



GowinSynthesis 用户指南

SUG550-1.8,2024-08-09

版权所有 © 2024 广东高云半导体科技股份有限公司

GOWIN高云、Gowin、GowinSynthesis、云源以及高云均为广东高云半导体科技股份有限公司注册商标，本手册中提到的其他任何商标，其所有权利属其拥有者所有。未经本公司书面许可，任何单位和个人都不得擅自摘抄、复制、翻译本档内容的部分或全部，并不得以任何形式传播。

免责声明

本档并未授予任何知识产权的许可，并未以明示或暗示，或以禁止反言或其它方式授予任何知识产权许可。除高云半导体在其产品的销售条款和条件中声明的责任之外，高云半导体概不承担任何法律或非法律责任。高云半导体对高云半导体产品的销售和 / 或使用不作任何明示或暗示的担保，包括对产品的特定用途适用性、适销性或对任何专利权、版权或其它知识产权的侵权责任等，均不作担保。高云半导体对文档中包含的文字、图片及其它内容的准确性和完整性不承担任何法律或非法律责任，高云半导体保留修改文档中任何内容的权利，恕不另行通知。高云半导体不承诺对这些文档进行适时的更新。

版本信息

日期	版本	说明
2019/08/02	1.0	初始版本。
2019/12/05	1.1	增加综合前后对象命名规则说明（适用于高云半导体云源软件 1.9.3 及以后版本）。
2020/03/03	1.2	增加支持 VHDL 语法的设计（适用于高云半导体云源软件 1.9.5 及以后版本）。
2020/05/29	1.3	<ul style="list-style-type: none">● 修改综合命名规则；● 增加 <code>syn_srlstyle</code>、<code>syn_noprune</code> 属性。
2020/09/14	1.4	增加 <code>black_box_pad_pin</code> 属性。
2021/10/28	1.5	<ul style="list-style-type: none">● 增加 <code>parallel_case</code>、<code>syn_black_box</code> 属性；● 修改第 6 章 Report 用户文档。
2023/06/30	1.6	更新第 5 章综合约束支持。
2024/05/09	1.7	<ul style="list-style-type: none">● 更新第 4 章硬件描述语言代码支持描述；● 更新第 5 章综合约束支持描述；● 更新第 6 章 Report 用户文档中的截图。
2024/08/09	1.8	<ul style="list-style-type: none">● 新增综合属性约束对象数量说明；● 更新图 6-4 Timing。

目录

目录	i
图目录	iii
表目录	iv
1 关于本手册	1
1.1 手册内容	1
1.2 相关文档	1
1.3 术语、缩略语	1
1.4 技术支持与反馈	2
2 概述	3
3 GowinSynthesis 的使用方法介绍	4
3.1 GowinSynthesis 输入输出	4
3.2 使用 GowinSynthesis 进行综合	4
3.3 综合前后的对象命名规则	4
3.3.1 综合后网表文件命名	5
3.3.2 综合后网表 module 命名	5
3.3.3 综合后网表 Instance 命名	5
3.3.4 综合后网表连线命名	5
4 硬件描述语言代码支持	6
4.1 寄存器的硬件描述语言代码支持	6
4.1.1 寄存器特征介绍	6
4.1.2 寄存器相关的约束	6
4.1.3 寄存器代码示例	6
4.2 RAM 的硬件描述语言代码支持	12
4.2.1 RAM 置换基本功能介绍	12
4.2.2 RAM 特征介绍	12
4.2.3 RAM 推导相关的约束	12
4.2.4 RAM 推导代码示例	13
4.3 DSP 的硬件描述语言代码支持	19
4.3.1 DSP 推导的基本功能介绍	19

4.3.2 DSP 特征介绍	20
4.3.3 DSP 相关的约束	20
4.3.4 DSP 推导代码示例	20
4.4 有限状态机的综合实现规则	27
4.4.1 有限状态机的综合规则	27
4.4.2 有限状态机代码示例	27
5 综合约束支持	30
5.1 black_box_pad_pin	31
5.2 full_case	33
5.3 parallel_case	34
5.4 syn_black_box	35
5.5 syn_dspstyle	37
5.6 syn_encoding	38
5.7 syn_insert_pad	39
5.8 syn_keep	40
5.9 syn_looplmit	41
5.10 syn_maxfan	41
5.11 syn_netlist_hierarchy	43
5.12 syn_noprune	44
5.13 syn_preserve	45
5.14 syn_probe	47
5.15 syn_ramstyle	48
5.16 syn_romstyle	49
5.17 syn_srlstyle	50
5.18 syn_tlvs_io/syn_elvds_io	52
5.19 translate_off/translate_on	53
6 Report 用户文档	55
6.1 Synthesis Message	55
6.2 Synthesis Details	55
6.3 Resource	56
6.4 Timing	56

图目录

图 4-1 示例 1 同步复位时钟触发器示意图	7
图 4-2 示例 2 同步置位且带有使能功能的触发器示意图	8
图 4-3 示例 3 异步复位且带有时钟使能功能的触发器示意图	9
图 4-4 示例 4 带有复位和高电平使能功能的锁存器示意图	9
图 4-5 示例 5 同步复位时钟触发器及逻辑电路示意图	10
图 4-6 示例 6 初始值为 0 的基本时钟触发器及逻辑电路示意图	11
图 4-7 示例 7 异步置位触发器示意图	12
图 4-8 示例 1 RAM 电路图	13
图 4-9 示例 2 RAM 电路图	14
图 4-10 示例 3 RAM 电路图	15
图 4-11 示例 4 RAM 电路图	16
图 4-12 示例 5 RAM 电路图	17
图 4-13 示例 6 pROM 电路图	18
图 4-14 示例 7 RAM 电路图	19
图 4-15 示例 1 DSP 电路图	22
图 4-16 示例 2 DSP 电路图	24
图 4-17 示例 3 DSP 电路图	25
图 4-18 示例 4 DSP 电路图	26
图 4-19 示例 5 DSP 电路图	27
图 6-1 Synthesis Message	55
图 6-2 Synthesis Details	55
图 6-3 Resource	56
图 6-4 Timing	56
图 6-5 Max Frequency Summary	57
图 6-6 Path Summary	57
图 6-7 连接关系、时延及扇出信息	57
图 6-8 Path Statistics	58

表目录

表 1-1 术语、缩略语.....	1
-------------------	---

1 关于本手册

1.1 手册内容

本手册主要描述高云半导体综合工具（GowinSynthesis）的功能及操作，旨在帮助用户快速熟悉 GowinSynthesis 软件的相关功能，指导用户设计，提高设计效率。本手册中的软件界面截图参考高云半导体云源软件（以下简称云源）1.9.10.01 版本，因软件版本升级，部分信息可能会略有差异，具体以用户软件版本的信息为准。

1.2 相关文档

通过登录高云半导体网站 www.gowinsemi.com.cn 可以下载、查看以下相关文档：[SUG100](#)，[Gowin 云源软件用户指南](#)

1.3 术语、缩略语

本手册中的相关术语、缩略语及相关释义请参见表 1-1。

表 1-1 术语、缩略语

术语、缩略语	全称	含义
BSRAM	Block Static Random Access Memory	块状静态随机存储器
DSP	Digital Signal Processing	数字信号处理
FPGA	Field Programmable Gate Array	现场可编程门阵列
FSM	Finite State Machine	有限状态机
GSC	Gowin Synthesis Constraint	GowinSynthesis综合约束文件
SSRAM	Shadow Static Random Access Memory	分布式静态随机存储器

1.4 技术支持与反馈

高云半导体提供全方位技术支持，在使用过程中如有任何疑问或建议，可直接与公司联系：

网址：www.gowinsemi.com.cn

E-mail：support@gowinsemi.com

Tel: +86 755 8262 0391

2 概述

本手册是高云半导体 RTL 设计综合工具 GowinSynthesis 的用户使用说明。

GowinSynthesis 是高云半导体（以下简称高云）自主研发的综合工具，采用了高云原创的 EDA 算法，基于产品硬件特性及硬件电路资源情况，实现 RTL 设计提取、算数优化、推导置换、资源共享、并行综合以及映射等技术，可快速对用户的 RTL 设计进行优化处理、资源检查、时序分析。

GowinSynthesis 针对高云 FPGA 芯片，为 FPGA 设计人员提供了最有效的设计实现方法。在设计实现方面，提供了时序分析、资源检查、原语逻辑分析等功能；以及提供了详细的综合信息。**GowinSynthesis** 生成基于高云器件原语库的综合后网表，可作为高云布局布线工具的输入文件，实现了面积和速度最优平衡结果，提高了软件编译效率，以及布通率。该软件具有如下特点：

- 支持 Verilog/SystemVerilog、VHDL 设计以及混合设计输入
- 支持超大规模设计，可为复杂可编程逻辑设计提供优秀的综合解决方案
- 支持查找表、寄存器、锁存器、算术逻辑单元的推断映射
- 支持 memory 推断映射以及与逻辑资源平衡
- 支持 DSP 的推断映射以及与逻辑资源平衡
- 支持 FSM 的综合优化
- 支持综合属性及综合指令，以满足不同应用条件下综合结果的要求

3 GowinSynthesis 的使用方法介绍

3.1 GowinSynthesis 输入输出

GowinSynthesis 以工程文件 (.prj) 格式读入用户 RTL 文件，工程文件由云源自动生成。GowinSynthesis 工程文件中除指定用户 RTL 文件外，同时还指定了综合器件信息、用户约束文件信息（综合属性约束文件）、综合后网表文件 (.vg) 信息及部分综合选项，如综合顶层模块 (top module) 指定、文件包含路径 (include path) 指定等。

3.2 使用 GowinSynthesis 进行综合

在云源 Process 窗口右键单击 Synthesize，选择 Configuration，该页面可以指定 top module，设置 include path，选择支持语言版本，配置相关综合选项。

在云源 Process 窗口双击 Synthesize 执行综合，Output 窗口会输出综合执行过程中的信息。GowinSynthesis 综合后生成综合报告及门级网表文件，双击 Process 窗口中的 Synthesis Report 和 Netlist File 即可查看具体内容。

具体操作流程请参考文档 [SUG100, Gowin 云源软件用户指南](#) > 4.4.3 Synthesize 章节。

3.3 综合前后的对象命名规则

为便于用户的验证与调试，GowinSynthesis 在整个综合过程中将最大程度的保留用户原始 RTL 设计信息，如用户设计中的 module 模块信息、primitive/module instance 例化名称信息、用户定义的 wire/reg 连线名称等。对必须要经过优化或转化重新生成的对象，这些对象的名称也将由用户定义的连线名称通过一些衍生规则来生成，具体规则如下。

3.3.1 综合后网表文件命名

综合后网表文件名称取决于工程文件 (*.prj) 中指定的输出网表的文件名称。

若工程文件中未指定综合后网表名称，默认生成名称同工程文件名后缀为.vg 的综合后网表文件。

3.3.2 综合后网表 module 命名

综合后网表的 module 名称与 RTL 设计命名一致，多次例化的模块会以 _idx 后缀区分，module 的例化名与 RTL 设计一致。

3.3.3 综合后网表 Instance 命名

用户 RTL 设计中存在 Instance，若该 Instance 在综合过程中不被优化，则在综合后网表中保持 Instance 名称不变。

综合过程中生成的 Instance，Instance 名称来源于该 Instance 在用户 RTL 设计中所表示的功能设计块的外部输出信号名，若该功能设计块有多个输出信号，则该 Instance 名取决于第一个输出信号的名称。

对综合过程中生成的 Instance，Instance 名组成除上述所述的信号名外其后还会根据类型添加后缀，buf 类型后缀为 _ibuf、_obuf 及 _jobuf，其他为 _s 的后缀，s 后的数字为名称被节点引用的次数。

在指定 flatten 输出时，原有的子模块 Instance 名称需要层级表示时，斜线 “/” 将作为层级分隔符。

3.3.4 综合后网表连线命名

用户在 RTL 设计中定义的 wire/reg 信号，若该信号在综合过程中未被优化，则综合后网表的相关模块仍将保留该信号名。

GowinSynthesis 的综合过程中，会针对某些 RTL 设计中的一整块功能设计模块进行置换或优化，综合后，网表中这些功能设计模块的输出信号名会被保留，对模块内部的信号，将根据输出信号名称做衍化，具体为在原始信号名称基础上增加相关数字后缀 (_idx) 来形成衍化后的信号名。

当多位宽的信号 (bus 格式) 名作为衍生信号名，衍生出其他信号名或 Instance 名称时，该信号名称中的 bus 位信息将以下划线加位信息 (_idx) 的形式被保留。

在指定 flatten 输出时，下划线 “_” 将作为层级分隔符。

4 硬件描述语言代码支持

4.1 寄存器的硬件描述语言代码支持

4.1.1 寄存器特征介绍

寄存器包含触发器和锁存器。

触发器

触发器全部为 D 触发器，定义时赋初值。其复位/置位方式有两种：同步复位/置位和异步复位/置位。同步复位/置位指只有当时钟信号 **CLK** 的上升沿或下降沿到来，且 **reset/set** 为高电平时，复位/置位才能完成；异步复位/置位指 **reset** 和 **set** 信号值的变化会激发过程进入到执行状态，只要 **reset/set** 由低电平变为高电平，复位/置位即可完成，不受时钟信号 **CLK** 的控制。

锁存器

锁存器触发方式含高电平触发及低电平触发，定义时赋初值，FPGA 设计最好规避锁存器。高电平触发是当控制信号为高电平时，锁存器允许数据端信号通过；低电平触发是当控制信号为低电平时，锁存器允许数据端信号通过。

4.1.2 寄存器相关的约束

用户可使用 **preserve** 属性对寄存器进行约束。当有此约束时，除了输出悬空的寄存器会被优化外，其他寄存器都会原样保留到综合结果中，详细请参考 **syn_preserve** 章节。

4.1.3 寄存器代码示例

高云芯片设计同步复位时钟触发器的初始值仅可置为 0，同步置位时钟触发器的初始值仅可置为 1，故当用户在 RTL 中设置的同步时钟触发器的初始值与同步时钟触发器的初始值不同时，**GowinSynthesis** 将优先依据 RTL 中的初始值，转换同步时钟触发器的类型。异步时钟触发器不做处理。具体转换策略是：

RTL 设计为同步复位时钟触发器，但指定初始值为 1 时，**GowinSynthesis** 将其替换为同步置位的时钟触发器，将原同步复位信号添加相关逻辑以实现同步置位功能。

RTL 设计为同步置位时钟触发器,但指定初始值为 0 时, GowinSynthesis 将其替换为普通触发器,并在原同步置位信号上添加相关逻辑作为触发器数据端输入。

不指定触发器初始值

CLK 上升沿触发的触发器和 CLK 下降沿触发的触发器的区别仅仅是 CLK 触发方式不同,所以下面只列出能综合出 CLK 上升沿触发的触发器的示例。

示例 1 可被综合为同步复位的时钟触发器

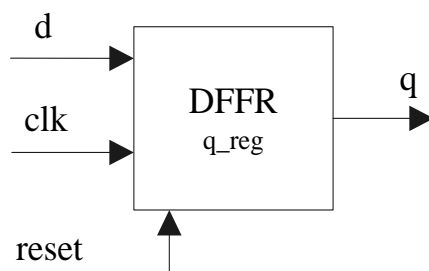
```

module top (q, d, clk, reset);
input d;
input clk;
input reset;
output q;
reg q_reg;
always @(posedge clk)begin
    if(reset)
        q_reg<=1'b0;
    else
        q_reg<=d;
end
assign q = q_reg;
endmodule

```

同步复位时钟触发器示意如图 4-1 所示。

图 4-1 示例 1 同步复位时钟触发器示意图



示例 2 可被综合为同步置位且带有时钟使能功能的触发器

```

module top (q, d, clk, ce, set);
input d;

```

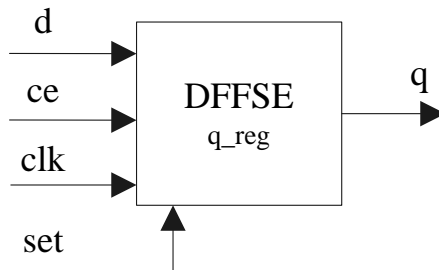
```

input clk;
input ce;
input set;
output q;
reg q_reg;
always @(posedge clk)begin
    if(set)
        q_reg<=1'b1;
    else if(ce)
        q_reg<=d;
end
assign q = q_reg;
endmodule

```

同步置位且带有时钟使能功能的触发器示意如图 4-2 所示。

图 4-2 例 2 同步置位且带有使能功能的触发器示意图



例 3 可被综合为异步复位且带有时钟使能功能的触发器

```

module top (q, d, clk, ce, clear);
input d;
input clk;
input ce;
input clear;
output q;
reg q_reg;
always @(posedge clk or posedge clear)begin
    if(clear)
        q_reg<=1'b0;
    else if(ce)

```

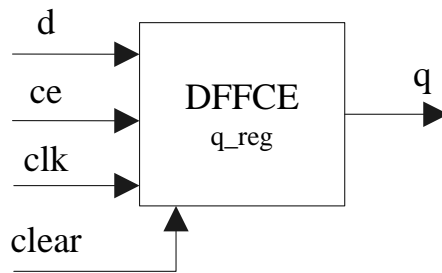
```

        q_reg<=d;
    end
    assign q = q_reg;
endmodule

```

异步复位且带有时钟使能功能的触发器示意如图 4-3 所示。

图 4-3 示例 3 异步复位且带有时钟使能功能的触发器示意图



示例 4 可被综合为带有复位和高电平使能功能的锁存器

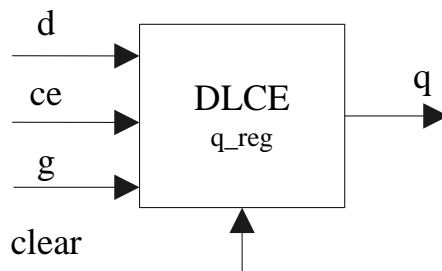
```

module top(d,g,clear,q,ce);
    input d,g,clear,ce;
    output q;
    reg q_reg;
    always @(g or d or clear or ce) begin
        if(clear)
            q_reg <= 0;
        else if(g && ce)
            q_reg <= d;
    end
    assign q = q_reg;
endmodule

```

带有复位和高电平使能功能的锁存器示意如图 4-4 所示。

图 4-4 示例 4 带有复位和高电平使能功能的锁存器示意图



指定触发器初始值

示例 5 为同步复位的时钟触发器，其初始值应为 0，但 RTL 中设置初始值为 1，将被综合为初始值为 1 的同步置位的时钟触发器及用于实现同步复位功能的逻辑电路。

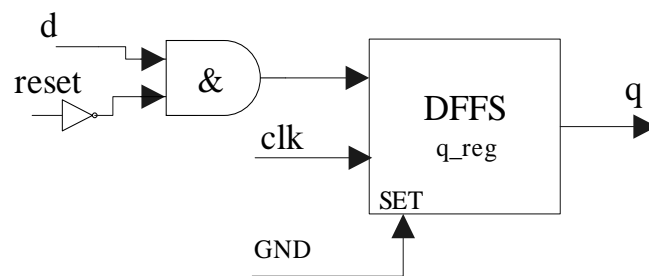
```

module top (q, d, clk, reset);
  input d;
  input clk;
  input reset;
  output q;
  reg q_reg = 1'b1;
  always @(posedge clk)begin
    if(reset)
      q_reg<=1'b0;
    else
      q_reg<=d;
  end
  assign q = q_reg;
endmodule

```

如上的同步复位时钟触发器示意如图 4-5 所示。

图 4-5 示例 5 同步复位时钟触发器及逻辑电路示意图



示例 6 为同步置位的时钟触发器，其初始值应为 1，但 RTL 中设置初始值为 0，将被综合为初始值为 0 的普通时钟触发器及一个用于实现同步置位功能的逻辑电路。

```

module top (q, d, clk, set);
  input d;
  input clk;
  input set;
  output q;

```

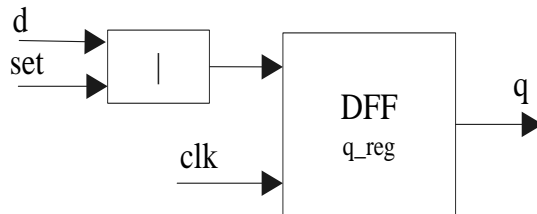
```

reg q_reg = 1'b0;
always @(posedge clk)begin
    if(set)
        q_reg<=1'b1;
    else
        q_reg<=d;
end
assign q = q_reg;
endmodule

```

如上初始值为 0 的基本时钟触发器及逻辑电路示意如图 4-6 所示。

图 4-6 示例 6 初始值为 0 的基本时钟触发器及逻辑电路示意图



示例 7 为设置初值为 1 的异步置位触发器。

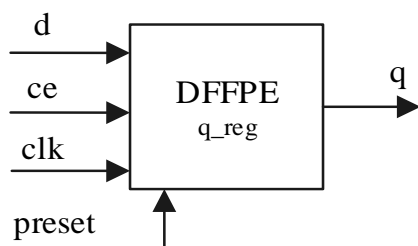
```

module top (q, d, clk, ce, preset);
input d;
input clk;
input ce;
input preset;
output q;
reg q_reg = 1'b1;
always @(posedge clk or posedge preset)begin
    if(preset)
        q_reg<=1'b1;
    else if(ce)
        q_reg<=d;
end
assign q = q_reg;
endmodule

```

如上的异步置位触发器示意如图 4-7 所示。

图 4-7 示例 7 异步置位触发器示意图



4.2 RAM 的硬件描述语言代码支持

4.2.1 RAM 置换基本功能介绍

RAM 置换是 RTL 综合过程中将用户设计中的存储功能部分推导置换为块状静态随机存储器 (BSRAM) 或分布式随机存储器 (SSRAM) 的过程。用户在设计 RTL 时既可以直接实例化 BSRAM 或 SSRAM 原语,也可以写不依赖器件的 RTL 格式的存储器描述。对 RTL 格式的存储器块, GowinSynthesis 将依据 RTL 描述, 将符合相应条件的 RTL 描述置换为相应的 RAM 模块。

当逻辑模块需要使用 BSRAM 来实现时, 需要满足以下原则:

1. 所有的输出寄存器有相同的控制信号;
2. RAM 必须为同步存储器, 不可以有异步的控制信号相连, GowinSynthesis 不支持异步 RAM;
3. 需要在读地址或者输出端连接寄存器。

4.2.2 RAM 特征介绍

BSRAM

BSRAM 的配置模式分为单端口模式, 双端口模式, 伪双端口模式, 只读模式; 读模式分为寄存器输出模式 (pipeline) 及旁路模式 (bypass) 两种; 写模式支持普通模式 (normal Mode)、通写模式 (write-through Mode) 及先读后写模式 (read-before-write Mode) 三种。

SSRAM

配置模式分为单端口模式、伪双端口模式及只读模式三种, SSRAM 不支持双端口模式。

4.2.3 RAM 推导相关的约束

syn_ramstyle 指定存储器的实现方式, syn_romstyle 指定只读存储器的实现方式。

如果设计中指定要生成 SSRAM 或者 BSRAM, 请使用约束语句 ram_style、rom_style 或 syn_srlstyle 来控制。

约束语句具体使用方式请参考 syn_ramstyle、syn_srlstyle 章节。

4.2.4 RAM 推导代码示例

按照 RAM 的不同特征举例如下：

示例 1 为 1 个写端口，1 个读端口并且读写端口地址相同的存储器，可以被综合为普通模式的单端口 BSRAM。

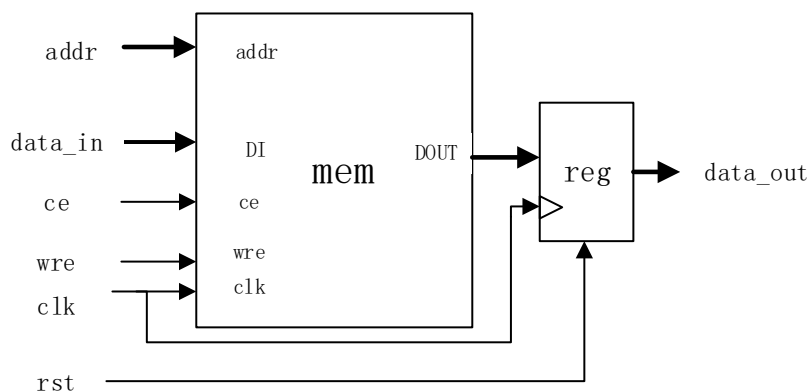
```

module normal(data_out, data_in, addr, clk, ce, wre, rst);
output [7:0]data_out;
input [7:0]data_in;
input [7:0]addr;
input clk,wre,ce,rst;
reg [7:0] mem [255:0];
reg [7:0] data_out;
always@(posedge clk or posedge rst)
if(rst)
    data_out <= 0;
else
    if(ce & !wre)
        data_out <= mem[addr];
always @(posedge clk)
    if (ce & wre)
        mem[addr] <= data_in;
endmodule

```

如上的单端口 BSRAM 电路描述示意图如图 4-8 所示。

图 4-8 示例 1 RAM 电路图



示例 2 为 1 个写端口，1 个读端口并且读写端口地址相同的存储器，当 wre 为 1 时，输入数据可以直接传输给输出，此案例被综合为普通写模式的单端口 BSRAM。

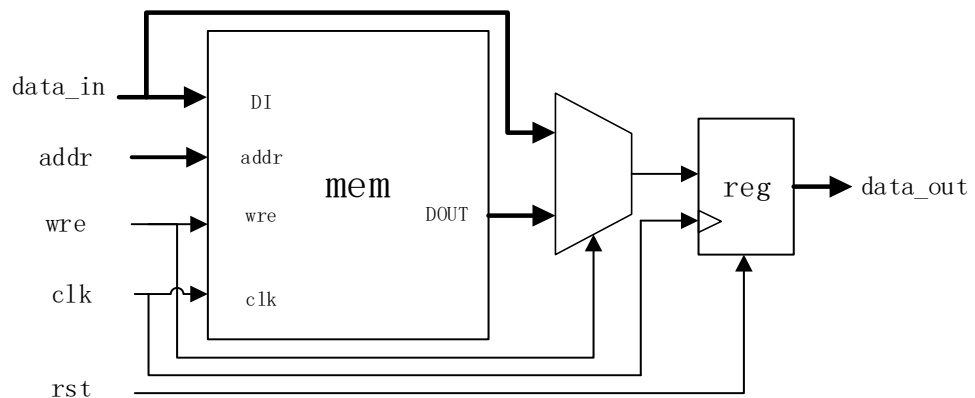
```

module wt11(data_out, data_in, addr, clk, wre, rst);
output [31:0]data_out;
input [31:0]data_in;
input [6:0]addr;
input clk, wre, rst;
reg [31:0] mem [127:0];
reg [31:0] data_out;
always@(posedge clk or posedge rst)
if(rst)
    data_out <= 0;
else if(wre)
    data_out <= data_in;
else
    data_out <= mem[addr];
always @(posedge clk)
if (wre)
    mem[addr] <= data_in;
endmodule

```

如上的单端口 BSRAM 电路描述示意图如图 4-9 所示。

图 4-9 示例 2 RAM 电路图



示例 3 为 1 个写端口，1 个读端口并且读写端口地址相同的存储器，当 wre 为 1 时，输入数据写入存储器中，此案例被综合为先读后写模式的单端口 BSRAM。

```

module read_first_01(data_out, data_in, addr, clk, wre);
output [31:0]data_out;
input [31:0]data_in;

```

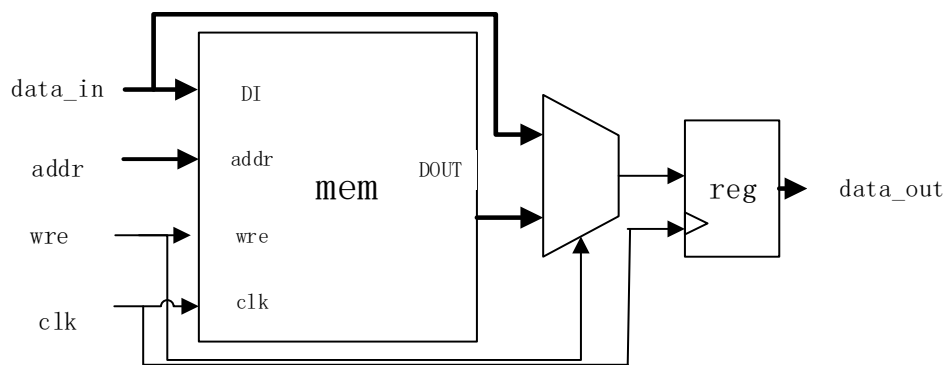
```

input [6:0]addr;
input clk,wre;
reg [31:0] mem [127:0];
reg [31:0] data_out;
always @(posedge clk)
begin
    if (wre)
        mem[addr] <= data_in;
        data_out <= mem[addr];
    end
endmodule

```

如上的单端口 BSRAM 电路描述示意图如图 4-10 所示。

图 4-10 示例 3 RAM 电路图



示例 4 为 2 个写端口, 1 个读端口的存储器, 一个写端口没有 **wre** 信号, 另外一个写端口有 **wre** 信号, 1 个读端口吸收异步复位的寄存器。此案例被综合为写模式 A 端为普通模式, B 端为先读后写模式, 读模式为寄存器输出模式的异步复位双端口 BSRAM。

```

module read_first_02_1(data_outa, data_ina, addra, clka, rsta, cea,
wrea, ocea, data_inb, addrb, clk, ceb );
output [17:0]data_outa;
input [17:0]data_ina,data_inb;
input [6:0]addra,addrb;
input clka, rsta,cea, wrea,ocea;
input clk, ceb;
reg [17:0] mem [127:0];
reg [17:0] data_outa;

```

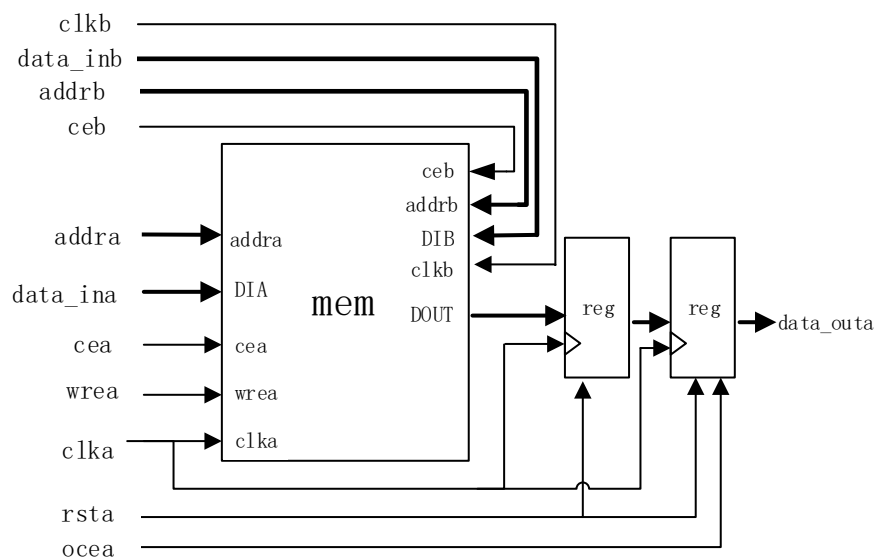
```

reg [17:0] data_out_rega,data_out_regb;
always @(posedge clkb)
if (ceb)
    mem[addrb] <= data_inb;
always@(posedge clka or posedge rsta)
if(rsta)
    data_out_rega <= 0;
else begin
    data_out_rega <= mem[addra];
end
always@(posedge clka or posedge rsta)
if(rsta)
    data_outa <= 0;
else if (ocea)
    data_outa <= data_out_rega;
always @(posedge clka)
if (cea & wrea)
    mem[addra] <= data_ina;
endmodule

```

如上的双端口 BSRAM 电路描述示意图如图 4-11 所示。

图 4-11 示例 4 RAM 电路图



示例 5 为 1 个读端口，1 个写端口且读写地址不同的存储器，被综合为写模式为普通模式，读模式为旁路模式的伪双端口 BSRAM。

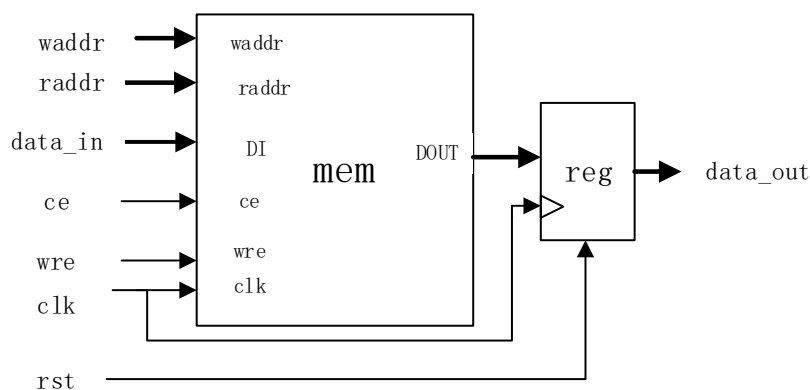
```

module read_first_wp_pre_1(data_out, data_in, waddr, raddr,clk, rst,ce);
output [10:0]data_out;
input [10:0]data_in;
input [6:0]raddr,waddr;
input clk, rst,ce;
reg [10:0] mem [127:0];
reg [10:0] data_out;
always@(posedge clk or posedge rst)
if(rst)
    data_out <= 0;
else if(ce)
    data_out <= mem[raddr];
always @(posedge clk)
if (ce) mem[waddr] <= data_in;
endmodule

```

如上的伪双端口 BSRAM 电路描述示意图如图 4-12 所示。

图 4-12 示例 5 RAM 电路图



示例 6 为具有初值的只有 1 个读端口的存储器，被综合为读模式为旁路模式的异步置位只读存储器。

```

module test_invce (clock,ce,oce,reset,addr,dataout) ;
input clock,ce,oce,reset;
input [5:0] addr;
output [7:0] dataout;
reg [7:0] dataout;
always @(posedge clock or posedge reset)
if(reset) begin

```



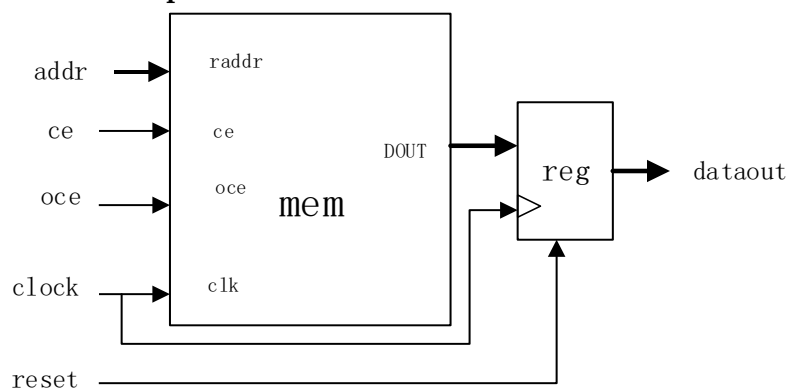
```

        dataout <= 0;
    end else begin
    if (ce & oce) begin
    case (addr)
    6'b000000: dataout <= 32'h87654321;
    6'b000001: dataout <= 32'h18765432;
    6'b000010: dataout <= 32'h21876543;
    .....
    6'b111110: dataout <= 32'hdef89aba;
    6'b111111: dataout <= 32'hef89abce;
    default: dataout <= 32'hf89abcde;
    endcase
    end
    end
    endmodule

```

如上的只读存储器电路描述示意图如图 4-13 所示。

图 4-13 示例 6 pROM 电路图



示例 7 为 shift register 模式的存储器，被综合为普通模式的伪双端口 BSRAM。

```

module seqshift_bsram (clk, din, dout);
parameter SRL_WIDTH = 65;
parameter SRL_DEPTH = 16;
input clk;
input [SRL_WIDTH-1:0] din;
output [SRL_WIDTH-1:0] dout;
reg [SRL_WIDTH-1:0] regBank[SRL_DEPTH-1:0];

```

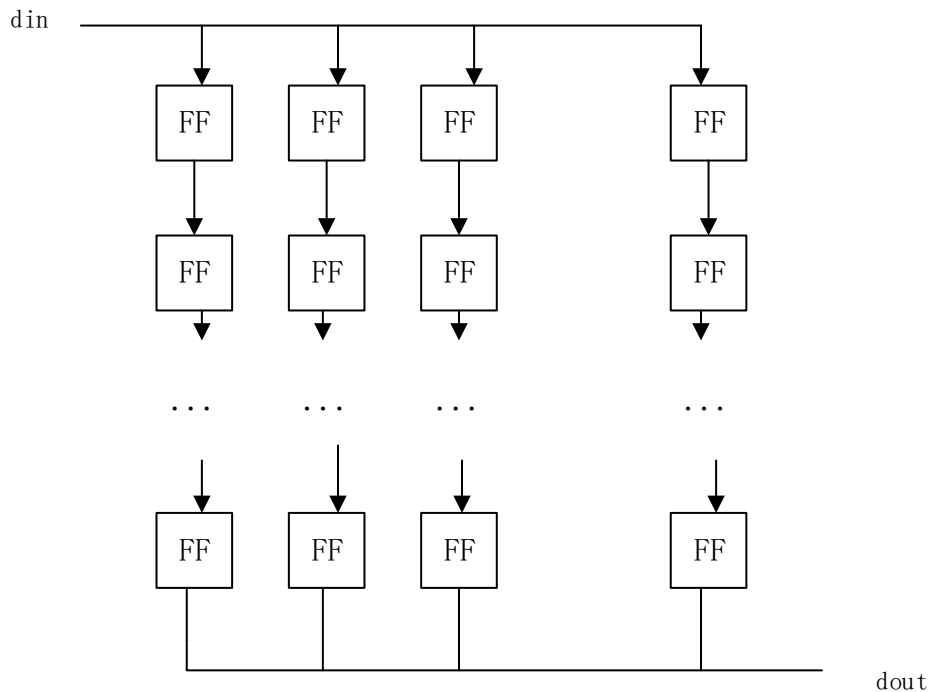
```

integer i;
always @(posedge clk) begin
    for (i=SRL_DEPTH-1; i>0; i=i-1) begin
        regBank[i] <= regBank[i-1];
    end
    regBank[0] <= din;
end
assign dout = regBank[SRL_DEPTH-1];
endmodule

```

如上的伪双端口 BSRAM 电路描述示意图如图 4-14 所示。

图 4-14 示例 7 RAM 电路图



注！

更多示例，请参见高云官网 GowinSynthesis 推导置换编码模板文档
[“GowinSynthesis Inference Coding Template”](#)。

4.3 DSP 的硬件描述语言代码支持

4.3.1 DSP 推导的基本功能介绍

DSP 推导是综合过程中将用户设计中的乘法及部分加法推导置换为 DSP 的算法。用户在设计 RTL 时既可以实例化 DSP 也可以写 RTL 格式的 DSP 描述，GowinSynthesis 将依据 RTL 描述，将符合相应条件的 RTL 描述推导置换为相应的 DSP 模块。

DSP 模块具有乘法以及加法和寄存器的功能。当用户当前使用的器件不

支持 DSP 模块时，GowinSynthesis 会使用逻辑电路来实现乘法器的功能。

4.3.2 DSP 特征介绍

高云 DSP 分为乘法器、乘加器、预加器及累加器。具有以下功能：

1. 支持输入符号位不同的乘法置换；
2. 支持同步或异步模式；
3. 支持乘法的级联；
4. 支持乘法的累加；
5. 支持预加功能；
6. 支持寄存器的吸收，包括输入寄存器，输出寄存器，旁路寄存器的吸收。

4.3.3 DSP 相关的约束

`syn_dspstyle` 用来控制具体对象或者全局范围的乘法器使用 DSP 还是逻辑电路实现。

`syn_perserve` 用来保留寄存器，当 DSP 周围的寄存器有此属性时，DSP 不可以吸收此寄存器。

约束语句的具体使用方式请参考 `syn_dspstyle`、`syn_preserve` 章节。

4.3.4 DSP 推导代码示例

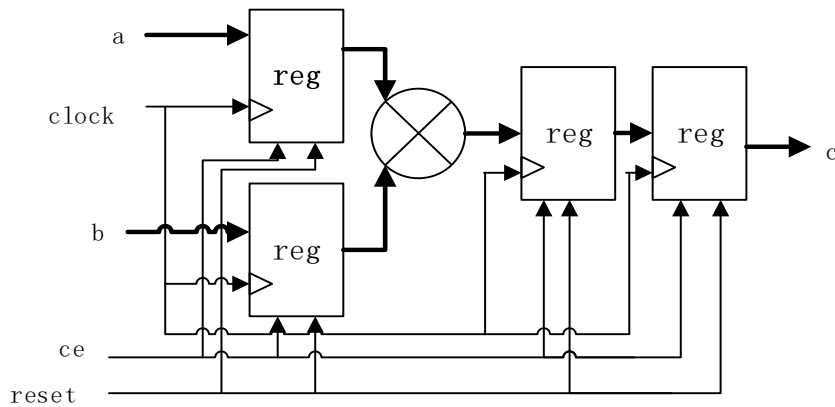
示例 1 可综合出带符号位的同步复位的乘法器，该乘法器的输入寄存器为 `ina` 和 `inb`，输出寄存器为 `out_reg`，旁路寄存器为 `pp_reg`。

```
module top(a,b,c,clock,reset,ce);
parameter a_width = 18;
parameter b_width = 18;
parameter c_width = 36;
input signed [a_width-1:0] a;
input signed [b_width-1:0] b;
input clock;
input reset;
input ce;
output signed [c_width-1:0] c;
reg signed [a_width-1:0] ina;
reg signed [b_width-1:0] inb;
reg signed [c_width-1:0] pp_reg;
reg signed [c_width-1:0] out_reg;
wire signed [c_width-1:0] mult_out;
```

```
always @(posedge clock) begin
    if(reset)begin
        ina<=0;
        inb<=0;
    end else begin
        if(ce)begin
            ina<=a;
            inb<=b;
        end
    end
end
assign mult_out=ina*inb;
always @(posedge clock) begin
    if(reset)begin
        pp_reg<=0;
    end else begin
        if(ce)begin
            pp_reg<=mult_out;
        end
    end
end
always @(posedge clock) begin
    if(reset)begin
        out_reg<=0;
    end else begin
        if(ce)begin
            out_reg<=pp_reg;
        end
    end
end
assign c=out_reg;
endmodule
```

如上的乘法器电路描述示意图如图 4-15 所示。

图 4-15 示例 1 DSP 电路图



示例 2 可综合出异步模式的乘加器, 该乘加器的输入端寄存器为 a0_reg、a1_reg、b0_reg 和 b1_reg, 输出端寄存器为 s_reg, 旁路寄存器为 p0_reg 和 p1_reg。

```

module top(a0, a1, b0, b1, s, reset, clock, ce);
parameter a0_width=18;
parameter a1_width=18;
parameter b0_width=18;
parameter b1_width=18;
parameter s_width=37;
input unsigned [a0_width-1:0] a0;
input unsigned [a1_width-1:0] a1;
input unsigned [b0_width-1:0] b0;
input unsigned [b1_width-1:0] b1;
input reset, clock, ce;
output unsigned [s_width-1:0] s;
wire unsigned [s_width-1:0] p0, p1, p;
reg unsigned [a0_width-1:0] a0_reg;
reg unsigned [a1_width-1:0] a1_reg;
reg unsigned [b0_width-1:0] b0_reg;
reg unsigned [b1_width-1:0] b1_reg;
reg unsigned [s_width-1:0] p0_reg, p1_reg, s_reg;
always @(posedge clock or posedge reset)
begin
    if(reset)begin
        a0_reg <= 0;
    end
end

```

```
        a1_reg <= 0;
        b0_reg <= 0;
        b1_reg <= 0;
    end else begin
        if(ce)begin
            a0_reg <= a0;
            a1_reg <= a1;
            b0_reg <= b0;
            b1_reg <= b1;
        end
    end
end
assign p0 = a0_reg*b0_reg;
assign p1 = a1_reg*b1_reg;
always @(posedge clock or posedge reset)
begin
    if(reset)begin
        p0_reg <= 0;
        p1_reg <= 0;
    end else begin
        if(ce)begin
            p0_reg <= p0;
            p1_reg <= p1;
        end
    end
end
end
assign p = p0_reg - p1_reg;
always @(posedge clock or posedge reset)
begin
    if(reset)begin
        s_reg <= 0;
    end else begin
        if(ce) begin
            s_reg <= p;
        end
    end
end
```

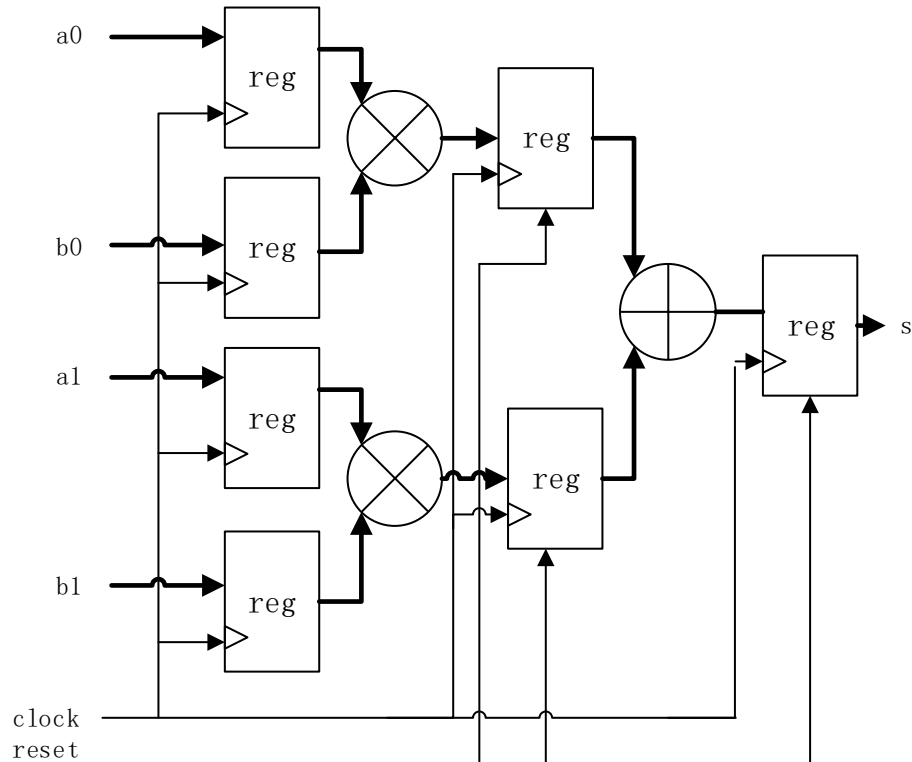
```

end
end
end
assign s = s_reg;
endmodule

```

如上的乘加器电路描述示意图如图 4-16 所示。

图 4-16 示例 2 DSP 电路图



示例 3 可以综合出两个无符号位的乘加器，这两个乘加器为级联关系。

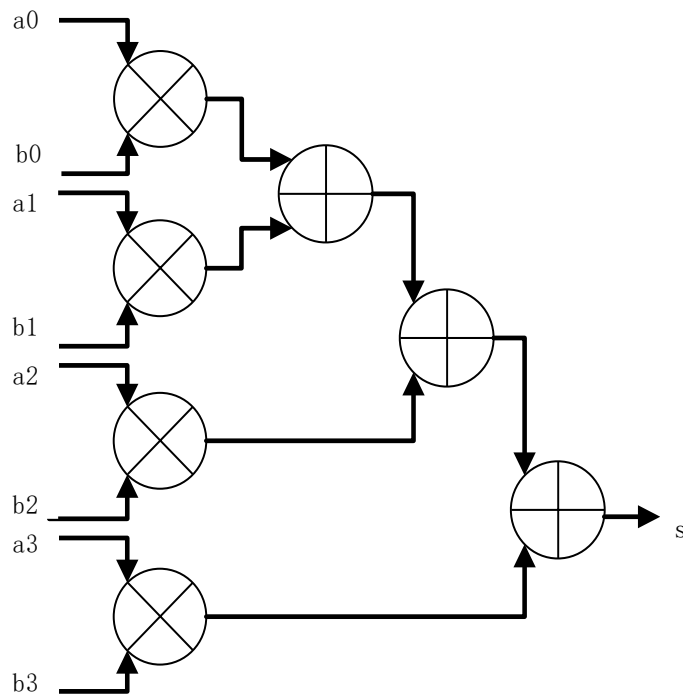
```

module top(a0, a1, a2, b0, b1, b2, a3, b3, s);
parameter a_width=18;
parameter b_width=18;
parameter s_width=36;
input unsigned [a_width-1:0] a0, a1, a2, b0, b1, b2, a3, b3;
output unsigned [s_width-1:0] s;
assign s=a0*b0+a1*b1+a2*b2+a3*b3;
endmodule

```

如上的乘加器电路描述示意图如图 4-17 所示。

图 4-17 示例 3 DSP 电路图



示例 4 可以综合出一个符号位为 0 的乘法器和符号位为 0 的预加器，该乘法器的一个输入端与预加器的输出端 b 相互连接。

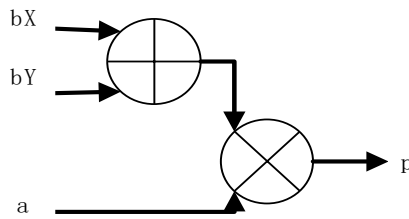
```

module top(a, bX, bY, p);
parameter a_width=36;
parameter b_width=18;
parameter p_width=54;
input [a_width-1:0] a;
input [b_width-1:0] bX, bY;
output [p_width-1:0] p;
wire [b_width-1:0] b;
assign b = bX + bY;
assign p = a*b;
endmodule

```

如上的乘加器电路描述示意图如图 4-18 所示。

图 4-18 示例 4 DSP 电路图



示例 5 可综合出一个符号位为 0 的乘法累加器，该乘法累加器的输出寄存器为 *s*。

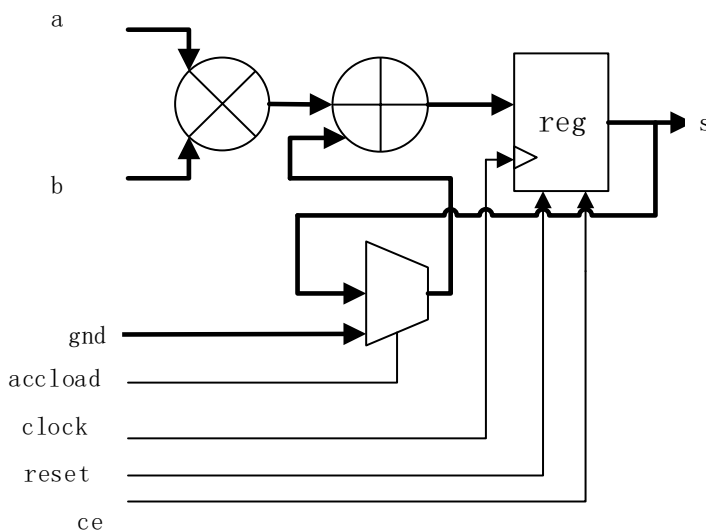
```

module acc(a, b, s, accload, reset, ce, clock);
  parameter a_width=36; //18 36
  parameter b_width=18; //18 36
  parameter s_width=54; //54
  input unsigned [a_width-1:0] a;
  input unsigned [b_width-1:0] b;
  input accload, reset, ce, clock;
  output unsigned [s_width-1:0] s;
  wire unsigned [s_width-1:0] s_sel;
  wire unsigned [s_width-1:0] p;
  reg [s_width-1:0] s;
  assign p = a*b ;
  assign s_sel = (accload == 1'b1) ? s : 54'h00000000;
  always @(posedge clock)
  begin
    if(reset)begin
      s <= 0;
    end else begin
      if(ce)begin
        s <= s_sel + p;
      end
    end
  end
endmodule

```

如上的乘法累加器电路描述示意图如图 4-19 所示。

图 4-19 示例 5 DSP 电路图



注！

更多示例，请参见 GowinSynthesis 推导置换编码模板文档“[GowinSynthesis_Inference_Coding_Template](#)”。

4.4 有限状态机的综合实现规则

4.4.1 有限状态机的综合规则

GowinSynthesis 支持有限状态机 (Finite State Machine, FSM) 的综合，编码方式支持独热码、格雷码、二进制码等。有限状态机的综合结果与状态机的编码方式、编码个数、编码位宽、编码约束等信息有关。在不设定编码约束的情况下，GowinSynthesis 自动选用独热码、格雷码或者二进制实现状态机；若有编码约束的情况下优先根据约束指定的编码方式进行实现，有关状态机的编码约束请参考 `syn_encoding` 章节。

注！

有限状态机的输出若直接驱动输出端口，GowinSynthesis 不将其作为状态机进行综合，此时状态机上的编码约束将被忽略。

4.4.2 有限状态机代码示例

有限状态机的综合规则介绍如下。

独热码状态机

若 RTL 设计中状态机采用独热码进行编码，在不设定编码约束的情况下，GowinSynthesis 默认选择独热码实现状态机的功能，在有编码约束的情况下则依据约束指定的编码方式实现状态机功能。独热码的编码方式举例如下。

```
reg [3:0] state,next_state;
parameter state0=4'b0001;
parameter state1=4'b0010;
parameter state2=4'b0100;
parameter state3=4'b1000;
```

上述案例中 RTL 采用独热码编码，GowinSynthesis 采用独热码进行实现。

格雷码状态机

若 RTL 设计中状态机采用格雷码进行编码，在不设定编码约束的情况下，GowinSynthesis 默认选择格雷码实现状态机的功能，在有编码约束的情况下则依据约束指定的编码方式实现状态机功能。格雷码的编码方式举例如下。

```
reg [3:0] state,next_state;  
parameter state0=2'b00;  
parameter state1=2'b01;  
parameter state2=2'b11;  
parameter state3=2'b10;
```

上述案例中 RTL 采用格雷码编码，GowinSynthesis 采用格雷码进行实现。

二进制码或其它编码状态机

若 RTL 设计中状态机采用二进制码进行编码，既不是独热码又不是格雷码，在不设定编码约束的情况下，GowinSynthesis 将根据编码的个数和位宽选择相应的编码进行实现。选取原则如下所示，若编码的个数大于编码的有效位宽则选用二进制码进行实现，若编码的个数小于等于编码的有效位宽则选用独热码进行实现；在有编码约束的情况下则依据约束指定的编码方式实现状态机功能。

示例 1

```
reg [5:0] state,next_state;  
parameter state0= 6'b000001;  
parameter state1= 6'b000011;  
parameter state2= 6'b000000;  
parameter state3= 6'b010101;
```

上述案例中编码个数为 4，编码的位宽为 6，其中有效位宽为 5，可见编码的个数小于编码的有效位宽，采用独热码进行实现。

示例 2

```
reg [2:0] state,next_state;  
parameter state0=3'b001;  
parameter state1=3'b010;  
parameter state2=3'b011;  
parameter state3=3'b100;
```

上述案例中编码个数为 4，编码的有效位宽为 3，可见编码的个数大于编码的有效位宽，采用二进制码进行实现。

示例 3

```
reg [5:0] state,next_state;  
parameter state0= 1;  
parameter state1= 3;  
parameter state2= 6;  
parameter state3= 15;
```

上述案例中编码的个数为 4，编码采用 10 进制，有效位宽换算为二进制后是 4 位，可见编码的个数等于编码的有效位宽，采用独热码进行实现。

5 综合约束支持

属性约束用于设置综合项目中优化选择、功能实现方式、输出网表格式等的各种属性，使综合结果更好地满足用户的设计功能和使用场景。属性设置可以写在约束文件中，也可以嵌入在源码中。

本章描述了 RTL 文件中的约束和 GowinSynthesis 约束文件 GSC (Gowin Synthesis Constraint) 约束的语法。Verilog 文件是大小写敏感的，因此指令及属性必须严格按照语法描述的进行输入。约束语句中一条属性约束须写在一行中，中间不可以有换行符等进行分隔，语句末尾需要添加分号。

RTL 文件中的约束

RTL 文件中的约束必须在约束对象 (object) 的定义语句中添加，添加在定义语句结束的分号之前。每个属性约束只能作用于单个约束对象，如需约束多个对象需多次添加属性约束，如 “reg dout1 /*synthesis syn_preserve=1*/, dout2 /*synthesis syn_preserve=1*/;”，或 “reg dout1 /*synthesis syn_preserve=1*/; reg dout2 /*synthesis syn_preserve=1*/;”。语句中的约束属性值 (setting_value) 如果为字符串，则 setting_value 值前后需添加双引号，如果 setting_value 值为数字，则 setting_value 值前后不可以添加双引号。

GSC

GSC 约束分为 Instance 类型的约束、Net 类型的约束、Port 类型的约束及全局对象约束。在书写过程中为了区分不同的类型，有不同的语法形式，约束对象必须使用双引号将其括起来，attributeName (属性名称) 与约束属性值不需要使用双引号或其他符号标识，之间的等号前后可以有空格，GSC 约束中支持注释，注释使用 “//”。具体语句语法如下：

```
INS "object" attributeName=setting_value;  
NET "object" attributeName=setting_value;  
PORT "object" attributeName=setting_value;  
GLOBAL attributeName=setting_value;
```

约束语句开头为 INS，object 必须为 instance 名称，instance 包括 module/entity instance 及 primitive instance，instance 名称中不包含中括号，即 bus 时不要写 temp[15:0]，写成 temp 即可。

约束语句开头为 **NET**，约束对象必须为 **net** 名称。

约束语句开头为 **PORT**，约束对象必须为 **port** 名称。

约束语句开头为 **GLOBAL**，说明后面的属性约束是全局属性约束，约束对象为全局。

约束中的对象名称必须与网表中的名字匹配，名字中间不可以有空格，对象名称中支持通配符，使用“/”来区分名字之间的层级关系。使用通配符时对象名前加 **w** 来区分，比如 **w "object"**。

约束属性值 (**setting_value**) 的设定值可能是用户直接指定的值、继承于上层结构的值，和该属性的默认值。取值的优先顺序为 **GSC** 中的直接值 > **RTL** 中的直接值 > **GSC** 中的继承值 > **RTL** 中的继承值 > 默认值，拥有多个继承值时，取距离指定名字最近（层级最低）的值。例如，查询 **A/D/C/mult1**（“/”来表示模块名字之间的层级关系）的 **MULT_STYLE** 属性，其有直接值为 **dsp**；查询“**A/D/C**”的 **MULT_STYLE** 属性，其没有直接值，而 **MULT_STYLE** 属性可以继承，所以找到并继承“**A/D**”的属性值 **logic**；查询“**A/D/C/mult1**”的 **MULT_STYLE**，其既有“**A/D**”的继承值，又有直接值，由于直接值优先级高，所以最终取直接值 **DSP**。

5.1 black_box_pad_pin

描述

指定黑盒子的 **IO pads** 对外部环境是可见的。该属性只对黑盒子的 **io pad** 起作用。

该属性只可在 **RTL** 文件中指定。

语法

Verilog 约束语法

```
Verilog object /* synthesis black_box_pad_pin=portList */;
```

VHDL 约束语法

```
attribute black_box_pad_pin : string;
```

```
attribute black_box_pad_pin of object: objectType is portList;
```

注！

- **object**: 是一个黑盒子定义的 **module** 或 **component**。
- **objectType**: 为 **object** 的 **type**，如 **component**。
- 用双引号括起来的 **PortList** 是一个无空格、以逗号分隔的列表，该列表列出了黑盒子上的端口名称。

示例

Verilog 约束示例

```
module top(clk, in1, in2, out1, out2,D,E);
```

```
input clk;
```

```
input [1:0]in1;
```

```

    input [1:0]in2;
    output [1:0]out1;
    output [1:0]out2;
    output D,E;
    .....
    black_box_add U2 (in1, in2, out2,D,E);
endmodule

module black_box_add(A, B, C, D,E)/* synthesis syn_black_box
black_box_pad_pin="D,E" */;
    input [1:0]A;
    input [1:0]B;
    output [1:0]C;
    output D,E;
endmodule

```

VHDL 约束示例

```

library ieee;
use ieee.std_logic_1164.all;
entity top is
    generic (width : integer := 4);
    port (in1,in2 : in std_logic_vector(width downto 0);
          clk : in std_logic;
          q : out std_logic_vector (width downto 0)
    );
end top;
architecture top1_arch of top is
    component test is
        generic (width1 : integer := 2);
        port (in1,in2 : in std_logic_vector(width1 downto 0);
              clk : in std_logic;
              q : out std_logic_vector (width1 downto 0)
        );
    end component;
    attribute black_box_pad_pin : string;
    attribute black_box_pad_pin of test : component is "q[4:0]";

```

```
begin  
test123 : test generic map (width) port map (in1,in2,clk,q);  
end top1_arch;
```

5.2 full_case

描述

`full_case` 只在 Verilog 的设计中使用。当在 `case`, `casex` 或者 `casez` 语句后添加此属性说明所有可能的值都已经给出, 不需要使用多余的硬件来保留信号值。

该属性只可在 RTL 文件中指定,只支持 Verilog 语法的设计。

语法

Verilog 约束语法

```
verilog case /* synthesis full_case*/
```

示例

Verilog 约束示例

示例 1 指定此部分电路不再需要使用多余的硬件来保留信号值

```
module top(...);  
.....  
always @(select or a or b or c or d)  
begin  
casez(select) /* synthesis full_case*/  
4'b???1: out=a;  
.....  
4'b1???: out=d;  
endcase  
end  
endmodule
```


5.3 parallel_case

指令。强制使用并行多路复用（parallel-multiplexed）结构而不是优先级编码（priority-encoded）结构。

该指令只可在 Verilog 文件中指定。

描述

case 语句默认被定义为用优先级顺序工作，只执行与选中值相匹配的第一条语句。优先级编码允许同时在几个输入端有输入信号，按输入信号排定的优先顺序，只对同时输入的几个信号中优先级最高的一个进行编码。

如果选择的 bus 是从当前模块外部驱动的，当前模块没有合法选中值的信息，软件必须创建一个禁用的逻辑链，以便匹配标签声明禁用所有后续语句。

但是如果知道选中的合法值，可以用 parallel_case 指令消除额外的优先级编码逻辑。

语法

Verilog 约束语法

```
object /* synthesis parallel_case */;
```

注！

- global support: No
- object: case、casex、casez 语句声明。
- setting_value: 不需取值。

示例

Verilog 约束示例

```
module test (out, a, b, c, d, select);
  output out;
  input a, b, c, d;
  input [3:0] select;
  reg out;
  always @(select or a or b or c or d)
  begin
    casez (select) /* synthesis parallel_case */
      4'b???1: out = a;
      4'b??1?: out = b;
      4'b?1??: out = c;
      4'b1???: out = d;
      default: out = 'bx;
    endcase
  end
```

```
end
endmodule
```

5.4 syn_black_box

描述

指定 `module` 或 `component` 为黑盒子。在综合时，黑盒子模块只定义它的接口，其内容是不可访问的和优化。无论模块是否为空，均作为黑盒子。

该属性只可在 RTL 文件中指定。

语法

Verilog 约束语法

```
object /* synthesis syn_black_box */;
```

VHDL 约束语法

```
attribute syn_black_box: boolean;
```

```
attribute syn_black_box of object : objectType is true;
```

注!

- `object`: 指定作用的对象，只可以是 `sub module/entity`。
- `objectType`: 为 `object` 的 `type`，如 `component`。

示例

Verilog 约束示例

```
module top(clk, in1, in2, out1, out2);
input clk;
input [1:0]in1;
input [1:0]in2;
output [1:0]out1;
output [1:0]out2;
add U1 (clk, in1, in2, out1);
black_box_add U2 (in1, in2, out2);
endmodule
```

```
module add (clk, in1, in2, out1);
.....
begin
out1 <= in1 + in2;
end
endmodule
```

```

module black_box_add(A, B, C)/* synthesis syn_black_box */;
.....
assign C = A + B;
endmodule

```

使用该属性之前，模块 `black_box_add` 的内容是可见的。使用该属性之后，模块 `black_box_add` 的内容不可见，变为黑盒子。

VHDL 约束示例

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity mux2_1_top is
port(
    dina : in bit;
    dinb : in bit;
    sel  : in bit;
    dout : out bit
);
end mux2_1_top;

architecture Behavioral of mux2_1_top is
.....
    attribute syn_black_box: boolean;
    attribute syn_black_box of mux2_1 : component is true;
begin
    u_mux2_1 : mux2_1
    port map(
        dina => dina,
        dinb => dinb,
        sel  => sel,
        dout => dout
    );
end Behavioral;

```

5.5 syn_dspstyle

描述

指定乘法器以专用 DSP 硬件模块或逻辑电路的方式实现，既可以作用于具体 module/entity instance，也可以作用于全局范围。该属性可在 GSC、RTL 文件中指定。

语法

GSC 约束语法

```
INS "object" syn_dspstyle =setting_value;
```

```
GLOBAL syn_dspstyle =setting_value;
```

Verilog 约束语法

```
Verilog object /* synthesis syn_dspstyle ="setting_value" */;
```

VHDL 约束语法

```
attribute syn_dspstyle:string;
```

```
attribute syn_dspstyle of object:objectType is "setting_value";
```

注！

- object: 指定作用的对象，可以是 wire、register、module/entity 名或 module/entity instance 名。
- objectType: 为 object 的 type，如 signal。
- setting_value: 乘法器实现方式，当前支持 dsp、logic。
- setting_value 为 logic: 把 object 映射为逻辑电路。
- setting_value 为 dsp: 把 object 映射为 DSP，也是默认情况。

示例

GSC 约束示例

示例 1 指定 instance 实现方式为 logic

```
INS "temp" syn_dspstyle=logic;
```

```
INS "aa0/mult/c" syn_dspstyle=logic;
```

示例 2 指定全局所有乘法器实现方式为 logic

```
GLOBAL syn_dspstyle=logic;
```

Verilog 约束示例

示例 1 指定 mult module 中的所有乘法器实现方式为 logic

```
module mult(...) /* synthesis syn_dspstyle = "logic" */;
```

```
.....
```

```
wire [15:0] temp;
```

```
assign temp = a*b;
```

```
.....
```

```
endmodule
```

示例 2 指定乘法器 *temp* 实现方式为 *logic*

```

module mult(...);
.....
wire [15:0] temp/* synthesis syn_dspstyle = "logic" */;
assign temp = a*b;
.....
endmodule

```

VHDL 约束示例

示例 1 指定乘法器 *result* 实现方式为 *logic*

```

entity Mult is
port(
.....
    result : out signed(23 downto 0));
    attribute syn_dspstyle:string;
    attribute syn_dspstyle of result : signal is "logic";
end Mult;

architecture Behavior of Mult is
    signal x1 : signed(11 downto 0);
    signal y1 : signed(11 downto 0);
begin
.....
    result <= x1 * y1;
end Behavior;

```

5.6 syn_encoding

描述

指定状态机编译器的编码方式，只可以作用于具体对象。
该属性只可在 RTL 文件中指定。

语法

Verilog 语法

```
verilog object /* synthesis syn_encoding = "setting_value" */;
```

VHDL 约束语法

```
attribute syn_encoding : string;
```

```
attribute syn_encoding of object : objectType is "setting_value";
```

注!

- **object**: 指定作用的对象, verilog 只可以是 register 变量, VHDL 只可以是 type 类型变量。
- **objectType**: 为 object 的类型, 这里一般是 type。
- **setting_value**: 状态机的编码方式, verilog 当前支持的约束编码方式 one hot; VHDL 当前支持的约束编码方式有 one hot、gray。

示例

Verilog 示例

示例 1 指定状态机按照 onehot 码的方式来进行编码

```
module test (...);
  reg [2:0] ps /* synthesis syn_encoding="onehot " */;
  .....
endmodule
```

VHDL 示例

示例 1 指定状态机按照 onehot 码的方式来进行编码

```
ENTITY fsm IS
  .....
END fsm;
ARCHITECTURE behaviour OF fsm IS
  TYPE state_type IS (s0,s1,s2,s3);
  SIGNAL present_state,next_state : state_type;
  attribute syn_encoding:string;
  attribute syn_encoding of state_type:type is "onehot";
  BEGIN
  .....
  END behaviour;
```

5.7 syn_insert_pad

描述

指定是否插入 I/O buffer。当属性值为 1 时, 插入 I/O buffer。

该属性只可在 GSC 文件中指定。

语法

GSC 约束语法

```
PORT "object" syn_insert_pad=setting_value;
```

注!

- **setting_value**: 0 或者 1。为 0 时, 移除 I/O buffer; 为 1 时, 插入 I/O buffer。

- **object**: 只可以为 port, 此约束只对 Input port 或 Output port 起作用, 对 Inout port 不起作用。

示例

GSC 示例

示例 1 指定插入 I/O buffer

```
PORT "out" syn_insert_pad=1;
```

示例 2 指定移除 I/O buffer

```
PORT "out" syn_insert_pad=0;
```

5.8 syn_keep

描述

指定 wire 作为占位符而将其保留、不进行优化, 保留 reg 请使用“syn_preserve”。

该属性只可在 RTL 文件中指定。

语法

Verilog 约束语法

```
Verilog object /* synthesis syn_keep= setting_value */;
```

VHDL 约束语法

```
attribute syn_keep : integer;
```

```
attribute syn_keep of object : objectType is 1;
```

注!

- **object**: 指定作用的对象, 只能是 wire、port 和组合逻辑。
- **objectType**: 为 object 的 type, 如 signal。
- **setting_value**: 只能为 0 或者 1, 为 1 时保留该 net 不进行优化。

示例

Verilog 约束示例

示例 1 指定 mywire 不被优化掉

```
module test (...);
```

```
.....
```

```
wire mywire /* synthesis syn_keep=1 */;
```

```
.....
```

```
endmodule
```

VHDL 约束示例

示例 1 指定 tmp0 不被优化掉

```
entity mux2_1 is
```

```
port(
```

```

        .....
    );
end mux2_1;
architecture Behavioral of mux2_1 is
    signal tmp0:bit;
    signal tmp1:bit;
    attribute syn_keep : integer;
    attribute syn_keep of tmp0 : signal is 1;
    .....
end Behavioral;

```

5.9 syn_looplimit

描述

指定设计中循环迭代限制的值，默认循环迭代次数为 2000。设计中如果循环迭代次数超过 2000，但是没有指定循环次数，综合过程中会报错。

该属性只可在 GSC 中设置。

语法

GSC 约束语法

GLOBAL syn_looplimit=setting_value

注！

setting_value: 只可以为数字，代表此设计中循环迭代次数的上限。

示例

GSC 约束示例

GLOBAL syn_looplimit=3000

5.10 syn_maxfan

描述

指定最大扇出值，既可以作用于具体对象，也可以作用于全局范围。

该属性可在 GSC、RTL 文件中指定。

语法

GSC 约束语法

INS "object" syn_maxfan=setting_value;

NET "object" syn_maxfan=setting_value;

GLOBAL syn_maxfan=setting_value;

Verilog 约束语法

Verilog object */* synthesis syn_maxfan = setting_value */* ;

VHDL 约束语法

attribute syn_maxfan : integer;

attribute syn_maxfan of object : objectType is setting_value;

注!

- object: 指定作用的对象, 可以是 wire、register、input、output、module/entity 名、module/entity instance 名, 对 CLK、CE、LSR 相关 input 不起作用。
- objectType: 为 object 的 type, 如 signal。
- setting_value: 大于 0 的整数。

示例

GSC 示例

示例 1 指定 instance 的最大扇出值为 10

INS "d" syn_maxfan=10;

示例 2 指定全局的最大扇出值为 100

GLOBAL syn_maxfan=100;

示例 3 指定 instance 的最大扇出值为 10

INS "aa0/mult/d" syn_maxfan=10;

示例 4 指定 net 的最大扇出值为 10

NET "aa0/mult/d" syn_maxfan=10;

Verilog 示例

示例 1 指定 module 内除 CLK 外的所有 instance 的最大扇出值为 3

module test (...) / synthesis syn_maxfan = 3*;*

.....

endmodule

示例 2 指定 instance 的最大扇出值为 3

module test (...);

reg [7:0] d / synthesis syn_maxfan = 3*;*

.....

endmodule

VHDL 示例

entity test is

.....

end test;

architecture rtl of test is

signal d : std_logic;

```

attribute syn_maxfan : integer;
attribute syn_maxfan of d : signal is 5;
.....
end rtl;

```

5.11 syn_netlist_hierarchy

描述

指定是否生成的层级结构的网表。默认值为 1 表示生成层级结构网表；设置为 0 时，则将层级结构网表进行扁平化输出。

该属性可在 GSC、RTL 文件中指定。

语法

GSC 约束语法

```
GLOBAL syn_netlist_hierarchy=setting_value;
```

Verilog 约束语法

```
Verilog object /* synthesis syn_netlist_hierarchy=setting_value */;
```

VHDL 约束语法

```
attribute syn_netlist_hierarchy: integer;
```

```
attribute syn_netlist_hierarchy of object : objectType is setting_value;
```

注！

- object: 指定作用的对象，只能是 top module/entity。
- objectType: 为 object 的 type，如 entity。
- setting_value: 0 或者 1，为 1 时，允许生成 hierarchy；设置属性值为 0，则扁平化输出层级结构的网表。

示例

GSC 示例

```
GLOBAL syn_netlist_hierarchy=0;
```

Verilog 示例

```
module rp_top (...) /* synthesis syn_netlist_hierarchy=1 */;
```

.....

```
endmodule
```

VHDL 示例

```
entity mux4_1_top is
```

```
port(
```

```
    dina : in bit;
```

```
    dinb : in bit;
```

```
    sel  : in bit;
```

```

        dout : out bit
    );
    attribute syn_netlist_hierarchy: integer;
    attribute syn_netlist_hierarchy of mux4_1_top: entity is 0;
end mux4_1_top;

```

5.12 syn_noprune

描述

保证 module/entity instance 或者 primitive instance 或黑盒子（包括原语）的输出全部悬空时是否被优化掉，既可以作用于具体对象，也可以作用于全局范围。

该属性仅在 RTL 文件中指定。

语法

Verilog 约束语法

```
Verilog object /* synthesis syn_noprune = setting_value */;
```

VHDL 约束语法

```
attribute syn_noprune : integer;
```

```
attribute syn_noprune of object: objectType is 1;
```

注！

- object: 指定作用的对象，可以是 module/entity instance 名或者 primitive instance 名及黑盒子。
- objectType: 为 object 的 type，如 signal。
- setting_value: 只可以为 0 或者 1。为 1 时保留 instance 及黑盒子；为 0 时根据需要优化相应的 instance 及黑盒子。

示例

Verilog 约束示例

```

module test (out1,out2,clk,in1,in2);
.....
noprune_bb u1(out1,in1)/*synthesis syn_noprune=1*/;
.....
endmodule

module noprube_bb(din,dout);
input din;
output dout;
endmodule

```

VHDL 约束示例

```
library ieee;
```

```

use ieee.std_logic_1164.all;
entity top is
.....
end entity top;
architecture arch of top is
component noprunereg_bb
port(
din : in std_logic;
dout : out std_logic);
end component noprunereg_bb;
signal o1_noprunereg : std_logic;
signal o2_reg : std_logic;
attribute syn_noprune : integer;
attribute syn_noprune of U1: label is 1;
attribute syn_noprune of o1_noprunereg : signal is 1;
.....
end architecture arch;

```

5.13 syn_preserve

描述

指定寄存器或寄存器逻辑是否被优化，既可以作用于具体对象，也可以作用于全局范围。

该属性可在 RTL 文件及 GSC 文件中指定。

语法

GSC 约束语法

```
INS "object" syn_preserve=setting_value;
```

```
GLOBAL syn_preserve=setting_value;
```

Verilog 约束语法

```
Verilog object /* synthesis syn_preserve = setting_value */;
```

VHDL 约束语法

```
attribute syn_preserve : integer;
```

```
attribute syn_preserve of object : objectType is setting_value;
```

注!

- object: 指定作用的对象, 可以是 register 名、module /entity 名及 module /entity instance 名。
- objectType: 为 object 的 type, 如 signal。

- `setting_value`: 0 或者 1。为 1 时保留对应寄存器；为 0 时根据需要优化对应的寄存器。

示例

GSC 示例

示例 1 指定保留 `reg1` 不被优化掉

```
INS "reg1" syn_preserve =1;
```

示例 2 指定保留设计中的所有寄存器

```
GLOBAL syn_preserve =1;
```

Verilog 示例

示例 1 指定保留 `module` 中的所有寄存器

```
module test (...) /* synthesis syn_preserve = 1 */;
```

.....

```
endmodule
```

示例 2 指定保留 `reg1` 不被优化掉

```
module test (...);
```

.....

```
reg reg1/* synthesis syn_preserve = 1 */;
```

.....

```
endmodule
```

VHDL 示例

示例 1 指定保留寄存器 `reg1`

```
entity syn_test is
```

```
    port (.....
```

```
    );
```

```
end syn_test;
```

```
architecture behave of syn_test is
```

```
    signal reg1 : std_logic;
```

```
    signal reg2 : std_logic;
```

```
    attribute syn_preserve : integer;
```

```
    attribute syn_preserve of reg1: signal is 1;
```

```
begin
```

.....

```
end behave;
```

5.14 syn_probe

描述

该属性通过插入探测点对设计中的内部信号进行测试和调试。被指定的探测点将以 **port** 的形式出现在顶层端口列表中。

该属性只可在 RTL 文件中指定。

语法

Verilog 约束语法

```
Verilog object /* synthesis syn_probe = setting_value */ ;
```

VHDL 约束语法

```
attribute syn_probe: string;
```

```
attribute syn_probe of object: objectType is " setting_value ";
```

注!

- **object**: 指定作用的对象，只可以是 **wire** 或 **register**。
- **objectType**: 为 **object** 的 **type**，如 **signal**。
- **setting_value** 值为 1: 插入探测点，根据 **net** 的名称自动得到探测 **port** 的名称。
- **setting_value** 值为 0: 不允许探测。
- **setting_value** 值为字符串: 插入一个指定名字的探测点。当 **setting_value** 指定的名字为 **bus** 时，插入的名称后会自动添加数字。
- **gowinSyn** 不支持 **setting_value** 的值与 **object** 名字或者 **module** 的 **port** 名相同。

示例

Verilog 约束示例，**probe_tmp** 被设置该属性后，**probe_tmp** 将被列在顶层输出端口列表中。

```
module test (...);
```

```
.....
```

```
reg [7:0] probe_tmp /* synthesis syn_probe=1*/;
```

```
.....
```

```
endmodule
```

VHDL 约束示例

```
entity halfadd is
```

```
port(.....);
```

```
end halfadd;
```

```
architecture add of halfadd is
```

```
    signal probe_tmp: std_logic;
```

```
    attribute syn_probe: string;
```

```
    attribute syn_probe of probe_tmp: signal is "probe_string";
```

```
.....
```

```
end;
```

5.15 syn_ramstyle

描述

指定存储器的实现方式，既可以作用于具体对象，也可以作用于全局范围。

该属性可在 GSC、RTL 文件中指定。

语法

GSC 约束语法

```
INS "object" syn_ramstyle =setting_value;
```

```
GLOBAL syn_ramstyle =setting_value;
```

Verilog 约束语法

```
Verilog object /* synthesis syn_ramstyle = "setting_value" */;
```

VHDL 约束语法

```
attribute syn_ramstyle:string;
```

```
attribute syn_ramstyle of object : objectType is "setting_value";
```

注！

- object: 指定作用的对象, 可以是 module/entity 名、module/entity instance 名或 register。
- objectType: 为 object 的 type, 如 signal。
- setting_value: 存储器的实现方式, 当前支持 block_ram、distributed_ram、registers、rw_check、no_rw_check。
- setting_value 为 registers: 将 inferred RAM 映射为 registers (触发器和逻辑电路), 而不是专用的 RAM 资源。
- setting_value 为 block_ram: 将 inferred RAM 映射为适当的设备专用内存, 其使用 FPGA 的专用内存资源。

示例

GSC 约束示例

示例 1 指定 instance 实现方式为 BSRAM

```
INS "mem" syn_ramstyle=block_ram;
```

示例 2 指定全局所有存储器实现方式为 SSRAM

```
GLOBAL syn_ramstyle=distributed_ram;
```

Verilog 约束示例

示例 1 指定 module 内所有的存储器实现方式为 block_ram, 并且不进行读写检查

```
module test (...) /* synthesis syn_ramstyle = "block_ram" */;
```

```
.....
```

```
endmodule
```

示例 2 指定 instance 实现方式为 BSRAM 硬核

```

    module test (...);
    .....
    reg [DATA_W - 1 : 0] mem [(2**ADDR_W) - 1 : 0] /* synthesis
syn_romstyle = "block_ram" */;
    .....
    endmodule

```

VHDL 约束示例

示例 1 指定存储器 *memory* 实现方式为 BSRAM

```

entity ram is
    GENERIC(bits:INTEGER:=8;
            words:INTEGER:=256);
    PORT(.....);
end ram;

ARCHITECTURE arch of ram IS
    TYPE vector_array IS ARRAY(0 TO words-1) OF
STD_LOGIC_VECTOR(bits-1 DOWNT0 0);
    SIGNAL memory:vector_array ;
        attribute syn_romstyle:string;
        attribute syn_romstyle of memory : signal is " block_ram";
    BEGIN
    .....
end arch;

```

5.16 syn_romstyle

描述

指定只读存储器的实现方式，既可以作用于具体对象，也可以作用于全局范围。

该属性可在 GSC、RTL 文件中指定。

语法

GSC 约束语法

```
INS "object" syn_romstyle =setting_value;
```

```
GLOBAL syn_romstyle =setting_value;
```

Verilog 约束语法

```
Verilog object /* synthesis syn_romstyle = "setting_value" */ ;
```

VHDL 约束语法


```
attribute syn_romstyle:string;
attribute syn_romstyle of object : objectType is "setting_value";
```

注！

- **object**: 指定作用的对象, 可以是 module/entity 名, module/entity instance 名或 register。
- **objectType**: 为 object 的 type, 如 signal。
- **setting_value**: 只读存储器的实现方式, 当前支持 block_rom, distributed_rom, logic。

示例

GSC 约束示例

示例 1 指定 instance 实现方式为 BSRAM

```
INS "mem" syn_romstyle=block_rom;
```

示例 2 指定全局所有存储器实现方式为 SSRAM

```
GLOBAL syn_romstyle=distributed_rom;
```

Verilog 约束示例

示例 1 指定 module 内的所有存储器实现方式为 SSRAM

```
module rom16_test(...)/*synthesis syn_romstyle="distributed_rom"*/;
```

.....

```
endmodule
```

VHDL 约束示例

示例 1 指定 module 内的所有存储器实现方式为 SSRAM

```
ENTITY rom is
```

.....

```
end rom;
```

```
ARCHITECTURE rom OF rom IS
```

```
    signal data_out :STD_LOGIC_VECTOR(bits-1 DOWNT0 0);
```

```
    attribute syn_romstyle:string;
```

```
    attribute syn_romstyle of data_out : signal is "block_rom";
```

.....

```
END rom;
```

5.17 syn_srlstyle

描述

指定 shift registers 的实现方式, 既可以作用于具体对象, 也可以作用于全局范围。shift register 可以使用 BSRAM、SSRAM、registers、bsram_sdp 来实现, 默认情况下是根据 shift register 中 register 的数量来决定使用哪种实现方式, 使用 syn_srlstyle 可以修改默认值。

该属性可在 GSC、RTL 文件中指定。

语法

GSC 约束语法

```
INS "object" syn_srlstyle =setting_value
```

```
GLOBAL syn_srlstyle =setting_value
```

Verilog 约束语法

```
Verilog object /* synthesis syn_srlstyle = "setting_value" */ ;
```

VHDL 约束语法

```
attribute syn_srlstyle:string;
```

```
attribute syn_srlstyle of object : objectType is "setting_value";
```

注!

- object: 指定作用的对象, 可以是 module/entity、module/entity instance 或 register, GSC 语法中不支持 module/entity。
- objectType: 为 object 的 type, 如 signal。
- setting_value: 存储器的实现方式, 当前支持 block_ram、distributed_ram、registers、bsram_sdp, 指定值为 bsram_sdp 时移位寄存器不 infer 出 SP。

示例

GSC 约束示例

示例 1. 指定 instance 实现方式为 BSRAM

```
INS "mem" syn_srlstyle=block_ram
```

示例 2. 指定全局所有存储器实现方式为 SSRAM

```
GLOBAL syn_srlstyle=distributed_ram
```

Verilog 约束示例

示例 1 指定 module 内寄存器实现方式为 block_ram

```
module test (...) /* synthesis syn_srlstyle = "block_ram" */;
```

```
.....
```

```
endmodule
```

示例 2 指定 instance 实现方式为 BSRAM

```
module test (...);
```

```
.....
```

```
reg [SRL_WIDTH-1:0] regBank[SRL_DEPTH-1:0]/* synthesis  
syn_srlstyle = "block_ram" */;
```

```
.....
```

```
endmodule
```

VHDL 约束示例

示例 1 指定 module 内寄存器实现方式为 register

```
entity ram is
```

```

    GENERIC(bits:INTEGER:=8;
            words:INTEGER:=256);
    PORT(.....);
    attribute syn_srlstyle:string;
    attribute syn_srlstyle of shiftreg : entity is "registers";
    end ram;

    ARCHITECTURE arch of ram IS
    .....
    end ram;

```

5.18 syn_tlvds_io/syn_elvds_io

描述

指定差分的 I/O buffer 映射的属性，既可以作用于具体对象，也可以作用于全局范围。该属性可在 GSC、RTL 文件中指定。

语法

GSC 约束语法

```
PORT "object" syn_tlvds_io =setting_value;
```

```
GLOBAL syn_tlvds_io =setting_value;
```

```
PORT "object" syn_elvds_io =setting_value;
```

```
GLOBAL syn_elvds_io =setting_value;
```

Verilog 约束语法

```
Verilog object /* synthesis syn_tlvds_io = setting_value */ ;
```

VHDL 约束语法

```
attribute syn_tlvds_io: integer;
```

```
attribute syn_tlvds_io of object: objectType is setting_value;
```

注！

- object: 指定作用的对象，可以是 module/entity 名，port 名。
- objectType: 为 object 的 type，如 signal。
- setting_value: 0 或者 1。

示例

GSC 约束示例

示例 1 指定 buffer 实现方式为 TLVDS

```
PORT "io" syn_tlvds_io =1;
```

```
PORT "iob" syn_tlvds_io =1;
```

示例 2 指定全局所有 buffer 实现方式为 TLVDS

```
GLOBAL syn_tlvs_io =1;
```

Verilog 约束示例

```
module elvds_iobuf(io,iob...);
inout io/*synthesis syn_elvds_io=1*/;
inout iob/*synthesis syn_elvds_io=1*/;
.....
endmodule
```

VHDL 约束示例

```
entity test is
port (in1_p : in std_logic;
in1_n : in std_logic;
clk : in std_logic;
out1 : out std_logic;
out2 : out std_logic);
attribute syn_tlvs_io: integer;
attribute syn_tlvs_io of in1_p,in1_n,out1,out2: signal is 1;
end test;
architecture arch of test is
.....
end arch
```

5.19 translate_off/translate_on

描述

translate_off/translate_on 必须成对出现，translate_off 之后的语句将在综合过程中被跳过，直到 translate_on 出现，常用于在综合时自动屏蔽一些语句。

该属性只可在 RTL 文件中指定。

语法

Verilog 约束语法

```
/* synthesis translate_off*/ ;
```

综合过程中被忽略的语句

```
/* synthesis translate_on*/
```

VHDL 约束语法

```
-- synthesis translate_off
```

综合过程中被忽略的语句

```
-- synthesis translate_on
```

示例

Verilog 约束示例

示例 1 `/*synthesis translate_off*/`与`/*synthesis translate_on*/`之间的 `assign Nout =a*b` 语句在综合过程中被忽略，不进行综合

```
module test (...);  
.....  
/*synthesis translate_off*/  
assign my_ignore=a*b;  
/*synthesis translate_on*/  
.....  
endmodule
```

VHDL 约束示例

```
entity top is  
port (  
.....  
);  
end top;  
architecture rtl of top is  
begin  
dout <= a + b;  
-- synthesis translate_off  
Nout <= a * b;  
-- synthesis translate_on  
end rtl;
```

6 Report 用户文档

report 文档是在执行综合之后，生成的统计报告文件，文件名为 *_syn.rpt.html(*为指定输出网表 vg 文件的名称), 包含 Synthesis Message、Synthesis Details、Resource、Timing。

6.1 Synthesis Message

Synthesis Message，即综合基本信息。主要包括综合的设计文件、当前 GowinSynthesis 版本、设计配置信息和运行时间等信息，如图 6-1 所示。

图 6-1 Synthesis Message

Synthesis Messages	
Report Title	GowinSynthesis Report
Design File	/n9k/share/gwsw/sw_pub/testcase/gw1nsr-2/SYN/rom16_case/src/rom_bp_async_rst_addr_5_dout_35.v
GowinSynthesis Constraints File	---
Tool Version	V1.9.9.03
Part Number	GW1NSR-LV4CQN48GC6/I5
Device	GW1NSR-4C
Created Time	Tue Apr 23 15:20:27 2024
Legal Announcement	Copyright (C)2014-2024 Gowin Semiconductor Corporation. ALL rights reserved.

6.2 Synthesis Details

Synthesis Details，该标题下打印设计文件的顶层模块、综合各阶段的运行时间和占用内存、总运行时间和总占用内存，如图 6-2 所示。

图 6-2 Synthesis Details

Synthesis Details	
Top Level Module	top
Synthesis Process	Running parser: CPU time = 0h 0m 0.109s, Elapsed time = 0h 0m 0.123s, Peak memory usage = 52.926MB Running netlist conversion: CPU time = 0h 0m 0s, Elapsed time = 0h 0m 0s, Peak memory usage = 52.992MB Running device independent optimization: Optimizing Phase 0: CPU time = 0h 0m 0s, Elapsed time = 0h 0m 0s, Peak memory usage = 53.133MB Optimizing Phase 1: CPU time = 0h 0m 0s, Elapsed time = 0h 0m 0s, Peak memory usage = 53.191MB Optimizing Phase 2: CPU time = 0h 0m 0s, Elapsed time = 0h 0m 0s, Peak memory usage = 53.254MB Running inference: Inferring Phase 0: CPU time = 0h 0m 0s, Elapsed time = 0h 0m 0s, Peak memory usage = 53.344MB Inferring Phase 1: CPU time = 0h 0m 0s, Elapsed time = 0h 0m 0s, Peak memory usage = 53.402MB Inferring Phase 2: CPU time = 0h 0m 0s, Elapsed time = 0h 0m 0s, Peak memory usage = 53.430MB Inferring Phase 3: CPU time = 0h 0m 0s, Elapsed time = 0h 0m 0s, Peak memory usage = 53.441MB Running technical mapping: Tech-Mapping Phase 0: CPU time = 0h 0m 0s, Elapsed time = 0h 0m 0s, Peak memory usage = 53.465MB Tech-Mapping Phase 1: CPU time = 0h 0m 0s, Elapsed time = 0h 0m 0s, Peak memory usage = 53.465MB Tech-Mapping Phase 2: CPU time = 0h 0m 0s, Elapsed time = 0h 0m 0s, Peak memory usage = 53.473MB Tech-Mapping Phase 3: CPU time = 0h 0m 0.039s, Elapsed time = 0h 0m 0.028s, Peak memory usage = 54.289MB Tech-Mapping Phase 4: CPU time = 0h 0m 0s, Elapsed time = 0h 0m 0s, Peak memory usage = 54.289MB Generate output files: CPU time = 0h 0m 0s, Elapsed time = 0h 0m 0.008s, Peak memory usage = 57.043MB
Total Time and Memory Usage	CPU time = 0h 0m 0.148s, Elapsed time = 0h 0m 0.159s, Peak memory usage = 57.043MB

6.3 Resource

Resource，即资源信息。主要包括资源使用统计和芯片占用统计。

资源使用表会统计用户设计中 I/O Port、I/O Buf、REG、LUT 等的数量。资源利用率表用于预估用户设计中 CFU Logics、Register、BSRAM、DSP 等在当前器件的资源占用率，如图 6-3 所示。

图 6-3 Resource

Resource		
Resource Usage Summary		
Resource	Usage	
I/O Port	32	
I/O Buf	32	
IBUF	24	
OBUF	8	
Register	1032	
DFFE	1032	
LUT	680	
LUT2	128	
LUT3	512	
LUT4	40	
INV	1	
INV	1	
Resource Utilization Summary		
Resource	Usage	Utilization
Logic	681(681 LUT, 0 ALU) / 4608	15%
Register	1032 / 4020	26%
--Register as Latch	0 / 4020	0%
--Register as FF	1032 / 4020	26%
BSRAM	0 / 10	0%

6.4 Timing

Timing，即时序统计。主要包括 Clock Summary、Max Frequency Summary、Detail Timing Paths Informations 等信息。

Clock Summary 主要描述网表的时钟信号信息，如下图 6-4 所示，给出一个时钟，频率为 100MHz，周期为 10ns，0ns 时为上升沿，5ns 时为下降沿。

图 6-4 Timing

Timing									
Clock Summary:									
NO.	Clock Name	Type	Period	Frequency(MHz)	Rise	Fall	Source	Master	Object
1	clk	Base	10.000	100.0	0.000	5.000			clk_ibuf/l

Max Frequency Summary 主要统计网表文件可以达到的时钟频率，并以此来衡量整个网表文件的时序是否达到要求。如下图 6-5 中所示。要求频率为 100MHz，实际时钟频率为 747.2MHz，满足时序要求。如果实际频率未达到要求频率，则不满足时序要求，需要进一步查看具体时序路径。

图 6-5 Max Frequency Summary

Max Frequency Summary:

No.	Clock Name	Constraint	Actual Fmax	Logic Level	Entity
1	clk	100.0(MHz)	747.2(MHz)	1	TOP

Detail Timing Paths Information 显示时序路径详细信息，默认为 5 条，所有时间值的默认单位均为纳秒。Path Summary 主要描述网表文件中的关键时序路径，起始节点及相关时延信息，如图 6-6 所示。Data Arrival Path 和 Data Require Path 主要描述关键时序路径，给出详细的连接关系，扇出信息如图 6-7 所示。Path Statistics 描述路径的时延信息，如图 6-8 所示。

图 6-6 Path Summary

Detail Timing Paths Information

Path 1

Path Summary:

Slack	8.662
Data Arrival Time	2.283
Data Required Time	10.945
From	reg2_s0
To	out2
Launch Clk	clk[R]
Latch Clk	clk[R]

图 6-7 连接关系、时延及扇出信息

Data Arrival Path:

AT	DELAY	TYPE	RF	FANOUT	NODE
0.000	0.000				clk
0.000	0.000	tCL	RR	1	clk_ibuf/I
0.982	0.982	tINS	RR	3	clk_ibuf/O
1.345	0.363	tNET	RR	1	reg2_s0/CLK
1.803	0.458	tC2Q	RF	3	reg2_s0/Q
2.283	0.480	tNET	FF	1	out2/D

Data Required Path:

AT	DELAY	TYPE	RF	FANOUT	NODE
10.000	0.000				clk
10.000	0.000	tCL	RR	1	clk_ibuf/I
10.982	0.982	tINS	RR	3	clk_ibuf/O
11.345	0.363	tNET	RR	1	out2/CLK
10.945	-0.400	tSu		1	out2

图 6-8 Path Statistics**Path Statistics:**

Clock Skew:	0.000
Setup Relationship:	10.000
Logic Level:	1
Arrival Clock Path Delay:	cell: 0.982, 73.009%; route: 0.363, 26.991%
Arrival Data Path Delay:	cell: 0.000, 0.000%; route: 0.480, 51.155%; tC2Q: 0.458, 48.845%
Required Clock Path Delay:	cell: 0.982, 73.009%; route: 0.363, 26.991%

