



北京大学

第五讲 SystemVerilog和FPGA

佟冬

tongdong@pku.edu.cn

北京大学计算机学院

课程回顾：布尔函数

□ 将一个开关函数 f 对于其变量每种可能取值的结果用表的形式表示。

– 1对应逻辑“真”；0对应逻辑“假”

□ 三个基本函数：与(AND)、或(OR)、非(NOT)的真值表

| $a \ b$ | $f(a, b) = a + b$ |
|---------|-------------------|
| 0 0 | 0 |
| 0 1 | 1 |
| 1 0 | 1 |
| 1 1 | 1 |

OR

| $a \ b$ | $f(a, b) = ab$ |
|---------|----------------|
| 0 0 | 0 |
| 0 1 | 0 |
| 1 0 | 0 |
| 1 1 | 1 |

AND

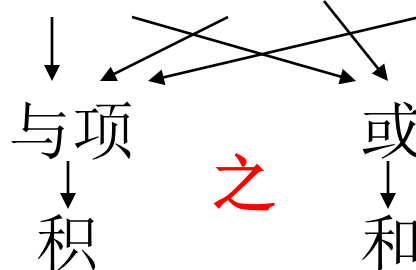
| a | $f(a) = \bar{a}$ |
|-----|------------------|
| 0 | 1 |
| 1 | 0 |

NOT

课程回顾：布尔函数范式

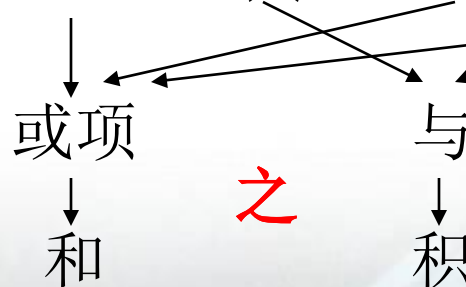
积之和(Sum of product, SOP)

$$f(A, B, C) = AB + A'C + AC'$$



和之积(Product of sum, POS)

$$f(A, B, C) = (A' + B + C)(B' + C + D')(A + C' + D)$$



课程回顾：二进制在电路中的对应

□ 电信号和逻辑值

- 在电路中，用电压的高低来表示逻辑值

| 电信号 | | 逻辑值 | |
|------|--------------|-------|-------|
| | | 正逻辑 | 负逻辑 |
| 高电压H | V_H^{\max} | 1 (真) | 0 (假) |
| 不稳定 | | | |
| 低电压L | V_L^{\min} | 0 (假) | 1 (真) |

- 一个信号被置为逻辑1称为有效的或者真。
- 一个信号被清为逻辑0称为无效的或者假
- 高有效信号（正逻辑）和低有效信号（负逻辑）
- 信号的极性(Polarity)表示信号是高有效或是低有效

课程回顾：基本逻辑门

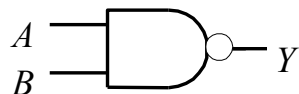
□ 与非门

| a | b | $f_{NAND}(a, b) = ab$ |
|-----|-----|-----------------------|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

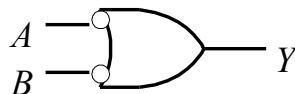
(a)

| A | B | Y |
|-----|-----|-----|
| L | L | H |
| L | H | H |
| H | L | H |
| H | H | L |

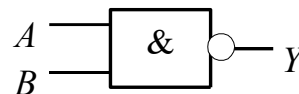
(b)



(c)



(d)

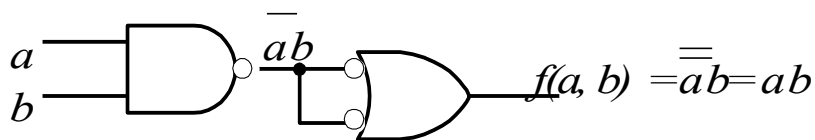


(e)

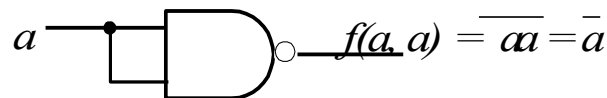
- (a) 与非门的逻辑功能
- (b) 与非门的电子功能
- (c) 标准符号表示
- (d) 替代符号（负逻辑）
- (e) IEEE 块符号表示

课程回顾：与非门的特性

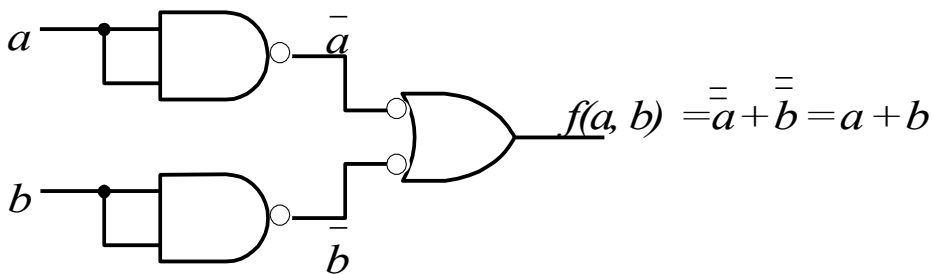
□ AND, OR, NOT门可以仅用NAND门来构造



AND gate



NOT gate



OR gate

课程回顾：与非门的开关模型

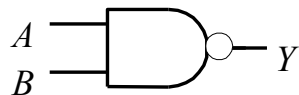
□ 与非门(NAND)

| a | b | $f_{NAND}(a, b) = ab$ |
|-----|-----|-----------------------|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

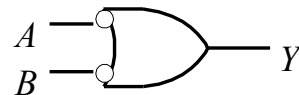
(a)

| A | B | Y |
|-----|-----|-----|
| L | L | H |
| L | H | H |
| H | L | H |
| H | H | L |

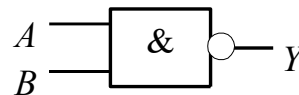
(b)



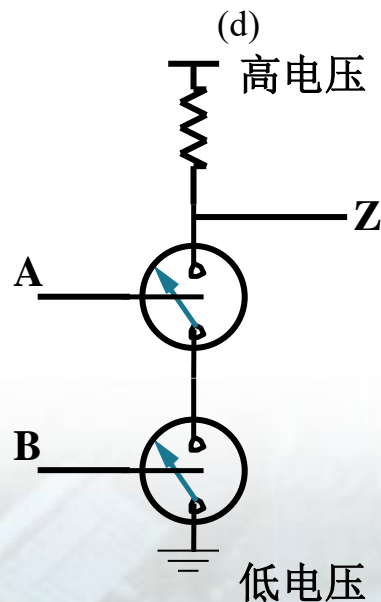
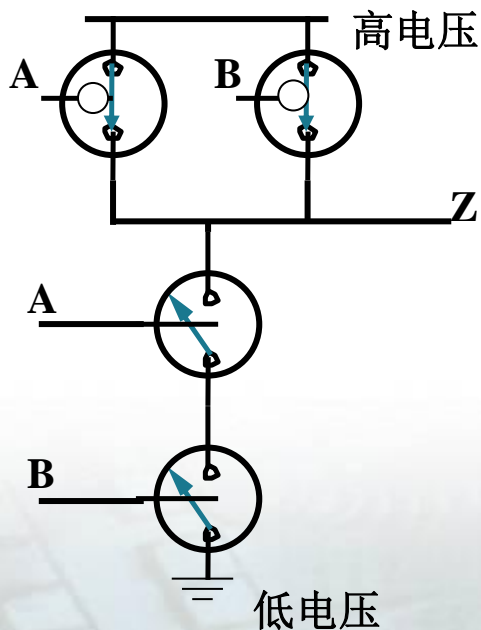
(c)



(d)

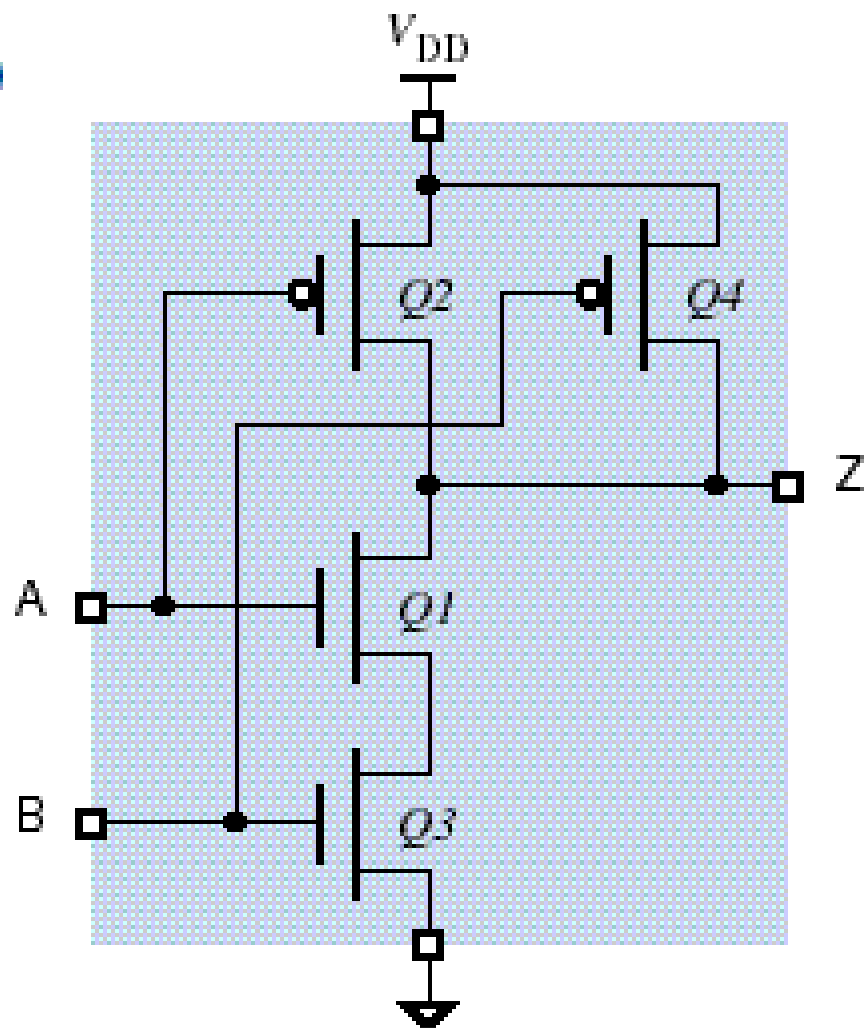


(e)



课程回顾：CMOS与非门(NAND)

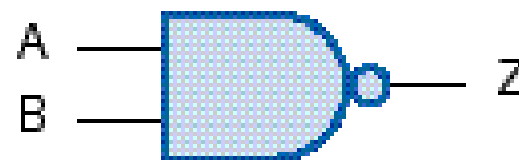
(a)



(b)

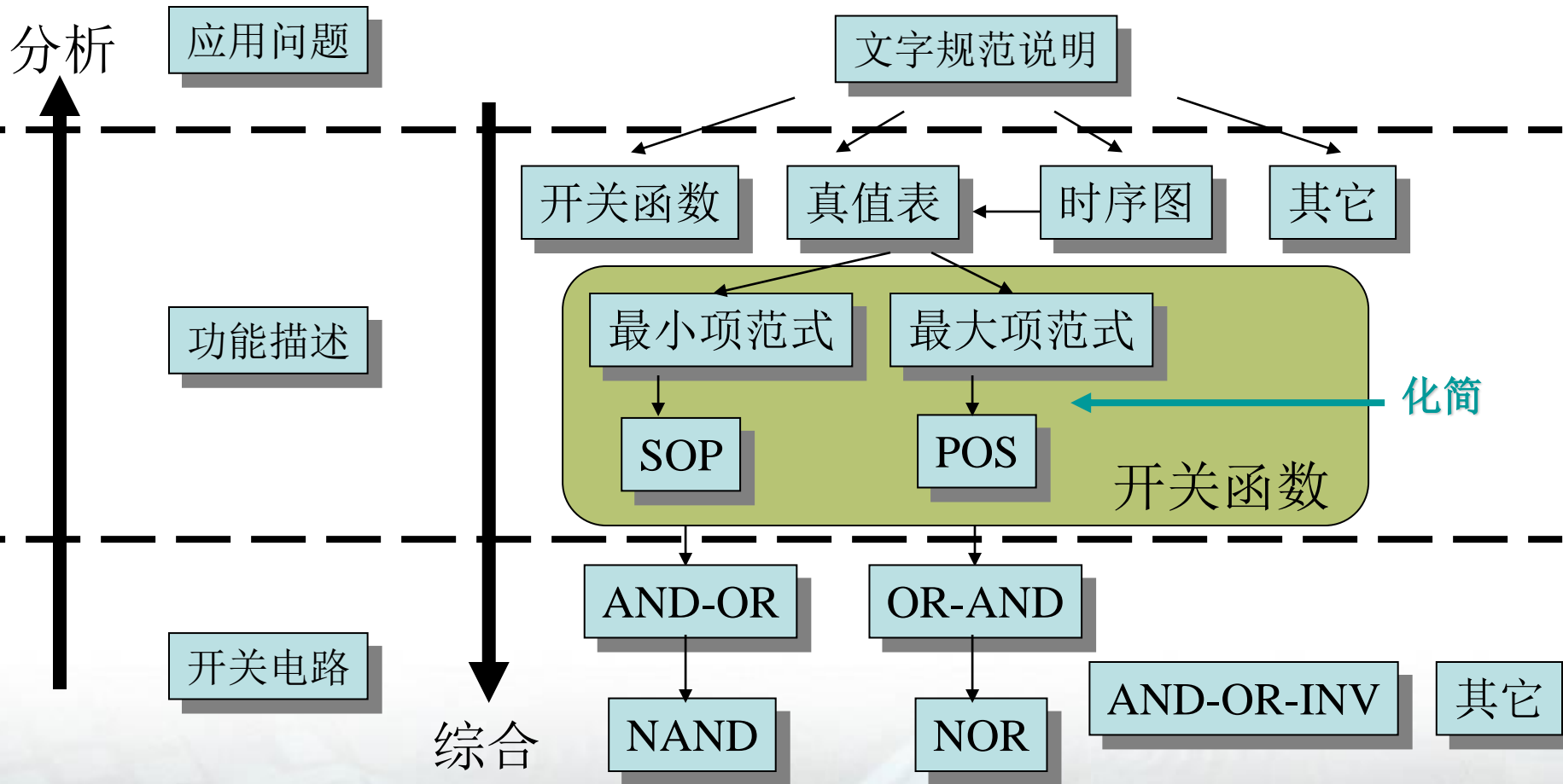
| A | B | $Q1$ | $Q2$ | $Q3$ | $Q4$ | Z |
|---|---|------|------|------|------|---|
| L | L | off | on | off | on | H |
| L | H | off | on | on | off | H |
| H | L | on | off | off | on | H |
| H | H | on | off | on | off | L |

(c)



组合电路分析与设计

□ 组合电路的分析与综合





SystemVerilog 硬件描述语言



数字电路的表示法

- 物理器件 Physical devices (transistors, relays)
- 开关 **Switches**
- 真值表 **Truth tables**
- 布尔代数 **Boolean algebra**
- 逻辑门 **Logic Gates**
- 波形图 **Waveform**
- 有穷状态行为 **Finite state behavior**
- 寄存器传输行为 **Register-transfer behavior**
- 并发抽象描述 Concurrent abstract specification

硬件描述语言

Hardware description languages (HDL)

- 在多种抽象级别上描述硬件
- 结构级描述 (Structural description)
 - 原理图的文本替代
 - 功能模块(module)的模块层次化组合
- 行为级/功能级描述 (Behavioral/functional description)
 - 描写模块做什么？而不描述如何做
 - 可综合产生模块电路
- 模拟语义

硬件描述语言HDLs

- ❑ Abel (~1983) - Data-I/O开发
 - 面向可编程逻辑器件
 - 对状态机的支持不够得好
- ❑ ISP (~1977) – CMU研究项目
 - 面向模拟，不能综合
- ❑ SystemVerilog (2005)
 - Verilog (~1985) - Gateway 开发 (被Cadence合并)
 - 类Pascal和C语言
 - 延迟delays只在模拟器中有效
 - 很容易有效地编程
- ❑ VHDL (~1987) – 美国国防部DoD发起的标准
 - 类Ada语言 (着重于可复用性和可维护性)
 - 显式的模拟语义
 - 很通用但也很繁琐

HDL和编程语言的关系

□ 程序结构Program Structure

- 多次例化相同类型的模块
- 通过原理图表明模块之间的连接关系
- 模块的层次化结构

□ 赋值Assignment

- 连续赋值continuous assignment (逻辑始终计算)
- 传播延迟propagation delay (计算耗费时间)
- 信号时序非常重要 (计算的结果何时产生效果)

□ 数据结构Data structures

- 位宽显式的拼写出来, 不支持动态结构
- 不支持指针

□ 并行性Parallelism

- 硬件是自然地并行运行 (必须支持多线程)
- 赋值并行的发生 (不仅仅是串行执行的)

事件驱动模拟：硬件是并行的！

U.S. Patent

Dec. 9, 1997

Sheet 3 of 8

5,696,94

U.S. Patent

Dec. 9, 1997

Sheet 5 of 8

5,696,942

Patent

Dec. 9, 1997

Sheet 6 of 8

5,696,942

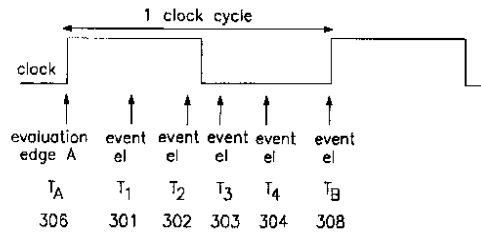


FIG. 3

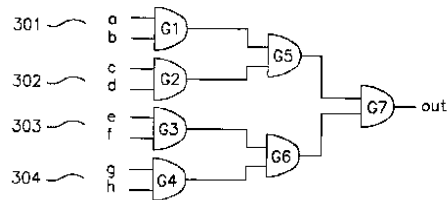


FIG. 4

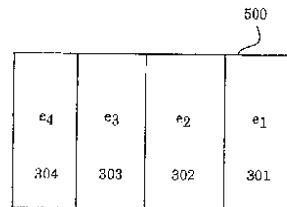


FIG. 5a

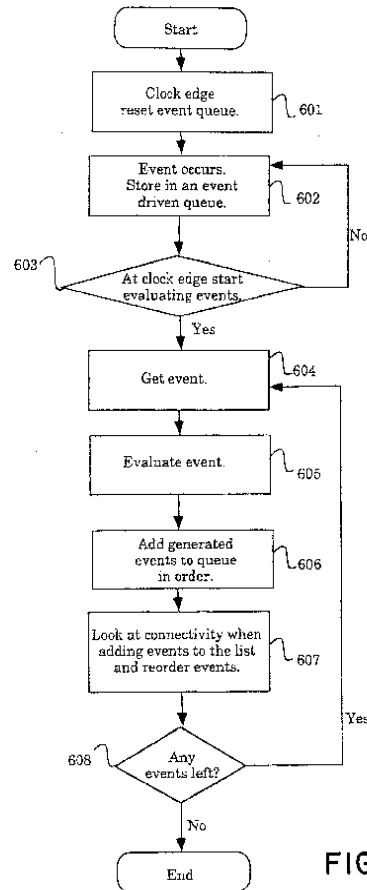


FIG. 6a

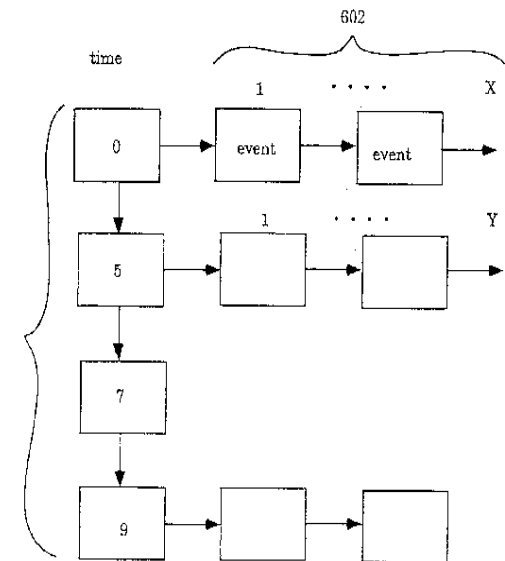


FIG. 6b

HDL和组合逻辑

- ❑ **Modules:** 说明输入，输出，双向和内部信号
- ❑ **连续赋值:** 一个门的输出任何时刻都是输入的函数结果（不需要调用）
- ❑ **传播延迟:** 输入影响输出的时间和延迟的概念
- ❑ **构成Composition:** 通过线将模块连接在一起
- ❑ **层次化:** 模块分解成一些功能模块

组合电路的SystemVerilog描述

□ SystemVerilog

□ 类型：

- Input/output
- net
- logic

□ 赋值：

- Assign
- 阻塞赋值=
- 选择赋值= $s ? a : b$

□ Always_comb

- 敏感向量表
- if-else
- case/casex/casez

□ 信号的合并与拆分

□ Verilog

□ 类型：

- Input/output
- Wire
- Reg

□ 赋值：

- Assign
- 阻塞赋值=
- 选择赋值= $s ? a : b$

□ Always @()

- 敏感向量表
- if-else
- case/casex/casez

□ 信号的合并与拆分

SystemVerilog 模块定义

```
module xor_gate (                // 模块定义
    input logic a, b,           // 输入端口定义
    output logic y);           // 输出端口定义

    <模块内功能描述>

endmodule                        // 模块结束
```

异或门XOR的SystemVerilog结构级描述

```
module xor_gate (  
    input  logic a, b,  
    output logic z );  
    logic abar, bbar, t1, t2;
```

```
// 模块定义  
// 输入端口定义  
// 输出端口定义  
// 内部信号定义
```

```
    not_gate inva(abar, a);  
    not_gate invb(bbar, b);  
    and_gate and1(t1, a, bbar);  
    and_gate and2(t2, b, abar);  
    or_gate or1(z, t1, t2);  
endmodule
```

```
// 实例化一个非门inva  
// 实例化一个非门invb  
// 实例化一个与门and1  
// 实例化一个与门and2  
// 实例化一个或门or1  
// 模块结束
```

SystemVerilog行为级描述- always_comb

| | |
|-------------------|--------------|
| module xor_gate (| // 模块定义 |
| input logic a, b, | // 输入端口定义 |
| output logic z); | // 输出端口定义 |
| always_comb | // 输入变化时模拟执行 |
| begin | // 单句可省略 |
| z = a ^ b; | // 位异或计算 |
| end | // 单句可省略 |
| endmodule | // 模块结束 |

SystemVerilog运算符和优先级

| | 操作符 | 操作含义 |
|---------|--------------|-----------------------------|
| Highest | ~ | NOT |
| | *, /, % | MUL, DIV, MOD |
| | +, - | PLUS, MINUS |
| | <<, >> | Logical Left/Right Shift |
| | <<<, >>> | Arithmetic Left/Right Shift |
| | <, <=, >, >= | Relative Comparison |
| | ==, != | Equality Comparison |
| Lowest | &, ~& | AND, NAND |
| | ^, ~^ | XOR, XNOR |
| | ~, ~ | OR, NOR |
| | ?: | Conditional |

~&, ~|, ~^三个操作只能用于缩位运算 (bit reduction operations)

SystemVerilog中的always中的阻塞赋值

```
module xor_gate (                                // 模块定义
    input  logic a, b,                          // 输入端口定义
    output logic z );                          // 输出端口定义
    logic  temp;                                // 内部信号

    always_comb                                // 输入变化时模拟执行
    begin
        temp = a ^ b;                          // 阻塞赋值;
        z = temp;                              // z = a ^ b;
    end                                         // always结束
endmodule                                       // 模块结束
```

SystemVerilog行为级描述- if-else

```
module xor_gate (                                // 模块定义
    input  logic a, b,                          // 输入端口定义
    output logic z );                          // 输出端口定义

    always_comb                                // 输入变化时模拟执行
    begin
        if (a) z=~b; else z = b;              // 0通过1取反
    end                                         // always结束

endmodule                                       // 模块结束
```

SystemVerilog数值的表示

| 数字 | 位数 | 基数 | 值 | 存储值 |
|--------------|----|----|-----|----------------|
| 3'b101 | 3 | 2 | 5 | 101 |
| 'b11 | ? | 2 | 3 | 000 ... 0011 |
| 8'b11 | 8 | 2 | 3 | 00000011 |
| 8'b1010_1011 | 8 | 2 | 171 | 10101011 |
| 3'd6 | 3 | 10 | 6 | 110 |
| 6'o42 | 6 | 8 | 34 | 100010 |
| 8'hAB | 8 | 16 | 171 | 10101011 |
| 42 | ? | 10 | 42 | 00 ... 0101010 |

x: 无效逻辑值

z: 三态门悬空输出

异或门的SystemVerilog行为级描述- case()

```
module xor_gate (                                // 模块定义
    input  logic a, b,                          // 输入端口定义
    output logic z );                          // 输出端口定义

    always_comb                                // 输入变化时模拟执行
    begin
        case(a)                                // case体
            1'b0: z = b;                       // 0通过
            1'b1: z = ~b;                      // 1取反
            default: z = x;                    // 默认输出
        endcase                                // case结束
    end                                         // always结束
endmodule                                       // 模块结束
```

异或门的SystemVerilog行为级描述- assign

```
module xor_gate (                // 模块定义
    input  logic a, b,           // 输入端口定义
    output logic z );           // 输出端口定义

    assign z = a ^ b;           // 无论输入值是否
                                // 变化都执行

endmodule                       // 模块结束
```

SystemVerilog中的- assign阻塞赋值

```
module xor_gate (                // 模块定义
    input  logic a, b,           // 输入端口定义
    output logic z );           // 输出端口定义

    assign temp = a ^ b;         // 阻塞赋值
    assign z = temp;             // z = a ^ b;

endmodule                       // 模块结束
```

SystemVerilog的 assign阻塞赋值的并行性

```
module xor_gate (                // 模块定义
    input  logic a, b,           // 输入端口定义
    output logic z );           // 输出端口定义
    logic temp;

    assign z = temp;              // z = a ^ b;
    assign temp = a ^ b;         // 阻塞赋值
    // 交换顺序仍然正确，描述的硬件是一样的

endmodule                        // 模块结束
```

SystemVerilog行为级描述 - 选择assign

```
module xor_gate (                // 模块定义
    input  logic a, b,           // 输入端口定义
    output logic z );           // 输出端口定义

    assign z = (a) ? ~b : b;      // 0通过1取反

endmodule                        // 模块结束
```

1-bit全加器进位的verilog行为级描述-casez

```
module add1( cout, a, b, cin);  
    input  logic a, b, cin;  
    output logic cout;  
  
    always_comb begin  
        casez({a, b, cin})  
            // 3位输入为1的个数：大于等于2，进位1; 小于2，进位0  
            3'b11?, 3'b1?1, 3'b?11: cout = 1;  
            3'b00?, 3'b0?0, 3'b?00: cout = 0;  
            default: cout = 1'bx;  
        endcase  
    end  
endmodule
```

SystemVerilog行为级描述-信号拼接

```
module adder2(  
    input logic [1:0] a,b;           // 2-bit输入信号  
    input logic cin;                 // 1-bit输入信号  
    out logic [1:0] s;               // 2-bit输出信号  
    out logic cout );                // 1-bit输出信号  
    assign {cout, s} = a  + b  + cin; // 信号拼接  
//      1bit  2bit  2bit  2bit  1bit  
endmodule
```

SystemVerilog

```
assign y = {c[2:1], {3{d[0]}}}, c[0], 3'b101};
```

2-bit全加器的Verilog行为级描述-信号拼接

```
module adder2(  
    input logic [1:0] a,b;           // 2-bit输入信号  
    input logic cin;                 // 1-bit输入信号  
    out logic [1:0] s;               // 2-bit输出信号  
    out logic cout );               // 1-bit输出信号  
  
    always_comb  
    begin  
        {cout, s} = a + b + cin;    // 信号拼接  
    end  
endmodule
```


2-bit 全加器的Verilog行为级描述- 信号连接

```
module adder2(  
    input logic [4:0] in5,           // 5-bit输入信号  
    out logic [2:0] out3 );          // 2-bit输出信号  
    logic [1:0] a, b, s;             // 2-bit连线  
    logic cin, cout;                 // 1-bit连线  
  
    assign {a, b, cin} = in5;        //信号拼接  
    // a = in5[4:3]; b = in5[2:1]; cin = in5[0];  
    assign {cout, s} = a + b + cin;   //信号拼接  
    assign out3 = {cout, s};         //信号拼接  
  
endmodule
```

测试平台Testbench

```
module testbench1();  
    logic a, b, c, y;  
  
    // instantiate device under test  
    sillyfunction dut(a, b, c, y);  
  
    // apply inputs one at a time  
    initial begin  
        a = 0; b = 0; c = 0; #10;  
        c = 1;                #10;  
        b = 1; c = 0;          #10;  
        c = 1;                #10;  
        a = 1; b = 0; c = 0; #10;  
        c = 1;                #10;  
        b = 1; c = 0;          #10;  
        c = 1;                #10;  
    end  
endmodule
```

Lab0 入门教程

□ FPGA环境

□ Lab0 4月10日 教学网提交

□ 将工程目录压缩成<学号>.zip文件

组合电路的实现技术

□ 标准逻辑门Standard gates

- 逻辑门封装芯片, 标准单元库

□ 规整逻辑Regular logic

- 多选器multiplexers
- 译码器decoders

□ 两级可编程逻辑Two-level programmable logic

- PALs, PLAs,

□ 存储器实现逻辑

- ROMs
- 现场可编程门阵列(Field-Programming Gate Array, FPGA)

存储器和组合逻辑

□ 用存储器组合逻辑实现

$$F0 = A' B' C + A B' C' + A B' C$$

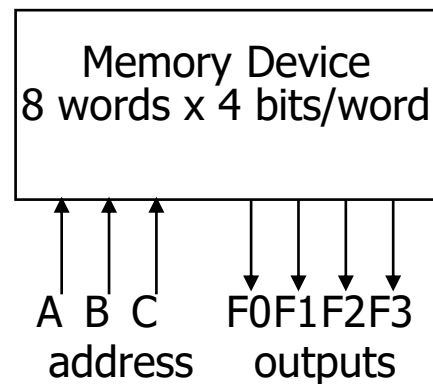
$$F1 = A' B' C + A' B C' + A B C$$

$$F2 = A' B' C' + A' B' C + A B' C'$$

$$F3 = A' B C + A B' C' + A B C'$$

| A | B | C | F0 | F1 | F2 | F3 |
|---|---|---|----|----|----|----|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 |

真值表

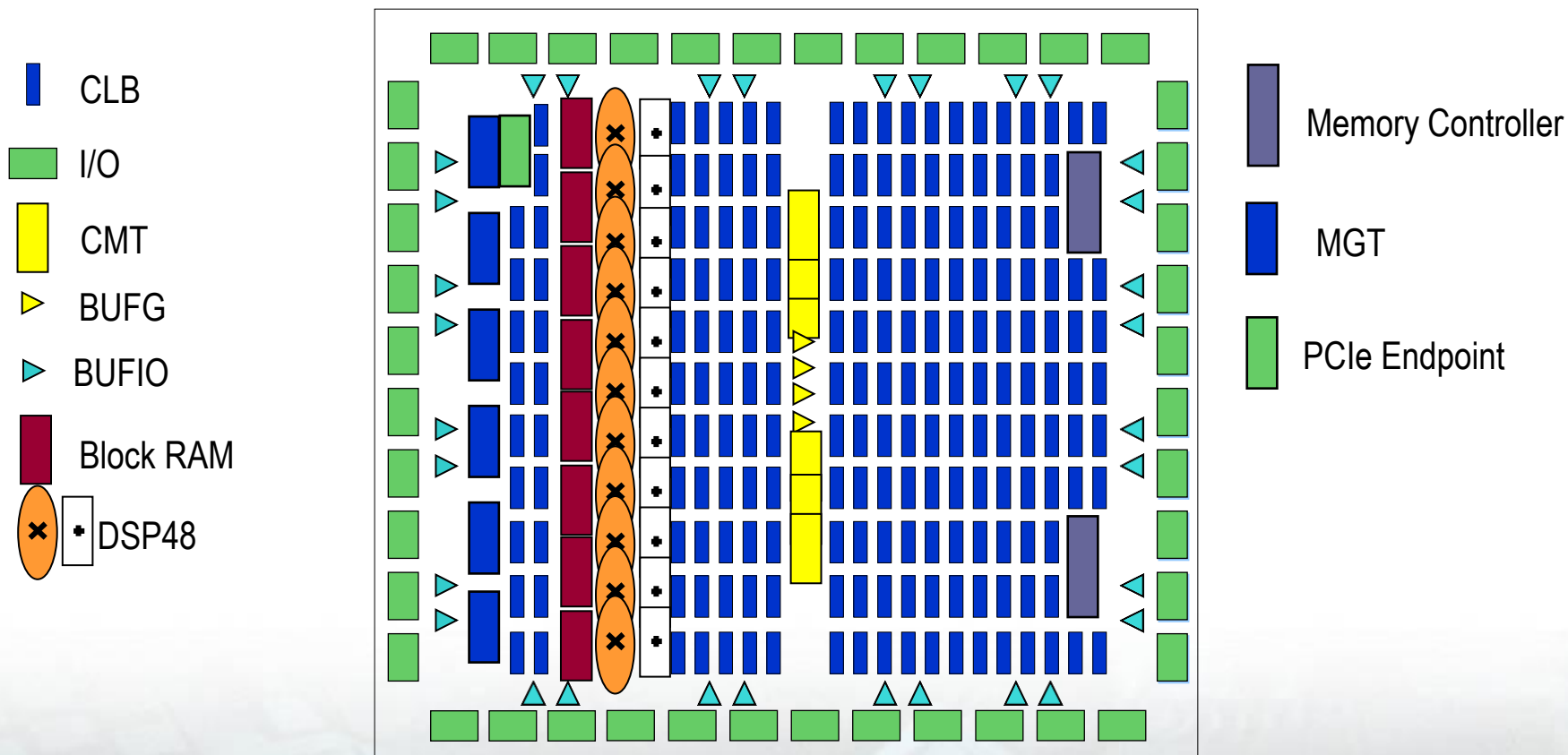


框图

Xilinx FPGA概述

□ FPGA Field Programmable Gate Array

— 现场可编程门阵列



Xilinx FPGA概述

□ 所有的Xilinx FPGAs都包含相同的基本资源

– 逻辑资源Logic Resources

- 片(Slices)组成CLB(Configurable Logic Block 可配置逻辑块)
 - 包含组合逻辑和寄存器资源
- 存储器Memory
- 乘法器Multipliers

– 互连资源Interconnect Resources

- 可编程互连资源Programmable interconnect
- 输入输出块IOBs
 - FPGA与外部的接口

– 其它资源Other resources

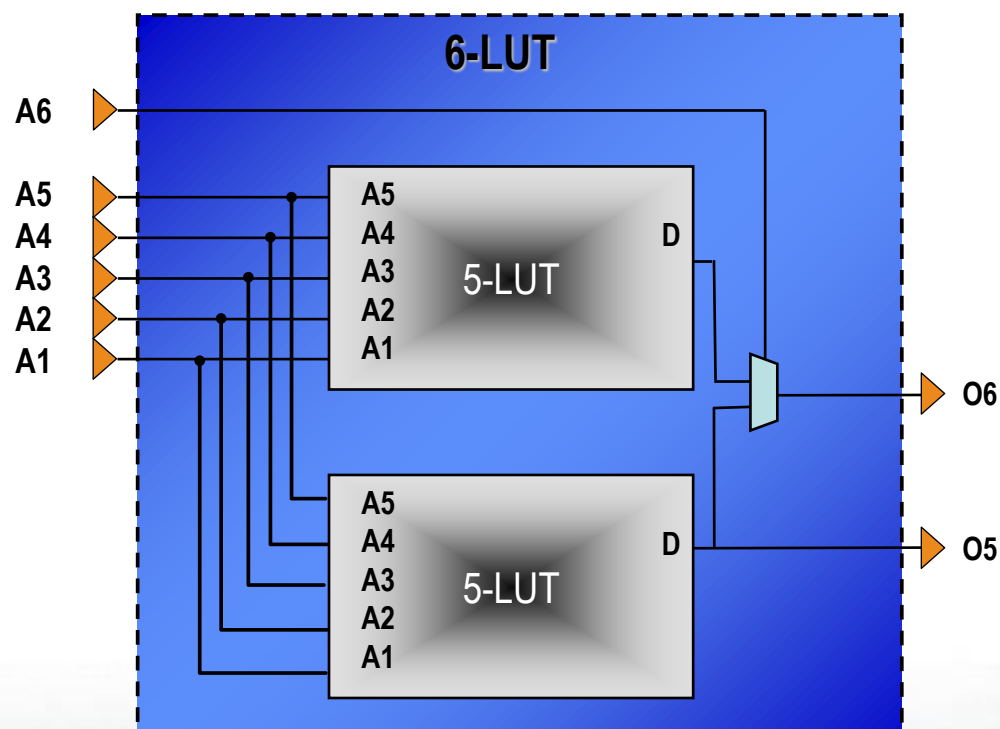
- 全局时钟缓冲器Global clock buffers
- 边缘检测逻辑Boundary scan logic



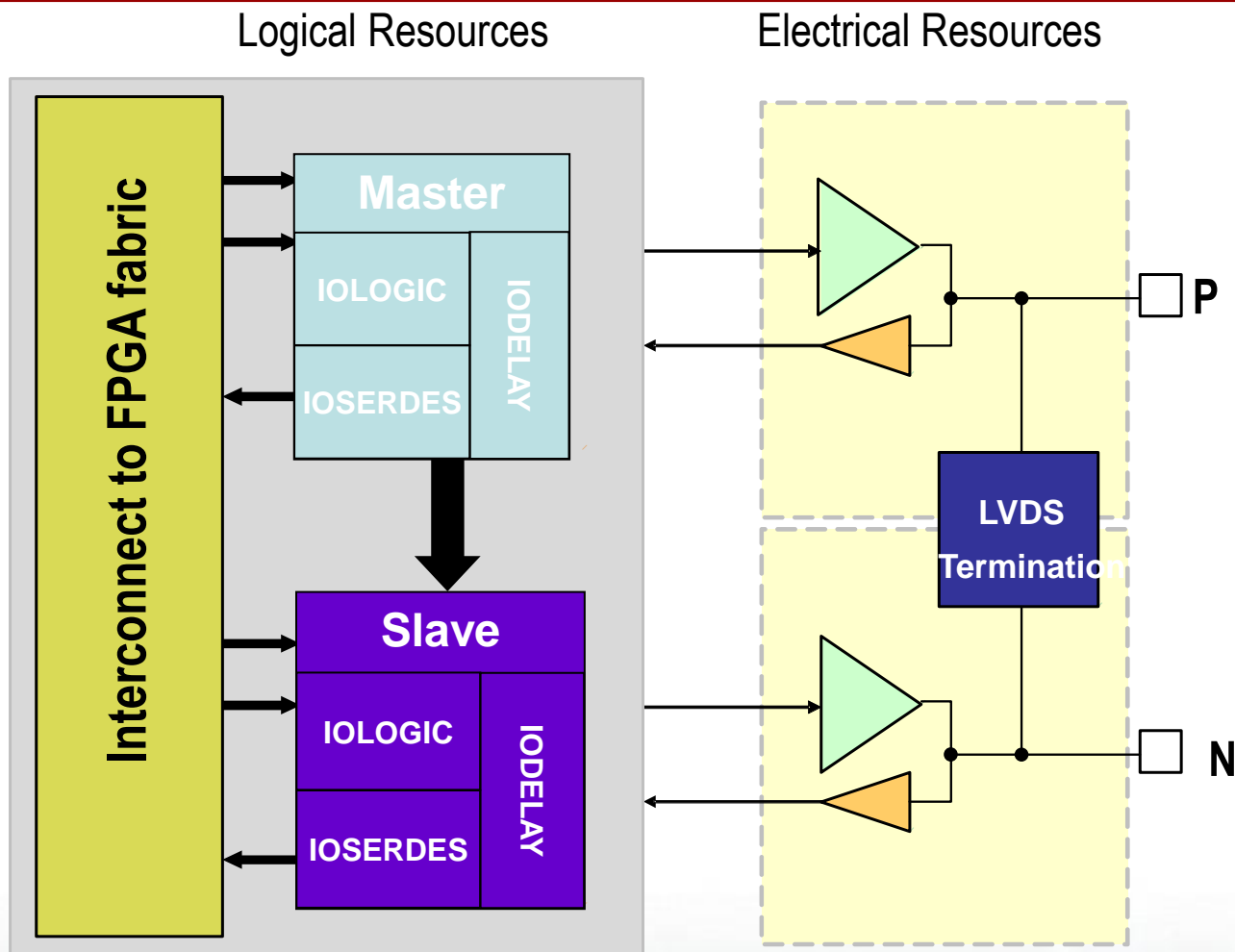
6-Input LUT with Dual Output

□ 6-input LUT 包含两个共用输入的5-input LUTs

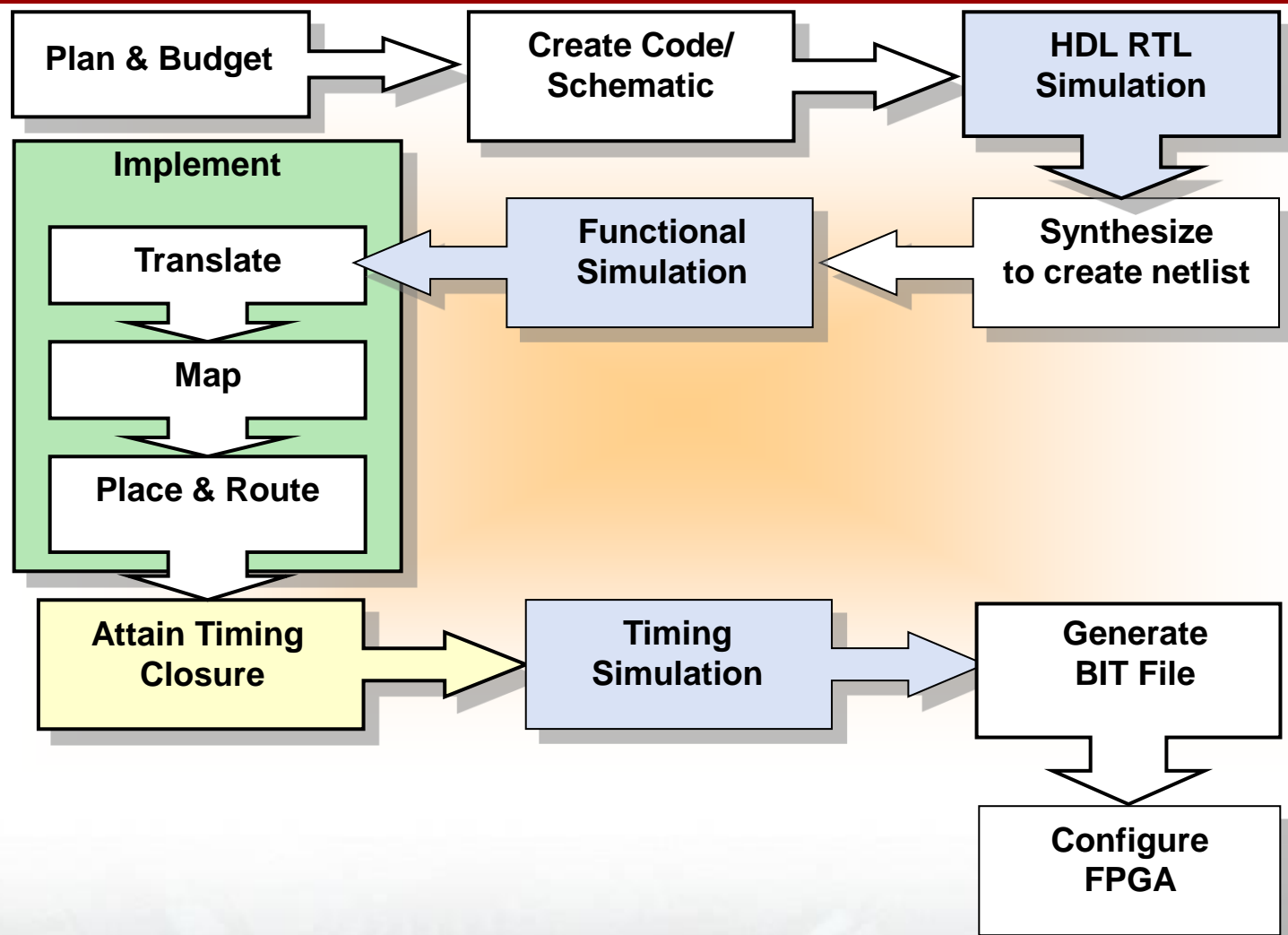
- 影响一个6-input LUT最优速度
- 一个或者两个输出
- 任意6变量的函数
- 两个独立的5变量函数



输入输出块(IOB)的结构图



Xilinx设计流程



电子设计自动化软件

- ❑ CAD, Computer-aid Design
- ❑ EDA, Electronic Design Automatic
- ❑ ESL, Electronic System Level

- ❑ 系统设计(C/C++, SystemC)
- ❑ 设计输入(Verilog, VHDL, Schematic)
- ❑ 设计验证(Verification)
- ❑ 设计综合(ASIC, FPGA, Analog)
- ❑ 后端(Back-end)设计

1 设计输入 Design Entry

- 原理图输入
- 激励和输出逻辑方程
- 状态表
- 状态图或者ASM图
- 硬件描述语言
 - Verilog
 - VHDL

2 综合 Synthesis

- 状态优化
- 选择状态分配方案
- 选择触发器类型
- 对组合逻辑进行优化

3 设计验证Verification

□ 功能分析与验证

- 模拟 Simulation
- 形式化验证 Formal Verification

□ 时序分析

- Timing Analysis
- Timing Methodology

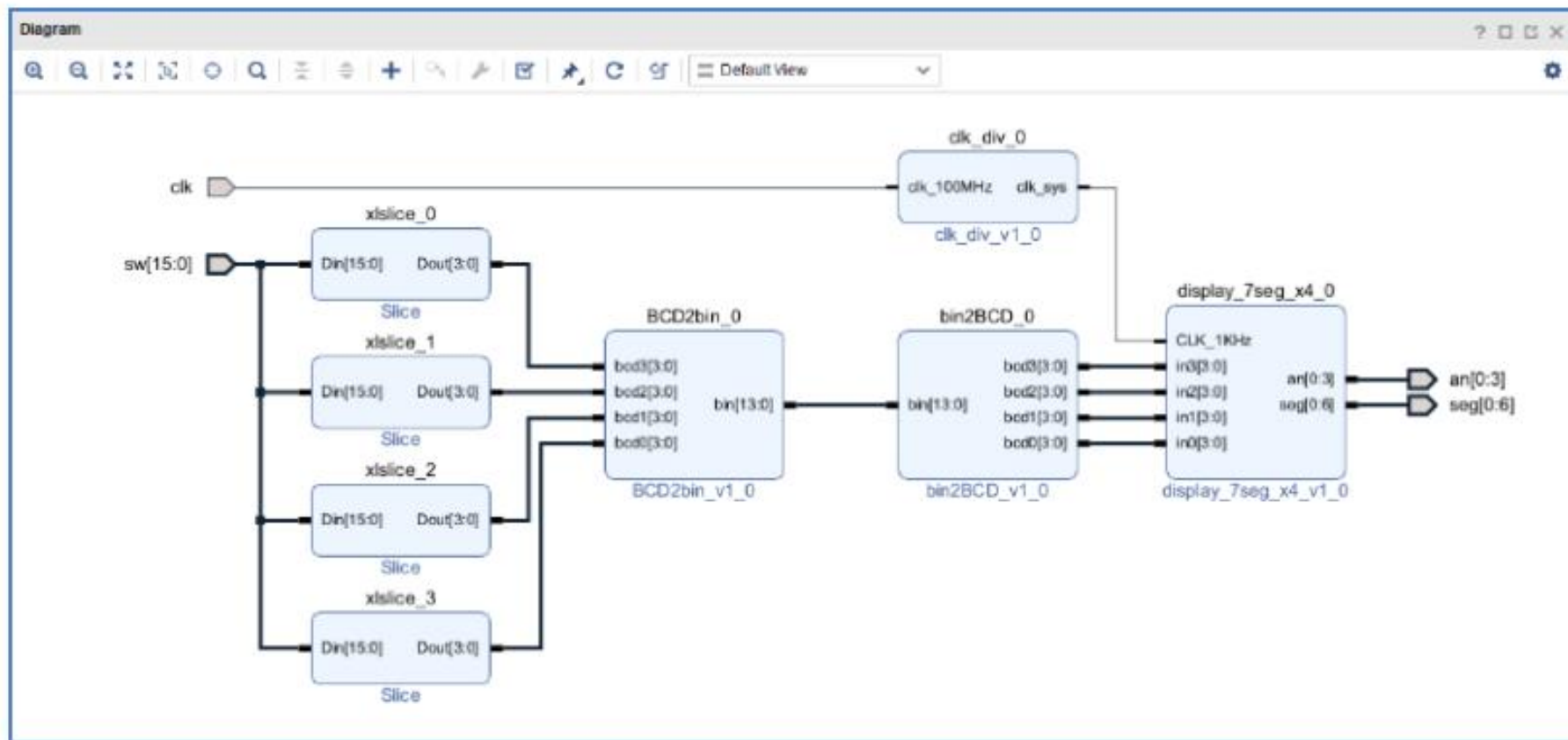
4后端(Back-end)设计

- 时钟树生成
- Place & Route
- 参数提取, 后仿真
- 物理验证 (信号完整性分析等)
- 形成加工工艺文件
 - FPGA
 - ASIC, GDSII
 - ...

Lab1：二进制和BCD的转换模块

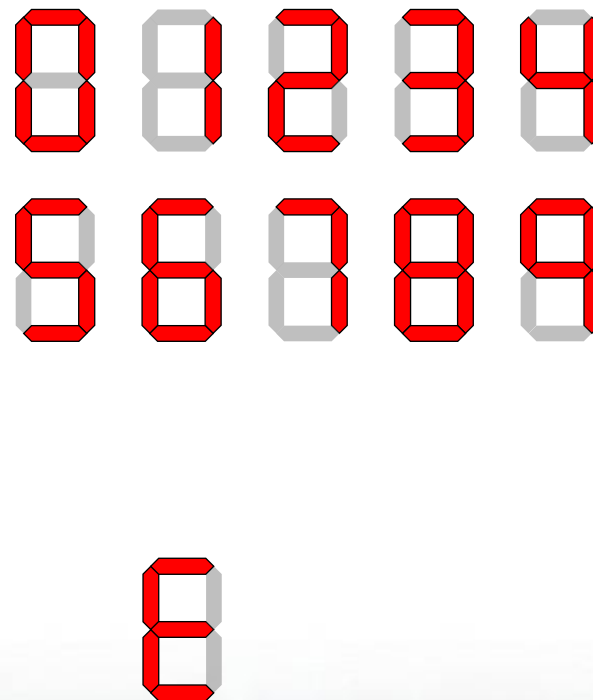
- Lab1 4月17日 00:00之前教学网提交
- 要求所有的命名都以学号后两位结尾！！！！
 - name_<姓名缩写>
- 将工程目录压缩成<学号>.zip文件

BCD和二进制相互转换以及7段译码器显示



添加出错显示E字母

```
always_comb
begin
    case (seg7_dec_in)
    //      ABC_DEFG
    0: seg = 7'b000_0001;
    1: seg = 7'b100_1111;
    2: seg = 7'b001_0010;
    3: seg = 7'b000_0110;
    4: seg = 7'b100_1100;
    5: seg = 7'b010_0100;
    6: seg = 7'b010_0000;
    7: seg = 7'b000_1111;
    8: seg = 7'b000_0000;
    9: seg = 7'b000_0100;
    default: seg = 7'bxxx_xxxx;
    endcase
end
```



8位二进制到BCD码的转换

| 操作 | 百位 | 十位 | 个位 | 二进制 | |
|--------|----|---------|---------|---------|---------|
| 十六进制 | | | | A | B |
| 开始 | | | | 1 0 1 0 | 1 0 1 1 |
| 移位3次 | | | 1 0 1 | 0 1 0 1 | 1 |
| 加3 | | | 1 0 0 0 | 0 1 0 1 | 1 |
| 移位4 | | 1 | 0 0 0 0 | 1 0 1 1 | |
| 移位5 | | 1 0 | 0 0 0 1 | 0 1 1 | |
| 移位6 | | 1 0 0 | 0 0 1 0 | 1 1 | |
| 移位 | | 1 0 0 0 | 0 1 0 1 | 1 | |
| 加3 | | 1 0 1 1 | 1 0 0 0 | 1 | |
| 移位 | 1 | 0 1 1 1 | 0 0 0 1 | | |
| 结束 BCD | 1 | 7 | 1 | | |

思考：BCD码如何转二进制？其它特殊进制的算法呢？

二进制到BCD码的转换模块

```
module binary2BCD(  
    input logic [13:0] binary,  
    output logic [3:0] thousands,  
    output logic [3:0] hundreds,  
    output logic [3:0] tens,  
    output logic [3:0] ones  
);  
  
    logic [29:0] shifter;  
    integer i;  
  
    always_comb  
    begin  
        shifter[13:0] = binary;  
        shifter[29:14] = 0;  
  
        for (i = 0; i < 14; i = i+1) begin  
            if (shifter[17:14] >= 5)  
                shifter[17:14] = shifter[17:14] + 3;  
            if (shifter[21:18] >= 5)  
                shifter[21:18] = shifter[21:18] + 3;  
            if (shifter[25:22] >= 5)  
                shifter[25:22] = shifter[25:22] + 3;  
            if (shifter[29:26] >= 5)  
                shifter[29:26] = shifter[29:26] + 3;  
            shifter = shifter << 1;  
        end // for  
        thousands = shifter[29:26];  
        hundreds = shifter[25:22];  
        tens = shifter[21:18];  
        ones = shifter[17:14];  
    end // always_comb  
endmodule
```



北京大学

Vivado IP集成设计教程

- (一) IP核封装
- (二) IP核管理
- (三) 原理图设计

参考：《Vivado IP集成器设计环境》



(一) IP封装 1. 创建工程文件

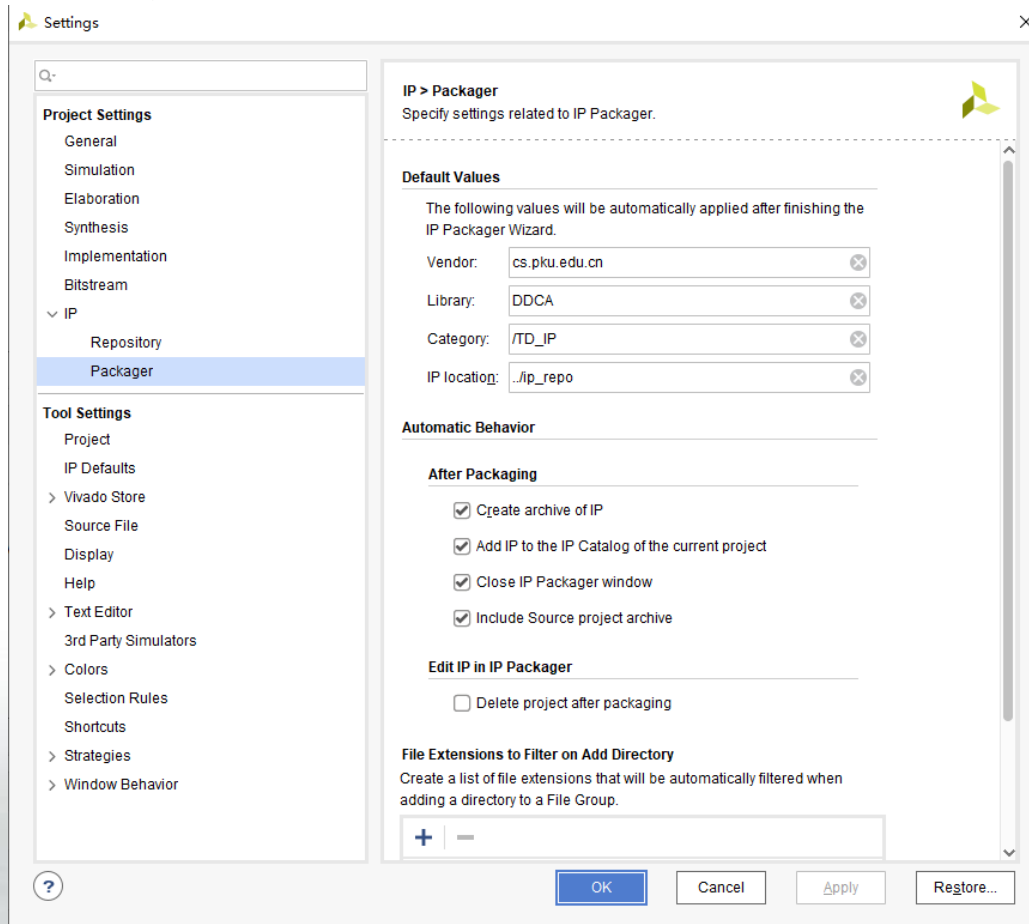
1. 打开Vivado设计开发软件，点击“Create New Project”选项，选择创建新的工程项目，点击“Next”按钮，开始创建工程项目。
2. 在“Project name”和“Project location”中修改工程名称和工程存放路径。工程名：clk_div；路径名ddca_lab_<姓名缩写>。同时勾选“Create project subdirectory”复选框，点击“Next”按钮。
3. 勾选“RTL Project”，勾选“Do not specify sources at this time”，点击“Next”按钮。
4. 在“Default Part”界面中，选择Boards->Basys3或者Parts->xc7a35tcpg236-1，点击“Next”按钮。
5. 在“New Project Summary”界面中，点击“Finish”按钮，完成新的工程项目的创建。

(一) IP封装 2. 添加设计文件并综合

1. 在Vivado左侧的“Flow Navigator”流程处理窗口中的“PROJECT MANAGER”选项中，展开并点击“Add Sources”选项。选择“Add or create design sources”选项，点击“Next”按钮。
2. 在“Add or Create Design Sources”界面中，点击“Create Files”按钮，新建clk_div.sv文件，选择SystemVerilog。点击“OK”按钮。
3. 在Vivado中间“Sources”窗口，“Design Sources”下，双击clk_div，打开编辑窗口。输入clk_div模块的源程序代码。
4. 在Vivado左侧的“Flow Navigator”流程处理窗口中的“SYNTHESIS”选项中，展开并点击“Run Synthesis”进行综合。在设计无误的情况下，会弹出“Synthesis successfully completed”窗口，点击“Cancel”按钮退出。

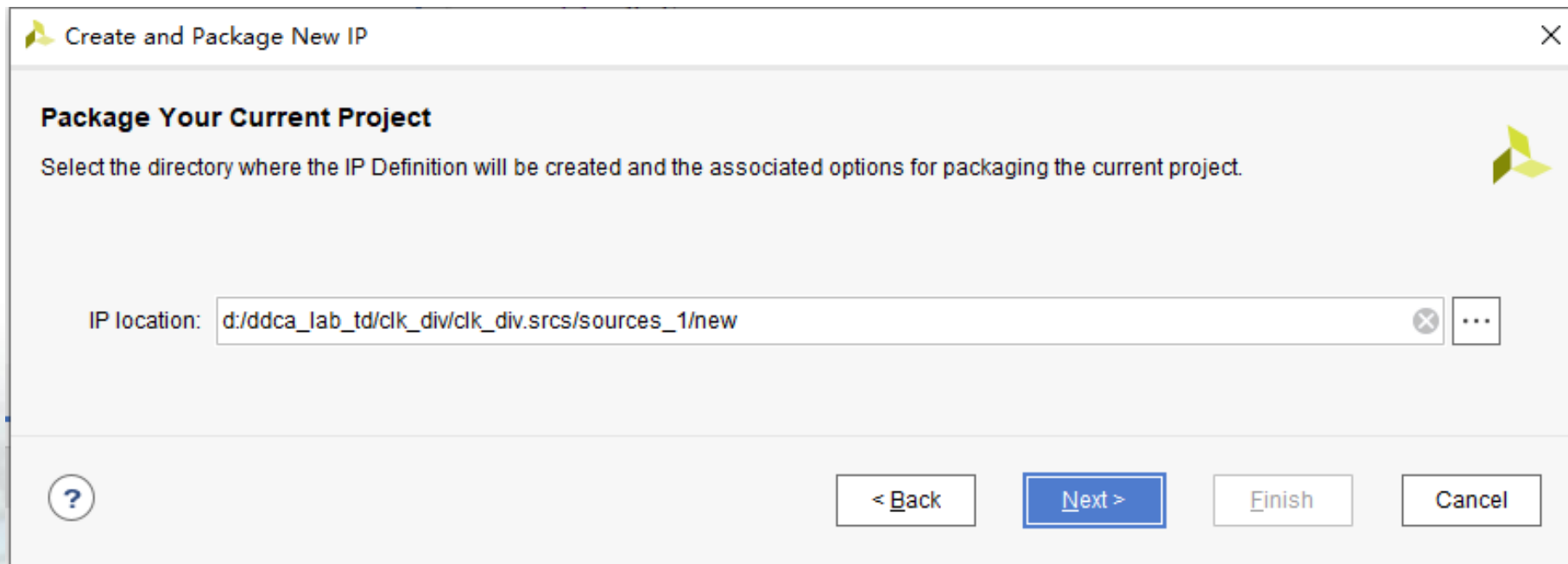
(一) IP封装 3. 设置定制IP属性

1. 在Vivado左侧的“Flow Navigator”流程处理窗口中的“PROJECT MANAGER”选项中，展开并点击“Setting”选项。
2. 在“Setting”对话框中展开“IP”选项并选择“Packager”。在Packager对话框中设定IP库名和目录。
 1. Vendor: cs.pku.edu.cn
 2. Library: DDCA
 3. Category: <姓名缩写>_IP
 4. IP location: <默认>
 5. 复选框如图所示
3. 点击“OK”完成设置。



(一) IP封装 4. 封装IP

1. 在Vivado当前工程主界面主菜单下，选择“Tool”菜单，选择“Create and IP Package...”选项。点击“Next”按钮进入下一步。
2. 进入“Chose Create Peripheral or Package IP”界面，选择“Package your current project”选项。点击“Next”按钮进入下一步。
3. 在“Package Your Current Project”界面中，可在“IP location”中数据IP存放位置，本教程选择<默认目录>。点击“Next”按钮进入下一步。点击“Finish”结束IP封装。



(一) IP封装 4. 编辑封装后IP

1. 在Vivado左侧的“Flow Navigator”流程处理窗口中的“PROJECT MANAGER”选项中，点击“Edit Packaged IP”，出现“Pagckaged IP”窗口。

Project Summary x clk_div.sv x Package IP - clk_div x

Packaging Steps

- ✓ Identification
- ✓ Compatibility
- ✓ File Groups
- ✎ Customization Parameters
- ✎ Ports and Interfaces
- Addressing and Memory
- ✎ Customization GUI
- Review and Package

Identification

Vendor: cs.pku.edu.cn

Library: DDCA

Name: clk_div

Version: 1.0

Display name: clk_div_v1_0

Description: clk_div_v1_0

Vendor display name:

Company url:

Root directory: d:/ddca_lab_td/clk_div/clk_div.srcs/sources_1/new

Xml file name: d:/ddca_lab_td/clk_div/clk_div.srcs/sources_1/new/component.xml

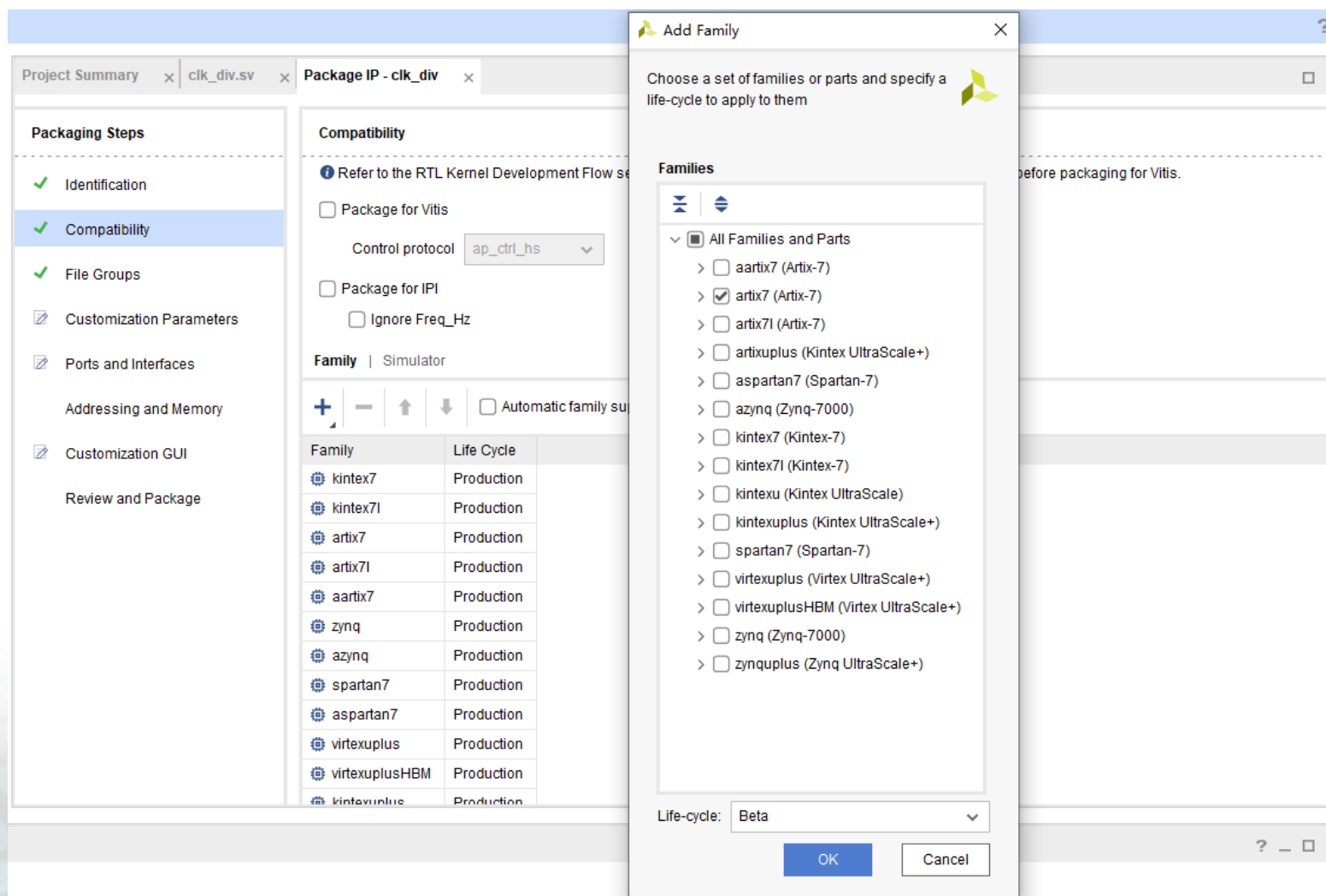
Categories

+ - ↑ ↓

/TD_IP

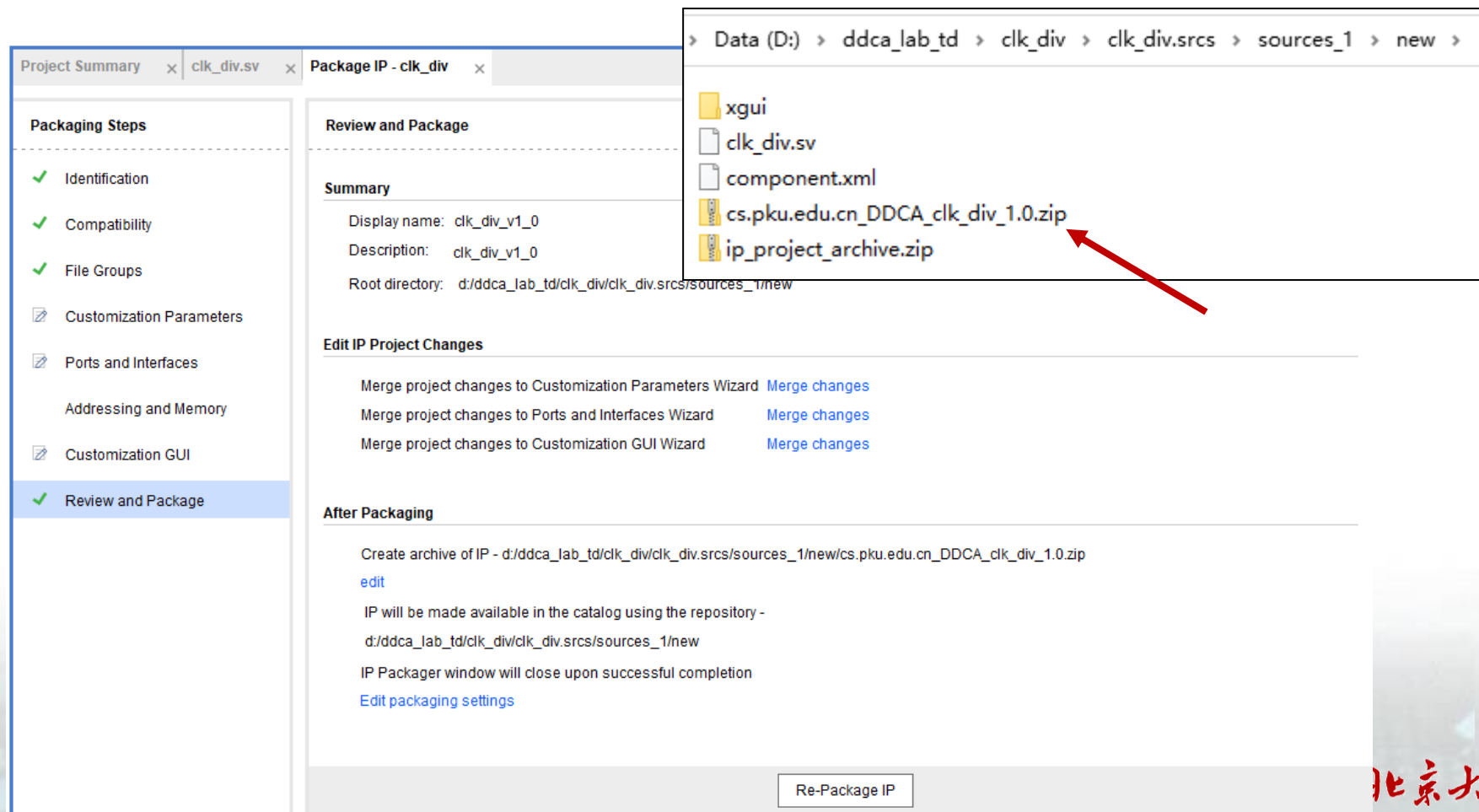
(一) IP封装 4. 编辑封装后IP

1. 在“Identification”对话框，本教程选择<默认>
2. 在“Compatibility”对话框，点击“+”添加aritx7系列芯片。



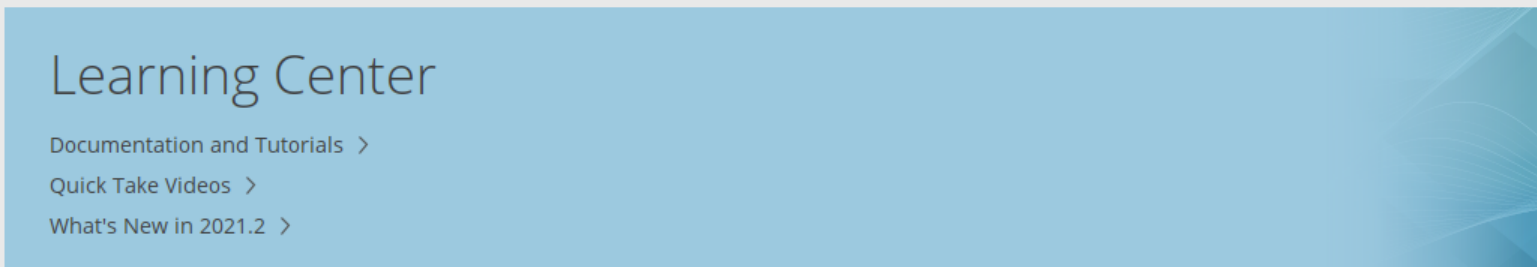
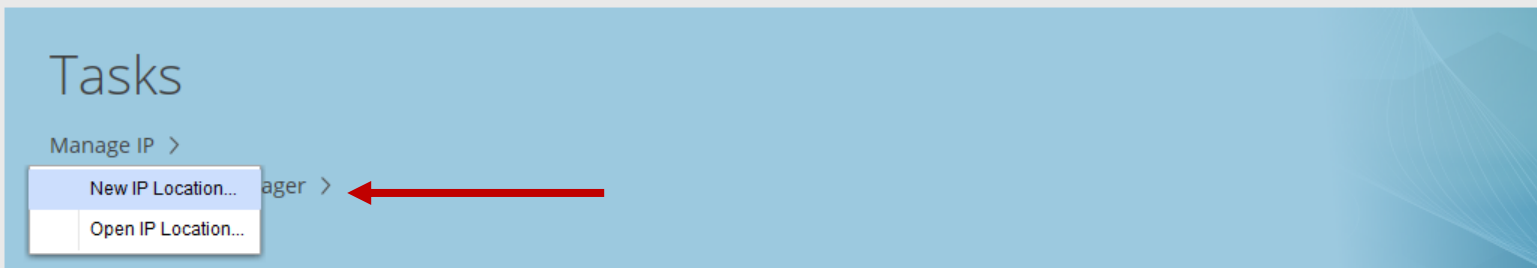
(一) IP封装 4. 编辑封装后IP

1. 在“Review and Package”对话框，点击“Package IP”按钮。
2. 每次修改后，必须重新封装，点击“Re-Package IP”
3. 关闭项目退出，记住IP封装.zip文件的位置。



(二) IP管理 1. 创建IP库

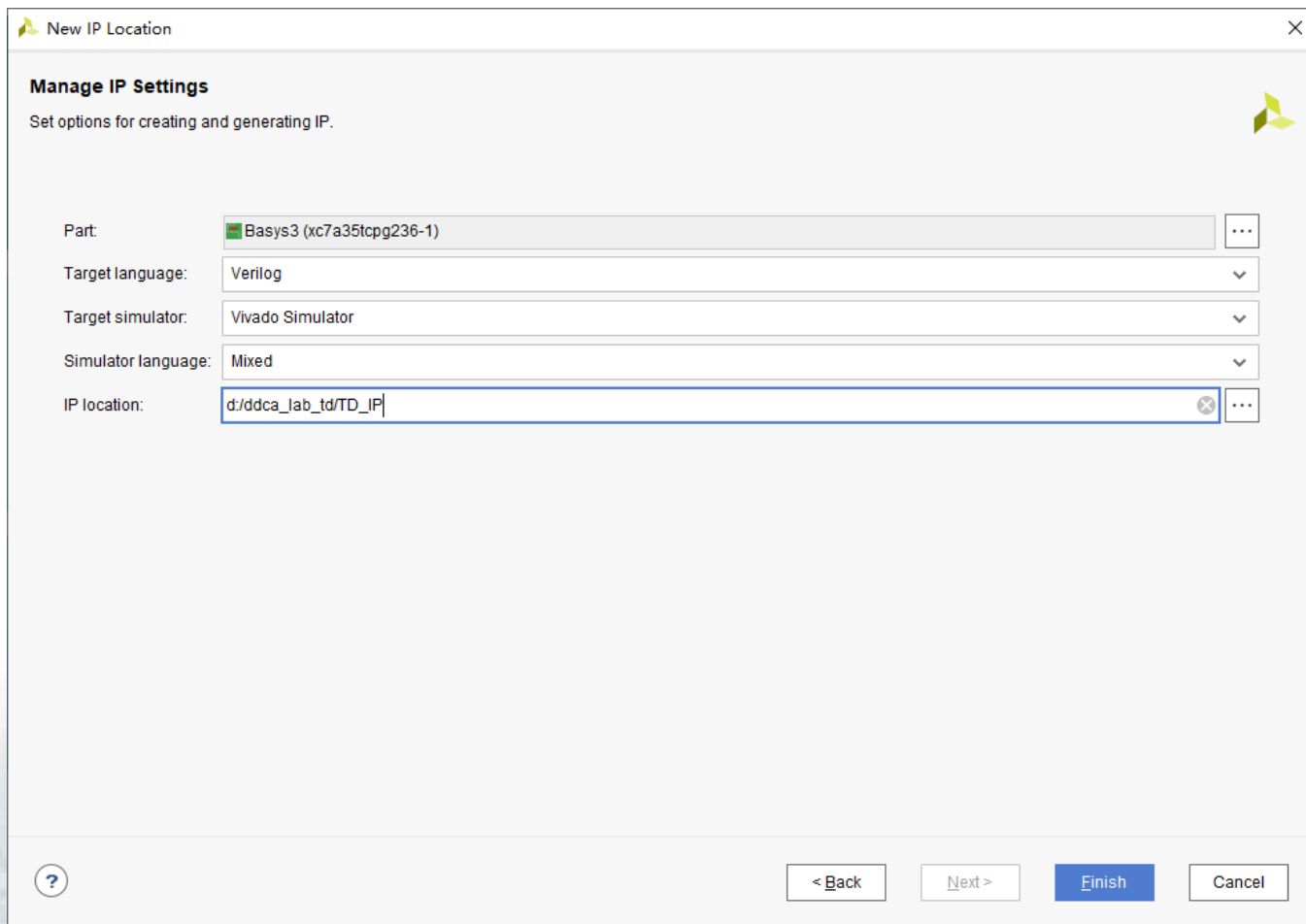
1. 打开Vivado设计开发软件，点击“Manage IP”选项，选择“New IP Location...”，创建IP库。



(二) IP管理 2. 设置IP库

1. 在“Manage IP Setting”对话框中，填写相应的信息。

1. Part选择Boards->Basys3，或者Parts->xc7a35tcpg236-1
2. IP Location，在个人目录下创建<姓名缩写>_IP目录
3. 点击“Finish”



New IP Location

Manage IP Settings
Set options for creating and generating IP.

Part: Basys3 (xc7a35tcpg236-1) ...

Target language: Verilog

Target simulator: Vivado Simulator

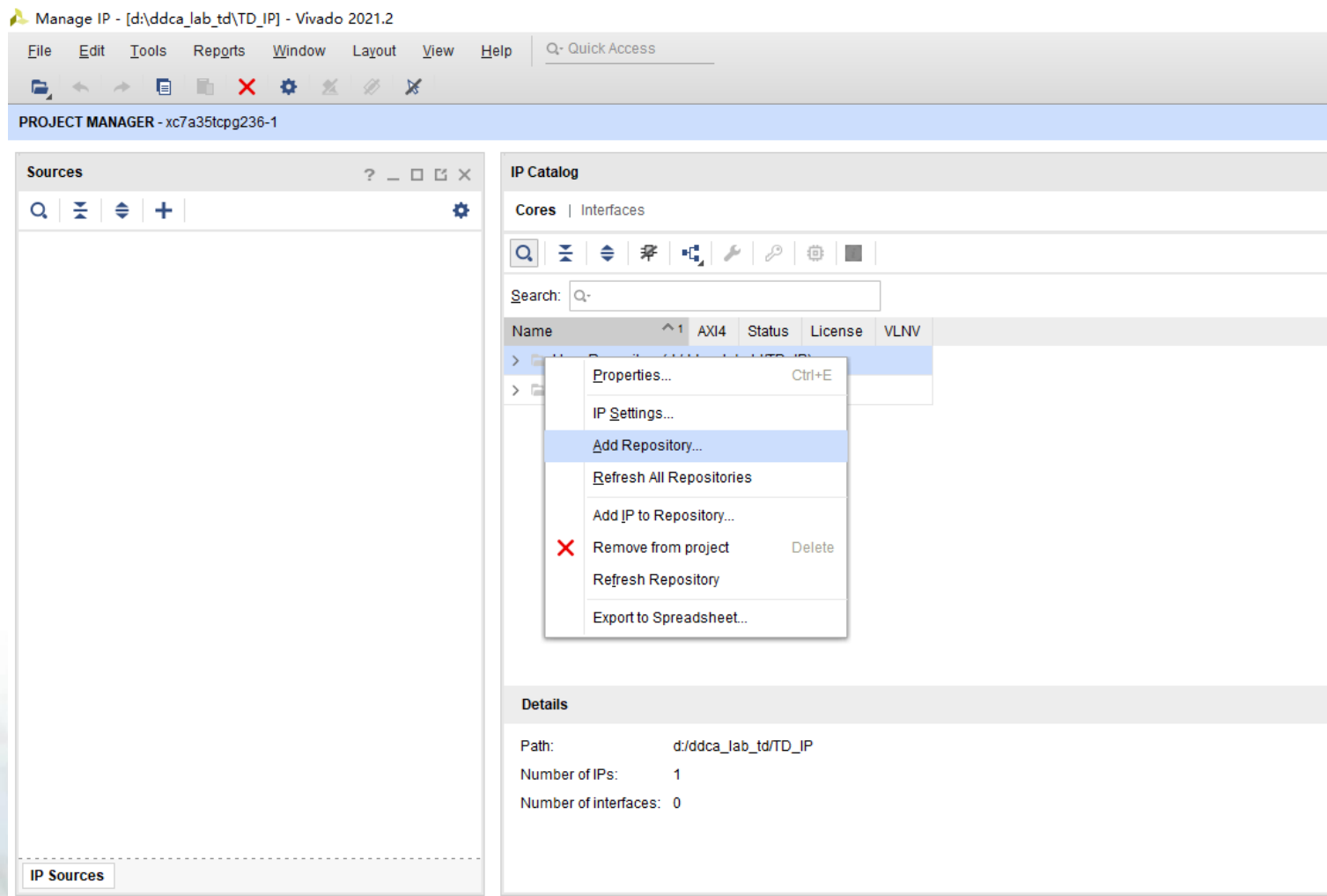
Simulator language: Mixed

IP location: d:/ddca_lab_td/TD_IP

< Back Next > Finish Cancel

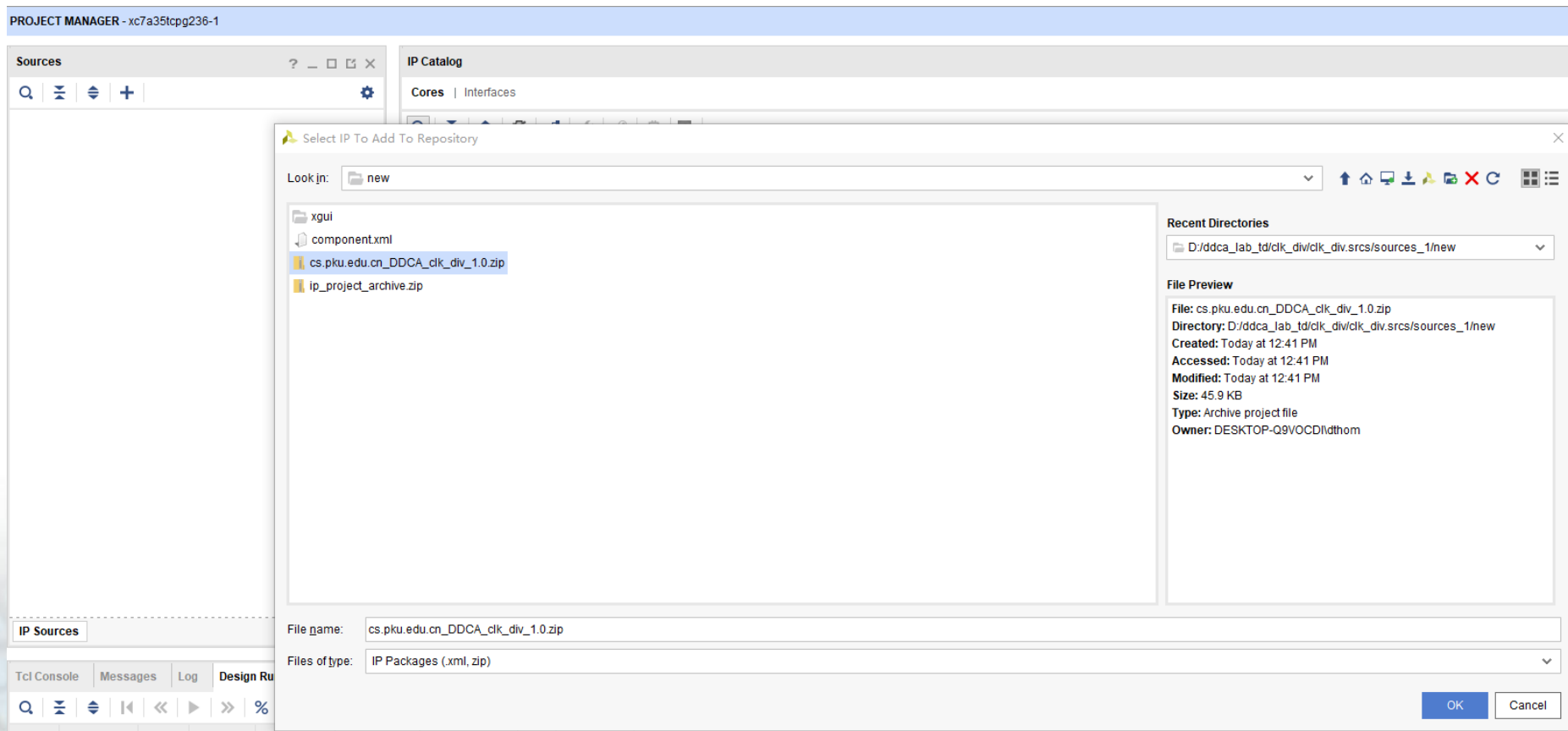
(二) IP管理 3. 添加自定义仓库

1. 在“IP Catalog”界面中，点击右键选择“Add Repository”，选择之前建好的IP目录， <姓名缩写>_IP



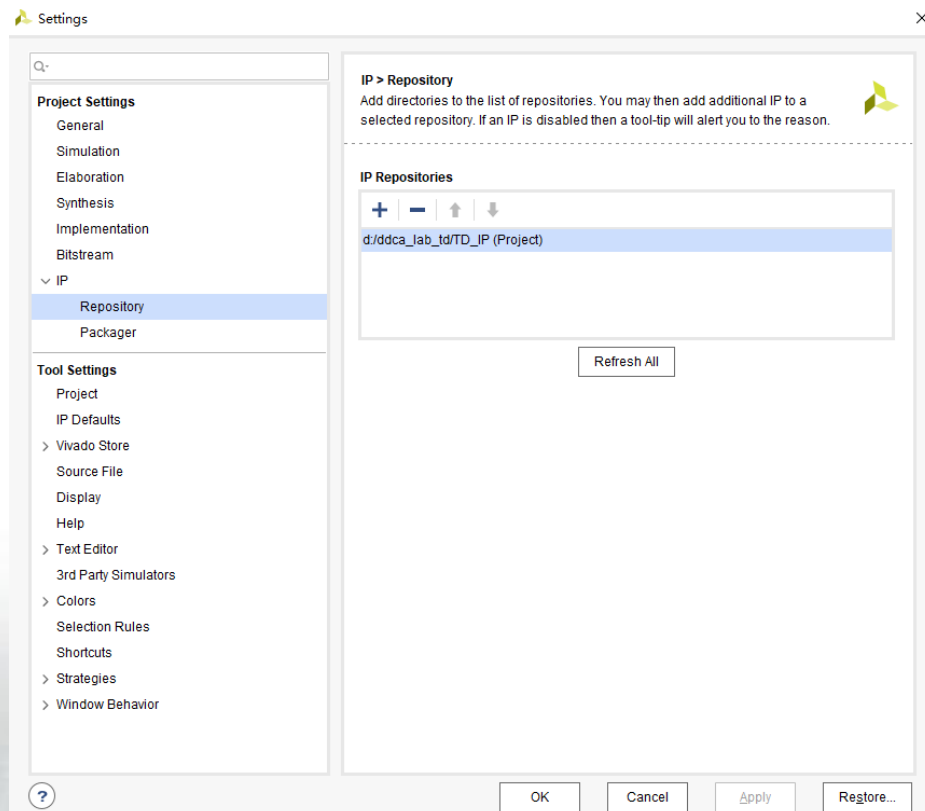
(二) IP管理 4. 添加自定义IP

1. 在“IP Catalog”界面中，点击右键选择“Add IP to Repository”，选择第一阶段生成好的封装后IP压缩文件，如下图所示。点击“OK”结束。关闭工程项目结束。
2. 反复进行上述步骤，生成display_7seg_x4的IP封装加入同一个IP库中。



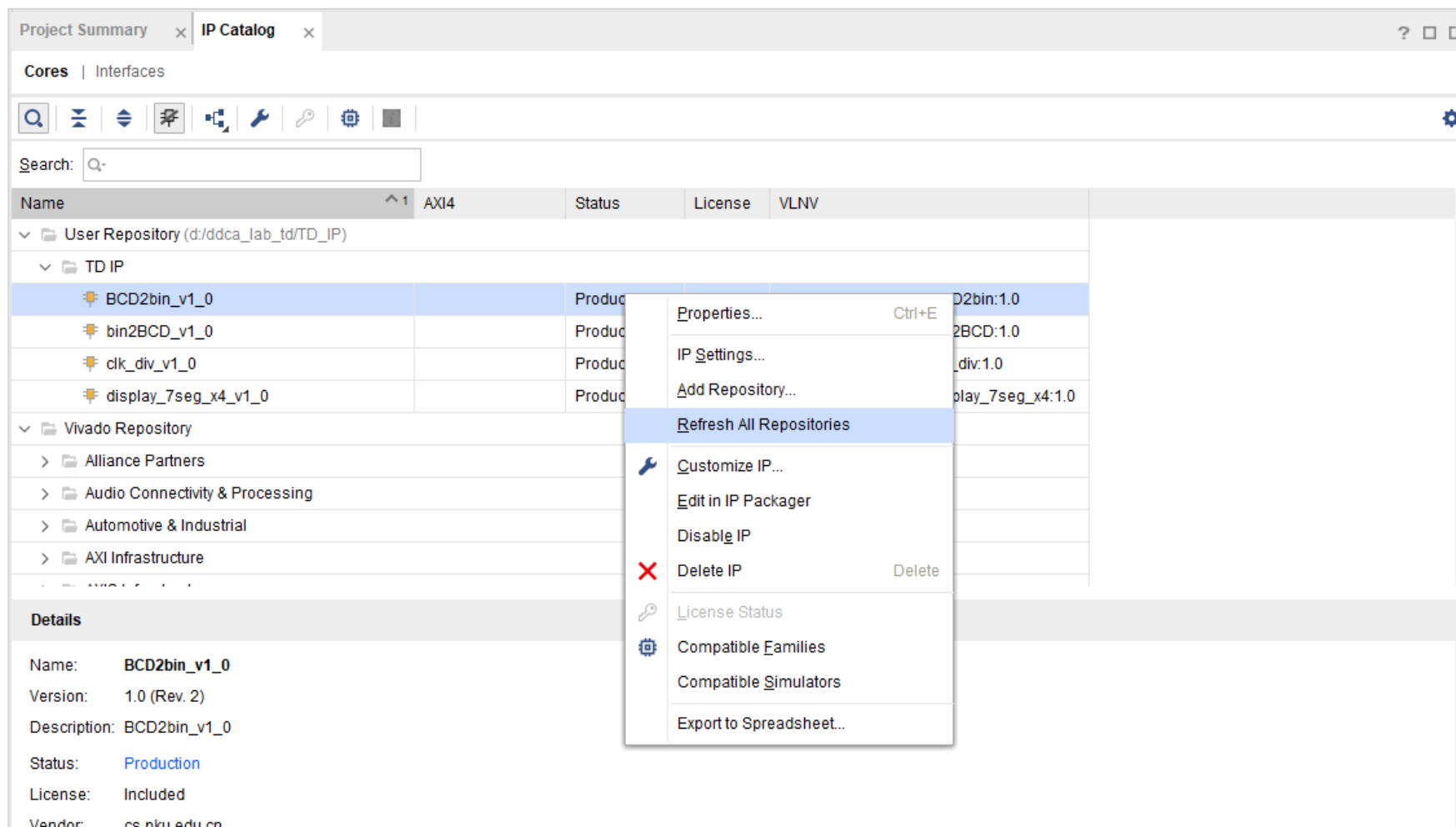
(三) 原理图设计 1. 添加IP仓库

1. 创建工程项目display_led
2. 在Vivado左侧的“Flow Navigator”流程处理窗口中的“PROJECT MANAGER”选项中，展开并点击“Setting”选项。
3. 在“Setting”对话框中展开“IP”选项并选择“Repository”。
4. 在IP->Repository对话框中，点击“+”加入上一步创建的IP库目录。
<姓名缩写>_IP。
5. 点击“OK”完成设置。
6. 如果任何IP有修改，必须点击“Refresh All”按钮更新!!!



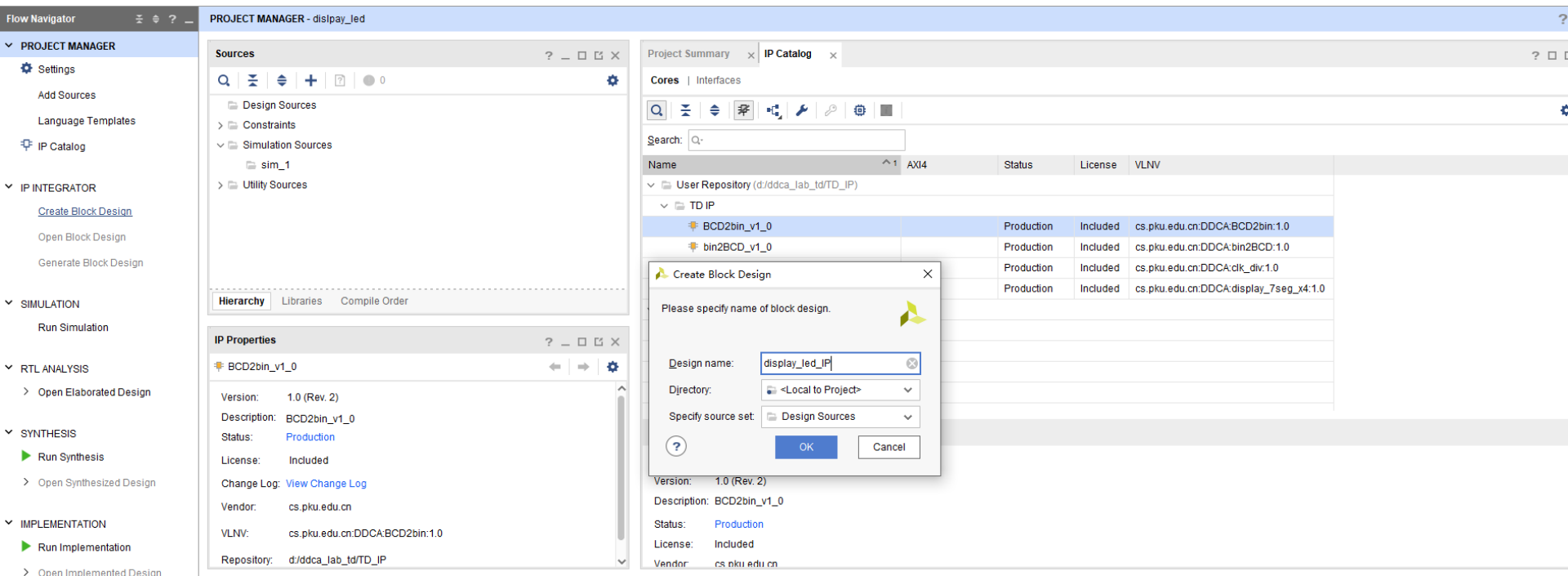
(三) 原理图设计 2. 查看IP仓库

1. 点击Vivado左侧的“IP Catalog”选项，可在右侧“IP Catalog”窗口中看到封装好的IP。
2. 如果任何IP有更新，应该点击右键选择“Refresh ALL Repositories”



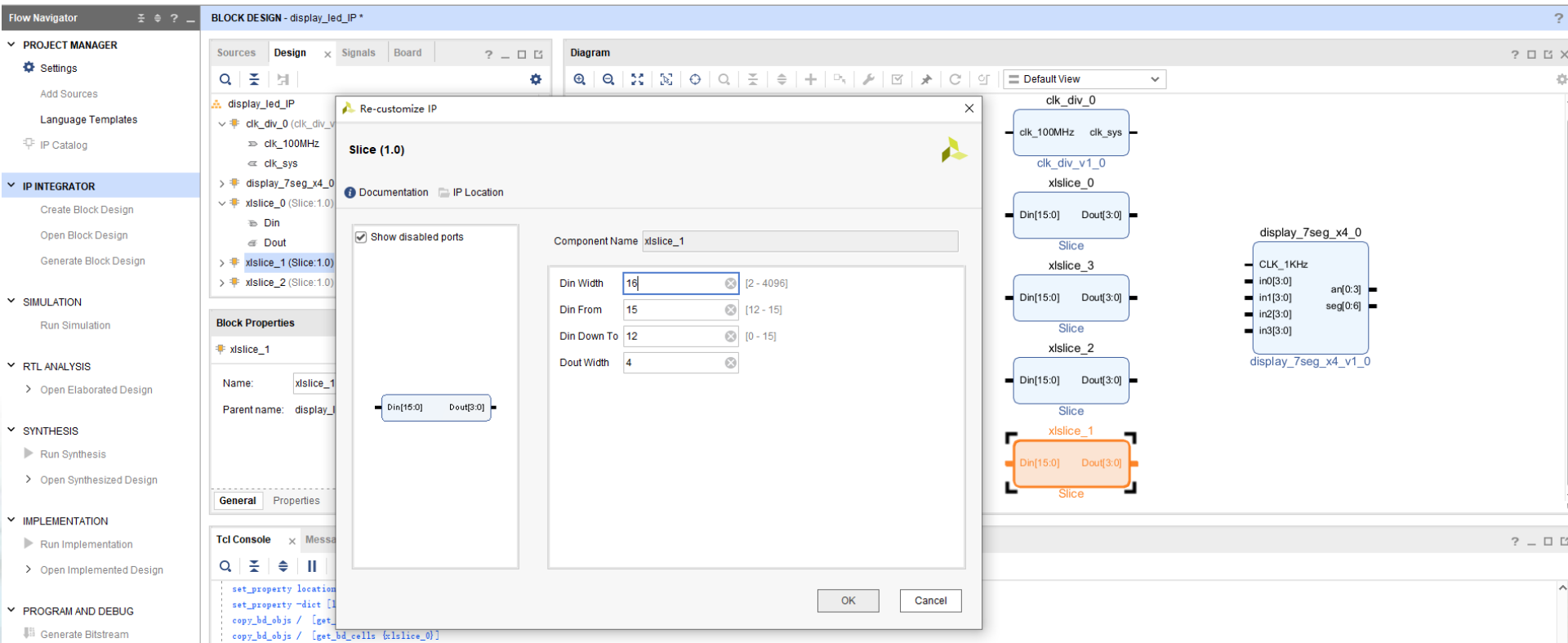
(三) 原理图设计 3. 创建原理图文件

1. 在Flow Navigator中展开IP Integrator，点击“Create Block Design”，将设计命名位“display_led_IP”，点击OK完成创建。
2. 在Diagram界面，点击“+”添加IP，可在搜索栏Search输入“clk”，选择自定义的IP核“clk_div_v1_0”，按回车键添加
3. 同理添加“display_7seg_x4” IP核到原理图中。



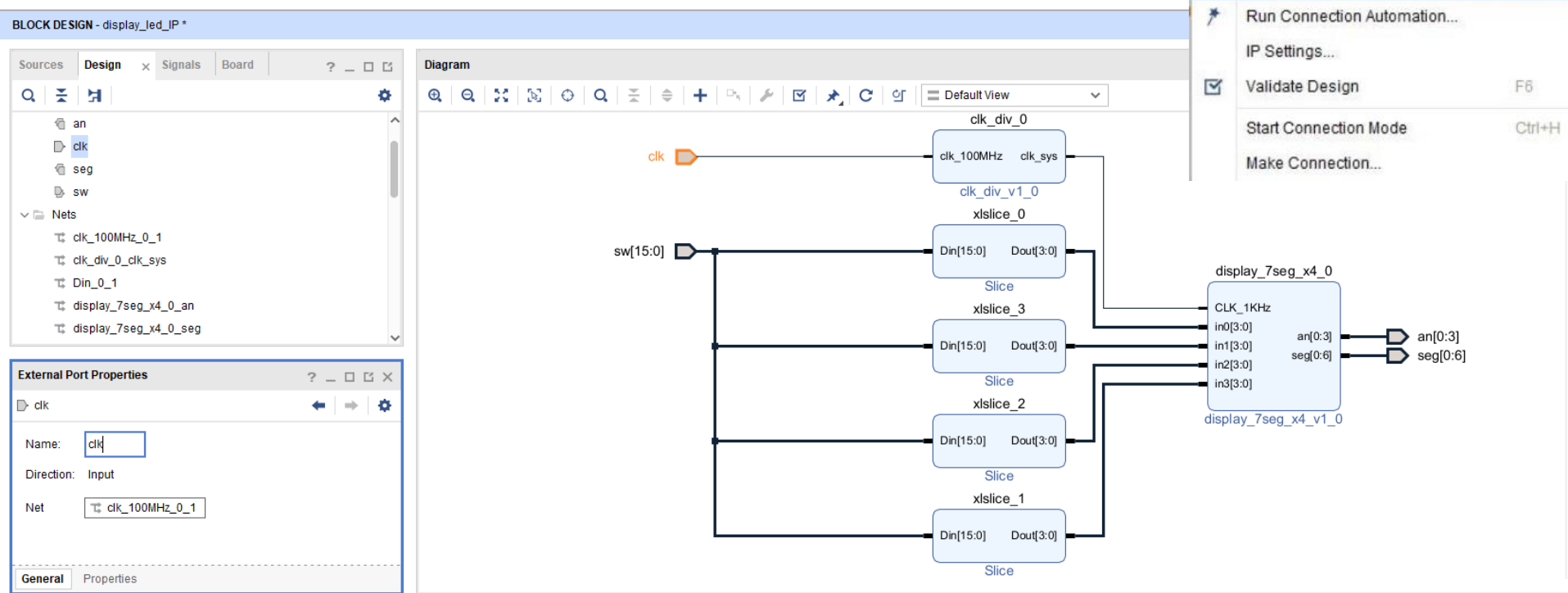
(三) 原理图设计 4. 添加分线器Slice

1. 添加4个分线器Slice IP，双击每个Slice方框，将输入Din With设为16，输出Dout from 和 Dout downto 分别设为(3,0), (7, 4), (11,8), (15,12)。Dout Width自动计算。这些slice分别对应display_7seg_x4的in0, in1, in2, in3四个端口。
2. 这一步将输入的16位总线信号，拆分成4个4位的总线信号。
3. 与Slice相对应是集线器Concat，可以将多个信号集合成一个总线信号。



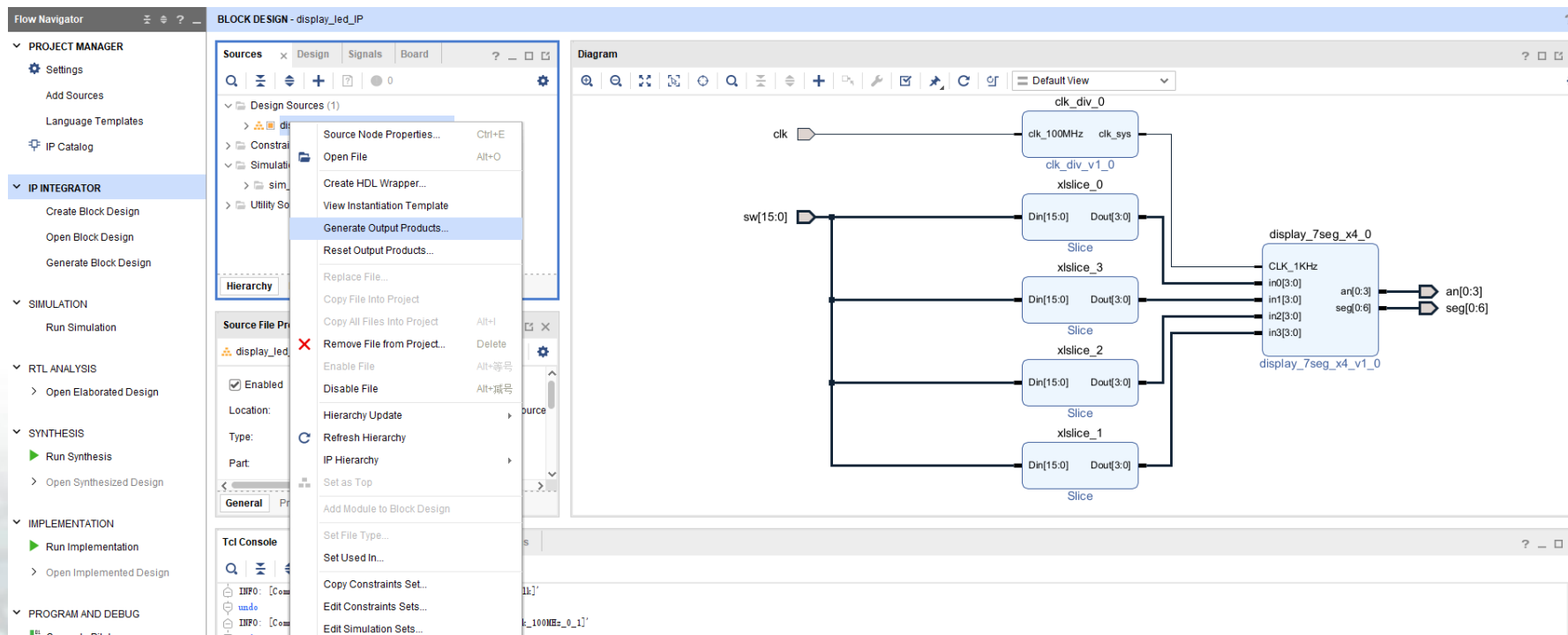
(三) 原理图设计 5. 添加分线器Slice

1. 点击模块端口，拖动鼠标与相应的端口连线。
2. 点击模块端口，点击鼠标右键，选择“Make External”，生成原理图对外端口。
3. 点击外部端口，在左侧的External Port Properties窗口中修改名称为Basys3的相应端口名。输入clk和sw，输出an和seg。
4. 双击clk_div，修改分频参数“10000”，时钟周期1ms



(三) 原理图设计 6. 生成项目文件

1. 在中间Sources项中，展开Design Sources，右键点击“display_led_IP”
2. 选择“Generate Output Products”自动生成相关文件。弹出窗口中，在“Synthesis Options”一项中选择“Out of context per IP”，然后点击Generate按钮。
3. 右键点击“display_led_IP”，选择“Create HDL Wrapper”生成顶层SystemVerilog文件。选择“Let Vivado manage wrapper and auto-update”，点击OK按钮。



(三) 原理图设计 7. 综合并生成Bit文件

1. 在Sources项中选中display_led_IP_wrapper，在左侧SYNTHESIS下选择“Run Synthesis”。
2. 在“Add Sources”中添加Constraints文件。
3. 在左侧“PROGRAM AND DEBUG”下选择“Generate Bitstream”。
4. 下载编程FPGA。

