CPLD （Complex Programmable Logic Device，复杂可编程逻辑器件）
IC （Integrated Circuit， 集成电路）
MCU（microprogramed control unit）微程序控制器
MPU（microprocessor unit）微处理器
HDL (Hardware Description Language，硬件描述语言)
RTL（Register Transfer Level，寄存器转换级电路）
IP (Intellectual Property，知识产权)
RAM（random access memory） 随机存取储存器
ROM（read only memory） 只读存储器
EPROM（Electrically Programmable Read-Only-Memory）可擦可编程只读存储器
API （Application Program Interface），应用程序界面
HAL（Hardware Abstraction Layer），硬件抽象层
UDP(User-Defined Primitives）用户定义原语
EDA 定义
EDA 技术就是以计算机为工作平台，以 EDA 软件工具为开发环境，以 PLD 器件或者 ASIC 专用集成电路为目标器件设计实现电路系统的一种技术。
现代 EDA 技术和 EDA 工具的共同特点:HDL 语言，标准化，开放性，库，综合
逻辑器件，固定逻辑器件（大量的"一次性工程成本" NRE）和 PLD（廉价，快速）
PLD 能实现任意数字逻辑?
任何组合逻辑函数均可化为"与或"表达式，用"与门—或门"二级电路实现，任何时序电路又都可以由组合电路加上存储元件（触发器）构成。因此，从原理上说，与或阵列加上触发器的结构就可以实现任意的数字逻辑。
EDA 设计流程。
设计输入：设计以开发软件的要求表达出来，如文本输入，原理图输入。
综合：将较高层次的设计描述自动转化为低级的设计描述（综合=转换（翻译）+优化）分为：行为综合，逻辑综合，版图综合
布局布线：将综合生成的电路逻辑网表映射到具体的目标器件中实现，并产生最终可下载文件的过程。
时序分析：按时间顺序采集到的数据进行数据分析
仿真：对设计电路的功能验证，分为行为仿真（前），功能仿真（综合前），时序仿真（综合后）
编程下载：指将配置后的编程文件装入到 PLD 器件的过程。
EDA 工具的两个主要功能是：综合和仿真。
综合器和编译器的区别。
① 编译器将软件程序翻译成某种特定的 CPU 机器代码，这种代码不代表硬件结构，更不能改变 CPU 的硬件结构，只能被动的为其特定的硬件电路结构所利用。只是机械式的一一对应的翻译。
② 综合器则不同，综合器转化（翻译）的目标是底层电路结构网表文件，它不依赖于任何特定硬件环境，能轻易的移植到任何通用硬件环境中，具有明显的能动性和创造性，不是机械式的一一对应的翻译，而是根据设计库、工艺库以及预先设置的各类约束条件，选择最优的方式完成电路结构的形成。
IP 核
定义：完成某种功能的设计模块。
分类：硬核、固核和软核。
软核：在寄存器级或门级对电路功能用 HDL 进行描述。
硬核：以版图形式实现的设计模块。
固核：完成了综合的功能块。
Nios II 软核处理器（硬件抽象层 HAL 是软硬件的桥梁）
最大特点：可配置性能
开发任务：定制 Nios II 处理器系统和软件开发
集成开发环境：Nios II IDE

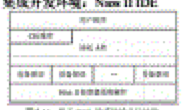
图3.16 基于 HAL 的系统的分层结构

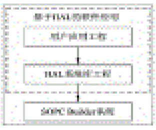

图 3.17 NiosII IDE 工程的结构

SOC
定义：系统芯片（SoC），或者称为芯片系统、片上系统。是指把一个完整的系统集成在一个芯片上。
构成：由微处理器核（MPU Core）、数字信号处理器核（DSP Core）、存储器核（RAM/ROM）、A/D、D/A 模拟及 USB 接口核等构成一个单片系统（SoC）。
SOPC：是可编程逻辑器件技术和 SOC 技术发展与融合的产物，在一个可编程芯片上实现一个电子系统的技术。
优点（PLD+SOC）
（1） 至少包含一个嵌入式处理器内核
（2） 具有小容量片内高速 RAM 资源
（3） 丰富的 IP Core 资源可供选择
（4） 足够的片上可编程逻辑资源
（5） 处理器调试接口和 FPGA 编程接口
（6） 包含部分可编程模拟电路
（7） 单芯片、低功耗、小封装

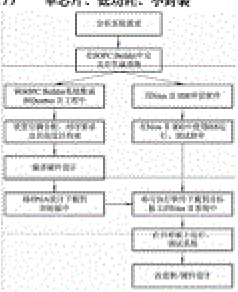
图3.18 SOPC 系统设计流程

设计思路
Bottom-up 设计，即自底向上的设计。
Top-down 设计，即自顶向下的设计。将设计分为系统级、功能级、门级、开关级等不同的层次，按照自上向下的顺序，对各个层次进行设计和仿真。
集成电路发展与芯片集成度同步，数字器件经历了从 SSI、MSI、LSI、VLSI，到 SOC。
数字集成化系统性能的四个特性：速度（时序和延时）、吞吐量、面积、功耗
硬件描述语言 HDL，具有特殊结构能够对硬件逻辑电路的功能进行描述的一种高级编程语言，可以设计硬件电路（芯片）或仿真硬件电路行为。
Verilog HDL
主要功能：描述电路的连接、仿真硬件电路
描述级别：系统级、算法级、寄存器传输级（RTL 级）、逻辑级、门级和开关级

基本结构：模块声明，端口声明，信号类型定义，逻辑功能描述，时序规范
行为描述方式：行为描述，数据流描述，结构化描述，（混合描述）
行为描述语句：条件语句、赋值语句和循环语句
Verilog HDL 不仅提供描述设计的能力，而且提供对激励、控制、存储响应和验证的建模能力
Verilog HDL 既适合于可综合的电路设计，也可胜任电路与系统的仿真
Reg 与 wire 的区别：①reg 是变量类型之一，wire 是线网类型之一 ②reg 变量只能在 always 或 initial 语句中赋值，而 wire 变量只能在连续赋值语句 assign 中赋值，或者通过模块实例的输出（和输入/输出）端口赋值。③进行初始化时，reg 变量的值为 x，wire 线网的值为 z，线网可以赋予强度值，而 reg 变量不能赋予强度值。
Verilog HDL 中有两类数据类型：**线网数据类型和寄存器数据类型**。线网类型表示构件间的**物理连线**，而寄存器类型表示抽象的**数据存储元件**。
赋值语句
b<=a;非阻塞赋值，块结束后完成赋值
b=a;阻塞赋值，赋值语句执行完后，块才结束
移位寄存器

```
module pipeo1 (q3, d, clk);
output [7:0] q3;
      input [7:0] d;
input   clk;
reg [7:0] q3, q2, q1;
  always @(posedge clk) begin
            q1 <= d;
q2 <= q1;
q3 <= q2;
end
endmodule
```

流水线技术
流水线，所谓流水线设计实际上就是把规模较大、层次较多的组合逻辑电路分为几个级，在每一级插入寄存器组暂存中间数据。
流水线加法器好处：工作速度快，在逻辑电路中加入若干寄存器来暂存中间结果，虽然多用了一些寄存器资源，但减小了每一级的电路的时延，提高整个加法器的运行频率。
为了保证数据存吐率，电路设计中的一个主要问题是要维持系统时钟的速度处于或高于 某一频率。如果延时路径较长，就必须在组合逻辑间插入触发器
流水线设计技术使用情形：在某些复杂逻辑功能的完成需要较长延时且会使系统难以运行在高的频率上，这时可使用流水线设计技术
流水线设计技术的好处：在长延时的逻辑功能块中插入触发器，使得复杂的逻辑操作分步完成，减小每个部分的延时，从而使系统的运行频率得以提高
流水线技术的缺点：增加了寄存器逻辑，即增加量芯片资源的耗用

有限状态机定义：由**寄存器逻辑**和**组合逻辑**构成的硬件时序电路，一般包括组合逻辑和寄存器逻辑两个部分。
寄存器逻辑的功能是存储有限状态机的内部状态（寄存器组的 0 和 1 构成有限状态）。
组合逻辑分为次态逻辑和输出逻辑两部分，
次态逻辑：确定有限状态机的下一个状态
输出逻辑：确定有限状态机的输出
常用的状态编码有**一位热码，格雷编码，约翰逊编码，顺序编码**四种方式
状态机可分为两类：**米里型（Mealy）和摩尔型**
区别：摩尔型状态机的输入发生变化时需要等待时钟的到来。
　　米里型的输出是在输入变化后立即变化，**多了输入到输出的逻辑路径**
//摩尔型状态机的输出信号**仅与当前状态有关**，即可以把摩尔型有限状态机的输出看成当前状态的函数；
米里型状态机的输出信号**不仅与当前状态有关，而且还与输入信号有关**，即可以把米里型有限状态机的输出看成是当前状态和所有输入信号的函数。
CPU 通过操作指令和硬件操作单元来控制功能的实现。
有限状态机通过**状态转移**来实现，**适用于** PLD。通过恰当的 Verilog **语言描述**和 EDA **工具综合**，可以生产性能优越的有限状态机。
有限状态机（Finite State Machine, FSM）是时序电路设计中经常采用的一种方式，尤其适于设计数字系统的**控制模块**。
优点：具有速度快、结构简单、可靠性高等优点，过程明确，适用于控制。
一般结构：（参照程序来理解）
1、说明部分:状态转换变量的定义和所有可能状态的说明
2、主控时序过程：状态机的运转和状态转换的过程
3、主控组合过程：根据当前状态和外部的信号发出控制信号，同时确定下一状态的走向
4、辅助过程：配合状态机工作的组合过程和时序过程
状态机有三种表示方法：状态图（state diagram）、状态表（state table）、流程图
状态机设计中主要包含三个对象：
(1)当前状态 ，或称为现态（current state，cs）
(2)下一个状态，或称为次态（Next State，ns）
(3)输出逻辑（out logic，ol）
相应的，在用verilog描述有限状态机时，有下面几种**描述方式**：
（1） 用三个过程描述：即现态（cs）、次态（ns）、输出逻辑（ol）各用一个always过程描述
（2） 双过程描述（CS+NS、OL双过程描述）：使用两个always过程来描述有限状态机，一个过程描述现态和次态时序逻辑（CS+NS）；另一个过程描述输出逻辑（OL）
（3） 双过程描述（CS、NS+OL双过程描述）：一个过程用来描述现态（CS）；另一个过程描述次态和输出逻辑（NS+OL）。
（4） 单过程描述 在单过程描述方式中，将状态机现态、次态、和输出逻辑（CS+NS+OL）放在一个always过程中进行描述。

```
module fsm1_seq101(clk,clr,x,z);
input clk,clr,x; output reg z; reg[1:0] state;
parameter S0=2'b00,S1=2'b01,S2=2'b11,S3=2'b10;
always @(posedge clk or posedge clr)
Begin        if(clr) state<=S0;
             else case(state)
S0:begin if(x) begin state<=S1; z=1'b0;end
else begin state<=S0; z=1'b0;end end
S1:begin if(x) begin state<=S1; z=1'b0;end
else begin state<=S2; z=1'b0;end end
S2:begin if(x) begin state<=S3; z=1'b0;end
else begin state<=S0; z=1'b0;end end
S3:begin    if(x) begin state<=S1; z=1'b1;end
else begin state<=S2; z=1'b1;end end
default:begin state<=S0; z=1'b0;end
endcase    end endmodule
```

```
module fsm2_seq101(clk,clr,x,z);
input clk,clr,x;
output reg z;
output reg[1:0] state,next_state;
parameter S0=2'b00,S1=2'b01,S2=2'b11,S3=2'b10;
always @(posedge clk or posedge clr)
begin        if(clr) state<=S0;
else state<=next_state;
end
always @(state or x)
begin
case (state)
S0:begin if(x) begin state<=S1; z=1'b0;end
else begin state<=S0; z=1'b0;end end
S1:begin if(x) begin state<=S1; z=1'b0;end
else begin state<=S2; z=1'b0;end end
S2:begin if(x) begin state<=S3; z=1'b0;end
else begin state<=S0; z=1'b0;end end
S3:begin    if(x) begin state<=S1; z=1'b1;end
else begin state<=S2; z=1'b1;end end
default:begin state<=S0; z=1'b0;end
endcase    end endmodule
```

```
module fsm3_seq101(clk,clr,x,z);
input clk,clr,x;
output reg z,next_state;
output reg[1:0] state;
parameter S0=2'b00,S1=2'b01,S2=2'b11,S3=2'b10;
always @(posedge clk or posedge clr)
begin        if(clr) state<=S0;
             else state<=next_state;
end
always @(state or x)
begin
case (state)
S0:begin if(x) next_state<=S1;
else        next_state<=S0; end
S1:begin if(x)           next_state<=S1;
else        next_state<=S2; end
S2:begin if(x) next_state<=S3;
else next_state<=S0; end
S3:begin if(x)           next_state<=S1;
else        next_state<=S2; end
default:    next_state<=S0;
Endcase    end
always @(state)
begin    case(state)
S3: z=1'b1;
default:z=1'b0;
endcase end endmodule
```

（1） 2 级

```
module adder_pipe2(cout,sum,ina,inb,cin,clk);
input[7:0] ina,inb; input cin,clk;
output reg[7:0] sum;
output reg cout;
reg[3:0] tempa,tempb,firsts;
reg firstc;
always @(posedge clk)
begin
  {firstc,firsts}=ina[3:0]+inb[3:0]+cin;
tempa=ina[7:4]; tempb=inb[7:4];
end
always @(posedge clk)
begin
{cout,sum[7:4]}=tempa+tempb+firstc;
sum[3:0]=firsts;
end
endmodule
```

（2） 4 级

```
module adder_pipe4(cout,sum,ina,inb,cin,clk);
output[7:0] sum;
output cout;
input[7:0] ina,inb;
input cin,clk;
reg tempc,firstc,secondc,thirdc, cout;
reg[1:0] firsts, thirda,thirdb;
reg[3:0] seconda, secondb, seconds;
reg[5:0] firsta, firstb, thirds;
reg[7:0] tempa,tempb,sum;
always @(posedge clk)
begin
  tempa=ina; tempb=inb; tempc=cin;
  end
always @(posedge clk)
begin
  {firstc,firsts}=tempa[1:0]+tempb[1:0]+tempc;
firsta=tempa[7:2]; firstb=tempb[7:2];
end
always @(posedge clk)
begin
{secondc,seconds}={firsta[1:0]+firstb[1:0]+firstc,firsts};
seconda=firsta[5:2]; secondb=firstb[5:2];
end
always @(posedge clk)
begin
{thirdc,thirds}={seconda[1:0]+secondb[1:0]+secondc,seconds
};
thirda=seconda[3:2];thirdb=secondb[3:2];
end
always @(posedge clk)
begin
{cout,sum}={thirda[1:0]+thirdb[1:0]+thirdco,thirds};
end
endmodule
```