第二章 行为建模

1. 模块
   a) 基本结构

      output 是 wire 的子类型，不允许对他用<=或=赋值，只能用 assign；

      如果需要用<=或=赋值，可以额外定义一个同名的寄存器

```
module MODULE_NAME(PORT1, PORT2, PORT3, PORT4);
   input     [1:0]     PORT1;
   output               PORT2;
   inout                PORT3;
   output    reg        PORT4;       // 同时声明 output 端口并定义 reg

   reg PORT2;     // output 端口声明和 reg 定义分开
   // ....
endmodule     // 注意没分号
```

      或

```
module MODULE_NAME(
   input     [1:0] PORT1,
   output              PORT2,
   inout               PORT3,
   output reg        PORT4,
);
   reg PORT2;
   // ...
endmodule     // 注意没分号
```

   b) 实例化（同面向对象的编程）

```
module MODULE_NAME;
   OTHER_MODULE_NAME inst_OTHER_MODULE_NAME(
      .PARAM1( WIRE1 ),
      .PARAM2( WIRE2 )
   );
endmodule     // 注意没分号
```

2. always 和 initial
   a) always 结构反复执行其中的语句，永远不会退出或终止

      initial 结构只执行一次

   b) 单个 always 结构或 initial 结构内的语句是顺序执行的
   c) 不同 always 结构或 initial 结构的语句是并发执行的

      注意：initial 和 always 也是并发的，并不是先执行 initial 初始化再启动 always
   d) 当出现@、#、wait 时，always 和 initial 会挂起，直到@指定的事件发生、#的延迟达到、wait 的表达式为真
   e) 规范—always 语句中描述硬件行为，initial 语句中描述模拟（仿真）的初始化工作
3. 赋值
   a) 持续赋值 assign
   b) 非阻塞赋值<=
   c) 阻塞赋值=

4. 条件语句 if – then – else
   a) 基本结构

```
if (CONDITION1) begin
   // ...
end else if (CONDITION2) begin
   // ...
end else begin
   // ···
end
```

   b) else 子句为可选项，与最近的不完整 if 配对（同 C 语言）
5. 多分支语句 case
   a) 基本结构

```
case (ir[15:14])  // 注意不是 begin-end
   2'b00 : begin     // 多条执行语句用 begin-end
      // ...
   end
   2'b01, 2'b10 : // 指定多个条件，单个执行语句不需要 begin-end
   default : // 其他
endcase     // 注意没分号
```

   b) case & casex & casez
   case 枚举的条件如果出现了 x（不确定）或 z（高阻态），则需要完全匹配；
   casez 把 z 看作无关值；
   casex 把 z 和 x 都看作无关值
   c) full case：将 case 指定为完备，告知综合器不要生成保持器

```
case (c_state) //synthesis full_case
   2'b00 : n_state <= state_1;
   2'b01 : n_state <= state_2;
endcase
```

   等价于

```
case (c_state)
   2'b00 : n_state <= state_1;
   2'b01 : n_state <= state_2;
   default : n_state <= 2'bxx;
endcase
```

   d) parallel case：告知综合器去除冗余的优先级逻辑生成
   否则如果 case 中出现多重匹配则只会选择第一个匹配的条件

```
case (CONDITION)  // synthesis parallel_case
   // ···
endcase
```

6. 条件操作符（同 C 语言）
   REG = CONDITION ? IF_TRUE : IF_FALSE;
7. 循环语句
   a) 四种循环
      i. repeat 循环指定次数

```
repeat (REPEAT_TIMES) begin
    // ...
end
```

ii. for 循环（同 C 语言）：索引一般用 integer 类型

```
for (ASSIGNMENT1; CONDITION; ASSIGNMENT2) begin
    // ...
end
```

iii. while 循环（同 C 语言）

```
while (CONDITION) begin
    // ...
end
```

iv. forever 循环：无休止的循环

```
forever begin
    // …
end
```

b) 循环条件不能使用所在的 always 块或 initial 块外部的变量

c) 循环的继续与退出

Verilog 不直接支持像 C 语言的 continue 和 break 语句，但可以借助命名块和 disable 语句实现——

```
begin : break
    while (CONDITION) begin : continue
        // ...
        disable continue;     // 继续，即跳过本次循环
        // ...
        disable break;        // 退出循环
    end     // continue
end     // ~~break
```

8. 连接运算符{}

可以将多组连线合并为一个总线进行表示，如{A,B}把 A、B 两组线合并在一起（A 在高位，B 在低位）

9. 任务和函数

a) 基本结构

i. 任务

```
task TASK_NAME;     // 注意有分号，而且不需要列举端口
    // 端口声明
    input    PORT1;     // 注意是分号
    output PORT2;     // task 中不需要显式地给 output 定义 reg 就可以对他赋值
    inout    PORT3;
    // 寄存器定义
    reg A,B,C;
    // 程序段
    begin
        // ...
    end
```

```
endtask    // 注意没分号
```

或

```
task TASK_NAME(
  // 端口声明
  input   PORT1,    // 注意是逗号
  output PORT2,    // task 中不需要显式地给 output 定义 reg 就可以对他赋值
  inout   PORT3    // 最后一项不能有逗号
);  // 注意有分号
  // 寄存器定义
  reg A,B,C;
  // 程序段
  begin
    // ...
  end
endtask    // 注意没分号
```

ii. 函数

```
function [1:0] FUNC_NAME;    // 注意有分号，且不需要声明端口
  // 端口声明
  input   PORT1;    // 注意是分号
  input   PORT2;    // function 的端口只能是 input
  // 寄存器定义
  reg A,B,C;
  // 程序段
  begin
    // ...
    FUNC_NAME = 2'b00;   // 返回值
  end
endfunction    // 注意没分号
```

或

```
function [1:0] FUNC_NAME(
  // 端口声明
  input   PORT1,    // 注意是逗号
  input   PORT2,     // function 的端口只能是 input
  input   PORT3    // 最后一项不能有逗号
);  // 注意有分号
  // 寄存器定义
  reg A,B,C;
  // 程序段
  begin
    // ...
    FUNC_NAME = 2'b00;   // 返回值
  end
endfunction    // 注意没分号
```

b) 任务与函数的比较

	任务	函数
调用	单独的过程语句，不能在持续赋值语句中调用	作为表达式的一个操作数，可以在过程和持续赋值语句中调用
输入和输出	通过参数输入输出，零个或多个各种类型的输入输出	通过参数输入，参数不能为output，至少有一个输入；通过函数名输出，不能将 inout 类型作为输出
定时和事件控制（#、@、wait）	可以包含	不可以包含
调用其他任务或函数	可以调用其他任务和函数	可以调用其他函数，但不能调用其他任务

10. 标识符的作用域
    a) 超前引用和非超前引用
    模块、任务、函数和命名块中的标识符都可以超前引用；
    寄存器 reg 和线网 wire 变量必须先定义后引用
    b) 层次名称：引用非本层次的标识符

```
module top;
   reg     r;   // top.r
   wire    w;   // top.w

   always begin : y    // top.y
      reg     q;   // top.y.q
   end

   task t;
      begin : c     // top.t.c
         reg     q;   // top.t.c.q
         disable y;   //  使前边的 always 块失效
      end
   endtask

endmodule     //  注意没分号
```

直接引用一个名称：只能引用跟自己同一层次的，或者上一层次的，不能引用下一层次或隔壁层次的；如 top.t.c 可以直接引用 top.y，但不能直接引用 top.y.q

第三章 并发进程【仅考察概念，填空题】

1. 事件（@）
   a) 事件控制的两种基本形式
      i. 监视数值的改变（电平触发或边沿触发）
      ii. 有名事件：监视事件这一抽象信号
   b) 事件控制语句
      边沿触发：@(negedge clock) q <= data;
      电平触发：@(~clock) q <= data;
   c) 或逻辑的事件组合
      @(negedge clock or posedge rst) q <= data;
      @(negedge clock, posedge rst) q <= data;
   d) 上升沿和下降沿
      i. 下降沿 negedge：1 到 0、1 到 x、x 到 0
      ii. 上升沿 posedge：0 到 1、0 到 x、x 到 1
   e) 有名事件

```
module top;
   A a();
   B b();
endmodule

module A;
   event go;  // 声明事件 go
   // ...
   #50 -> go;  // ->表示触发事件，这里事件名称为 go
   // ...
endmodule

module B;
   // ...
   @a.go    // 等待实例 a 的 go 事件触发
      abc <= 1'b0;
   // ...
endmodule
```

2. 等待语句（wait）
   只对电平敏感，不对电平的变化敏感；
   等待直到表达式 a 为 True：wait(a)
3. 有名块的终止 disable
   disable 除了终止有名块外，还将终止块中被调用的函数和任务，但不会终止其他的监听本块的事件控制
4. 内部赋值
   a) a = #25 b;
      先读出 b，然后延时 25 个单位时间，然后把前边读到的值赋给 a；
      （#25 a = b，先延时 25 个单位时间，然后读出 b 并立即赋给 a）
   b) q = @(posedge w) r;

先读出 r，在等到 w 的上升沿，然后赋值给 q

c) w = repeat(2) @(posedge clock) t;

先读出 t，然后等待两个 clock 的上升沿，再赋值给 w

d) 同一个 begin 内的<=

```
begin
   a <= #50 1;
   b <= #100 2;
   #100 c <= 1;
end
```

a 在 0 采样，50 赋值；

b 在 0 采样，100 赋值；

c 在 100 采样并赋值

第四章 逻辑级建模

1. 基元逻辑门
    1. 多输入单输出：and, nand, or, nor, xor, xnor
       将输出作为第一个参数，其他作为输出；
    2. 单输入多输出：not, buf
       将最后一个参数作为输入，其他作为输出
    3. 三态单输入多输出：bufif0, bufif1, notif0, notif1
       最后一个参数为使能端，倒数第二个参数为输入端
2. 逻辑值
    1. 0
    2. 1
    3. x，未知
    4. z，高阻态
    5. 逻辑运算规则

表 3.2(a) 取反运算符的运算规则

~	结 果
1	0
0	1
x	x

表 3.2(b) “按位与”运算规则

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

与：只要其中一个是0，结果必为0

表 3.2(c) “按位或”运算规则

|	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

或：只要其中一个是1，结果必为1

表 3.2(d) “按位异或”运算规则

^	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

表 3.2(e) “按位同或”运算规则

^~	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

3. 线网
    1. 普通线网 wire，带线与的线网 wand
    2. 基本定义
       wire w1; // 无延时的线网
       wire #(3,5) w2; // 带延时的线网，上升沿 3 个单位时间，下降沿 5 个单位时间
4. 持续赋值
   只有线网 wire 类型能被持续赋值；
   assign sum = a_i ^ b_i ^ c_i; // 注意必须是=而不是<=
   wire AXorB = a ^ b; // 声明线网的时候也可以直接持续赋值
5. 延迟（上升沿延迟、下降沿延迟、关断延迟）
   not #5 (ndata, data); // 匿名的非门
   nand #(12, 15) qQ(q, nq, wa);
   bufif1 #(3, 7, 13) qDriver(qOut, q, enable);
   线网同理

指定最小延迟、典型延迟、最大延迟：

#(d1_min : d1_typ : d1_max, d2_min : d2_typ : d2_max, d3_min : d3_typ : d3_max)

6. 时间单位定义

`timescale <time_unit>/<time_precision>

分别定义单位时间和时间精度

时间单位可以从 fs 到 s

第六章 逻辑综合

1. 使用门和组合逻辑

   assign out = (sel) ? A : B;

   assign f = (a & b & c) | (a & ~b & ~c) | (~a & (b |c));

   而不是逐一定义逻辑门和连线

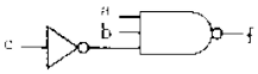
2. 使用过程语句来说明组合逻辑

```
always @ (a or b or c)    // 电平敏感
  if ( a== 1 )
    f = b;      // 阻塞式赋值
  else
    f = c;
```

3. Case 语句中 default 直接输出 x 表示无关项，否则如果 case 中没有列举出所有情况，综合器会刻意去为未列举出的情况加一个保持器；

```
module synCaseWithDC (f, a, b, c);
    output          f;
    input           a, b, c;
    reg             f;

    always @(a or b or c)
        case ({a, b, c})
            3'b001:         f = 1'b1;
            3'b010:         f = 1'b1;
            3'b011:         f = 1'b1;
            3'b100:         f = 1'b1;
            3'b110:         f = 1'b0;
            3'b111:         f = 1'b1;
            default:        f = 1'bx;
        endcase
endmodule
```



没有额外的保持器

4. Case 枚举的数值可以出现 x、z、?，都表示该条件无关
5. 循环语句 repeat 是不可综合的
6. 时序元件的推断

   a) 锁存器（电平触发）

      两种情况：显式的电平触发条件下的赋值、隐式的锁存器推断

      推断 always 语句中的锁存器：

      i. 至少存在一条不对输出进行赋值的控制路径

      ii. 敏感列表不允许包含任何边沿敏感的说明

      如：

```
always @(g or d or reset)
  if (~reset)
    Q= 0;
```

      敏感列表不包含边沿敏感的说明，if 语句中只说明了 reset==0 时的操作，没有说

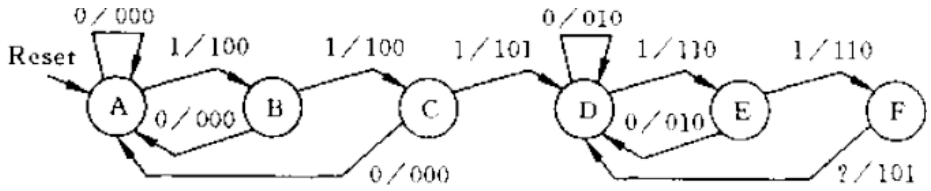
明 reset==1 时的操作，此时综合器会为 reset 保存上一次的数值（即额外布置了锁存器）

应尽量避免使用锁存器，容易产生毛刺，时序不确定，而且 FPGA 往往综合成一个逻辑门加一个触发器，浪费资源；

b) 触发器（边沿触发）
两种情况：显式的边沿触发条件下的赋值、隐式的推断

触发器推断类似锁存器，只不过是在边沿触发的 always 块中

7. always 时序逻辑
   a) 敏感列表中每个信号都要指定边沿，只能包括时钟、复位、置位条件
   b) 第一个语句必须是 if 语句
   c) 必须先对置为和复位条件进行测试
   d) 赋值必须用非阻塞是赋值，尽管正规的阻塞式赋值也能正确的综合，但它无法被正确的模拟，所以应当避免
   e) 不能在同一个 always 块中既描述时序逻辑又描述组合逻辑
8. 三态器件的推断：使用了高阻态
9. 有限状态机（FSM）描述
   a) 应用场景
      i. 电路有时序规律或逻辑顺序
      ii. 电路输出与特定状态有关
   b) 摩尔（Moorc）状态机：当前状态决定输出
      米利（Mealy）状态机：当前状态和输入共同决定输出
   c) 状态转换图



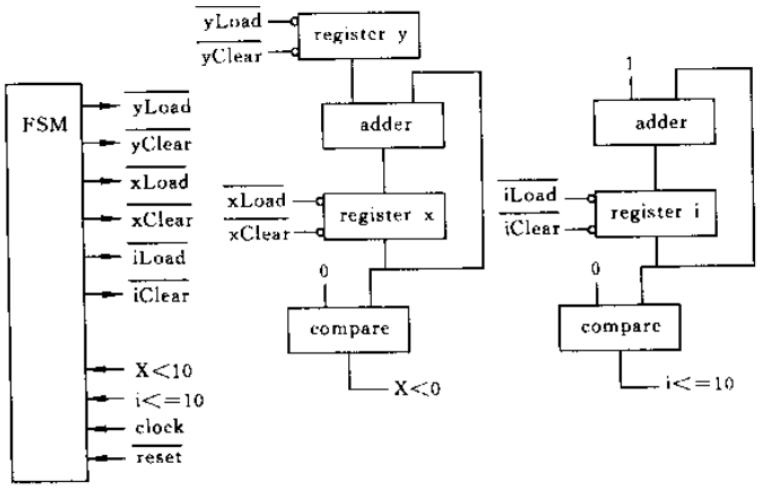
   d) 状态编码
      i. 二进制
      ii. 格雷码
      iii. 独热码：每个状态只有一位为高电平
      减少组合逻辑的使用量，减少毛刺，但占用比较多的触发器；
   e) 三段式描述
      状态推进、状态改变、输出控制

```
always @(posedge clk_i or negedge rst_i)
if(~rst_i)
    c_state <= S_IDLE;
else
    c_state <=n_state;


always @ *
begin
   case(c_state)
      S_IDLE:
        if(in)
        n_state = S_S0;
        else
        n_state = S_IDLE;
      S_S0:
        if(in)
        n_state = S_S0;
        else
        n_state = S_IDLE;
       default: n_state = S_IDLE;
   endcase
end

always @ *
begin
   case(c_state)
      S_IDLE:
        out = 1'b0;
        n_state = S_IDLE;
      S_S0:
        if(in)
        out = 1'b1;
        else
        out = 1'b0;
       default: out = 1'b0;
   endcase
end
```

10. 数据通道，RTL 建模



将系统分解为输入接口、数据通道、输出接口；

输入接口直接把系统输入保存到寄存器当中；

数据通道进一步按功能分解成若干个小模块，各个小模块处理某一部分或某一阶段的数据，从输入接口或上一级的模块获取输入，并传递给下一级；

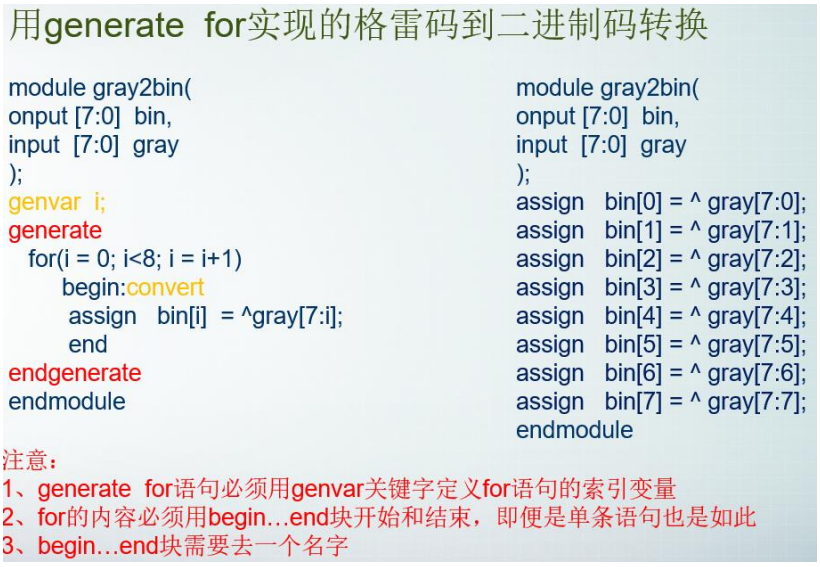
输出接口则负责对外输出；

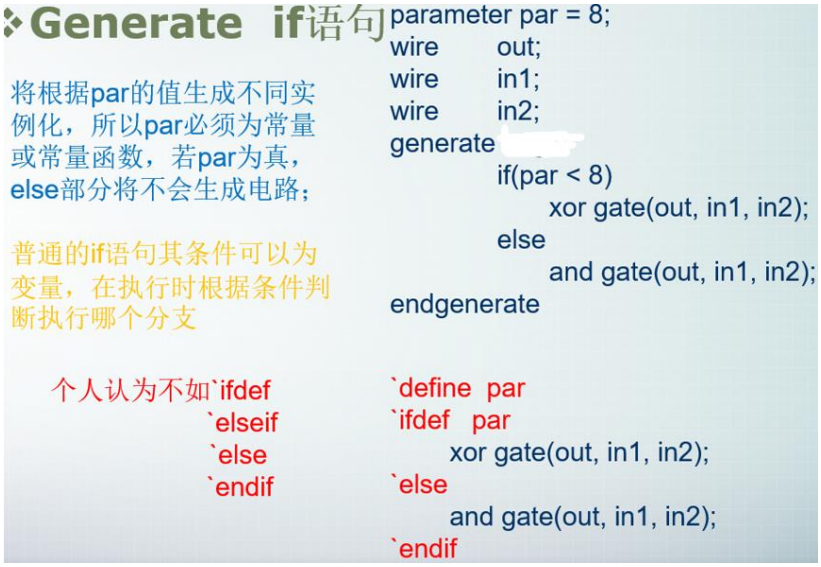
以上工作的流程一般用 FSM 来控制协调；

杂七杂八

1. 算数运算符和比较运算符只要操作数中出现了 x 或 z，则运算结果为 x
2. 连接操作{}中常数必须指定宽度
3. 敏感列表指定为*表示对 always 块中所有条件、右值用到的量都敏感
4. 常见的不可综合语句
   a) initial
   b) 延时#
   c) while、repeat、forever
   d) 带延时的 task
5. 一个变量只能在一个 always 里赋值
6. 模块的输入输出端口
   a) 输入端口只能是 wire 类型，可以由 wire 或 reg 驱动
   b) 输出端口可以是 wire 或 reg 类型，只能由 wire 驱动
7. 参数
   a) parameter PARAM = 8;
      全局参数，上层文件也可以修改这个参数
   b) localparam PARAM = 8;
      局部参数，上层文件不能修改这个参数
8. 宏定义
   `define M 5
   在整个编译期间无论在哪个文件 M 都是有效的
9. 系统任务与函数
   a) $display：按格式控制符指定方式打印输出，每次执行都会打印
   b) $monitor：输出变化时做出响应，监测变量，变量改变时打印
   c) $time：显示当前仿真时间
   d) $stop：暂停仿真，执行 run 指令后可以继续仿真
   e) $finish：结束仿真
10. reg 变量在声明时可以进行初始化（对应 initial 的 0 时刻）；
    功能模块不要作这个初始化，应当在 reset 信号到来时进行初始化
11. 有符号数
    a) Reg 和 wire 可以声明为带符号变量
       reg signed [31:0] data_reg;
    b) 常数可以指定为有符号数：8'sh1b（八位有符号十六进制表达）
    c) 算数移位：>>>和<<<
    d) 系统函数$signed 和$unsigned 实现有符号数和无符号数的转换
    e) 可变向量域
       vector[31-:8]等价于 vector[31:24],
       vector[16+:8]等价于 vector[23:16]
12. 总线和数组
    定义：reg [31:0] bram[255:0];
    引用：（先指定数组索引再指定某一位）bram[200][30]
    虽然 verilog-2001 允许多维数组，但电路不易理解，应避免
13. 生成块（verilog-2001 支持）
    允许 for 根据索引生成、if 或 case 根据条件生成；

generate 语句都只在编译时有效；





Case 与 if 类似

14. 位宽声明的索引可以从小到大，也可以从大到小
    大端模式：reg [0:7] a;
    小端模式：reg [7:0] a;
15. &、|、^
    a) 双目运算符：同 C 语言
    b) 单目运算符：作为缩位运算符，又称归约运算符
       运算过程如下，首位和第二位运算，运算结果再跟第三位，新的结果再跟第四位运算，依次类推，直到最后一位，最终得到一位运算结果
    c) 复合运算符，&、|、^可以和~构成符合运算符~&、~|、~^表示与非、或非、同或
    d) 同或除了可以写成~^也可以写成^~
    e) 注意奇校验时计算奇偶校验位，应为 parity=~(^data)而不是 parity=~^data
16. 8'b1、8'bz、8'b1z、8'bz1z
    8'b1 是 0b00000001；
    8'bz 是 0bzzzzzzzz；

8’b1z 是 0b0000001z；

8’bz1z 是 0bzzzzzz1z；

17. 0、x、z 都是逻辑假，只有 1 才是逻辑真
18. 示例数组构造模块实例

```
module reggac (Q, D, clock, clear);
   output [7:0] Q;
   input   [7:0] D;
   input clock, clear;

   // 分别引用 Q、D 线，共用 clear、clock 线
   dff r[7:0] (Q,D,clear,clock);
endmodule
```

将被展开为

```
module reggac (Q, D, clock, clear);
   output [7:0] Q;
   input   [7:0] D;
   input clock, clear;

   dff r[7] (Q[7],D[7],clear,clock);
   dff r[6] (Q[6],D[6],clear,clock);
   dff r[5] (Q[5],D[5],clear,clock);
   dff r[4] (Q[4],D[4],clear,clock);
   dff r[3] (Q[3],D[3],clear,clock);
   dff r[2] (Q[2],D[2],clear,clock);
   dff r[1] (Q[1],D[1],clear,clock);
   dff r[0] (Q[0],D[0],clear,clock);

endmodule
```

而

```
module reggac (Q, D, clock, clear);
   output [7:0] Q;
   input   [7:0] D;
   input clock, clear;

   // 分别引用 Q、D 线，共用 clear、clock 线
   dff r[3:0] (Q,D,clear,clock);
endmodule
```

将被展开为

```
module reggac (Q, D, clock, clear);
   output [7:0] Q;
   input   [7:0] D;
   input clock, clear;
```

```
    dff r[3] (Q[7:6],D[7:6],clear,clock);
    dff r[2] (Q[5:4],D[5:4],clear,clock);
    dff r[1] (Q[3:2],D[3:2],clear,clock);
    dff r[0] (Q[1:0],D[1:0],clear,clock);

endmodule
```