计算机组织与结构(II)
Computer Organization and Architecture

信息科学与工程学院
毫米波国家重点实验室

课程内容

计算机组织与结构

一、数字系统设计的一些概念和技巧

二、VHDL与Verilog语言

三、硬件设计模块的仿真与测试

四、软件开发环境VIVADO

五、硬件平台简介

课程内容

计算机组织与结构

一、数字系统设计的一些概念和技巧

二、VHDL与Verilog语言

三、硬件设计模块的仿真与测试

四、软件开发环境VIVADO

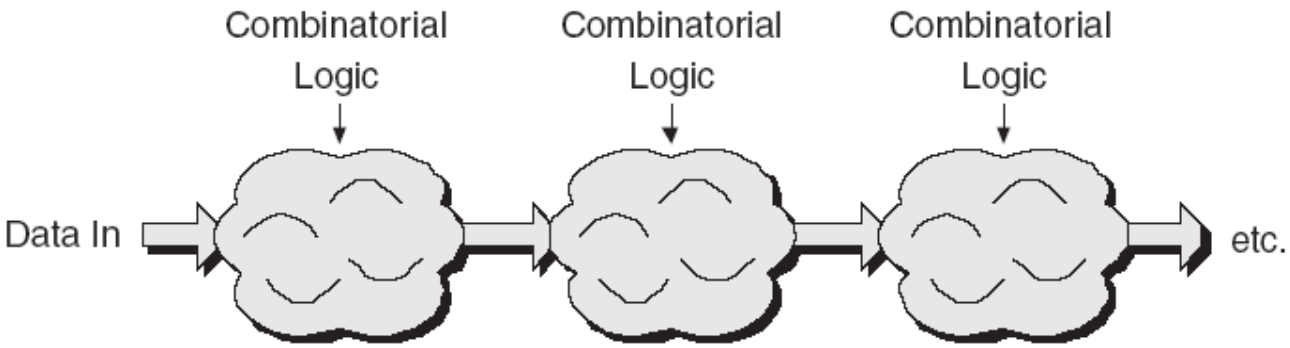
五、硬件平台简介

资源共享

- 资源共享是一种优化技术，该技术使得可以使用单个功能模块（比如说一个加法器或一个比较器）来实现一系列操作。比如说，一个乘法器可能先处理两个值A和B，然后同样是这个乘法器，再来处理C和D。资源共享的典型应用是时分复用（Time-Division Multiplexing－TDM）。

流水线设计（Pipeline）

- 假设我们要设计的电路或电路的一部分可以由一些组合逻辑块串联实现，如图表示。
- 假设每个组合逻辑块需要使用N纳秒来实现其功能，而我们假设现在有3个这样的组合逻辑块，那么从数据进入第一个块到最后输出，共花费3×N纳秒的时间。

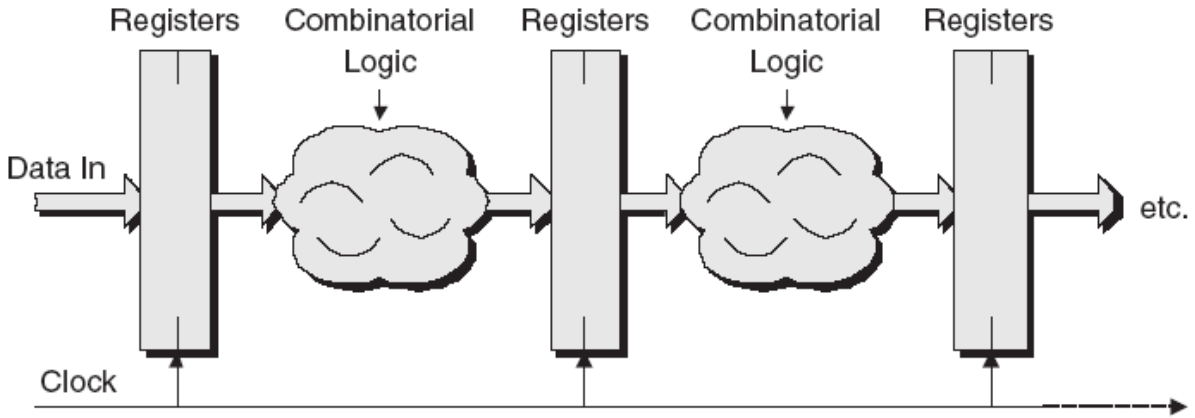


流水线设计（Pipeline）

- 通常情况下，组合电路中，在我们得到有效的输出数据之前，无法输入新的数据。这样带来的结果是每个组合逻辑块在一次数据处理过程中（3×N纳秒）只有N纳秒在有效工作。那么如何每个块都充分的利用起来呢？答案就是使用流水线操作技术，将逻辑块与块之间用寄存器进行隔离，以提高效率和速度。

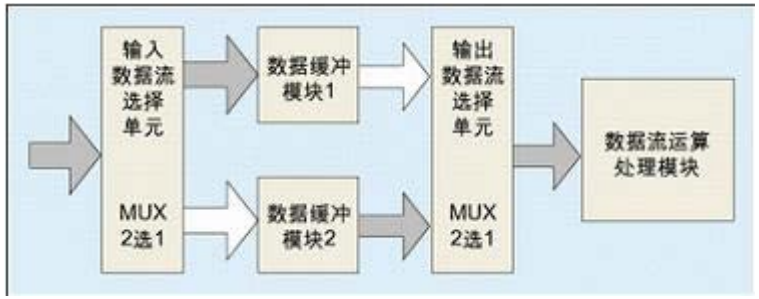
流水线设计（Pipeline）

- 所有寄存器使用共同的时钟信号，在每一个有效的时钟边沿时刻，寄存器的值都要更新一次，更新的值是该寄存器前一级的组合电路的输出。于是这些值被逐级传输直到输出。在这种情况下，一旦“水泵中的水被抽满”，流水线饱和运作，处理一个数据的时间将变为N纳秒。



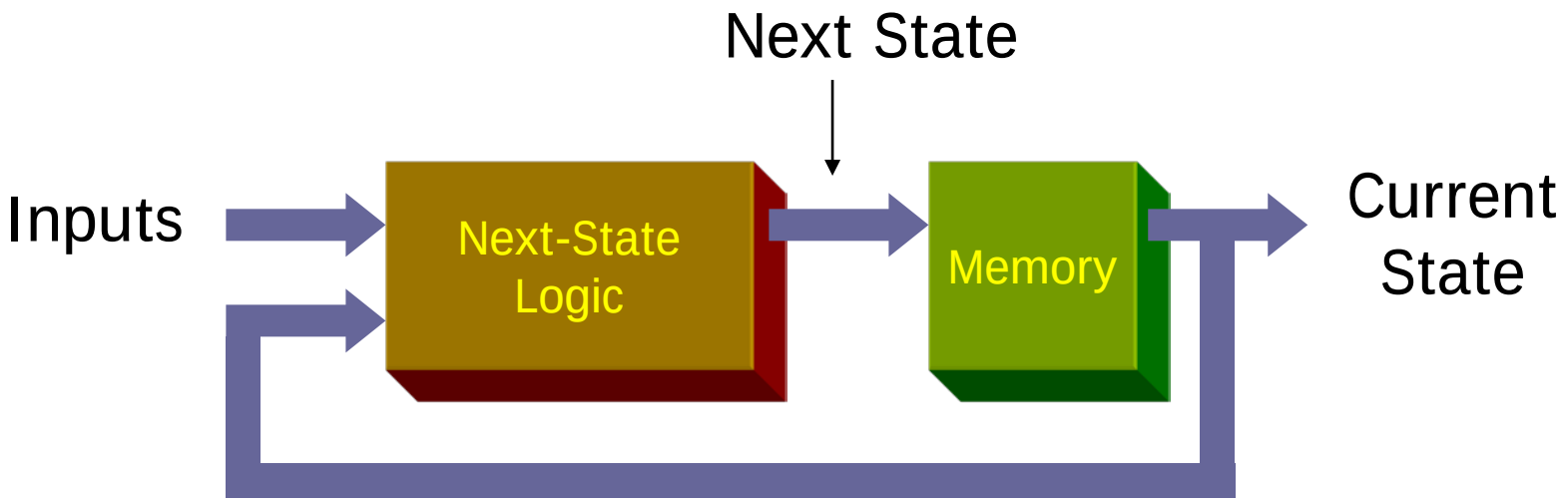
乒乓操作

- 在第一个缓冲周期，将输入的数据流缓存到“数据缓冲模块1”；在第2个缓冲周期，通过“输入数据选择单元”的切换，将输入的数据流缓存到“数据缓冲模块2”，同时将“数据缓冲模块1”缓存的第1个周期数据通过“输出数据选择单元”的选择，送到 “数据流运算处理模块”进行运算处理；在第3个缓冲周期通过“输入数据选择单元”的再次切换，将输入的数据流缓存到“数据缓冲模块1”，同时将“数据缓冲模块2”缓存的第2个周期的数据通过“输出数据选择单元”切换，送到“数据流运算处理模块”进行运算处理。如此循环。



时序逻辑-状态机

- All state machines have the general feedback structure consisting of:
  - Combinational logic implements the next state logic
    - Next state (ns) of the machine is formed from the current state (cs) and the current inputs
  - State register holds the value of current state



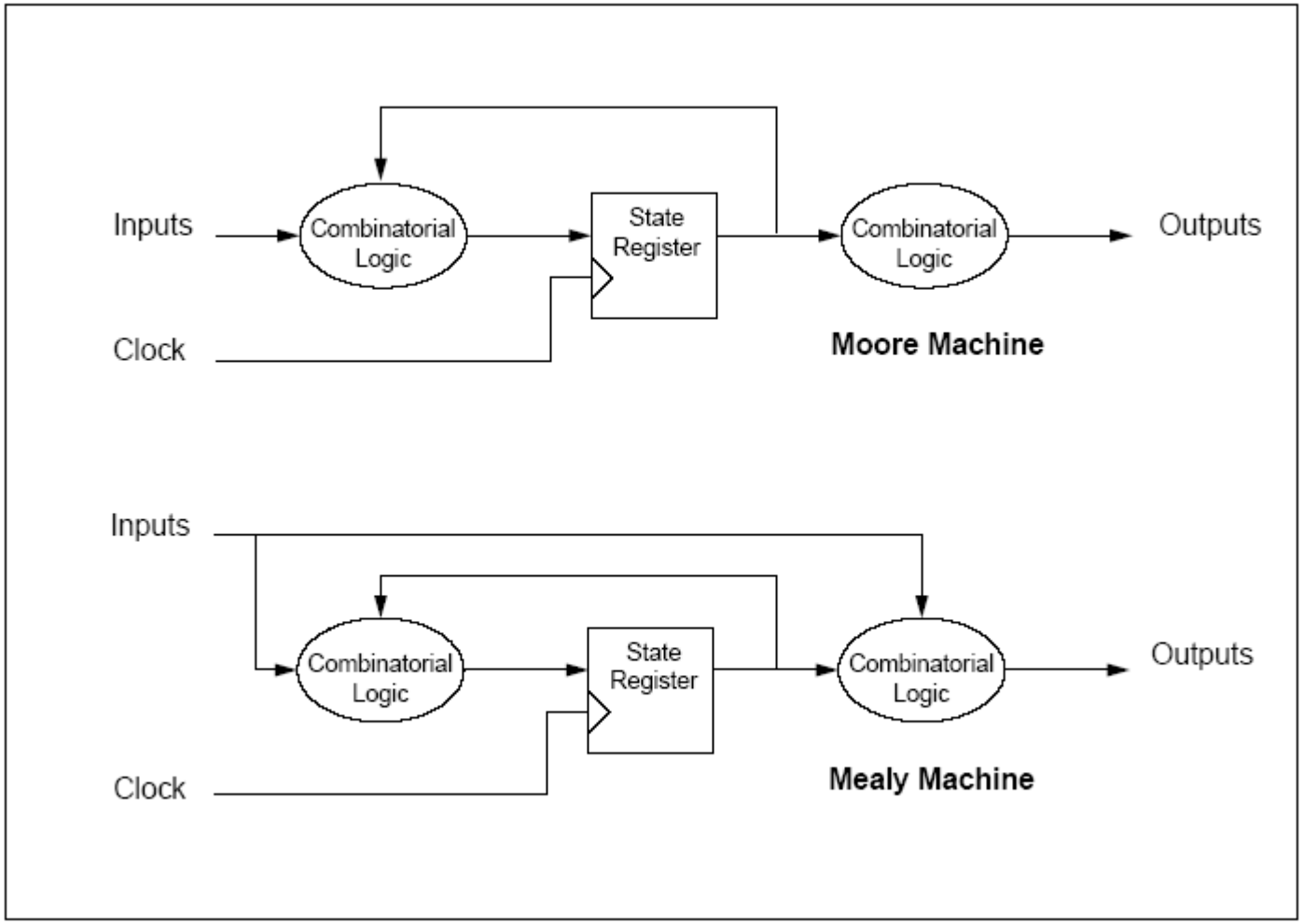
状态机设计

- 状态机是大型电子设计的基础，通常用状态机来实现数字系统的控制器.
- 最基本的两种状态机方式:
  - Moore型

    较简单的一种状态机，输出仅是当前状态的函数。
  - Mealy型

    输出是当前状态和输入状态的函数.

Moore *VS* Mealy



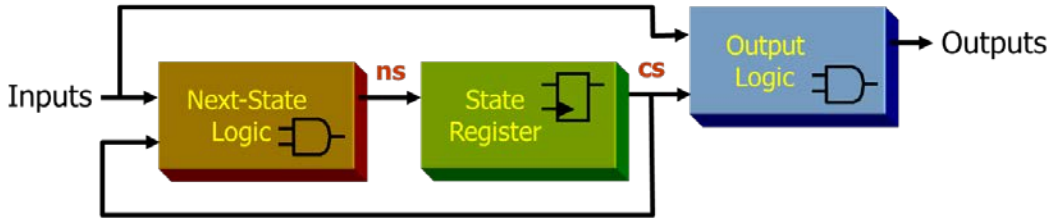
时序逻辑-Moore/Mealy状态机

计算机组织与结构

Moore State Machine

- Next state depends on the current state and the inputs but the output depends only on the present state
  - next_state(t) = h(current_state(t), input(t))
  - output = g(current_state(t))

Mealy State Machine



- Next state and the outputs depend on the current state and the inputs
  - next_state(t) = h(current_state(t), input(t))
  - output(t) = g(current_state(t), input(t))

课程内容

计算机组织与结构

一、数字系统设计的一些概念和技巧

二、VHDL与Verilog语言

三、硬件设计模块的仿真与测试

四、软件开发环境VIVADO

五、硬件平台简介

Verilog与VHDL

- Verilog HDL和VHDL是目前世界上最流行的两种硬件描述语言，均为IEEE标准，被广泛地应用于基于可编程逻辑器件的项目开发。二者都是在20世纪80年代中期开发出来的，前者由Gateway Design Automation公司（该公司于1989年被Cadence公司收购）开发，后者由美国军方研发。

- **Verilog与VHDL比较**
  - VHDL的设计之初就更加针对标准化进行设计，Verilog则具有简明、高效的代码风格
  - Verilog的逻辑门级、开关级电路描述能力更强，VHDL的系统级抽象描述能力则比Verilog强。

VHDL简介

- **V**ery high speed integrated circuit **H**ardware **D**escription **L**anguage
- 1987底,VHDL被IEEE和美国国防部确认为标准硬件描述语言。此后VHDL在电子设计领域得到了广泛的接受,并逐步取代了原有的非标准的硬件描述语言。现在,VHDL和Verilog作为IEEE的工业标准硬件描述语言,又得到众多EDA公司的支持,在电子工程领域,已成为事实上的通用硬件描述语言。

VHDL基本结构

- 库 (LIBRARY)
  用来存储可编译的设计单元

- 实体 (ENTITY)
  描述设计模块的输入输出端口类型

- 结构体 (ARCHITECTURE)
  描述电路的具体功能

VHDL基本结构—库

- 库调用语句放在程序的最前面，最常用的库调用语句为:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
```

VHDL基本结构—实体

- 实体定义设计的全部输入输出信号.
  格式如下:

```
ENTITY 实体名 IS
PORT
 (输入输出信号列表);
END 实体名;
```

VHDL基本结构—实体

- 一个计数器的实体部分

```
ENTITY count_m16 IS
PORT(
reset  : IN      std_logic;
clk    : IN      std_logic;
co     : OUT   std_logic;
count :BUFFER std_logic_vector(3 DOWNTO 0));
END count_m16;
```

VHDL基本结构—实体

- 端口模式
  1. IN　　输入信号，不能给输入端口赋值
  2. OUT　　输出信号，不能在内部反馈使用
  3. INOUT　双向信号
  4. BUFFER　输出信号，可在内部反馈

- 数据类型

  std_logic : 0(0)，1(1)，Z(高阻态)，X(不定态).

  std_logic_vector : std_logic 的矢量形式

VHDL基本结构—结构体

- 结构体描述实体的结构或行为，格式为：

ARCHITECTURE 结构体名 OF 实体名 IS

定义语句｛内部信号、常数、数据类型、函数｝

BEGIN

并行处理语句;

进程语句(PROCESS);

END 结构体名;

VHDL基本结构—结构体

- 上述计数器的结构体部分

```
ARCHITECTURE behave OF count_m16  IS
BEGIN
    PROCESS (clk)
    BEGIN
        IF (clk'event and clk='1')THEN
            IF (reset='1')THEN count<="0000";co<='0';
              ELSIF (count="1111")THEN
            count<="0000";co<='1';
              ELSE count<=count+1; co<='0';
            END IF;
        END IF;
    END PROCESS;
END behave;
```

VHDL基本结构—结构体

- 一个加法器结构体的例子

```
ARCHITECTURE behave OF adder8 IS
BEGIN
    SUM1<=ADD_A+ADD_B;
    SUM2<=ADD_C+ADD_D;
    SUM3<=ADD_E+ADD_F;
END behave;
```

VHDL基本结构—结构体—数据对象

- **常量**—在设计描述中保持特定值不变。
  CONSTANT 常数名:数据类型:=表达式;
  CONSTANT width : integer:=8;

- **信号**—声明内部信号，在元件间起互连作用.
  SIGNAL 信号名:数据类型;
  SIGNAL a:std_logic_vector(3 downto 0);

- **变量**—用于声明进程或子程序中的局部值;
  VARIABLE 变量名:数据类型;
  VARIABLE a : std_logic;

VHDL基本结构—结构体-信号vs变量

- 信号是全局量,常在结构体中声明:

```
ARCHITECTURE  behave OF Entity_Name IS
SIGNAL  sig_temp: std_logic;
```

- 变量是局部量，常在进程中声明：

```
ARCHITECTURE  behave OF Entity_Name IS
BEGIN
PROCESS (…)
VARIABLE  var_temp: std_logic;
```

VHDL基本结构—结构体-信号vs变量

计算机组织与结构

- 信号赋值的符号为“<=”

  如： SIG_temp <= ‘1’；

- 变量赋值的符号为“:=”

  如：VAR_temp := ‘1’；

VHDL描述方法

- 并发语句(Concurrent Statement)
  - 简单赋值语句
  - 条件赋值语句
  - 选择信号赋值语句
- 顺序语句(Sequential Statement)
  - IF 语句
  - CASE 语句
- 进程语句(Process Statement)

VHDL并发语句

- 所有的并发语句都是并行执行
- 并发语句不关心在程序中的位置
- 并发语句的输出依赖于输入

Output depends on Input only without any conditional constraint

```
Entity test1 Is
Port ( a, b : in bit;
       c, d : out bit);
end test1;
architecture test1_body of test1 is
begin
c <= a and b;
d <= a or b;
end test1_body;
```

```
Entity test1 Is
Port ( a, b : in bit;
       c, d : out bit);
end test1;
architecture test1_body of test1 is
begin
d <= a or b;
c <= a and b;
end test1_body;
```

Does not care the position within the coding

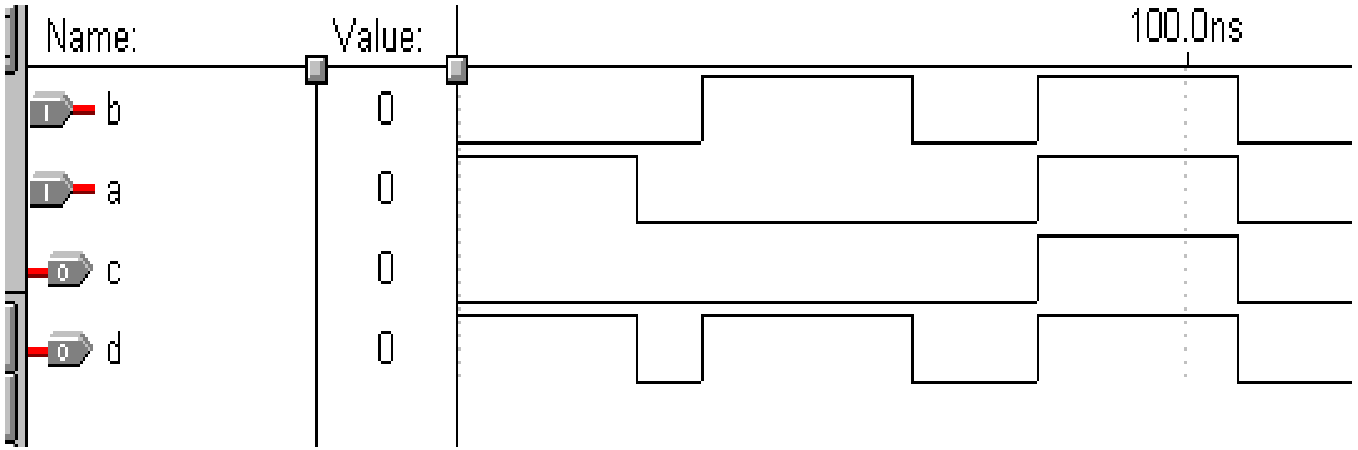
VHDL并发语句

c <= a and b;
d <= a or b;

d <= a or b;
c <= a and b;

C = A and B

D = A OR B



VHDL并发语句

- **简单赋值语句**

赋值目标<=表达式;

……

赋值目标<=表达式;

如 c<=a+b;

c<=a and b;

c<=a;

VHDL并发语句

- 条件赋值语句

目标 <= 表达式1 WHEN 赋值条件1 ELSE
表达式2 WHEN 赋值条件2 ELSE
……
表达式N;

如:y <= a0 WHEN s=“00” ELSE
a1 WHEN s=“01” ELSE
a2 WHEN s=“10” ELSE
a3;

VHDL并发语句

- 选择信号赋值语句

WITH 选择表达式 SELECT
赋值目标信号 <= 表达式 WHEN 选择值,
……
表达式 WHEN 选择值;

如: WITH s SELECT
y<= a0 WHEN “00”,
a1 WHEN “01”,
a2 WHEN “10”,
a3 WHEN OTHERS;

VHDL顺序语句

- IF语句

```
PROCESS (s, a0, a1, a2, a3 )
BEGIN
    IF s="00" THEN y<=a0;
    ELSIF s="01"THEN y<=a1;
    ELSIF s="10"THEN y<=a2;
    ELSE  y<=a3;
    END IF;
END PROCESS;
```

VHDL顺序语句

- **CASE语句**

```
PROCESS (s, a0, a1, a2, a3 )
BEGIN
    CASE s IS
    WHEN “00”   => y<=a0;
    WHEN “01”   => y<=a1;
    WHEN “10”   => y<=a2;
    WHEN others=> y<=a3;
    END CASE;
END PROCESS;
```

VHDL进程语句

- 进程语句由一段程序构成，各个进程之间是并行执行的，进程内部是顺序执行的。一个结构体可以包含多个进程语句。
- PROCESS 必须要有敏感信号表（SENTIVITY LIST），敏感表中的信号变化导致进程触发.

PROCESS (敏感信号表)
[声明区];--此处声明局部变量等.
BEGIN --进程开始
[顺序语句];
END PROCESS;--进程结束

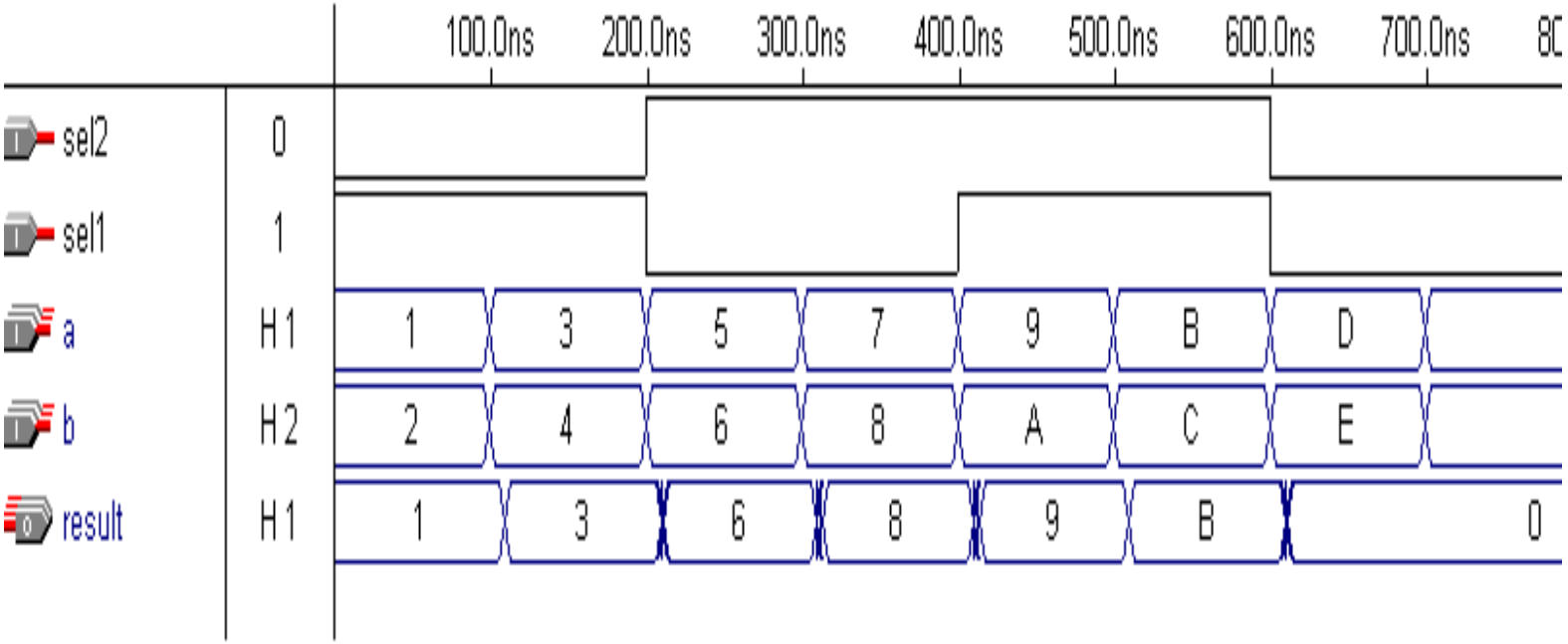
VHDL进程语句（EXAMPLE 1）

```
entity test is
port(a,b      : in std_logic_vector(3 downto 0);
     sel1,sel2: in std_logic;
     result: out std_logic_vector(3 downto 0));
end test;

architecture behave of test is
begin
  process(sel1,sel2,a,b)
   begin
          if(sel1='1') then result<=a;
          elsif(sel2='1') then result<=b;
          else result<="0000";
          end if;
  end process;
end behave;
```

计算机组织与结构

VHDL进程语句（EXAMPLE 1）

计算机组织与结构

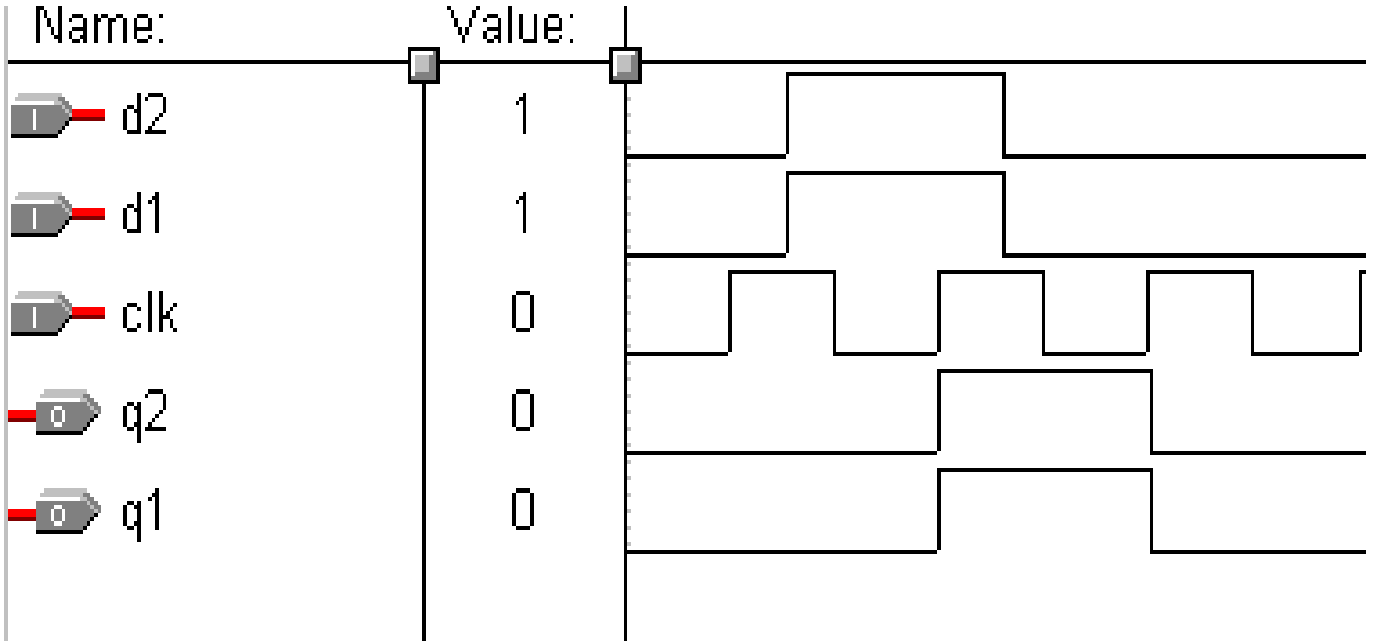


VHDL进程语句（EXAMPLE 2）

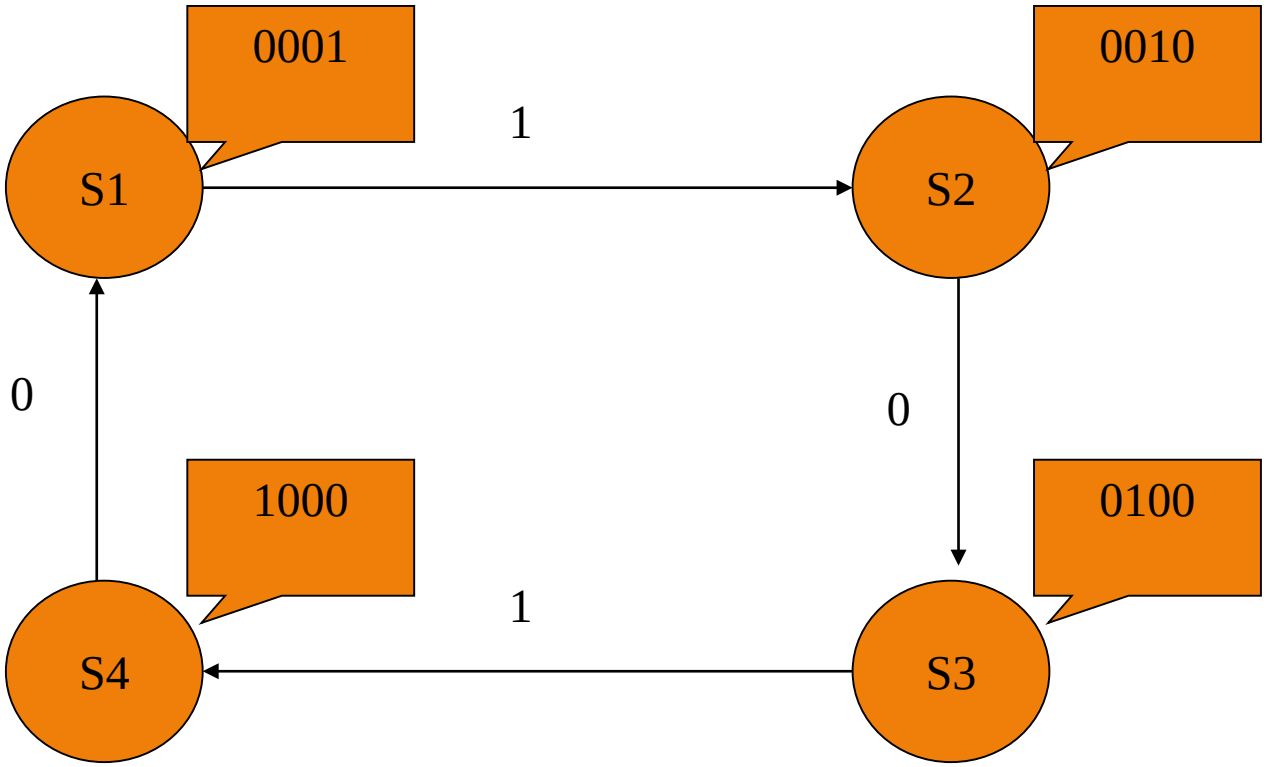
```
Entity test1 IS
PORT ( clk, d1, d2 : in std_logic;
       q1, q2 : out std_logic);
END test1;

ARCHITECTURE  test1_body OF test1 IS
BEGIN
    PROCESS (clk, d1)
    BEGIN
    IF (clk'event and clk = '1') THEN  q1 <= d1; END IF;
    END PROCESS;

    PROCESS (clk, d2)
    BEGIN
    IF (clk'event and clk= '1') THEN   q2 <= d2; END IF;
    END PROCESS;
END test1_body;
```

VHDL进程语句（EXAMPLE 2）

- 上述两个程序的仿真波形是相同的，如下所示：



Moore Machine



Moore Machine

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY moore_example IS
PORT
    (clk, datain, reset : IN std_logic;
     dataout: OUT std_logic_vector(3 downto 0));
END moore_example;

ARCHITECTURE behave OF moore_example IS
TYPE state_type IS ( s1,s2,s3,s4);
SIGNAL state: state_type;
BEGIN
```

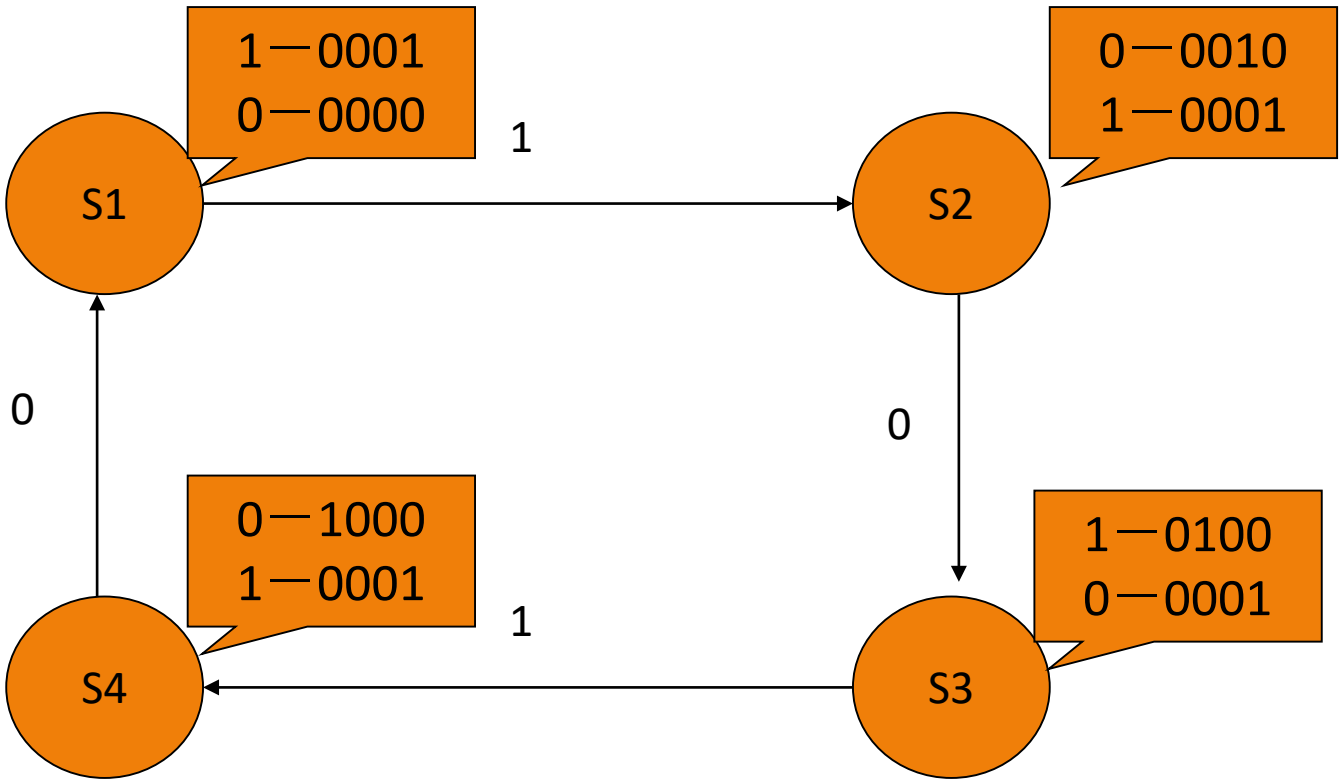
Moore Machine

```
PROCESS (clk, reset )
    IF reset='1' THEN  state<=s1;
    ELSIF ( clk'event and clk='1') THEN
        CASE state IS
          WHEN s1 => IF datain='1' THEN state<=s2; END IF;
          WHEN s2 => IF datain='0' THEN state<=s3; END IF;
          WHEN s3 => IF datain='1' THEN state<=s4; END IF;
          WHEN s4 => IF datain='0' THEN state<=s1; END IF;
        END CASE;
    END IF;
END PROCESS;
```

Moore Machine

```
PROCESS (state)
BEGIN
    CASE state IS
        WHEN s1 => dataout<="0001";
        WHEN s2 => dataout<="0010";
        WHEN s3 => dataout<="0100";
        WHEN s4 => dataout<="1000";
    END CASE;
END PROCESS;
END behave;
```

Mealy Machine



Mealy Machine


```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY mealy_example IS
PORT
   (clk, datain, reset : IN std_logic;
    dataout: OUT std_logic_vector(3 downto 0));
END mealy_example;

ARCHITECTURE behave OF mealy_example IS
TYPE state_type IS ( s1,s2,s3,s4);
SIGNAL state: state_type;
BEGIN
```

Mealy Machine

```
PROCESS (clk, reset )
IF reset='1' THEN  state<=s1;
ELSIF ( clk'event and clk='1') THEN
    CASE state IS
      WHEN s1=>IF datain='1' THEN state<=s2;END IF;
      WHEN s2=>IF datain='0' THEN state<=s3;END IF;
      WHEN s3=>IF datain='1' THEN state<=s4;END IF;
      WHEN s4=>IF datain='0' THEN state<=s1;END IF;
    END CASE;
END IF;
END PROCESS;
```

计算机组织与结构

Mealy Machine

```
PROCESS (state)
BEGIN
    CASE state IS
    WHEN s1 => IF datain='1' THEN dataout<="0001";
                    ELSE dataout<="0000" END IF;
    WHEN s2 => IF datain='0' THEN dataout<="0010";
                    ELSE dataout<="0001"; END IF;
    WHEN s3 => IF datain='1' THEN dataout<="0100";
                    ELSE dataout<="0001"; END IF;
    WHEN s4 => IF datain='0' THEN dataout<="1000";
                    ELSE dataout<="0001"; END IF;
    END CASE;
END PROCESS;
END behave;
```

Verilog简介

- 1983年由Gateway Design Automation(GDA)公司Philip Moorby为其模拟器产品开发的硬件建模语言
- 1987年Synonsys公司开始使用Verilog HDL行为语言作为综合工具的输入。
- 1990年，Cadence公司成立OVI(Open Verilog International)组织来负责推广Verilog
- 1995年，IEEE制定了Verilog HDL标准，即IEEE Std 1364 - 1995
- 2000年，IEEE公布的Verilog 2001标准，其大幅度地提高了系统级和可综合性能。

Verilog的基本结构–模块（module）

- 模块是Verilog的基本描述单位，用于描述某个设计的功能或结构及与其它模块通信的外部接口

- 例子

```
module HalfAdder (A, B, Sum, Carry);
   input  A, B;
   output  Sum, Carry;
   assign   Sum=A^B;
   assign   Carry=A&B;
endmodule
```

Verilog的基本结构-模块（module）

- Verilog 模块结构完全嵌在 module 和 endmodule声明语句之间
- 每个Verilog程序包括四个主要部分：端口定义、I/O说明、内部信号声明、功能定义。
- 模块中，可以采用下述方式描述设计：
  - 数据流方式
  - 行为方式
  - 结构方式
  - 上述方式的混合

Verilog的基本运算符

Bitwise operators

~	NOT
&	AND
|	OR
^	EXOR

More Operators:

>>	Shift right
<<	Shift left
+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulus

Relational Operators:

==	Equal to
!=	Not equal
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
&&	AND
||	OR

Verilog数据流描述方式

- 使用连续赋值语句

assign [delay] LHS_net=RHS_expression;

- 例子

```
module HalfAdder(A, B, Sum, Carry);
      input  A, B;
      output  Sum, Carry;
      assign  Sum=A^B;
      assign  Carry=A&B;
 endmodule
```

Verilog行为描述方式

- 使用过程语句描述：
  - initial语句：只执行一次
  - always语句：循环重复执行

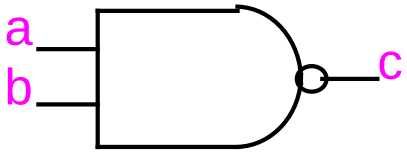
- 例子：

```
module HalfAdder(A, B, Sum, Carry);
      input  A, B;
      output  Sum, Carry;
      always@(A or B)
        begin
          Sum=A^B;
          Carry=A&B;
        end
 endmodule
```

Verilog结构化描述方式

- 用Verilog直接描述逻辑图
- 可使用
  - 内置门：not, and, or…
  - 开关级：nmos, cmos, tran…
  - 用户自定义的结构
  - 模块实例：其它module单元

Verilog组合逻辑-举例

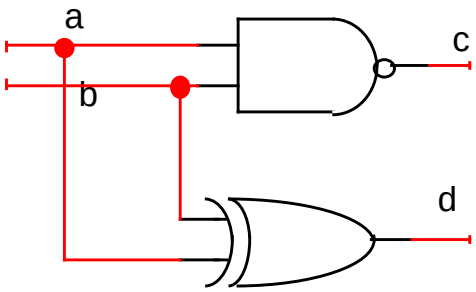



```
module example1(c,a,b);
   input a, b;
   output c;

   // Functionality
   assign c = ~(a & b);
endmodule
```

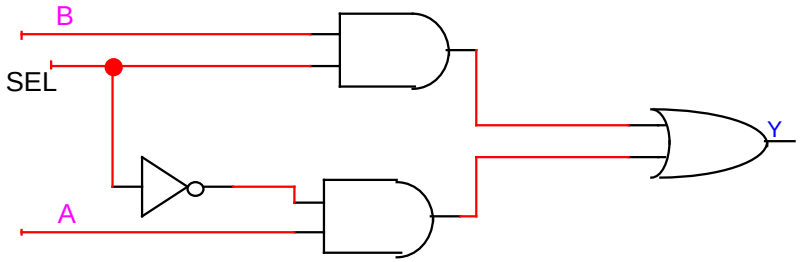
```
module  example2 (a,b,c);
// Port modes
input a,b;
output c;

// Registered identifiers
reg c;

// Functionality
always @ (a or b)
      c = ~(a & b);
endmodule
```

Sensitivity list

Verilog组合逻辑-举例



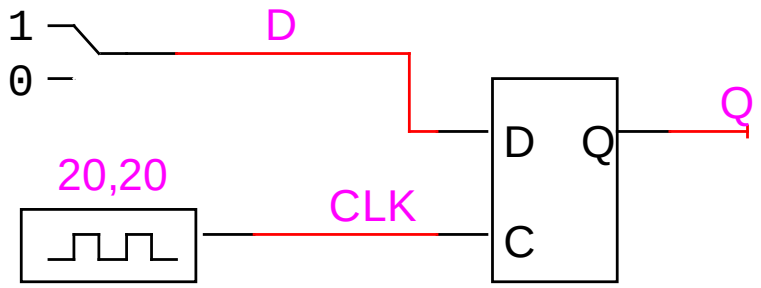

```
module example3(a,b,c,d);

    // Port modes
    input a, b;
    output c;
    output d;

    // Registered identifiers
    reg c,d;

    // Functionality
    always @ (a or b)
        begin
      c <= ~(a & b);
      d <= a ^ b;
       end

endmodule
```




```
module Mux2 (A, B, Sel, Y);

    input A, B, Sel;
    output Y;
    reg Y;

    // Functionality
    always @ (A or B or Sel)
        if (Sel==0)
            Y = A;
        else
            Y = B;

endmodule
```

Verilog时序逻辑-举例


```
module D_FF (D, Clock, Q);

    input D, Clock;
    output Q;
    // Registered identifiers
    reg Q;

    // Functionality
    always @ (posedge Clock)
        Q <= D;

endmodule
```



Sensitivity list

?

```
module D_FF (D,Clock,Q,Q_bar,Reset);
input D,Clock,Reset;
output Q, Q_bar;
reg Q;
always @ (posedge Clock or posedge Reset)
    if (Reset == 1)
        Q <= 0;
    else
        Q <= D;
assign Q_bar = ~Q;
endmodule
```

Verilog时序逻辑-状态机设计模板

计算机组织与结构

Tips

Break FSMs into four blocks:
State definitions-*Next state calculations (decoder)-Registers or flip-flops calculation-Output calculations (logic)*

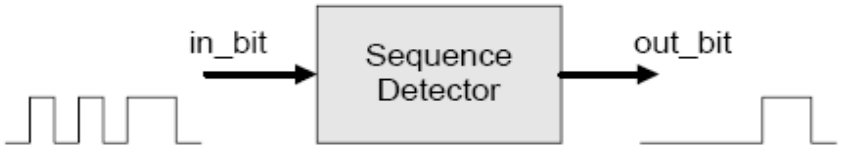
```
//state flip-flops
reg [2:0] state, next_state;

//1、state definitions
parameter S0=2'b00 S1=2'b 01, S2=2'b 10, S3=2'b11,…

// 2、State machine descriptions，next state calculations
always @(state or… .)
begin case (state)
…..
End

//3、register or flip-flop calculation
always@(posedge clk)
state<=next_state

//4、Output calculations
Output=f(state, inputs)
```

```
module mod_name ( ... );
        input ... ;
        output ... ;

        parameter size = ... ;
        reg [size-1: 0] current_state;
        wire [size-1: 0] next_state;

        // State definitions
        `define state_0 2'b00
        `define state_1 2b01

        always @ (current_state or the_inputs) begin
                // Decode for next_state with case or if statement
                // Use blocked assignments for all register transfers to ensure
                // no race conditions with synchronous assignments
        end

        always @ (negedge reset or posedge clk) begin
                if (reset == 1'b0) current_state <= state_0;
                else               current_state <= next_state;
        end

        //Output assignments
endmodule
```

Next State Logic

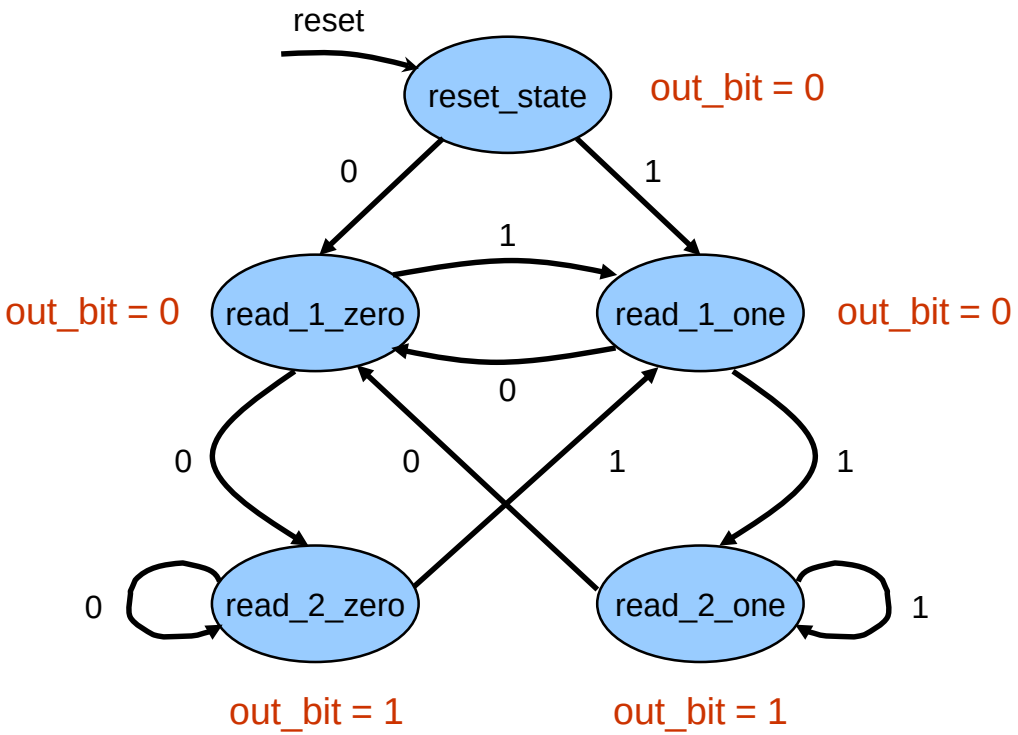
State Register

Verilog时序逻辑-Moore状态机举例



Functionality: Detect two successive 0s or 1s in the serial input bit stream

FSM Flow-Chart



计算机组织与结构

Verilog时序逻辑- Moore状态机举例

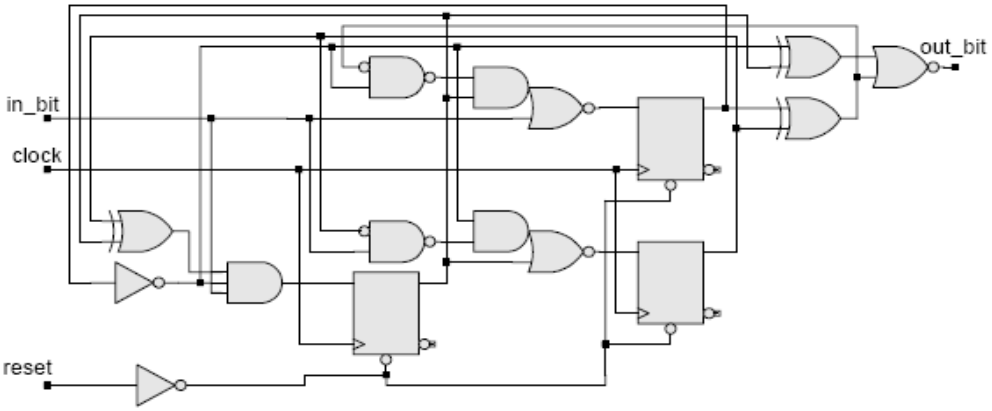
计算机组织与结构

```
module seq_detect (clock, reset, in_bit, out_bit);
  input clock, reset, in_bit;
  output out_bit;

  reg [2:0] state_reg, next_state;

  // State declaration
  parameter   reset_state   =   3'b000;
  parameter   read_1_zero =   3'b001;
  parameter   read_1_one   =   3'b010;
  parameter   read_2_zero =   3'b011;
  parameter   read_2_one   =   3'b100;

  // state register
  always @ (posedge clock or posedge reset)
    if (reset == 1)
      state_reg <= reset_state;
    else
      state_reg <= next_state;

  // next-state logic
```

```
always @ (state_reg or in_bit)
  case (state_reg)
    reset_state:
        if (in_bit == 0)
            next_state = read_1_zero;
        else if (in_bit == 1)
            next_state = read_1_one;
        else next_state = reset_state;
    read_1_zero:
        if (in_bit == 0)
            next_state = read_2_zero;
        else if (in_bit == 1)
            next_state = read_1_one;
        else next_state = reset_state;
    read_2_zero:
        if (in_bit == 0)
            next_state = read_2_zero;
        else if (in_bit == 1)
            next_state = read_1_one;
        else next_state = reset_state;
```

Verilog时序逻辑-Moore状态机举例


```
        read_1_one:
            if (in_bit == 0)
                next_state = read_1_zero;
            else if (in_bit == 1)
                next_state = read_2_one;
            else  next_state = reset_state;
        read_2_one:
            if (in_bit == 0)
                next_state = read_1_zero;
            else if (in_bit == 1)
                next_state = read_2_one;
            else next_state = reset_state;
        default: next_state = reset_state;
    endcase

    assign out_bit = ((state_reg == read_2_zero) || (state_reg == read_2_one)) ? 1 : 0;
endmodule
```



课程内容

计算机组织与结构

一、数字系统设计的一些概念和技巧

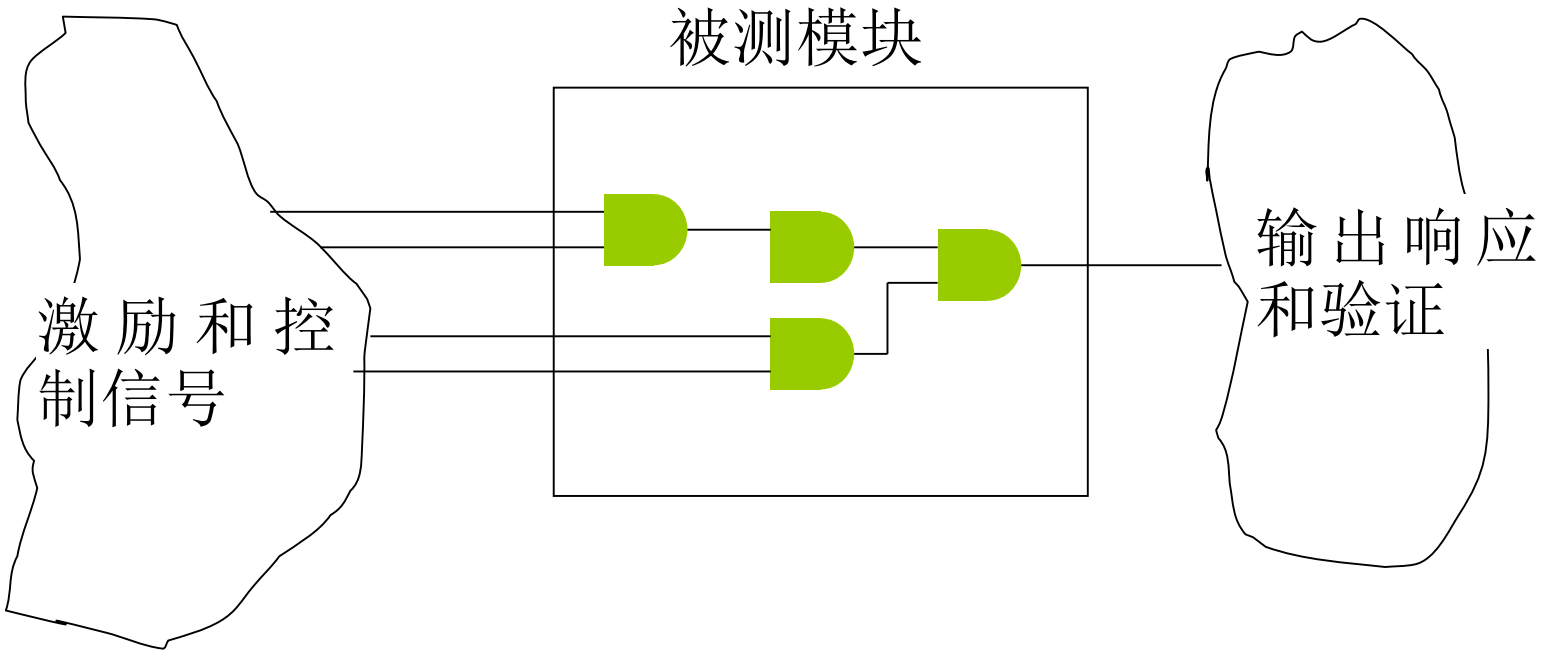
二、VHDL与Verilog语言

三、硬件设计模块的仿真与测试

四、软件开发环境VIVADO

五、硬件平台简介

Verilog模块的测试



Verilog模块的测试 - 测试文件结构

- 测试文件通常包含以下内容：

```
module Test_bench(); //通常无输入输出
        信号或变量声明定义
        使用 initial  或 always语句产生激励
        例化待测试模块
        监控和比较输出响应
Endmodule
```

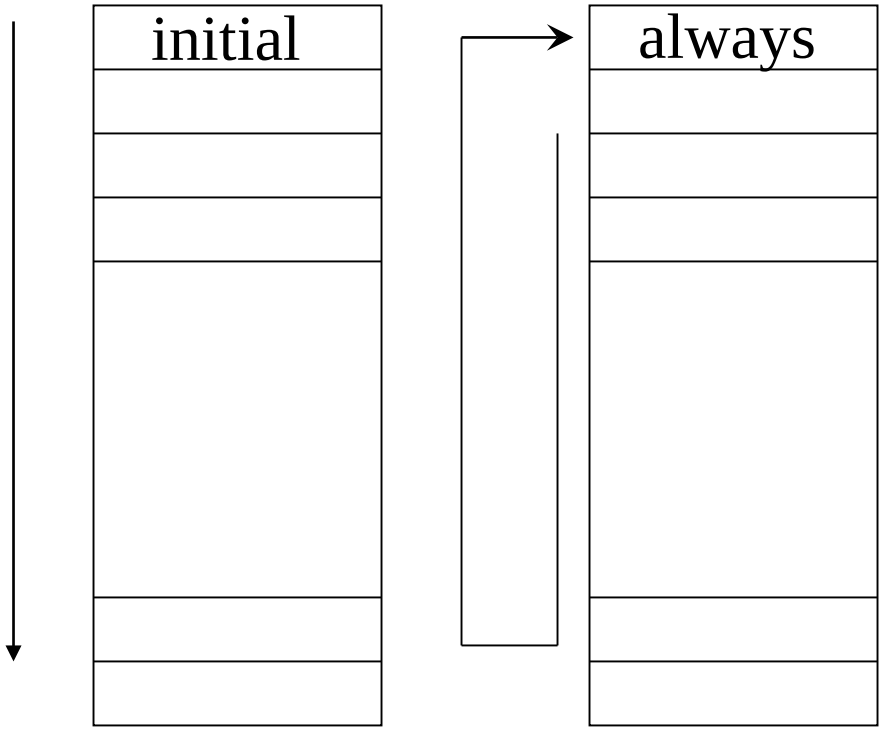
Verilog模块的测试 - 结构举例

测试模块常见的形式：

```
module testfirstmodule;
    reg ...;          //被测模块输入
    wire...;          //被测模块输出

    initial
          begin
          ...;
          end ... ...//产生测试激励或控制信号

    always #delay
          begin
          ...;
          end  ... ...//产生测试激励或控制信号

    ForTestingModule m(.in1(ina), .in2(inb), .out1(outa));
   //被测模块的实例引用

endmodule
```

Verilog模块的测试

测试模块中常用的过程块：



所有的过程块都在0时刻同时启动；它们是并行的，在模块中不分前后。

- initial块只执行一次
- always块 只要符合触发条件可以循环执行

Verilog模块的测试 -信号的初始化

- 代码中的变量的初始化可以用initial进行初始化，也可以在定义的时候进行初始化
- 在一定的触发条件下，例如复位中进行初始化
- 利用Verilog语言的读文件功能，从文本文件中读取数据（该数据可以通过C/C++、MATLAB 等软件语言生成）

Verilog模块的测试 - 激励源产生

绝对时间:

```
initial begin
    Reset = 1;  //仿真时间零点激励
    Load = 0;   //仿真时间零点激励
    Count = 0;  //仿真时间零点激励
    #100 Reset = 0;
    //绝对时间100激励

    #20 Load = 1;
    //绝对时间120激励
    //相对上一个时间点20
    #20 Count  = 1;
    //绝对时间140激励
    //相对上一个时间点20
end
```

相对时间:

```
 always @ (posedge clock)
Count <= Count + 1;   //绝对时间的递增

initial begin
    if (Count <= 5)    begin
      Reset = 1; Load = 0;
    end
    else begin  //触发事件，产生下列激励
          Reset = 0; Load = 1;
          end
End

initial begin
    if (Count == 0110) begin
         Load <= 0;
         $display("Terminal.");
         end
end
```

Verilog模块的测试 – 测试时钟产生

- 基于initial 语句的方法：

```
parameter clk_period = 10;
reg clk;
initial begin
        clk = 0;
        forever
                # (clk_period/2) clk = ~clk;
End
```

- 基于always 语句的方法：

```
parameter  clk_period = 10;
reg clk;
initial
    clk = 0;
always # (clk_period/2) clk = ~clk;
```

Verilog模块的测试 - 复位产生

计算机组织与结构

- 异步复位:

```
parameter rst_repiod = 100;
reg rst_n;
initial begin
    rst_n = 0;
    # rst_repiod;
    rst_n = 1;
End
```

- 同步复位:

```
parameter rst_repiod = 100;
reg rst_n;

initial begin
    rst_n = 1;
    @( posedge clk);
     rst_n = 0;
    # rst_repiod;
    @( posedge clk);
     rst_n = 1;

end
```

Verilog模块的测试 - 结果输出

- 关键词$display和$monitor实现结果的输出

在终端中打印信号的ASCII值

```
initial begin
    $timeformat(-9,1,"ns",12);                          //设置输出时钟格式
    $display(" Time Clk Rst Ld SftRg Data Sel");  //显示输入的字符串
    $monitor("%t %b %b %b %b %b %b",                    //设置输出信号格式
    $realtime, clock, reset, load, shiftreg, data, sel); //指定输出的信号
End
```

- 关键词$stop停止仿真

Verilog模块的测试－仿真时间与精度

- 关键字timescale定义测试文件的单位时间，和仿真的精度

‘timescale reference_time/precision

其中，reference_time是单位时间的度量，precision决定了仿真的推进延迟精度，同时也设置了仿真的推进步进单位。

例如

‘timescale 1 ns / 1 ps　　//度量参考为1ns，精度为1ps

#5 reset = 1;　// 5个仿真时间延迟，相当于5×1ns = 5ns 的仿真时间

- 布局布线时将仿真的时延与和物理器件的时延相关联

Verilog仿真文件设计示例

- 实例

```
`timescale 1ns/100ps
`include " ToTestModule.v"
module testclu;
reg   Clk,Rst,C,Zero;

//加入激励信号
initial
    begin
    #0
      Clk=0;
      Rst=1;
      C=0;
      #100  Rst=0;
      #200  Rst=1;
      #200  C=1;
      #200  $stop;
end

always  #10  Clk=~Clk;

ToTestModule
    test(Clk,Rst,C,Zero);


endmodule
```

课程内容

计算机组织与结构

一、数字系统设计的一些概念和技巧

二、VHDL与Verilog语言

三、硬件设计模块的仿真与测试

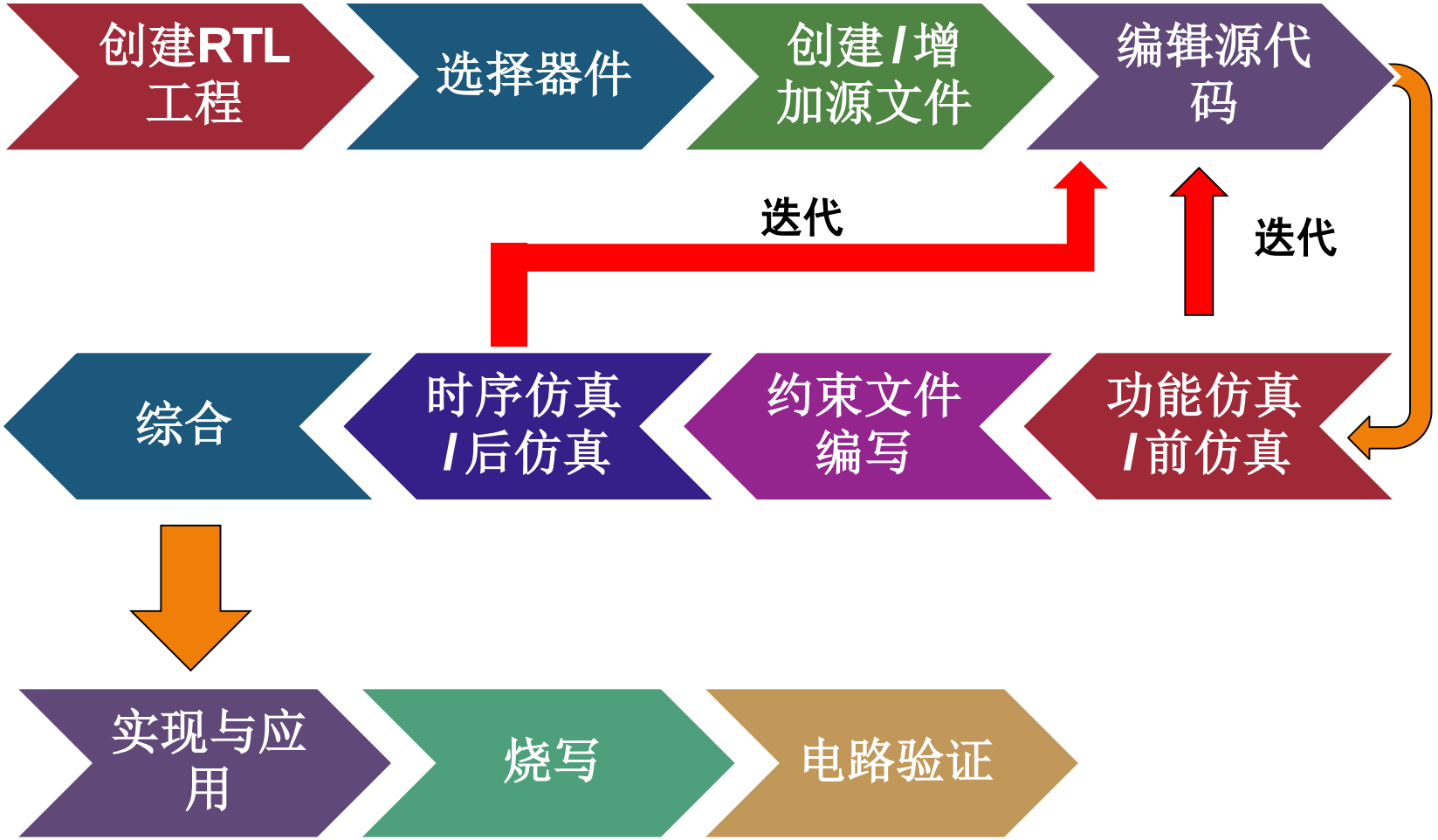
四、软件开发环境VIVADO

五、硬件平台简介

Vivado设计套件简介

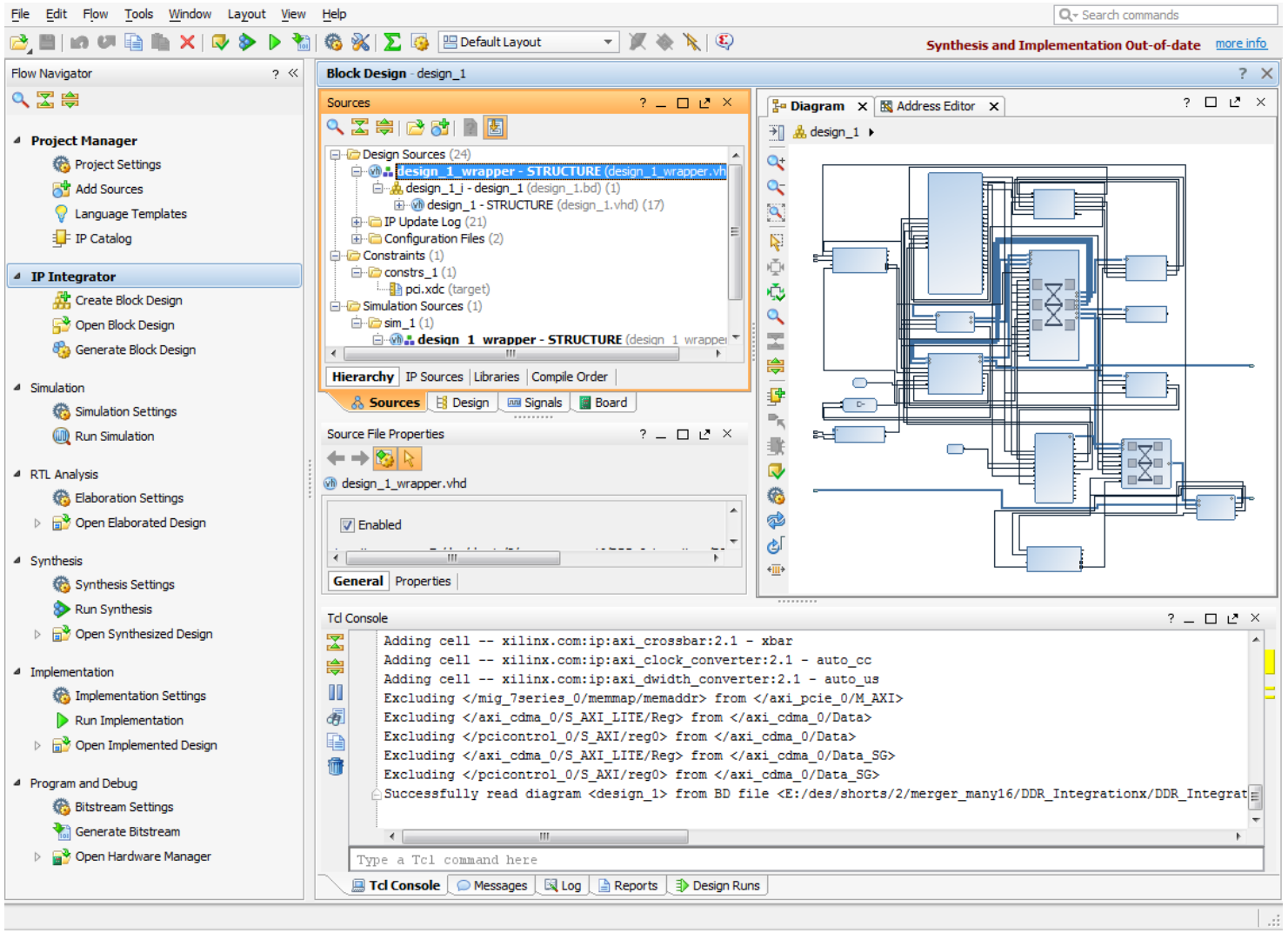
- Vivado 是 Xilinx 公司支持 7 系列， UltraScale 及 UltraScale+系列等更高性能FPGA而设计的开发工具
- Vivado的算法考虑到不断增长的FPGA容量的情况下，采用新的确定性布局布线和路由算法，对全局数据结构具有相同的视图
- Vivado中除SDK和Vivado HLS之外的所有工具被集成在用户图形化接口中

Vivado数字逻辑开发流程

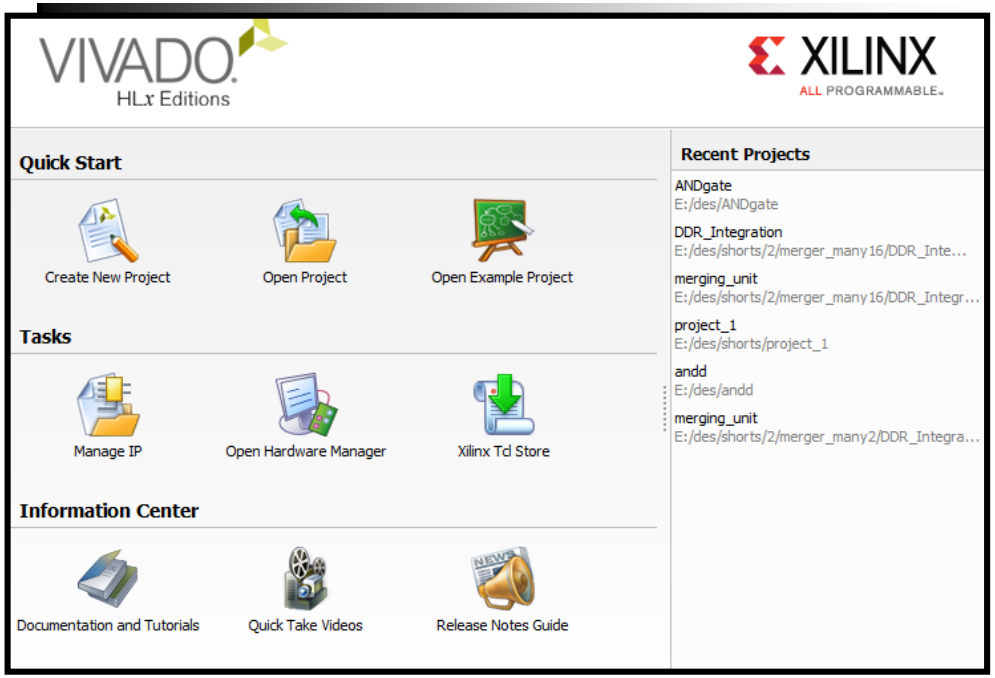


Vivado GUI

计算机组织与结构

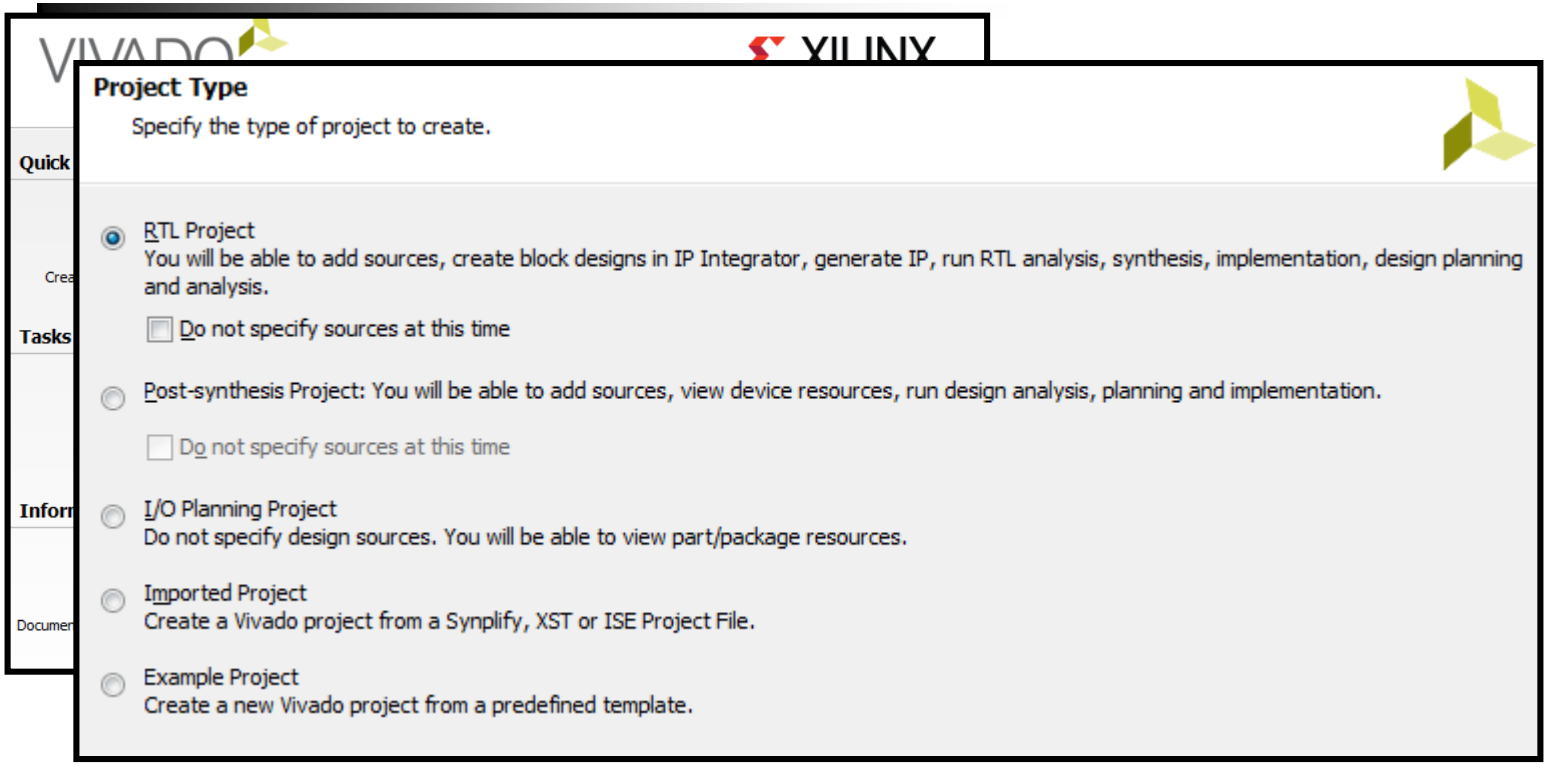


创建新的工程



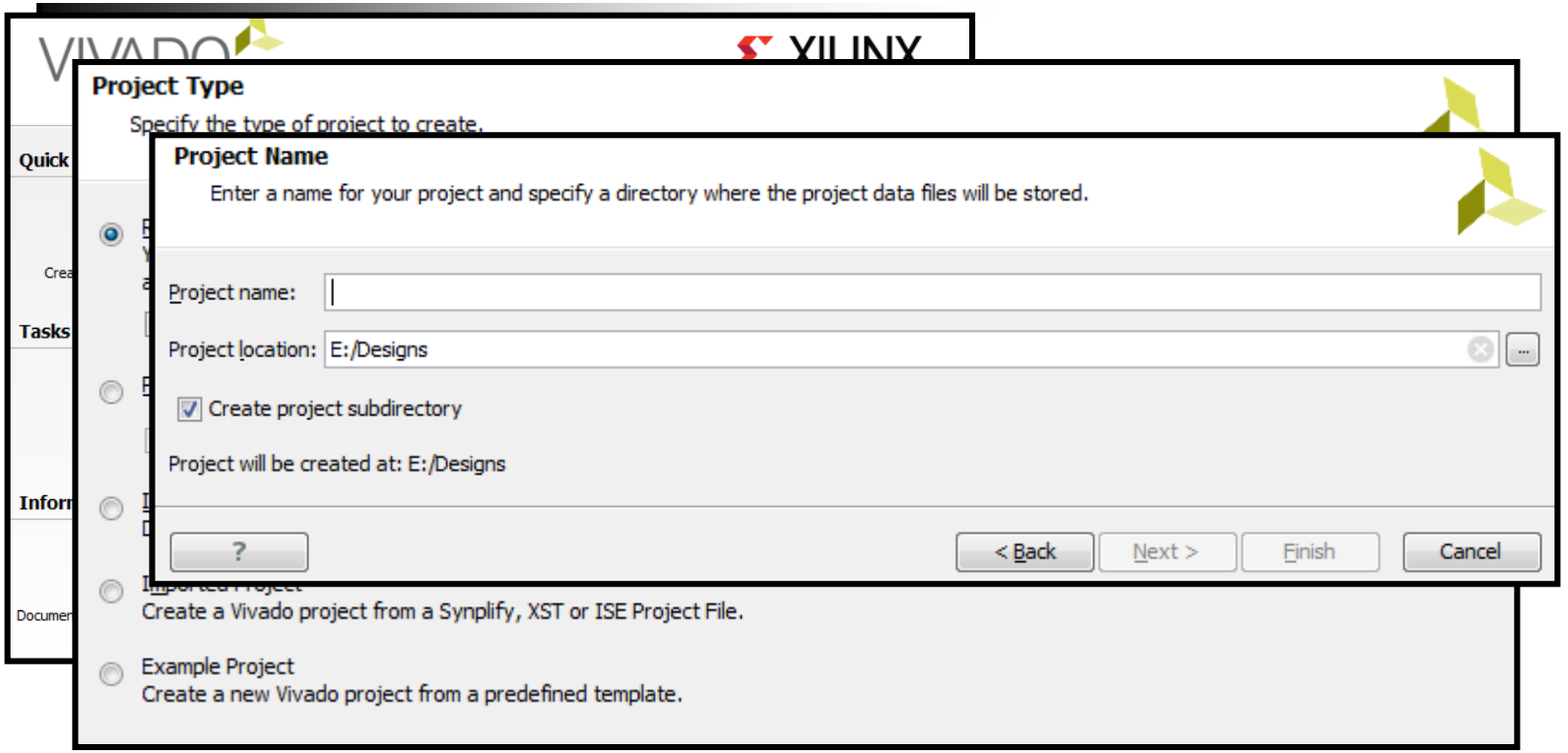
计算机组织与结构

创建新的工程



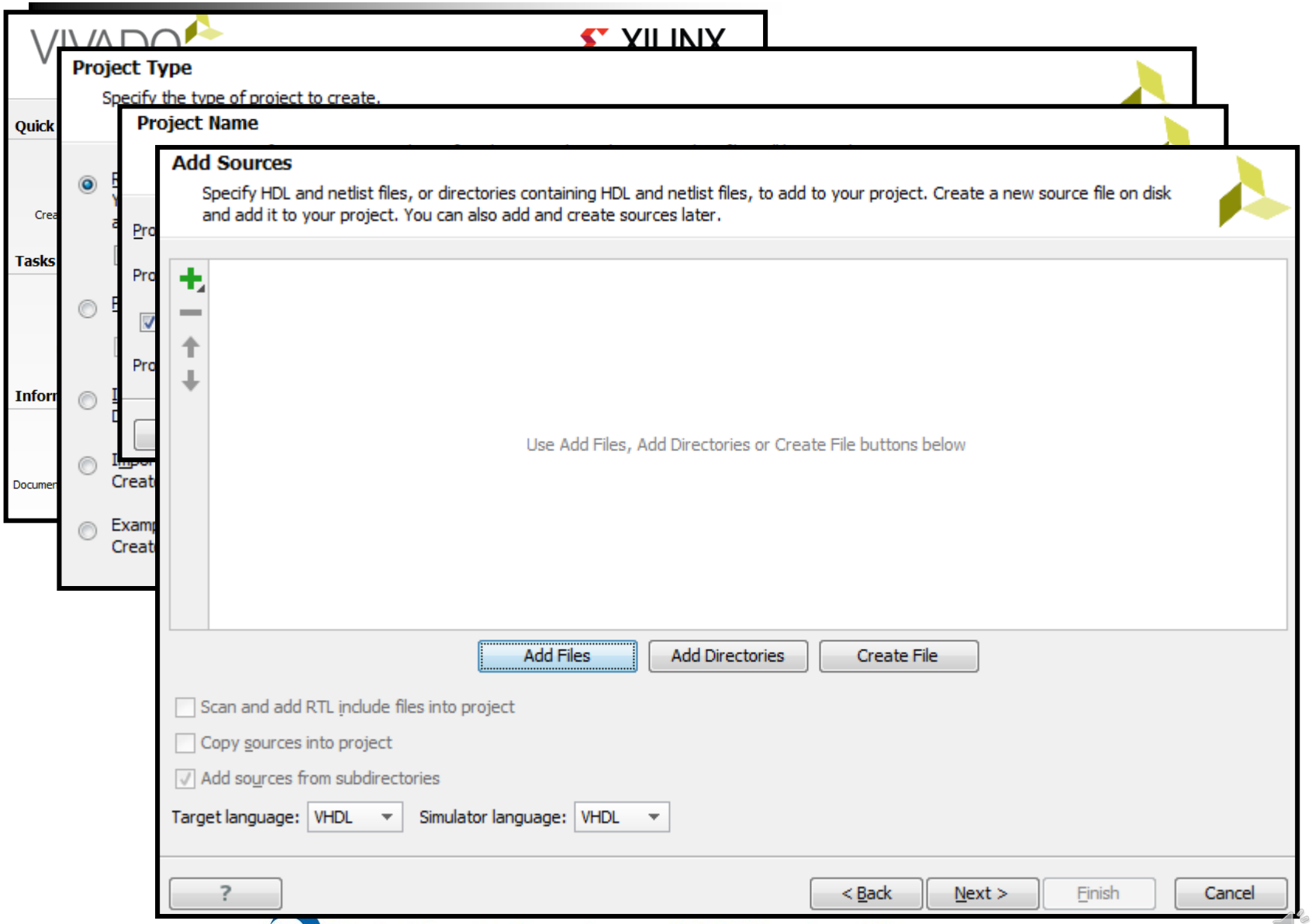
计算机组织与结构

创建新的工程



计算机组织与结构

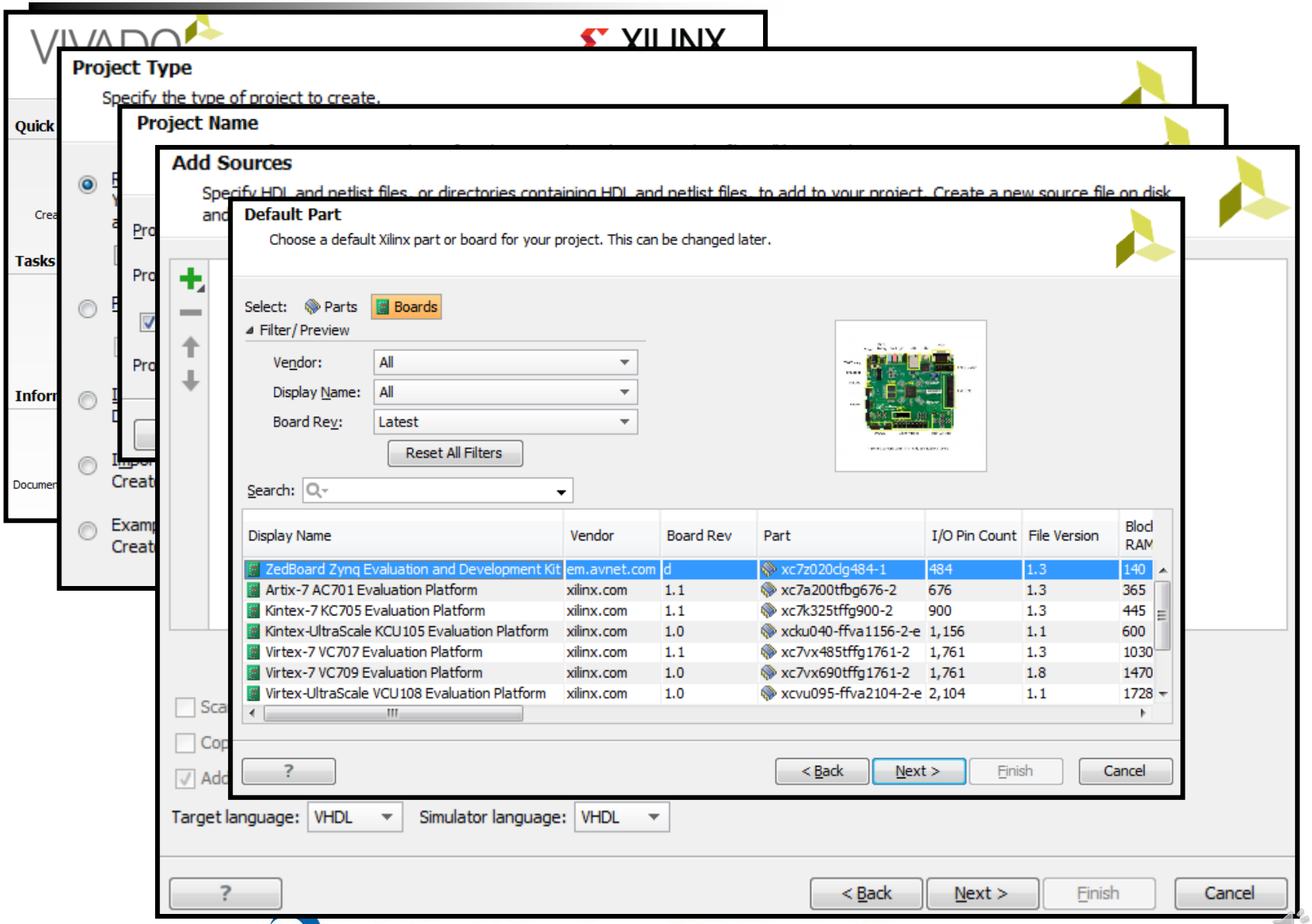
创建新的工程



计算机组织与结构

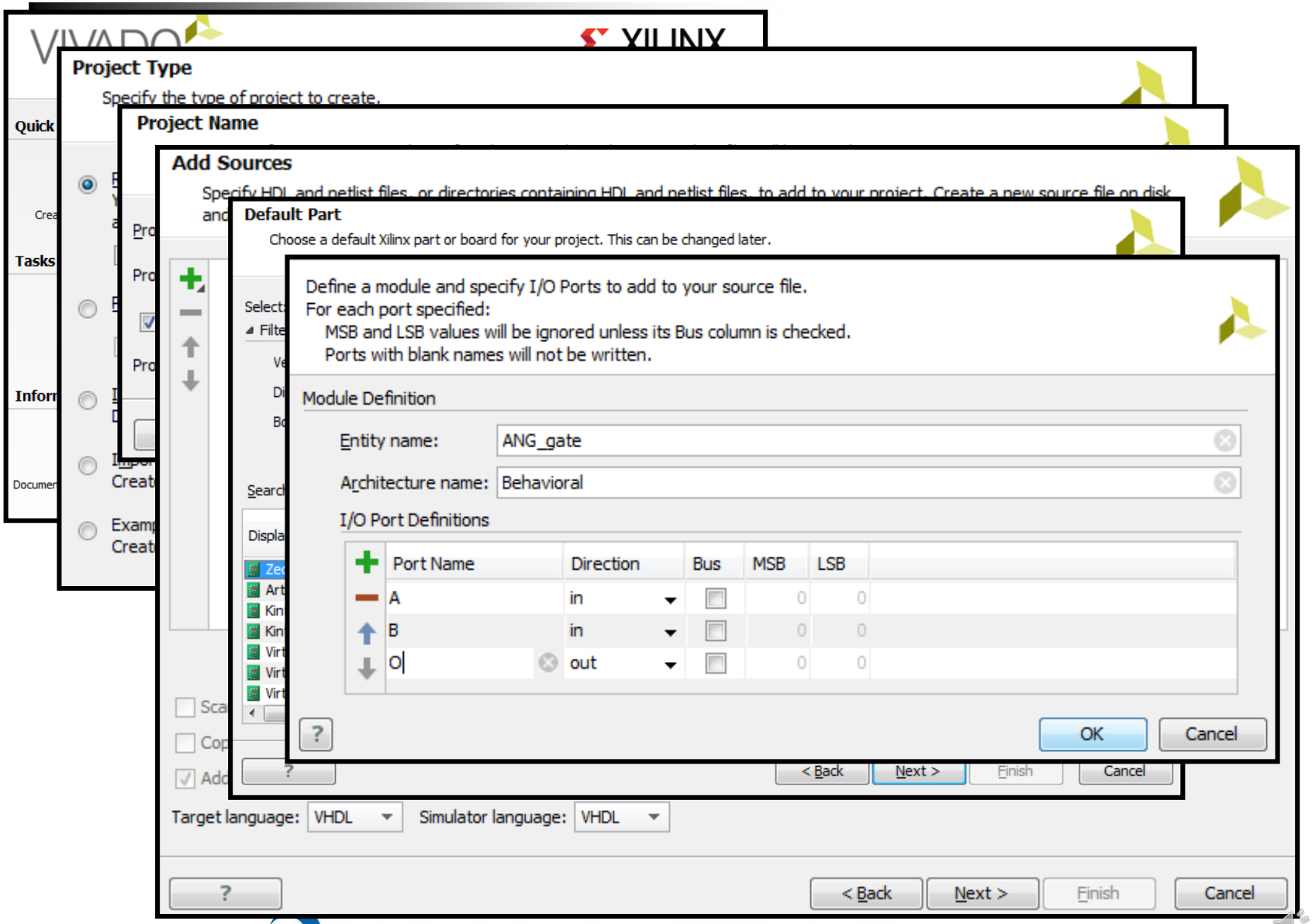
创建新的工程

计算机组织与结构



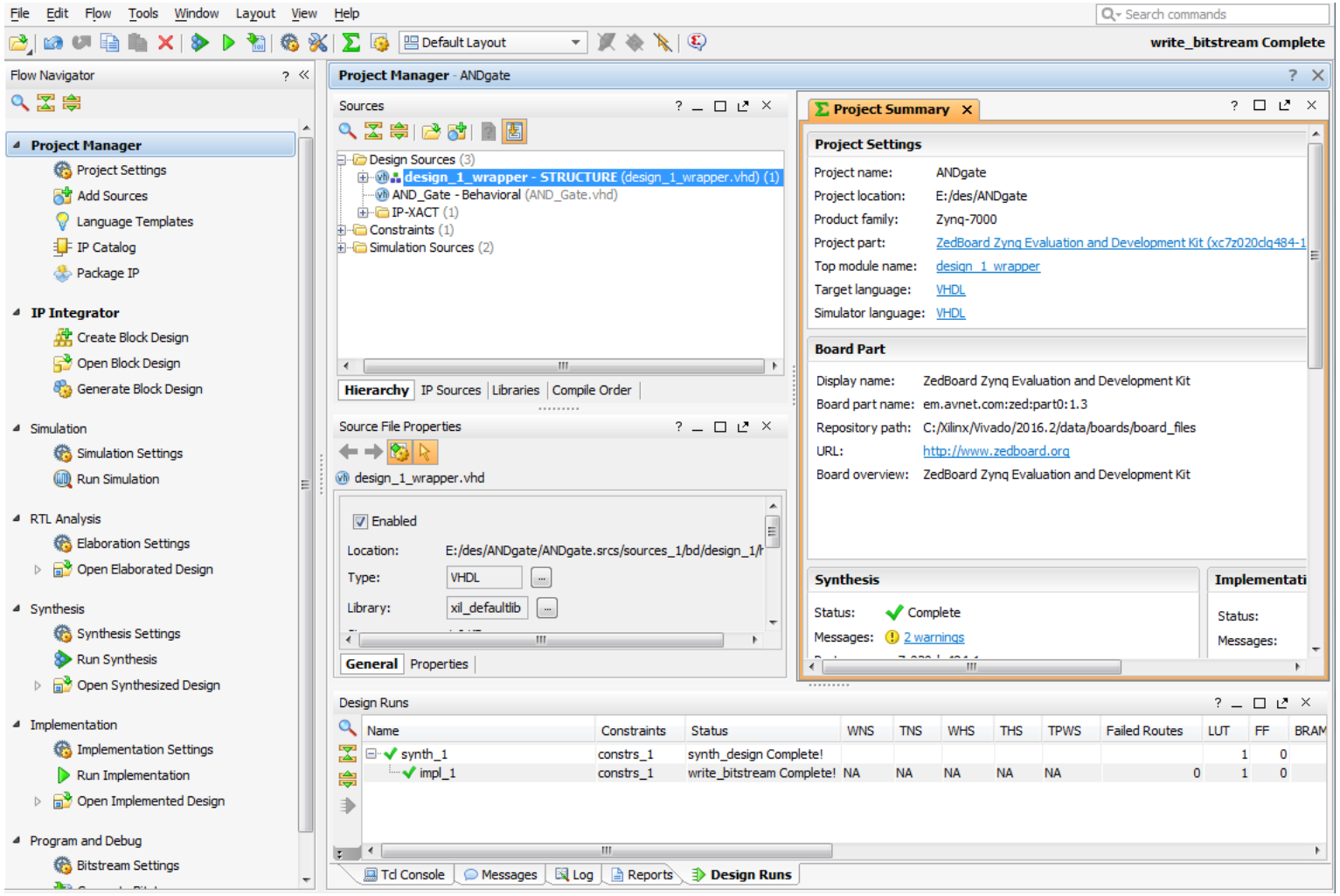
创建新的工程

计算机组织与结构



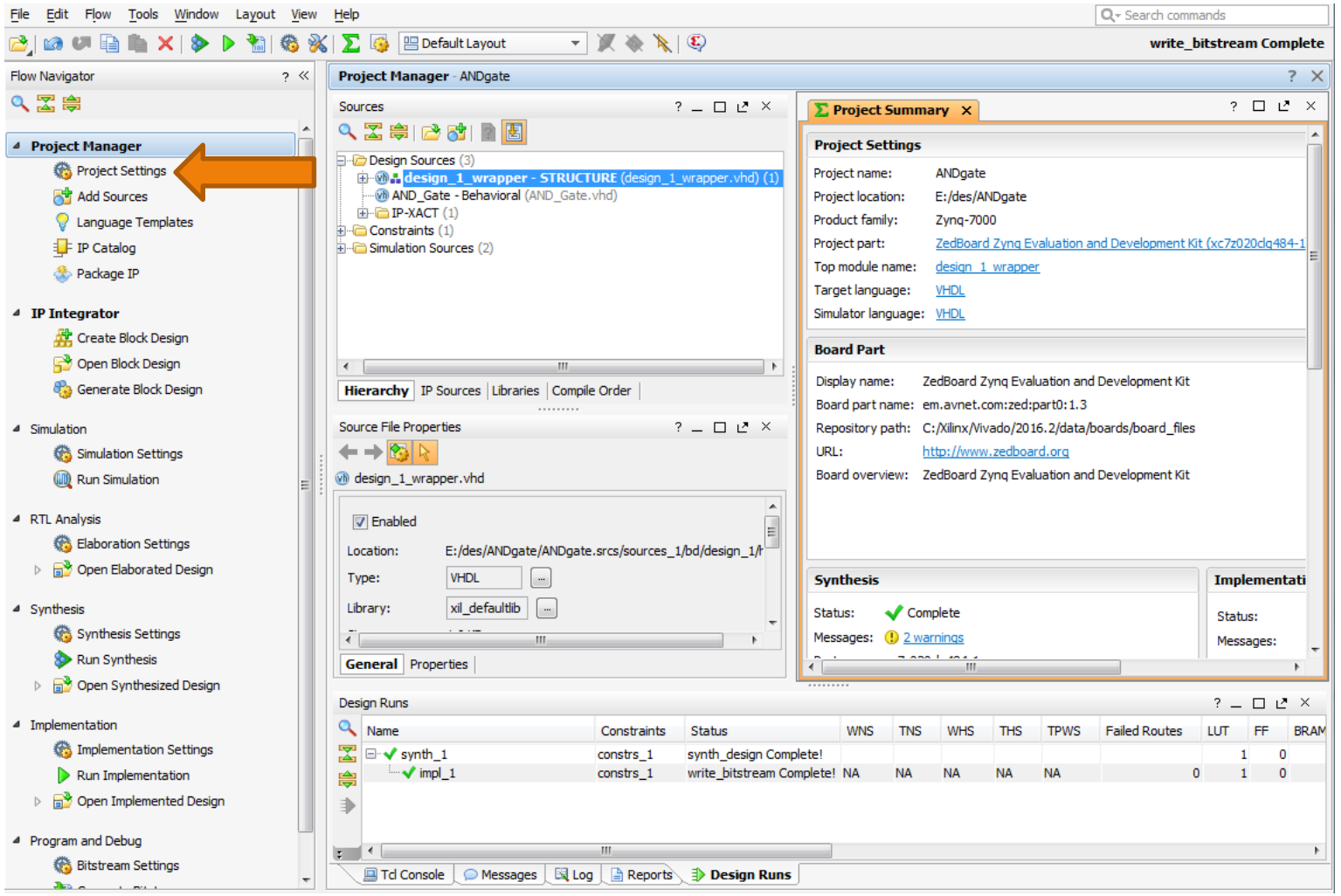
工程管理器

计算机组织与结构



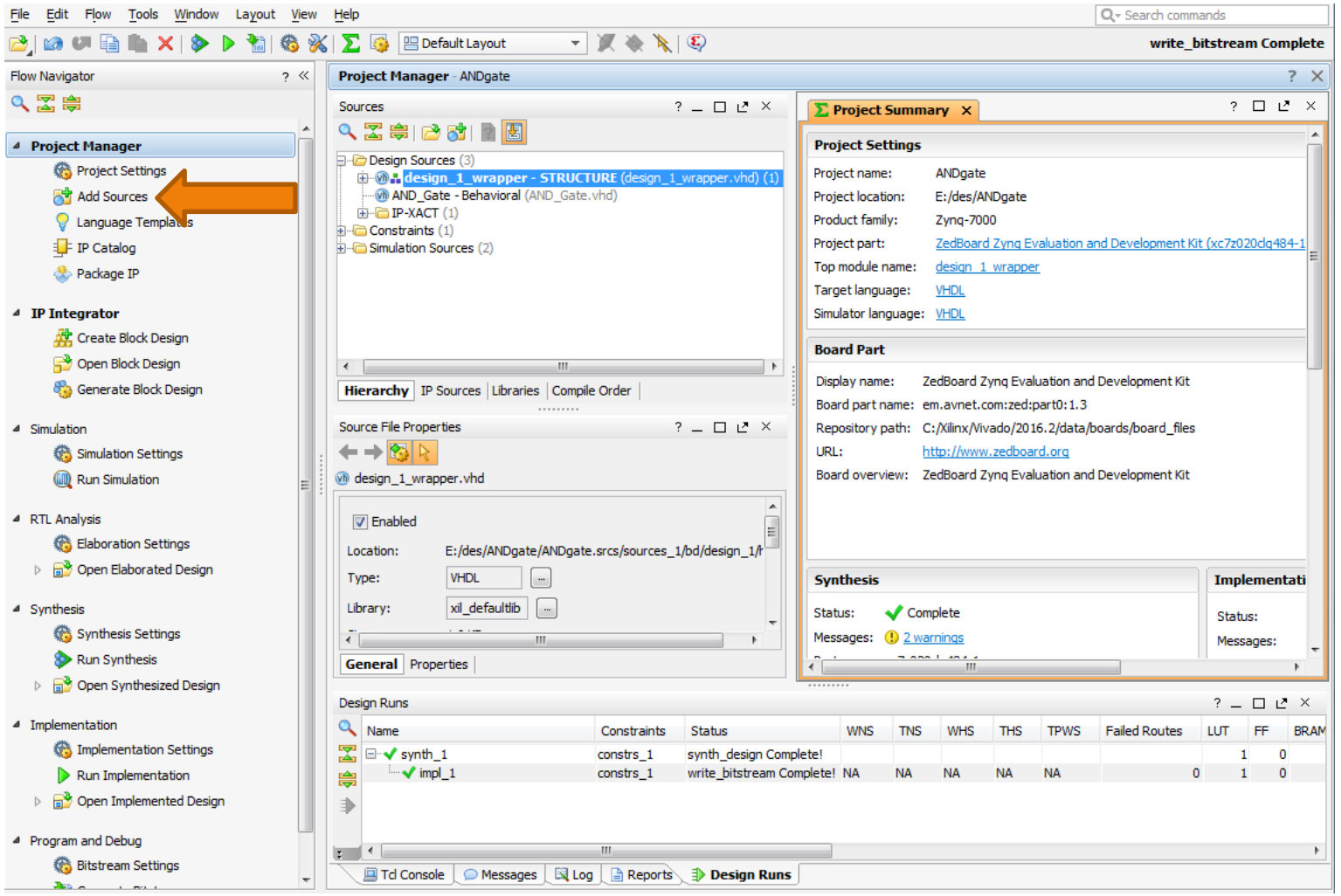
工程管理器

计算机组织与结构



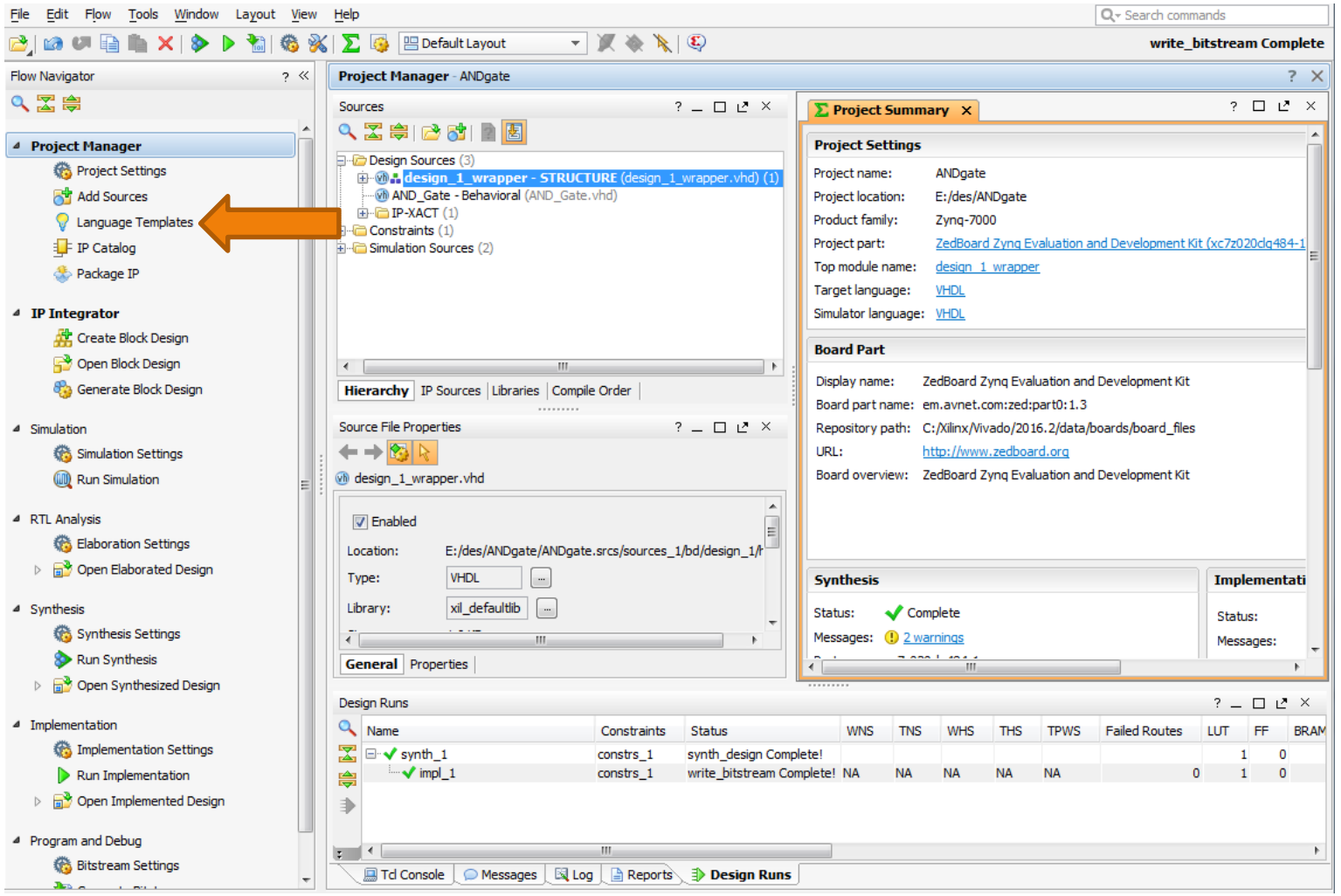
工程管理器

计算机组织与结构



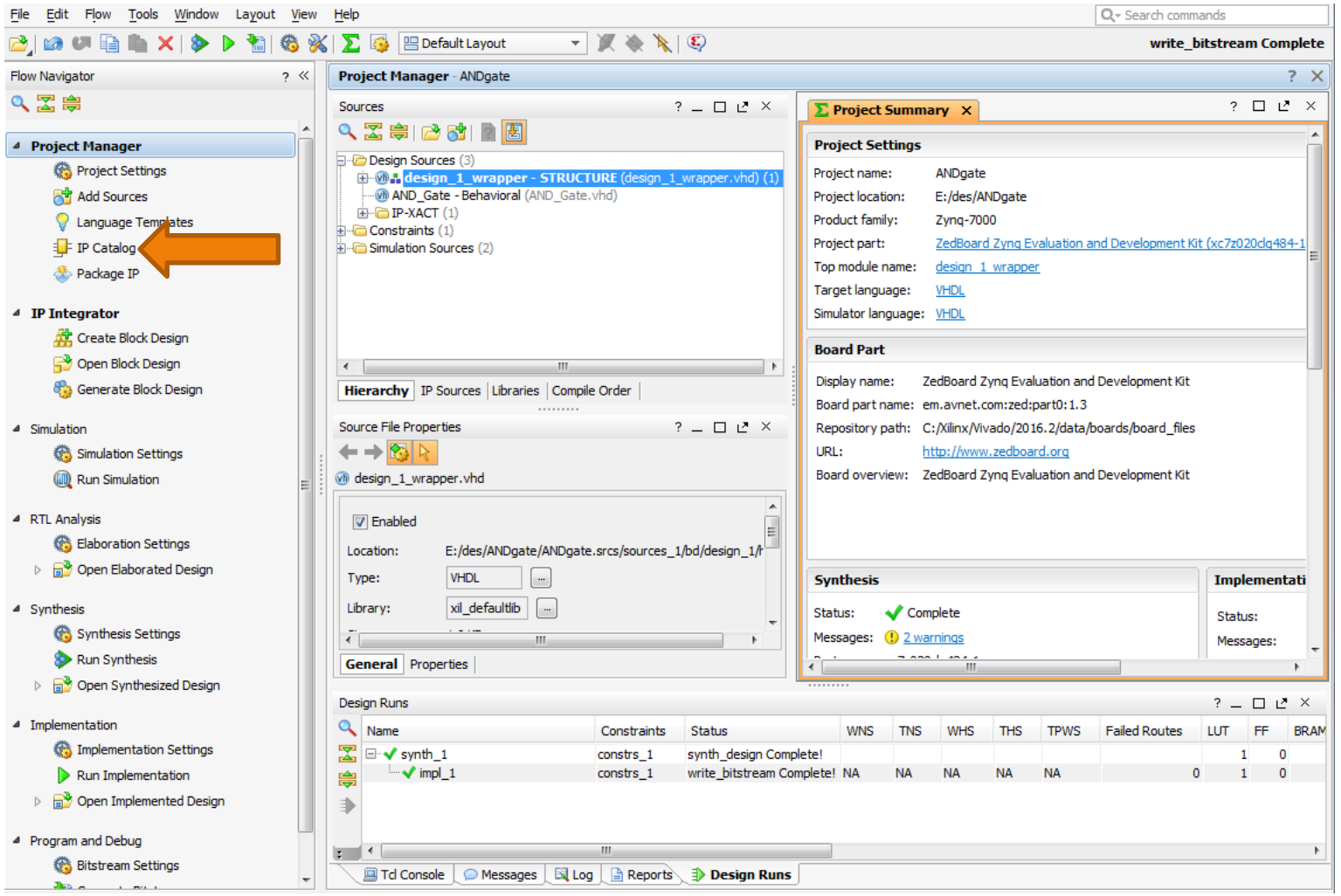
工程管理器

计算机组织与结构



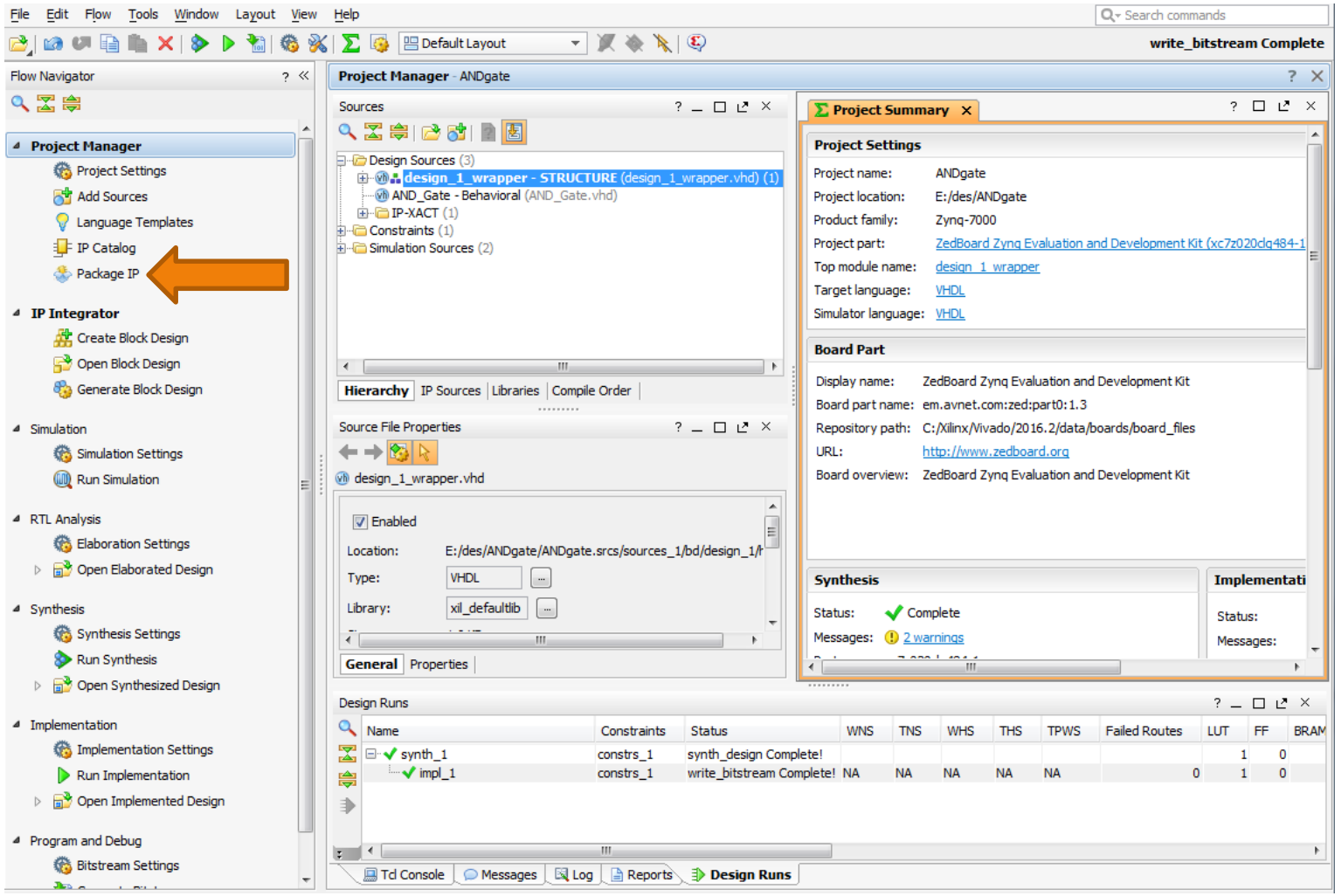
工程管理器

计算机组织与结构



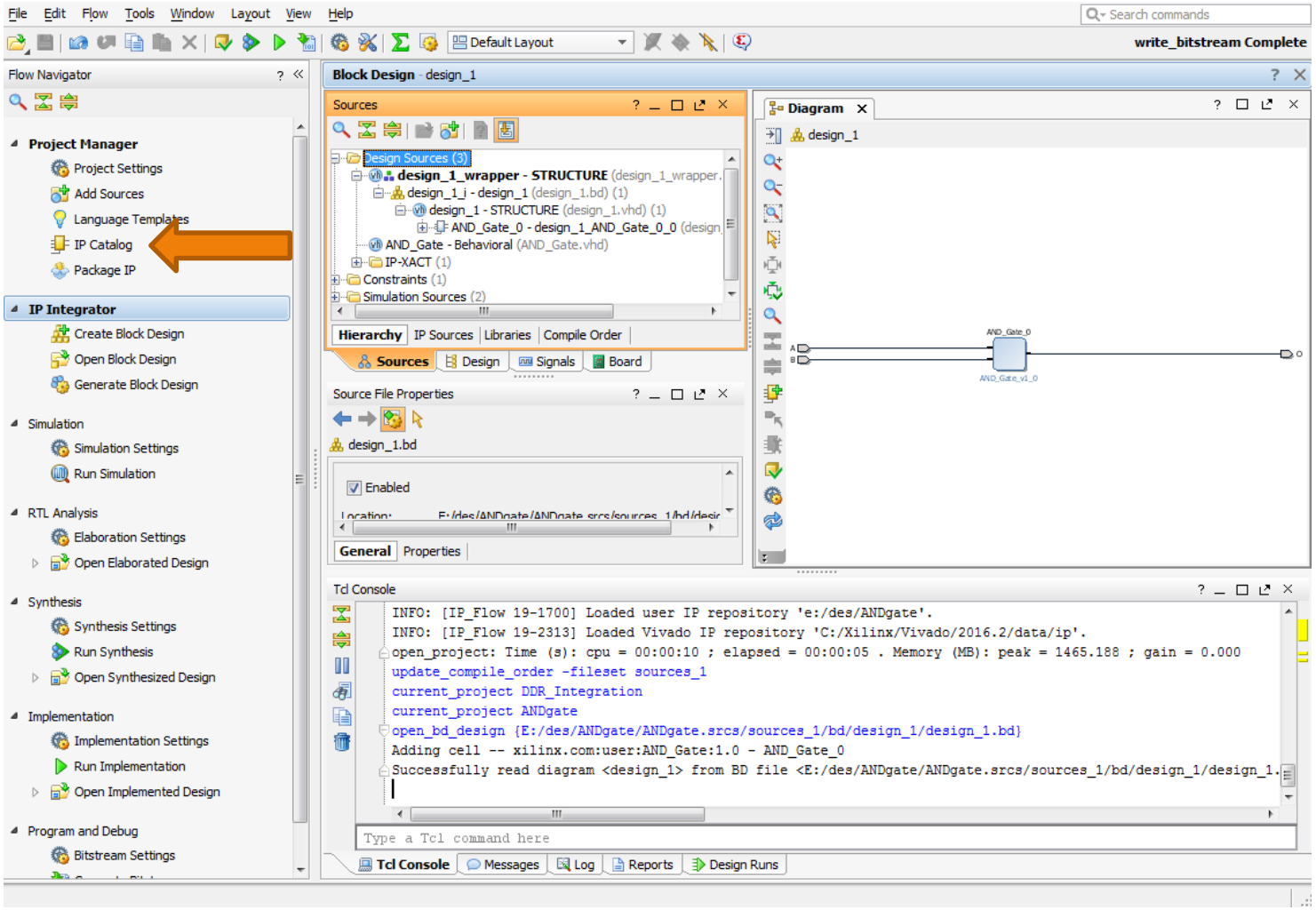
工程管理器

计算机组织与结构



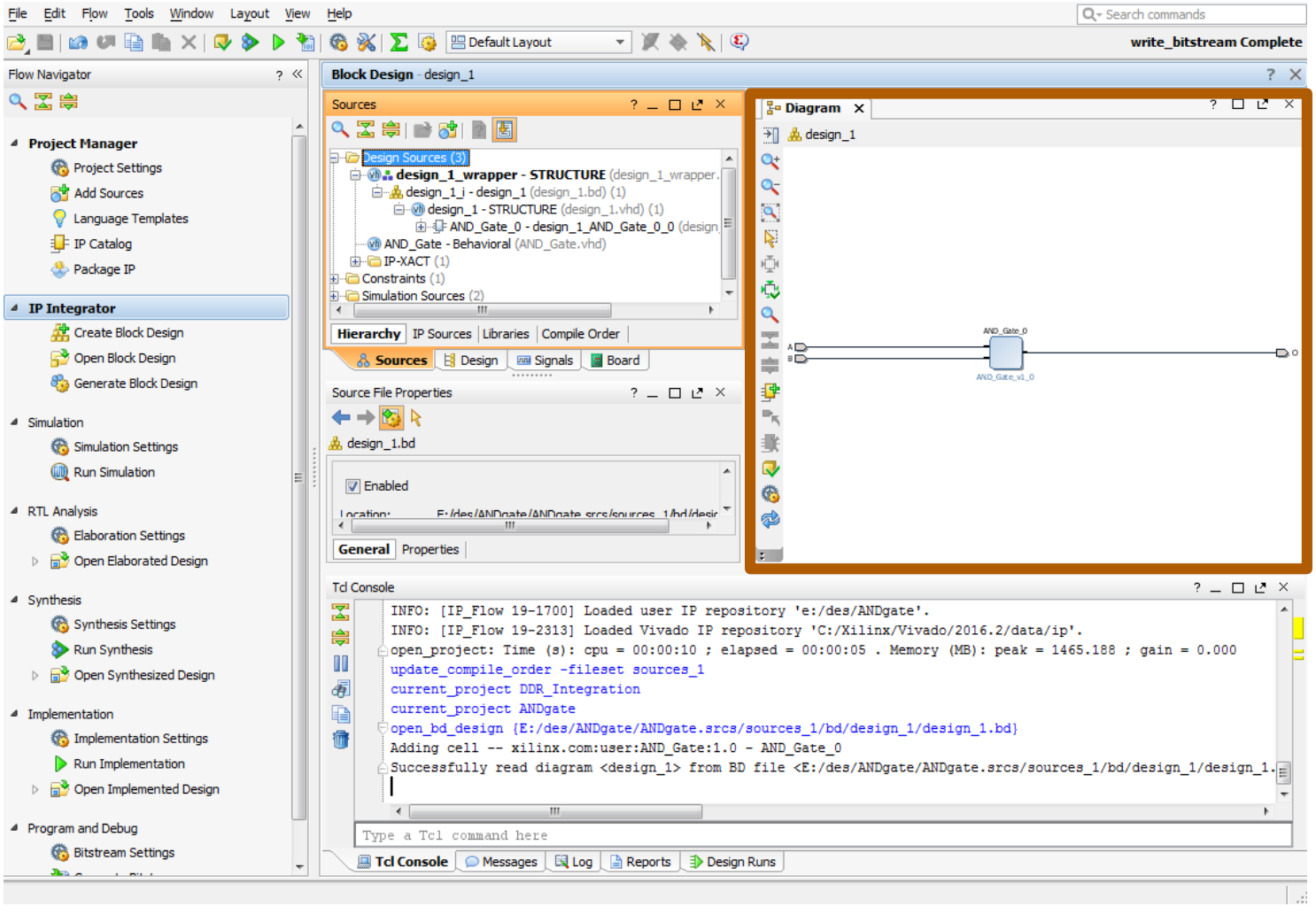
IP核

计算机组织与结构



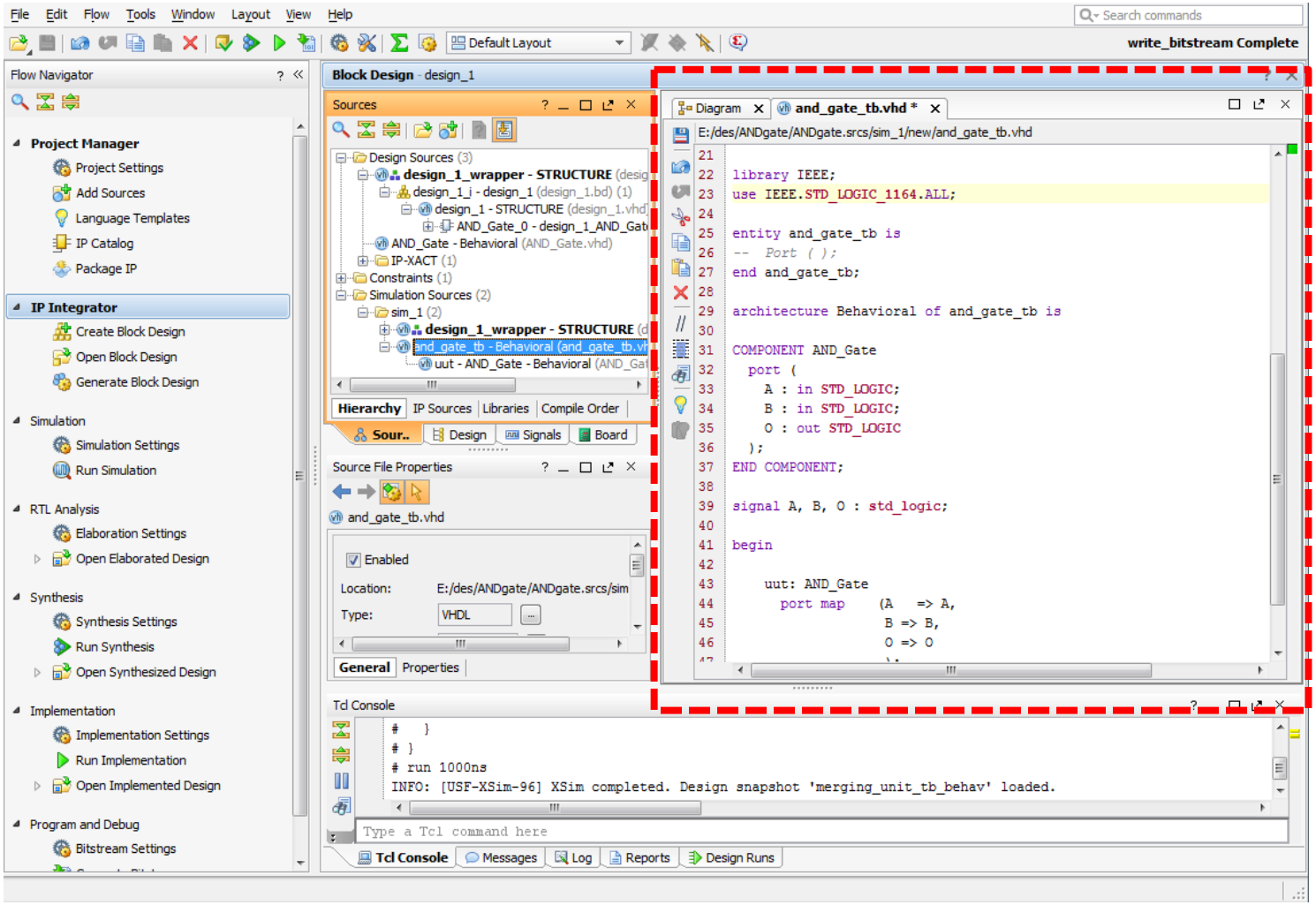
IP核

计算机组织与结构

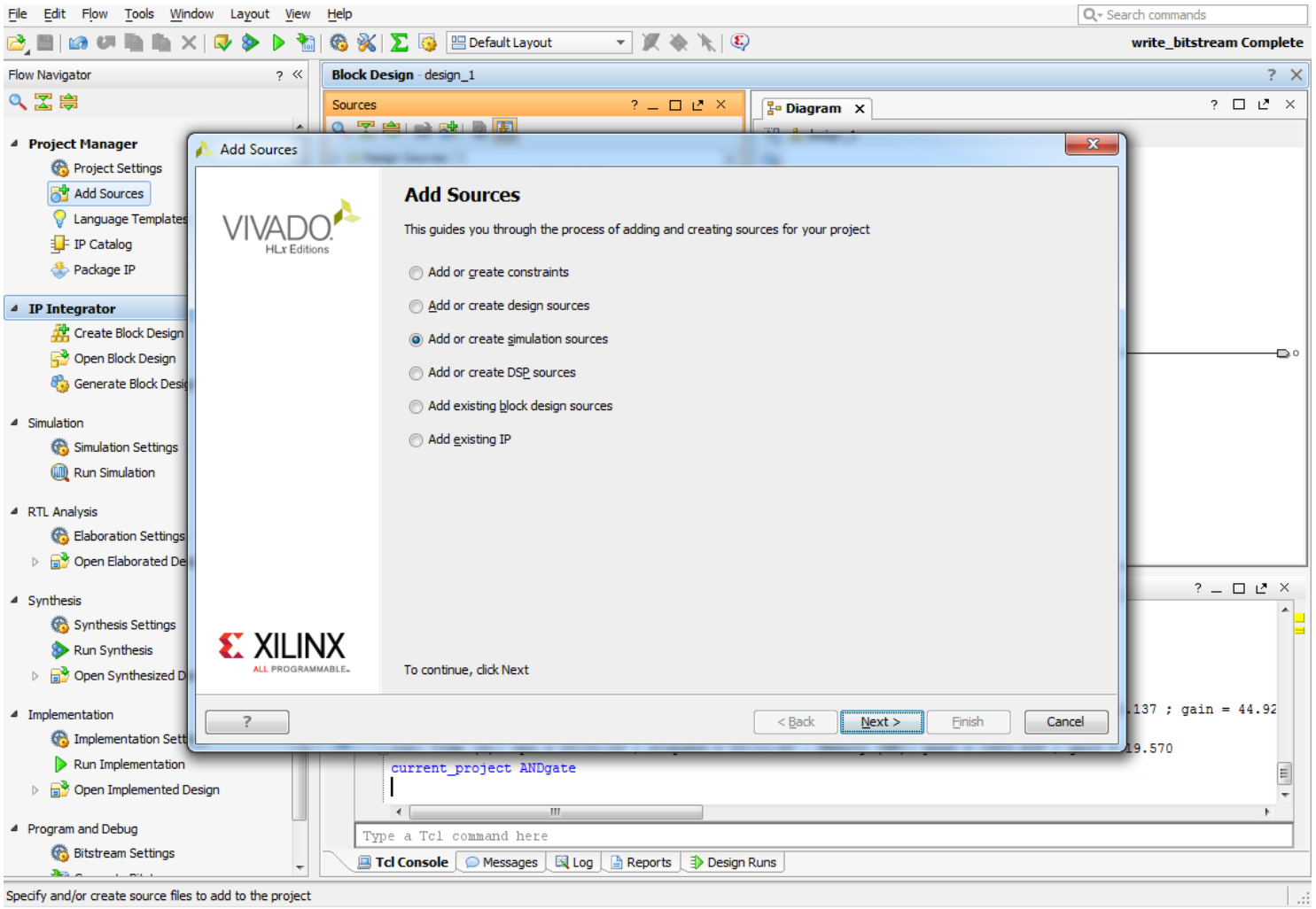


编辑源代码

计算机组织与结构



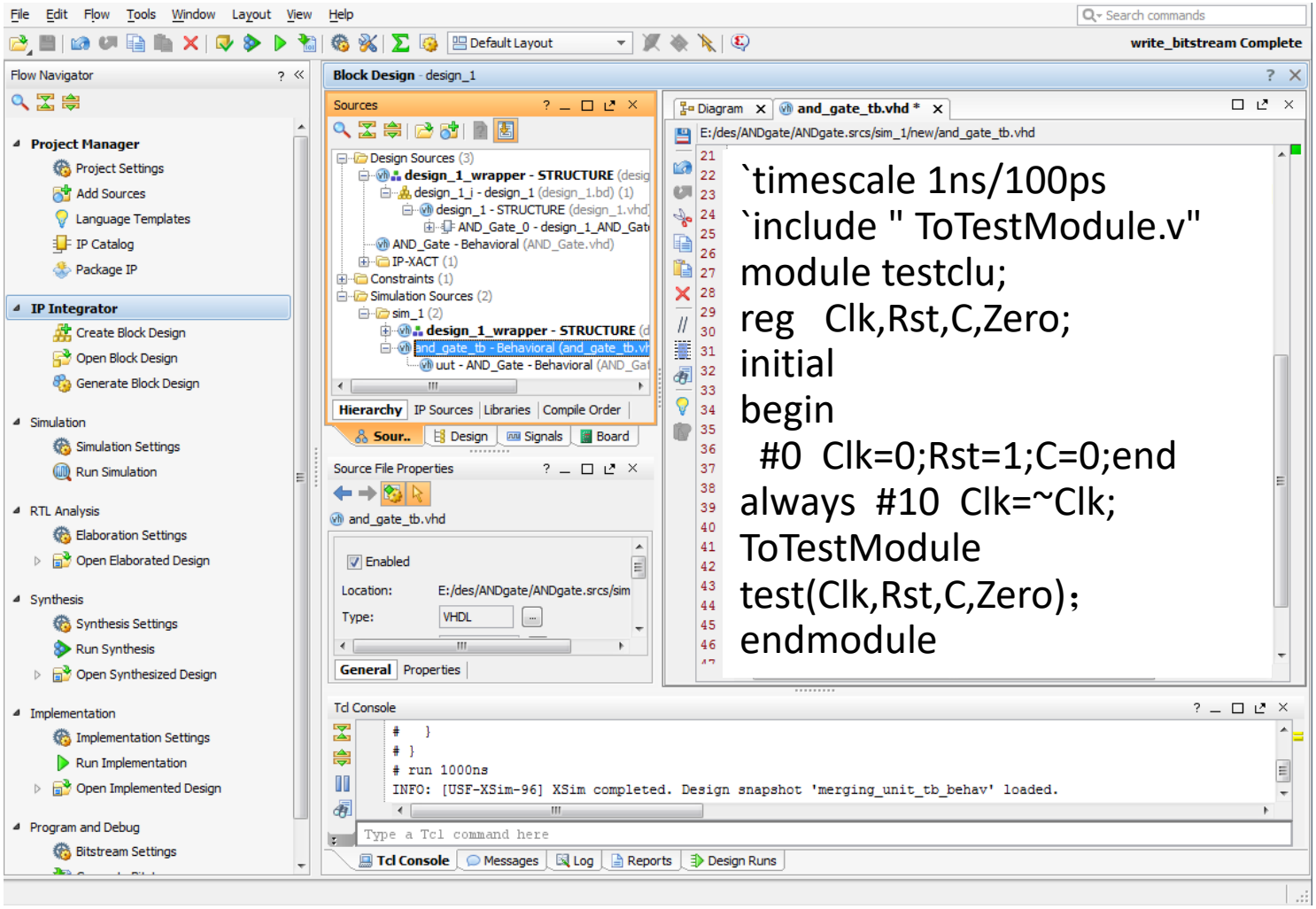
仿真：创建测试文件/testbench



计算机组织与结构

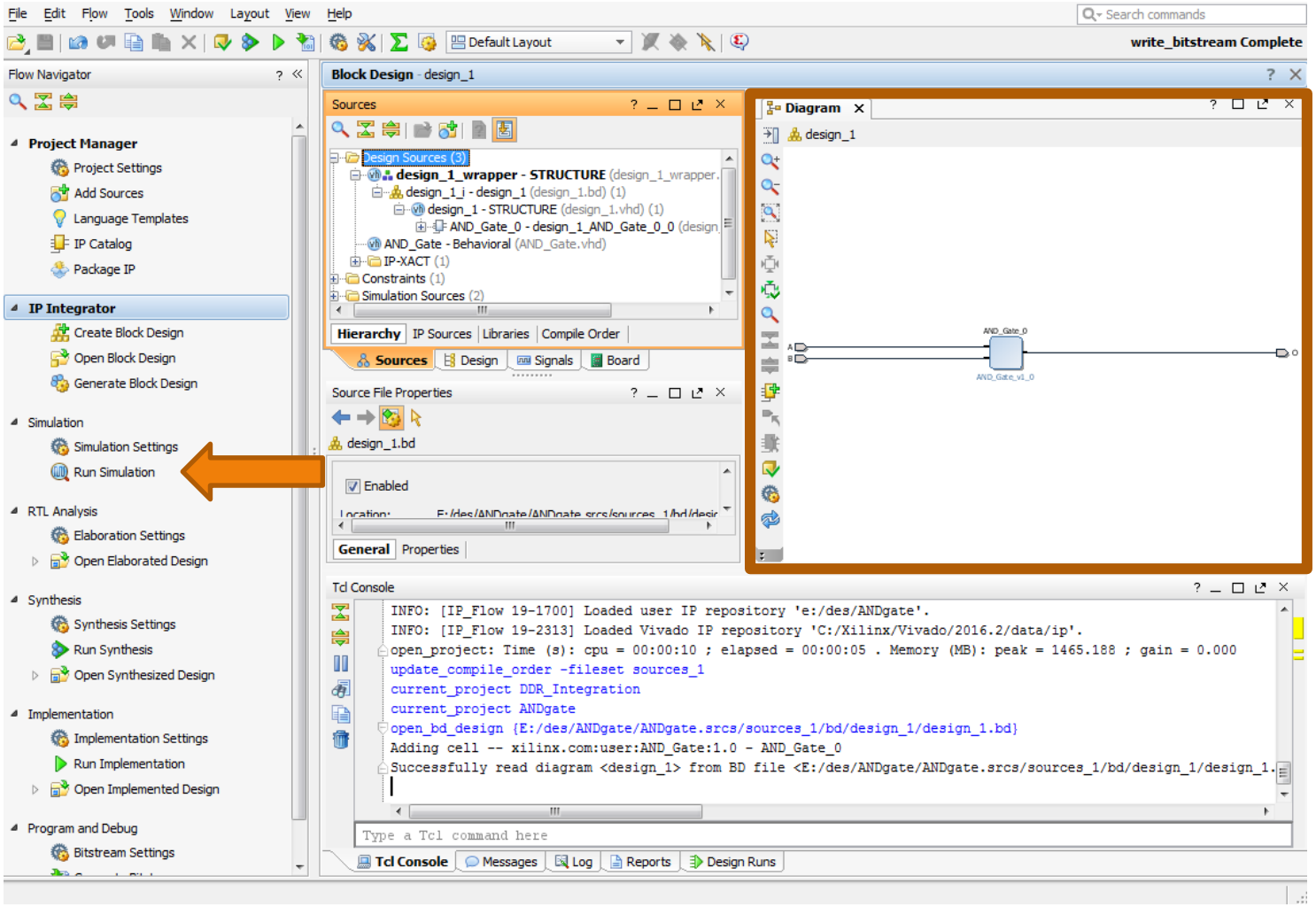
仿真：编辑测试文件/testbench

计算机组织与结构



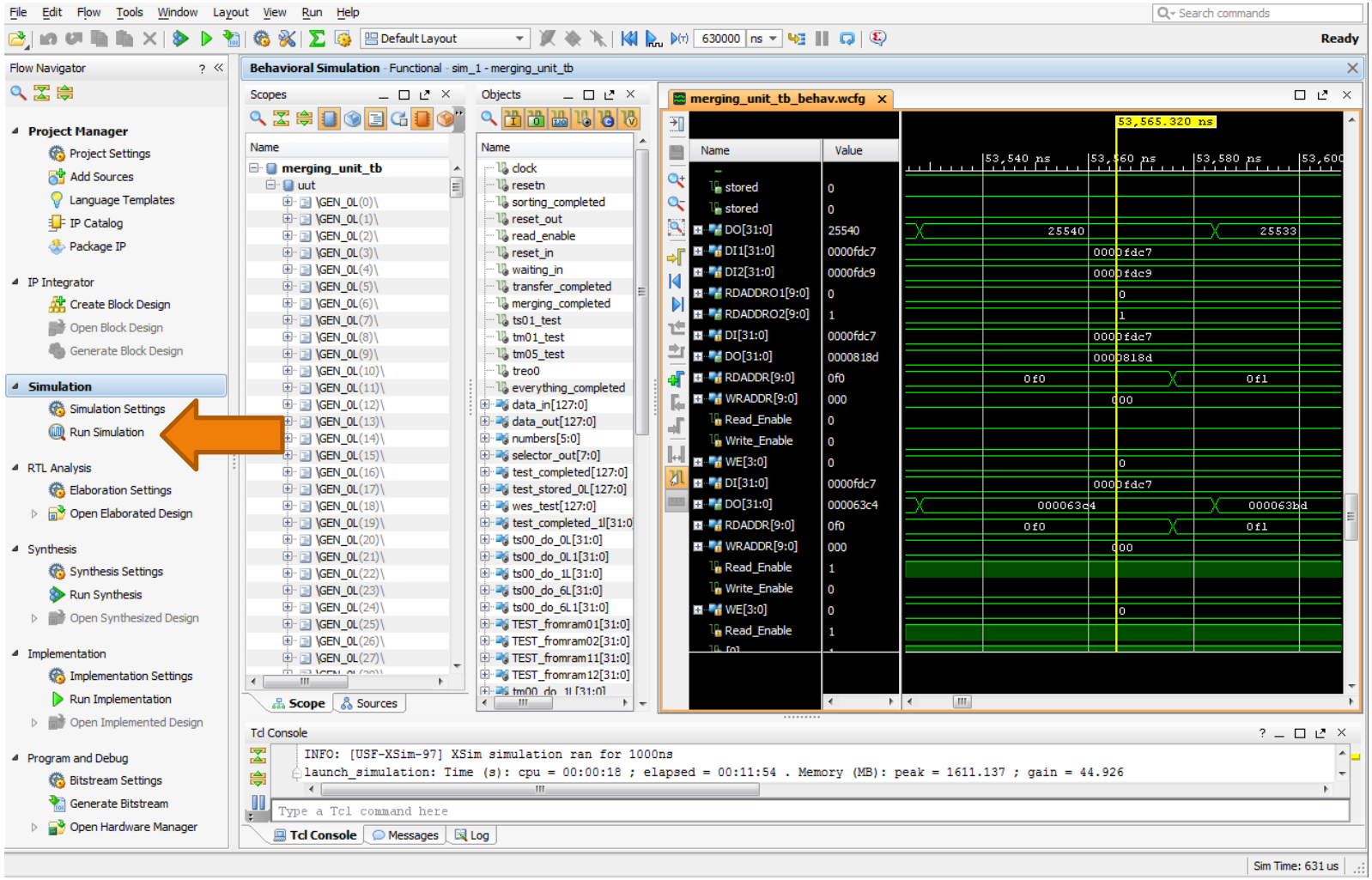
仿真：Simulation

计算机组织与结构



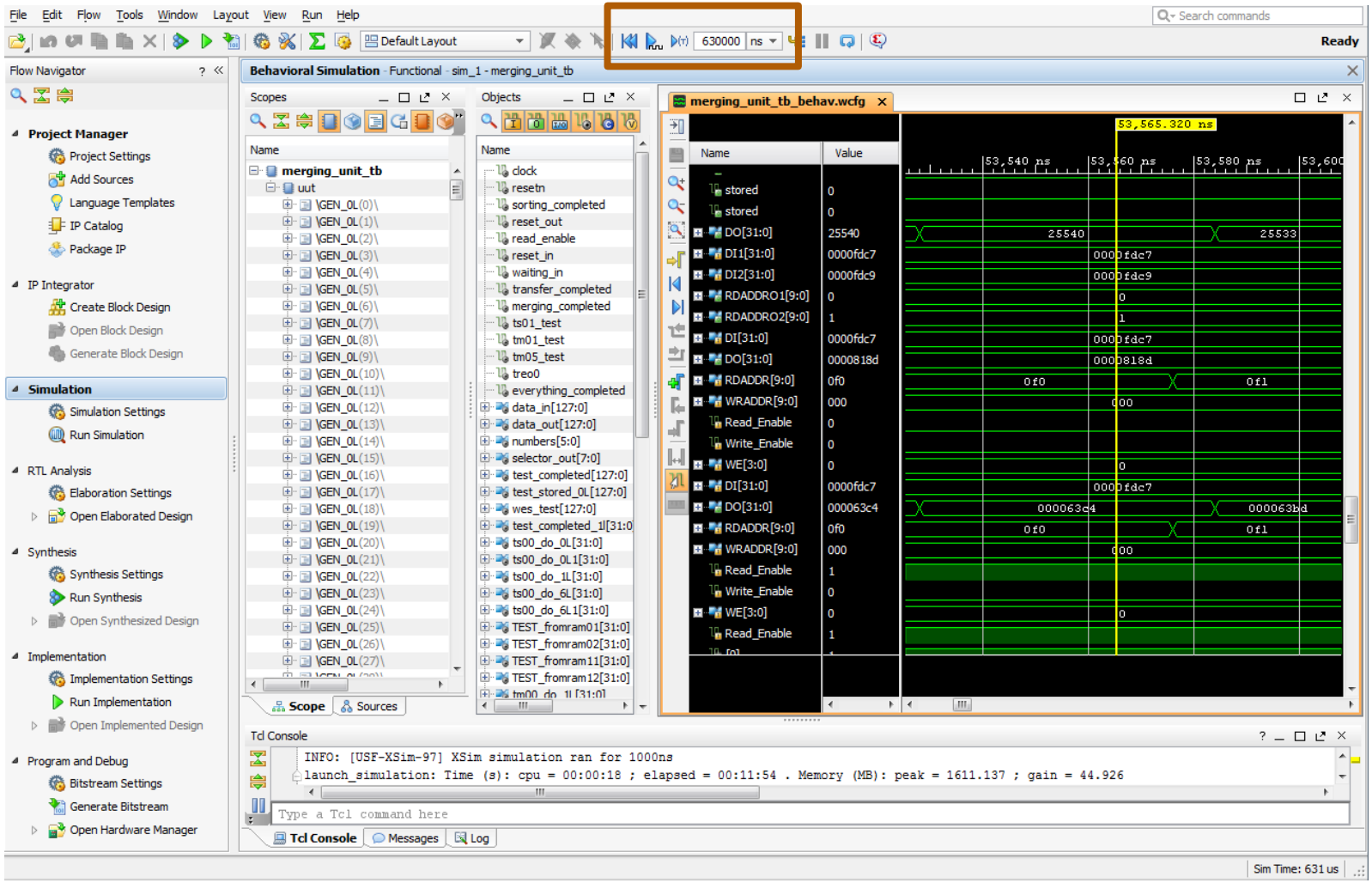
仿真：运行仿真环境

计算机组织与结构



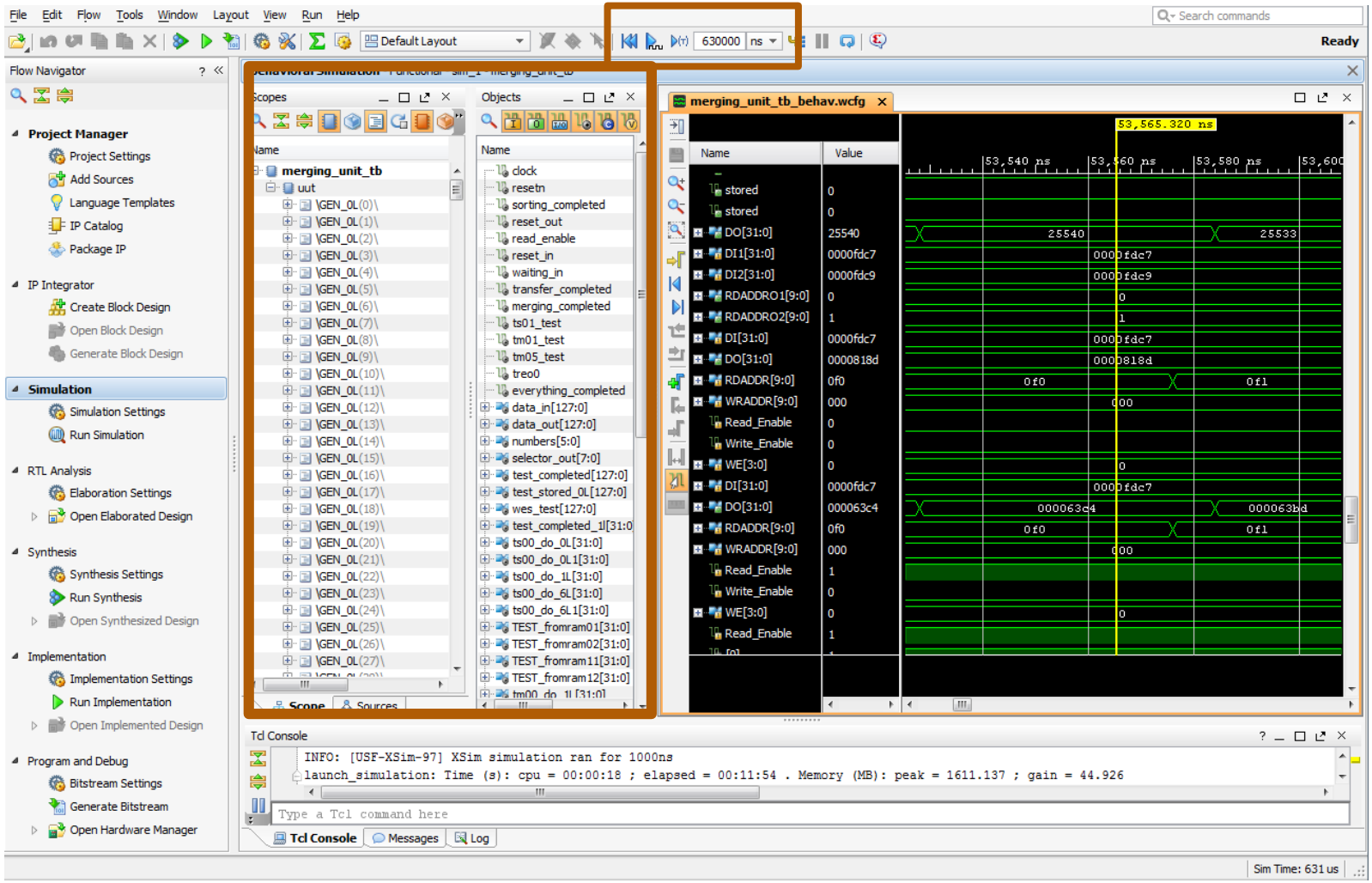
仿真：运行仿真环境

计算机组织与结构



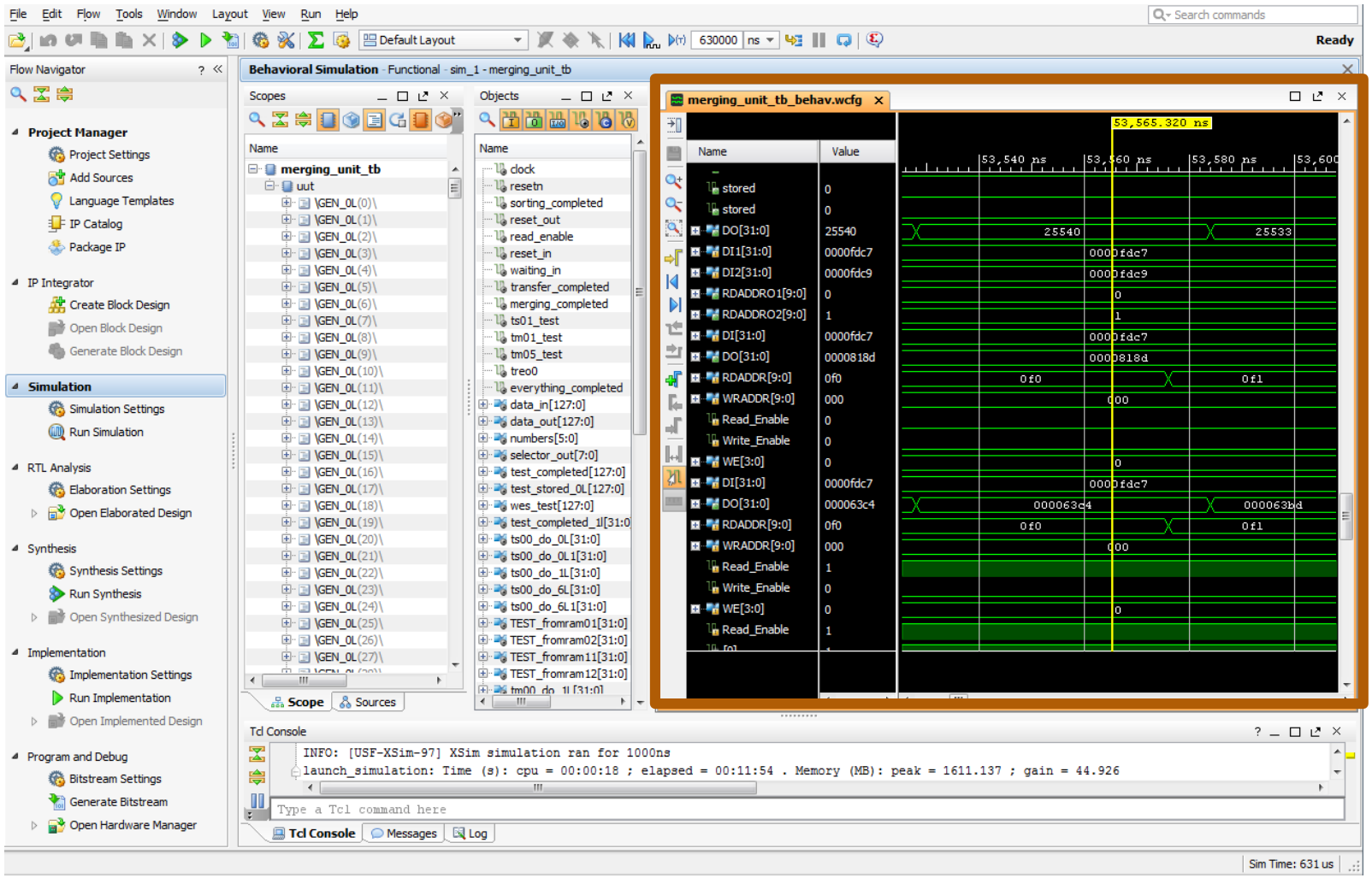
仿真：运行仿真环境

计算机组织与结构

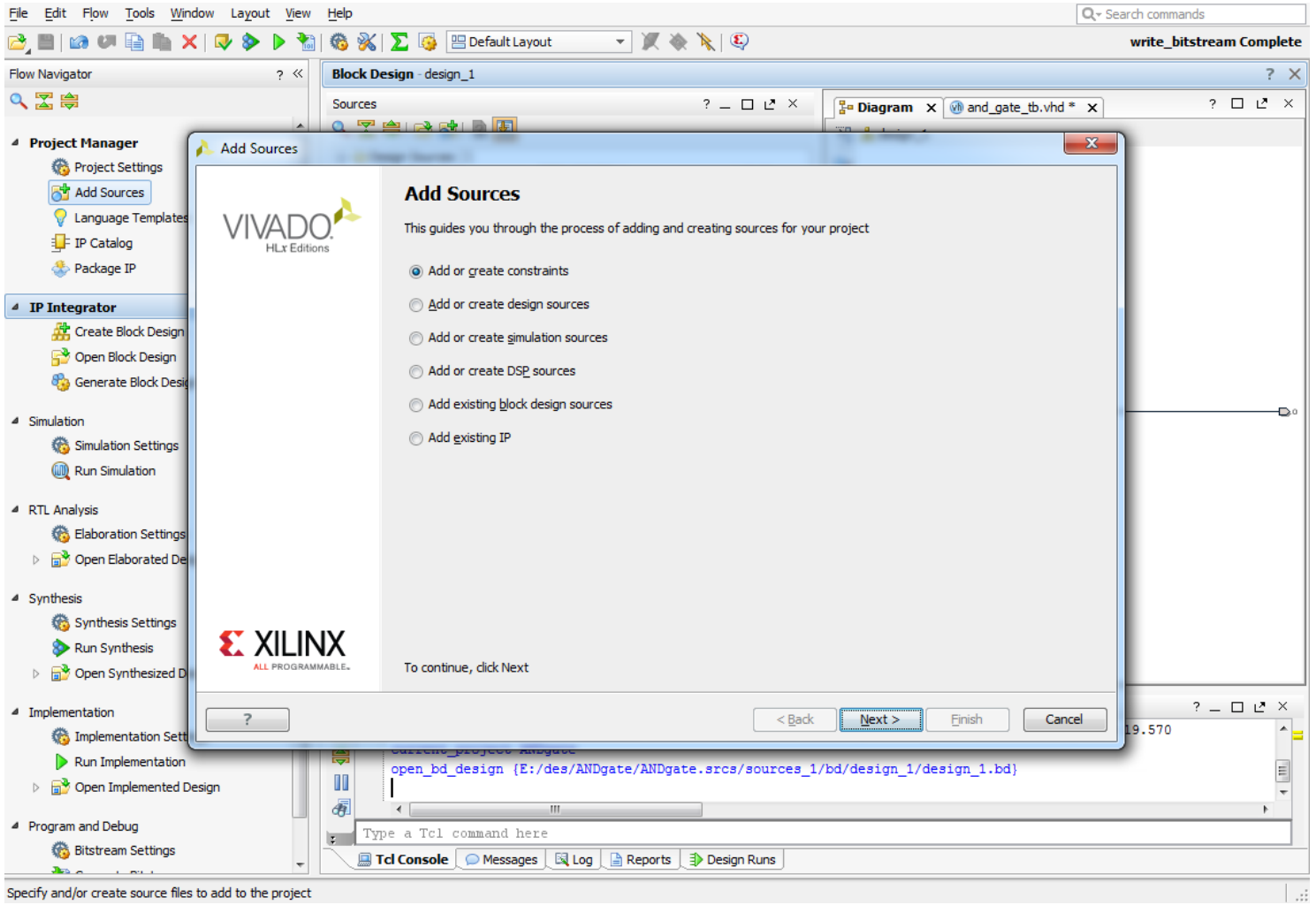


仿真：运行仿真环境

计算机组织与结构



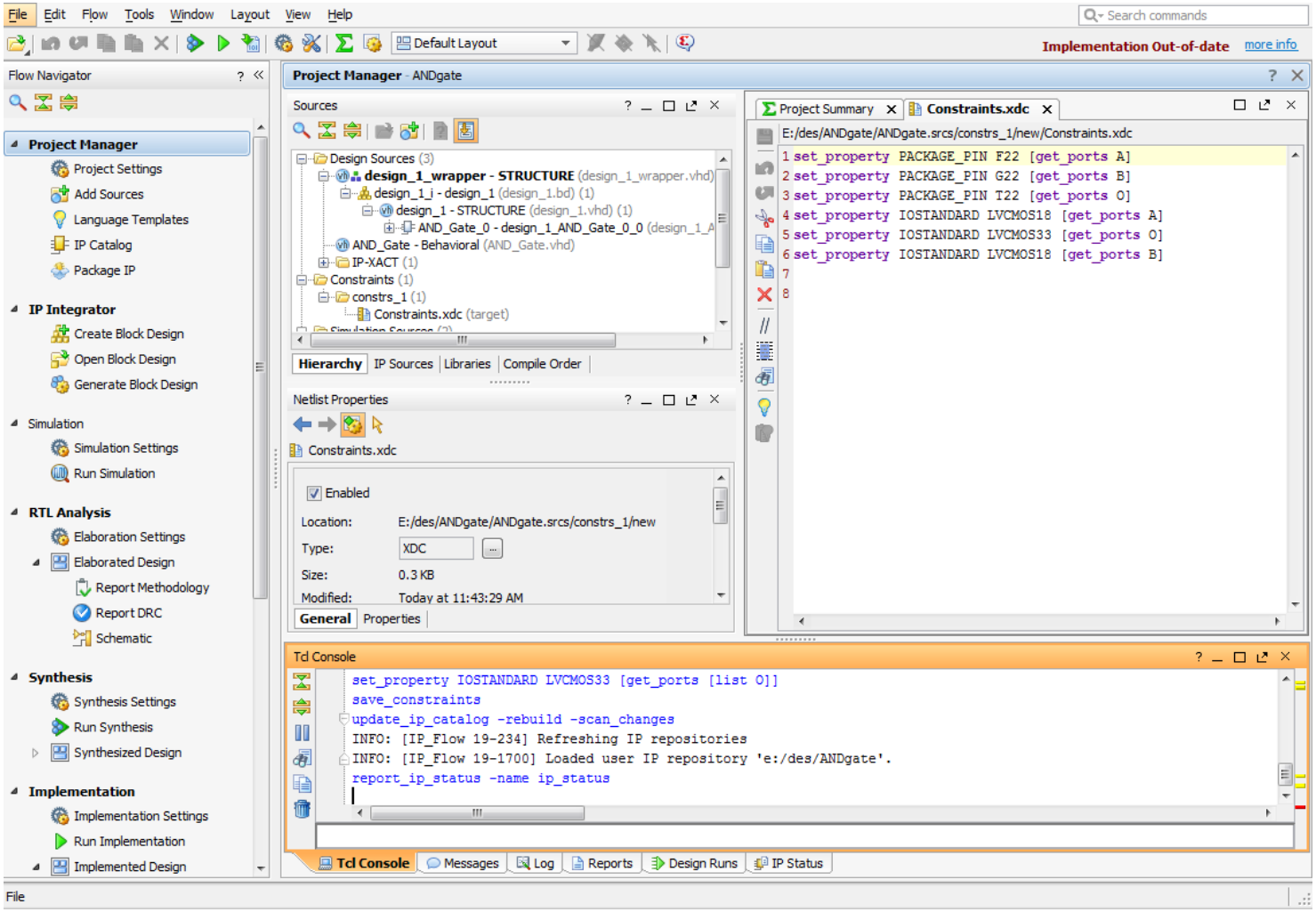
约束文件：创建/添加XDC约束文件



计算机组织与结构

约束文件：编辑XDC约束文件

计算机组织与结构



设计约束/Specifying constraints

- XDC约束文件替代UCF约束
- XDC约束包含：
  - 工业标准Synopsys设计约束（SDC）
  - Xilinx专有物理性约束
- XDC约束特性：
  - 不是简单的字符串，而是遵循Tcl语法的命令
  - 像Vivado Tcl解释器的任何其他Tcl命令一样解释
  - 与其他Tcl命令相同的方式读入与解析

约束文件：UCF向XDC的迁移

UCF	SDC
TIMESPEC PERIOD	create_clock create_generated_clock
OFFSET = IN <x> BEFORE <clk>	set_input_delay
OFFSET = OUT <x> BEFORE <clk>	set_output_delay
FROM:TO "TS_"*2	set_multicycle_path
FROM:TO	set_max_delay
TIG	set_false_path
NET "clk_p" LOC = AD12	set_property LOC AD12 [get_ports clk_p]
NET "clk_p" IOSTANDARD = LVDS	set_property IOSTANDARD LVDS [get_ports clk_p]

Source: Vivado Design Suite Migration Methodology Guide (UG911) p 23

约束文件：举例

结合开发板的XDC文件进行解释

ISE约束文件：ucf

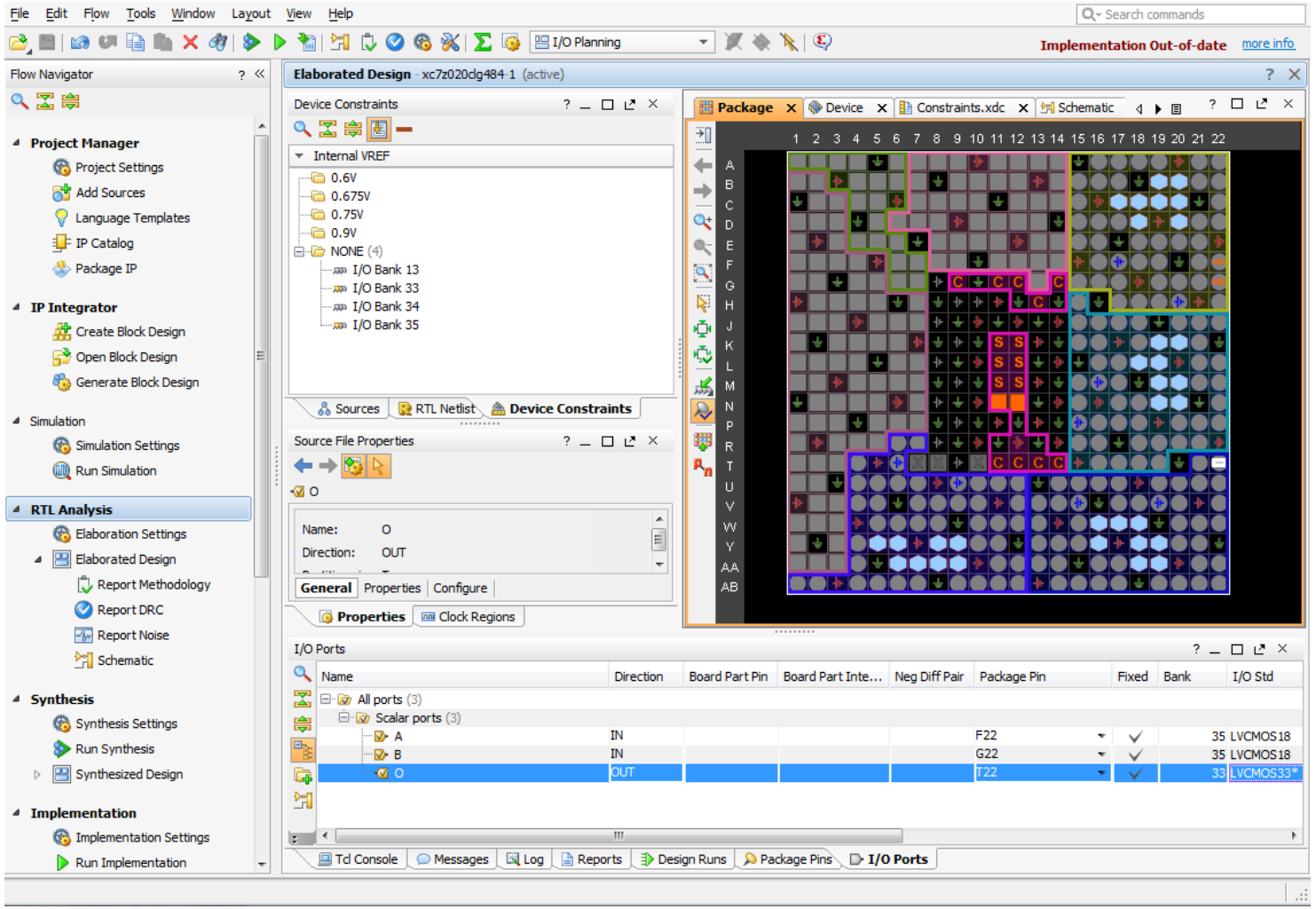
```
# Buttons
NET "btnc"          LOC=N17 | IOSTANDARD=LVCMOS33;
# LEDs
NET "led<0>"        LOC=H17 | IOSTANDARD=LVCMOS33;
```

Vivado约束文件：xdc

```
#Buttons
set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { BTNC }];
# LEDs
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { LED[0] }];
```

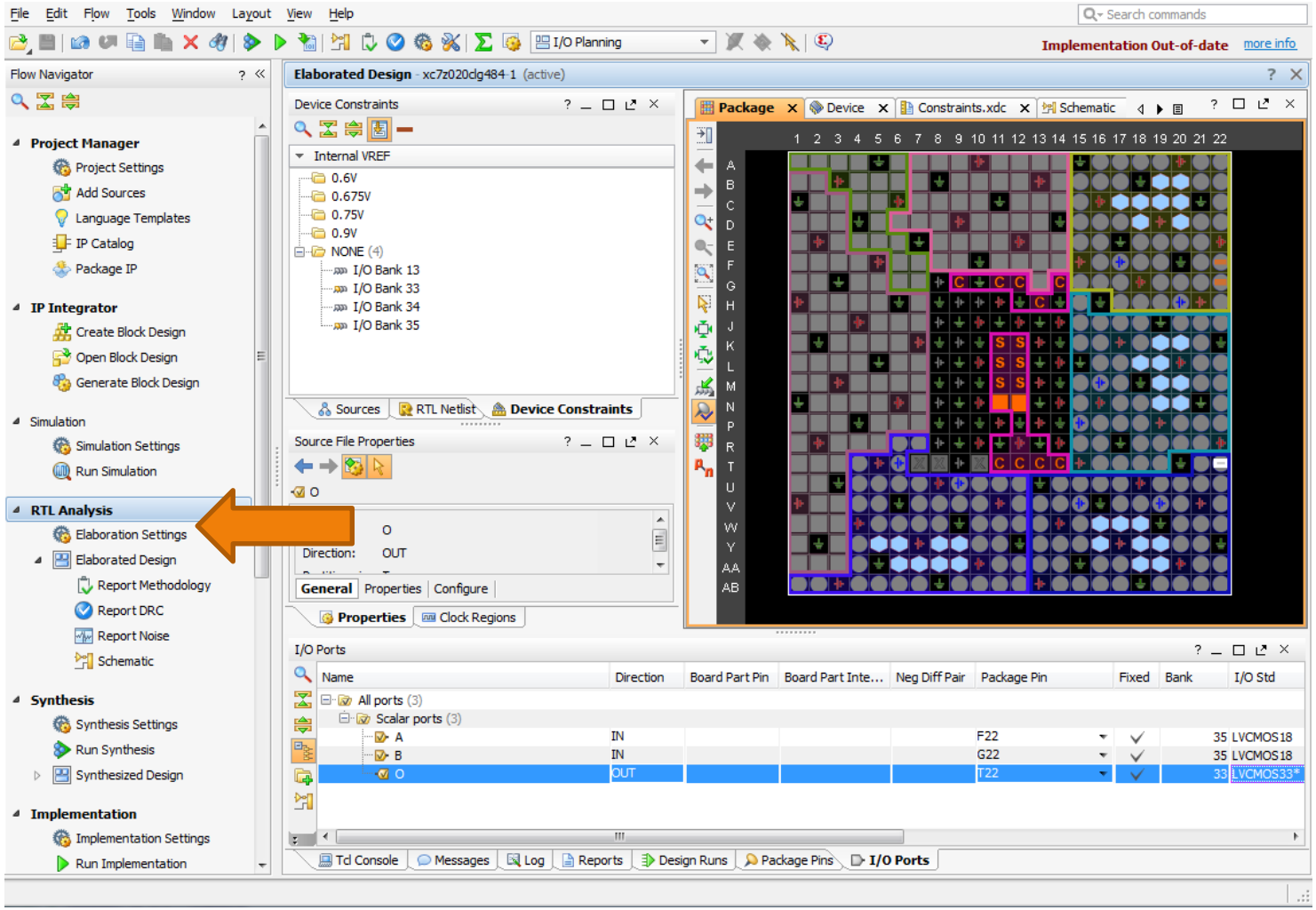
约束：输入输出规划IO Planning

计算机组织与结构



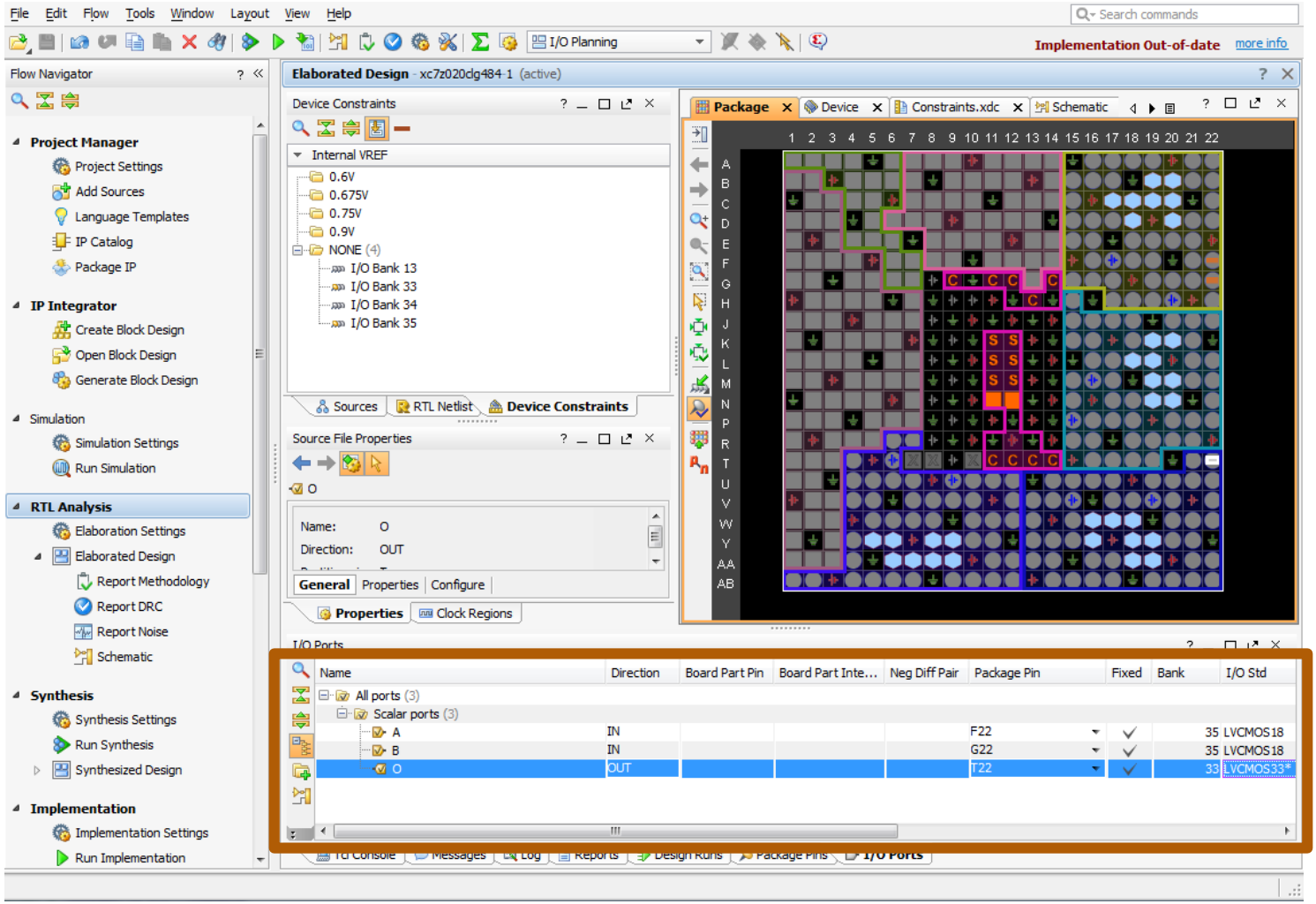
约束：输入输出规划IO Planning

计算机组织与结构



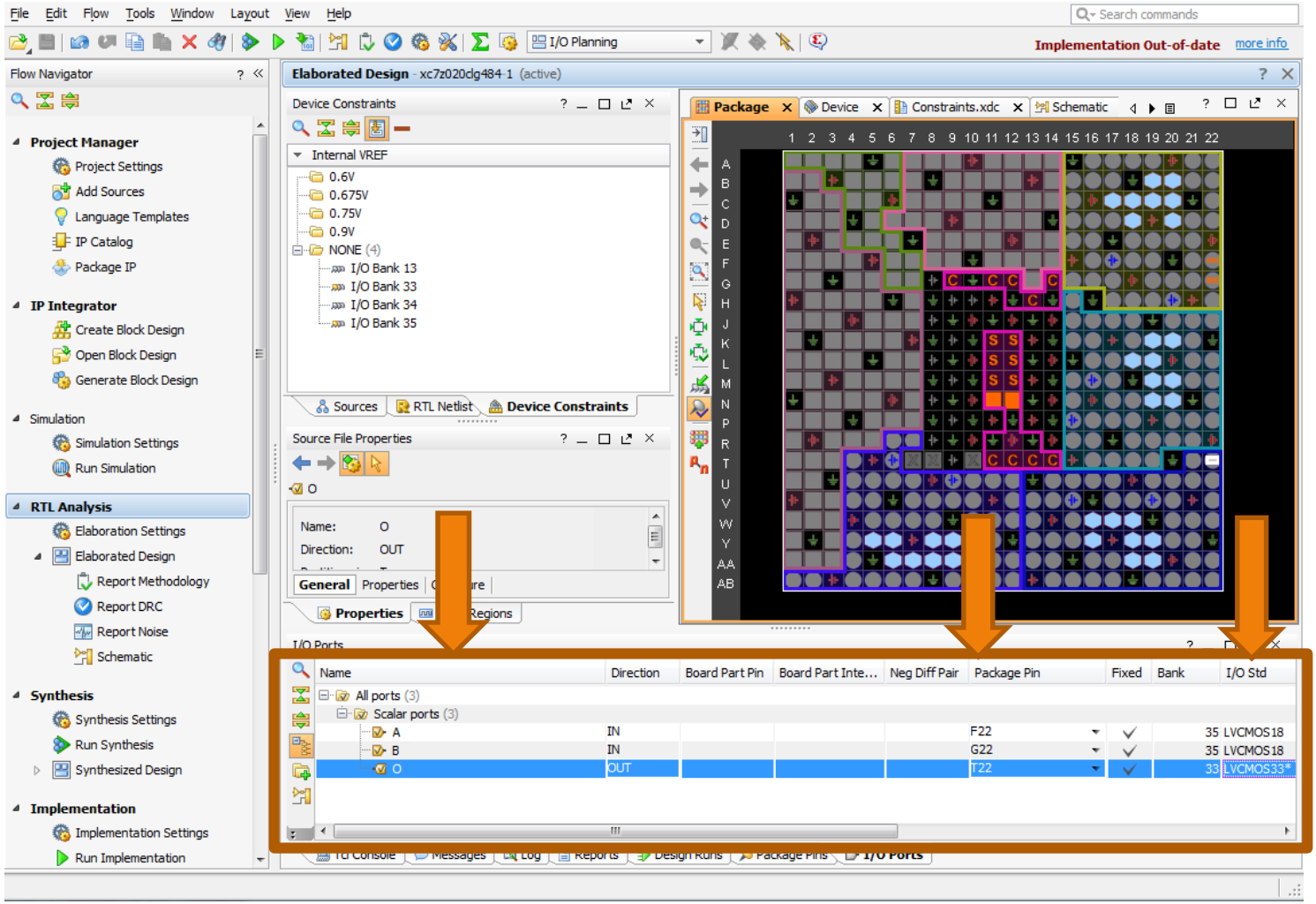
约束：输入输出规划IO Planning

计算机组织与结构



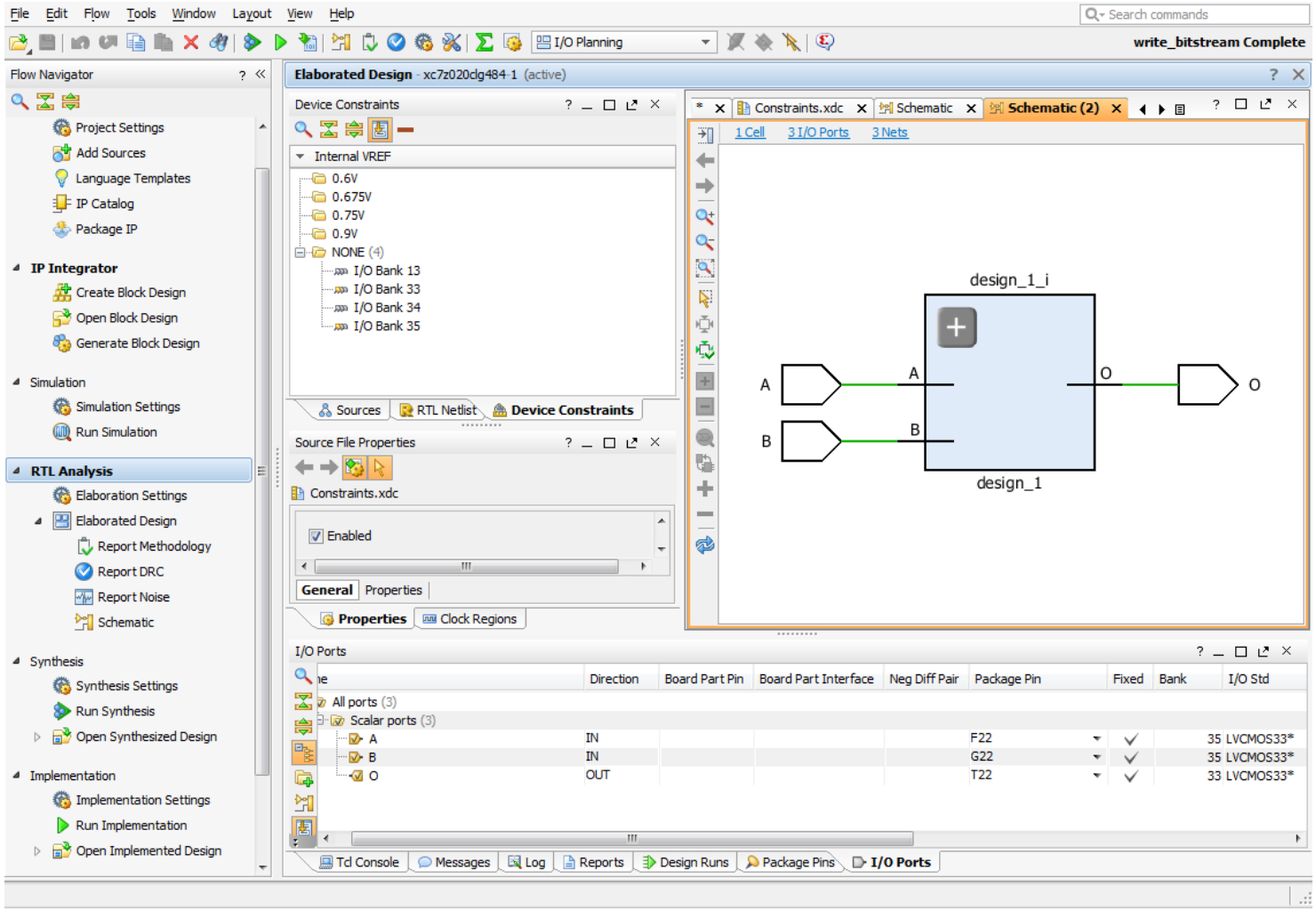
约束：输入输出规划IO Planning

计算机组织与结构



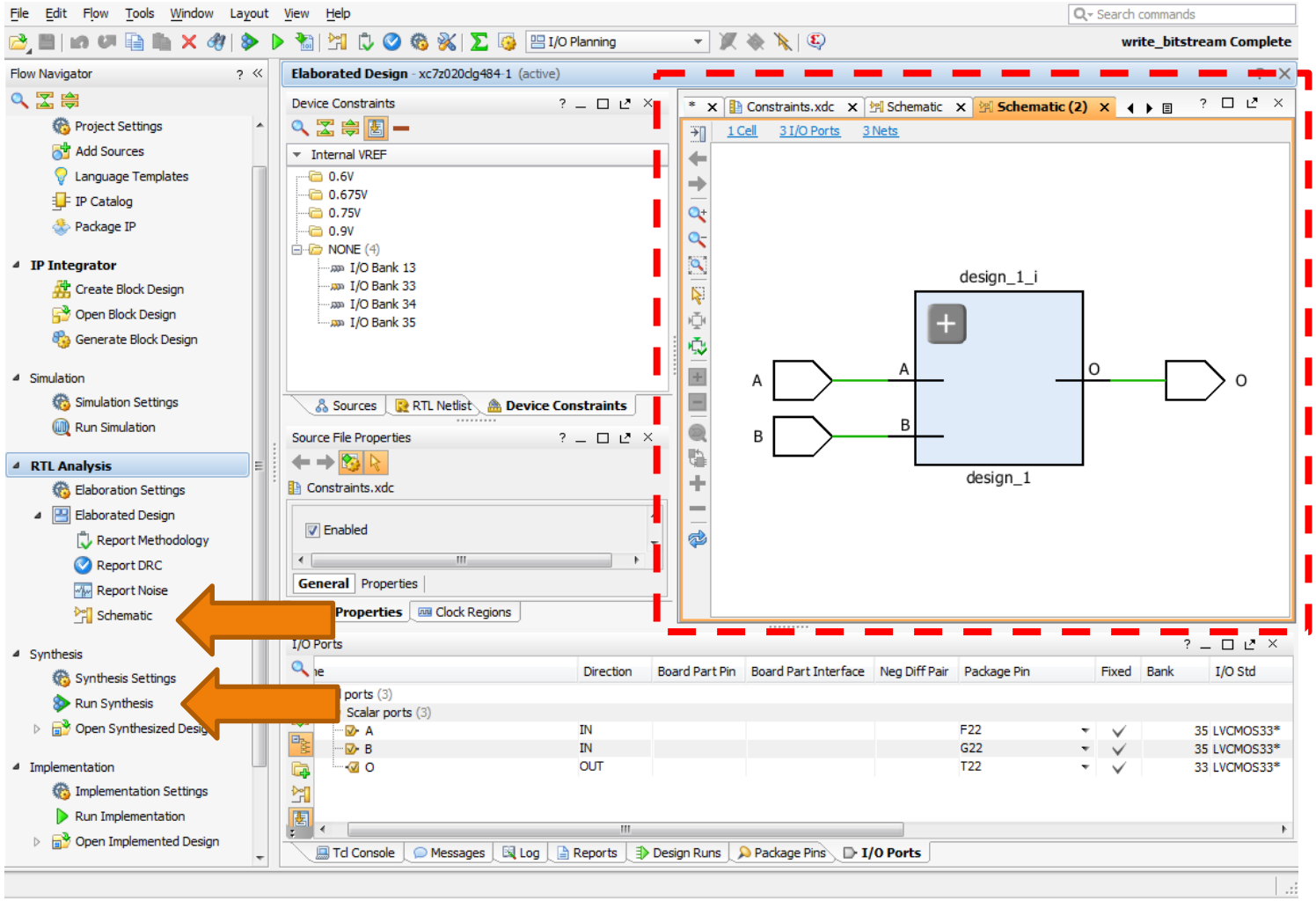
精细设计-原理图

计算机组织与结构



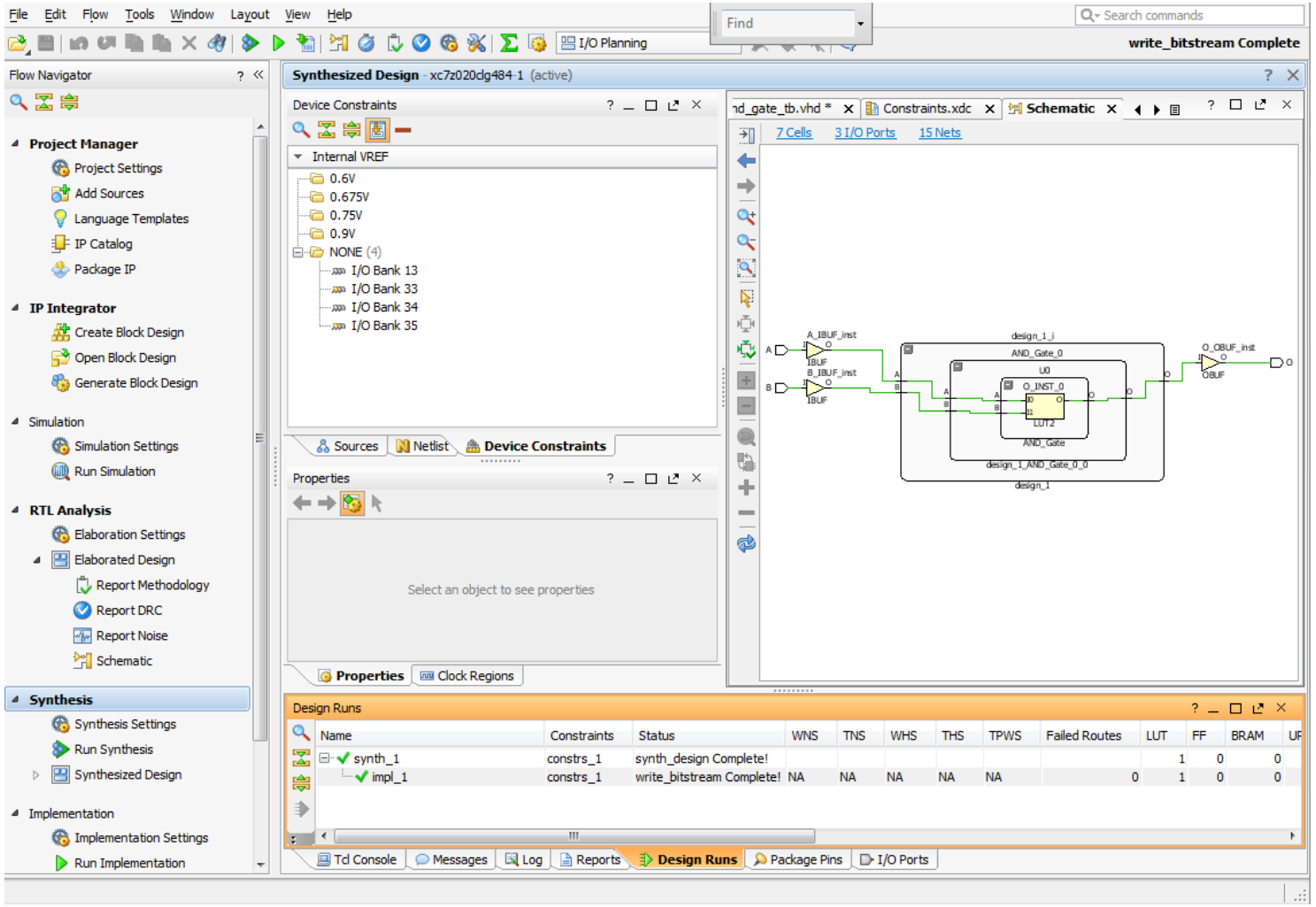
精细设计-原理图

计算机组织与结构



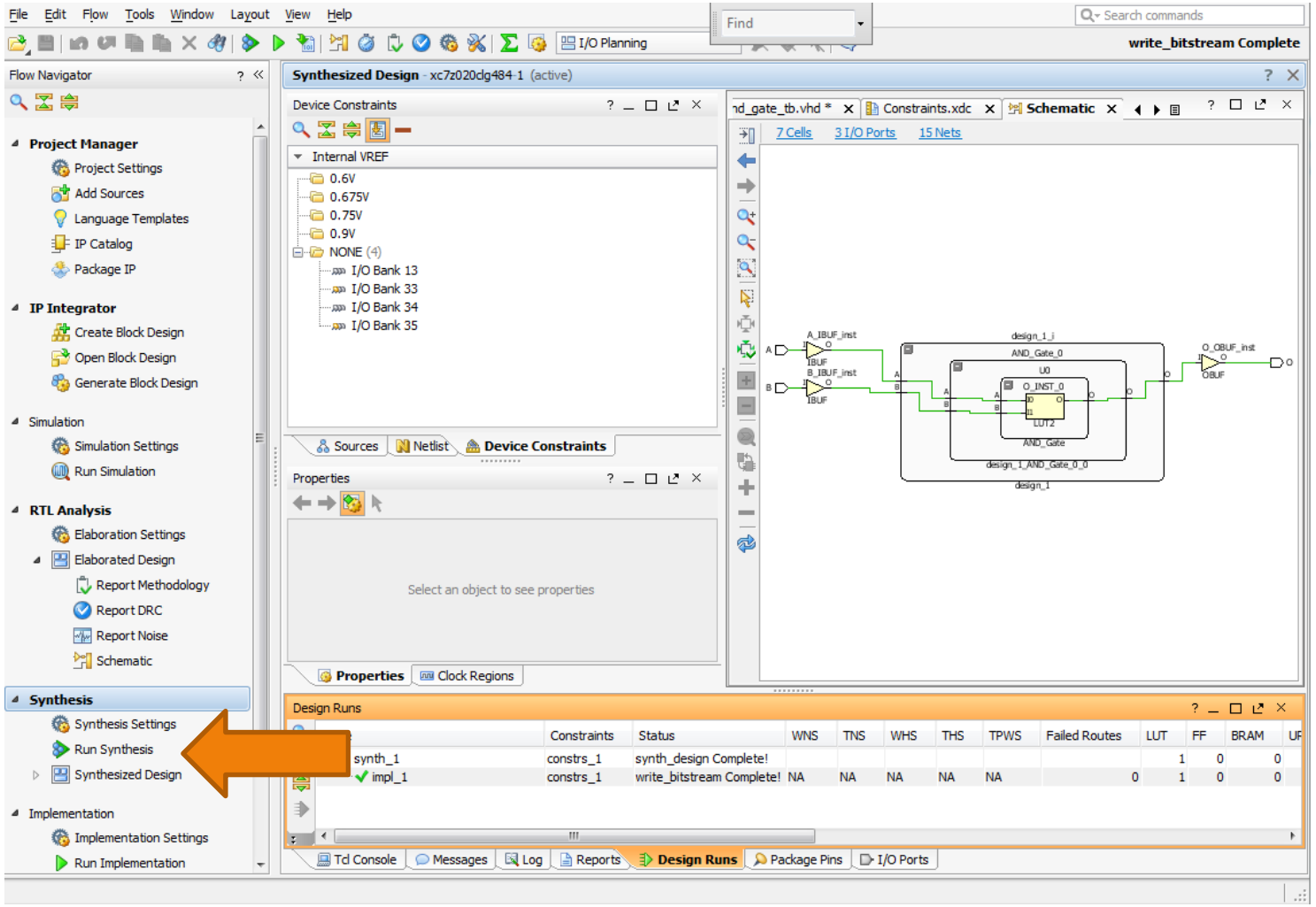
综合/Synthesis

计算机组织与结构



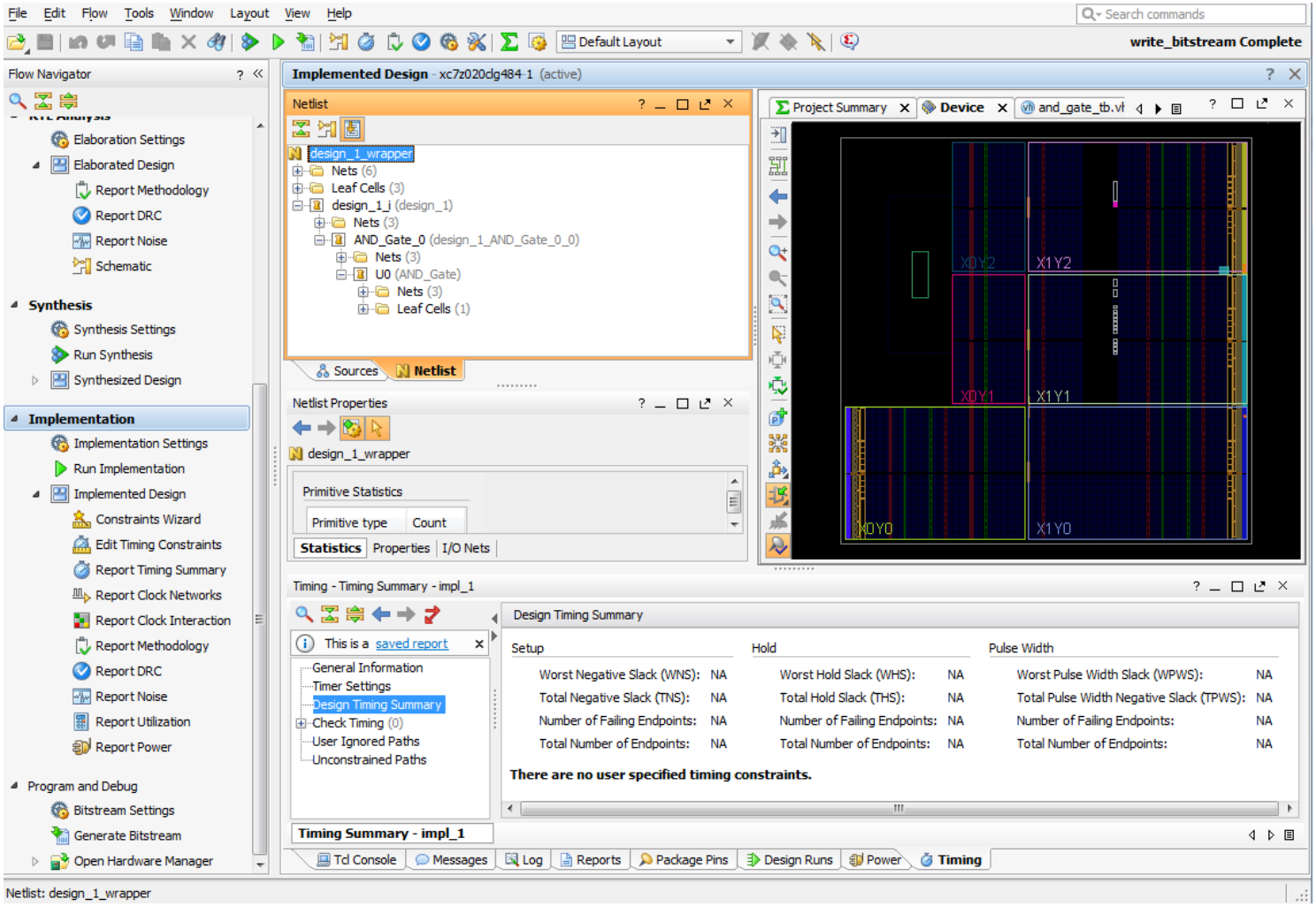
综合/Synthesis

计算机组织与结构



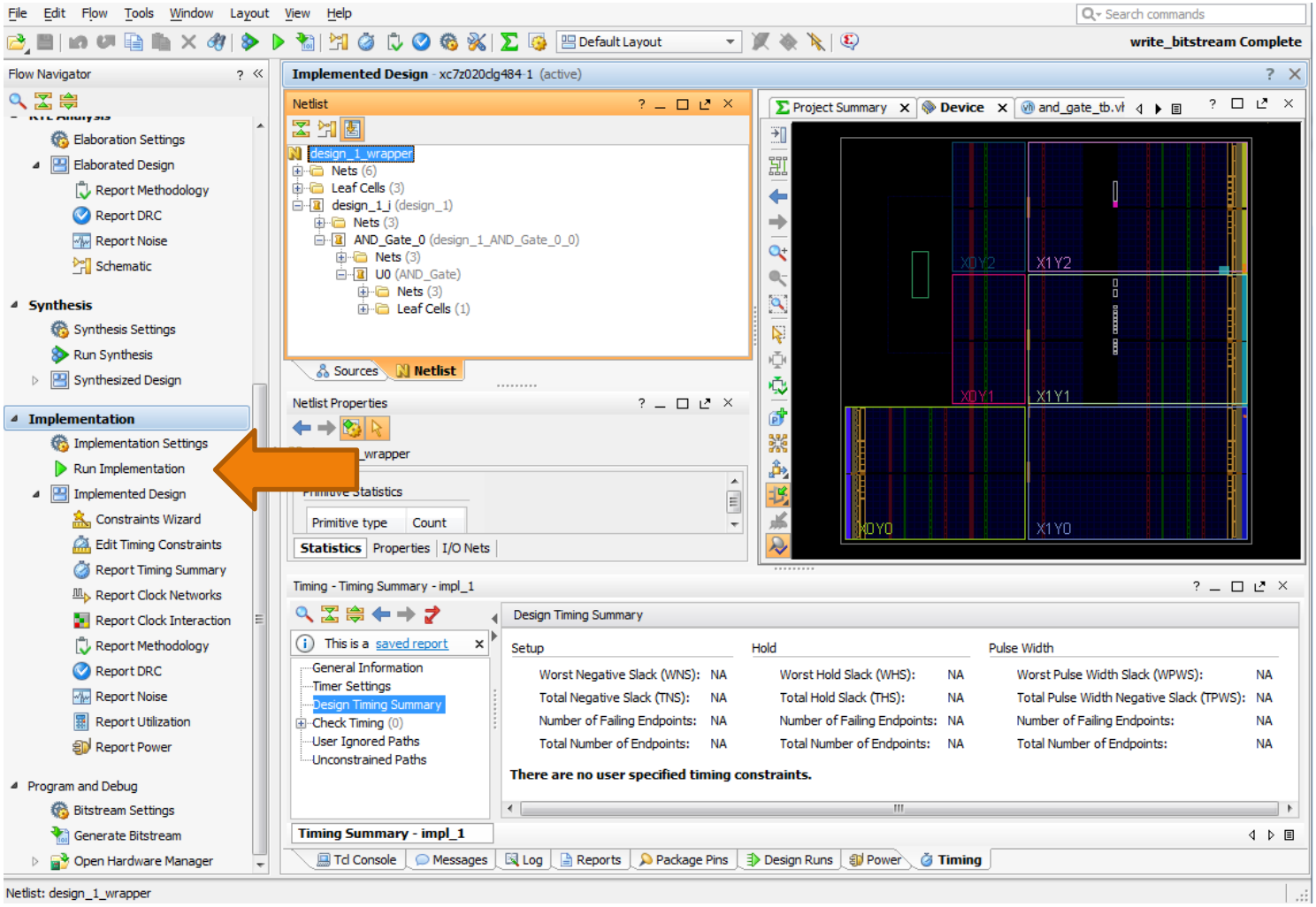
应用/Implementation

计算机组织与结构



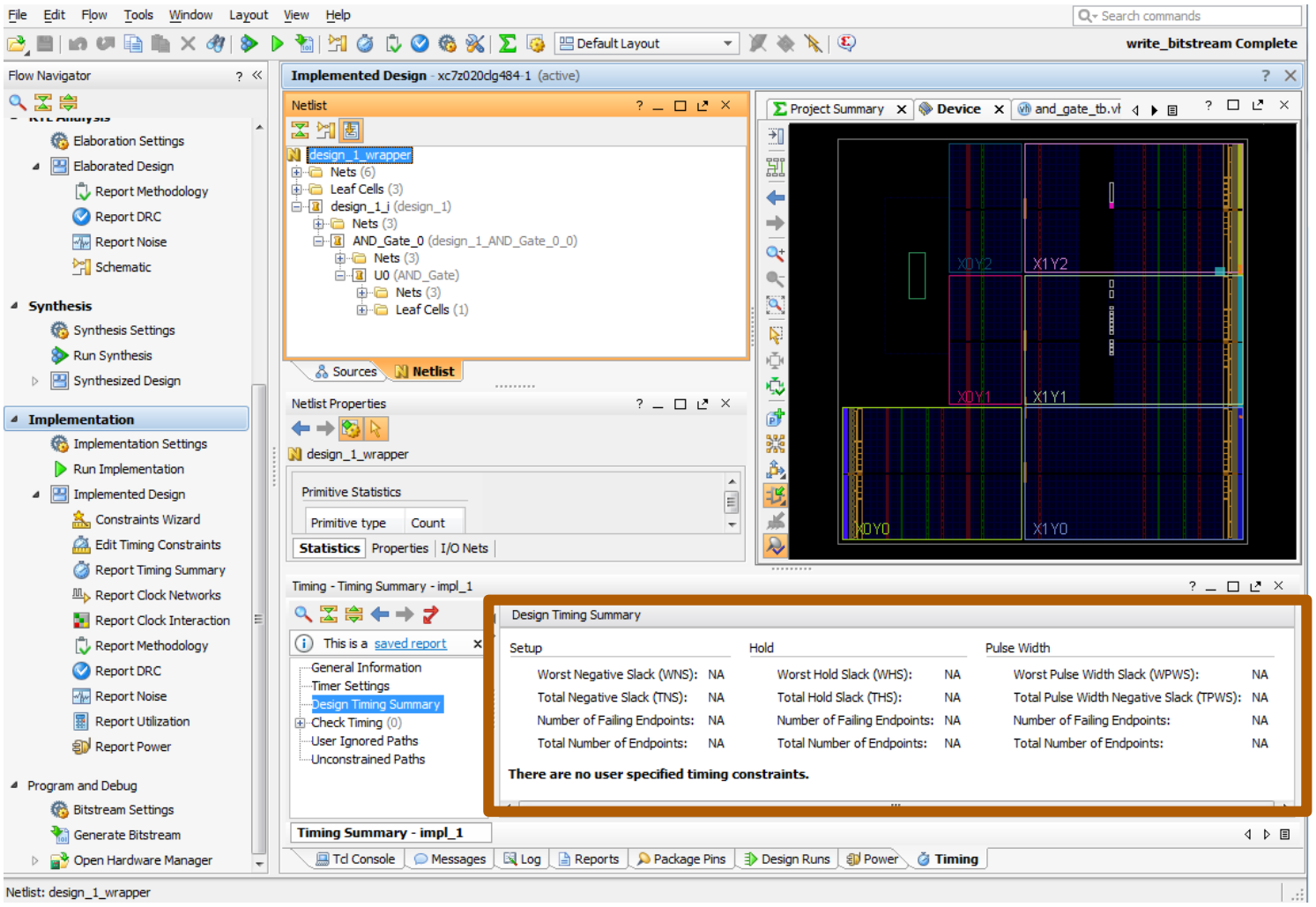
应用/Implementation

计算机组织与结构



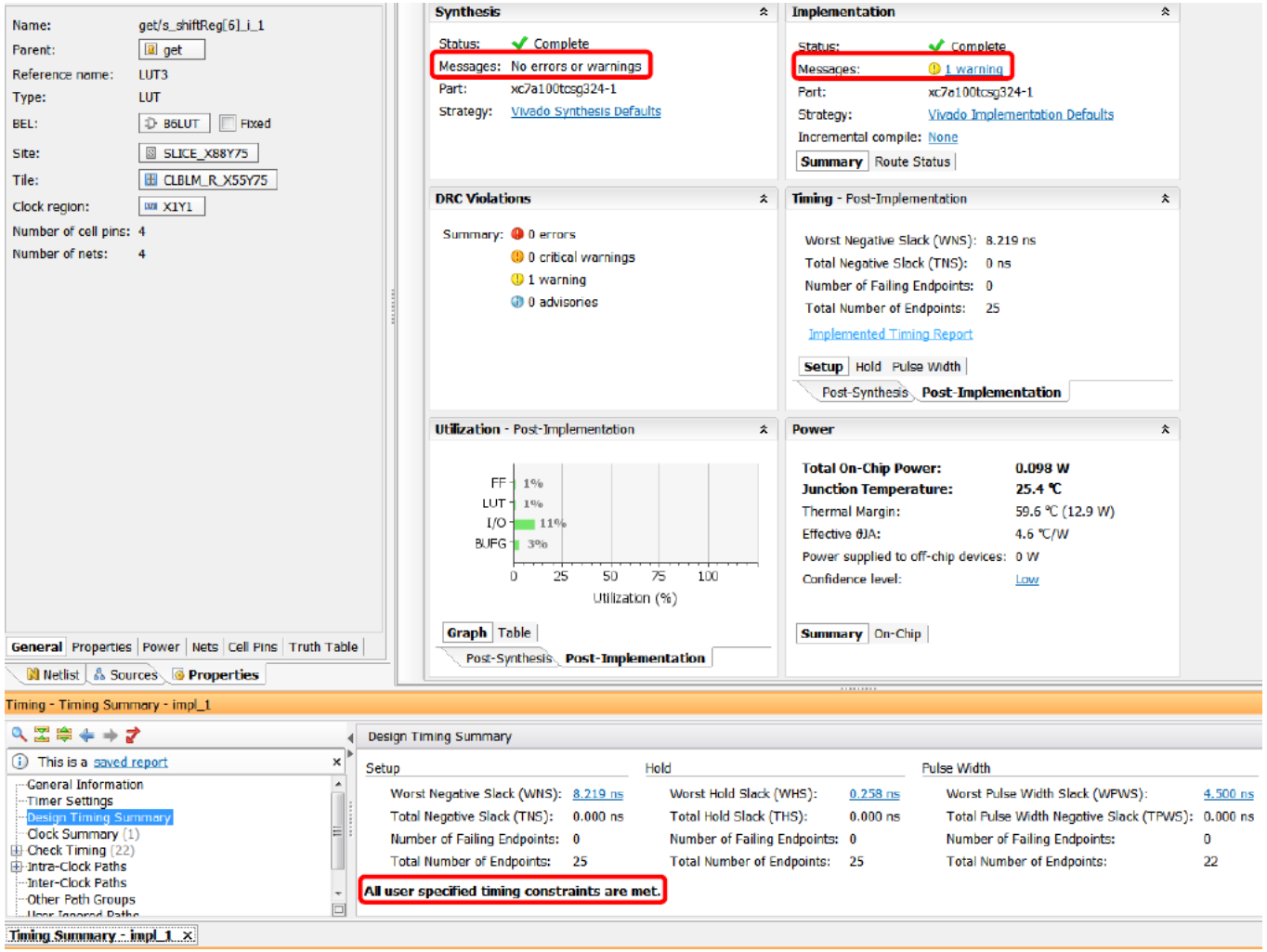
应用/Implementation

计算机组织与结构



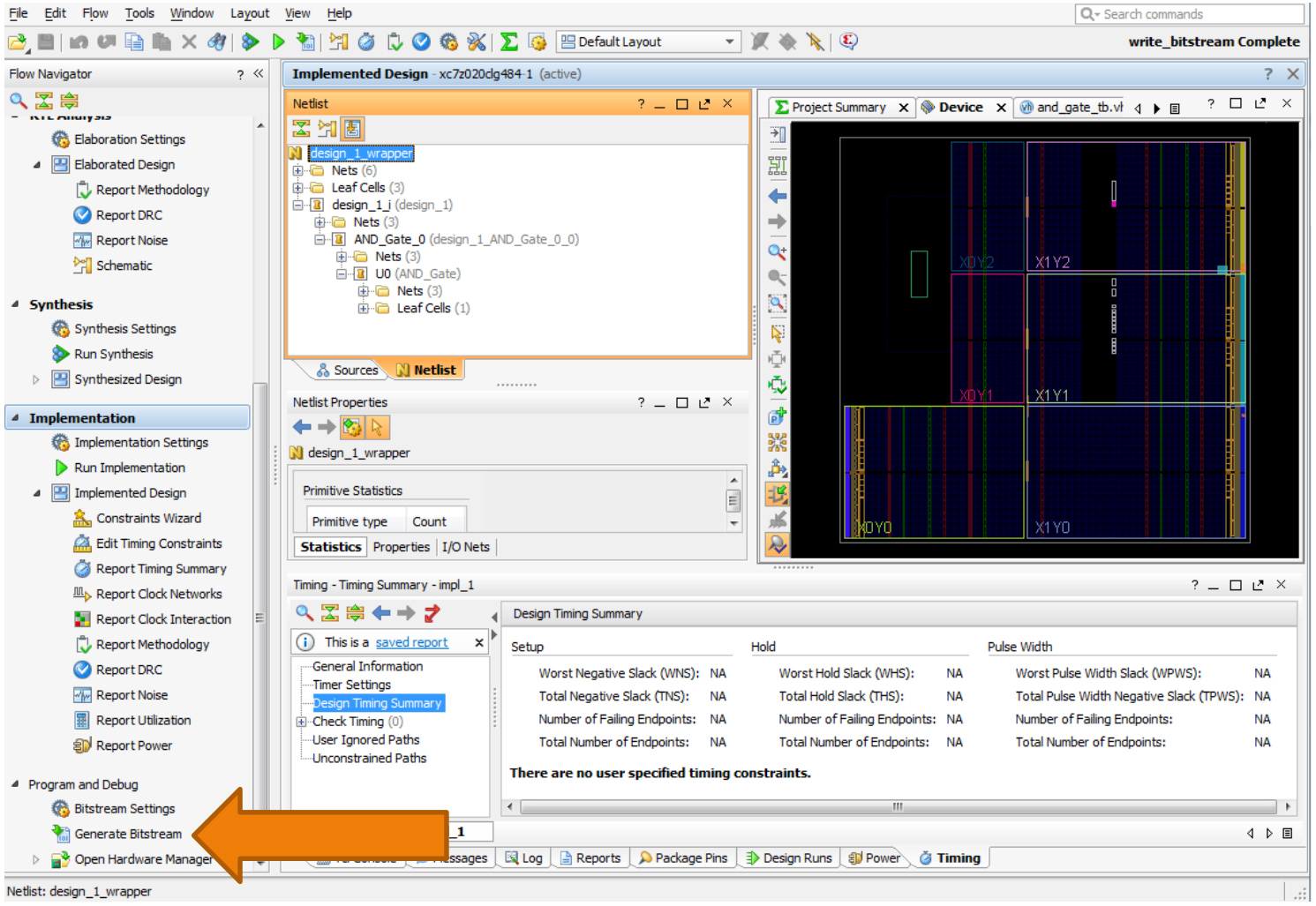
产生设计报告

计算机组织与结构



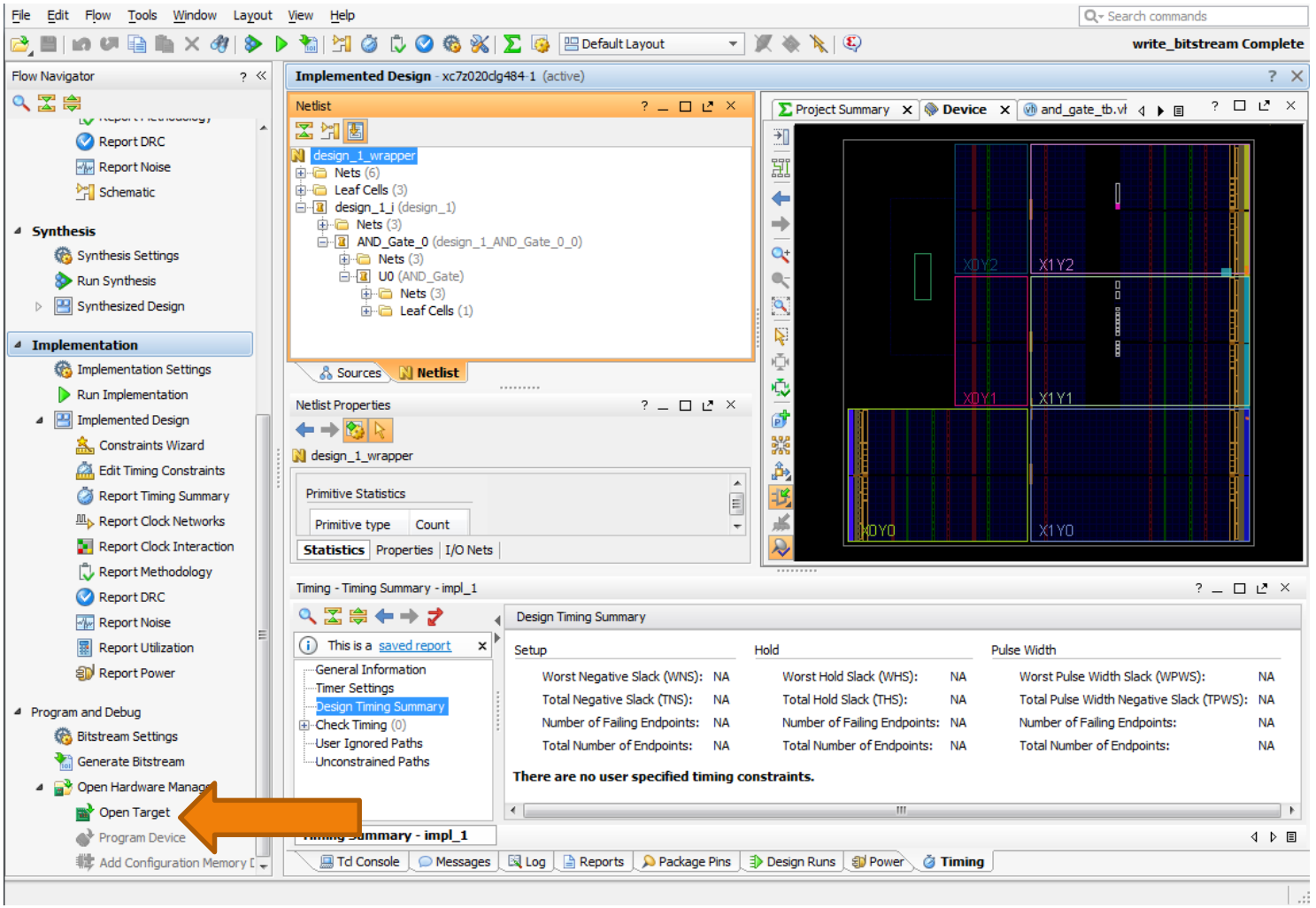
生成Bit流文件
Generating Bitstream

计算机组织与结构

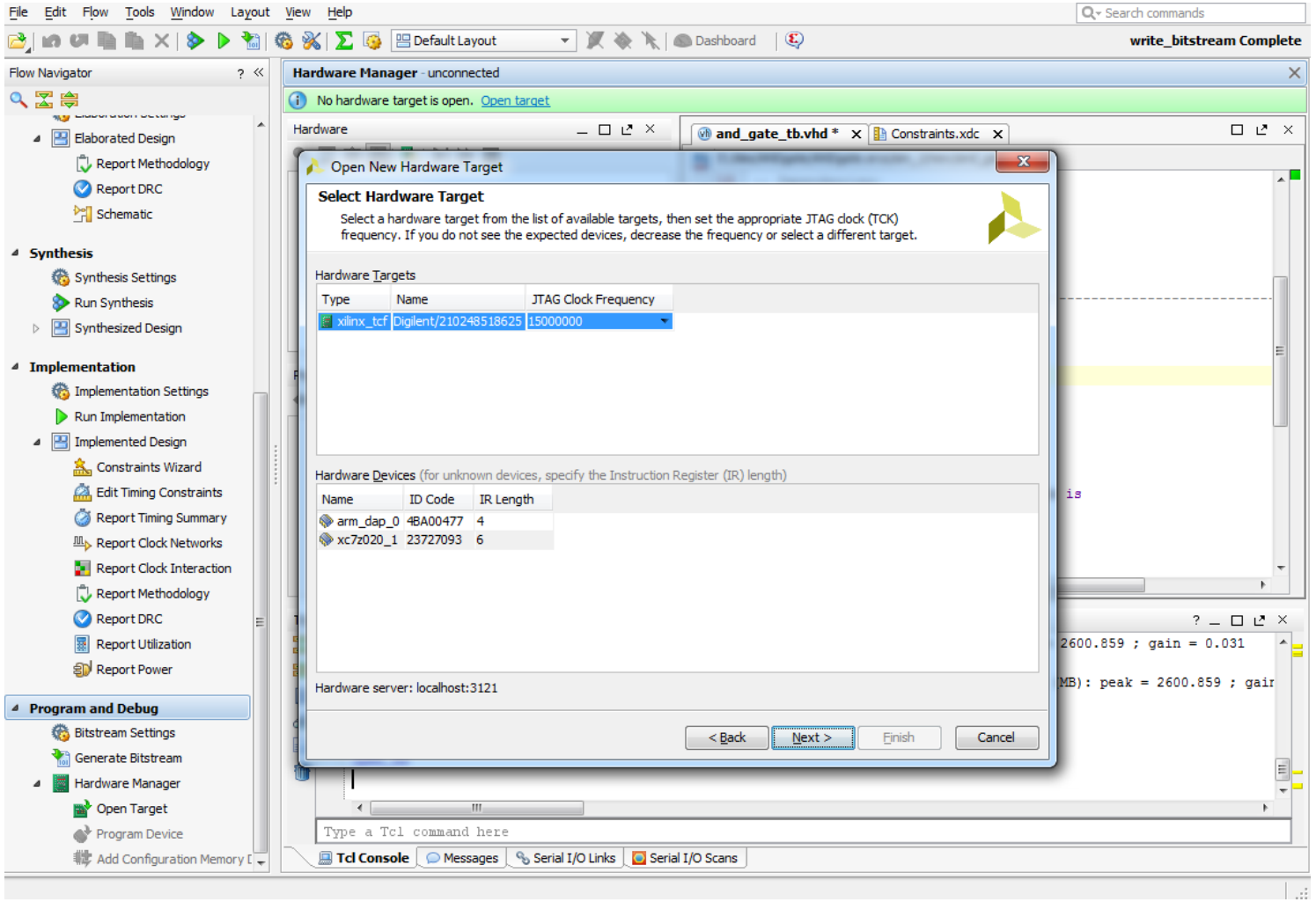


硬件管理器：打开目标设备

计算机组织与结构



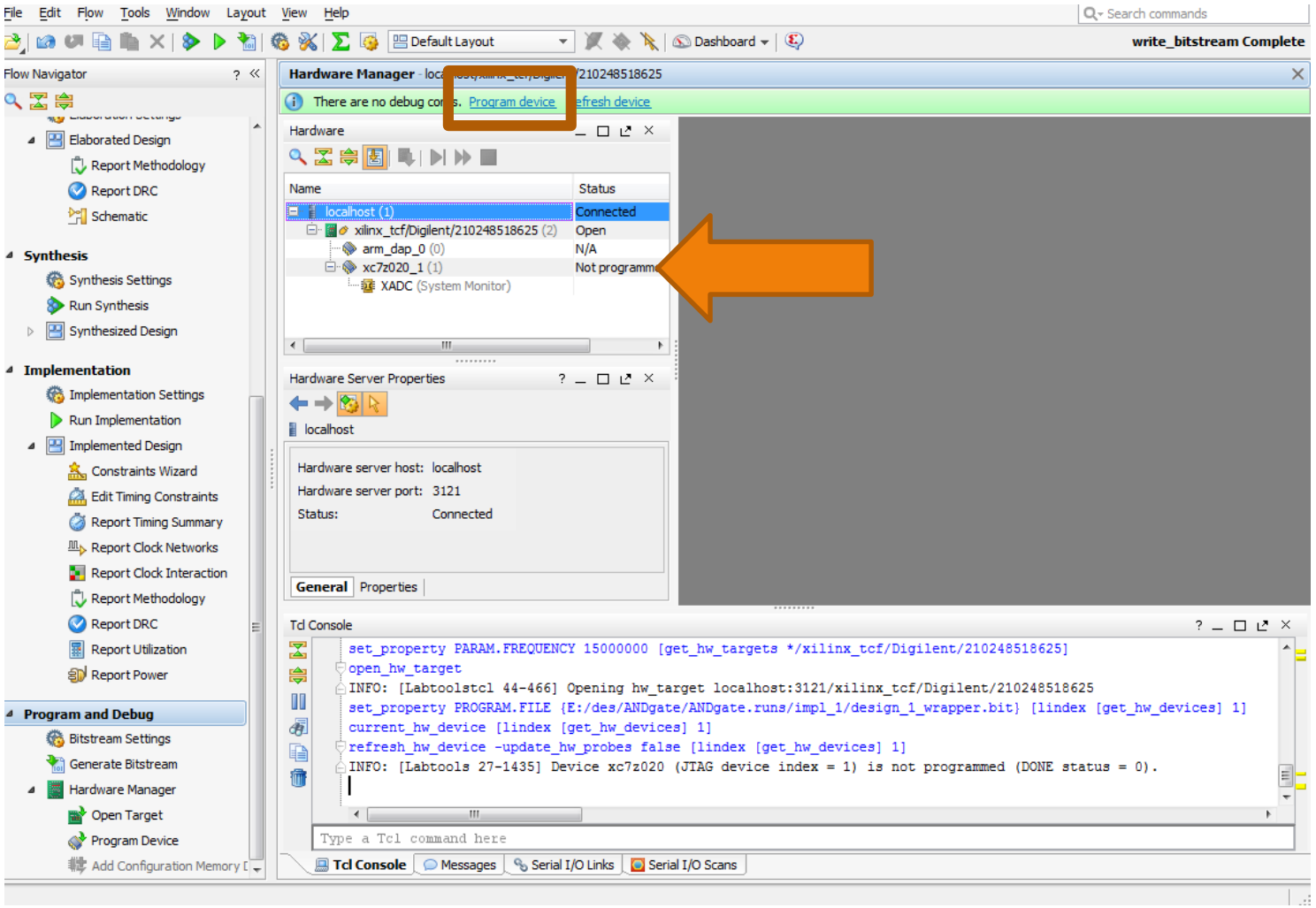
硬件管理器：打开目标设备



计算机组织与结构

硬件管理器：对设备进行编程

计算机组织与结构



课程内容

计算机组织与结构

一、数字系统设计的一些概念和技巧

二、VHDL与Verilog语言

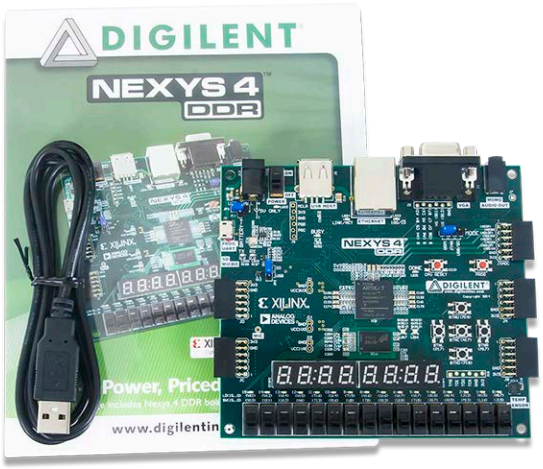
三、硬件设计模块的仿真与测试

四、软件开发环境VIVADO

五、硬件平台简介

硬件资源与参考资源

- 开发板型号：Nexys 4 DDR
  - FPGA型号：Artix-7 / XC7A100T-1CSG324C
  - 用户使用手册和原理图
  - 开发板的Vivado配置文件
  - 约束文件XDC和UCF
- 资源链接



https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/

使用开发板在Vivado中设计流程

- **Vivado中配置开发板**

https://reference.digilentinc.com/reference/software/vivado/board-files?redirect=1

- **编程步骤/LED&Switch**

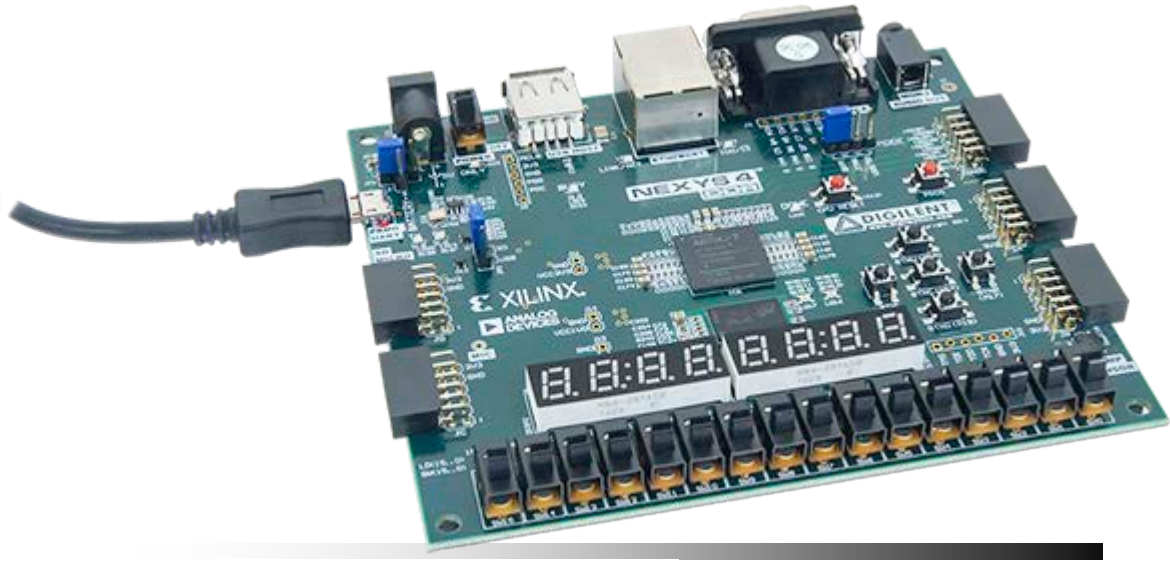
https://reference.digilentinc.com/learn/programmable-logic/tutorials/nexys-4-ddr-programming-guide/start

- **示例程序/GPIO**

https://reference.digilentinc.com/learn/programmable-logic/tutorials/nexys-4-ddr-gpio-demo/start

设计的总体要求

- 善于利用按键、开关、LED灯、数码管的输入、输出与显示功能。
- 所设计的功能展示能够充分直观体现。
- 锁定管脚参考Nexys4 DDR的参考手册。



Q&A?