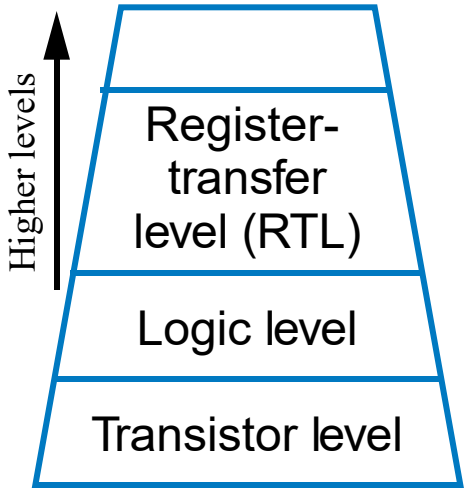
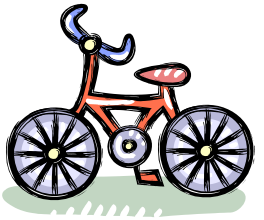
寄存器传输级（RTL）设计方法

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

寄存器传输级设计方法

	Step	Description
Step 1	*Capture a high-level state machine*	Describe the system's desired behavior as a high-level state machine. The state machine consists of states and transitions. The state machine is "high-level" because the transition conditions and the state actions are more than just Boolean operations on bit inputs and outputs.
Step 2	*Create a datapath*	Create a datapath to carry out the data operations of the high-level state machine.
Step 3	*Connect the datapath to a controller*	Connect the datapath to a controller block. Connect external Boolean inputs and outputs to the controller block.
Step 4	*Derive the controller's FSM*	Convert the high-level state machine to a finite-state machine (FSM) for the controller, by replacing data operations with setting and reading of control signals to and from the datapath.

寄存器传输设计方法特点

- 用寄存器保存中间数据，模仿算法中的变量
  - 寄存器用作通用的存储器来保存中间数据
- 用专用的数据通路来实现所有要求的寄存器操作
  - 数据的处理、数据的联通和数据的存储的电路
- 用专用的控制通路来规定寄存器操作的顺序
  - 算法通常是描述一个动作序列
  - 对应的需要一个电路来控制什么时间进行什么寄存器操作

算法 → 规定数据如何处理和如何在寄存器之间传输的一个动作序列

基本的RT运算（或RT操作）

$$r_{dest} = f(r_{src1}, r_{scr2}, \cdots, r_{scrn})$$

r_{dest} 目的寄存器

$r_{src1}, r_{scr2}, \cdots, r_{scrn}$ 源寄存器

f 所做的运算，可以是任何运算，可用组合逻辑实现

常见的RT运算：

$r \leftarrow 1$ 常数1存入寄存器r

$r \leftarrow r$ 寄存器r中的内容再存回寄存器r，即寄存器内容保存不变

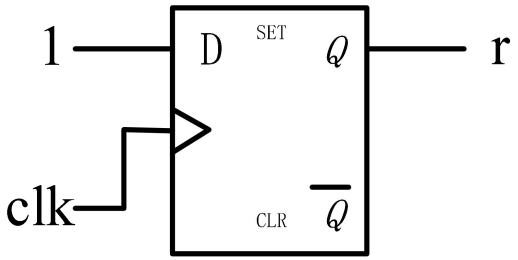
$r \leftarrow r << 3$ 寄存器r中的内容左移3位再存回寄存器r

$r0 \leftarrow r1$ 寄存器r1中的内容存入寄存器r0

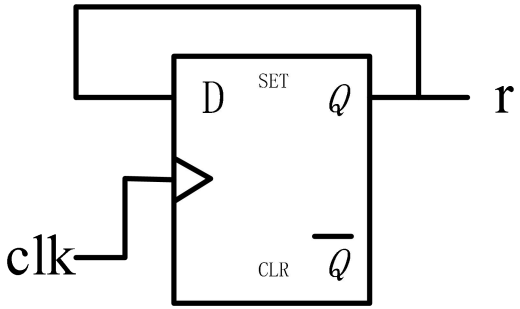
$n \leftarrow n-1$ 寄存器n中的内容减1再存回寄存器n

$y \leftarrow a \oplus b \oplus c$ 寄存器a、b、c、d中的内容异或再存回寄存器y

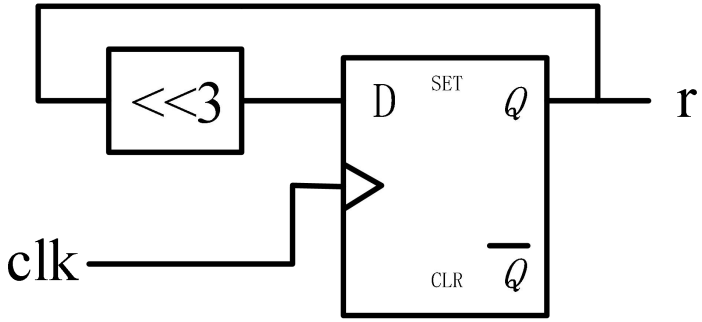
寄存器和算法中变量的不同在于其中有系统时钟的作用



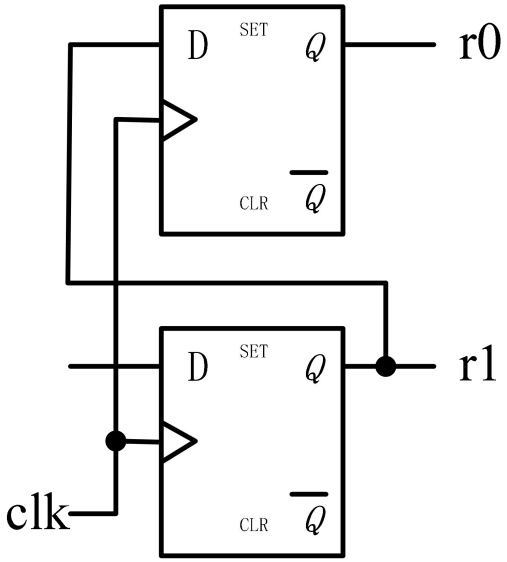
$$r \leftarrow 1$$



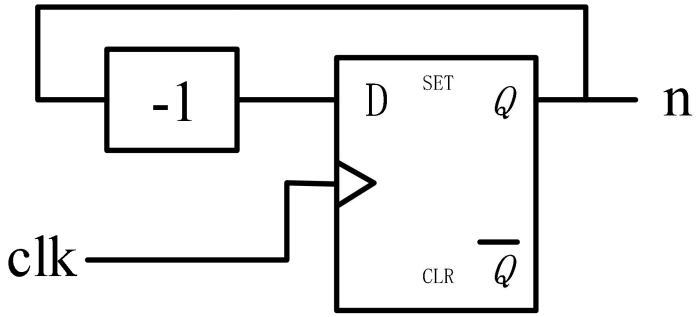
$$r \leftarrow r$$



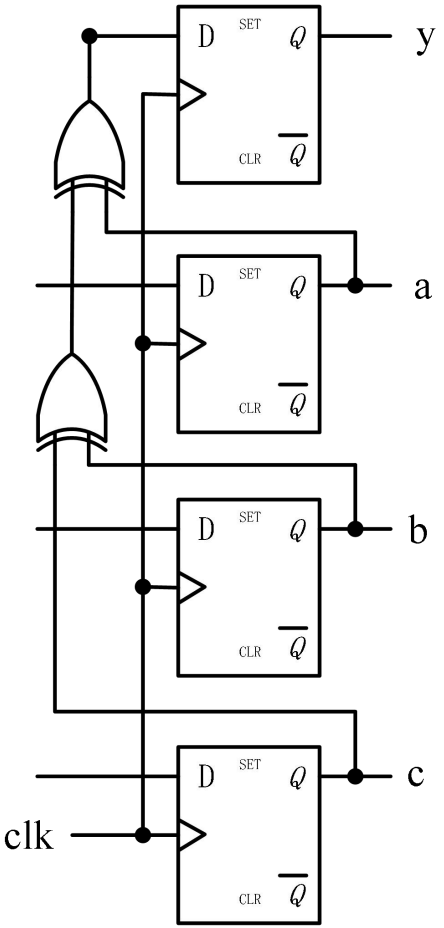
$$r \leftarrow r << 3$$



$$r0 \leftarrow r1$$



$$n \leftarrow n-1$$



$$y \leftarrow a \oplus b \oplus c$$

RT运算的详细过程：

1. 在时钟上升沿，源寄存器在经过时钟到q的延时后得到新的源数据

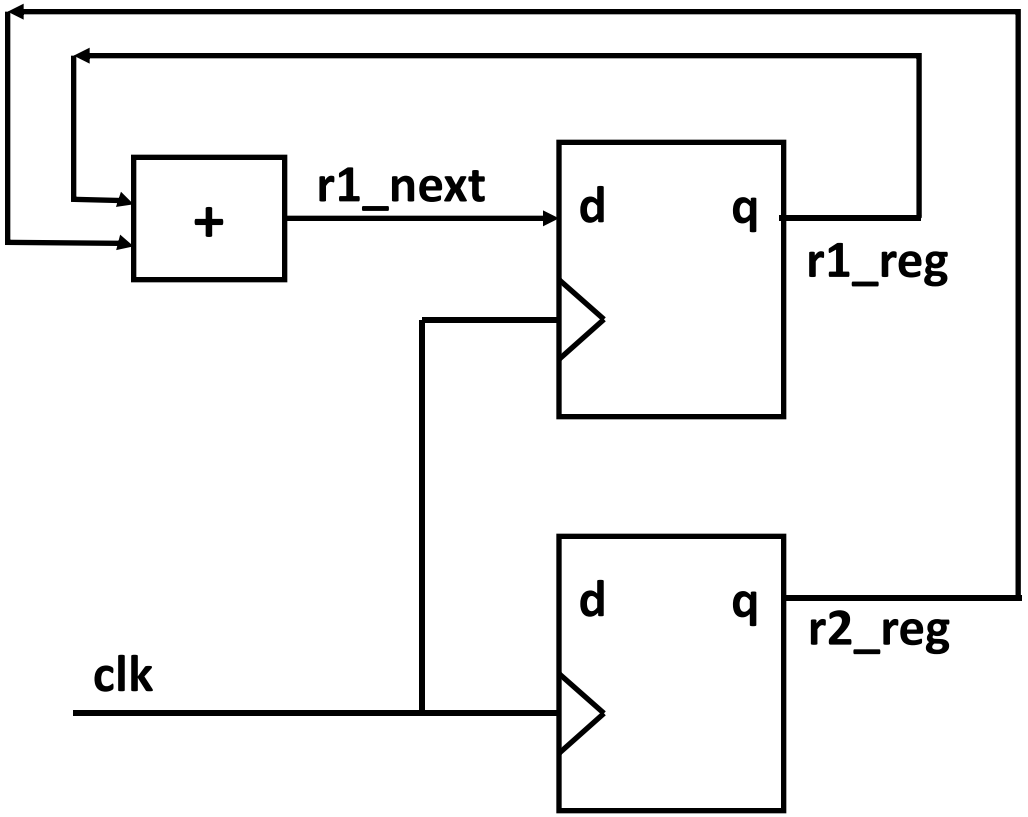
2. 组合逻辑实现所需要完成的运算，假定时钟周期足够长，可以容纳组合逻辑的延时和寄存器的建立时间

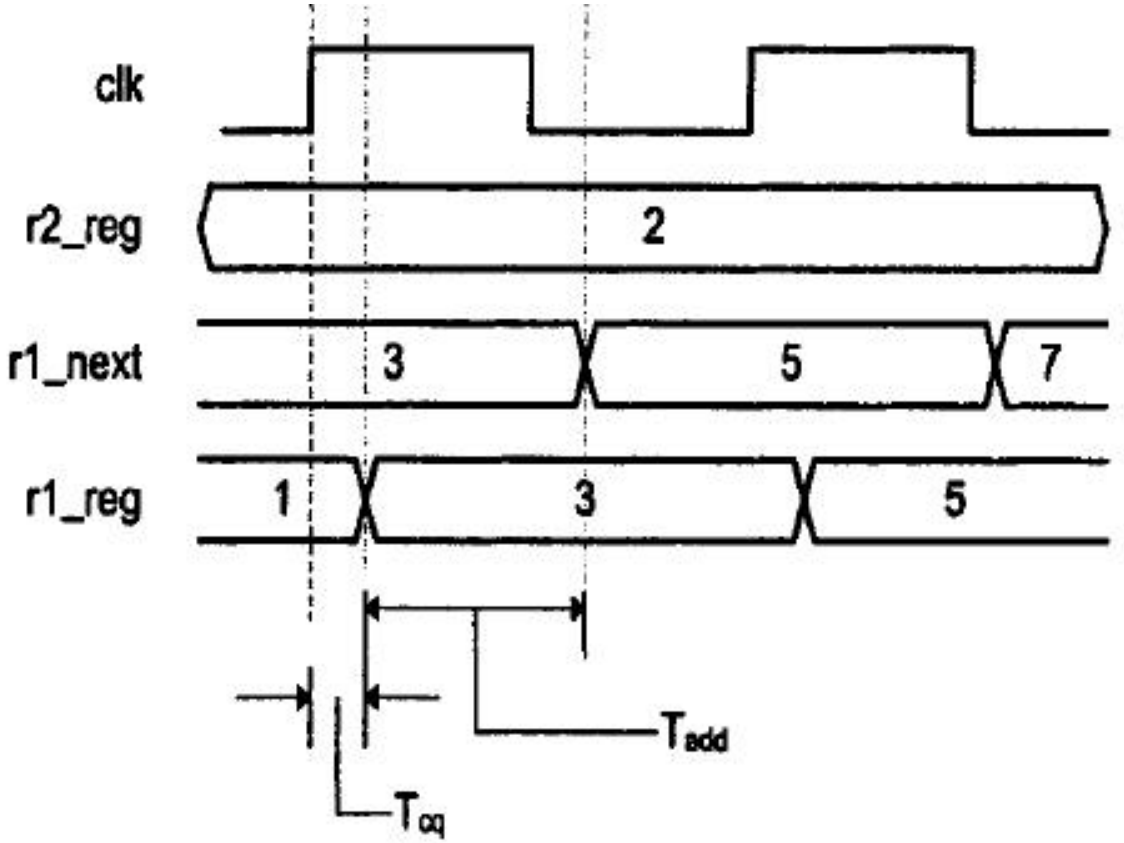
3. 在下一个时钟上升沿，对运算结果采样，写入目的寄存器

$$r_1 \leftarrow r_1 + r_2$$

```
r1_next <= r1_reg + r2_reg;

r1_reg <= r1_next 在下一个时钟上升沿
```





$$r_1 \leftarrow r_1 + r_2$$

r1_next <= r1_reg + r2_reg;

r1_reg <= r1_next 在下一个时钟上升沿

RT运算和数据通路：

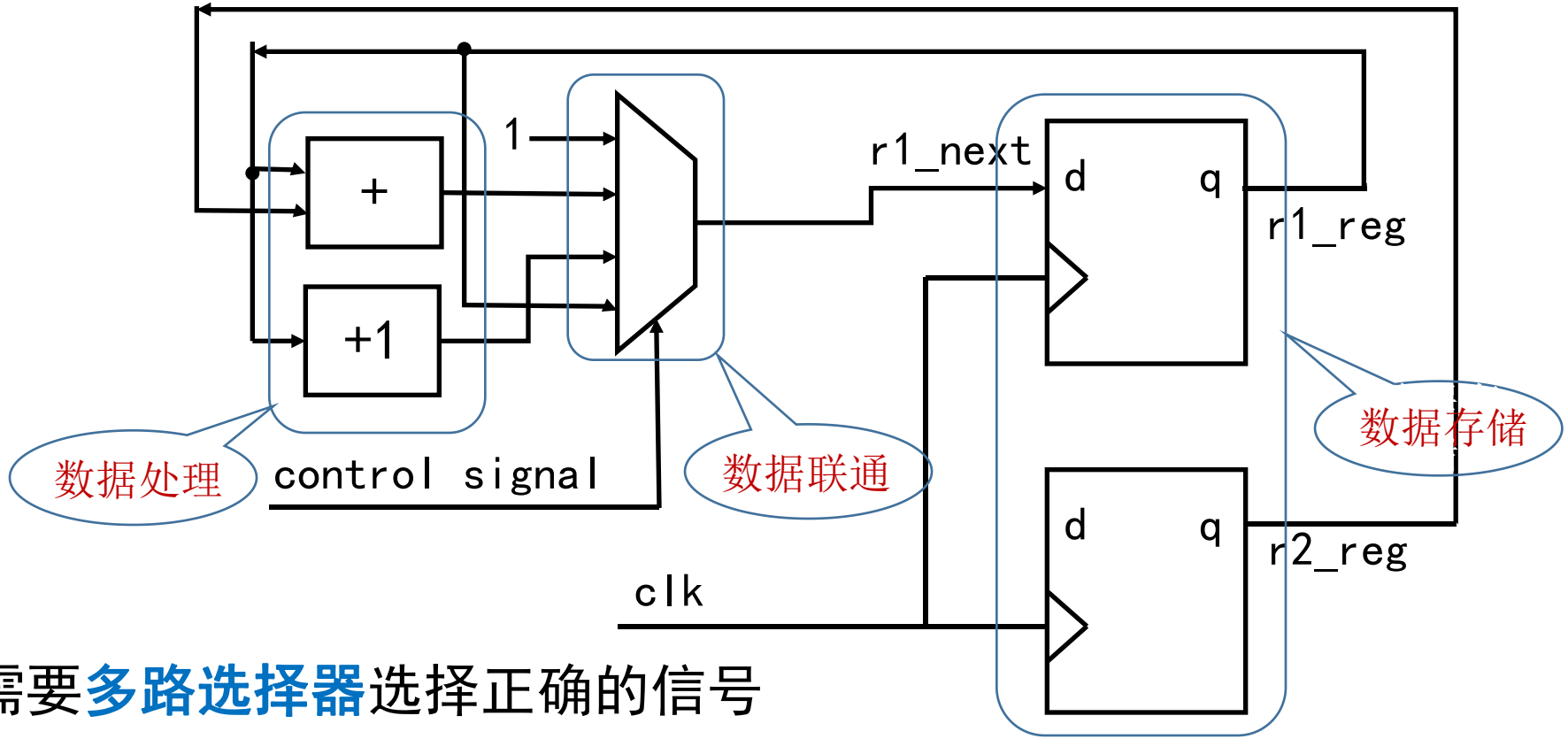
•算法通常有多个步骤，不同的步骤中写入**目的寄存器**的结果不同

$$r_1 \leftarrow 1$$

$$r_1 \leftarrow r_1 + r_2$$

$$r_1 \leftarrow r_1 + 1$$

$$r_1 \leftarrow r_1$$



- 有多种可能性，因此需要**多路选择器**选择正确的信号
- 设计中可能有多个寄存器，对每个寄存器重复这一过程，得到的结构就是基本的、未经优化的数据通路

数据通路：可以实现算法中要求的所有的RT运算（操作）

需要规定：
在什么时间执行哪个RT运算（操作）

控制通路

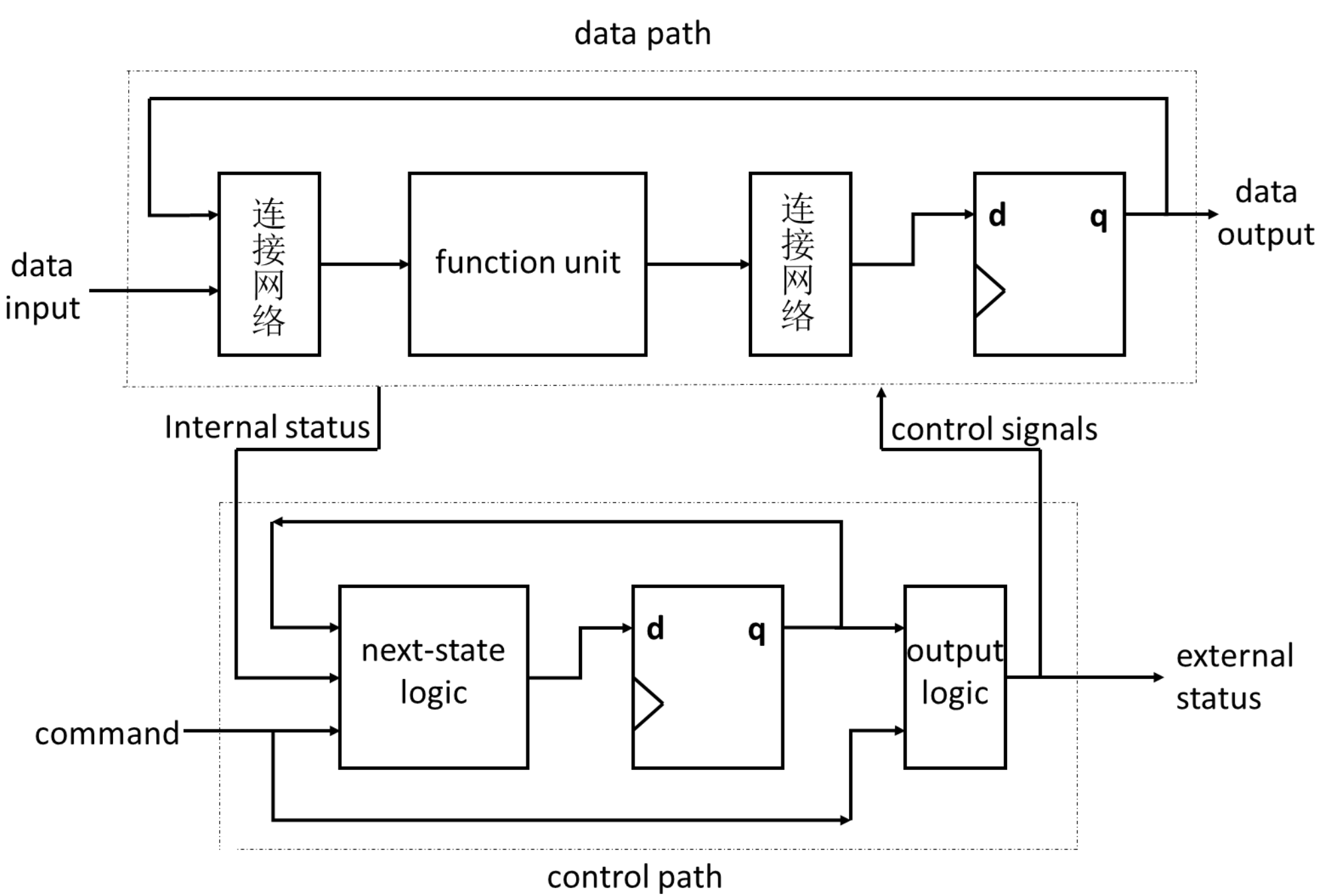
可以完美实现

状态机

可以强制规定按一定顺序执行一系列动作

按时钟转换状态

根据输入条件，可以转入不同的分支，从而可以改变执行动作的顺序，可以用来实现算法中的分支结构，如IF和LOOP等


data path
连接网络
function unit
连接网络
d
q
data input
data output
Internal status
control signals
next-state logic
d
q
output logic
command
external status
control path

Step1:算法和高层次状态机（算法->算法状态图（ASMD）->硬件实现）

- 例：实现一个乘法器

要求：

用一个加法器实现

如：7*5可以用7+7+7+7+7实现

不是一个高效的方法

如何把算法转换为硬件

从C、其他软件语言或伪代码开始

```
if (a_in = 0 or b_in = 0) then{
        r = 0;}
else{
        a = a_in;
        n = b_in;
        r = 0;
        while (n != 0){
                r = r + a;
                n = n - 1;}}
r_out = r;
```

Step1：算法和高层次状态机（算法→算法状态图ASM）

```
if (a_in = 0 or b_in = 0) then{
        r = 0;}
else{
        a = a_in;
        n = b_in;
        r = 0;
        while (n != 0){
                r = r + a;
                n = n - 1;}}
r_out = r;
```

```
        if (a_in = 0 or b_in = 0) then{
                r = 0;}
        else{
                a = a_in;
                n = b_in;
                r = 0;
op:             r = r + a;
                n = n – 1;
                if  (n = 0) then{
                        go to stop;}
                else{
                        go to op;}}
stop:   r_out = r;
```

ASM图中没有循环结构

用硬件实现算法，首先应定义输入输出

输入信号：

- a_in, b_in: 输入操作数，8-bit std_logic_vector
- start：启动命令，有效时乘法器开始工作
- clk: 系统时钟
- reset: 异步复位信号，用于系统初始化

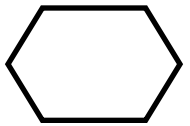
输出信号：

- r_out: 乘积输出，16-bit std_logic_vector
- ready：给外部的状态信号，电路为idle并且准备好接收新数据时有效，也可以看作前一次乘法运算已经完成，运算结果已准备好

算法状态图（ASM图）

- 算法状态图是另一种表示状态机的方法
  - •和状态图包含的信息相同
  - •表述会更清楚
  - •易于构成FSMD
  - •易于转换 为VHDL代码

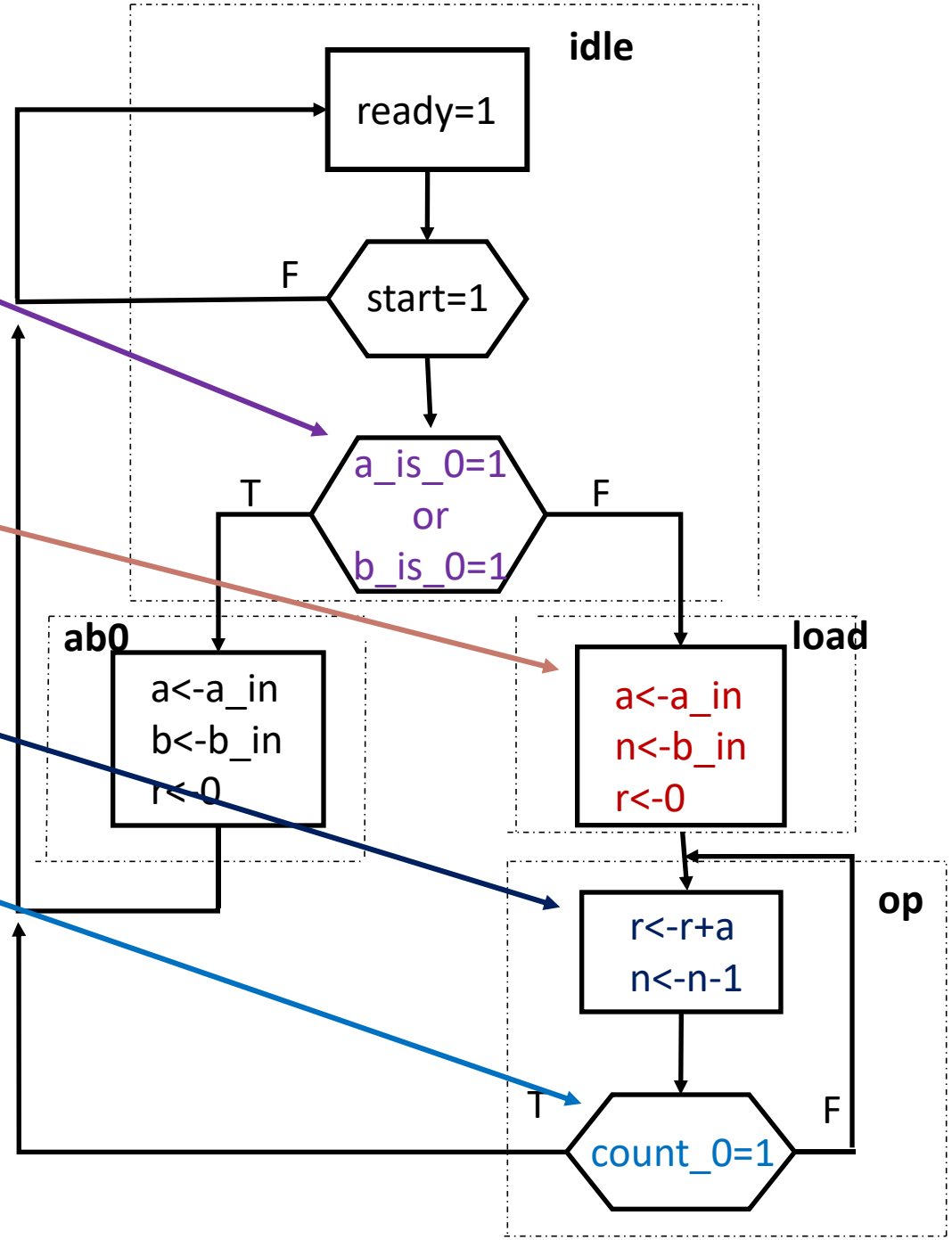
- ASM图
  - •状态框：表示FSM中的一个状态
  - •决定框：决定状态的路径



算法→

算法状态图ASM

```
        if (a_in = 0 or b_in = 0) then{
                r = 0;}
        else{
                a = a_in;
                n = b_in;
                r = 0;
op:             r = r + a;
                n = n – 1;
                if  (n = 0) then{
                        go to stop;}
                else{
                        go to op;}}
stop:   r_out = r;
```



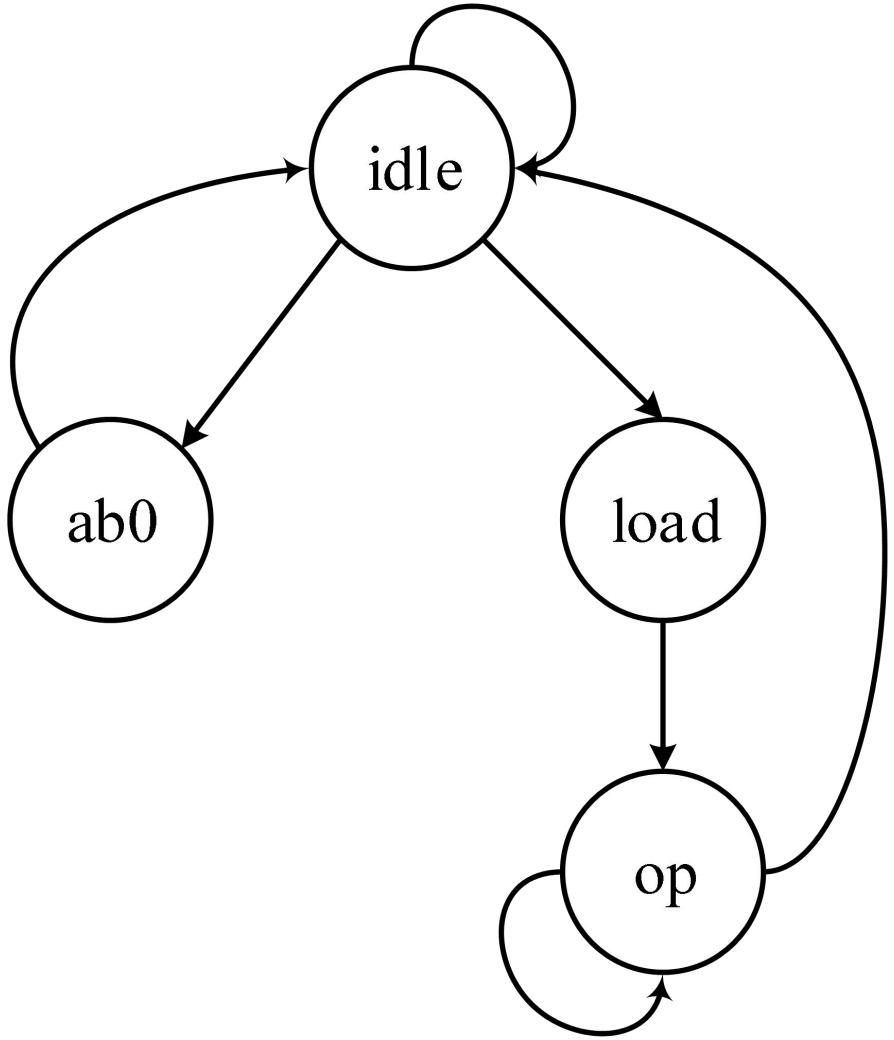
用寄存器r, a, n来模拟算法中的变量r, a， n

算法→算法状态图ASMD→有限状态图FSMD）

- 用n、a、r寄存器模仿三个变量
- 用RT运算实现顺序语句
- 状态划分：4个状态

  每个状态框可以划分

  为一个状态



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

Step2：产生数据通路

1. 列出所有可能的RT运算

2. 根据目的寄存器，把RT运算分组

3. 对每一组RT运算：

   构建目标寄存器

   构建RT运算的组合逻辑

   如果目的寄存器涉及多个RT运算，加上多路选择器

4. 加上必要的电路产生状态信号

r寄存器相关的RT运算

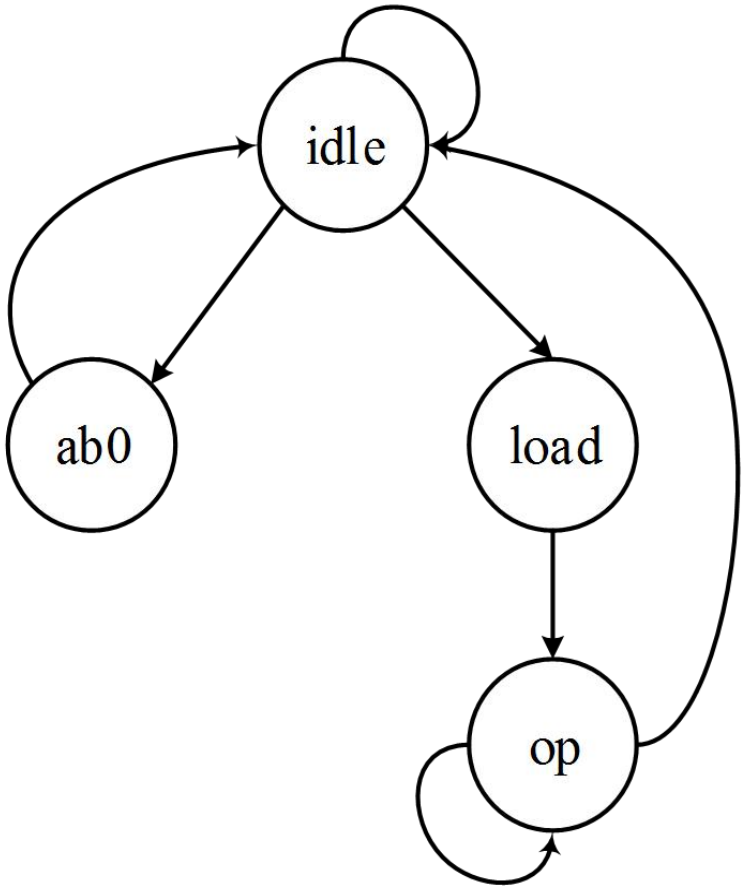
```
r <- r     在idle状态
r <- 0     在load和ab0状态
r <- r + a   在op状态
```

n寄存器相关的RT运算

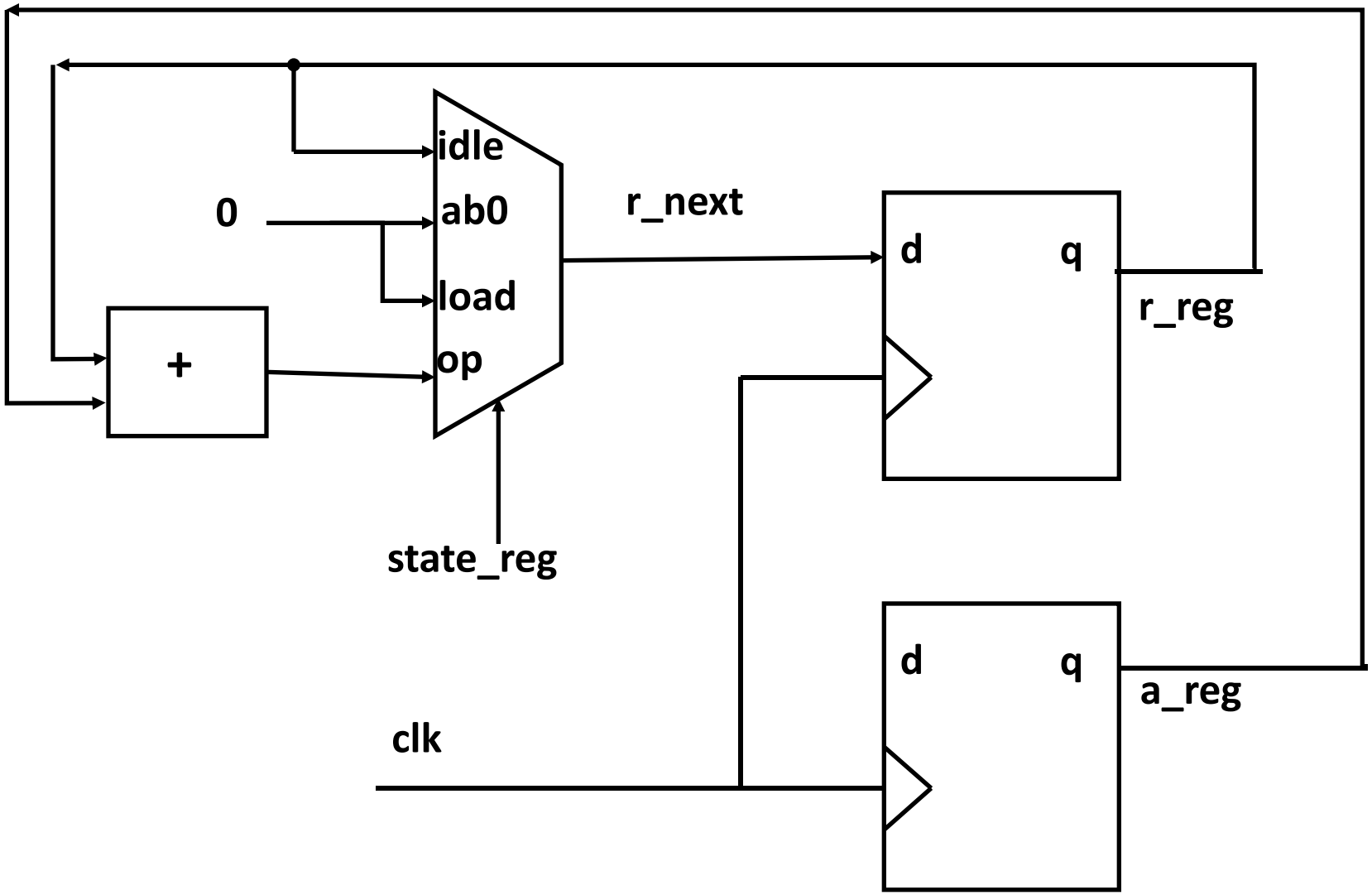
```
n <- n   在idle状态
n <- b_in    在load和ab0状态
n <- n - 1   在op状态
```

a寄存器相关的RT运算

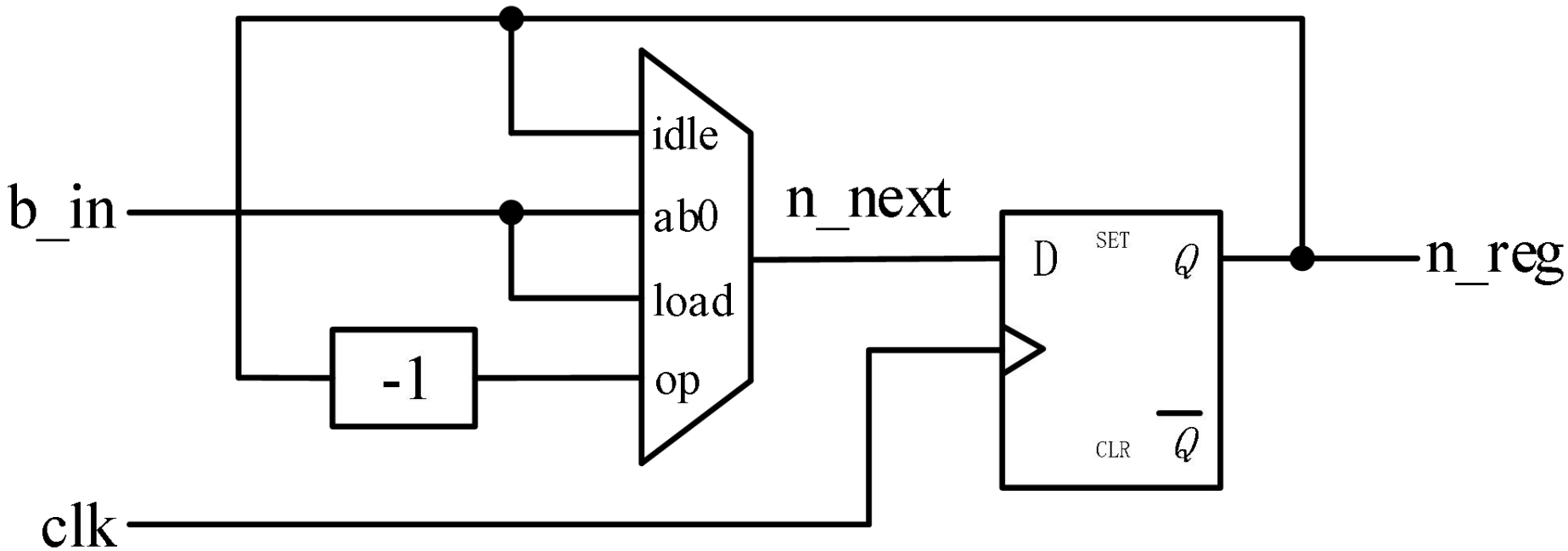
```
a <- a     在idle状态和op状态
a <- a_in    在load和ab0状态
```



r寄存器相关的数据通路



n寄存器相关的数据通路



a寄存器相关的数据通路