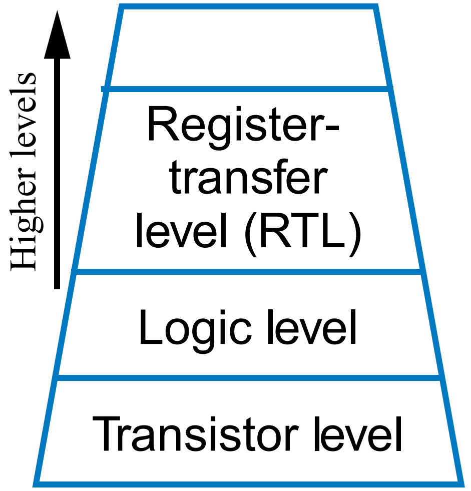
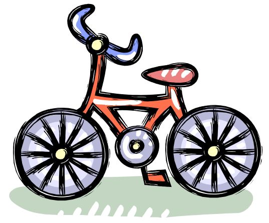
寄存器传输级（RTL）设计方法

寄存器传输级设计方法

- 用**寄存器**保存中间数据
  - 寄存器用于通用存储，就像算法中的变量
- 用专用的**数据通路（data path）**实现运算
  - 数据处理电路
  - 连接网络
  - 寄存器
- 用专用的**控制通路（control path）**规定运算的顺序
  - 状态机
  - 在什么时间完成什么运算



Levels of digital design abstraction



Appropriate building blocks:
Tires, seat, pedals
Not:
Rubber, glue, metal

寄存器传输级设计方法

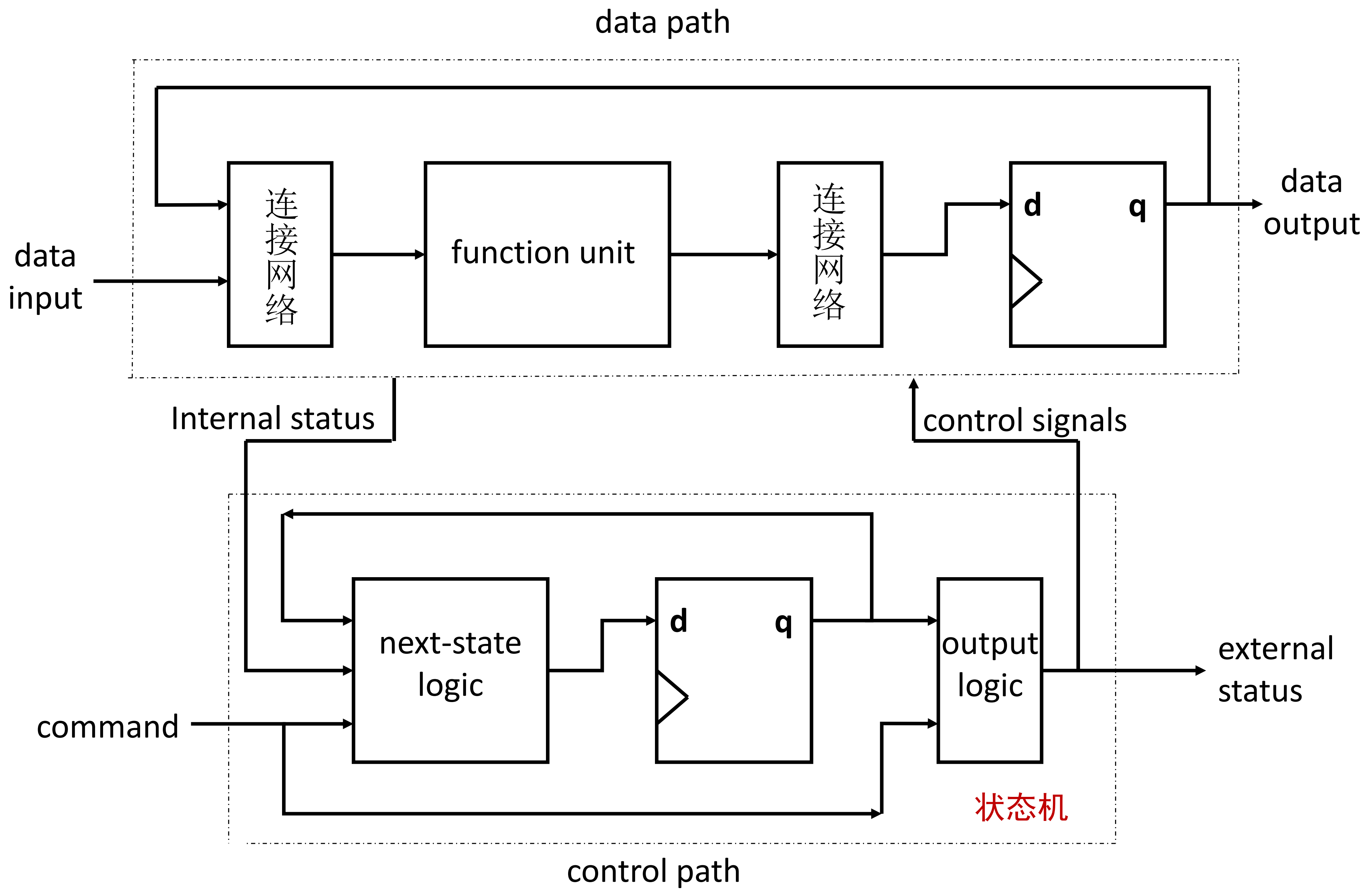
Step1：算法和高层次状态机

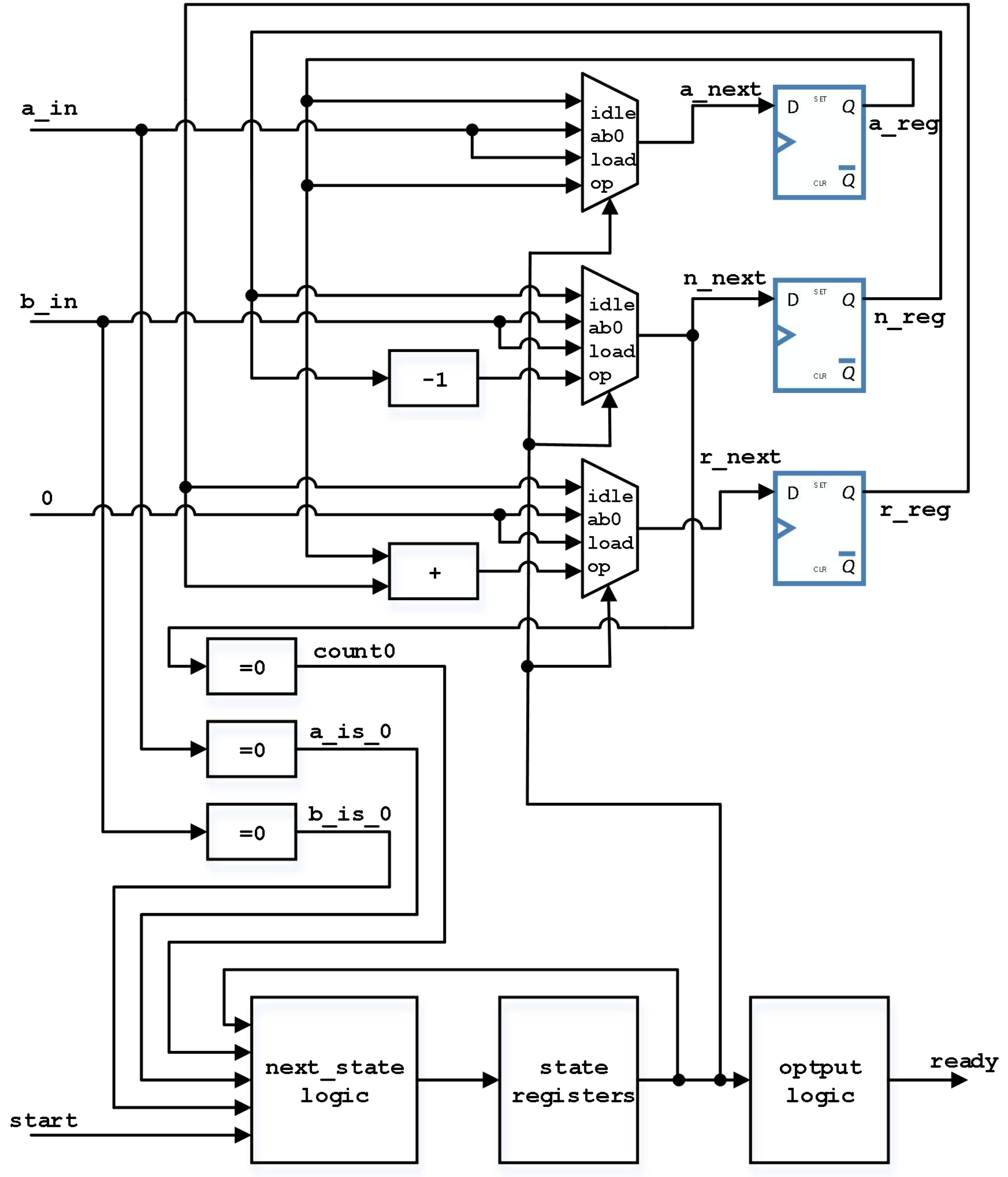
Step2：产生数据通路

Step3：连接数据通路和控制器

Step4：产生控制器状态机

Step3:连接数据通路和控制器




a_in
b_in
0
idle
ab0
load
op
a_next
n_next
r_next
D
Q
a_reg
n_reg
r_reg
-1
+
=0
count0
a_is_0
b_is_0
start
next_state logic
state registers
optput logic
ready

寄存器传输级设计方法

Step1：算法和高层次状态机

Step2：产生数据通路

Step3：连接数据通路和控制器

Step4：产生控制器状态机

控制通路

- 控制通路包括以下输入输出

  外部命令输入

  内部状态输入

  控制信号输出

  外部状态输出

- 时钟信号

  数据通路是规则的时序电路

  状态机是随机的时序电路

  数据通路和控制通路通常用一个时钟信号同步

例：乘法器

输入：决定框中的信号

start：外部命令

a_is_0，b_is_0

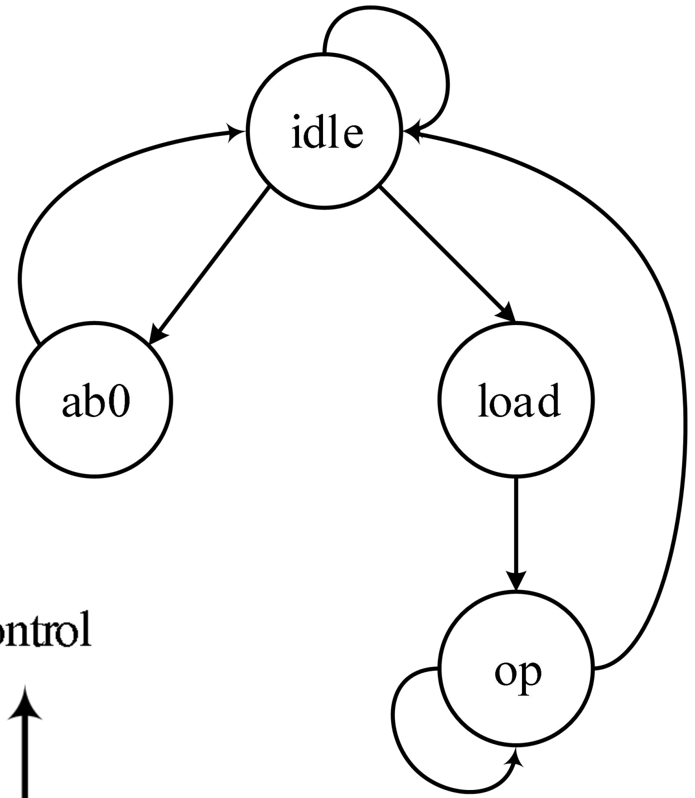
count_0：内部状态

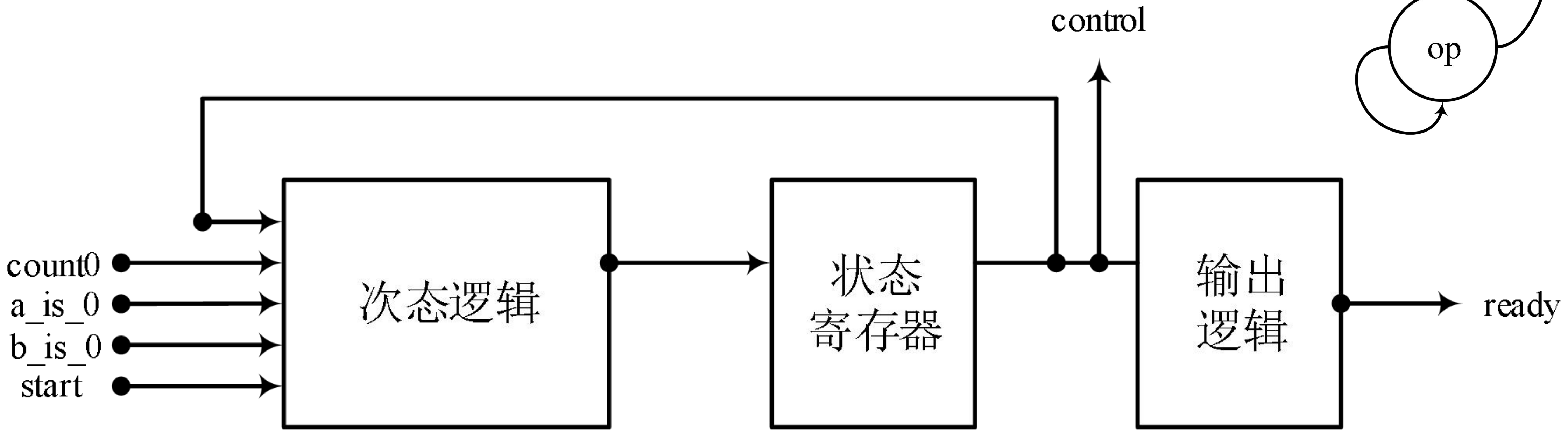
输出：

外部状态信号ready

输出控制信号control

控制通路—状态机设计


idle
ab0
load
op


control
count0
a_is_0
b_is_0
start
次态逻辑
状态寄存器
输出逻辑
ready

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity sequ_mult is
port(
        clk, reset : in std_logic;
        start : in std_logic;
        a_in, b_in : in std_logic_vector(7 downto 0);
        ready : out std_logic;
        p : out std_logic_vector(15 downto 0));
end sequ_mult;
architecture rtl of sequ_mult is
        constant WIDTH : integer := 8;
        type state_type is (idle, ab0, load, op);
        signal state_reg, state_next : state_type;
        signal a_is_0, b_is_0, count_0 : unsigned(WIDTH-1 downto 0);
        signal n_reg, n_next : unsigned(WIDTH-1 downto 0);
        signal a_reg, a_next : unsigned(WIDTH-1 downto 0);
        signal r_reg, r_next : unsigned(2*WIDTH-1 downto 0);
        signal adder_out : unsigned(2*WIDTH-1 downto 0);
        signal sub_out : unsigned(WIDTH-1 downto 0);
begin
```

```
-----------control path : state register----------------
           process(clk, reset)
           begin
                      if reset = '1' then
                                 state_reg <= idle;
                      elsif clk’event and clk = ‘1’ then
                                 state_reg <= state_next;
                      end if;
           end process;
-----------control path : output logic ----------
           ready <= '1' when state_reg = idle else '0';
```

```
-----------control path : next logic/output logic ----------
process(state_reg, start, a_is_0, b_is_0, start, count_0)
begin
        case state_reg is
                when idle =>
                        if start = '1' then
                                if a_is_0 = '1' or b_is_0 = '1' then
                                        state_next <= ab0;
                                else
                                        state_next <= load;
                                end if;
                        else
                                state_next <= idle;
                        end if;
                when ab0 =>     state_next <= idle;
                when load =>    state_next <= op;
                when op =>
                        if count_0 = '1' then
                                state_next <= idle;
                        else
                                state_next <= op;
                        end if;
        end case;
end process;
```

```
--------------data path -----------------
             process(clk, reset)
             begin
                          if reset = '1' then
                                       a_reg <= (others => '0');
                                       n_reg <= (others => '0');
                                       r_reg <= (others => '0');
                          elsif clk'event and clk = '1' then
                                       a_reg <= a_next;
                                       n_reg <= n_next;
                                       r_reg <= r_next;
                          end if;
             end process;
-----------data path: function unit----------
             adder_out <= ("00000000" & a_reg) + r_reg;
             sub_out <= n_reg - 1;
```

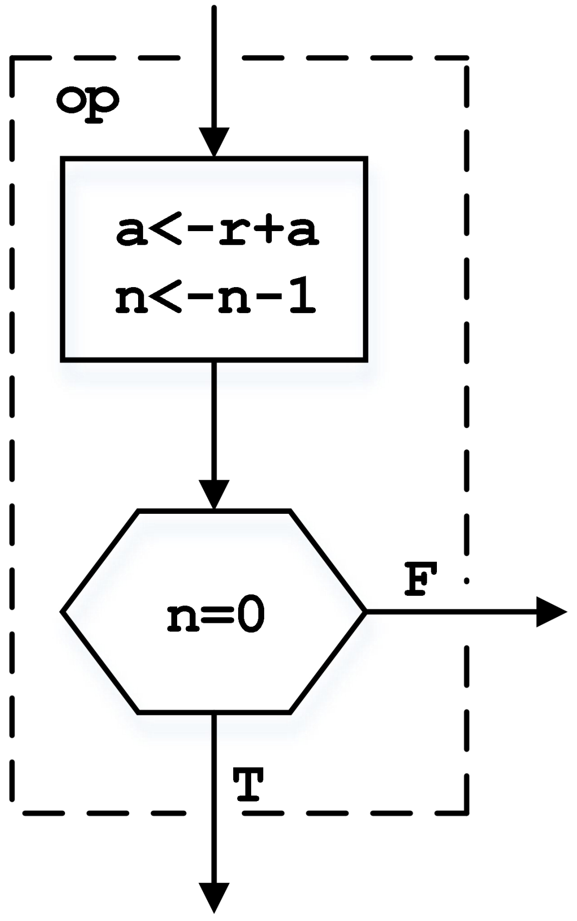
```
-----------data path: routing multiplexer----------
        process(state_reg, a_reg, n_reg, r_reg, a_in, b_in, adder_out, sub_out)
        begin
                case state reg is
                        when idle =>
                                a_next <= a_reg;
                                n_next <= n_reg;
                                r_next <= r_reg;
                        when ab0 =>
                                a_next <= a_in;
                                n_next <= b_ing;
                                r_next <= (others => '0');
                        when load =>
                                a_next <= a_reg;
                                n_next <= n_reg;
                                r_next <= r_reg;
                        when op =>
                                a_next <= a_reg;
                                n_next <= sub_out;
                                r_next <= adder_out;
                end case;
        end process;
```

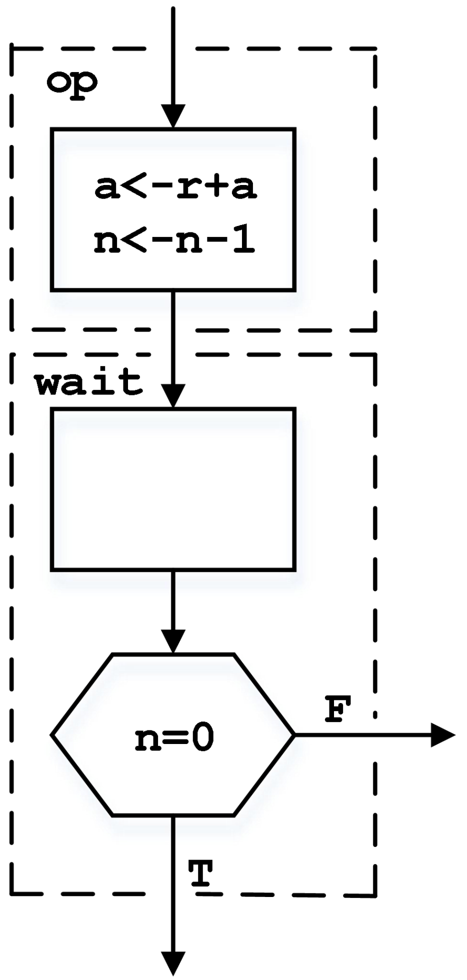
```
-----------data path: status----------
          a_is_0 <= '1' when a_in = "00000000" else '0';
          b_is_0 <= '1' when b_in = "00000000" else '0';
          count_0 <= '1' when n_next = "00000000" else '0';

          p <= r_reg;
end rtl;
```

```
if  (n = 0) then{
            go to stop;}
else{
            go to op;}}
```

- 软件算法：顺序的
  n用计数器跟踪迭代次数
  n在n=n-1语句更新

- ASMD图中
  n <- n – 1和决定框是在同一状态
  n在这个状态之后更新
  判断时用的是旧的n值
  条件n = 0：会多一次迭代，导致结果不正确

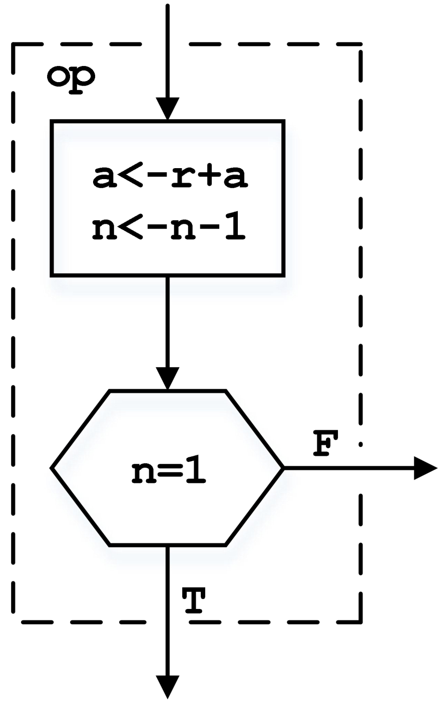


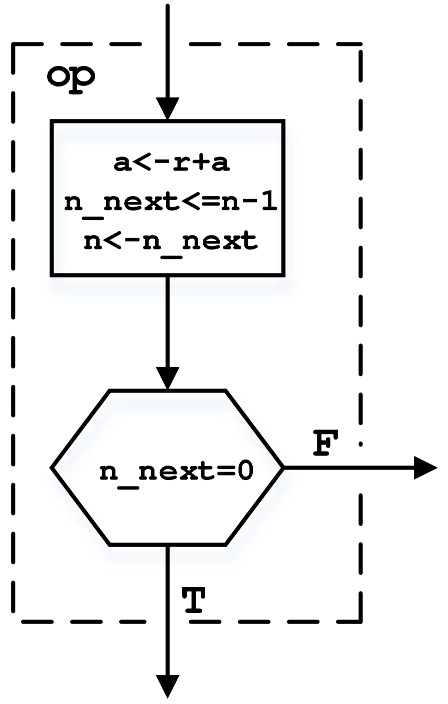


- 方法一：
  用条件n = 1 结束循环

- 方法二：
  人为插入一个等待状态wait
  条件判断时n已经更新
  每次迭代引入一个时钟周期
  性能下降

- 方法三：
  用n的下一次值判断





寄存器传输设计方法的关键如何得到是算法高效的ASMD图

在决定框中的布尔表达式中使用寄存器

上面的例子中：状态信号

a_is_0

b_is_0

count_0

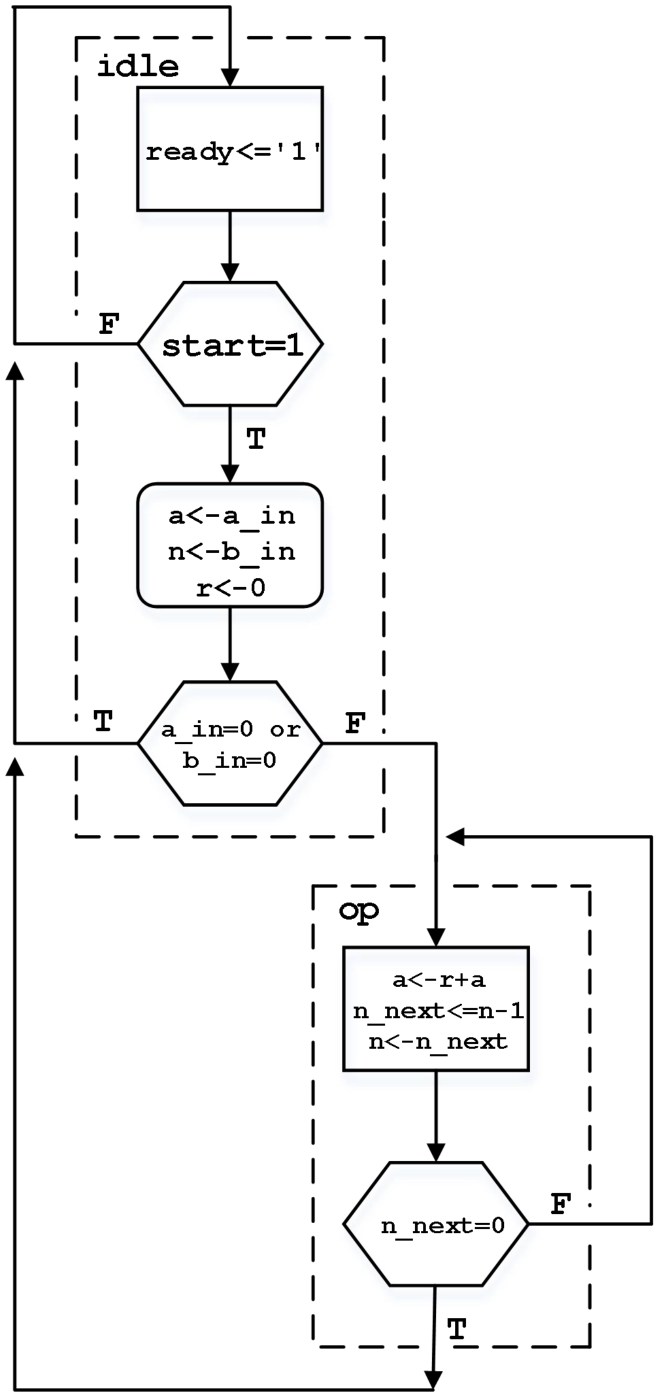
另一种方法：

用寄存器和输入信号直接表示布尔条件

a_in，b_in

乘法器的改进

- 前面的设计
  - 在idle状态，在决定框中用到a_in b_in
  - 在ab0或load状态把a_in、b_in 加载到寄存器
  - 实际加载是在两个时钟周期之后
- 优化
  - 对start、a_in、b_in同时采样



```
architecture rtl2 of sequ_mult is
        constant WIDTH : integer := 8;
        type state_type is (idle, op);
        signal state_reg, state_next : state_type;
        signal n_reg, n_next : unsigned(WIDTH-1 downto 0);
        signal a_reg, a_next : unsigned(WIDTH-1 downto 0);
        signal r_reg, r_next : unsigned(2*WIDTH-1 downto 0);
begin
---------state and data register----------------
        process(clk,  reset)
        begin
                if reset = '1' then
                        state_reg <= idle;
                        a_reg <= (others => '0');
                        n_reg <= (others => '0');
                        r_reg <= (others => '0');
                elsif clk'event and clk = '1' then
                        state_reg <= state_next;
                        a_reg <= a_next;
                        n_reg <= n_next;
                        r_reg <= r_next;
                end if;
        end process;
```

```
----------combinational logic--------------------
process(start, state_reg, a_reg, n_reg, r_reg, a_in, b_in)
begin
        a_next <= a_reg;        n_next <= n_reg;
        r_next <= r_reg;        ready <= '0';
        case state_reg is
                when idle =>
                        if start = '1' then
                                a_next <= a_in;         n_next <= b_in;
                                r_next <= (others => '0');
                                if a_in = "00000000" or b_in = "00000000" then
                                        state_next <= idle;
                                else
                                        state_next <= op;
                                end if;
                        end if;
                        ready <= '1';
                when op =>
                        n_next <= n_reg - 1;
                        r_next <= r_reg + ("00000000" & a_reg);
                        if n_next = "00000000" then
                                state_next <= idle;
                        else
                                state_next <= op;
                        end if;
        end case;
end process;
r <= r_reg;
end rtl2;
```