Gowin HDL 编码风格

用户指南

SUG949-1.0,2020-08-31

版权所有 © 2020 广东高云半导体科技股份有限公司

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、翻译本文档内容的部分或全部，并不得以任何形式传播。

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，或以禁止发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。

目录

目录..i
图目录..v
表目录..vi
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..1
2 HDL 编码风格要求..3
2.1 HDL 编码总体要求...3
2.1.1 分层次编码综合要求...3
2.1.2 流水线编码设计要求...3
2.1.3 并行条件和优先级条件比较...3
2.1.4 避免生成锁存器...4
2.1.5 全局复位和局部复位...4
2.1.6 时钟使能...4
2.1.7 选择器...4
2.1.8 双向缓存...4
2.1.9 跨时钟域处理...4
2.1.10 BSRAM 和 SSRAM 编码风格...5
2.1.11 DSP 编码风格..5
2.1.12 资源共享..6
2.2 FSM 状态机编码要求...6
2.2.1 状态机总体描述...6
2.2.2 状态机状态编码方式...6
2.2.3 安全状态机的初始状态值和默认状态值...6
2.2.4 完整条件和并行条件...7

2.3 低功耗编码 7
2.4 避免综合和仿真不一致的编码要求 7
2.4.1 敏感列表 7
2.4.2 阻塞赋值和非阻塞赋值 7
2.4.3 信号扇出 7
3 LVDS 编码规范 8
3.1 TLVDS_IBUF 8
3.2 ELVDS_IBUF 8
3.3 TLVDS_OBUF 9
3.4 ELVDS_OBUF 9
3.5 TLVDS_TBUF 9
3.6 ELVDS _TBUF 9
3.7 TLVDS_IOBUF 10
3.8 ELVDS _IOBUF 10
4 CLU 编码规范 12
4.1 LUT 12
4.1.1 查找表形式 12
4.1.2 选择器形式 12
4.1.3 逻辑运算形式 12
4.2 ALU 13
4.2.1 ADD 功能 13
4.2.2 SUB 功能 13
4.2.3 ADDSUB 功能 13
4.2.4 NE 功能 14
4.3 FF 14
4.3.1 DFF 15
4.3.2 DFFE 15
4.3.3 DFFSE 15
4.3.4 DFFRE 15
4.3.5 DFFPE 16
4.3.6 DFFCE 16
4.3.7 DFFN 17
4.3.8 DFFNE 17
4.3.9 DFFNSE 17
4.3.10 DFFNRE 18
4.3.11 DFFNPE 18
4.3.12 DFFNCE 18

4.4 LATCH 19
4.4.1 DL 19
4.4.2 DLE 20
4.4.3 DLCE 20
4.4.4 DLPE 20
4.4.5 DLN 21
4.4.6 DLNE 21
4.4.7 DLNCE 21
4.4.8 DLNPE 22
5 BSRAM 编码规范 23
5.1 DPB/DPX9B 23
5.1.1 读地址经过 register 23
5.1.2 读地址不经过 register 24
5.1.3 memory 定义时赋初值 26
5.1.4 readmemb/readmemh 方式赋初值 27
5.2 SP/SPX9 29
5.2.1 读地址经过 register 29
5.2.2 读地址不经过 register 30
5.2.3 memory 定义时赋初值 30
5.2.4 readmemb/readmemh 方式赋初值 31
5.2.5 移位寄存器形式 33
5.2.6 Decoder 形式 33
5.3 SDPB/SDPX9B 34
5.3.1 memory 无初值 35
5.3.2 memory 定义时赋初值 35
5.3.3 readmemb/readmemh 方式赋初值 36
5.3.4 移位寄存器形式 37
5.3.5 不对称类型 38
5.3.6 Decoder 形式 39
5.4 pROM/pROMX9 40
5.4.1 case 语句赋初值 40
5.4.2 memory 定义时赋初值 42
5.4.3 readmemb/readmemh 方式赋初值 42
6 SSRAM 编码规范 45
6.1 RAM16S 类型 45
6.1.1 Decoder 形式 45

6.1.2 Memory 形式 46
6.1.3 移位寄存器形式 47
6.2 RAM16SDP 类型 48
6.2.1 Decoder 形式 48
6.2.2 Memory 形式 49
6.2.3 移位寄存器形式 49
6.3 ROM16 50
6.3.1 Decoder 形式 50
6.3.2 Memory 形式 51
7 DSP 编码规范 52
7.1 Pre-adder 52
7.1.1 预加功能 52
7.1.2 预减功能 54
7.1.3 移位功能 55
7.2 Multiplier 57
7.3 ALU54D 58
7.4 MULTALU 59
7.4.1 A*B±C 功能 59
7.4.2 ∑(A*B)功能 61
7.4.3 A*B+CASI 功能 62
7.5 MULTADDALU 63
7.5.1 A0*B0±A1*B1±C 功能 64
7.5.2 ∑(A0*B0±A1*B1)功能 66
7.5.3 A0*B0±A1*B1+CASI 功能 67

图目录

图 2-1 Gowin 云源软件 IP Core Generator 存储器 5
图 2-2 Gowin 云源软件 IP Core Generator DSP 6

表目录

表 1-1 术语、缩略语 1
表 4-1 FF 相关的原语 14
表 4-2 LATCH 相关的原语 19
表 6-1 SSRAM 相关的原语 45

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
HDL	Hardware Description Language	硬件描述语言
FSM	Finite State Machine	有限状态机
DRC	Design Rule Check	设计规则检查
CLU	Configurable Logic Unit	可配置逻辑单元
CLS	Configurable Logic Slice	可配置逻辑片
BSRAM	Block SRAM	块状静态随机存储器
SSRAM	Shadow SRAM	附加静态随机存储器
DSP	Digital Signal Processor	数字信号处理器

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 HDL 编码风格要求

2.1 HDL 编码总体要求

2.1.1 分层次编码综合要求

对于复杂的系统级设计，分层次编码是很有必要的。分层次编码可以一次综合所有的模块，也可以按层次结构分别综合各个模块。当一次综合所有模块时，设计可以被综合成一个无层次结构的模块或者多个带层次结构的模块。

这两个策略各有优缺点，在复杂的系统级设计中分层次编码更具优势，原因在于分层次编码更容易定位问题，同时提高了模块的重用性并缩短了开发周期。下面是创建分层次编码结构的一些建议。

- 顶层模块只用于调用子模块，控制逻辑尽量在各个子模块实现；
- 管脚输入输出缓存需要在顶层例化；
- 所有的输入、输出和双向管脚需要在顶层例化；
- 只允许在顶层使用双向管脚的三态声明；
- 尽可能保证模块所有输出信号全部使用寄存器，这样的好处在于：
  - 把相关逻辑放在一个模块里，可以保证资源共享并优化关键路径。
  - 分割不相关逻辑到不同模块，则可以采用不同的优化策略，比如速度优先或者面积优先。

2.1.2 流水线编码设计要求

流水线设计通过重构多逻辑级数的数据路径并增加时钟周期分割逻辑级数达到提高设计性能的目的。流水线结构是提高数据路径速度的有效方法，但需要注意的是它会增加数据路径的传输时延。

2.1.3 并行条件和优先级条件比较

if-then-else 结构生成优先级条件逻辑，而 case 结构生成并行条件逻辑。if-then-else 结构可以包含多个不同的条件表达式，但是 case 结构只能包含一个表达式。在条件互斥的情况下，可以认为 if-then-else 结构和 case 结构

是等效的。

2.1.4 避免生成锁存器

用户在 FPGA 设计中应该避免使用锁存器，综合工具通过组合逻辑环路实现锁存器，不可避免地造成资源浪费和性能下降，同时组合逻辑环路给静态时序分析带来了很大困难。

综合工具在组合逻辑条件表达式不完整情况下会生成锁存器，比如 if-then-else 结构中没有 else 或者 case 结构中没有 default。为了避免不必要的锁存器，在条件语句中需要把所有条件都遍历到。

2.1.5 全局复位和局部复位

高云器件包含一个专用的全局复置位网络，直接连接到器件的内部逻辑，可用作异步/同步复位或异步/同步置位，CLU 和 I/O 中的寄存器均可以独立配置。全局复位资源并不占用普通绕线资源，并且遍布芯片各个角落。在设计中主要有两种方法调用全局复位资源，即使用全局复位去控制 FPGA 中的所有模块或者选用扇出最大的复位作为全局复位以节省绕线资源。局部复位一般扇出较小，可以考虑使用普通绕线作为复位信号。

2.1.6 时钟使能

FPGA 设计中应该避免使用门控时钟，门控时钟会导致很多不可控的时序问题，比如不可预知的时钟飘移。Gowin 器件 CLS 结构中包含专用时钟使能信号，用户可以使用时钟使能达到门控时钟同样的功能而不必担心时序问题。以下是使用 Gowin 器件时钟使能的一些注意事项。

- 只有寄存器支持时钟使能，锁存器不支持；
- 同一个 CLU 里面的各个 CLS 共用一个时钟使能信号；
- 所有寄存器时钟使能高有效；
- 时钟使能优先级高于同步复位。

2.1.7 选择器

查找表可以灵活配置以实现二选一选择器、三选一选择器、四选一选择器或五选一选择器等等，用户还可以通过调用多个四输入查找表级联实现更大的选择器。

2.1.8 双向缓存

使用双向缓存可以节约管脚，控制输出使能达到节省功耗的目的。用户可以关闭综合工具自动管脚输入输出缓存插入选项，为特定的管脚例化特定的管脚输入输出缓存。

2.1.9 跨时钟域处理

当数据路径从一个时钟域跨越到另一个时钟域时，需小心处理信号的跨

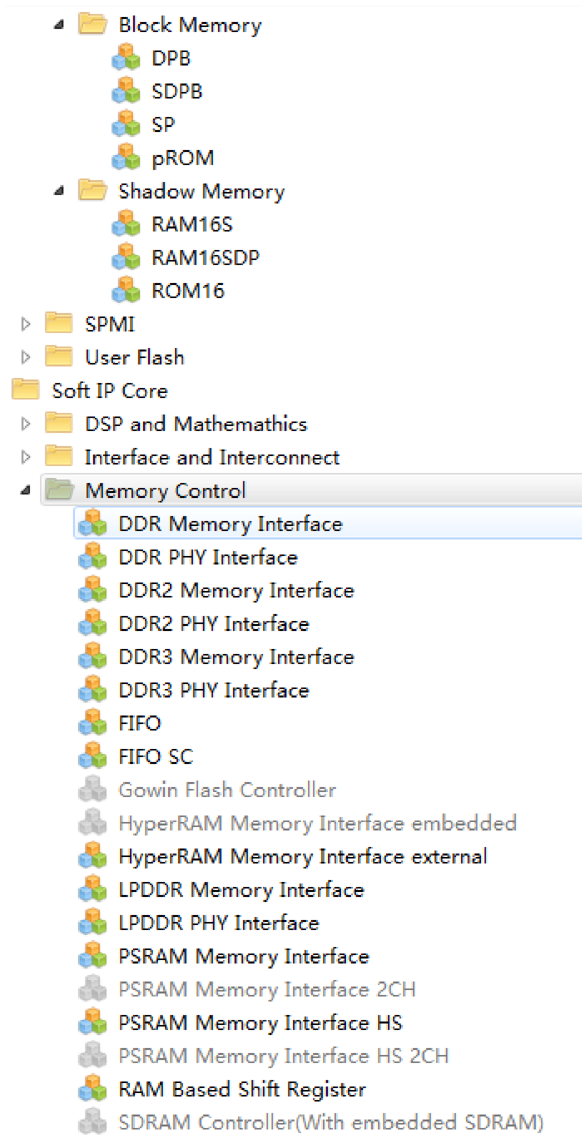
时钟域传输，避免亚稳态的产生。对于单比特信号，建议使用三级寄存器结构消除亚稳态；对于多比特信号，建议使用异步 FIFO。

2.1.10 BSRAM 和 SSRAM 编码风格

虽然随机存储器的 RTL 行为级描述可定制且非常直观，但不同的编码风格可能会产生不同的综合结果。

对于 Gowin 器件而言，建议使用 Gowin 云源软件 IP Core Generator 工具产生 BSRAM 和 SSRAM。Gowin 器件支持多种存储结构，包括双端口 RAM、单端口 RAM、伪双端口 RAM、ROM、同步 FIFO 以及异步 FIFO，如图 2-1 所示。

图 2-1 Gowin 云源软件 IP Core Generator 存储器



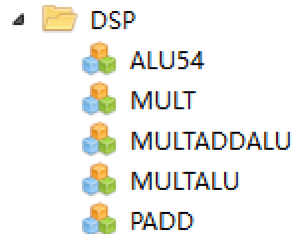
2.1.11 DSP 编码风格

虽然 DSP 的 RTL 行为级描述可定制且非常直观，但不同的编码风格可

能会产生不同的综合结果。

对于 Gowin 器件而言，建议使用 Gowin 云源软件 IP Core Generator 工具产生 DSP。Gowin 器件支持多种 DSP 结构，包括 MULT、MULTALU、MULTADDALU 以及 PADD。

图 2-2 Gowin 云源软件 IP Core Generator DSP



2.1.12 资源共享

SynplifyPro 综合工具默认支持资源共享，这样的好处是可以节约资源，但是资源共享会增加绕线资源并造成局部拥塞。当出现时序收敛问题时，建议把全局资源共享设置关闭，并对特定的模块使用综合属性/*synthesis syn_sharing = "on" */。

2.2 FSM 状态机编码要求

有限状态机在时钟边沿完成当前状态到下一个状态的跳转，下面主要讨论状态机编码的方法和策略。

2.2.1 状态机总体描述

状态机实现主要有两种方式，一种是在一个进程中同时处理状态跳转和状态输出；另一种则在两个独立的进程中分别处理状态跳转和状态输出。建议用户使用第二种方式，它不仅便于阅读和修改，而且对于状态无寄存直接输出情况，第二种方式可以保证不会引起额外的时延。

2.2.2 状态机状态编码方式

目前主要有二进制编码、独热编码以及格雷编码三种状态机编码方式。二进制编码及格雷编码使用较少的触发器，但使用较多的组合逻辑，而独热编码恰好相反。

独热编码的最大优势在于状态比较时仅仅需要比较一个比特位，从而一定程度上简化了译码逻辑。虽然在表示同等数量的状态时，独热编码占用较多的比特位，即使用较多的触发器，但这些触发器占用的面积可以和译码电路省下来的面积相抵消。

因此，对于状态数小于 5 的状态机，建议使用二进制编码和格雷编码；对于状态数大于 5 的状态机，建议使用独热编码。

2.2.3 安全状态机的初始状态值和默认状态值

安全状态机在上电后必须初始化进入一个初始有效状态，用户可以通过

上电复位或全局复位操作进入一个初始有效状态。同时状态机必须有一个默认状态值来保证状态机不会进入非法状态，当代码中有状态组合没有被遍历时，非法状态就会产生。

2.2.4 完整条件和并行条件

RTL 语言有专门的属性来定义默认状态，完整条件可以确保不会出现非法状态；并行条件则确保所有状态都是互斥的，在同一时间只有一个条件满足。

2.3 低功耗编码

以面积为优化目标可以降低逻辑和布线资源使用率，进而降低功耗，建议使用Gowin云源软件IP Core Generator去调用Gowin器件内部基本单元，以得到低功耗的实现，消除已知的毛刺以降低电源功耗，降低信号翻转率并在适当时刻进入休眠状态可降低系统功耗。

2.4 避免综合和仿真不一致的编码要求

特定编码风格会导致综合和仿真的结果不一致，原因在于仿真工具会忽略一些错误信息而这些错误信息会被综合工具检测出来，这些错误一般可以通过 BKM 检测出来，因此编码风格建议采用 Gowin 编码风格。

2.4.1 敏感列表

组合逻辑的信号敏感列表必须包含所有输入输出信号以确保综合和仿真结果一致。

2.4.2 阻塞赋值和非阻塞赋值

组合逻辑一般建议使用阻塞赋值；时序逻辑一般建议使用非阻塞赋值。

2.4.3 信号扇出

信号扇出控制用于保证综合后的扇出在一个合适的范围，综合工具通过复制电路来减少信号扇出。对特定信号使用综合属性 syn_maxfan 可以得到更好的时序收敛效果。Gowin 器件架构可以使用专用时钟网络处理大扇出的时钟信号，并使用专用全局复位网络处理大扇出的复位信号。但是综合工具倾向于对大扇出的逻辑进行电路复制，复制电路会带来一系列副作用，用户需要根据实际情况灵活使用综合属性 syn_maxfan 以避免生成过多的复制电路。

3 LVDS 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。

3.1 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(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
```

3.2 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
```

3.3 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
```

3.4 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
```

3.5 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
```

3.6 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;
input 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
```

3.7 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
```

3.8 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 tlvds_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
```

4 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。

4.1 LUT

输入查找表 LUT，常用的 LUT 结构有 LUT1、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：

4.1.1 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
```

4.1.2 选择器形式

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

4.1.3 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
```

4.2 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。

4.2.1 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
```

4.2.2 SUB 功能

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

4.2.3 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
```

4.2.4 NE 功能

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

4.3 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，原语介绍详见表 4-1，本节以 DFF、DFFE、DFFSE、DFFRE、DFFPE、DFFCE、DFFN、DFFNE、DFFNSE、DFFNRE、DFFNPE、DFFNCE 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。

表 4-1 FF 相关的原语

原语	描述	初始值
DFF	D 触发器	1'b0
DFFE	带时钟使能 D 触发器	1'b0
DFFS	带同步置位 D 触发器	1'b1
DFFSE	带时钟使能、同步置位 D 触发器	1'b1
DFFR	带同步复位 D 触发器	1'b0
DFFRE	带时钟使能、同步复位 D 触发器	1'b0
DFFP	带异步置位 D 触发器	1'b1
DFFPE	带时钟使能、异步置位 D 触发器	1'b1
DFFC	带异步复位 D 触发器	1'b0
DFFCE	带时钟使能、异步复位 D 触发器	1'b0
DFFN	下降沿 D 触发器	1'b0
DFFNE	下降沿带时钟使能 D 触发器	1'b0
DFFNS	下降沿带同步置位 D 触发器	1'b1
DFFNSE	下降沿带时钟使能、同步置位 D 触发器	1'b1
DFFNR	下降沿带同步复位 D 触发器	1'b0
DFFNRE	下降沿带时钟使能、同步复位 D 触发器	1'b0
DFFNP	下降沿带异步置位 D 触发器	1'b1
DFFNPE	下降沿带时钟使能、异步置位 D 触发器	1'b1
DFFNC	下降沿带异步复位 D 触发器	1'b0
DFFNCE	下降沿带时钟使能、异步复位 D 触发器	1'b0

4.3.1 DFF

```
module dff (clk, d, q );
input clk, d;
output   reg q=1'b0;
always @(posedge clk)begin
      q <= d;
end
endmodule
```

4.3.2 DFFE

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

4.3.3 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
```

4.3.4 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
```

4.3.5 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
```

4.3.6 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
```

4.3.7 DFFN

```
module dffn (clk, d, q );
input clk, d;
output   reg q= 1'b0;
always @(negedge clk)begin
      q <= d;
end
endmodule
```

4.3.8 DFFNE

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

4.3.9 DFFNSE

```
module dffnse_init1 (clk, d, ce, set, q );
input clk, d, ce, set;
output   reg q= 1'b1;
always @(negedge clk)begin
      if (set)begin
            q <= 1'b1;
      end
      else begin
            if (ce)begin
                  q <= d;
```

```
            end
        end
    end
endmodule
```

4.3.10 DFFNRE

```
module dffnre_init1 (clk, d, ce, rst, q );
input clk, d, ce, rst;
output   reg q= 1'b0;
always @(negedge clk)begin
        if (rst)begin
            q <= 1'b0;
        end
        else begin
            if (ce)begin
                q <= d;
            end
        end
    end
endmodule
```

4.3.11 DFFNPE

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

4.3.12 DFFNCE

```
module dffnce_test (clk, d, ce, clear, q );
```

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

4.4 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。原语介绍详见表 4-2 所示，本节以 DL、DLE、DLCE、DLPE、DLN、DLNE、DLNCE、DLNPE 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。

表 4-2 LATCH 相关的原语

原语	描述	初始值
DL	数据锁存器	1'b0
DLE	带锁存使能的数据锁存器	1'b0
DLC	带异步清零的数据锁存器	1'b0
DLCE	带异步清零和锁存使能的数据锁存器	1'b0
DLP	带异步预置位的数据锁存器	1'b1
DLPE	带异步预置位和锁存使能的数据锁存器	1'b1
DLN	低电平有效的数据锁存器	1'b0
DLNE	带锁存使能的低电平有效的数据锁存器	1'b0
DLNC	带异步清零的低电平有效的数据锁存器	1'b0
DLNCE	带异步清零和锁存使能的低电平有效的数据锁存器	1'b0
DLNP	带异步预置位的低电平有效的数据锁存器	1'b1
DLNPE	带异步预置位和锁存使能的低电平有效的数据锁存器	1'b1

4.4.1 DL

```
module rtl_DL (Q, D, G);
input D, G;
output reg Q=1'b0;
always @(D or G ) begin
```

```
    if (G)begin
        Q <= D;
    end
end
endmodule
```

4.4.2 DLE

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

4.4.3 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
```

4.4.4 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
```

4.4.5 DLN

```
module rtl_DLN (Q, D, G);
input D, G;
output reg Q= 1'b0;
always @(D or G) begin
    if (!G )begin
        Q <= D;
    end
end
endmodule
```

4.4.6 DLNE

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

4.4.7 DLNCE

```
module rtl_DLNCE (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
```

4.4.8 DLNPE

```
module rtl_DLNPE (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
```

5 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）。

5.1 DPB/DPX9B

DPB/DPX9B 的存储空间分别为 16K bit/18K bit，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 3 种写模式（normal 模式、write-through 模式和 read-before-write 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。

5.1.1 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 normal(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
```

5.1.2 读地址不经过 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 normal(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 reg [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
```

5.1.3 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 read-before-write、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 normal(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)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)begin
                data_outb <= mem[addrb];
            end
        end
    end

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

5.1.4 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 read-before-write、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 normal(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)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)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。

5.2 SP/SPX9

SP/SPX9 存储空间为 16K bit/18K bit，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 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。

5.2.1 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 normal(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
```

5.2.2 读地址不经过 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 wt(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
```

5.2.3 memory 定义时赋初值

memory 定义时赋初值需 Verilog language 选择 system Verilog。以

read-before-write、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 rbw(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)begin
            data_out <= mem[addr];
        end
    end
end
always @(posedge clk)begin
    if (ce & wre)begin
        mem[addr] <= data_in;
    end
end
endmodule
```

5.2.4 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 normal(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
```

5.2.5 移位寄存器形式

移位寄存器形式综合出 SP/SPX9 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+1；
2. 添加属性约束 syn_srlstyle= "block_ram"，且 memory 深度=2 的 n 次方+1。

以 read-before-write、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   p_seqshift(clk, we, din, dout);
parameter width=18;
parameter depth=17;
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
```

5.2.6 Decoder 形式

以 read-before-write、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
            data_out[0] <= mem0[addr];
            data_out[1] <= mem1[addr];
            data_out[2] <= mem2[addr];
            data_out[3] <= mem3[addr];
      end
end
endmodule
```

5.3 SDPB/SDPX9B

SDPB/SDPX9B 存储空间分别为 16K bit/18K bit，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 需满足下述条件之一：

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

5.3.1 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 normal(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
```

5.3.2 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 normal(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
```

5.3.3 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 normal(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
```

5.3.4 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>=3 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。

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
```

5.3.5 不对称类型

不对称类型综合出 SDPB，需添加属性约束 syn_ramstyle = "no_rw_check"。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 normal(dout, din, ada, clka, cea, adb, clkb, ceb,
rstb,oce)/*synthesis syn_ramstyle = "no_rw_check"*/;
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
```

5.3.6 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
```

5.4 pROM/pROMX9

pROM/pROMX9(16K/18K Block ROM)，16K/18K 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
2. 使用 syn_romstyle = "block_rom"

5.4.1 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 normal (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
```

5.4.2 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 prom_inference ( 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
```

5.4.3 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 rom_inference ( 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

6 SSRAM 编码规范

SSRAM 是附加静态随机存储器，可配置成单端口模式，伪双端口模式和只读模式，如表 6-1 所示。

表 6-1 SSRAM 相关的原语

原语	描述
RAM16S1	地址深度 16，数据宽度为 1 的单端口 SSRAM
RAM16S2	地址深度 16，数据宽度为 2 的单端口 SSRAM
RAM16S4	地址深度 16，数据宽度为 4 的单端口 SSRAM
RAM16SDP1	地址深度 16，数据宽度为 1 的伪双端口 SSRAM
RAM16SDP2	地址深度 16，数据宽度为 2 的伪双端口 SSRAM
RAM16SDP4	地址深度 16，数据宽度为 4 的伪双端口 SSRAM
ROM16	地址深度 16，数据宽度为 1 的只读 ROM

6.1 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 需满足以下其中一个条件：

1. 读地址和输出不经过 register，地址深度*数据位宽>=8；
2. 读地址或输出经过 register，8<=地址深度*数据位宽<=1024；
3. 使用属性约束 syn_ramstyle="distributed_ram"。

6.1.1 Decoder 形式

以 RAM16S4 为例，介绍 Decode 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
```

6.1.2 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 5.1.4，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 normal(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
```

6.1.3 移位寄存器形式

GowinSynthesis 和 synplifyPro 将移位寄存器形式设计综合为 RAM16S 的条件不同：

GowinSynthesis 需满足以下条件之一：

1. memory 深度>=4 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。

synplifyPro 需满足以下条件之一：

1. memory 深度>=4 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= " distributed_ram"，且 memory 深度=2 的 n 次方+1。

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
```

6.2 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 需满足以下其中一个条件：

1. 读地址和输出不经过 register，地址深度*数据位宽>=8；
2. 读地址或输出经过 register，8<=地址深度*数据位宽<=1024；
3. 使用属性约束 syn_ramstyle="distributed_ram"。

6.2.1 Decoder 形式

以 RAM16SDP4 为例，介绍 Decode 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
```

6.2.2 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 5.1.4，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 normal(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
```

6.2.3 移位寄存器形式

GowinSynthesis 和 synplifyPro 将移位寄存器形式设计综合为 RAM16SDP 的条件不同：

GowinSynthesis 需满足以下条件之一：

1. memory 深度>=4 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。

synplifyPro 需满足以下条件之一：

1. memory 深度>=4 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= " distributed_ram"，且 memory 深度=2 的 n 次方+1。

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
```

6.3 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"。

6.3.1 Decoder 形式

```
module test (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
```

6.3.2 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 5.1.4，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
```

7 DSP 编码规范

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

7.1 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 无法单独实现，需与 Multiplier 相配合才可实现出来。

7.1.1 预加功能

以 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
```

7.1.2 预减功能

以 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
```

7.1.3 移位功能

以 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 = b0_reg + b1_reg;
assign b1_padd= bX1 + bY1;

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

endmodule
```

7.2 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 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
```

7.3 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 ALU5D，数据位宽 width 需在[48,54]区间，否则需添加属性约束 syn_dspstyle="dsp"。以 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
```

7.4 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。

7.4.1 A*B±C 功能

以 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
```

7.4.2 ∑(A*B)功能

以 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,accload);
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,accload;
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

assign c_sel = (accload == 1) ? c : 0;
always @(posedge clock or posedge reset)
begin
```

```
    if(reset) begin
        out_reg <= 0;
    end else if(ce) begin
        out_reg <= c_sel + pp_reg;
    end
end

assign c=out_reg;
endmodule
```

7.4.3 A*B+CASI 功能

以 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
```

7.5 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。

7.5.1 A0*B0±A1*B1±C 功能

以 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
```

7.5.2 ∑(A0*B0±A1*B1)功能

以 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
```

7.5.3 A0*B0±A1*B1+CASI 功能

以 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
```

GOWIN高云
智慧逻辑 定制未来