ADS Language Support Reference Manual 用户手册

（Version 1.0）

深圳市紫光同创电子有限公司

文档版本修订记录

版本号	发布日期	修订记录
V1.0	2022.07.12	初始版本

目录

表目录

图目录

1 Verilog Language Support Constructs

本章节主要介绍 ADS 综合工具支持的 verilog 语言结构。

ADS Verilog Language Support Constructs 主要分为以下种：

- Supported Verilog Constructs
- Unsupported Verilog Constructs
- Ignored Verilog Language Constructs

1.1 ADS 支持的 verilog 语言结构

1.1.1 Module and Macromodule

The module/endmodule block is the basic compilation unit in Verilog, the module begin with module(macromodule) and terminated with endmodule. module 的模板如下。

module| macromodule module_name(port_list);

port_declaration;

net/variable declaration;

// continuous assignment

wire signal_name;

assign signal_name = expression;

always @(event_expression)

begin

    statements;

end

endmodule

在 verilog module 中可以使用 continuous assignment，always block, module instantiation, gate instantiation 等描述硬件

1.1.2 Supported 和 Unsupported Verilog Constructs

下表为 supported 和 unsupported verilog 语法结构。不可支持的 verilog constructs 会报错。

Supported Verilog Constructs	Unsupported Verilog Constructs
Register data type: reg, integer, time	Register data type: event, real, realtime
Net data type: wire, tri, tri0, tri1, wand, wor, triand, trior	Net data type: trireg
gate instantiation: bufif0, bufif1, notif0, notif1, and, nand, or, nor, xor, xnor, buf, not	gate instantiation: tranif0, tranif1, rtranif0, rtranif1, nmos, pmos, rnmos, rpmos, cmos, rcmos, tran, rtran, pulldown, pullup
module instantiantion	User-defined primitives(UDPs)，UDPs begin with keyword primitive and terminated with endprimitive
input, output, inout	
parameter, localparam, specparam genvar	time 和 realtime 类型的 parameter
always construct, task, function	event trigger
generate/endgenerate generate if, generate for, generate case	
continuous assignments	
procedural statements: Blocking procedural assignments = Nonblocking procedural assignments <= begin/end block, if-else-if, case, casex, casez, for, while, disable, System task($readmemb, $readmemh)	procedural statements: The left hand value must be register. Do not use = and <= for the same register fork/join, assign/desassign force/release, wait, forever, repeat
Binary Operators: +, -, *, /, %, **, <, >, <=, >=, ==, !=, ===, !==, &&, ||,	

&, |, ~|, ^~, ~^, ^, <<, >>, <<<, >>>
Unary Operators
+, -, !, &, ~, |, ^, ~&, ~|, ~^
other:
?:, { }, {{ }}

Compiler directives:
`include, `define, `endif, `ifdef,
`ifndef, `else, `elsif, `undef, `default_nettype

表 1-1 Supported 和 Unsupported Verilog Constructs

1.1.3 Ignored Verilog Language Constructs

设计中的code中包含Ignored Verilog Language Constructs，ADS综合工具会ignore并继续执行。

- initial block
- delay and delay control
- strength

  drive_strength ::= (strength0 , strength1) | (strength1 , strength0) | (strength0 , highz1) | (strength1 , highz0) | (highz1 , strength0) | (highz0 , strength1)

  charge_strength ::= (small) | (medium) | (large)
- specify block
- config/endconfig
- system task enable

  Ignored expected for $readmemb and $readmemh in initial
- Compiler Directives

  `line, `celldefine, `endcelldefine, `resetall, `timescale, `unconnected_drive, `nounconnected_drive

1.2 ADS 支持的 data type 语言结构

Verilog data type 分为如下几类：

- Register Data Type
- Net Data Type

- 其他 Data Type

1.2.1 Register Data Type

支持的 register data type 如下表中所示：

Data Type	Description
reg	默认为 1 bit wide，如果超过 1 bit，则需要 range declaration 设置 reg 的位宽
integer	默认位宽为 32 bit，不允许有 range declaration
time	默认位宽为 64 bit，不允许有 range declaration

表 1-2 支持的 register data type

设置变量 vector 宽度的 range declaration 中表达式必须为常量。

如：

reg [7:0] reg1;//变量 reg1 的宽度为 8

reg [DATA_WIDTH-1 :0] reg2;//变量 reg2 的宽度为 DATA_WIDTH，DATA_WIDTH 须为常量值

1.2.2 Net Data Type

支持的 net data type 如下表所示：

Data Type	Description
wire	signal gate 或 continuous assignment 驱动的 net
tri	多驱动数据类型
tri0	多驱动数据类型，综合时当成 wire 处理
tri1	多驱动数据类型，综合时当成 wire 处理
wand	线与
wor	线或

表 1-3 支持的 net data type

1.2.3 其他 Data Type

Data Type	Description
parameter	为变量指定常量值
localparam	local parameter 参数，不允许值的重写
genvar	generate for 中 index control 的变量

表 1-4 其他 Data Type

example:

parameter P = 10;

localparam WIDTH = 10;

genvar i;

1.3 Instantiation

1.3.1 module instantiation

module 实例化实现可以实现模块的层次化描述。建议代码风格如下：

```
module AND(A,B,Y);
output      Y;
input       A,B;
parameter N=5;
assign   Y = A && B;
endmodule

module AND(a,b,y);
output y;
input   a,b;
ADN #(10) inst_name(.A(a),.B(b),.Y(y));
endmodule
```

进行 module instantiation 时必须给定 inst_name，否则报错退出。port connect 的连接方式可以是 order 或 named 类型，不能采用两者混用方式。

1.3.2 gate instantiation

gate instantiation 的语法结构：

gatetypekeyword [instname] (portlist);

ADS 支持的 gate type 如下：

gate type	Description
and	与，实现输入信号的与运算
nand	与非
or	或，
nor	或非
xor	异或
xnor	同或
buf	缓冲器
not	非
bufif0	三态缓冲，控制信号低有效
bufif1	三态缓冲，控制信号高有效
notif0	三态非，控制信号低有效
notif1	三态非，控制信号高有效

表 1-5 gate type

1.4 Statements

Statements occur within procedures such as always, initial, task and function。包括 case statement, loop statement, condition statement, blocking assignment, noblocking assignment 等。

1.4.1 Case Statement

A case statement include begin keywords case, casex or casez, terminal keyword endcase.

case keyword	Description
case	allows branching on multiple conditional expressions based on case statement matching
casex	'x' or 'z' value in the case expression is treated as don't care
casez	'z' value in the case expression is treated as don't care
default	its statement will be selected when none of the select expressions is valid
endcase	terminal of case, casex, casez statement

表 1-6 Case Statement

1.4.2 Loop statement

Loop statement 包括 for statement 和 while statement

loop keyword	Description
for	it is used to modify blocks of procedural statements. The first assignment is executed initially and then the expression is evaluated repeatedly based on condition value
while	It executes the given statement until the expression becomes true

表 1-7 Loop statement

1.4.3 Conditional Statement

If statement is conditional statement, the keywords include if、else and else if。Syntax of if statement as follows：

if (condition1)

statement1;

[else if (condition2) statement2;]

[else statement3;]

1.4.4 Blocking assignment and Noblocking assignment

阻塞赋值和非阻塞赋值用在 always、initial construct 中，赋值对象为 register 类型的变量。对同一 register 变量不能同时使用阻塞赋值和非阻塞赋值。

```
always @ (posedge clk or negedge r)
  if (~r)
    q <= 1'b0;
  else
    q <= d; // noblocking assignment
```

1.4.5 Disable Statement

The disable statement provides the ability to terminate the activity associated with concurrently active. It can be used within blocks and tasks to disable the particular block or task containing the disable statement. syntax form of disable statement as follows：

```
disable block_id/task_id;
```

1.5 Operator

1.5.1 Binary Operator

+, -, *, /, %, **, <, >, <=, >=, ==, !=, ===, !==, &&, ||, &, |, ~|, ^~, ~^, ^, <<, >>, <<<, >>>

上述运算符的相关表述如下：

Operators	Usage	Function
+	a + b	a plus b
-	a - b	a minus b
*	a * b	a multiplied by b
/	a / b	a divided by b
%	a % b	a modulus b(ADS unsupported)
**	a ** b	a to the power of b
<	a < b	a less than b

>	a > b	a greater than b
<=	a <= b	a less than or equal to b
>=	a >= b	a greater than or equal to b
==	a == b	a logical equality to b
!=	a != b	a logical inequality to b
===	a === b	a case equality to b
!==	a !==b	a case inequality to b
&&	a && b	a logical and b
||	a || b	a logical or b
&	a & b	a bitwise and b
|	a | b	a bitwise or b
~|	a ~| b	a bitwise nor b
^~,~^	a ^~ b, a ~^ b	a bitwise equivalence b
^	a ^ b	a bitwise exclusive or b
<<	a << b	logical left shift
>>	a >> b	logical right shift
<<<	a <<< b	arithmetic left shift
>>>	a >> b	arithmetic right shift

表 1-8 运算操作符

1.5.2 Unary Operators

+, -, !, ~, &, |, ^, ~&, ~|, ~^

上述 unary operators 运算符的相关表述如下：

Operators	Usage	Function
+	+a	Arithmetic plus
-	-a	Arithmetic minus
!	!a	Logical negation
~	~a	Bitwise negation
&	&a	Reduction and
|	|a	Reduction or
^	^a	Reduction xor
~&	~&a	Reduction nand
~|	~|a	Reduction nor
~^	~^a	Reduction xnor

表 1-9 unary operators 运算符

1.5.3 Other

? : , { }, {{ }}

Operators	Usage	Function
? :	sel ? a : b	Conditional
{ }	{a, b}	Concatenation
{{ }}	{a{b}}	replication

表 1-10 其他操作符

1.6 Blocks

1.6.1 begin/end block

begin/end block groups multiple statements into always block, example as follows：

```
module test (in1, in2, out1, out2, clk);
input in1, in2, clk;
output reg out1, out2;
always@(posedge clk)
begin
out1 <= in1 & in2;
out2 <= in1 | in2;
end
endmodule
```

1.6.2 generate/endgerate block

A generate block is created using one of the generate-loop, generate-conditional, or generate-case format. The block begin with keyword generate and terminated with endgenerate.

```
//generate if
generate begin
  if(sel)
    assign dout1 = d1;
  else
    assign dout2 = d2;
  end
endgenerate

//generate for, j must be genvar type
generate
      for (j=0; j<=3; j =j + 1)
      begin : u
      adder8 add (sum[8*j+7:8*j], c0[j+1],
       a[8*j+7:8*j], b[8*j+7:8*j], c0[j]);
      end
```

```
//generate case
generate
        case (A)
           0: assign Y = 2'b00;
           1: assign Y = 2'b01;
           2: assign Y = 2'b10;
           3: assign Y = 2'b11;
        endcase
    endgenerate
```

2 SystemVerilog Language Support Constructs

2.1 ADS 支持的 SystemVerilog data type 语言结构

2.1.1 SystemVerilog 支持的基本数据类型

shortint	2-state, SystemVerilog data type, 16-bit signed integer
int	2-state, SystemVerilog data type, 32-bit signed integer
longint	2-state, SystemVerilog data type, 64-bit signed integer
byte	2-state, SystemVerilog data type, 8-bit signed integer or ASCII character
bit	2-state, SystemVerilog data type, user-defined vector size
logic	4-state, SystemVerilog data type, user-defined vector size

表 2-1 System Verilog 支持的基本数据类型

2-state 和 4-state 的区别：

4-state(4-valued) data types 的 value 值可以是 1, 0, x, z 四种

2-state(2-valued) data types 的 value 值可以是 1 和 0

数据类型 byte，shortint，int，integer 和 logicint 默认是有符号，数据类型 bit，logic 和 reg 默认是无符号的，这些类型的数组类型相同。

2.1.2 用户自定义数据类型

System Verilog 数据类型可以使用 typedef 扩展为用户自定义的类型。

example：

```
module sub6 (
input [11:0] in,
input [3:0] in2,
output [3:0] out0, out1, out2, out3
);
```

```
typedef struct {
   bit [3:0] a;
   bit [15:0] b;
} st_type;
logic [3:0] pipe [3:0];
logic [3:0] pipe_in [1:0];
st_type st_data;
always @(*) begin
   pipe_in = { in[11:8], in[7:4] };
end
always @(*) begin
   st_data.a = in[3:0];
end
always @(*) begin
      pipe = {pipe_in[0], st_data.a, in2, 4'b1111};
end
assign out0 = pipe[0];
assign out1 = pipe[1];
assign out2 = pipe[2];
assign out3 = pipe[3];
endmodule
```

2.1.3 struct

结构体是一个名称下的变量或常量的集合，整个集合可以用结构体名进行引用。结构体不同于数组，数组是同类型同大小的数据的集合，结构体是不同类型和大小的数据成员的集合。

需要注意两点：

如果 struct_union_member 中有 void 类型，则必须为 tagged unions。

如果 packed_dimension 在 struct 中使用，则必须声明为 packed 类型。

结构体成员中的 data type 类型可以是 shortint，int，longint，byte，bit，logic，reg，integer 和 time 及用户自定义的一些数据类型。

默认情况下，结构体是 unpacked 的，即结构体的成员使用一个共同的结构体名称，但是它们之间是相互独立的。packed 类型的结构体是按照指定的顺序以相邻的位来存储结构体成员。packed 结构体被当做一个向量存储，结构体的第一个成员在向量的最左侧，即最高位；结构体的最后一个成员为向量的最低位。

packed 结构体可以通过成员名引用，也可以通过结构体向量相应的位来引用。如下两种方式代表 data 结构体的相同成员。

```
struct packed {
    logic valid;
logic [7:0] tag;
bit [15:0] data;
} data;
data.tag = 8'hfe;
data[23:16] = 8'hfe;//两种方式都是对 struct 的 tag 成员进行赋值。
```

当前 ADS 不支持的结构体的形式：

- 结构体声明中有 packed dimension 和 unpacked dimension
- 对结构体成员变量的 bit/part select 引用，如 str.a[3:0], str.a[2]

2.2 System Verilog 支持的多维数组

2.2.1 多维 unpacked 数组

Unpacked dimensions 是声明在数组名后的 range 信息。

example：

```
reg [7:0] mem [7:0][15:0]; //unpacked dimensions[7:0][15:0]
```

Unpacked dimensions 可以为 single range 的形式，如 reg [7:0] mem [8]，表示 mem 有 8 个元素，等价于 reg [7:0] mem [0:7]。

多维 unpacked 数组赋值注意事项：

- 支持整个多维 unpacked 数组变量的赋值和读取，但 unpacked dimensions 的个数和每个维度的元素个数及每个元素的宽度需要保持一致。
- port 端口的 unpacked dimension 只支持一维，如果 unpacked dimension 超过一维则报错处理。

- 支持 unpacked 数组的 index 或 part-select 方式的赋值，如 m = mem[1]，或 m1 = mem[1:2];，如果为 mem[1:2]，range 必须为常量，且 unpacked dimension 为一维。

2.2.2 多维 packed 数组

Packed dimension 是声明在数组名前的 range 信息。

example:

wire [7:0] [15:0] tmp; //packed dimensions [7:0][15:0]

多维 packed 数组赋值注意事项：

- 支持整个多维 packed 数组变量的赋值和读取。
- 支持 packed 数组的 index 或 part-select 方式的赋值和读取，如 m = tmp[1]，或 m1 = tmp[1:0]。
- 支持多维 packed 数组的 index 为变量的形式，如 m = tmp[idx]，当前 ADS 只支持赋值语句右值多维 packed 数组的 index 为变量。

2.3 Operators and Expressions

2.3.1 assignment pattern

Assignment pattern 一般用来对 unpacked 数组和结构体等进行初始化或赋值。
Assignment pattern 由大括号和撇号组成。

Assignment pattern 的语法结构：

'{ listofValues }

listofValues 为逗号隔开的 value 值列表，value 值有以下几种形式：

listofValues：

'{ default: value }

'{ type:value }

'{ index:value }

'{ name:value }

'{ value }

'{const{ value} }

default:value 表示将 value 值赋给 unpacked 数组或结构体成员的所有元素

index:value 表示将 value 值赋给 unpacked 数组的下标为 index 的元素

'{ value1,value2,…,valueN }表示按照顺序将对应的 value 值赋给 unpacked 数组的各个元素

当前 ADS 不支持'{ type:value }和'{ name:value }形式的 assignment pattern。

Example：

```
module test(in,reset,out);
parameter SIZE=4;
input reset;
input [1:0] in[SIZE-1:0];
output reg [1:0] out [SIZE-1:0];
reg [1:0] tmp[SIZE-1:0] = '{SIZE{2'b10}};   //initialize tmp by assignment pattern
always @(*)
    if (reset)
        out = tmp;
    else
        out = in;
endmodule
```

2.4 always construct

2.4.1 always_comb

SystemVerilog 中的 always_comb 用于组合逻辑建模，always_comb 进程中必须为组合逻辑。如果不是表示组合逻辑需要给 warning。

always_comb 的使用需要注意以下两点：

- 不允许有敏感列表。
- 不能包含 block timing，event controls(@(*), @(event_expression), @*,etc)，fork,，join 或 wait statement。

always_comb 会默认将该进程中所有赋值语句的右值加入敏感列表，左值变量不能在其他进程中进行赋值。

2.4.2 always_latch

SystemVerilog always_latch 进程块用于锁存逻辑建模。同 always_comb 一样，always_latch 也不能包含敏感列表及 block timing，event controls，fork,，join 或 wait statement 语句。

2.4.3 always_ff

SystemVerilog always_ff 进程块用于时序逻辑建模。always_ff 必须包含 event control，且进程块中赋值语句的左侧变量不能在其它进程中进行赋值。

2.5 Hierarchy

2.5.1 Compilation Units

SystemVerilog 支持使用编译单元进行单独编译。Compilation units 编译单元允许在 package，module，interface 等的外部进行变量等的定义。Units 是对所有 module 可见的。

编译单元中的变量等可以同时在 module 中进行定义，如果不是显式的引用编译单元中的变量，则优先使用 module 内部的同名变量。

example:

```
function [7:0] func(input [7:0] in1, input [7:0] in2);
    begin
        func = in1 + in2;
    end
endfunction

module test(in1, in2, clk, out1, out2);
input [7:0] in1, in2;
input clk;
output reg [7:0] out1, out2;
function [7:0] func(input [7:0] in1, input [7:0] in2);
    begin
        func = in1 | in2;
    end
endfunction
always @(posedge clk)
```

```
begin
    out1 = func(in1, in2);//local func
    out2 = $unit::func(in1, in2);//global func in unit
end
endmodule
```

2.5.2 Packages

Package 的概念来源于 VHDL，package 允许在一个或多个 compilation units，modules 或 interfaces 间共享数据类型，用户自定义数据类型，parameter，function 和 task 等。package 中的变量只能被访问读取，不能在 module 中进行写操作。

package 中的成员的引用方式有以下几种：

- 通过::操作符直接引用，如 pack::mul(a,b);
- 使用 import 导入特定的 package items，将特定的 items 导入到 module 中
- 使用通配符*将 package 中的 items 导入 module，如 import pack::*;
- 在当前的 package 中可以通过 import 导入另一个 package 中的 items

example:

```
package mypack;
    parameter FACT_OP = 2;
    task factorial(input integer operand, output [1:0] out1);
        integer nFuncCall = 0;
        begin
            if (operand == 0)
            begin
                out1 = 1;
            end
            else
            begin
                nFuncCall++;
                factorial((operand-1), out1);
                out1 = out1 * operand;
```

```
                end
            end
        endtask
endpackage

import mypack::*;

module src (input [1:0] a, input [1:0] b,
        output logic [2:0] out );
        logic [1:0] out_tmp;
        always_comb
        factorial(FACT_OP,out_tmp);
        assign out = a + b + out_tmp;
endmodule
```

3 组合逻辑和时序逻辑建模

3.1 组合逻辑建模

组合逻辑建模实现方式有三种：

- 门级建模
- continuous assignment
- 过程赋值 always 语句

3.1.1 门级建模

门级建模即通过 verilog 的 gate instantiation 方式来实现。ADS 综合工具支持的门级组合逻辑如下：

模块名	逻辑功能
and	与，实现输入信号的与运算
nand	与非
or	或，
nor	或非
xor	异或
xnor	同或
buf	缓冲器
not	非
bufif0	三态缓冲，控制信号低有效
bufif1	三态缓冲，控制信号高有效
notif0	三态非，控制信号低有效
notif1	三态非，控制信号高有效

表 3-1 门级组合逻辑关系表

3.1.2 continuous assignment

连续赋值语句必须以 assign 开头。在与门级电路相同的代码层次实现。

语法结构如下：

assign lvalue = expression;

continuous assignment 的左值必须为 wire 类型变量。

3.1.3 过程赋值 always 语句

always 语句进行组合逻辑建模的代码风格如下，always 语句后为 event control。当 A 值发生变化时，会执行 begin/end 中的行为。

```
always @(A)
begin
    Y[15:0]  = A + 22;
Y[31:16] = A + 33;
end
```

除了上述直接在@后添加敏感列表还可以直接用@(*)代替，避免敏感列表不全引起的仿真 mismatch。

```
always @(*)
begin
    Y[15:0]  = A + 22;
Y[31:16] = B + 33;
end
```

使用 always 实现组合逻辑建模需要注意 latch 的产生。如下代码，if (G)不成立分支，Q 保持旧值，会有 latch 产生。

```
always @(*)
begin
if (G)
   Q = D;
end
```

3. 2 时序逻辑建模

时序逻辑与组合逻辑的不同之处在于触发器的产生。时序逻辑建模方式为过程赋值 always 语句。时序逻辑电路由组合电路和触发器组成，赋值方式可以是 blocking assignment 和 noblocking assignment，左值须为 reg 类型变量。推荐使用 noblocking assignment 赋值。

3.2.1 普通 DFF

```
reg [3:0] q;
always @(negedge clk)
q <= d;
```

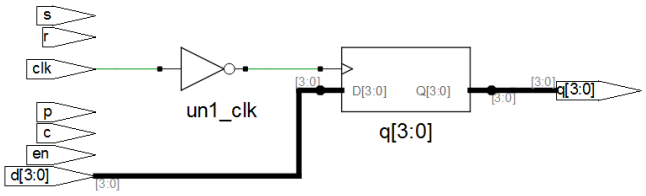

图 3-1 普通 DFF 关系图

3.2.2 异步复位 DFF

```
always @ (posedge clk or posedge r)
begin
  if (r)
     q <= 1'b0;
  else
     q <= d;
end
```

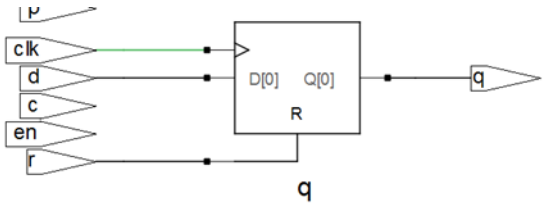

图 3-2 异步复位 DFF 关系图

3.2.3 异步置位 DFF

```
always @ (posedge clk or posedge s)
begin
  if (s)
    q <= 1'b1;
  else
    q <= d;
end
```

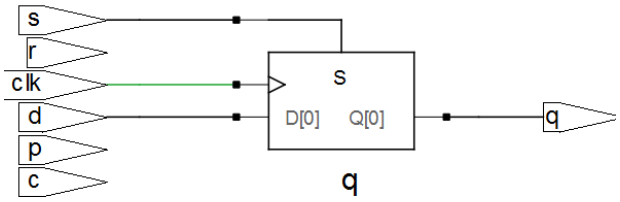

图 3-3 异步置位 DFF 关系图

3.2.4 异步复位置位 DFF

```
always @ (posedge clk or negedge r or posedge s)
  if (~r)
    q <= 1'b0;
  else if (s)
    q <= 1'b1;
  else
    q <= d;
```

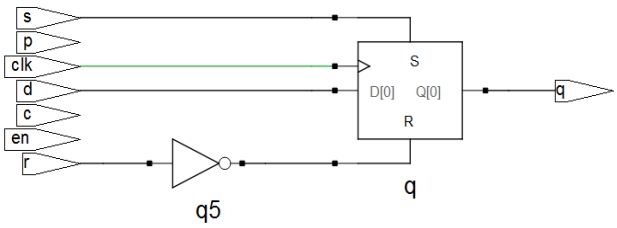

图 3-4 异步复位置位 DFF 关系图

3.2.5 有 enable 及异步置位复位 DFF

```
always @ (posedge clk or posedge r or posedge s)
begin
```

```
 if (r)
   q <= 1'b0;
 else if (s)
   q <= 1'b1;
 else if (en)
   q <= d;
end
```

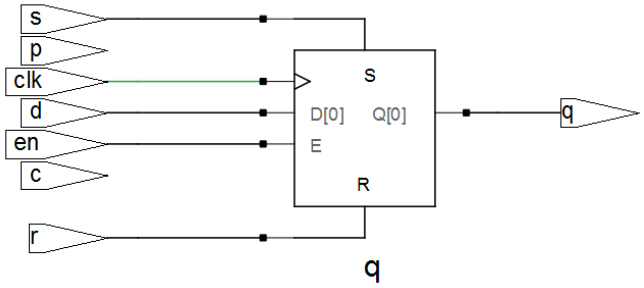

图 3-5 enable 异步置位复位 DFF

3.2.6 同步复位 DFF

```
always @ (posedge clk)
 if (r)
   q <= 1'b0;
 else
   q <= d;
```

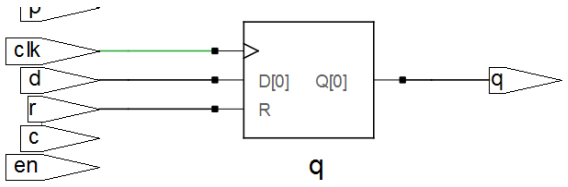

图 3-6 同步复位 DFF 关系图

3.2.7 同步置位 DFF

```
always @ (posedge clk)
 if (s)
   q <= 1'b1;
```

```
else
  q <= d;
```

3.2.8 同步复位置位 DFF

```
always @ (posedge clk)
 if (r)
   q <= 1'b0;
 else if (s)
   q <= 1'b1;
 else
   q <= d;
```

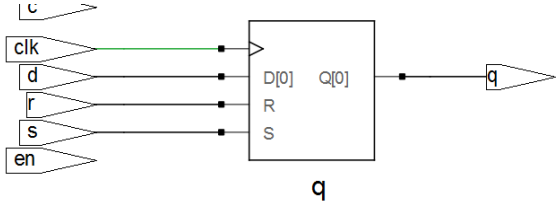

图 3-7 同步复位置位 DFF 关系图

3.2.9 有 enable 及同步复位置位 DFF

```
always @ (posedge clk)
 if (rst)
   q <= 1'b0;
 else if (set)
   q <= 1'b1;
 else if(en)
   q <= d;
```

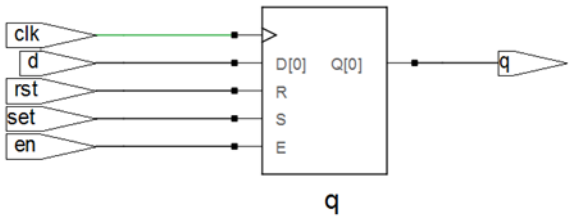

图 3-8 enable 同步复位置位 DFF 关系图

4 RAM 与 ROM 建模

4. 1 RAM 建模

ram 可以分为两类，分别为 Distributed Ram 和 Block Ram，通过 attribute syn_ramstyle 可以用于指定 ram 是否该 map 到 Block Ram 或者 Distributed Ram 或者纯 logic 的方式，关于 syn_ramstyle 的简单说明如下：

Decorated object: module, ram instance

Attribute subtype: string

Attribute values: block_ram, select_ram, registers, no_rw_check

Verilog syntax: object/*synthesis syn_ramstyle = "block_ram"*/;

根据读端口是否与时钟同步，ram 可以分为 async read 和 sync read 两大类，Distributed Ram 属于前者，Block Ram 属于后者。

其中 sync read ram 又可以根据读写端口同时有效的行为，分为 READ_BEFORE_WRITE, TRANSPARENT_WRITE, NORMAL_WRITE 三种模式。这些属性体现在 WRITE_MODE_A 和 WRITE_MODE_B 参数中。

从对应的 GTP 器件角度来看，Distributed Ram 对应的 GTP 器件如下：

Logos2: GTP_RAM32X1DP, GTP_RAM32X1SP, GTP_RAM32X2DP, GTP_RAM32X2SP, GTP_RAM64X1DP, GTP_RAM64X1SP, GTP_RAM128X1DP, GTP_RAM128X1SP, GTP_RAM256X1SP;

Logos: GTP_RAM32X1DP, GTP_RAM32X1SP;

Titan, Compact, PGL22G: GTP_RAM16X1DP, GTP_RAM16X1SP;

Block Ram 对应的 GTP 器件如下：

Logos2: GTP_DRM36K_E1, GTP_DRM18K_E1;

Compact: GTP_DRM9K_E1;

Logos, Titan, PGL22G: GTP_DRM18K, GTP_DRM9K;

4.1.1 RAM 数据初始化

对于 Ram 的数据进行初始化赋值，目前 ADS 综合工具只支持以下两种形式：

4.1.1.1 Two-dimensional array with data in initial statement

reg [7 : 0] mem [31 : 0]

initial begin

```
mem[0] = 8'b01111110;
mem[1] = 8'b10000011;
mem[2] = 8'b00000101;
mem[3] = 8'b00001001;
mem[4] = 8'b00001101;
……
mem[27] = 8'b00110011;
mem[28] = 8'b00100101;
mem[29] = 8'b01001110;
mem[30] = 8'b01110100;
mem[31] = 8'b11100101;
end
```

4.1.1.1 initial with $readmemb and $readmemh

using two-dimensional array with data in text file, The RAM is initialized from a text file using the system tasks $readmemb and $readmemh

```
reg [3 : 0] mem [0 : 7];
initial $readmemb("mem.data", mem);
// Example of content "mem.data" file
1011    // addr=0
1000    // addr=1
0000    // addr=2
1000    // addr=3
0010    // addr=4
0101    // addr=5
1111    // addr=6
1001    // addr=7
```

4.1.2 Distributed Ram

SINGLE-PORT

```
always @(posedge clk)
begin
```

```
    if (en)
        begin
        if (we)
            RAM[addr] <= di;
        end
end
assign do = RAM[addr];
```

map 到 GTP 器件分别如下所示：

Logos2: GTP_RAM32X1SP, GTP_RAM32X2SP, GTP_RAM64X1SP, GTP_RAM128X1SP ,GTP_RAM256X1SP;

Logos: GTP_RAM32X1SP;

Titan, Compact, PGL22G: GTP_RAM16X1SP;

4.1.2.1 DUAL_PORT

```
always @(posedge clk)
begin
        if (en)
        begin
            if (we)
                RAM[write_addr] <= di;
        end
end
assign do = RAM[read_addr]
```

map 到 GTP 器件分别如下所示：

Logos2: GTP_RAM32X1DP, GTP_RAM32X2DP, GTP_RAM64X1DP, GTP_RAM128X1DP;

Logos: GTP_RAM32X1DP;

Titan, Compact, PGL22G: GTP_RAM16X1DP;

4.1.3 Block Ram

4.1.3.1 SINGLE_PORT

single port 的 block ram 的推荐建模方式如下四种：

- WRITE_MODE_A = "READ BEFORE WRITE", map 到 DRM，具体的 DRM 的 size 根据架构不同而有变化。

推荐的代码风格：

```
always @(posedge clk)
begin
    if (en)
    begin
        if (we)
            RAM[addr] <= di;
        do <= RAM[addr];
    end
end
```

- WRITE_MODE_A = "TRANSPARENT WRITE" style1：

```
always @(posedge clk)
begin
    if (en)
    begin
        if (we)
        begin
            RAM[addr] <= di;
            do <= di;
        end
        else
            do <= RAM[addr];
    end
end
```

- WRITE_MODE_A = "TRANSPARENT WRITE" style2：

```
always @(posedge clk)
```

```
begin
    if (en)
    begin
        if (we)
            RAM[addr] <= di;
        read_addr <= addr;
    end
end
assign do = RAM[read_addr];
```

- WRITE_MODE_A = "NORMAL WRITE",map 到 DRM，具体的 DRM 的 size 根据架构不同而有变化

```
always @(posedge clk)
begin
    if (en)
    begin
        if (we)
            RAM[addr] <= di;
        else
            do <= RAM[addr];
    end
end
```

4.1.3.2 SIMPLE-DUAL-PORT

简单双端口模式，一个端口写数据，一个端口读数据，读写之间不构成任何模式，当两个端口读写数据同时有效且地址相同时，为非法操作，此时无法保证读出的数据是新写入的数据还是以前的旧数据，需要通过用户逻辑加以规避。当 map 到 DRM，具体的 DRM 的 size 根据架构不同而有变化。

```
always @(posedge clk)
begin
    if (en)
    begin
```

```
        if (we)
            RAM[addra] <= di;
        dob <= RAM[addrb];
    end
end
```

4.1.3.3 TURE DUAl PORT

由两个独立的 single port ram 合并而成，map 为 DRM，具体的 DRM 的 size 根据架构不同而有变化，可参考上述提到的 Block Ram 对应的 GTP 器件，值得注意的是，若同时通过两个端口对同一地址进行读写操作会引起冲突，同地址写写，读写冲突均属于非法操作，其当前操作不能正常完成，需要通过用户逻辑加以规避。

4. 2 ROM 建模

rom,同 ram 类似，也可以分为两类，分别为 Distributed Rom 和 Block Rom，通过 attribute syn_romstyle 可以用于指定 rom 是否该 map 到 Block Rom 或者 Distributed Rom 或者纯 logic 的方式，关于 syn_romstyle 的简单说明如下：

Decorated object: module, rom instance

Attribute subtype: string

Attribute values: block_rom, select_rom, logic

Verilog syntax: object/*synthesis syn_romstyle = "block_rom"*/;

根据读端口是否与时钟同步，rom 可以分为 async read 和 sync read 两大类，Distributed Rom 属于前者，Block Rom 属于后者。

从对应的 GTP 器件角度来看，Distributed Rom 对应的 GTP 器件如下：

GTP_ROM32X1, GTP_ROM32X2, GTP_ROM64X1, GTP_ROM128X1, GTP_ROM256X1;

Block Rom 对应的 GTP 器件如下：

Logos2: GTP_DRM36K_E1, GTP_DRM18K_E1;

Compact: GTP_DRM9K_E1;

Logos, Titan, PGL22G: GTP_DRM18K, GTP_DRM9K;

4.2.1 Distributed Rom

4.2.1.1 One-dimensional array with data in case statement

```
module rom_case(addr, dataout);
```

```
input [4 : 0] addr;
output [7 : 0] dataout;
reg [7 : 0] dataout;
always @(addr)begin
    case (addr)
        5'b00000: dataout = 8'b10000011;
        5'b00001: dataout = 8'b00000101;
        5'b00010: dataout = 8'b00001001;
        5'b00011: dataout = 8'b00001101;
        5'b00100: dataout = 8'b00010001;
        5'b00101: dataout = 8'b00011001;
        5'b00110: dataout = 8'b00100001;
        5'b00111: dataout = 8'b10110100;
        5'b01000: dataout = 8'b11000000;
        5'b01001: dataout = 8'b10110001;
        5'b01010: dataout = 8'b00110101;
        5'b01011: dataout = 8'b01110010;
        5'b01100: dataout = 8'b11100011;
        5'b01101: dataout = 8'b00111111;
        5'b01110: dataout = 8'b01010101;
        5'b01111: dataout = 8'b00110100;
        5'b10000: dataout = 8'b10110000;
        5'b10001: dataout = 8'b00010001;
        5'b10010: dataout = 8'b10110011;
        5'b10011: dataout = 8'b00101011;
        5'b10100: dataout = 8'b11101110;
        5'b10101: dataout = 8'b01110111;
        5'b10110: dataout = 8'b01110101;
        5'b10111: dataout = 8'b01000011;
        5'b11000: dataout = 8'b01011100;
```

```
                5'b11001: dataout = 8'b00010100;
                5'b11010: dataout = 8'b00110011;
                5'b11011: dataout = 8'b00100101;
                5'b11100: dataout = 8'b01001110;
                5'b11101: dataout = 8'b01110100;
                5'b11110: dataout = 8'b11100101;
                5'b11111: dataout = 8'b01111110;
                default: dataout = 8'b00000000;
            endcase
end
endmodule
```

4.2.1.2 Two-dimensional array with data in initial statement

```
module rom_case(addr, dataout);
    input [4 : 0] addr;
    output [7 : 0] dataout;
    reg [7 : 0] rom [31 : 0];
    initial begin
     rom[0] = 8'b01111110;
     rom[1] = 8'b10000011;
     rom[2] = 8'b00000101;
     rom[3] = 8'b00001001;
     rom[4] = 8'b00001101;
    rom[5] = 8'b00010001;
    rom[6] = 8'b00011001;
    rom[7] = 8'b00100001;
    rom[8] = 8'b10110100;
    rom[9] = 8'b11000000;
    rom[10] = 8'b10110001;
     rom[11] = 8'b01110010;
    rom[12] = 8'b11100011;
```

```
        rom[13] = 8'b11100011;
        rom[14] = 8'b00111111;
        rom[15] = 8'b01010101;
        rom[16] = 8'b00110100;
        rom[17] = 8'b10110000;
        rom[18] = 8'b00010001;
        rom[19] = 8'b10110011;
        rom[20] = 8'b00101011;
        rom[21] = 8'b11101110;
        rom[22] = 8'b01110111;
        rom[23] = 8'b01110101;
        rom[24] = 8'b01000011;
        rom[25] = 8'b01011100;
        rom[26] = 8'b00010100;
        rom[27] = 8'b00110011;
        rom[28] = 8'b00100101;
        rom[29] = 8'b01001110;
        rom[30] = 8'b01110100;
        rom[31] = 8'b11100101;
        end
        assign dataout = rom[addr];
endmodule
```

4.2.1.3 initial with $readmemb and $readmemh

```
module rom_case(addr, Z);
    input [2 : 0] addr;
    output [3 : 0] Z;
    reg [3 : 0] rom [0 : 7];
    initial $readmemb("rom.data", rom);
    assign Z = rom[addr];
endmodule
```

```
// Example of content "rom.data" file
    1011    // addr=0
    1000    // addr=1
    0000    // addr=2
    1000    // addr=3
    0010    // addr=4
    0101    // addr=5
    1111    // addr=6
    1001    // addr=7
```

4.2.2 Block Rom

4.2.2.1 One-dimensional array with data in case statement

```
module rom_case(clock, addr, dataout);
     input clock;
    input [4 : 0] addr;
    output [7 : 0] dataout;
    reg [7 : 0] dataout;
     reg [4 : 0] read_addr;

    always @(read_addr)begin
        case (read_addr)
            5'b00000: dataout <= 8'b10000011;
            5'b00001: dataout <= 8'b00000101;
            5'b00010: dataout <= 8'b00001001;
            5'b00011: dataout <= 8'b00001101;
            5'b00100: dataout <= 8'b00010001;
            5'b00101: dataout <= 8'b00011001;
            5'b00110: dataout <= 8'b00100001;
            5'b00111: dataout <= 8'b10110100;
            5'b01000: dataout <= 8'b11000000;
```

```
            5'b01001: dataout <= 8'b10110001;
            5'b01010: dataout <= 8'b00110101;
            5'b01011: dataout <= 8'b01110010;
            5'b01100: dataout <= 8'b11100011;
            5'b01101: dataout <= 8'b00111111;
            5'b01110: dataout <= 8'b01010101;
            5'b01111: dataout <= 8'b00110100;
            5'b10000: dataout <= 8'b10110000;
            5'b10001: dataout <= 8'b00010001;
            5'b10010: dataout <= 8'b10110011;
            5'b10011: dataout <= 8'b00101011;
            5'b10100: dataout <= 8'b11101110;
            5'b10101: dataout <= 8'b01110111;
            5'b10110: dataout <= 8'b01110101;
            5'b10111: dataout <= 8'b01000011;
            5'b11000: dataout <= 8'b01011100;
            5'b11001: dataout <= 8'b00010100;
            5'b11010: dataout <= 8'b00110011;
            5'b11011: dataout <= 8'b00100101;
            5'b11100: dataout <= 8'b01001110;
            5'b11101: dataout <= 8'b01110100;
            5'b11110: dataout <= 8'b11100101;
            5'b11111: dataout <= 8'b01111110;
            default: dataout <= 8'b00000000;
        endcase
    end

    always @(posedge clock)
        read_addr <= addr;
endmodule
```

4.2.2.2 Two-dimensional array with data in initial statement

```
module rom_case(clock, addr, dataout);
    input clock;
   input [4 : 0] addr;
   output [7 : 0] dataout;
    reg [7 : 0] dataout;
   reg [7 : 0] rom [31 : 0];

   initial begin
    rom[0] = 8'b01111110;
    rom[1] = 8'b10000011;
    rom[2] = 8'b00000101;
    rom[3] = 8'b00001001;
    rom[4] = 8'b00001101;
   rom[5] = 8'b00010001;
   rom[6] = 8'b00011001;
   rom[7] = 8'b00100001;
   rom[8] = 8'b10110100;
   rom[9] = 8'b11000000;
   rom[10] = 8'b10110001;
    rom[11] = 8'b01110010;
   rom[12] = 8'b11100011;
    rom[13] = 8'b11100011;
    rom[14] = 8'b00111111;
    rom[15] = 8'b01010101;
    rom[16] = 8'b00110100;
    rom[17] = 8'b10110000;
    rom[18] = 8'b00010001;
    rom[19] = 8'b10110011;
    rom[20] = 8'b00101011;
```

```
        rom[21] = 8'b11101110;
        rom[22] = 8'b01110111;
        rom[23] = 8'b01110101;
        rom[24] = 8'b01000011;
        rom[25] = 8'b01011100;
        rom[26] = 8'b00010100;
        rom[27] = 8'b00110011;
        rom[28] = 8'b00100101;
        rom[29] = 8'b01001110;
        rom[30] = 8'b01110100;
        rom[31] = 8'b11100101;
        end

        always @(posedge clock)
                dataout <= rom[addr];
endmodule
```

4.2.2.3 initial with $readmemb and $readmemh

```
module rom_case(clock, addr, Z);
        input clock;
        input [2 : 0] addr;
        output [3 : 0] Z;
        reg [3 : 0] rom [0 : 7];
        initial $readmemb("rom.data", rom);
        always @(posedge clock)
                dataout <= rom[addr];
endmodule

        // Example of content "rom.data" file
                1011      // addr=0
                1000      // addr=1
```

```
0000    // addr=2
1000    // addr=3
0010    // addr=4
0101    // addr=5
1111    // addr=6
1001    // addr=7
```

5 FSM 建模

FSM（Finite State Machine）是时序电路重要的设计方法和电路结构。FSM 是由一组触发器和组合逻辑构成的电路，触发器每一个数据组合就称作一个状态，FSM 根据不同的状态输出和状态转换来有条不紊的控制时序电路。图 5-1 是 FSM 的基本结构框图。其中 X 代表输入信号，Y 代表输出信号，Q 代表存储电路的当前状态，Z 代表存储电路的输入，也就是下一个状态。

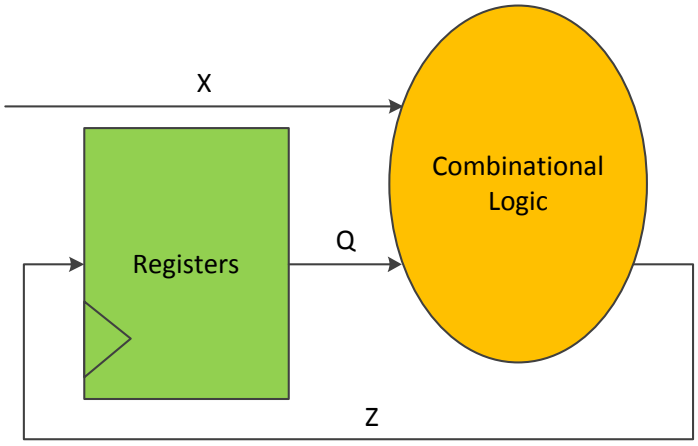

图 5-1 FSM 结构框图

5. 1 推荐的 FSM 建模风格

FSM 的描述通常包括状态复位，状态转换和输出逻辑描述三个主要方面，根据描述这几方面所采用的不同结构，推荐三类典型的描述结构：single process，two processes，three processes。下面对这三类典型的建模风格逐一举例说明。

5.1.1 Single Process FSM 建模

顾名思义，此类建模风格将 FSM 的复位，状态转换和输出逻辑都描述在一个 process 里。举例如下：

```
module fsm(clk, reset, X, Y);

input clk, reset, X;

output reg Y;

reg [1:0] state;

parameter state_1 = 2'b00;

parameter state_2 = 2'b01;
```

```
parameter state_3 = 2'b10;
parameter state_4 = 2'b11;

always@(posedge clk or posedge reset) begin
    if (reset) begin
        state = state_1;
        Y = 1'b1;
    end
    else
        case (state)
            state_1: begin
                if (X == 1'b1)      state = state_2;
                else    state = state_3;
                Y = 1'b1;
            end
            state_2: begin
                state = state_4;
                Y = 1'b1;
            end
            state_3: begin
                state = state_4;
                Y = 1'b0;
            end
            state_4: begin
                state = state_1;
                Y = 1'b1;
            end
        endcase
end
endmodule
```

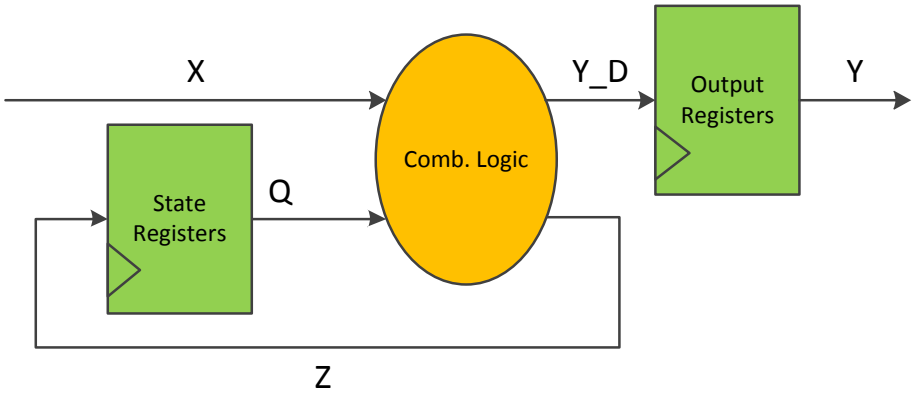

图 5-2 single process coding style 综合后 FSM 结构框图

5.1.2 Two Processes FSM 建模

此类建模风格由两个 process 来描述。其中一个 process 用于描述 FSM 的复位和状态转换，另一个 process 用于描述 FSM 的输出逻辑。举例如下：

```
module fsm(clk, reset, X, Y);
input clk, reset, X;
output reg Y;
reg [1:0] state;
parameter state_1 = 2'b00;
parameter state_2 = 2'b01;
parameter state_3 = 2'b10;
parameter state_4 = 2'b11;

always@(posedge clk or posedge reset)
    if (reset)
        state = state_1;
    else
        case (state)
            state_1:
                if (X == 1'b1)      state = state_2;
                else    state = state_3;
            state_2:
```

```
                state = state_4;
            state_3:
                state = state_4;
            state_4:
                state = state_1;
        endcase


always@(state)
if (reset)
    Y = 1'b1;
else
    case(state)
        state_1: Y = 1'b1;
        state_2: Y = 1'b1;
        state_3: Y = 1'b0;
        state_4: Y = 1'b1;
    endcase
endmodule
```

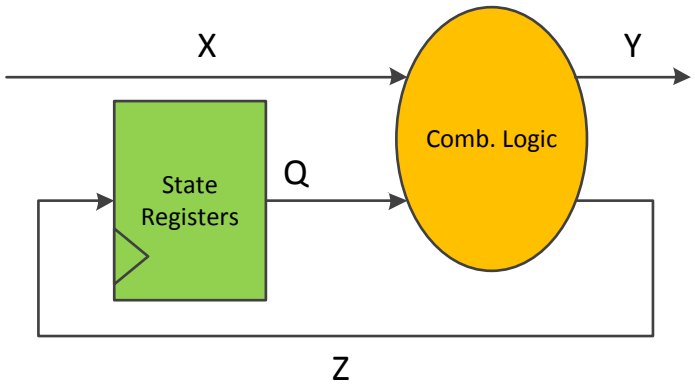

图 5-3 two processes coding style 综合后 FSM 结构框图

5.1.3 Three Processes FSM 建模

此类建模风格由三个 process 来描述。其中第一个 process 用于描述 FSM 的复位和下一状态跳转；第二个 process 用于描述状态转换逻辑，最后一个 process 用于描述 FSM 的输出逻辑。举例如下：

```
module fsm(clk, reset, X, Y);
input clk, reset, X;
output reg Y;
reg [1:0] state;
reg [1:0] next_state;
parameter state_1 = 2'b00;
parameter state_2 = 2'b01;
parameter state_3 = 2'b10;
parameter state_4 = 2'b11;

always@(posedge clk or posedge reset)
    if (reset)
        state = state_1;
    else
        state = next_state;

always@(state or X)
    case (state)
        state_1:
            if (X == 1'b1)      next_state = state_2;
            else    next_state = state_3;
        state_2:
            next_state = state_4;
        state_3:
            next_state = state_4;
        state_4:
            next_state = state_1;
    endcase

always@(state)
```

```
if (reset)
    Y = 1'b1;
else
    case(state)
        state_1: Y = 1'b1;
        state_2: Y = 1'b1;
        state_3: Y = 1'b0;
        state_4: Y = 1'b1;
    endcase
endmodule
```

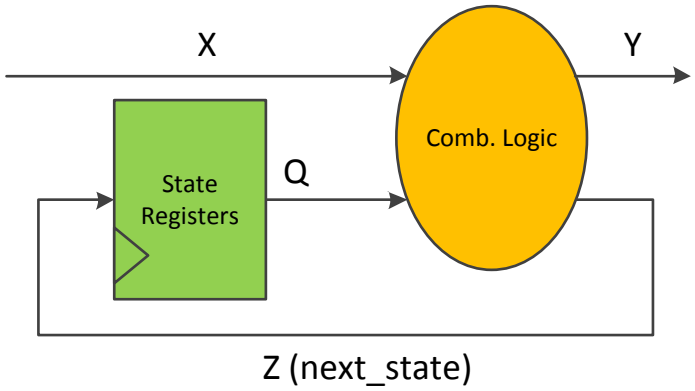

图 5-4 Three processes coding style 综合后 FSM 结构框图

5.2 为 FSM 指定复位描述

为 FSM 指定同步或异步复位信号，以保证电路在上电时令 FSM 进入指定的合法状态，或在运行时根据需要复位状态。

5.3 使用 parameter 指定不同状态

建议通过定义 parameter 的形式指定 FSM 的合法状态，这样可以增加代码的可读性。例如：

```
parameter idle = 3'b000;
parameter ready = 3'b001;
parameter read = 3'b010;
parameter write = 3'b011;
parameter wait = 3'b100;
```

…

5. 4 使用 case 语句描述 FSM

推荐在 always 结构中通过 case 语句描述 FSM。描述的主要特征为 case 语句的控制表达式即为状态变量，各个分支表达式即为不同的合法状态。在不同的分支中根据当前状态和输入条件描述下一状态值。

5.5 根据需要指定 FSM 编码方式

根据不同的优化目标可选用不同形式的状态编码。Verilog HDL 代码中可以通过

/*synthesis syn_encoding = "value"*/

的形式指定编码类型。其中 value 可指定以下几种值。

5.5.1 onehot

每个状态对应编码的一个 bit，不同的状态之间跳转只有两个寄存器需要动作。通常可减小组合逻辑电路的规模，占用的状态寄存器数目和状态数相等。由于 FPGA 器件内部含有丰富的寄存器资源，因此该编码是较为适合 FPGA 器件的一种编码。另外，由于其动作特点此种编码的可靠性也是相对较高的。

5.5.2 gray

相邻编码仅有一个 bit 不同，状态跳转时寄存器动作较小，有助于提高系统稳定性。

5.5.3 sequential

自然二进制编码，状态寄存器数目最少。

5.5.4 original

保持源文件描述的原始编码。

5.5.5 safe

safe 编码的功能是，一旦 FSM 进入任意非法状态，可以通过 safe logic 第一时间将 FSM 跳转回合法状态。旨在增强设计的稳定性，可以与前述几种编码联合使用，例如：

/*synthesis syn_encoding = "safe, onehot"*/

safe 编码有些情况下开销较大，比如 FSM 状态数较多，又将 onehot encoding 与 safe encoding 联合使用，那么 safe logic 就比较庞大。

6 附录

6.1 Module and Macromodule Declaration

The module/endmodule block is the basic compilation unit in Verilog, the module begin with module(macromodule) and terminated with endmodule. The basic syntax form as follows:

module| macromodule test(A, B, Y);

input A, B;

output Y;

assign Y = A & B;

endmodule

6.2 Port Declaration

Port 有三种类型，分别为 input、output 和 inout。

6.3 Parameter Declaration

parameter 分为 module parameter，local parameter 和 specify parameter，对应的 keywords 为 parameter，localparam 和 specparam。

parameter 的 declaration 格式为：

parameter|localparam|specparam [data type] param_assignments

6.4 Reg Type Declaration

6.4.1 Supported Reg Type Declaration

ADS supported reg declaration keywords are reg、integer、time。

reg 没有 data type，默认为 1 bit wide，如果超过 1 bit，则需要 range declaration 设置 reg 的位宽。

integer 的默认位宽为 32 bit，不允许有 range declaration。

time 的默认位宽为 64 bit，不允许有 range declaration。

6.4.2 Unsupported Reg Type Declaration

Unsupported reg declaration keywords are event、real、realtime。

6.5 Net Declaration

6.5.1 Supported Net Type Declaration

ADS supported 的 net type 为 wire, tri, tri**0**, tri**1**, triand, trior, wand, wor, supply**0** 和 supply**1**(允许有初值)。

6.5.2 Unsupported Net Type Declaration

ADS unsupported 的 net type 为 trireg。

6.6 Continuous Assign

Continuous assignment 完成对 net 变量的赋值。continuous assignment 包含左值和右值两个部分，左值的类型如下：

Net (vector or scalar)

Constant bit-select of a vector net

Constant part-select of a vector net

Constant indexed part-select of a vector net

Concatenation or nested concatenation of any of the above left hand side

6.7 Instantiation

6.7.1 Module Instantiation

6.7.1.1 Supported module instantiation

Supported module instantiation syntax form as follows：

module_identifier [parameter_value_assignment] module_instance {,module_instance }

parameter_value_assignment 完成 instantiated module 中 parameter 值的设置。parameter 值除了使用上述方式设置之外，还可以使用 defparam。语法结构如下：

defparam inst_name.parameter_name = value;

6.7.1.2 Unsupported Module Instantiation

User-defined primitives(UDPs)，UDPs begin with keyword primitive and terminated with endprimitive, the syntax form as follows：

primitive

…

endprimitive

6.7.2 Gate instantiation

6.7.2.1 Supported gate instantiation

gate instantiation syntax as follows：

gatetypekeyword [instname] (portlist);

supported gate keywords：

enable_gatetype	n_input_gatetype	n_output_gatetype
bufif0	and	buf
bufif1	nand	not
notif0	or	
notif1	nor	
	xor	
	xnor	

表 6-1 supported gate keywords

6.7.2.2 Unsupported gate instantiation

unsupported gate keywords：

pass_en_switchtype	mos_switchtype	cmos_switchtype	pass_switchtype
tranif0	nmos	cmos	tran
tranif1	pmos	rcmos	rtran
rtranif0	rnmos		

rtranif1	rpmos		

表 6-2 Unsupported gate instantiation

6.8 Always Construct

The always construct repeats continuously throughout the duration of the simulation, syntax as follows：

always statement

6.9 Generate Region

Generate region begin with keyword generate and terminated with endgenerate. Syntax as follows：

generate { module_or_generate_item } endgenerate

6.10 Generate Block

Supported

6.11 Loop Generate Construct

Supported

Syntax form as follows：

for (genvar_initialization ; genvar_expression ; genvar_iteration)

generate_block

6.12 Conditional Generate Construct

condition generate construct 包含 generate if 和 generate case。

6.12.1 Generate If Construct

Supported

Syntax form as follows：

if (constant_expression)

generate_block_or_null

[else generate_block_or_null]

6.12.2 Generate Case Construct

Supported

Syntax as follows：

case (constant_expression)

case_generate_item { case_generate_item }

endcase

6.13 Function Construct

Supported

6.14 Task Construct

Supported

6.15 Operators

6.15.1 Binary Operators

+, -, *, /, %, **, <, >, <=, >=, ==, !=, ===, !==, &&, ||, &, |, ~|, ^~, ~^, ^, <<, >>, <<<, >>>

上述运算符的相关表述如下：

Operators	Usage	Function
+	a + b	a plus b
-	a - b	a minus b
*	a * b	a multiplied by b
/	a / b	a divided by b
%	a % b	a modulus b(ADS unsupported)
**	a ** b	a to the power of b
<	a < b	a less than b

>	a > b	a greater than b
<=	a <= b	a less than or equal to b
>=	a >= b	a greater than or equal to b
==	a == b	a logical equality to b
!=	a != b	a logical inequality to b
===	a === b	a case equality to b
!==	a !==b	a case inequality to b
&&	a && b	a logical and b
||	a || b	a logical or b
&	a & b	a bitwise and b
|	a | b	a bitwise or b
~|	a ~| b	a bitwise nor b
^~,~^	a ^~ b, a ~^ b	a bitwise equivalence b
^	a ^ b	a bitwise exclusive or b
<<	a << b	logical left shift
>>	a >> b	logical right shift
<<<	a <<< b	arithmetic left shift
>>>	a >>> b	arithmetic right shift

表 6-3 Binary Operators

6.15.2 Unary Operators

+, -, !, ~, &, |, ^, ~&, ~|, ~^

上述 unary operators 运算符的相关表述如下：

Operators	Usage	Function
+	+a	Arithmetic plus
-	-a	Arithmetic minus
!	!a	Logical negation
~	~a	Bitwise negation
&	&a	Reduction and
|	|a	Reduction or
^	^a	Reduction xor
~&	~&a	Reduction nand
~|	~|a	Reduction nor
~^	~^a	Reduction xnor

表 6-4 unary operators

6.15.3 Other

? : , { }, {{ }}

Operators	Usage	Function
? :	sel ? a : b	Conditional
{ }	{a, b}	Concatenation
{{ }}	{a{b}}	replication

表 6-5 other operators

6.15.4 Unsupported Operators

- /运算符中除数必须为 2 的幂次方(2**n)。

- % ADS unsupported

6.16 Statements

6.16.1 Supported Statements

Statements occur within procedures such as always, initial, task and function, the statements type as follows：

- Blocking procedural assignments = and Nonblocking procedural assignments <=

The left hand value must be register. Do not use = and <= for the same register

- Seq block

  begin/end block groups multiple statements into always block, example as follows：

  ```
  module test (in1, in2, out1, out2);
  input in1, in2;
  output reg out1, out2;
  always@(in1, in2)
  begin
      out1 = in1 & in2;
      out2 = in1 | in2;
  end
  endmodule
  ```

- Task enable
- If statement

  If statement is conditional statement, the keywords include if、else and else if。Syntax of if statement as follows：

  ```
  if (condition1)
      statement1;
  [ else if (condition2) statement2;]
  [else statement3;]
  ```

- Case statement

  A case statement include begin keywords case, casex or casez, terminal keyword endcase. casex means that 'x' or 'z' value in the case expression is treated as don't care. casez means that 'z'

value in the case expression is treated as don't care. If there is default keyword, its statement will be selected when none of the select expressions is valid.

- For statement

for statement is one of loop statements , it is used to modify blocks of procedural statements. The first assignment is executed initially and then the expression is evaluated repeatedly based on condition value.

- Event control statement

@identifier

@(*)

@*

@(identifier, identifier,…)

@(posedge identifier, …)

@(negedge identifier, …)

- Delay control statement

delay control statement 中的 delay 实现方式为#delay_value or #(mintypmax[,mintypmax[, mintypmax]]），在 ADS 综合中，delay control 中的 delay is ignored。

- While statement

It is one of loop statements, syntax of while statement as follows.

while (expression) statement

It executes the given statement until the expression becomes true.

- Disable statement

The disable statement provides the ability to terminate the activity associated with concurrently active. It can be used within blocks and tasks to disable the particular block or task containing the disable statement. syntax form of disable statement as follows：

disable block_id/task_id

- System task

$**readmemb and **$readmemh in initial is supported

6.16.2 Unsupported Statements

- par block

fork/join block

- procedural statements

  the keyword of procedural statements is assign/desassign force/release

- wait statement

  It is one of loop statements, syntax from of wait statement as follows：

  wait (expression) statement

- forever statement

  It is one of loop statements, repeats the ensuing statement continuously. syntax from of forever statement as follows：

  forever statement

- repeat statement

  Executes a given statement a fixed number of times, the number of executions is defined by the expression following the repeat keyword, syntax from of repeat statement as follows：

  repeat (expression) statement

- event trigger

  syntax from of event trigger as follows：

  ->event_identifier

6.17 Expressions

6.17.1 Supported Expressions

- Number

  Number types of ADS supported are integer number, string number and based number. The based number form has binary number, octal number, decimal number and hexadecimal number.

- Name

  simple name and escape name, the escape name syntax is \simple_name

- System function

  A name following the dollar($) is interpreted as a system function. Only **$**signed and **$**unsigned system function is supported.

6.17.2 Unsupported Expressions

System function expect $signed and $unsigned

6.17.3 Data Type

6.17.4 signed/unsigned

Supported

6.17.5 range

Supported

Range in reg/net/port declaration specifies the variable is scalar or vector, the syntax of range as following.

[expr : expr], [expr +: expr], [expr -: expr]

可以使用 range 对变量进行 bit select 和 partial select, bit select 为 vecotorName[expr],partial select 为 vecotorName[expr:expr]、vecotorName[expr+:expr]、 vecotorName[expr-:expr]。

6.18 Compiler directives

All Verilog compiler directives are preceded by the (`) character。

The supported compiler directives as follows：

- `include

The file inclusion (`include) compiler directive is used to insert the entire contents of a source file in another file during compilation. Syntax as follows:

`include "filename"

- `ifdef, `else , `elsif , `ifndef, `endif
- `undef

Removes the definition of a previously defined macro.

- `define

The directive `define creates a macro for text substitution. This directive can be used both inside and outside module definitions. Example as follows.

`define MIN 10

- `default_nettype

The directive `default_nettype controls the net type created for implicit net declarations. It can be used only outside of module definitions. Option of `default_nettype is wire, tri, tri0 , tri1, wand, triand , wor , trior, trireg, uwire, none. Example as follows.

```
`default_nettype none
```

7 Ignored Verilog Language Constructs

7.1 Initial Construct

Ignored

7.2 Delay and Delay Control

Ignored

7.3 Strengths

Ignored

drive_strength ::= (strength0 , strength1) | (strength1 , strength0) | (strength0 , highz1) |
(strength1 , highz0) | (highz1 , strength0) | (highz0 , strength1)

charge_strength ::= (small) | (medium) | (large)

7.4 Specify Block

Ignored

7.5 Config Declaration

Ignored

7.6 SystemTaskEnable

Ignored expected for **$**readmemb and **$**readmemh in initial

7.7 Compiler Directives

`line, `celldefine, `endcelldefine, `resetall, `timescale, `unconnected_drive, `nounconnected_drive

免责声明

版权声明

免责声明

1、本文档仅提供阶段性信息，所含内容可根据产品的实际情况随时更新，恕不另行通知。如因本文档使用不当造成的直接或间接损失，本公司不承担任何法律责任。

2、本文档按现状提供，不负任何担保责任，包括对适销性、适用于特定用途或非侵权性的任何担保，和任何提案、规格或样品在他处提到的任何担保。本文档在此未以禁止反言或其他方式授予任何知识产权使用许可，不管是明示许可还是暗示许可。

公司保留任何时候在不事先声明的情况下对公司系列产品相关文档的修改权利。