


# Gowin HDL コーディングスタイル ユーザーガイド

## 著作権について(2023)

著作権に関する全ての権利は、**Guangdong Gowin Semiconductor Corporation** に留保されています。

**GOWIN高云**、、Gowin、GowinSynthesis、及びGOWINSEMIは、当社により、中国、米国特許商標庁、及びその他の国において登録されています。商標又はサービスマークとして特定されたその他全ての文字やロゴは、それぞれの権利者に帰属しています。何れの団体及び個人も、当社の書面による許可を得ず、本文書の内容の一部もしくは全部を、いかなる視聴覚的、電子的、機械的、複写、録音等の手段によりもしくは形式により、伝搬又は複製をしてはなりません。

## 免責事項

当社は、GOWINSEMI Terms and Conditions of Sale (GOWINSEMI取引条件)に規定されている内容を除き、(明示的か又は黙示的に拘わらず)いかなる保証もせず、また、知的財産権や材料の使用によりあなたのハードウェア、ソフトウェア、データ、又は財産が被った損害についても責任を負いません。当社は、事前の通知なく、いつでも本文書の内容を変更することができます。本文書を参照する何れの団体及び個人も、最新の文書やエラッタ(不具合情報)については、当社に問い合わせる必要があります。

## バージョン履歴

日付	バージョン	説明
2020/08/31	1.0J	初版。
2021/06/10	1.1J	シングルエンド <b>Buffer</b> の推論例を追加。
2022/05/31	1.2J	コーディングスタイルの説明を削除。
2023/04/20	1.3J	CHAPTER「 <b>7 Arora V DSP</b> のコーディングスタイル」を追加。
2023/06/30	1.4J	CHAPTER「 <b>4 BSRAM</b> のコーディングスタイル」およびCHAPTER「 <b>5 SSRAM</b> のコーディングスタイル」を更新。

# 目次

目次 .....	i
表一覧 .....	v
<b>1 本マニュアルについて .....</b>	<b>1</b>
1.1 マニュアル内容 .....	1
1.2 関連ドキュメント .....	1
1.3 用語、略語 .....	1
1.4 テクニカル・サポートとフィードバック .....	2
<b>2 Buffer のコーディングスタイル .....</b>	<b>3</b>
2.1 IBUF .....	3
2.2 TLVDS_IBUF .....	3
2.3 ELVDS_IBUF .....	4
2.4 OBUF .....	4
2.5 TLVDS_OBUF .....	5
2.6 ELVDS_OBUF .....	5
2.7 TBUF .....	5
2.8 TLVDS_TBUF .....	6
2.9 ELVDS_TBUF .....	6
2.10 IOBUF .....	6
2.11 TLVDS_IOBUF .....	7
2.12 ELVDS_IOBUF .....	8
<b>3 CLU のコーディングスタイル .....</b>	<b>9</b>
3.1 LUT .....	9
3.1.1 ルックアップテーブル .....	9
3.1.2 マルチプレクサ .....	9
3.1.3 論理演算 .....	9
3.2 ALU .....	10
3.2.1 ADD 機能 .....	10
3.2.2 SUB 機能 .....	10

3.2.3 ADDSUB 機能.....	11
3.2.4 NE 機能 .....	11
3.3 FF .....	11
3.3.1 DFFSE.....	11
3.3.2 DFFRE.....	12
3.3.3 DFFPE.....	12
3.3.4 DFFCE.....	13
3.4 LATCH.....	13
3.4.1 DLCE.....	13
3.4.2 DLPE.....	14
<b>4 BSRAM のコーディングスタイル.....</b>	<b>15</b>
4.1 DPB/DPX9B .....	15
4.1.1 読み出しアドレスがレジスタを経由する.....	15
4.1.2 読み出しアドレスがレジスタを経由しなく、出力がレジスタを経由する.....	16
4.1.3 メモリ定義時の初期値の割り当て .....	18
4.1.4 readmemb/readmemh 文による初期値割り当て .....	20
4.2 SP/SPX9 .....	22
4.2.1 読み出しアドレスがレジスタを経由する.....	22
4.2.2 読み出しアドレスがレジスタを経由しなく、出力がレジスタを経由する.....	23
4.2.3 メモリ定義時の初期値の割り当て .....	24
4.2.4 readmemb/readmemh 文による初期値割り当て .....	25
4.2.5 シフトレジスタの形式.....	26
4.2.6 Decoder の形式 .....	27
4.3 SDPB/SDPX9B .....	28
4.3.1 memory 初期値なし .....	28
4.3.2 メモリ定義時の初期値の割り当て .....	29
4.3.3 readmemb/readmemh 文による初期値割り当て .....	30
4.3.4 シフトレジスタの形式.....	31
4.3.5 非対称タイプ .....	32
4.3.6 Decoder の形式 .....	33
4.4 pROM/pROMX9 .....	34
4.4.1 case 文による初期値割り当て .....	35
4.4.2 メモリ定義時の初期値の割り当て .....	36
4.4.3 readmemb/readmemh 文による初期値割り当て .....	37
<b>5 SSRAM のコーディングスタイル.....</b>	<b>39</b>
5.1 RAM16S .....	39

5.1.1 Decoder の形式 .....	39
5.1.2 Memory の形式 .....	40
5.1.3 シフトレジスタの形式 .....	41
5.2 RAM16SDP .....	42
5.2.1 Decoder の形式 .....	42
5.2.2 Memory の形式 .....	43
5.2.3 シフトレジスタの形式 .....	44
5.3 ROM16 .....	44
5.3.1 Decoder の形式 .....	44
5.3.2 Memory の形式 .....	45
<b>6 DSP のコーディングスタイル<sup>[1]</sup> .....</b>	<b>47</b>
6.1 Pre-adder .....	47
6.1.1 前置加算機能 .....	47
6.1.2 前置減算機能 .....	49
6.1.3 シフト機能 .....	51
6.2 Multiplier .....	52
6.3 ALU54D .....	54
6.4 MULTALU .....	55
6.4.1 $A*B \pm C$ 機能 .....	55
6.4.2 $\sum(A*B)$ 機能 .....	57
6.4.3 $A*B + CASI$ 機能 .....	58
6.5 MULTADDALU .....	60
6.5.1 $A0*B0 \pm A1*B1 \pm C$ 機能 .....	60
6.5.2 $\sum(A0*B0 \pm A1*B1)$ 機能 .....	62
6.5.3 $A0*B0 \pm A1*B1 + CASI$ 機能 .....	64
<b>7 Arora V DSP のコーディングスタイル .....</b>	<b>67</b>
7.1 前置加算機能 .....	67
7.1.1 静的加算/減算機能 .....	67
7.1.2 動的加算/減算機能 .....	67
7.2 乗算機能 .....	68
7.3 乗算加算機能 .....	70
7.3.1 静的加算/減算機能 .....	70
7.3.2 動的加算/減算機能 .....	70
7.4 累積機能 .....	72
7.4.1 静的累積機能 .....	72
7.4.2 動的累積機能 .....	73

---

7.5 カスケード接続機能 .....	75
7.5.1 静的カスケード接続機能 .....	75
7.5.2 動的カスケード接続機能 .....	75
7.6 シフト機能 .....	77

# 表一覽

表 1-1 用語、略語 .....	1
-------------------	---



# 1 本マニュアルについて

## 1.1 マニュアル内容

このマニュアルでは、主に Gowin HDL コーディングスタイルについて説明します。

## 1.2 関連ドキュメント

GOWIN セミコンダクターのホームページ [www.gowinsemi.com/ja](http://www.gowinsemi.com/ja) から、以下の関連ドキュメントをダウンロード及び閲覧できます。

- Gowin ソフトウェア ユーザーガイド([SUG100](#))
- Gowin プリミティブ ユーザーガイド([SUG283](#))

## 1.3 用語、略語

表 1-1 に、本マニュアルで使用される用語、略語、及びその意味を示します。

表 1-1 用語、略語

用語、略語	正式名称	意味
BSRAM	Block Static Random Access Memory	ブロック SRAM
CLU	Configurable Logic Unit	コンフィギュラブル論理ユニット
DSP	Digital Signal Processing	デジタル信号処理
FSM	Finite State Machine	有限状態機械
HDL	Hardware Description Language	ハードウェア記述言語
SSRAM	Shadow Static Random Access Memory	分散 SRAM

## 1.4 テクニカル・サポートとフィードバック

GOWIN セミコンダクターは、包括的な技術サポートをご提供しています。使用に関するご質問、ご意見については、直接弊社までお問い合わせください。

ホームページ : [www.gowinsemi.com/ja](http://www.gowinsemi.com/ja)

E-mail : [support@gowinsemi.com](mailto:support@gowinsemi.com)

# 2 Buffer のコーディングスタイル

Buffer(バッファ)は、機能によってシングルエンド Buffer、エミュレート LVDS(ELVDS)、および True LVDS(TLVDS)に分類できます。エミュレート LVDS および True LVDS のプリミティブ実装は、属性制約を追加する必要があるため、インスタンス化することをお勧めします。

## 2.1 IBUF

IBUF(Input Buffer)は入力バッファです。2 つのコーディング方法を次に示します。

方法 1 :

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

方法 2 :

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

## 2.2 TLVDS\_IBUF

TLVDS\_IBUF(True LVDS Input Buffer)は、真の差動入力バッファです。このプリミティブを実現するためのコーディングを次に示します。

```
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)は、エミュレート差動入力バッファです。このプリミティブを実現するためのコーディングを次に示します。

```
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)は出力バッファです。2つのコーディング方法を次に示します。

方法 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)は、真の差動出力バッファです。このプリミティブを実現するためのコーディングを次に示します。

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

## 2.6 ELVDS\_OBUF

ELVDS\_OBUF(Emulated LVDS Output Buffer)は、エミュレート差動出力バッファです。このプリミティブを実現するためのコーディングを次に示します。

```
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)は、アクティブ Low のトライステートバッファです。2つのコーディング方法を次に示します。

方法 1 :

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

```
endmodule
```

方法 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)は、真の差動トライステートバッファで、アクティブ **Low** です。このプリミティブを実現するためのコーディングを次に示します。

```
module tlvsd_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)は、エミュレート差動トライステートバッファで、アクティブ **Low** です。このプリミティブを実現するためのコーディングを次に示します。

```
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)は双方向バッファです。**OEN** が **High** の場合は入力バッファとして使用され、**Low** の場合は出力バッファとして使用されます。2 つのコーディング方法を次に示します。

方法 1 :

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

方法 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)は、真の双方向バッファです。OEN が High の場合は、真の差動入力バッファとして使用され、OEN が Low の場合は、真の差動出力バッファとして使用されます。このプリミティブを実現するためのコーディングを次に示します。

```
module tlvsd_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)は、エミュレート差動双方向バッファです。OEN が High の場合はエミュレート差動入力バッファとして使用され、OEN が Low の場合はエミュレート差動出力バッファとして使用されます。このプリミティブを実現するためのコーディングを次に示します。

```
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 のコーディングスタイル

コンフィギャラブル論理ユニット(Configurable Logic Unit、CLU)は、FPGA 製品の基本的な構成要素で、CLU モジュールは MUX/LUT/ALU/FF/LATCH などの機能を実現することができます。

## 3.1 LUT

LUT(ルックアップテーブル)には、LUT1、LUT2、LUT3、および LUT4 などがあります。それぞれ異なるビット幅を持っています。LUT の実装方法は次のとおりです。

### 3.1.1 ルックアップテーブル

```
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 マルチプレクサ

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

### 3.1.3 論理演算

```
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)は 2 入力算術論理演算装置で、ADD/SUB/ADDSUB/NE などの機能を実現できます。

### 3.2.1 ADD 機能

4 ビットの全加算器と 4 ビットの半加算器を例として、ALU の ADD 機能を説明します。

#### 4 ビットの全加算器

4 ビットの全加算器を実現するためのコーディングは次に示すとおりです。

```
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 ビットの半加算器

4 ビットの半加算器を実現するためのコーディングは次に示すとおりです。

```
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 機能

```
module sub(a,b,sub);  
input [3:0] a,b;
```

```
output [3:0] sub;  
assign sub=a-b;  
endmodule
```

### 3.2.3 ADDSUB 機能

```
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 機能

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

## 3.3 FF

フリップフロップは、シーケンシャル回路で一般的に使用される基本的なコンポーネントです。FPGA の内部シーケンシャルロジックは、FF 構造によって実現できます。一般的に使用される FF には、DFF、DFFE、DFFS、DFFSE などがあります。これらの FF は、リセットモードやトリガモードなどにおいて異なります。Arora V の FF プリミティブには、DFFSE、DFFRE、DFFPE、および DFFCE のみが含まれます。ここでは、DFFSE、DFFRE、DFFPE、DFFCE を例に説明します。コーディングで reg 信号を定義する際には、初期値を追加することをお勧めします。各タイプのレジスタの初期値については、『Gowin コンフィギュラブル機能ユニット(CFU) ユーザーガイド([UG288](#))』を参照してください。

### 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
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
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
end
endmodule
```

## 3.4 LATCH

ラッチは、1 ビットの情報を保持できる、レベルトリガの回路です。Arora V の LATCH プリミティブには、DLCE と DLPE のみが含まれます。ここでは、DLCE、DLPE を例に説明します。コーディングで **reg** 信号を定義する際には、初期値を追加することをお勧めします。各タイプのラッチの初期値については、『Gowin コンフィギャラブル機能ユニット(CFU)ユーザーガイド([UG288](#))』を参照してください。

### 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 のコーディングスタイル

BSRAM は、静的アクセス機能を備えたブロック状のスタティック RAM です。構成モードにより、シングルポート・モード(SP/SPX9)、デュアルポート・モード(DPB/DPX9B)、セミ・デュアルポート・モード(SDPB/SDPX9B)、および読み出し専用モード(pROM/pROMX9)があります。

## 4.1 DPB/DPX9B

DPB/DPX9B はそれぞれメモリ領域が 16K ビット/18K ビットであるデュアルポート・モードの BSRAM です。A ポートと B ポートは個別に読み出し/書き込みを実現できます。2 種類の読み出しモード(bypass モードと pipeline モード)と 3 種類の書き込みモード(Normal モード、write-through モード、read-before-write モード)がサポートされます。ここでは、読み出しアドレスのレジスタ経由や初期値の読み出しについて説明します。

注記：

55nm プロセスのデバイスでは、read-before-write 書き込みモードのコーディングスタイルは推奨されません。

### 4.1.1 読み出しアドレスがレジスタを経由する

これは、読み出しアドレスレジスタが制御信号によって制御されない場合にのみサポートされます。write-through、bypass、同期リセットモードの DPB を例に説明します。そのコーディングスタイルは次のとおりです。

```
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 clk)begin
    addrb_reg<=addrb;
end

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

```

#### 4.1.2 読み出しアドレスがレジスタを経由しなく、出力がレジスタを経由する

読み出しアドレスがレジスタを経由しない場合、出力はレジスタを経由する必要があります。1 レベルのレジスタを経由する場合は **bypass** モード、2 レベルのレジスタを経由する場合は **pipeline** モードです。**Normal**、**pipeline**、同期リセットモードの **DPB** を例に説明します。そのコーディングスタイルは次のとおりです。

```

module normal(data_outa, data_ina, addra, clka, cea, ocea, wrea, rsta,
data_outb, data_inb, addrb, clk, 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
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
end

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

```

### 4.1.3 メモリ定義時の初期値の割り当て

メモリ定義時の初期値割り当ては **GowinSynthesis** でのみサポートされており、言語として **System Verilog** を選択する必要があります。  
**read-before-write**、**bypass**、同期リセットモードの **DPB** を例に説明します。  
 そのコーディングスタイルは次のとおりです。

```

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
```

```

end

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

```

#### 4.1.4 readmemb/readmemh 文による初期値割り当て

readmemb/readmemh 文によって初期値を割り当てる場合は、パスの区切り文字として「/」を使用してください。read-before-write、bypass、同期リセットモードの DPB を例に説明します。そのコーディングスタイルは次のとおりです。

```

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 の書き込み形式は次のとおりです。

```
0001
0010
0011
0100
```

0101

0110

0111

1000

注記：

Windows OS でサポートされているパス区切り文字「¥」を使用する場合は、エスケープ文字(例えば、E:¥¥dpb.mi)が必要になります。

## 4.2 SP/SPX9

SP/SPX9 はメモリ領域が 16K ビット/18K ビットであるシングルポート・モードの BSRAM です。シングルポートの読み出し/書き込みは 1 つのクロックにより制御されます。2 つの読み出しモード(bypass モードと pipeline モード)と 3 つの書き込みモード(normal モード、write-through モード、read-before-write モード)がサポートされます。SP/SPX9 として合成するには、メモリ(シフトレジスタの形式を除く)は次の条件の少なくとも 1 つを満たす必要があります。

1. データ幅\*アドレス深さ  $\geq 1024$

2. syn\_ramstyle = "block\_ram"を使用。

ここでは、読み出しアドレスのレジスタ経由や初期値の読み出しについて説明します。

### 4.2.1 読み出しアドレスがレジスタを経由する

これは、読み出しアドレスレジスタが制御信号によって制御されない場合にのみサポートされます。write-through モードの SP が合成されます。出力がレジスタを経由しない場合、そのコーディングスタイルは次のとおりです。

```
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
```

#### 4.2.2 読み出しアドレスがレジスタを経由しなく、出力がレジスタを経由する

読み出しアドレスがレジスタを経由しない場合、出力はレジスタを経由する必要があります。1 レベルのレジスタを経由する場合は **bypass** モード、2 レベルのレジスタを経由する場合は **pipeline** モードです。**write-through**、**bypass**、同期リセットモードの **SPX9** を例に説明します。そのコーディングスタイルは次のとおりです。

```
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
end
```

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

### 4.2.3 メモリ定義時の初期値の割り当て

メモリ定義時の初期値割り当ての場合、言語として **System Verilog** を選択する必要があります。**read-before-write**、**bypass**、同期リセットモードの **SP** を例に説明します。そのコーディングスタイルは次のとおりです。

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



```
end  
endmodule
```

#### 4.2.4 readmemb/readmemh 文による初期値割り当て

readmemb/readmemh 文によって初期値を割り当てる場合は、パスの区切り文字として「/」を使用してください。Normal、pipeline、同期リセットモードの SP を例に説明します。そのコーディングスタイルは次のとおりです。

```
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 の書き込みの形式は次のとおりです。

```

12
34
56
78
9a
bc
de
ff

```

## 4.2.5 シフトレジスタの形式

シフトレジスタの形式で **SP/SPX9** として合成するには、次のいずれかの条件を満たす必要があります。

1. アドレス深さ  $\geq 5$ 、アドレス深さ  $\times$  データ幅  $\geq 256$ 、アドレス深さ  $= 2^n + 1$ 。
2. 属性制約 `syn_srlstyle = "block_ram"` を追加し、アドレス深さ  $= 2^n + 1$  かつアドレス深さ  $\geq 5$ 。

**read-before-write**、**bypass**、同期リセットモードの **SPX9** を例に説明します。そのコーディングスタイルは次のとおりです。

```

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
```

#### 4.2.6 Decoder の形式

read-before-write、bypass、同期リセットモードの SP を例に説明します。そのコーディングスタイルは次のとおりです。

```
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

```

## 4.3 SDPB/SDPX9B

SDPB/SDPX9B のメモリ領域はそれぞれ 16K bit/18K bit で、その動作モードはセミ・デュアルポート・モードです。2 種類の読み出しモード(bypass モードと pipeline モード)と 1 種類の書き込みモード(Normal モード)がサポートされます。SDPB/SDPX9B として合成するには、メモリ(シフトレジスタの形式を除く)は次の条件の少なくとも 1 つを満たす必要があります。

1. データ幅\*アドレス深さ>=1024。
2. syn\_ramstyle = "block\_ram"を使用。

### 4.3.1 memory 初期値なし

bypass、同期リセットモードの SDPB を例に説明します。そのコーディングスタイルは次のとおりです。

```

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

```

### 4.3.2 メモリ定義時の初期値の割り当て

pipeline、非同期リセットモードの SDPB を例に説明します。そのコーディングスタイルは次のとおりです。

```

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
end
endmodule

```

### 4.3.3 readmemb/readmemh 文による初期値割り当て

readmemb/readmemh 文によって初期値を割り当てる場合は、パスの区切り文字として「/」を使用してください。bypass、非同期リセットモードのSDPBを例に説明します。そのコーディングスタイルは次のとおりです。

```

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 の書き込みの形式は次のとおりです。

```

12
34
56
78
9a
bc
de
ff

```

#### 4.3.4 シフトレジスタの形式

次のいずれかの条件を満たす必要があります。

1. アドレス深さ  $\geq 5$ 、アドレス深さ  $\times$  データ幅  $\geq 256$ 、アドレス深さ  $= 2^n + 1$ 。

2. 属性制約 `syn_srlstyle = "block_ram"`を追加し、アドレス深さ  $! = 2^n + 1$  かつアドレス深さ  $> = 5$ 。

`bypass`、同期リセットモードの **SDPX9B** を例に説明します。そのコーディングスタイルは次のとおりです。

```
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 非対称タイプ

`bypass`、同期リセットモードの **SDPB** を例に説明します。そのコーディングスタイルは次のとおりです。

```
module normal(dout, din, ada, clka, cea, adb, clk, 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 の形式

bypass、同期リセットモードの SDPB を例に説明します。そのコーディングスタイルは次のとおりです。

```

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
```

## 4.4 pROM/pROMX9

pROM/pROMX9(16K/18K Block ROM)は 16K/18K のブロック ROM です。  
pROM/pROMX9 は読み出し専用モードで、2 種類の読み出しモード

(bypass モードと pipeline モード)をサポートします。pROM/pROMX9 の割り当て方法には、case 文、readmemb/readmemh、およびメモリ定義時の割り当てが含まれます。pROM を合成する場合は、次の条件の少なくとも 1 つを満たす必要があります。

1. データ幅\*アドレス深さ $\geq$  1024 かつアドレス深さ $>32$
2. syn\_romstyle = "block\_rom"を使用

#### 4.4.1 case 文による初期値割り当て

bypass、同期リセットモードの pROM を例に説明します。そのコーディングスタイルは次のとおりです。

```
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
```

#### 4.4.2 メモリ定義時の初期値の割り当て

メモリ定義時の初期値割り当ての場合、言語として **System Verilog** を選択する必要があります。**bypass**、同期リセットモードの **pROM** を例に説明します。そのコーディングスタイルは次のとおりです。

```
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

```

#### 4.4.3 readmemb/readmemh 文による初期値割り当て

readmemb/readmemh 文によって初期値を割り当てる場合は、パスの区切り文字として「/」を使用してください。pipeline、非同期リセットモードの pROM を例に説明します。そのコーディングスタイルは次のとおりです。

```

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
end

```

```
        else begin
            data_out_reg <= mem[addr];
        end
    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 のデータは次のとおりです :

```
11001100
11001100
11001100
11001100
17001100
11001100
16001100
1f001100
11111100
11111100
11001110
11000111
11000111
11001110
11011100
11001110
```

# 5 SSRAM のコーディングスタイル

SSRAM は分散スタティック RAM で、シングルポート・モード、セミ・デュアルポート・モード、および読み出し専用モードに構成できます。

## 5.1 RAM16S

RAM16S のタイプには、異なる出力ビット幅の RAM16S1、RAM16S2、および RAM16S4 が含まれます。RAM16S の実装には、Decoder、Memory、およびシフトレジスタなどの形式があります。RAM16S として合成するには、メモリ(シフトレジスタの形式を除く)は次のいずれかの条件を満たす必要があります。

1. 読み出しアドレスと出力はレジスタを経由しません。
2. 出力はレジスタを経由し、アドレス深さ\*データ幅< 1024。
3. 属性制約 `syn_ramstyle = "distributed_ram"`を使用します。

### 5.1.1 Decoder の形式

RAM16S4 を例に説明します。その Decoder の形式のコーディングスタイルは次のとおりです。

```
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 の形式

Memory には、初期値の割り当てによって、初期値なし、定義時割り当て、および readmemh/readmemb などのタイプがあります。定義時割り当ておよび readmemh/readmemb による割り当てについては、[4.1.4 readmemb/readmemh 文による初期値割り当て](#)を参照して下さい。

RAM16S4 を例に初期値なし Memory を説明します。そのコーディングスタイルは次のとおりです。

```
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

```

### 5.1.3 シフトレジスタの形式

次のいずれかの条件を満たす必要があります。

1. アドレス深さ $>3$ 、 $8 < \text{アドレス深さ} \times \text{データ幅} \leq 256$ 、アドレス深さ $= 2^n$ 。
2. 属性制約 `syn_srlstyle = "distributed_ram"`を追加し、アドレス深さ $= 2^n$ 、アドレス深さ $>3$ 、アドレス深さ $\times$ データ幅 $>8$ 。

GowinSynthesis による RAM16S4 の合成を例に説明します。そのコーディングスタイルは次のとおりです。

```

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

RAM16SDP のタイプには、異なる出力ビット幅の RAM16SDP1、RAM16SDP2、および RAM16SDP4 が含まれます。RAM16SDP の実装には、Decoder、Memory、およびシフトレジスタなどの形式があります。RAM16SDP として合成するには、メモリ(シフトレジスタの形式を除く)は次のいずれかの条件を満たす必要があります。

1. 読み出しアドレスと出力はレジスタを経由しません。
2. 読み出しアドレスと出力はレジスタを経由し、アドレス深さ\*データ幅<1024。
3. 属性制約 `syn_ramstyle = "distributed_ram"`を使用します。

### 5.2.1 Decoder の形式

RAM16SDP4 を例に説明します。その Decoder の形式のコーディングスタイルは次のとおりです。

```
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
endmodule
```

```
        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 の形式

Memory には、初期値の割り当てによって、初期値なし、定義時割り当て、および readmemh/readmemb などのタイプがあります。定義時割り当ておよび readmemh/readmemb による割り当てについては、4.1.4 を参照して下さい。

RAM16SDP4 を例に説明します。その Memory の形式のコーディングスタイルは次のとおりです。

```
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
```

### 5.2.3 シフトレジスタの形式

シフトレジスタが RAM16SDP に合成される場合、次のいずれかの条件を満たす必要があります。

1. アドレス深さ>3、アドレス深さ\*データ幅<=256、アドレス深さ!= 2<sup>n</sup>。
2. 属性制約 `syn_srlstyle = "distributed_ram"`を追加し、アドレス深さ!= 2<sup>n</sup>、アドレス深さ>3、アドレス深さ\*データ幅>8

GowinSynthesis による RAM16SDP4 の合成を例に説明します。そのコーディングスタイルは次のとおりです。

```
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 はアドレス深さが 16、データ幅が 1 の読み出し専用メモリで、メモリの内容は INIT によって初期化されます。ROM16 には、Decoder、Memory などの形式があります。属性制約 `syn_romstyle = "no_rw_check"` を追加する必要があります。

### 5.3.1 Decoder の形式

```
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
```

### 5.3.2 Memory の形式

Memory には、初期値の割り当てによって、初期値なし、定義時割り当て、および readmemh/readmemb などのタイプがあります。定義時割り当ておよび readmemh/readmemb による割り当てについては、4.1.4 を参照して下さい。

```
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 のコーディングスタイル[1]

注記：

[1] Arora V FPGA の DSP のコーディングスタイルについては、7 Arora V DSP のコーディングスタイルを参照してください。

DSP ブロックには、前置加算器(Pre-Adder)、乗算器(MULT)、および 54 ビット算術論理演算装置(ALU54D)が含まれます。

## 6.1 Pre-adder

Pre-adder は前置加算器で、前置加算、前置減算及びシフト機能を実現します。Pre-adder はビット幅によって 9 ビット幅の PADD9 と 18 ビット幅の PADD18 の 2 種類に分類します。Pre-adder を実装するには、Multiplier と併用して推論する必要があります。

### 6.1.1 前置加算機能

AREG と BREG を経由する同期リセットモードの PADD9-MULT9X9 を例として、前置加算機能の実現を紹介します。そのコーディングスタイルは次のとおりです。

```
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
```



```

        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
end

assign dout = s_reg;

endmodule

```

### 6.1.2 前置減算機能

AREG と BREG を経由する同期リセットモードの PADD9-MULT9X9 を例として、前置減算機能の実現を紹介します。そのコーディングスタイルは次のとおりです。

```

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 シフト機能

AREG と BREG を経由する非同期リセットモードの PADD18-MULT18X18 を例として、シフト機能の実現を紹介します。そのコーディングスタイルは次のとおりです。

```
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** は乗算器です。MDIA と MDIB は乗算器の乗数入力信号で、MOUT は積の出力信号です。DOUT=A\*B という乗算を実現できます。

**Multiplier**、データ幅によって 9x9、18x18、36x36 などの乗算器に構成でき、それぞれプリミティブの MULT9X9、MULT18X18、MULT36X36 に対応します。AREG、BREG、OUT\_REG、PIPE\_REG を経由する非同期リセットモードの MULT18X18 を例に説明します。そのコーディングスタイルは次のとおりです。

```

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)は 54 ビットの算術論理演算を実現する 54 ビットの算術論理演算装置です。ALU54D として合成するには、データ幅は[48,54]の範囲内でなければならず、そうでない場合は、属性制約 `syn_dspstyle = "dsp"`を追加する必要があります。AREG、BREG、OUT\_REG を経由する非同期リセットモードの ALU54D を例に説明します。そのコーディングスタイルは次のとおりです。

```

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'h0000000000000000;
    reg [width-1:0] b_reg=54'h0000000000000000;
    reg [width-1:0] s_reg=54'h0000000000000000;
    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
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 モードでは、乗算器の出力は 54-bit ALU 演算が実行されます。MULTALU36X18 と MULTALU18X18 があります。MULTALU18X18 のみがインスタンス化可能です。MULTALU36X18 には、次の 3 つの演算を実現できます :  $DOUT=A*B \pm C$ 、 $DOUT=\Sigma(A*B)$ 、 $DOUT=A*B+CASI$ 。

### 6.4.1 $A*B \pm C$ 機能

AREG、BREG、CREG、PIPE\_REG、OUT\_REG を経由する非同期リセットモードの MULTALU36X18 を例として、 $DOUT=A*B+C$  という演算を紹介します。そのコーディングスタイルは次のとおりです。

```

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'h000000;
reg signed [s_width-1:0] p0_reg=54'h000000000000000;
reg signed [s_width-1:0] o0_reg=54'h000000000000000;
reg signed [s_width-2:0] c_reg=54'h000000000000000;
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 $\Sigma(A*B)$ 機能

PIPE\_REG、OUT\_REG を経由する非同期リセットモードの MULTALU36X18 を例として、 $DOUT = \Sigma(A*B)$  という演算を紹介します。そのコーディングスタイルは次のとおりです。

```

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'h0000000000000000;
reg signed [c_width-1:0] out_reg=54'h0000000000000000;
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 機能

PIPE\_REG、OUT\_REG を経由する非同期リセットモードの MULTALU36X18 を例として、 $DOUT=A*B+CASI$  という演算を紹介します。そのコーディングスタイルは次のとおりです。

```

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'h0000000000000000;
reg signed [s_width-1:0] p1_reg=54'h0000000000000000;
reg signed [s_width-1:0] p2_reg=54'h0000000000000000;
reg signed [s_width-1:0] s0_reg=54'h0000000000000000;
reg signed [s_width-1:0] s1_reg=54'h0000000000000000;
reg signed [s_width-1:0] o0_reg=54'h0000000000000000;

```

```
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
end
assign s=s1_reg;
endmodule

```

## 6.5 MULTADDALU

MULTADDALU(The Sum of Two Multipliers with ALU)は乗算の加算後の累積または reload 演算を実現する ALU 機能付きの乗算加算器で、対応するプリミティブは MULTADDALU18X18 です。次の 3 つの演算をサポートします :  $DOUT=A0*B0 \pm A1*B1 \pm C$ 、 $DOUT = \Sigma (A0*B0 \pm A1*B1)$ 、 $DOUT=A0*B0 \pm A1*B1 + CASI$ 。

### 6.5.1 $A0*B0 \pm A1*B1 \pm C$ 機能

A0REG、A1REG、B0REG、B1REG、PIPE0\_REG、PIPE1\_REG、OUT\_REG を経由する非同期リセットモードの MULTADDALU18X18 を例として、 $DOUT = A0*B0 \pm A1*B1 \pm C$  という演算を紹介します。そのコーディングスタイルは次のとおりです。

```

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'h000000000000000;
reg signed [s_width-1:0] p1_reg=54'h000000000000000;
reg signed [s_width-1:0] s_reg=54'h000000000000000;

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 $\Sigma(A0*B0 \pm A1*B1)$ 機能

PIPE0\_REG、PIPE1\_REG、OUT\_REG を経由する非同期リセットモードの MULTADDALU18X18 を例として、 $DOUT = \Sigma(A0*B0 \pm A1*B1)$  という演算を紹介します。そのコーディングスタイルは次のとおりです。

```

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'h0000000000000000;
reg unsigned [s_width-1:0] p1_reg=54'h0000000000000000;
reg unsigned [s_width-1:0] s=54'h0000000000000000;
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
```

```

    end
end
endmodule

```

### 6.5.3 $A0*B0 \pm A1*B1 + CASI$ 機能

PIPE0\_REG、PIPE1\_REG、OUT\_REG を経由する非同期リセットモードの MULTADDALU18X18 を例として、 $DOUT = A0*B0 \pm A1*B1 + CASI$  という演算を紹介します。そのコーディングスタイルは次のとおりです。

```

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'h0000000000000000;
reg signed [s_width-1:0] p1_reg=54'h0000000000000000;
reg signed [s_width-1:0] p2_reg=54'h0000000000000000;
reg signed [s_width-1:0] p3_reg=54'h0000000000000000;
reg signed [s_width-1:0] s0_reg=54'h0000000000000000;
reg signed [s_width-1:0] s1_reg=54'h0000000000000000;
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
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
end
assign s=s1_reg;
endmodule
```

# 7 Arora V DSP のコーディングスタイル

Arora V FPGA 製品の DSP リソースは、乗算、前置加算、累積、シフトなどの機能を実現できます。各 DSP は、2 つの独立したクロック信号、2 つの独立したクロックイネーブル信号、2 つの独立したリセット信号を持っています。

## 7.1 前置加算機能

Arora V FPGA 製品の DSP リソースは、26 以下のビット幅の符号付き前置加算を実装でき、静的加算/減算、動的加算/減算、同期/非同期リセットモードをサポートしています。ビット幅によっては、MULTALU27X18 や MULT27X36 に推論することができます。前置加算は、推論するために乗算と組み合わせる必要があります。

### 7.1.1 静的加算/減算機能

MULTALU27X18 を例として、静的前置加算/減算機能の実現を紹介します。そのコーディングスタイルは次のとおりです。

```
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 動的加算/減算機能

MULTALU27X18 を例として、動的前置加算/減算機能の実現を紹介します。そのコーディングスタイルは次のとおりです。

```
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 乗算機能

Arora V FPGA 製品の DSP リソースは、27X36 以下のビット幅の符号付き乗算を実装でき、ビット幅によっては、MULTALU27X18、MULT12X12、MULT27X36 に推論することができます。AREG、BREG、PIPE\_REG、

OUT\_REG を経由する非同期リセットモードの MULTALU27X18 を例に説明します。そのコーディングスタイルは次のとおりです。

```
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
end

assign out=out_reg;
endmodule

```

## 7.3 乗算加算機能

Arora V FPGA 製品の DSP リソースは、27X18 以下のビット幅の符号付き乗算加算を実装でき、静的加算/減算、動的加算/減算、同期/非同期リセットモードをサポートしています。乗算加算機能は、ビット幅によって MULTALU27X18 や MULTADDALU12X12 に推論することができます。

### 7.3.1 静的加算/減算機能

静的乗算加算機能の MULTADDALU12X12 を例として、静的加算/減算機能の実現を紹介します。そのコーディングスタイルは次のとおりです。

```

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 動的加算/減算機能

AREG、BREG、CREG、PREG、OREG を経由する同期リセットモードの動的乗算加算機能の MULTALU27X18 を例として、動的加算/減算機能の実現を紹介します。そのコーディングスタイルは次のとおりです。

```
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
end

assign out=out_reg;
endmodule

```

## 7.4 累積機能

Arora V FPGA 製品の DSP リソースは、27X18 以下のビット幅の符号付き累積を実装でき、静的累積、動的累積、同期/非同期リセットモードをサポートしています。累積機能は、ビット幅によって MULTALU27X18 や MULTADDALU12X12 に推論することができます。

### 7.4.1 静的累積機能

OREG を経由する同期リセットモードの MULTALU27X18 を例として、静的累積機能の実現を紹介します。そのコーディングスタイルは次のとおりです。

```

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 動的累積機能

AREG、BREG、OREG を経由する同期リセットモードの MULTALU27X18 を例として、動的累積機能の実現を紹介します。そのコーディングスタイルは次のとおりです。

```

module top(a,b,clk,ce,rst,accsel,out);
parameter PRE_LOAD = 48'h0000000000012;

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 カスケード接続機能

Arora V FPGA 製品の DSP リソースは、27X18 以下のビット幅の符号付きカスケード接続を実装できます。カスケード接続をサポートするプリミティブには、MULTALU27X18 と MULTADDALU12X12 があります。

### 7.5.1 静的カスケード接続機能

2 つの MULTADDALU12X12 のカスケード接続を例として、静的カスケード接続機能の実現を紹介します。そのコーディングスタイルは次のとおりです。

```
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 動的カスケード接続機能

2 つの MULTALU27X18 のカスケード接続を例として、動的カスケード接続機能の実現を紹介します。そのコーディングスタイルは次のとおりです。

```
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 シフト機能

Arora V FPGA 製品の DSP リソースは、27X18 以下のビット幅の符号付きシフトを実装できます。シフトをサポートするプリミティブには、MULTALU27X18 があります。

入力 **a** が 2 レジスタを経由することを例として、シフト機能の実現を紹介します。そのコーディングスタイルは次のとおりです。

```

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
```

