Gowin HDL 编码风格

用户指南

SUG949-1.7.1, 2024-10-25

免责声明

本文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
2020/08/31	1.0	初始版本。
2021/06/10	1.1	添加单端 BUF 的 infer 案例。
2022/05/31	1.2	删除编码风格要求描述。
2023/04/20	1.3	新增第 7 章 Arora V DSP 编码规范。
2023/06/30	1.4	更新第 4 章 BSRAM 编码规范和第 5 章 SSRAM 编码规范。
2023/11/30	1.5	更新第 4 章 BSRAM 编码规范和第 5 章 SSRAM 编码规范。
2024/03/29	1.6	更新第 4 章 BSRAM 编码规范。
2024/06/28	1.7	更新 3.1 节 LUT 描述。
2024/10/25	1.7.1	更新 4.1.2 节读地址不经过 Register，输出经过 Register 中的案例。

目录

表目录

1 关于本手册

1.1 手册内容

本手册主要描述高云 HDL 编码风格要求及原语的 HDL 编码实现，旨在帮助用户快速熟悉高云 HDL 编码风格和原语实现，指导用户设计，提高设计效率。

1.2 相关文档

通过登录高云半导体网站 www.gowinsemi.com.cn 可下载、查看以下相关文档：

- SUG100, Gowin 云源软件用户指南
- SUG283, Gowin 原语用户指南

1.3 术语、缩略语

表 1-1 中列出了本手册中出现的相关术语、缩略语及相关释义。

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
CLU	Configurable Logic Unit	可配置逻辑单元
DSP	Digital Signal Processing	数字信号处理
FSM	Finite State Machine	有限状态机
HDL	Hardware Description Language	硬件描述语言
LUT	Look-up Table	查找表
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 Buffer 编码规范

Buffer 缓冲器，具有缓存功能。根据不同功能，可分为单端 buffer、模拟 LVDS（ELVDS）和真 LVDS（TLVDS）。模拟 LVDS 和真 LVDS 的原语实现需添加相应的属性约束，建议采用实例化方式编码。

2.1 IBUF

IBUF(Input Buffer)，输入缓冲器。其编码形式可采用如下 2 种方式：

方式 1：

```
module ibuf (o, i);
input i ;
output o ;
assign o = i ;
endmodule
```

方式 2：

```
module ibuf (o, i);
input i ;
output o ;
buf IB (o, i);
endmodule
```

2.2 TLVDS_IBUF

TLVDS_IBUF(True LVDS Input Buffer)，真差分输入缓冲器。该原语的实现需要添加属性约束，其编码形式可如下所示：

```
module tlvds_ibuf_test(in1_p, in1_n,   out);
input in1_p/* synthesis syn_tlvds_io = 1*/;
input in1_n/* synthesis syn_tlvds_io = 1*/;
output reg out;
```

```
always@(in1_p or in1_n) begin
    if (in1_p != in1_n) begin
        out = in1_p;
    end
end
endmodule
```

2.3 ELVDS_IBUF

ELVDS_IBUF(Emulated LVDS Input Buffer)，模拟差分输入缓冲器。该原语的实现需要添加属性约束，其编码形式可如下所示：

```
module elvds_ibuf_test (in1_p, in1_n, out);
input in1_p/* synthesis syn_elvds_io = 1*/;
input in1_n/* synthesis syn_elvds_io = 1*/;
output reg out;
always@(in1_p or in1_n)begin
    if (in1_p != in1_n) begin
        out = in1_p;
    end
end
endmodule
```

2.4 OBUF

OBUF(Output Buffer)，输出缓冲器。其编码形式可采用如下 2 种方式：

方式 1：

```
module obuf (o, i);
input i ;
output o ;
assign o = i ;
endmodule
```

方法 2：

```
module obuf (o, i);
input i ;
output o ;
buf OB (o, i);
endmodule
```

2.5 TLVDS_OBUF

TLVDS_OBUF(True LVDS Output Buffer)，真差分输出缓冲器。该原语的实现需要添加属性约束，其编码形式可如下所示：

```
module tlvds_obuf_test(in,out1,out2);
input in;
output out1/* synthesis syn_tlvds_io = 1*/;
output out2/* synthesis syn_tlvds_io = 1*/;
assignout1 = in;
assign out2 = ~out1;
endmodule
```

2.6 ELVDS_OBUF

ELVDS_OBUF(Emulated LVDS Output Buffer)，模拟差分输出缓冲器。该原语的实现需要添加属性约束，其编码形式可如下所示：

```
module elvds_obuf_test(in,out1,out2);
input in;
output out1/* synthesis syn_elvds_io = 1*/;
output out2/* synthesis syn_elvds_io = 1*/;
assign out1= in;
assign out2 = ~out1;
endmodule
```

2.7 TBUF

TBUF(Output Buffer with Tri-state Control)，三态缓冲器，低电平使能。其编码形式可采用如下 2 种方式：

方式 1：

```
module tbuf (in, oen, out);
input in, oen;
output out;
assign out= ~oen ? in :1'bz;
endmodule
```

方式 2：

```
module tbuf (out, in, oen);
input in, oen;
output out;
```

```
bufif0 TB (out, in, oen);
endmodule
```

2.8 TLVDS_TBUF

TLVDS_TBUF(True LVDS Tristate Buffer)，真差分三态缓冲器，低电平使能。该原语的实现需要添加属性约束，其编码形式可如下所示：

```
module tlvds_tbuf_test(in, oen, out1,out2);
input in;
input oen;
output out1/* synthesis syn_tlvds_io = 1*/;
output out2/* synthesis syn_tlvds_io = 1*/;
assign out1 = ~oen ? in : 1'bz;
assign out2 = ~oen ? ~in : 1'bz;
endmodule
```

2.9 ELVDS _TBUF

ELVDS_TBUF(Emulated LVDS Tristate Buffer)，模拟差分三态缓冲器，低电平使能。该原语的实现需要添加属性约束，其编码形式可如下所示：

```
module elvds_tbuf_test(in, oen, out1,out2);
input in, oen;
output out1/* synthesis syn_elvds_io = 1*/;
output out2/* synthesis syn_elvds_io = 1*/;
assign out1 = ~oen ? in : 1'bz;
assign out2 = ~oen ? ~in : 1'bz;
endmodule
```

2.10 IOBUF

IOBUF(Bi-Directional Buffer)，双向缓冲器。当 OEN 为高电平时，作为输入缓冲器；OEN 为低电平时，作为输出缓冲器。其编码形式可采用如下 2 种方式：

方式 1：

```
module iobuf (in, oen,   io, out);
input in,oen;
output out;
inout io;
assign io= ~oen ? in :1'bz;
```

```
assign out = io;
endmodule
```

方式 2：

```
module iobuf (out, io, i, oen);
input i,oen;
output out;
inout io;
buf OB (out, io);
bufif0 IB (io,i,oen);
endmodule
```

2.11 TLVDS_IOBUF

TLVDS_IOBUF(True LVDS Bi-Directional Buffer)，真差分双向缓冲器，当 OEN 为高电平时，作为真差分输入缓冲器；OEN 为低电平时，作为真差分输出缓冲器。该原语的实现需要添加属性约束，其编码形式可如下所示：

```
module tlvds_iobuf(o, io, iob, i, oen);
output reg o;
inout io /* synthesis syn_tlvds_io = 1 */;
inout iob /* synthesis syn_tlvds_io = 1 */;
input i, oen;
bufif0 ib(io, i, oen);
notif0 yb(iob, i, oen);
always @(io or iob)begin
    if (io != iob)begin
        o <= io;
    end
end
endmodule
```

2.12 ELVDS _IOBUF

ELVDS_IOBUF(Emulated LVDS Bi-Directional Buffer)，模拟差分双向缓冲器，当 OEN 为高电平时，作为模拟差分输入缓冲器；OEN 为低电平时，作为模拟差分输出缓冲器。该原语的实现需要添加属性约束，其编码形式可如下所示：

```
module elvds_iobuf(o, io, iob, i, oen);
output o;
```

```
inout io /* synthesis syn_elvds_io = 1 */;
inout iob /* synthesis syn_elvds_io = 1 */;
input i, oen;
reg o;
bufif0 ib(io, i, oen);
notif0 yb(iob, i, oen);
always @(io or iob)begin
    if (io != iob)begin
        o <= io;
    end
end
endmodule
```

3 CLU 编码规范

可配置逻辑单元 CLU（Configurable Logic Unit）是构成 FPGA 产品的基本单元，CLU 模块可实现 MUX/LUT/ALU/FF/LATCH 等模块的功能。

3.1 LUT

输入查找表 LUT，常用的 LUT 原语有 LUT2、LUT3、LUT4，其区别在于查找表输入位宽的不同，其实现方式可采用如下几种。

3.1.1 查找表形式

```
module rtl_LUT4 (f, i0, i1,i2,i3);
parameter INIT = 16'h2345;
input i0, i1, i2,i3;
output f;
assign f=INIT[{i3,i2,i1,i0}];
endmodule
```

3.1.2 选择器形式

```
module rtl_LUT3 (f,a,b,sel);
input a,b,sel;
output f;
assign f=sel?a:b;
endmodule
```

3.1.3 逻辑运算形式

```
module top(a,b,c,d,out);
input [3:0]a,b,c,d;
output [3:0]out;
```

```
assign out=a&b&c|d;
endmodule
```

3.2 ALU

ALU（2-input Arithmetic Logic Unit）2 输入算术逻辑单元，综合工具可以综合出 ADD/SUB/ADDSUB/NE 等功能。

3.2.1 ADD 功能

以 4 位全加器和 4 位半加器为例介绍 ALU 的 ADD 功能：

4 位全加器

4 位全加器其编码形式可如下所示：

```
module add(a,b,cin,sum,cout);
input [3:0] a,b;
input cin;
output [3:0] sum;
output cout;
assign {cout,sum}=a+b+cin;
endmodule
```

4 位半加器

4 位半加器其编码形式可如下所示：

```
module add(a,b,sum,cout);
input [3:0] a,b;
output [3:0] sum;
output cout;
assign {cout,sum}=a+b;
endmodule
```

3.2.2 SUB 功能

```
module sub(a,b,sub);
input [3:0] a,b;
output [3:0] sub;
assign sub=a-b;
endmodule
```

3.2.3 ADDSUB 功能

```
module addsub(a,b,c,sum);
input [3:0] a,b;
input c;
output [3:0] sum;
assign sum=c?(a-b):(a+b);
endmodule
```

3.2.4 NE 功能

```
module ne(a, b, cout);
input [11:0] a, b;
output cout;
assign cout = (a != b) ? 1'b1 : 1'b0;
endmodule
```

3.3 FF

触发器是时序电路中常用的基本元件，FPGA 内部的时序逻辑都可通过 FF 结构实现，常用的 FF 有 DFF、DFFE、DFFS、DFFSE 等，其区别在于复位方式、触发方式等方面。Arora V 的触发器原语仅只有 DFFSE、DFFRE、DFFPE、DFFCE。本节以 DFFSE、DFFRE、DFFPE、DFFCE 为例介绍寄存器的实现，其他寄存器类型原语的实现可参考这几类寄存器。编码定义 reg 信号时，建议添加初始值，不同类型寄存器的初始值可参考 *UG288，Gowin 可配置功能单元(CFU)用户指南*。

3.3.1 DFFSE

```
module dffse_init1 (clk, d, ce, set, q );
input clk, d, ce, set;
output   reg q=1'b1;
always @(posedge clk)begin
    if (set)begin
        q <= 1'b1;
    end
    else begin
        if (ce)begin
            q <= d;
        end
```

```
        end
    end
endmodule
```

3.3.2 DFFRE

```
module dffre_init1 (clk, d, ce, rst, q );
input clk, d, ce, rst;
output   reg q= 1'b0;
always @(posedge clk)begin
        if (rst)begin
                q <= 1'b0;
        end
        else begin
                if (ce)begin
                        q <= d;
                end
        end
    end
endmodule
```

3.3.3 DFFPE

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

3.3.4 DFFCE

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

3.4 LATCH

锁存器是一种对电平触发的存储单元电路，其可在特定输入电平作用下改变状态。Arora V 的锁存器原语只有 DLCE、DLPE。本节以 DLCE、DLPE 为例介绍锁存器的实现，其他锁存器类型原语的实现可参考这几类锁存器。编码定义 reg 信号时，建议添加初始值，不同类型锁存器的初始值可参考 *UG288，Gowin 可配置功能单元(CFU)用户指南*。

3.4.1 DLCE

```
module rtl_DLCE (Q, D, G, CE, CLEAR);
input D, G, CLEAR, CE;
output reg Q=1'b0;
always @(D or G or CLEAR or CE ) begin
    if (CLEAR)begin
        Q <= 1'b0;
    end
    else begin
        if (G && CE)begin
            Q <= D;
        end
```

```
    end
end
endmodule
```

3.4.2 DLPE

```
module rtl_DLPE (Q, D, G, CE, PRESET);
input D, G, PRESET, CE;
output reg Q= 1'b1;
always @(D or G or PRESET or CE ) begin
    if(PRESET)begin
        Q <= 1'b1;
    end
    else begin
        if (G && CE)begin
            Q <= D;
        end
    end
end
endmodule
```

4 BSRAM 编码规范

BSRAM 块状静态随机存储器，具有静态存取功能。根据配置模式，可分为单端口模式（SP/SPX9）、双端口模式（DPB/DPX9B）、伪双端口（SDPB/SDPX9B）和只读模式（pROM/pROMX9）。

4.1 DPB/DPX9B

DPB/DPX9B 的存储空间分别为 16Kbit/18Kbit，其工作模式为双端口模式，端口 A 和端口 B 均可分别独立实现读/写操作，可支持 2 种读模式（bypass 模式和 pipeline 模式）和 2 种写模式（normal 模式、write-through 模式）。本节以读地址是否经过 register、初值读取等方面进行实现说明。

4.1.1 读地址经过 register

读地址经过 register 时，仅支持读地址 register 无控制信号控制的情况。以 write-through 、bypass、同步复位模式的 DPB 为例介绍其实现，其编码形式可如下所示：

```
module top(data_outa, data_ina, addra, clka,cea, wrea,data_outb, data_inb, addrb, clkb,ceb, wreb);
output [7:0]data_outa,data_outb;
input [7:0]data_ina,data_inb;
input [10:0]addra,addrb;
input clka,wrea,cea;
input clkb,wreb,ceb;
reg [7:0] mem [2047:0];
reg [10:0]addra_reg,addrb_reg;

always@(posedge clka)begin
    addra_reg<=addra;
end
```

```
always @(posedge clka)begin
    if (cea & wrea) begin
        mem[addra] <= data_ina;
    end
end
assign data_outa = mem[addra_reg];

always@(posedge clkb)begin
    addrb_reg<=addrb;
end



always @(posedge clkb)begin
    if (ceb & wreb) begin
        mem[addrb] <= data_inb;
    end
end
assign data_outb = mem[addrb_reg];
endmodule
```

4.1.2 读地址不经过 Register，输出经过 Register

读地址不经过 register 时，输出必须经过 register，经过一级 register 时为 bypass 模式；经过两级 register 时为 pipeline 模式。以 normal、pipeline、同步复位模式的 DPB 为例介绍其实现，其编码形式可如下所示：

```
module top(data_outa, data_ina, addra, clka, cea, ocea, wrea, rsta,
data_outb, data_inb, addrb, clkb, ceb, oceb, wreb, rstb);
output reg [15:0]data_outa,data_outb;
input [15:0]data_ina,data_inb;
input [9:0]addra,addrb;
input clka,wrea,cea,ocea,rsta;
input clkb,wreb,ceb,oceb,rstb;
reg [15:0] mem [1023:0];
reg [15:0] data_outa_reg=16'h0000;
reg [15:0] data_outb_reg=16'h0000;
always@(posedge clka)begin
```

```
        if(rsta)begin
            data_outa <= 0;
        end
        else begin
            if (ocea)begin
                data_outa <= data_outa_reg;
            end
        end
    end
    always@(posedge clka)begin
        if(rsta)begin
            data_outa_reg <= 0;
        end
        else begin
            if(cea & !wrea)begin
                data_outa_reg <= mem[addra];
            end
        end
    end
    always @(posedge clka)begin
        if (cea & wrea) begin
            mem[addra] <= data_ina;
        end
    end

    always@(posedge clkb )begin
        if(rstb)begin
            data_outb <= 0;
        end
        else begin
            if (oceb)begin
                data_outb <= data_outb_reg;
            end
        end
```

```
end
always@(posedge clkb )begin
    if(rstb)begin
        data_outb_reg <= 0;
    end
    else begin
        if(ceb & !wreb)begin
            data_outb_reg <= mem[addrb];
        end
    end
end

always @(posedge clkb)begin
    if (ceb & wreb) begin
        mem[addrb] <= data_inb;
    end
end
endmodule
```

4.1.3 memory 定义时赋初值

memory 定义时赋初值仅 GowinSynthesis 支持，且 Verilog language 需选择 system Verilog。以 normal、bypass、同步复位模式的 DPB 为例介绍其实现，其编码形式可如下所示：

```
module top(data_outa, data_ina, addra, clka,cea, wrea,rsta,data_outb, data_inb, addrb, clkb,ceb, wreb,rstb);
output [3:0]data_outa,data_outb;
input [3:0]data_ina,data_inb;
input [2:0]addra,addrb;
input clka,wrea,cea,rsta;
input clkb,wreb,ceb,rstb;
reg [3:0] mem [7:0]={4'h1,4'h2,4'h3,4'h4,4'h5,4'h6,4'h7,4'h8};

reg [3:0] data_outa=4'h0;
reg [3:0] data_outb=4'h0;
always@(posedge clka)begin
```

```
        if(rsta)begin
            data_outa <= 0;
        end
        else begin
            if(cea & !wrea)begin
                data_outa <= mem[addra];
            end
        end
    end

    always @(posedge clka)begin
        if (cea & wrea) begin
            mem[addra] <= data_ina;
        end
    end

    always@(posedge clkb)begin
        if(rstb)begin
            data_outb <= 0;
        end
        else begin
            if(ceb & !wreb)begin
                data_outb <= mem[addrb];
            end
        end
    end

    always @(posedge clkb)begin
        if (ceb & wreb) begin
            mem[addrb] <= data_inb;
        end
    end
    endmodule
```

4.1.4 readmemb/readmemh 方式赋初值

readmemb/ readmemh 方式赋值在进行使用时需注意路径的书写，请以‘/’作为路径分隔符。以 normal、bypass、同步复位模式的 DPB 为例介绍其实现，其编码形式可如下所示：

```
module top(data_outa, data_ina, addra, clka,cea, wrea,rsta,data_outb, data_inb, addrb, clkb,ceb, wreb,rstb);
output [3:0]data_outa,data_outb;
input [3:0]data_ina,data_inb;
input [2:0]addra,addrb;
input clka,wrea,cea,rsta;
input clkb,wreb,ceb,rstb;
reg [3:0] mem [7:0];
reg [3:0] data_outa=4'h0;
reg [3:0] data_outb=4'h0;

initial begin
    $readmemb ("E:/dpb.mi", mem);
end

always@(posedge clka)begin
    if(rsta)begin
        data_outa <= 0;
    end
    else begin
        if(cea & !wrea)begin
            data_outa <= mem[addra];
        end
    end
end

always @(posedge clka)begin
    if (cea & wrea) begin
        mem[addra] <= data_ina;
    end
```

```
end

always@(posedge clkb)begin
    if(rstb)begin
        data_outb <= 0;
    end
    else begin
        if(ceb & !wreb)begin
            data_outb <= mem[addrb];
        end
    end
end

always @(posedge clkb)begin
    if (ceb & wreb) begin
        mem[addrb] <= data_inb;
    end
end
endmodule
```

dpb.mi 书写形式如下所示：

```
0001
0010
0011
0100
0101
0110
0111
1000
```

注！

如果以 windows 系统支持的路径分隔符‘\’，需要加转义字符，如 E:\\dpb.mi。

4.1.5 byte-enable 功能

byte-enable 功能的推断仅 Arora V BSRAM 支持，其编码形式可如下所示：

```
module Gowin_DPB (douta, doutb, clka, cea, reseta, wrea, clkb, ceb, resetb, wreb, ada, dina, adb, dinb, byte_ena, byte_enb);
output reg [15:0] douta;
output reg [15:0] doutb;
input clka,cea,reseta,wrea;
input clkb,ceb,resetb,wreb;
input [9:0] ada,adb;
input [15:0] dina,dinb;
input [1:0] byte_ena,byte_enb;

reg[15:0]mem[2**10-1:0];

always@(posedge clka)begin
    if(cea & wrea)begin
        if(byte_ena[0])begin
            mem[ada][7:0]<=dina[7:0];
        end
        if(byte_ena[1])begin
            mem[ada][15:8]<=dina[15:8];
        end
    end
end
always@(posedge clka)begin
    if(reseta)begin
        douta<=0;
    end
    else begin
        if(cea & !wrea)begin
            douta<=mem[ada];
        end
```

```
        end
    end


    always@(posedge clkb)begin
        if(ceb & wreb)begin
            if(byte_enb[0])begin
                mem[adb][7:0]<=dinb[7:0];
            end
            if(byte_enb[1])begin
                mem[adb][15:8]<=dinb[15:8];
            end
        end
    end
    always@(posedge clkb)begin
        if(resetb)begin
            doutb<=0;
        end
        else begin
            if(ceb & !wreb)begin
                doutb<=mem[adb];
            end
        end
    end

    endmodule
```

4.2 SP/SPX9

SP/SPX9 存储空间为 16Kbit/18Kbit，其工作模式为单端口模式，由一个时钟控制单端口的读/写操作，可支持 2 种读模式（bypass 模式和 pipeline 模式）和 3 种写模式（normal 模式、write-through 模式和 read-before-write 模式）。若综合出 SP/SPX9，memory（除移位寄存器形式）需满足至少满足下述条件之一：

1. 数据位宽*地址深度>=1024
2. 使用 syn_ramstyle = "block_ram"。

本节以读地址是否经过 register、初值读取等方面进行实现说明。

注！

对于 GW5A(S)(T)-138 器件，不支持 read-before-write 写模式编码风格。

4.2.1 读地址经过 register

读地址经过 register 时，仅支持读地址 register 无控制信号控制的情况，该类形式会综合出 write-through 模式的 SP。以输出不经过 register 为例介绍其实现，其编码形式可如下所示：

```
module top(data_out, data_in, addr, clk,ce, wre);
output [9:0]data_out;
input [9:0]data_in;
input [9:0]addr;
input clk,wre,ce;
reg [9:0] mem [1023:0];
reg [9:0]addr_reg=10'h000;

always@(posedge clk)begin
    addr_reg<=addr;
end
always @(posedge clk)begin
    if (ce & wre) begin
        mem[addr] <= data_in;
    end
end
assign data_out = mem[addr_reg];
endmodule
```

4.2.2 读地址不经过 Register，输出经过 Register

读地址不经过 register 时，输出必须经过 register，经过一级 register 时为 bypass 模式；经过两级 register 时为 pipeline 模式。以 write-through、bypass、同步复位模式的 SPX9 为例介绍其实现，其编码形式可如下所示：

```
module top(data_out, data_in, addr, clk,ce, wre,rst);
output reg [17:0]data_out=18'h00000;
input [17:0]data_in;
input [9:0]addr;
```

```
input clk,wre,ce,rst;
reg [17:0] mem [1023:0];
always@(posedge clk )begin
    if(rst)begin
        data_out <= 0;
    end
    else begin
        if(ce & wre)begin
            data_out <= data_in;
        end
        else begin
            if (ce & !wre)begin
                data_out <= mem[addr];
            end
        end
    end
end
always @(posedge clk)begin
    if (ce & wre)begin
        mem[addr] <= data_in;
    end
end
endmodule
```

4.2.3 memory 定义时赋初值

memory 定义时赋初值需 Verilog language 选择 system Verilog。以 normal、bypass、同步复位模式的 SP 为例介绍其实现，其编码形式可如下所示：

```
module top(data_out, data_in, addr, clk,ce, wre,rst) /*synthesis syn_ramstyle="block_ram"*/;
output [15:0]data_out;
input [15:0]data_in;
input [2:0]addr;
input clk,wre,ce,rst;
reg [15:0] mem [7:0]={16'h0123,16'h4567,16'h89ab,16'hcdef,
```

```
16'h0147,16'h0258,16'h789a,16'h5678};
    reg [15:0] data_out=16'h0000;
    always@(posedge clk )begin
        if(rst)begin
            data_out <= 0;
        end
        else begin
            if(ce & !wre)begin
                data_out <= mem[addr];
            end
        end
    end
    always @(posedge clk)begin
        if (ce & wre)begin
            mem[addr] <= data_in;
        end
    end
    endmodule
```

4.2.4 readmemb/readmemh 方式赋初值

readmemb/ readmemh 方式赋值使用时需注意路径的书写，请以‘/’作为路径分隔符。以 normal、pipeline、同步复位模式的 SP 为例介绍其实现，其编码形式可如下所示：

```
    module top(data_out, data_in, addr, clk, ce, oce, wre, rst)/*synthesis
syn_ramstyle="block_ram"*/;
    output reg [7:0]data_out=8'h00;
    input [7:0]data_in;
    input [2:0]addr;
    input clk,wre,ce,oce,rst;
    reg [7:0] mem [7:0];
    reg [7:0] data_out_reg=8'h00;
    initial begin
        $readmemh ("E:/sp.mi", mem);
    end
```

```
always@(posedge clk )begin
    if(rst)begin
        data_out <= 0;
    end
    else begin
        if(oce)begin
            data_out <= data_out_reg;
        end
    end
end
always@(posedge clk)begin
    if(rst)begin
        data_out_reg <= 0;
    end
    else begin
        if(ce & !wre)begin
            data_out_reg <= mem[addr];
        end
    end
end
always @(posedge clk)begin
    if (ce & wre) begin
        mem[addr] <= data_in;
    end
end
endmodule
```

sp.mi 书写格式如下：

```
12
34
56
78
9a
bc
```

de

ff

4.2.5 Decoder 形式

以 normal、bypass、同步复位模式的 SP 为例介绍其实现，其编码形式可如下所示：

```
module top (data_out, data_in, addr, clk,wre,rst)/*synthesis
syn_ramstyle="block_ram"*/;
parameter init0 = 16'h1234;
parameter init1 = 16'h5678;
parameter init2 = 16'h9abc;
parameter init3 = 16'h0147;

output reg[3:0]data_out;
input [3:0]data_in;
input [3:0]addr;
input clk,wre,rst;

reg [15:0] mem0=init0;
reg [15:0] mem1=init1;
reg [15:0] mem2=init2;
reg [15:0] mem3=init3;

always @(posedge clk)begin
    if (wre) begin
      mem0[addr] <= data_in[0];
      mem1[addr] <= data_in[1];
      mem2[addr] <= data_in[2];
      mem3[addr] <= data_in[3];
  end
end
always @(posedge clk)begin
    if(rst)begin
        data_out<=16'h00;
    end
```

```
        else begin
            if (!wre) begin

            data_out[0] <= mem0[addr];
            data_out[1] <= mem1[addr];
            data_out[2] <= mem2[addr];
            data_out[3] <= mem3[addr];
          end
        end
    end
    endmodule
```

4.2.6 byte-enable 功能

byte-enable 功能的推断仅 Arora V BSRAM 支持，其编码形式可如下所示:

```
module Gowin_SP (dout, clk, ce, reset, wre, ad, din, byte_en);
output reg[35:0] dout;
input clk,ce,reset,wre;
input [7:0] ad;
input [35:0] din;
input [3:0] byte_en;

reg[35:0]mem[2**8-1:0];

always@(posedge clk)begin
    if(ce & wre)begin
        if(byte_en[0])begin
            mem[ad][8:0]<=din[8:0];
        end
        if(byte_en[1])begin
            mem[ad][17:9]<=din[17:9];
        end
        if(byte_en[2])begin
            mem[ad][26:18]<=din[26:18];
```

```
            end
            if(byte_en[3])begin
                mem[ad][35:27]<=din[35:27];
            end
        end
    end

    always@(posedge clk)begin
        if(reset)begin
            dout<=0;
        end
        else begin
            if(ce & !wre)begin
                dout<=mem[ad];
            end
        end
    end
    endmodule
```

4.3 SDPB/SDPX9B

SDPB/SDPX9B 存储空间分别为 16Kbit/18Kbit，工作模式为伪双端口模式，可支持 2 种读模式（bypass 模式和 pipeline 模式）和 1 种写模式（normal 模式）。若综合出 SDPB/SDPX9B，memory（除移位寄存器形式）需满足下述条件之一：

1. 数据位宽*地址深度>=1024；
2. 使用 syn_ramstyle = "block_ram"。

4.3.1 memory 无初值

以 bypass、同步复位模式的 SDPB 为例介绍其实现，其编码形式可如下所示：

```
module top(dout, din, ada, adb, clka, cea, clkb, ceb, resetb);
output reg[15:0]dout=16'h0000;
input [15:0]din;
input [9:0]ada, adb;
input clka, cea,clkb, ceb, resetb;
```

```
reg [15:0] mem [1023:0];

always @(posedge clka)begin
    if (cea )begin
        mem[ada] <= din;
    end
end

always@(posedge clkb)begin
    if(resetb)begin
        dout <= 0;
    end
    else if(ceb)begin
        dout <= mem[adb];
    end
end

endmodule
```

4.3.2 memory 定义时赋初值

以 pipeline、异步复位模式的 SDPB 为例介绍其实现形式，其编码形式可如下所示：

```
module top(data_out, data_in, addra, addrb, clka, cea, clkb, ceb,oce, rstb)/*synthesis syn_ramstyle="block_ram"*/;
output reg[15:0]data_out=16'h0000;
input [15:0]data_in;
input [2:0]addra, addrb;
input clka, cea, clkb, ceb, rstb,oce;
reg [15:0] mem [7:0]={16'h0123,16'h4567,16'h89ab,16'hcdef, 16'h0147, 16'h0258,16'h789a,16'h5678};
reg [15:0] data_out_reg=16'h0000;

always @(posedge clka)begin
    if (cea )begin
        mem[addra] <= data_in;
```

```
        end
    end

    always@(posedge clkb or posedge rstb)begin
        if(rstb)begin
            data_out_reg <= 0;
        end
        else if(ceb)begin
            data_out_reg<= mem[addrb];
        end
    end
    always@(posedge clkb or posedge rstb)begin
        if(rstb)begin
            data_out <= 0;
        end
        else if(oce)begin
            data_out<= data_out_reg;
        end
    end
    endmodule
```

4.3.3 readmemb/readmemh 方式赋初值

readmemb/ readmemh 方式赋值在进行使用时需注意路径的书写，请以‘/’作为路径分隔符。以 bypass、异步复位模式的 SDPB 为例介绍其实现，其编码形式可如下所示：

```
module top(dout, din, ada, adb, clka, cea, clkb, ceb, resetb)/*synthesis syn_ramstyle="block_ram"*/;
output reg[7:0]dout=8'h00;
input [7:0]din;
input [2:0]ada, adb;
input clka, cea,clkb, ceb, resetb;
reg [7:0] mem [7:0];
initial begin
    $readmemh ("E:/sdpb.mi", mem);
end
```

```
always @(posedge clka)begin
    if (cea )begin
        mem[ada] <= din;
    end
end

always@(posedge clkb or posedge resetb)begin
    if(resetb)begin
        dout <= 0;
    end
    else if(ceb)begin
        dout <= mem[adb];
    end
end

endmodule
```

sdpb.mi 书写格式如下：

```
12
34
56
78
9a
bc
de
ff
```

4.3.4 移位寄存器形式

移位寄存器形式综合出 BSRAM 需满足下述条件之一：

1. memory 深度>=5 且 memory 深度*数据位宽>256，且 memory 深度！=2 的 n 次方+1；
2. 添加属性约束 syn_srlstyle="block_ram"，且 memory 深度！=2 的 n 次方+1, memory 深度>=5。

以 bypass、同步复位模式的 SDPX9B 介绍其实现，其编码形式可如下所示：

```
module    p_seqshift(clk, we, din, dout);
parameter width=18;
parameter depth=16;
input clk, we;
input [width-1:0] din;
output [width-1:0] dout;

reg [width-1:0] regBank[depth-1:0];

always @(posedge clk) begin
    if (we) begin
        regBank[depth-1:1] <= regBank[depth-2:0];
        regBank[0] <= din;
    end
end

assign dout = regBank[depth-1];
endmodule
```

4.3.5 不对称类型

不对称类型综合出 SDPB。以 bypass、同步复位模式的 SDPB 为例介绍其实现，其编码形式可如下所示：

```
module top(dout, din, ada, clka, cea, adb, clkb, ceb, rstb,oce);
parameter adawidth = 8;
parameter diwidth = 6;
parameter adbwidth = 7;
parameter dowidth = 12;

output [dowidth-1:0]dout;
input [diwidth-1:0]din;
input [adawidth-1:0]ada;
input [adbwidth-1:0]adb;
```

```
input clka,cea,clkb,ceb,rstb,oce;
reg [diwidth-1:0]mem [2**adawidth-1:0];
reg [dowidth-1:0]dout_reg;
localparam b = 2**adawidth/2**adbwidth ;
integer j ;

always @(posedge clka)begin
    if (cea)begin
        mem[ada] <= din;
    end
end

always@(posedge clkb )begin
    if(rstb)begin
        dout_reg <= 0;
    end
    else begin
        if(ceb)begin
            for(j = 0;j < b;j = j+1)
                dout_reg[((j+1)*diwidth-1)-: diwidth]<=
mem[adb*b+j];
        end
    end
end
assign dout = dout_reg;
endmodule
```

4.3.6 Decoder 形式

以 bypass、同步复位模式的 SDPB 为例介绍其实现，其编码形式可如下所示：

```
module top (data_out, data_in, wad, rad,rst, clk,wre)/*synthesis
syn_ramstyle="block_ram"*/;;
parameter init0 = 16'h1234;
parameter init1 = 16'h5678;
parameter init2 = 16'h9abc;
```

```
parameter init3 = 16'h0147;

output reg[3:0] data_out;
input [3:0]data_in;
input [3:0]wad,rad;
input clk,wre,rst;

reg [15:0] mem0=init0;
reg [15:0] mem1=init1;
reg [15:0] mem2=init2;
reg [15:0] mem3=init3;
always @(posedge clk)begin
      if (wre) begin
         mem0[wad] <= data_in[0];
         mem1[wad] <= data_in[1];
         mem2[wad] <= data_in[2];
         mem3[wad] <= data_in[3];
    end
end
always @(posedge clk)begin
      if(rst)begin
            data_out<=16'h00;
      end
      else begin
            data_out[0] <= mem0[rad];
            data_out[1] <= mem1[rad];
            data_out[2] <= mem2[rad];
            data_out[3] <= mem3[rad];
      end
end
endmodule
```

4.3.7 byte-enable 功能

byte-enable 功能的推断仅 Arora V BSRAM 支持，其编码形式可如下所示:

```
module Gowin_SDPB (dout, clka, cea, clkb, ceb, reset, ada, din, adb, byte_ena);

output reg[31:0] dout;
input clka,cea;
input clkb,ceb;
input reset;
input [7:0] ada,adb;
input [31:0] din;
input [3:0] byte_ena;

reg[31:0]mem[2**8-1:0];

always@(posedge clka)begin
    if(cea)begin
        if(byte_ena[0])begin
            mem[ada][7:0]<=din[7:0];
        end
        if(byte_ena[1])begin
            mem[ada][15:8]<=din[15:8];
        end
        if(byte_ena[2])begin
            mem[ada][23:16]<=din[23:16];
        end
        if(byte_ena[3])begin
            mem[ada][31:24]<=din[31:24];
        end
    end
end
always@(posedge clkb)begin
    if(reset)begin
```

```
            dout<=0;
        end
        else begin
            if(ceb)begin
                dout<=mem[adb];
            end
        end
    end
    endmodule
```

4.4 pROM/pROMX9

pROM/pROMX9(16K/18K Block ROM)，16Kbit/18Kbit 块状只读储存器。其工作模式为只读模式，可支持 2 种读模式（bypass 模式和 pipeline 模式）。pROM/pROMX9 的赋值方式有 case 语句、readmemb/readmemh、memory 定义时赋值等方式赋值。若综合出 pROM，需至少满足下述条件之一：

1. 数据位宽*地址深度>=1024，且地址深度>32
2. 使用 syn_romstyle = "block_rom"

4.4.1 case 语句赋初值

以 bypass、同步复位模式的 pROM 为例介绍其实现，其编码形式可如下所示：

```
module top(clk,rst,ce,addr,dout)/*synthesis
syn_romstyle="block_rom"*/ ;
input clk;
input rst,ce;
input [4:0] addr;
output reg [31:0] dout=32'h00000000;

always @(posedge clk )begin
    if (rst) begin
        dout <= 0;
    end
    else begin
        if(ce)begin
            case(addr)
                5'h00: dout <= 32'h52853fd5;
```

```
            5'h01: dout <= 32'h38581bd2;
            5'h02: dout <= 32'h040d53e4;
            5'h03: dout <= 32'h22ce7d00;
            5'h04: dout <= 32'h73d90e02;
            5'h05: dout <= 32'hc0b4bf1c;
            5'h06: dout <= 32'hec45e626;
            5'h07: dout <= 32'hd9d000d9;
            5'h08: dout <= 32'haacf8574;
            5'h09: dout <= 32'hb655bf16;
            5'h0a: dout <= 32'h8c565693;
            5'h0b: dout <= 32'hb19808d0;
            5'h0c: dout <= 32'he073036e;
            5'h0d: dout <= 32'h41b923f6;
            5'h0e: dout <= 32'hdce89022;
            5'h0f: dout <= 32'hba17fce1;
            5'h10: dout <= 32'hd4dec5de;
            5'h11: dout <= 32'ha18ad699;
            5'h12: dout <= 32'h4a734008;
            5'h13: dout <= 32'h5c32ac0e;
            5'h14: dout <= 32'h8f26bdd4;
            5'h15: dout <= 32'hb8d4aab6;
            5'h16: dout <= 32'hf55e3c77;
            5'h17: dout <= 32'h41a5d418;
            5'h18: dout <= 32'hba172648;
            5'h19: dout <= 32'h5c651d69;
            5'h1a: dout <= 32'h445469c3;
            5'h1b: dout <= 32'h2e49668b;
            5'h1c: dout <= 32'hdc1aa05b;
            5'h1d: dout <= 32'hcebfe4cd;
            5'h1e: dout <= 32'h1e1f0f1e;
            5'h1f: dout <= 32'h86fd31ef;
            default: dout <= 32'h8e9008a6;
        endcase
    end
```

```
    end
end
endmodule
```

4.4.2 memory 定义时赋初值

memory 定义时赋初值需 Verilog language 选择 system Verilog。以 bypass、同步复位模式的 pROM 为例介绍其实现形式，其编码形式可如下所示：

```
module top ( clk, addr,rst, data_out)/* synthesis syn_romstyle = "block_rom" */;
input clk;
input rst;
input [3:0] addr;
output reg[3:0] data_out;
reg [3:0] mem [15:0]={4'h1,4'h2,4'h3,4'h4,4'h5,4'h6,4'h7,4'h8,4'h9,4'ha, 4'hb, 4'hc,4'hd,4'he,4'hf,4'hd};

always @(posedge clk)begin
    if (rst) begin
      data_out <= 0;
   end
    else begin
        data_out <= mem[addr];;
    end
end
endmodule
```

4.4.3 readmemb/readmemh 方式赋初值

readmemb/ readmemh 方式赋值在进行使用时需注意路径的书写，请以‘/’作为路径分隔符。以 pipeline、异步复位模式的 pROM 为例介绍其实现，其编码形式可如下所示：

```
module top ( clk, addr, rst,oce,data_out);
input clk;
input rst,oce;
input [4:0] addr;
output reg   [31:0] data_out;
```

```
reg [31:0] mem [31:0] /* synthesis syn_romstyle = "block_rom" */;
reg [31:0] data_out_reg;
initial begin
    $readmemh ("E:/prom.ini", mem);
end
always @(posedge clk or posedge rst)begin
    if(rst)begin
        data_out_reg <=0;
    end
    else begin
        data_out_reg <= mem[addr];
    end
end

always @(posedge clk or posedge rst)begin
    if(rst)begin
        data_out <=0;
    end
    else begin
        data_out <= data_out_reg;
    end
end
endmodule
```

prom.ini 数据如下：

```
11001100
11001100
11001100
11001100
17001100
11001100
16001100
1f001100
11111100
```

11111100
11001110
11000111
11000111
11001110
11011100
11001110

5 SSRAM 编码规范

SSRAM 是分布式静态随机存储器，可配置成单端口模式，伪双端口模式和只读模式。

5.1 RAM16S 类型

RAM16S 类型包含 RAM16S1、RAM16S2、RAM16S4，其区别在于输出位宽宽度。RAM16S 类型可以采用 Decoder、Memory、移位寄存器等形式进行书写，若综合出 RAM16S，memory（除移位寄存器形式）需满足以下其中一个条件：

1. 读地址和输出不经过 register；
2. 输出经过 register，地址深度*数据位宽<1024；
3. 使用属性约束 syn_ramstyle="distributed_ram"。

5.1.1 Decoder 形式

以 RAM16S4 为例，介绍 Decoder 形式的实现，其编码形式可如下所示：

```
module top (data_out, data_in, addr, clk,wre);
parameter init0 = 16'h1234;
parameter init1 = 16'h5678;
parameter init2 = 16'h9abc;
parameter init3 = 16'h0147;

output [3:0]data_out;
input [3:0]data_in;
input [3:0]addr;
input clk,wre;
reg [15:0] mem0=init0;
reg [15:0] mem1=init1;
```

```
reg [15:0] mem2=init2;
reg [15:0] mem3=init3;

always @(posedge clk)begin
    if (wre) begin
      mem0[addr] <= data_in[0];
      mem1[addr] <= data_in[1];
      mem2[addr] <= data_in[2];
      mem3[addr] <= data_in[3];
   end
end

assign data_out[0] = mem0[addr];
assign data_out[1] = mem1[addr];
assign data_out[2] = mem2[addr];
assign data_out[3] = mem3[addr];
endmodule
```

5.1.2 Memory 形式

memory 形式可根据初值形式分为 memory 无初值形式、memory 定义时赋初值以及 readmemh/readmemb 形式。memory 定义时赋初值以及 readmemh/readmemb 形式请参考 4.1.4 readmemb/readmemh 方式赋初值，本节不再赘述。

以 RAM16S4 为例，介绍 memory 无初值形式的实现，其编码形式可如下所示：

```
module top(data_out, data_in, addr, clk, wre);
output [3:0]data_out;
input [3:0]data_in;
input [3:0]addr;
input clk,wre;
reg [3:0] mem [15:0];
always @(posedge clk)begin
    if ( wre) begin
        mem[addr] <= data_in;
    end
```

```
end
assign data_out = mem[addr];
endmodule
```

5.1.3 移位寄存器形式

需满足以下条件之一：

1. memory 深度>3 且 8<memory 深度*数据位宽<=256，且 memory 深度=2 的 n 次方；
2. 添加属性约束 syn_srlstyle= "distributed_ram"，且 memory 深度=2 的 n 次方, memory 深度>3 且 memory 深度*数据位宽>8。

以 GowinSynthesis 综合出 RAM16S4 为例，介绍移位寄存器形式的实现，其编码形式可如下所示：

```
module    p_seqshift(clk, we, din, dout);
parameter width=18;
parameter depth=4;
input clk, we;
input [width-1:0] din;
output [width-1:0] dout;

reg [width-1:0] regBank[depth-1:0];

always @(posedge clk) begin
    if (we) begin
        regBank[depth-1:1] <= regBank[depth-2:0];
        regBank[0] <= din;
    end
end

assign dout = regBank[depth-1];
endmodule
```

5.2 RAM16SDP 类型

RAM16SDP 类型包含 RAM16SDP1、RAM16SDP2、RAM16SDP4，其区别在于输出位宽宽度。RAM16SDP 类型可以采用 Decoder、Memory、移位寄存器等形式进行书写，若综合出 RAM16SDP，memory（除移位寄存器形式）需满足以下其中一个条件：

1. 读地址和输出不经过 register；
2. 读地址或输出经过 register，地址深度*数据位宽<1024；
3. 使用属性约束 syn_ramstyle="distributed_ram"。

5.2.1 Decoder 形式

以 RAM16SDP4 为例，介绍 Decoder 形式的实现，其编码形式可如下所示：

```
module top (data_out, data_in, wad, rad, clk,wre);
parameter init0 = 16'h1234;
parameter init1 = 16'h5678;
parameter init2 = 16'h9abc;
parameter init3 = 16'h0147;

output [3:0]data_out;
input [3:0]data_in;
input [3:0]wad,rad;
input clk,wre;
reg [15:0] mem0=init0;
reg [15:0] mem1=init1;
reg [15:0] mem2=init2;
reg [15:0] mem3=init3;

always @(posedge clk)begin
      if (wre) begin
         mem0[wad] <= data_in[0];
         mem1[wad] <= data_in[1];
         mem2[wad] <= data_in[2];
         mem3[wad] <= data_in[3];
    end
end

assign data_out[0] = mem0[rad];
assign data_out[1] = mem1[rad];
assign data_out[2] = mem2[rad];
```

```
assign data_out[3] = mem3[rad];
endmodule
```

5.2.2 Memory 形式

memory 形式可根据初值形式分为 memory 无初值形式、memory 定义时赋初值以及 readmemh/readmemb 形式。memory 定义时赋初值以及 readmemh/readmemb 形式请参考 4.1.4 readmemb/readmemh 方式赋初值，本节不再赘述。

以 RAM16SDP4 为例，介绍 Memory 形式的实现，其编码形式可如下所示：

```
module top(data_out, data_in, addra, clk, wre, addrb);
output [3:0]data_out;
input [3:0]data_in;
input [3:0] addra ,addrb;
input clk,wre;

reg [3:0] mem [15:0];

always @(posedge clk)begin
    if (wre)begin
        mem[addra] <= data_in;
    end
end

assign data_out = mem[addrb];

endmodule
```

5.2.3 移位寄存器形式

移位寄存器若综合为 RAM16SDP 类型，需满足以下条件之一：

1. memory 深度>3 且 memory 深度*数据位宽<=256，且 memory 深度!=2 的 n 次方；
2. 添加属性约束 syn_srlstyle= " distributed_ram"，且 memory 深度!=2 的 n 次方，memory 深度>3 且 memory 深度*数据位宽>8。

以 GowinSynthesis 综合出 RAM16SDP4 为例，介绍移位寄存器形式的

实现，其编码形式可如下所示：

```
module    p_seqshift(clk, we, din, dout);
parameter width=18;
parameter depth=7;
input clk, we;
input [width-1:0] din;
output [width-1:0] dout;

reg [width-1:0] regBank[depth-1:0];

always @(posedge clk) begin
    if (we) begin
        regBank[depth-1:1] <= regBank[depth-2:0];
        regBank[0] <= din;
    end
end

assign dout = regBank[depth-1];
endmodule
```

5.3 ROM16

ROM16 是地址深度为 16，数据位宽为 1 的只读存储器，存储器的内容通过 INIT 进行初始化，ROM16 可以采用 Memory、Decoder 等形式进行书写，ROM16 的综合需要添加属性约束 syn_romstyle ="distributed_rom"。

5.3.1 Decoder 形式

```
module top(addr,dataout)/*synthesis
syn_romstyle="distributed_rom"*/ ;
input [3:0] addr;
output reg    dataout=1'h0;
always @(*)begin
    case(addr)
        4'h0: dataout <= 1'h0;
        4'h1: dataout <= 1'h0;
        4'h2: dataout <= 1'h1;
```

```
            4'h3: dataout <= 1'h0;
            4'h4: dataout <= 1'h1;
            4'h5: dataout <= 1'h1;
            4'h6: dataout <= 1'h0;
            4'h7: dataout <= 1'h0;
            4'h8: dataout <= 1'h0;
            4'h9: dataout <= 1'h1;
            4'ha: dataout <= 1'h0;
            4'hb: dataout <= 1'h0;
            4'hc: dataout <= 1'h1;
            4'hd: dataout <= 1'h0;
            4'he: dataout <= 1'h0;
            4'hf: dataout <= 1'h0;
            default: dataout <=  1'h0;
        endcase
    end
    endmodule
```

5.3.2 Memory 形式

memory 形式可根据初值形式分为 memory 无初值形式、memory 定义时赋初值以及 readmemh/readmemb 形式。memory 定义时赋初值以及 readmemh/readmemb 形式请参考 4.1.4，本节不再赘述。

```
    module top (addr,dataout)/*synthesis
syn_romstyle="distributed_rom"*/;
    input [3:0] addr;
    output reg dataout=1'b0;

    parameter init0 = 16'h117a;
    reg [15:0] mem0=init0;
    always @(*)begin
        dataout <= mem0[addr];
        end
    endmodule
```

6 DSP 编码规范[1]

注！

[1] Arora V 产品 DSP 编码规范参见第 7 章 Arora V DSP 编码规范。

DSP（Digital Signal Processing）数字信号处理包含预加器（Pre-Adder），乘法器（MULT）和 54 位算术逻辑单元（ALU54D）。

6.1 Pre-adder

Pre-adder 是预加器，实现预加、预减和移位功能。Pre-adder 按照位宽分为两种，分别是 9 位宽的 PADD9 和 18 位宽的 PADD18。Pre-adder 需与 Multiplier 相配合才可推断出来。

6.1.1 预加功能

以经过 AREG、BREG、同步复位模式的 PADD9-MULT9X9 为例介绍 PADD 预加功能的实现，其编码形式可如下所示：

```
module top(a0, b0, b1, dout, rst, clk, ce);
input [7:0] a0;
input [7:0] b0;
input [7:0] b1;
input rst, clk, ce;
output [17:0] dout;
reg [8:0] p_add_reg=9'h000;
reg [7:0] a0_reg=8'h00;
reg [7:0] b0_reg=8'h00;
reg [7:0] b1_reg=8'h00;
reg [17:0] pipe_reg=18'h00000;
reg [17:0] s_reg=18'h00000;
```

```
always @(posedge clk)begin
    if(rst)begin
        a0_reg <= 0;
        b0_reg <= 0;
        b1_reg <= 0;
    end else begin
        if(ce)begin
            a0_reg <= a0;
            b0_reg <= b0;
            b1_reg <= b1;
        end
    end
end
always @(posedge clk)begin
    if(rst)begin
            p_add_reg <= 0;
    end else begin
        if(ce)begin
                p_add_reg <= b0_reg+b1_reg;
        end
    end
end


always @(posedge clk)
begin
    if(rst)begin
            pipe_reg <= 0;
    end else begin
        if(ce) begin
            pipe_reg <= a0_reg*p_add_reg;
        end
    end
end
```

```
always @(posedge clk)
begin
    if(rst)begin
        s_reg <= 0;
    end else begin
        if(ce) begin
            s_reg <= pipe_reg;
        end
    end
end

assign dout = s_reg;

endmodule
```

6.1.2 预减功能

以经过 AREG、BREG、同步复位模式的 PADD9-MULT9X9 为例介绍 PADD 预减功能的实现，其编码形式可如下所示：

```
module top(a0, b0, b1, dout, rst, clk, ce);
input [7:0] a0;
input [7:0] b0;
input [7:0] b1;
input rst, clk, ce;
output [17:0] dout;
reg [8:0] p_add_reg=9'h000;
reg [7:0] a0_reg=8'h00;
reg [7:0] b0_reg=8'h00;
reg [7:0] b1_reg=8'h00;
reg [17:0] pipe_reg=18'h00000;
reg [17:0] s_reg=18'h00000;

always @(posedge clk)begin
    if(rst)begin
        a0_reg <= 0;
```

```
            b0_reg <= 0;
            b1_reg <= 0;
        end else begin
            if(ce)begin
                a0_reg <= a0;
                b0_reg <= b0;
                b1_reg <= b1;
            end
        end
    end
    always @(posedge clk)begin
        if(rst)begin
                p_add_reg <= 0;
        end else begin
            if(ce)begin
                    p_add_reg <= b0_reg-b1_reg;
            end
        end
    end


    always @(posedge clk)
    begin
        if(rst)begin
                pipe_reg <= 0;
        end else begin
            if(ce) begin
                pipe_reg <= a0_reg*p_add_reg;
            end
        end
    end
    always @(posedge clk)
    begin
        if(rst)begin
```

```
        s_reg <= 0;
    end else begin
        if(ce) begin
            s_reg <= pipe_reg;
        end
    end
end

assign dout = s_reg;

endmodule
```

6.1.3 移位功能

以经过 AREG、BREG、异步复位模式的 PADD18-MULT18X18 介绍 PADD 移位功能的实现，其编码形式可如下所示：

```
module top(a0, a1, b0, b1, p0, p1, clk, ce, reset);
parameter a_width=18;
parameter b_width=18;
parameter p_width=36;
input [a_width-1:0] a0, a1;
input [b_width-1:0] b0, b1;
input clk, ce, reset;
output [p_width-1:0] p0, p1;
wire [b_width-1:0] b0_padd, b1_padd;
reg [b_width-1:0] b0_reg=18'h00000;
reg [b_width-1:0] b1_reg=18'h00000;
reg [b_width-1:0] bX1=18'h00000;
reg [b_width-1:0] bY1=18'h00000;

always @(posedge clk or posedge reset)
begin
    if(reset)begin
        b0_reg <= 0;
        b1_reg <= 0;
```

```
            bX1 <= 0;
            bY1 <= 0;
        end else begin
            if(ce)begin
                b0_reg <= b0;
                b1_reg <= b1;
                bX1 <= b0_reg;
                bY1 <= b1_reg;
            end
        end
    end
    assign b0_padd = bX1 + b1_reg;
    assign b1_padd= b0_reg + bY1;

    assign p0 = a0 * b0_padd;
    assign p1 = a1 * b1_padd;

    endmodule
```

6.2 Multiplier

Multiplier 是 DSP 的乘法器单元，乘法器的乘数输入信号定义为 MDIA 和 MDIB，乘积输出信号定义为 MOUT，可实现乘法运算：DOUT=A*B。

Multiplier 根据数据位宽可配置成 9x9，18x18，36x36 等乘法器，分别对应原语 MULT9X9，MULT18X18，MULT36X36。以经过 AREG、BREG、OUT_REG、PIPE_REG、异步复位模式的 MULT18X18 为例介绍其实现，其编码形式可如下所示：

```
module top(a,b,c,clock,reset,ce);
input signed [17:0] a;
input signed [17:0] b;
input clock;
input reset;
input ce;
output signed [35:0] c;

reg signed [17:0] ina=18'h00000;
```

```
reg signed [17:0] inb=18'h00000;
reg signed [35:0] pp_reg=36'h000000000;
reg signed [35:0] out_reg=36'h000000000;
wire signed [35:0] mult_out;

always @(posedge clock or posedge reset)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 or posedge reset)begin
    if(reset)begin
        pp_reg<=0;
    end else begin
        if(ce)begin
            pp_reg<=mult_out;
        end
    end
end

always @(posedge clock or posedge reset)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
```

6.3 ALU54D

ALU54D（54-bit Arithmetic Logic Unit）是 54 位算术逻辑单元，实现 54 位的算术逻辑运算。若综合出 ALU54D，数据位宽 width 需在[48,54]区间，否则需添加属性约束 syn_dspstyle="dsp"。以经过 AREG、BREG、OUT_REG、异步复位模式的 ALU54D 介绍其实现，其编码形式可如下所示：

```
module top(a, b, s, accload, clk, ce, reset);
parameter width=54;
input signed [width-1:0] a, b;
input accload, clk, ce, reset;
output signed [width-1:0] s;
wire signed [width-1:0] s_sel;
reg [width-1:0] a_reg=54'h00000000000000;
reg [width-1:0] b_reg=54'h00000000000000;
reg [width-1:0] s_reg=54'h00000000000000;
reg acc_reg=1'b0;

always @(posedge clk or posedge reset)
begin
    if(reset)begin
        a_reg <= 0;
        b_reg <= 0;
    end else begin
        if(ce)begin
            a_reg <= a;
            b_reg <= b;
        end
    end
end
always @(posedge clk)
begin
```

```
    if(ce)begin
        acc_reg <= accload;
    end
end

assign s_sel = (acc_reg == 1) ? s : 0;
always @(posedge clk or posedge reset)
begin
    if(reset)begin
        s_reg <= 0;
    end else begin
        if(ce)begin
            s_reg <= s_sel + a_reg + b_reg;
        end
    end
end
assign s = s_reg;
endmodule
```

6.4 MULTALU

MULTALU 模式实现一个乘法器输出经过 54-bit ALU 运算，包括 MULTALU36X18 和 MULTALU18X18，其中 MULTALU18X18 仅支持实例化形式。MULTALU36X18 有三种运算功能：DOUT=A*B±C、DOUT=∑(A*B)、DOUT=A*B+CASI。

6.4.1 A*B±C 功能

以经过 AREG、BREG、CREG、PIPE_REG、OUT_REG、异步复位模式的 MULTALU36X18 介绍 DOUT=A*B+C 功能的实现，其编码形式可如下所示：

```
module top(a0, b0, c,s, reset, ce, clock);
parameter a_width=36;
parameter b_width=18;
parameter s_width=54;
input signed [a_width-1:0] a0;
input signed [b_width-1:0] b0;
input signed [s_width-2:0] c;
```

```
input reset, ce, clock;
output signed [s_width-1:0] s;
wire signed [s_width-1:0] p0;
reg signed [a_width-1:0] a0_reg=36'h000000000;
reg signed [b_width-1:0] b0_reg=18'h00000;
reg signed [s_width-1:0] p0_reg=54'h00000000000000;
reg signed [s_width-1:0] o0_reg=54'h00000000000000;
reg signed [s_width-2:0] c_reg=54'h00000000000000;
always @(posedge clock or posedge reset)begin
    if(reset)begin
        a0_reg <= 0;
        b0_reg <= 0;
        c_reg <= 0;
    end else begin
        if(ce)begin
            a0_reg <= a0;
            b0_reg <= b0;
              c_reg <= c;
        end
    end
end

assign p0 = a0_reg * b0_reg;
always @(posedge clock or posedge reset)begin
    if(reset)begin
        p0_reg <= 0;
        o0_reg <= 0;
    end else begin
        if(ce)begin
            p0_reg <= p0;
            o0_reg <= p0_reg+c_reg;
        end
    end
end
```

```
assign s = o0_reg;
endmodule
```

6.4.2 ∑(A*B)功能

以经过 PIPE_REG、OUT_REG、异步复位模式的 MULTALU36X18 介绍 DOUT=∑(A*B)功能的实现，其编码形式可如下所示：

```
module top(a,b,c,clock,reset,ce);
parameter a_width = 36;
parameter b_width = 18;
parameter c_width = 54;
input signed [a_width-1:0] a;
input signed [b_width-1:0] b;
input clock;
input reset,ce;
output signed [c_width-1:0] c;

reg signed [c_width-1:0] pp_reg=54'h00000000000000;
reg signed [c_width-1:0] out_reg=54'h00000000000000;
wire signed [c_width-1:0] mult_out,c_sel;
reg acc_reg0=1'b0;
reg acc_reg1=1'b0;

assign mult_out=a*b;
always @(posedge clock or posedge reset) begin
    if(reset)begin
        pp_reg<=0;
    end else begin
        if(ce)begin
            pp_reg<=mult_out;
        end
    end
end
```

```
always @(posedge clock or posedge reset)
begin
    if(reset) begin
        out_reg <= 0;
    end else if(ce) begin
        out_reg <= c + pp_reg;
    end
end

assign c=out_reg;
endmodule
```

6.4.3 A*B+CASI 功能

以经过 PIPE_REG、OUT_REG、异步复位模式的 MULTALU36X18 介绍 DOUT=A*B+CASI 功能的实现，其编码形式可如下所示：

```
module top(a0, a1, a2, b0, b1, b2, s, reset, ce, clock);
parameter a_width=36;
parameter b_width=18;
parameter s_width=54;
input signed [a_width-1:0] a0, a1, a2;
input signed [b_width-1:0] b0, b1, b2;
input reset, ce, clock;
output signed [s_width-1:0] s;
wire signed [s_width-1:0] p0, p1, p2, s0, s1;
reg signed [s_width-1:0] p0_reg=54'h00000000000000;
reg signed [s_width-1:0] p1_reg=54'h00000000000000;
reg signed [s_width-1:0] p2_reg=54'h00000000000000;
reg signed [s_width-1:0] s0_reg=54'h00000000000000;
reg signed [s_width-1:0] s1_reg=54'h00000000000000;
reg signed [s_width-1:0] o0_reg=54'h00000000000000;

assign p0 = a0 * b0;
assign p1 = a1 * b1;
assign p2 = a2 * b2;
```

```
always @(posedge clock or posedge reset)
begin
    if(reset)begin
        p0_reg <= 0;
        p1_reg <= 0;
        p2_reg <= 0;
        o0_reg <= 0;
    end else begin
        if(ce)begin
            p0_reg <= p0;
            p1_reg <= p1;
            p2_reg <= p2;
            o0_reg <= p0_reg;
        end
    end
end

assign s0 = o0_reg + p1_reg;
always @(posedge clock or posedge reset)
begin
    if(reset)begin
        s0_reg <= 0;
    end else begin
        if(ce)begin
            s0_reg <= s0;
        end
    end
end

assign s1 = s0_reg + p2_reg;
always @(posedge clock or posedge reset)
begin
    if(reset)begin
        s1_reg <= 0;
```

```
        end else begin
            if(ce)begin
                s1_reg <= s1;
            end
        end
    end
    assign s=s1_reg;
    endmodule
```

6.5 MULTADDALU

MULTADDALU（The Sum of Two Multipliers with ALU）是带 ALU 功能的乘加器，实现乘法求和后累加或 reload 运算，对应的原语为 MULTADDALU18X18。有三种运算功能：DOUT=A0*B0±A1*B1±C、DOUT=∑(A0*B0±A1*B1)、DOUT=A0*B0±A1*B1+CASI。

6.5.1 A0*B0±A1*B1±C 功能

以经过 A0REG、A1REG、B0REG、B1REG、PIPE0_REG、PIPE1_REG、OUT_REG、异步复位模式的 MULTADDALU18X18 介绍 DOUT= A0*B0 ± A1*B1±C 功能的实现，其编码形式可如下所示：

```
module top(a0, a1, b0, b1, c,s, reset, clock, ce);
parameter a0_width=18;
parameter a1_width=18;
parameter b0_width=18;
parameter b1_width=18;
parameter s_width=54;
input signed [a0_width-1:0] a0;
input signed [a1_width-1:0] a1;
input signed [b0_width-1:0] b0;
input signed [b1_width-1:0] b1;
input [53:0] c;
input reset, clock, ce;
output signed [s_width-1:0] s;
wire signed [s_width-1:0] p0, p1, p;
reg signed [a0_width-1:0] a0_reg=18'h00000;
reg signed [a1_width-1:0] a1_reg=18'h00000;
reg signed [b0_width-1:0] b0_reg=18'h00000;
```

```
reg signed [b1_width-1:0] b1_reg=18'h00000;
reg signed [s_width-1:0] p0_reg=54'h00000000000000;
reg signed [s_width-1:0] p1_reg=54'h00000000000000;
reg signed [s_width-1:0] s_reg=54'h00000000000000;

always @(posedge clock)begin
    if(reset)begin
        a0_reg <= 0;
        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)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

assign p = p0_reg + p1_reg+c;

always @(posedge clock)begin
    if(reset)begin
        s_reg <= 0;
    end
      else begin
        if(ce) begin
            s_reg <= p;
        end
    end
end

assign s = s_reg;
endmodule
```

6.5.2 ∑(A0*B0±A1*B1)功能

以经过 PIPE0_REG、PIPE1_REG、OUT_REG、异步复位模式的 MULTADDALU18X18 介绍 DOUT= ∑(A0*B0±A1*B1)功能的实现，其编码形式可如下所示：

```
module acc(a0, a1, b0, b1, s, accload, ce, reset, clk);
parameter a_width=18;
parameter b_width=18;
parameter s_width=54;
input unsigned [a_width-1:0] a0, a1;
input unsigned [b_width-1:0] b0, b1;
input accload, ce, reset, clk;
output unsigned [s_width-1:0] s;
wire unsigned [s_width-1:0] s_sel;
wire unsigned [s_width-1:0] p0, p1;
reg unsigned [s_width-1:0] p0_reg=54'h00000000000000;
```

```
reg unsigned [s_width-1:0] p1_reg=54'h00000000000000;
reg unsigned [s_width-1:0] s=54'h00000000000000;
reg acc_reg0=1'b0;
reg acc_reg1=1'b0;

assign p0 = a0*b0;
assign p1 = a1*b1;
always @(posedge clk or posedge reset)begin
    if(reset) begin
        p0_reg <= 0;
        p1_reg <= 0;
    end else if(ce) begin
        p0_reg <= p0;
        p1_reg <= p1;
    end
end
always @(posedge clk)begin
     if(ce) begin
        acc_reg0 <= accload;
        acc_reg1 <= acc_reg0;
    end
end
assign s_sel = (acc_reg1 == 1) ? s : 0;
always @(posedge clk or posedge reset)begin
    if(reset) begin
        s <= 0;
    end else if(ce) begin
        s <= s_sel + p0_reg - p1_reg;
    end
end
endmodule
```

6.5.3 A0*B0±A1*B1+CASI 功能

以经过 PIPE0_REG、PIPE1_REG、OUT_REG、异步复位模式的 MULTADDALU18X18 介绍 DOUT= A0*B0±A1*B1+CASI 功能的实现，其编码形式可如下所示：

```
module top(a0, a1, a2, b0, b1, b2, a3, b3, s, clock, ce, reset);
parameter a_width=18;
parameter b_width=18;
parameter s_width=54;
input signed [a_width-1:0] a0, a1, a2, b0, b1, b2, a3, b3;
input clock, ce, reset;
output signed [s_width-1:0] s;
wire signed [s_width-1:0] p0, p1, p2, p3, s0, s1;
reg signed [s_width-1:0] p0_reg=54'h00000000000000;
reg signed [s_width-1:0] p1_reg=54'h00000000000000;
reg signed [s_width-1:0] p2_reg=54'h00000000000000;
reg signed [s_width-1:0] p3_reg=54'h00000000000000;
reg signed [s_width-1:0] s0_reg=54'h00000000000000;
reg signed [s_width-1:0] s1_reg=54'h00000000000000;
reg signed [a_width-1:0] a0_reg=18'h00000;
reg signed [a_width-1:0] a1_reg=18'h00000;
reg signed [a_width-1:0] a2_reg=18'h00000;
reg signed [a_width-1:0] a3_reg=18'h00000;
reg signed [a_width-1:0] b0_reg=18'h00000;
reg signed [a_width-1:0] b1_reg=18'h00000;
reg signed [a_width-1:0] b2_reg=18'h00000;
reg signed [a_width-1:0] b3_reg=18'h00000;
always @(posedge clock or posedge reset)begin
    if(reset)begin
        a0_reg <= 0;
        a1_reg <= 0;
        a2_reg <= 0;
        a3_reg <= 0;
        b0_reg <= 0;
        b1_reg <= 0;
```

```
            b2_reg <= 0;
            b3_reg <= 0;
        end else begin
            if(ce)begin
                a0_reg <= a0;
                a1_reg <= a1;
                a2_reg <= a2;
                a3_reg <= a3;
                b0_reg <= b0;
                b1_reg <= b1;
                b2_reg <= b2;
                    b3_reg <= b3;
            end
        end
    end

    assign p0 = a0_reg*b0_reg;
    assign p1 = a1_reg*b1_reg;
    assign p2 = a2_reg*b2_reg;
    assign p3 = a3_reg*b3_reg;

    always @(posedge clock or posedge reset)begin
        if(reset)begin
            p0_reg <= 0;
            p1_reg <= 0;
            p2_reg <= 0;
               p3_reg <= 0;
        end else begin
            if(ce)begin
                p0_reg <= p0;
                p1_reg <= p1;
                p2_reg <= p2;
                p3_reg <= p3;
            end
```

```
    end
end

assign s0 = p0_reg + p1_reg;
always @(posedge clock or posedge reset)begin
    if(reset)begin
        s0_reg <= 0;
    end else begin
        if(ce)begin
            s0_reg <= s0;
        end
    end
end
assign s1 = s0_reg + p2_reg - p3_reg;
always @(posedge clock or posedge reset)begin
    if(reset)begin
        s1_reg <= 0;
    end else begin
        if(ce)begin
            s1_reg <= s1;
        end
    end
end
assign s=s1_reg;
endmodule
```

7 Arora V DSP 编码规范

Arora V DSP（Digital Signal Processing）数字信号处理，可以实现乘法、预加、累加、移位等功能，每个 DSP 有 2 个独立的 clock 时钟信号、2 个独立的 ce 使能信号、2 个独立的 reset 复位信号。

7.1 预加功能

Arora V DSP 可以实现 26 以下位宽的有符号预加操作，支持静态加减操作、动态加减操作以及同步异步复位模式等。根据位宽的不同可以推断为 MULTALU27X18 和 MULT27X36。预加操作需与乘法操作相配合才可推断出来。

7.1.1 静态预加减功能

以 MULTALU27X18 为例介绍静态预加减功能的实现，其编码形式可如下所示：

```
module top(a,b,d,out);
input signed[15:0]a;
input signed[15:0]b;
input signed[15:0]d;
output signed[31:0]out;
assign out=(a+d)*b;
endmodule
```

7.1.2 动态预加减功能

以 MULTALU27X18 为例介绍动态预加减功能的实现，其编码形式可如下所示：

```
module top(clk,ce,reset,a,b,d,psel,out);
input signed[15:0]a;
input signed[15:0]b;
```

```
input signed[15:0]d;
input psel;
input [1:0]clk,ce,reset;
output reg signed[31:0]out;
reg signed[31:0]preg;
always@(posedge clk[0])begin
    if(reset[0])begin
        preg<=0;
    end
    else begin
        if(ce[0])begin
            preg<=psel ? (a+d)*b : (a-d)*b;
        end
    end
end

always@(posedge clk[1])begin
    if(reset[1])begin
        out<=0;
    end
    else begin
        if(ce[1])begin
            out<=preg;
        end
    end
end
endmodule
```

7.2 乘法功能

Arora V DSP 可以实现 27X36 以下位宽的有符号乘法操作，根据位宽不同可以推断为MULTALU27X18、MULT12X12、MULT27X36。以经过AREG、BREG、PIPE_REG、OUT_REG、异步复位模式的 MULTALU27X18 为例介绍其实现，其编码形式可如下所示：

```
module top(a,b,clk,ce,rst,out);
```

```
input signed[26:0]a;
input signed[17:0]b;
input    [1:0]clk,ce,rst;
output signed[34:0]out;
reg signed[26:0]a_reg;
reg signed[17:0]b_reg;
reg signed[34:0]pp_reg,out_reg;
always@(posedge clk[0] or posedge rst[0])begin
    if (rst[0])begin
        a_reg<=0;
        b_reg<=0;
    end
    else begin
        if(ce[0])begin
            a_reg<=a;
            b_reg<=b;
        end
    end
end
always@(posedge clk[1] or posedge rst[1])begin
    if (rst[1])begin
        pp_reg<=0;
    end
    else begin
        if(ce[1])begin
            pp_reg<=a_reg*b_reg;
        end
    end
end
always@(posedge clk[1] or posedge rst[1])begin
    if (rst[1])begin
        out_reg<=0;
    end
    else begin
```

```
            if(ce[1])begin
                out_reg<=pp_reg;
            end
        end
    end

    assign out=out_reg;
    endmodule
```

7.3 乘加功能

Arora V DSP 可以实现 27X18 以下位宽的有符号乘加操作，支持静态的加减操作、动态的加减操作以及同步异步复位模式等。乘加功能根据位宽的不同可以推断为 MULTALU27X18 和 MULTADDALU12X12。

7.3.1 静态加减功能

以静态乘加功能的 MULTADDALU12X12 为例介绍静态加减功能的实现，其编码形式可如下所示：

```
module top(a,b,d0,d1,out);
input signed[11:0]a;
input signed[11:0]b;
input signed[11:0]d0;
input signed[11:0]d1;
output signed[24:0]out;
assign out=a*b + d0*d1;
endmodule
```

7.3.2 动态加减功能

以经过 AREG、BREG、CREG、PREG、OREG、同步复位模式的动态乘加功能的 MULTALU27X18 为例介绍动态加减功能的实现，其编码形式可如下所示：

```
module top(a,b,c,clk,ce,rst,sel,addsub,out);
input signed[11:0]a;
input signed[11:0]b;
input signed[47:0]c;
input   [1:0]clk,ce,rst;
input sel;
```

```
input[1:0] addsub;
output signed[47:0]out;
reg signed[11:0]a_reg;
reg signed[11:0]b_reg;
reg signed[47:0]c_reg;

reg signed[47:0]pp_reg,out_reg;
wire signed[47:0]c_sig;
assign c_sig=sel ? c_reg : 0;
always@(posedge clk[0])begin
    if (rst[0])begin
        a_reg<=0;
        b_reg<=0;
        c_reg<=0;
    end
    else begin
        if(ce[0])begin
            a_reg<=a;
            b_reg<=b;
            c_reg<=c;
        end
    end
end
always@(posedge clk[1])begin
    if (rst[1])begin
        pp_reg<=0;
    end
    else begin
        if(ce[1])begin
            pp_reg<=a_reg*b_reg;
        end
    end
end
always@(posedge clk[1])begin
```

```
        if (rst[1])begin
            out_reg<=0;
        end
        else begin
            if(ce[1])begin
                out_reg<= addsub==2'b00 ? pp_reg + c_sig :
                          addsub==2'b01 ? -pp_reg + c_sig:
                          addsub==2'b10 ? pp_reg - c_sig : -pp_reg-
c_sig;
            end
        end
    end

    assign out=out_reg;
    endmodule
```

7.4 累加功能

Arora V DSP 可以实现 27X18 以下位宽的有符号累加操作，支持静态累加操作、动态累加操作以及同步异步复位模式等。累加功能根据位宽的不同可以推断为 MULTALU27X18 和 MULTADDALU12X12。

7.4.1 静态累加功能

以经过 OREG、同步复位模式的 MULTALU27X18 为例介绍静态累加功能的实现，其编码形式可如下所示：

```
module top(a,b,clk,ce,rst,out);
parameter widtha=27;
parameter widthb=18;

input signed[26:0]a;
input signed[17:0]b;
input clk,ce,rst;
output signed[44:0]out;
reg signed[44:0]out_reg;
wire signed[44:0]s_sel;
wire signed[44:0]mout;
```

```
assign s_sel= out_reg;
assign mout=a*b;
always@(posedge clk)begin
    if (rst)begin
        out_reg<=0;
    end
    else begin
        if(ce)begin
            out_reg<=s_sel+mout;
        end
    end
end
assign out=out_reg;
endmodule
```

7.4.2 动态累加功能

以经过 AREG、BREG、OREG、同步复位模式的 MULTALU27X18 为例介绍动态累加功能的实现，其编码形式可如下所示：

```
module top(a,b,clk,ce,rst,accsel,out);
parameter PRE_LOAD = 48'h000000000012;

input signed[26:0]a;
input signed[17:0]b;
input   [1:0]clk,ce,rst;
input accsel;
output signed[44:0]out;
reg signed[44:0]out_reg;
wire signed[44:0]s_sel;
wire signed[44:0]mout;
reg signed[26:0]a_reg;
reg signed[17:0]b_reg;
always@(posedge clk[0])begin
    if (rst[0])begin
        a_reg<=0;
```

```
            b_reg<=0;
        end
        else begin
            if(ce[0])begin
                a_reg<=a;
                b_reg<=b;
            end
        end
    end

    assign s_sel= accsel == 1'b1 ? out_reg : PRE_LOAD;
    assign mout=a_reg*b_reg;

    always@(posedge clk[1])begin
        if (rst[1])begin
            out_reg<=0;
        end
        else begin
            if(ce[1])begin
                out_reg<=s_sel+mout;
            end
        end
    end

    assign out=out_reg;
    endmodule
```

7.5 级联功能

Arora V DSP 可以实现 27X18 以下位宽的有符号级联操作，支持级联操作的原语有 MULTALU27X18 和 MULTADDALU12X12。

7.5.1 静态级联功能

以 2 个 MULTADDALU12X12 级联为例介绍静态级联功能的实现，其编码形式可如下所示：

```
module top(a,b,d0,d1,a1,b1,d2,d3,out);
```

```
input signed[11:0]a,a1;
input signed[11:0]b,b1;
input signed[11:0]d0,d2;
input signed[11:0]d1,d3;
output signed[24:0]out;
assign out=a*b + d0*d1+a1*b1 + d2*d3;
endmodule
```

7.5.2 动态级联功能

以 2 个 MULTALU27X18 级联为例介绍动态级联功能的实现，其编码形式可如下所示：

```
module top(clk,ce,rst,sel,a,b,a1,b1,out);
parameter widtha=27;
parameter widthb=18;
parameter widthout=widtha+widthb;

input signed[widtha-1:0]a,a1;
input signed[widthb-1:0]b,b1;
input [1:0]clk,ce,rst;
input sel;
output reg signed[widthout:0]out;
reg signed[widtha-1:0]a_reg,a1_reg;
reg signed[widthb-1:0]b_reg,b1_reg;
reg signed[widthout-1:0]p0_reg,p1_reg;

always@(posedge clk[0])begin
    if(rst[0])begin
        a_reg <= 0;
        a1_reg <= 0;
        b_reg <= 0;
        b1_reg <= 0;
    end
    else begin
        if(ce[0])begin
```

```
                a_reg <= a;
                a1_reg <= a1;
                b_reg <= b;
                b1_reg <= b1;
            end
        end
    end

    always@(posedge clk[1])begin
        if(rst[1])begin
            p0_reg <= 0;
            p1_reg <= 0;
        end
        else begin
            if(ce[1])begin
                p0_reg <= a_reg*b_reg;
                p1_reg <= a1_reg*b1_reg;
            end
        end
    end

    always@(posedge clk[1])begin
        if(rst[1])begin
            out <= 0;
        end
        else begin
            if(ce[1])begin
                out <= sel ? p0_reg + p1_reg : p1_reg;
            end
        end
    end
    endmodule
```

7.6 移位功能

Arora V DSP 可以实现 27X18 以下位宽的有符号移位操作，支持移位操作的原语有 MULTALU27X18。

以输入 a 经过两级 register 为例介绍移位功能的实现，其编码形式可如下所示：

```
module top(a, d0, d1, b0, b1, p0, p1, clk, ce, rst);
input signed[25:0] a,d0,d1;
input signed[17:0] b0,b1;
input clk, ce, rst;
output signed [44:0] p0, p1;
wire signed [26:0] paddsub0, paddsub1;
reg signed [25:0] a_reg0, a_reg1,a_reg2,d0_reg,d1_reg;

always @(posedge clk)
begin
    if(rst)begin
        a_reg0 <= 0;
        a_reg1 <= 0;
           a_reg2 <= 0;
        d0_reg <= 0;
        d1_reg <= 0;
    end else begin
        if(ce)begin
                a_reg0 <= a;
                a_reg1 <= a_reg0;
                a_reg2 <= a_reg1;
                d0_reg <= d0;
                d1_reg <= d1;
        end
    end
end
assign paddsub0 = a_reg0 + d0_reg;
assign paddsub1 = a_reg2 + d1_reg;
```

```
assign p0 = paddsub0 * b0;
assign p1 = paddsub1 * b1;
endmodule
```


GOWIN高云
智慧逻辑 定制未来