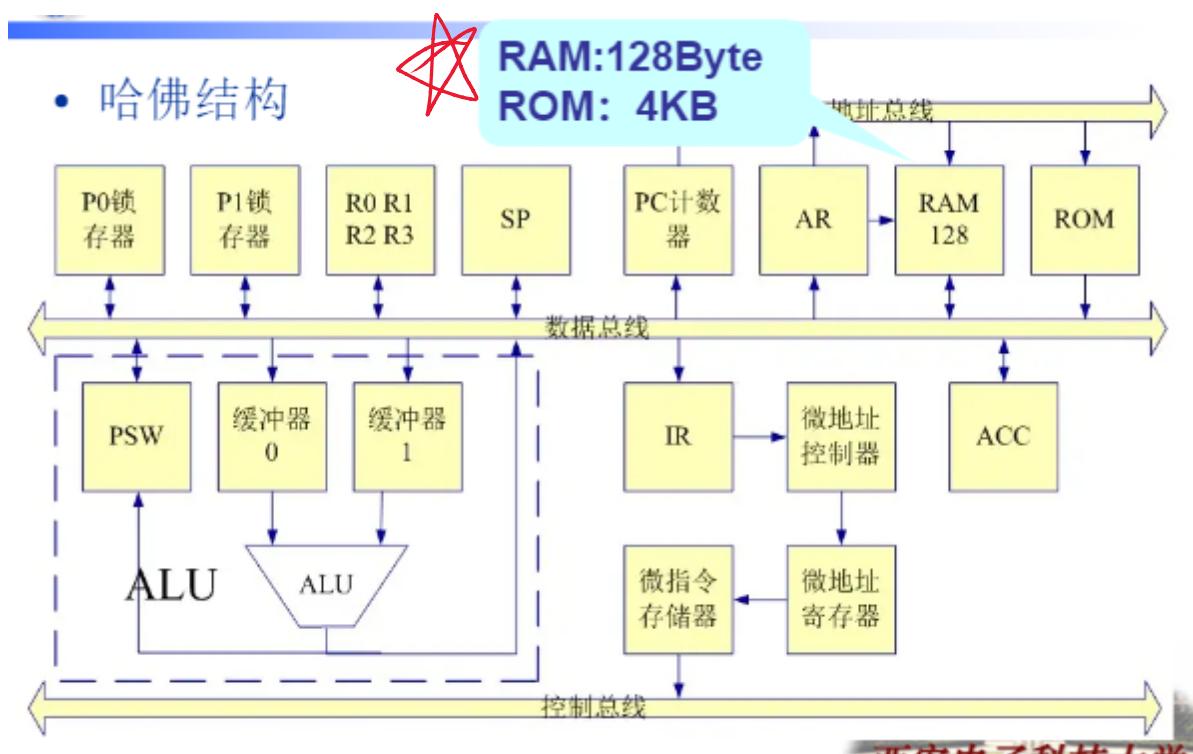
## Richard CPU分析

- Mealy状态机设计思路（输出与输入和状态有关）

输出由当前状态state和输入信号t0-t7决定

- 指令执行最多包含8个CPU周期（节拍周期），因此指令周期为8T，每个CPU周期为一个节拍t0, t1, ..., t7
- 哈佛结构
- 四种类型指令（传送类、逻辑运算类、算术运算类、调用及跳转类），共27条指令
- 微控制器将每条指令分解为若干微操作，为了提高执行效率，在时钟上升和下降边沿都有执行动作

## 设计CPU ❤



## 指令集

### 四种类型指令

- 传送类型
- 逻辑运算
- 算术运算

- 调用及跳转指令

## 寻址方式

- 立即寻址
- 直接寻址
- 寄存器直接寻址

## 设计思路:

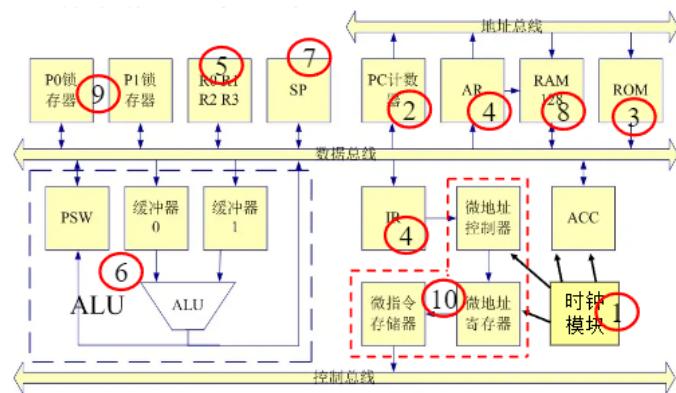
- 模块化设计方法 (分析模块功能、模块间信号传递) —— 结构描述方法
- 时钟控制信号
- 微程序控制方式

结构化描述方法, Mealy状态机

## 指令周期确定

定长指令周期, 一条指令8个CPU周期 (时钟周期), 一个节拍包含2个时钟周期

## 模块设计



### 时钟模块

### 程序计数器PC模块

### 程序存储器ROM模块

### 指令存储器IR模块

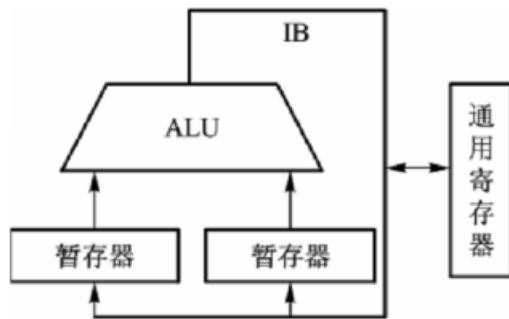
IR不作译码操作, 仅暂存数据。

### 寄存器组RNmok

### ALU模块

ALU三种基本结构

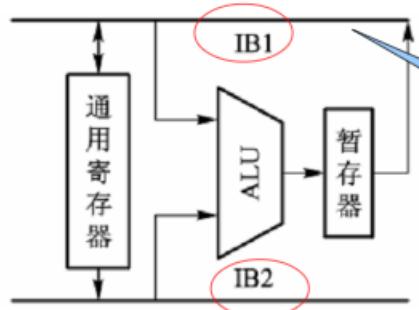
- 单总线结构



1. 所有部件都接到同一总线上，数据可以在任何两个寄存器之间，或者在任一个寄存器和ALU之间传送。
2. 在同一时间内，只能有一个操作数放在总线上进行传输
3. 需要分两次才能将两个操作数输入到ALU，并且需要A、B两个缓冲寄存器。

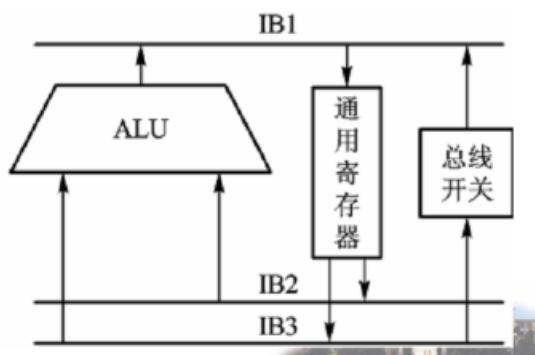
ALU运算结果是否可以直接送到总线上

- 双总线结构



1. 两个操作数同时加到ALU进行运算，只需一次操作控制，可得到运算结果。
2. ALU的输出不能直接加到总线上去，必须在ALU输出端设置缓冲寄存器。

- 三总线结构



1. ALU两个输入端分别连接两条总线，ALU的输出与第三条总线相连。
2. 附加直接传送功能：当一个操作数不需要修改，可通过总线开关将数据从输入总线直接传送到输出总线。

## 8位ALU设计

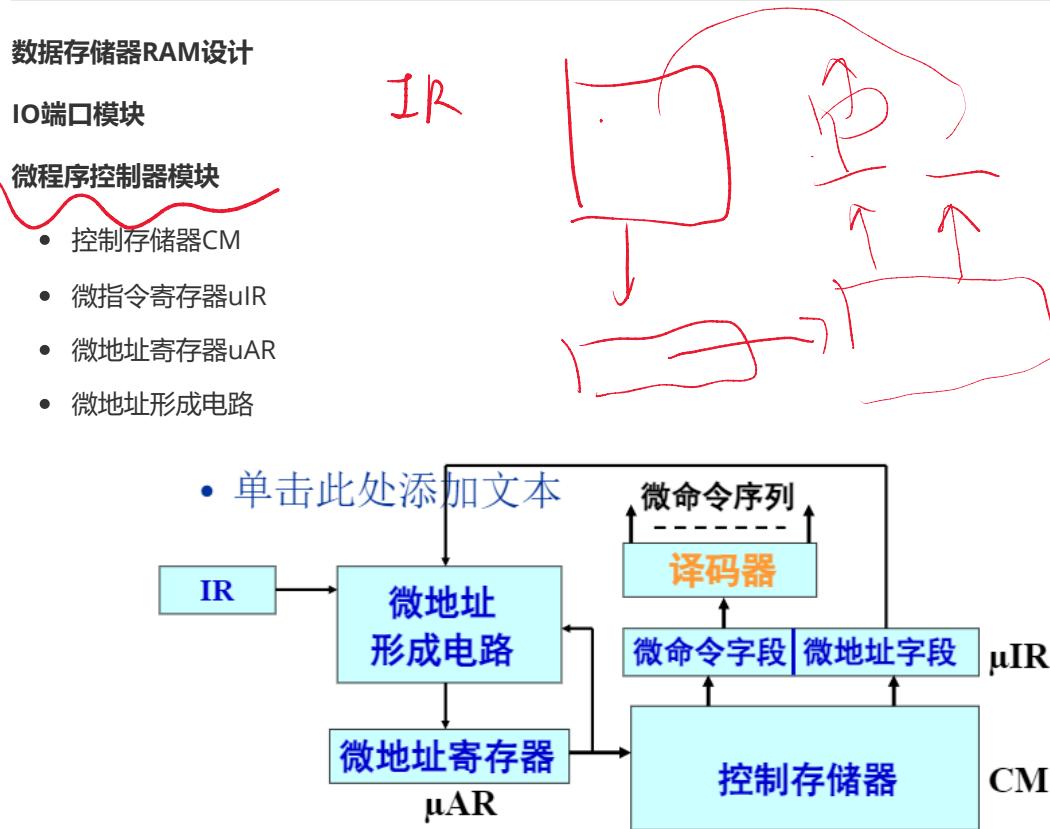
- 基于器件的8位ALU设计

暂存器74273 两个

算术逻辑运算单元74181

移位寄存器

- 基于VHDL的8位ALU设计



## ISA设计

### 微指令编码

- 直接控制编码
- 字段直接编译法
- 分段间接编译法

### 微指令格式

- 垂直型微指令——一条微指令执行一种基本操作
- 水平型微指令——一条微指令执行几种并行的基本操作

选择直接控制编码，增量方式产生后继地址，水平型微指令

- 39条控制信号
- 共27条指令——>八位微地址

因此微指令长度为48位（其中第12位是保留位）

# SoC系统测试与分析★

## SoC系统验证方法

分别进行功能验证和性能验证

验证SoC功能块的方法：

- 硬件建模
- 接口验证
- 软硬件协同验证与仿真
- 硬件仿真
- 系统级时序验证
- 物理验证

## SoC系统测试原理及方法

### 方法

- 基于神经网络的电路测试生成方法
- 二元判定图BDD
- 大规模集成电路与系统的可测性设计
- VLSI与系统的可测性设计
- JTAG接口（联合测试行动小组）



## JTAG边界扫描测试

引脚	描述	功能
TDI	测试数据输入 (Test Data Input)	测试指令和编程数据的串行输入引脚。数据在TCK的上升沿移入。
TDO	测试数据输出 (Test Data Output)	测试指令和编程数据的串行输出引脚，数据在TCK的下降沿移出。如果数据没有被移出时，该引脚处于高阻态。
TMS	测试模式选择 (Test Mode Select)	控制信号输入引脚，负责TAP控制器的转换。TMS必须在TCK的上升沿到来之前稳定。
TCK	测试时钟输入 (Test Clock Input)	时钟输入到BST电路，一些操作发生在上升沿，而另一些发生在下降沿。
TRST	测试复位输入 (Test Reset Input)	低电平有效，异步复位边界扫描电路（在IEEE规范中，该引脚可选）。

JTAG (Joint Test Action Group) 接口是一种用于测试和调试集成电路的标准接口。它通常被用于测试芯片上的逻辑电路、验证电路的功能和进行调试。

- TCK 为测试时钟输入
- TDI 为测试数据输入
- TDO 为测试数据输出
- TMS 为测试模式选择
- TRST 为测试复位，输入引脚，低电平有效

## 分类

- 用于测试芯片的电气特性，检测芯片是否有问题；
- 用于Debug

# SoC典型实例及技术展望★

## 基于ARM的SoC系统

### 硬核

### GPU系统

图形处理器 (Graphic Processing Unit, GPU)，优势

- 高效的并行性
- 高密集的运算
- 超长的图形流水线

## 基于PowerPC的SoC系统

### 硬核

### 多核处理器

## 分类

- 总线或者交换开关互连的处理器
- 流处理器、图形处理器
- 网络互连的处理器（分布式）

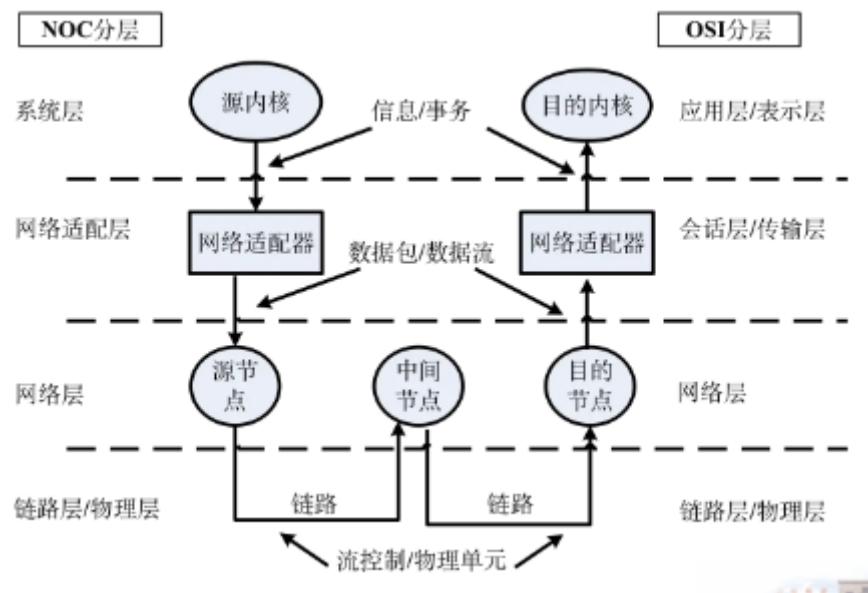
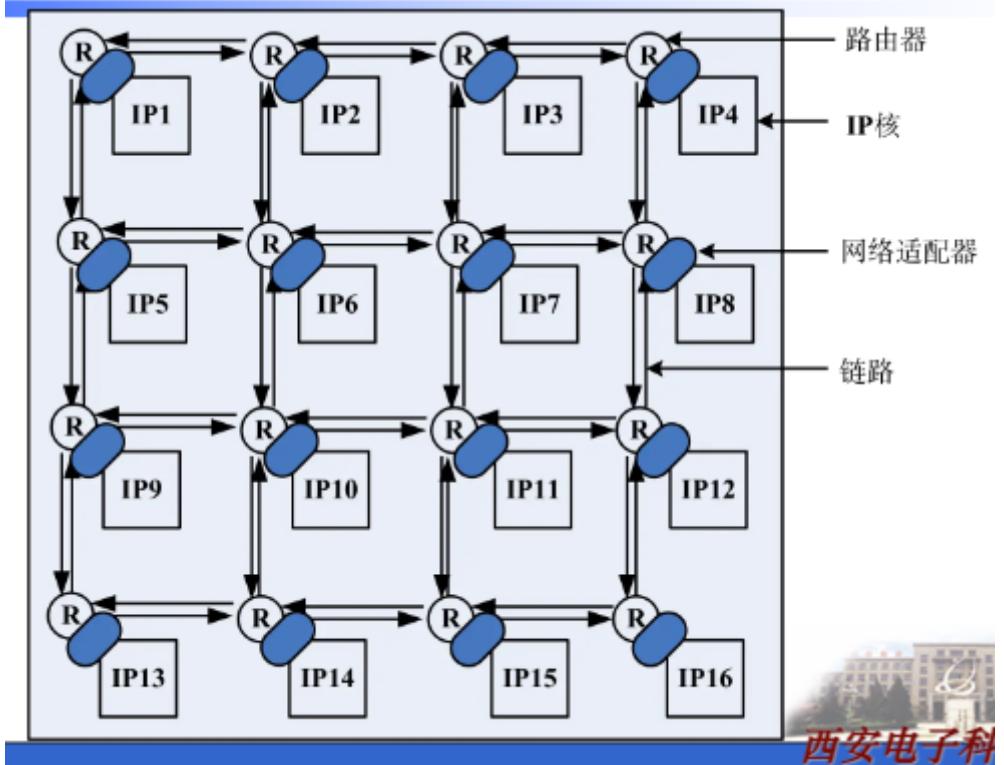
## 片上网络系统 ❤

### 采用路由和分组交换技术取代传统总线

片上网络 (Network on Chip, NoC) 借鉴了分布式计算系统的通信方式，采用路由和分组交换技术替代传统总线，是最有希望解决复杂片上通信问题的新方法。

## 结构

- 计算资源——一般由IP核和本地内存组成
- 通信网络——主要包含路由器、链路和网络适配器模块。



## 优势

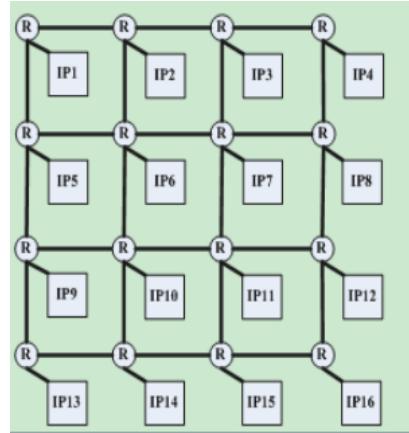
- 可扩展性
- 可重用性
- 可预测性
- 可定制性
- 设计灵活性
- 模块化
- 高性能

## NoC拓扑结构

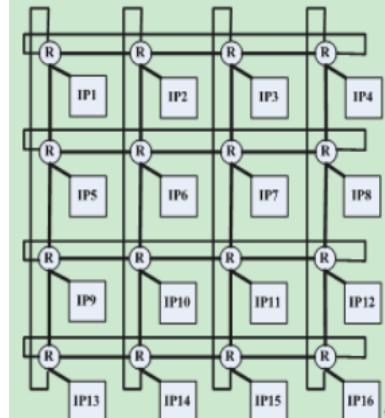
### 规则拓扑和不规则拓扑

拓扑结构	节点数	节点度	网络直径	链路数
2D Mesh	$N=n^2$	4	$2n-2$	$2(N-n)$
2D Torus	$N=n^2$	4	$n$	$2N$
Octagon	$N=n^2$	6	$\lceil N/4 \rceil$	$((N \bmod 8)+1) \times 12$
3D Mesh	$N=2^n$	4	$n$	$2(N-n)$
Spidergon	$N=n^2$	4	$\lceil N/4 \rceil$	$3N/2$
Cluster Mesh	$N=n^2$	4	$2n-2$	$5N-2n$
SPIN	$N=2^n$	4	$N/8$	$5N/2$
蝶网	$N=2^n$	2	$n+1$	$(n-1) \cdot N$

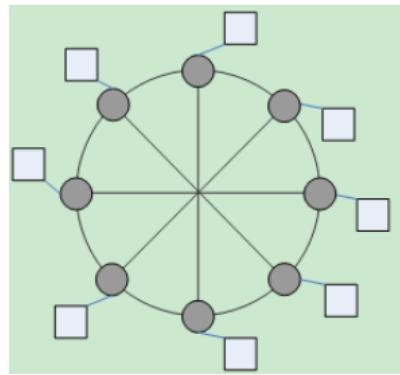
- 2D mesh拓扑



- 2D Torus拓扑

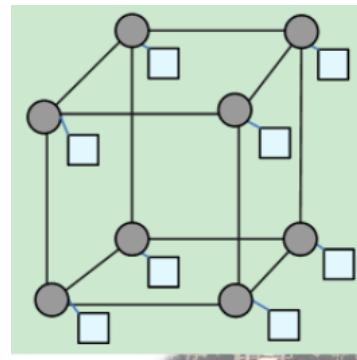


- Octagan拓扑

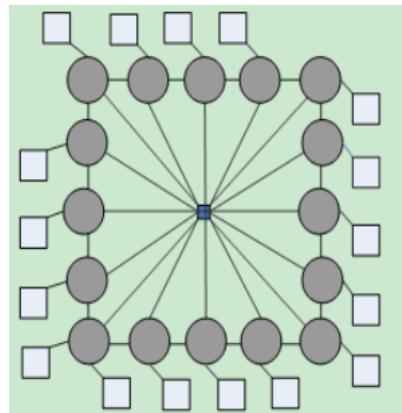


等八边形

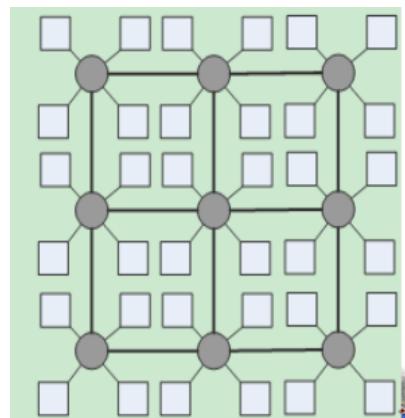
- 3D mesh拓扑



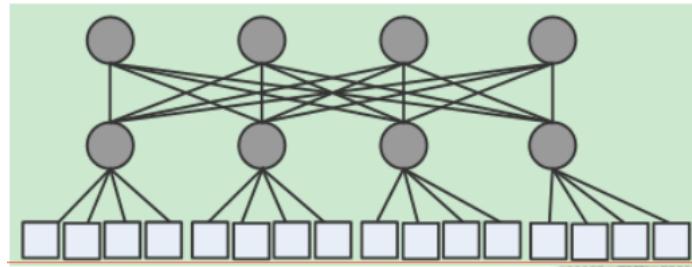
- spidergon拓扑



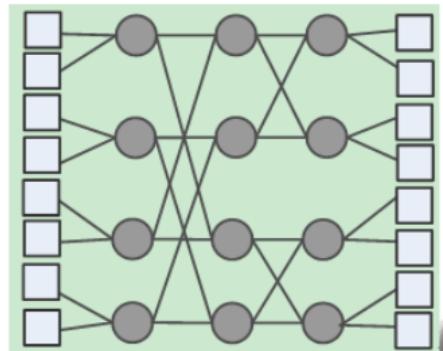
- Cluster mesh拓扑



- 一种可扩展的基于4叉胖树结构的SPIN拓扑结构



- 蝶网的基本构造



## 路由算法

### 分类

- 静态（确定）路由算法
- 动态（自适应）路由算法

## 确定性XY路由算法

确定性XY路由算法取决于源节点和目的节点的地址，与网络状况无关。

数据包首先在X方向进行传输，当数据包到达与目的节点同一列时，转向Y方向传输，最后到达目的节点。

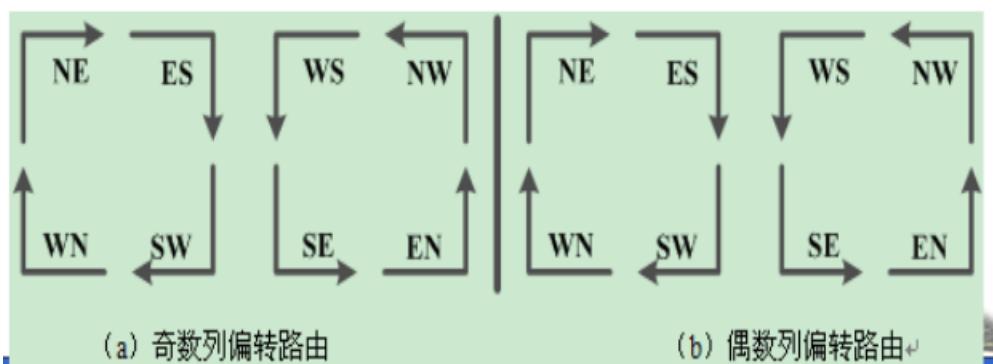
## 自适应的Nonth\_last、South\_last、Odd\_even和DyAD路由算法

### Odd\_even路由算法

禁止奇数列节点发生NW和SW的转向；

禁止偶数列节点发生EN和ES的转向；

禁止180度转向。



## NoC映射

指将通信任务、IP核映射到NoC体系结构处理单元的过程。

分类 (根据映射的结果是否改变)

- 动态映射
- 静态映射

## SoC优化设计★

### 同步设计优化问题

包括

- 条件判断语句优化

- 锁存的避免
- 无关态的使用

```
-- 不使用无关态
when others => dout <= '0'

-- 使用无关态
when others => dout <= '-'; -- 本质上时起了冗余电路删简的作用。
```

- 优先级问题

case语句无优先级，if语句有优先级（两者都是顺序语句）--这两者都是顺序执行语句  
multiple if statement和single if statement有不同的优先级顺序

- 多驱动与总线复用

多个信号源同时驱动一个负载，形成多驱动

要点：

- 行为级思维（软件思维）->硬件思维

```
rwproc: Process( cs, wr, rd )
Begin
    -- 而不是wr和rd分别驱动，造成多驱动问题
    cc1k <= wr and rd;
    if( cs = '0' ) then
        Addr( 15 downto 0 ) <= Ad(15 downto 0);
    elsif( cc1k'event and cc1k = '1' ) then
        Addr( 15 downto 0) <= Addr( 15 downto 0 ) + '1';
    end if;
End process;
```

- 总线复用
- 线与
- 双向端口的高阻态

```
-- 双向端口代码实现：分两个方向分别描述
-- db <= db;
If( dir = '1' ) then
    db <= da;
Else
    db <= "zzzz";
End if;

-- db >= da;
If( dir = '0' ) then
    da <= db;
Else
    da <= "zzzz";
End if;
```

- **毛刺的消除**

### 毛刺的产生

- 竞争和冒险
- 延时不平衡
- 线间干扰

### 毛刺消除方法

- 竞争冒险的避免
- Gray Coding (格雷码编码)
- 寄存器消除
- 其它

## 同步设计优化之时钟质量保证

- 时钟Skew (偏移/抖动，包括正偏移和负偏移) 的最小化

时钟偏移可能造成时序违背：建立时间违背和保持时间违背

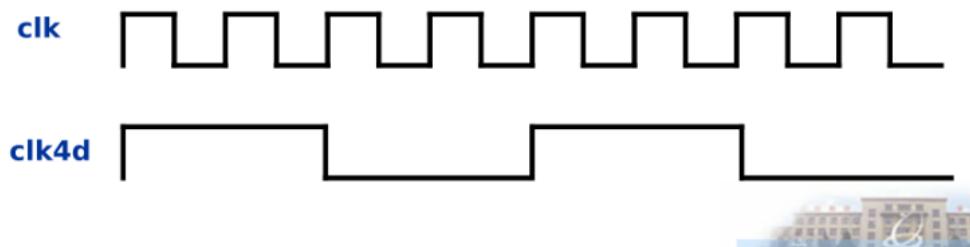
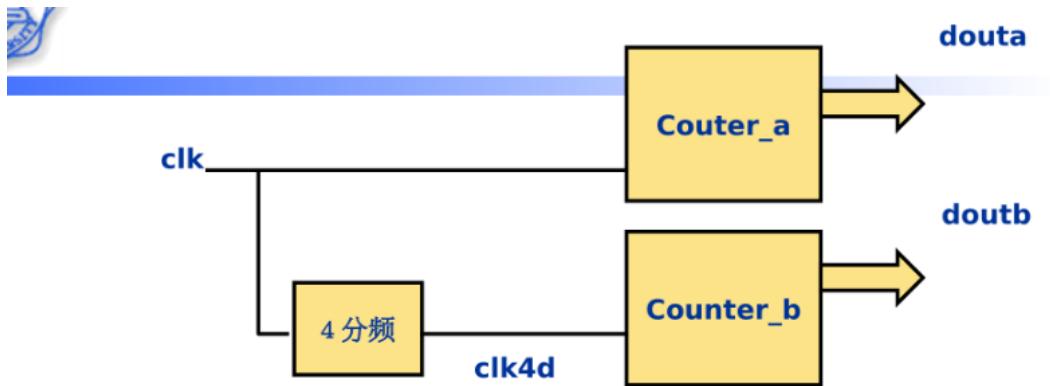
- **PLL** (锁相环) 的使用
- 避免使用门控时钟

危害：

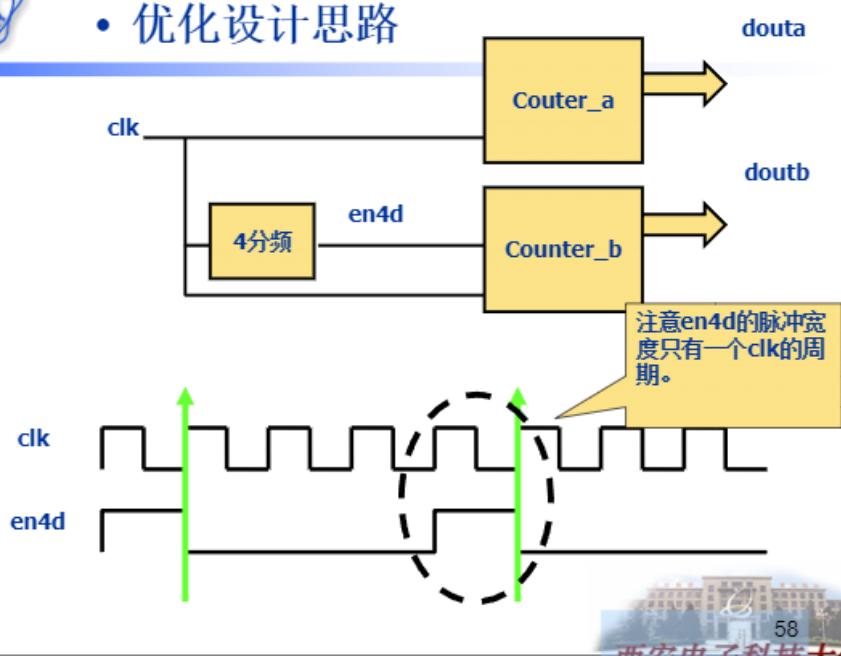
- 容易导致时序电路的误触发
- 增大了时钟skew
- 可以作为低功耗的手段

在设计中，尽量避免时钟通过组合电路，避免使用组合电路来产生时钟。

- 派生时钟（本质是分频）使用派生使能优化



## • 优化设计思路



58  
清华大学出版社

## 同步设计优化之路径延迟

**关键路径：**一个同步系统中的关键路径，就是它所有的路径中，路径延迟最长的那一条。同步系统的最高工作频率，等于关键路径延迟的倒数

### 延迟优化

- 避免长路径
- 优先级电路的延时优化
- 数据通路拷贝
- 数据运算式变换
- 变量运算优化
- 组合路径切割

- 状态机拆分
- 流水线技术
- 其它
- 双时钟沿问题
- 其它

## 异步设计优化问题

### 异步系统组成

- 全异步系统
- 异步多时钟系统

### 优化

- 消除亚稳态----同步化
- 注意其信号命名和模块划分方法
- 串行转并行: cascade □ Tree (for循环的优化)

#### 优化总结

##### 同步设计

尽量避免使用门控时钟，应以时钟使能来代替；  
 尽量避免使用派生时钟，应以派生使能来代替；  
 路径与路径延时的含义；路径延时与系统工作时钟的最高频率的关系；  
 缩小组合电路的规模，可以降低系统的路径延时从而提高最高工作速率  
 善于使用“信号搬移”技术进行延时优化，可以提高系统速度；手段包括优先级别电路中的模块搬移、数据通路拷贝、数据运算式变换等；  
 在优先级别电路中进行信号搬移时，要注意修正条件判断式，使得优先级别不变；  
 可能的情况下，应该尽量使用常量运算来代替变量(信号)运算；  
 可以使用状态机拆分、流水线等技术进行路径切割，达到延时优化的目的；  
 (寄存器输入和)寄存输出不仅可以提高系统工作速率，而且有利于模块和整体进行分别的静态时序分析

##### 异步设计

尽量避免使用异步设计；  
 异步多时钟系统中，注意做好时钟域之间的同步；  
 引入时钟域同步后，容易造成信号相位拉伸；  
 为避免矢量拉伸对数据矢量的破坏，可以考虑采用Gray码进行计数器编码；  
 为了在静态时序分析中快速地进行false path的设置，最好根据时钟域来进行信号命名。

## 高速设计的其它手段



### 高速设计的其他手段

(1) 串转并不但可以降低系统的延时，而且可以大幅度提高系统的数据吞吐率。二叉树结构可以有助于引入平衡的流水线。

(2) 安全的状态机描述：注意使用others分支，并且该分支不能以null来规定其行为，以处理状态机的“跑飞”。

# FPGA应用分析☆

## PLD

可编程逻辑器件

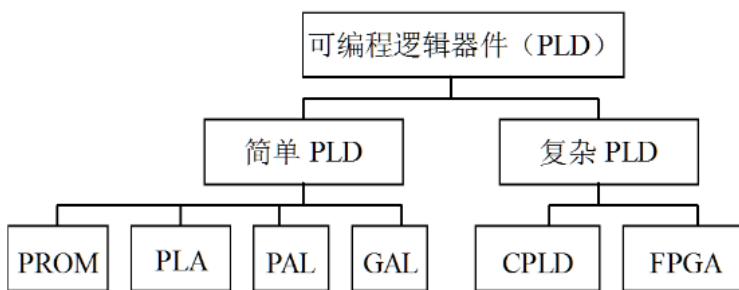
从原理上说“与或”阵列加上寄存器的结构就可以实现任何的数字逻辑电路；

### 结构

- “与阵列”和“或阵列”为主体——实现各种逻辑函数和逻辑功能
- 输入缓冲——增强输入信号的驱动能力
- 输出缓冲——对输出信号进行处理

### 分类

#### 1) 按集成度分



注意：仅有FPGA和CPLD输入复杂PLD，实际还包括EPLD

#### 2) 按照可以编程的次数分：

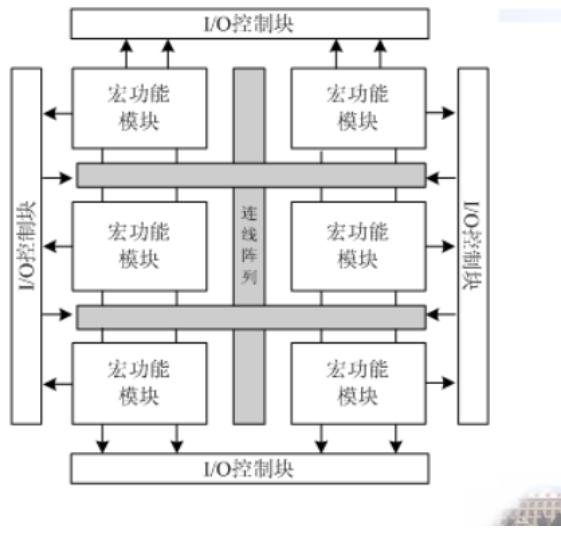
- ◆ 一次性编程器件 (OTP, One Time Programmable)
- ◆ 可多次编程器件

**OTP**类器件的特点是：只允许对器件编程一次，不能修改，而可多次编程器件则允许对器件多次编程，适合于在科研开发中使用。

## CPLD

### 结构

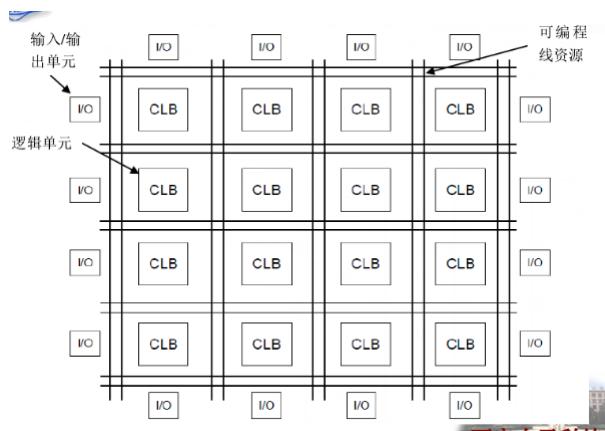
- 宏功能模块
- IO控制块
- 连线阵列



## FPGA

### 结构

- 可编程逻辑块PLB
- 输入输出模块IOB
- 互连资源IR



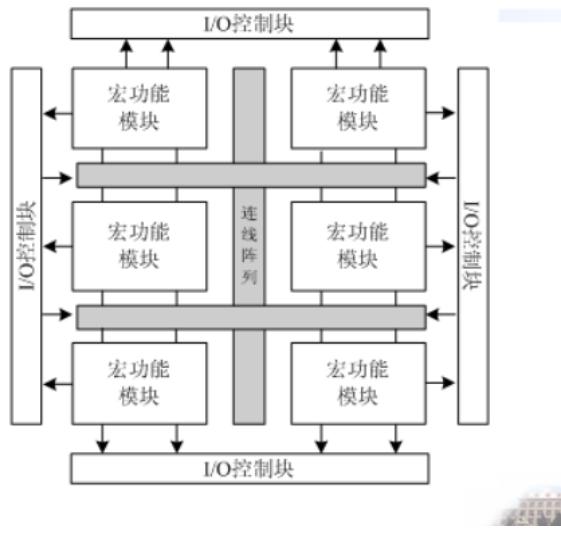
## SOPC技术

可编程片上系统技术SOPC

- 是片上系统SOC
- 可编程系统

## FPGA的嵌入式处理器

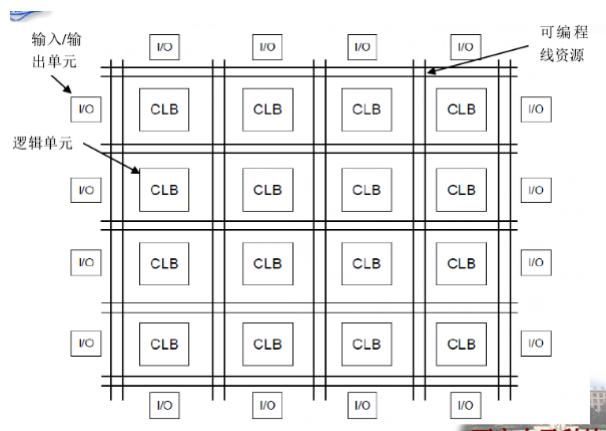
- 软核——leon2\leon3
- 固核——
- 硬核——PowerPC, ARM



## FPGA

### 结构

- 可编程逻辑块PLB
- 输入输出模块IOB
- 互连资源IR



## SOPC技术

可编程片上系统技术SOPC

- 是片上系统SOC
- 可编程系统

## FPGA的嵌入式处理器

- 软核——leon2\leon3
- 固核——
- 硬核——PowerPC, ARM