




# GowinSynthesis® ユーザーガイド

SUG550-1.5J, 2021-10-28

## 著作権について(2021)

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**、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かに拘わらず)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。本文書における全ての情報は、予備的情報として取り扱われなければなりません。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
2019/08/02	1.0J	初版。
2019/12/05	1.1J	合成前後のオブジェクト命名規則の説明を追加 (Gowin ソフトウェア 1.9.3 以降に適用)。
2020/03/03	1.2J	VHDL 言語による設計をサポート(Gowin ソフトウェア 1.9.5 以降に適用)。
2020/05/29	1.3J	<ul style="list-style-type: none"><li>● 合成の命名規則を変更。</li><li>● <code>syn_srstyle</code> と <code>syn_noprune</code> の属性を追加。</li></ul>
2020/09/14	1.4J	<code>black_box_pad_pin</code> の属性を追加。
2021/10/28	1.5J	<ul style="list-style-type: none"><li>● <code>parallel_case</code> 、 <code>syn_black_box</code> の属性を追加。</li><li>● 6 Report ファイルを変更。</li></ul>

# 目次

目次 .....	i
図一覧 .....	iii
表一覧 .....	iv
<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 概要 .....</b>	<b>3</b>
<b>3 GowinSynthesis の使用方法 .....</b>	<b>4</b>
3.1 GowinSynthesis の入出力 .....	4
3.2 GowinSynthesis による合成 .....	4
3.3 合成前後のオブジェクト命名規則 .....	4
3.3.1 合成後のネットリストファイルの命名 .....	4
3.3.2 合成後のネットリストのモジュールの命名 .....	5
3.3.3 合成後のネットリストのインスタンスの命名 .....	5
3.3.4 合成後のネットリストの回線の命名 .....	5
<b>4 HDL コードのサポート .....</b>	<b>6</b>
4.1 レジスタの HDL コードのサポート .....	6
4.1.1 レジスタの特徴 .....	6
4.1.2 レジスタの制約 .....	6
4.1.3 レジスタコードの例 .....	6
4.2 RAM の HDL コードのサポート .....	13
4.2.1 RAM 推論の概要 .....	13
4.2.2 RAM の特徴 .....	13
4.2.3 RAM 推論の制約 .....	13
4.2.4 RAM 推論のコードの例 .....	14
4.3 DSP の HDL コードのサポート .....	21
4.3.1 DSP 推論の概要 .....	21

4.3.2 DSP の特徴 .....	21
4.3.3 DSP の制約 .....	22
4.3.4 DSP 推論のコードの例 .....	22
4.4 有限状態機械の合成ルール .....	29
4.4.1 有限状態機械の合成ルール .....	29
4.4.2 有限状態機械のコード例 .....	29
<b>5 合成制約のサポート .....</b>	<b>32</b>
5.1 black_box_pad_pin .....	33
5.2 full_case .....	35
5.3 parallel_case .....	36
5.4 syn_black_box.....	37
5.5 syn_dspstyle.....	39
5.6 syn_encoding .....	40
5.7 syn_insert_pad .....	42
5.8 syn_keep .....	42
5.9 syn_looplimit.....	43
5.10 syn_maxfan .....	44
5.11 syn_netlist_hierarchy.....	45
5.12 syn_noprune.....	46
5.13 syn_preserve.....	48
5.14 syn_probe.....	49
5.15 syn_ramstyle .....	50
5.16 syn_romstyle .....	52
5.17 syn_srlstyle.....	53
5.18 syn_tlvds_io/syn_elvds_io.....	55
5.19 translate_off/translate_on.....	56
<b>6 Report ファイル .....</b>	<b>58</b>
6.1 Synthesis Message .....	58
6.2 Synthesis Details.....	58
6.3 Resource .....	59
6.4 Timing.....	60

# 図一覧

図 4-1 例 1 の同期リセットのフリップフロップの説明図.....	8
図 4-2 例 2 のクロックイネーブル付きの同期セットフリップフロップの説明図 .....	8
図 4-3 例 3 のクロックイネーブル付きの非同期リセットフリップフロップの説明図 .....	9
図 4-4 例 4 のアクティブ High のリセット機能付きラッチの説明図 .....	10
図 4-5 例 5 の同期リセットのフリップフロップおよび論理回路の説明図 .....	11
図 4-6 例 6 の初期値が 0 の通常のフリップフロップおよび論理回路の説明図 .....	11
図 4-7 例 7 の非同期セットのフリップフロップの説明図.....	13
図 4-8 例 1 の RAM の回路図 .....	15
図 4-9 例 2 の RAM の回路図 .....	16
図 4-10 例 3 の RAM の回路図 .....	17
図 4-11 例 4 の RAM の回路図 .....	18
図 4-12 例 5 の RAM の回路図 .....	19
図 4-13 例 6 の pROM の回路図.....	20
図 4-14 例 7 の RAM の回路図 .....	21
図 4-15 例 1 の DSP の回路図.....	24
図 4-16 例 2 の DSP の回路図.....	26
図 4-17 例 3 の DSP の回路図.....	27
図 4-18 例 4 の DSP の回路図.....	28
図 4-19 例 5 の DSP の回路図.....	29
図 6-1 Synthesis Message .....	58
図 6-2 Synthesis Details.....	59
図 6-3 Resource .....	59
図 6-4 Timing.....	60
図 6-5 Performance Summary .....	60
図 6-6 Path Summary.....	61
図 6-7 接続関係、遅延、およびファンアウト情報.....	61
図 6-8 Path Statistics.....	61

# 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

このマニュアルはユーザーが合成ツールである **GowinSynthesis** を使いこなせるよう、その機能および使用法について説明しています。本マニュアルに記載のソフトウェア GUI のスクリーンショットは、**Gowin** ソフトウェア 1.9.8.01 バージョンの場合のものです。ソフトウェアのバージョンアップデートにより、一部の内容が変更される場合があります。

## 1.2 関連ドキュメント

GOWIN セミコンダクターの公式 Web サイト [www.gowinsemi.com/ja](http://www.gowinsemi.com/ja) から、以下の関連ドキュメントがダウンロード、参考できます：

- Gowin ソフトウェア ユーザーガイド([SUG100](#))

## 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
FPGA	Field Programmable Gate Array	フィールド・プログラマブル・ゲート・アレイ
SSRAM	Shadow Static Random Access Memory	分散SRAM
BSRAM	Block Static Random Access Memory	ブロックSRAM
DSP	Digital Signal Processing	デジタル信号処理
FSM	Finite State Machine	有限状態機械
GSC	Gowin Synthesis Constraint	GowinSynthesis制約ファイル



## 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。

Web サイト : [www.gowinsemi.com/ja](http://www.gowinsemi.com/ja)

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

# 2 概要

このマニュアルは、合成ツールである **GowinSynthesis** のユーザーガイドです。

**GowinSynthesis** は、**Gowin** セミコンダクターが独自に開発した合成ツールです。**Gowin** 独自の **EDA** アルゴリズムを使用し、製品のハードウェア特性とハードウェア回路リソースに基づいて、**RTL** 設計抽出、算術最適化、推論・置き換え、リソース共有、並列合成、およびマッピングなどのテクノロジーを実現します。これにより、ユーザーの **RTL** 設計の最適化、リソース検査、およびタイミング解析を迅速に実行できます。

**GowinSynthesis** は、**FPGA** 設計者に **Gowin FPGA** チップの最も効果的な設計実装方法を提供します。設計の実装に関しては、タイミング解析、リソース検査、プリミティブ論理解析などの機能を提供するとともに、詳細な合成情報を提供します。**GowinSynthesis** は、**Gowin** のプリミティブライブラリに基づいた、**Gowin** 配置配線ツールの入力ファイルとしての合成後ネットリストを生成します。これにより、面積と速度の最適なバランスを実現し、ソフトウェアのコンパイル効率と配線性(**Routability**)を向上させることができます。このソフトウェアには次の特徴があります。

- **Verilog / SystemVerilog**、**VHDL** による設計、および混合設計の入力をサポート
- 超大規模設計をサポートし、複雑なプログラマブルロジック設計のための優れた合成ソリューションを提供
- ルックアップテーブル、レジスタ、ラッチ、および算術論理演算装置の推論・マッピングをサポート
- メモリの推論・マッピング、および論理リソース使用のバランスをサポート
- **DSP** の推論・マッピング、および論理リソース使用のバランスをサポート
- **FSM** の合成最適化をサポート
- さまざまなアプリケーション条件下での合成結果の要件を満たすために、合成属性と合成命令をサポート

# 3 GowinSynthesis の使用方法

## 3.1 GowinSynthesis の入出力

GowinSynthesis は、Gowin ソフトウェアによって自動的に生成される、プロジェクトファイル(.gprj)の形式でユーザーの RTL ファイルを読み込みます。ユーザーRTL ファイルに加えて、GowinSynthesis プロジェクトファイルは、合成デバイス、ユーザー制約ファイル(合成属性制約ファイル)、合成後ネットリストファイル(.vg)、およびいくつかの合成オプション(合成の top module、ファイルの include path など)も指定しています。

## 3.2 GowinSynthesis による合成

Process ビューで Synthesize を右クリックし、Configuration を選択します。表示されるページでは、top module や include path の設定と、言語バージョンの選択など、関連する合成オプションを構成できます。

Gowin ソフトウェアの Process ビューで Synthesize をダブルクリックして合成を実行すると、Output ビューでは合成情報が表示されます。GowinSynthesis は、合成後に合成レポートとゲートレベルのネットリストファイルを生成します。Process ビューで Synthesis Report と Netlist File をダブルクリックして、特定のコンテンツを表示します。

詳しくは、Gowin ソフトウェア ユーザーガイド([SUG100](#)) > 4.4.3 Synthesize を参照してください。

## 3.3 合成前後のオブジェクト命名規則

ユーザーの検証とデバッグのために、GowinSynthesis は合成中に、ユーザーデザインの module の情報、primitive/module インスタンス名、ユーザー定義の wire/reg 回線名など、ユーザーのオリジナルの RTL デザイン情報を最大限に保持します。最適化または変換しなければならない場合も、次のように、名前はユーザー定義の回線名に基づいていくつかの派生ルールに従って生成されます。

### 3.3.1 合成後のネットリストファイルの命名

合成後のネットリストファイルの名前は、プロジェクトファイルで指定

されている出力ネットリストのファイル名に依存します。

合成後のネットリスト名がプロジェクトファイルで指定されていない場合、デフォルトではプロジェクトファイル名と同じで、拡張子が.vg の合成後のネットリストファイルが生成されます。

### 3.3.2 合成後のネットリストのモジュールの命名

合成後のネットリストの **module** 名は、RTL デザインの名前と一致します。複数回インスタンス化されたモジュールは、サフィックスの **\_idx** によって区別されます。モジュールのインスタンス名は、RTL デザインと一致します。

### 3.3.3 合成後のネットリストのインスタンスの命名

ユーザーの RTL デザインでインスタンス化されたインスタンスが合成中に最適化されていない場合、合成後のネットリストではインスタンス名は変更されません。

合成プロセス中に生成されるインスタンスの名前は、当該インスタンスがユーザーの RTL デザインで表す機能デザインブロックの外部出力信号名から派生します。機能デザインブロックに複数の出力信号がある場合、インスタンス名は最初の出力信号の名前から派生することになります。

合成中に生成されたインスタンスの名前には、上記の信号名に加えて、タイプに従ってサフィックスが付けられます。**buf** のサフィックスは **\_ibuf**、**\_obuf**、**\_jobuf** で、それ以外の場合、サフィックスは **\_s** です。**s** の後の数は、名前がノードによって引用された回数を指します。

**flatten** 出力が指定され、元のサブモジュールのインスタンス名に階層表示が必要な場合、スラッシュ"/" が階層セパレータとして使用されます。

### 3.3.4 合成後のネットリストの回線の命名

RTL デザインでユーザーが定義した **wire/reg** 信号の場合、信号が合成中に最適化されない場合、ネットリストの関連モジュールはこの信号名を保持します。

GowinSynthesis の合成中、一部の RTL デザインの機能設計モジュール全体が置換または最適化されます。合成後、ネットリスト内のこれらの機能設計モジュールの出力信号の信号名が保持されます。これらのモジュールの内部信号の場合、名前は出力信号の名前から派生します。デジタルサフィックス(**\_idx**)は、オリジナルの信号名の後に追加されます。

マルチビット幅の信号(**bus** 形式)名が派生信号名として使用され、他の信号名またはインスタンス名が派生される場合、信号名のバスビット情報は **"\_idx"** の形式で保持されます。

2. **flatten** 出力が指定された場合、アンダースコア “\_” が階層セパレータとして使用されます。

# 4 HDL コードのサポート

## 4.1 レジスタの HDL コードのサポート

### 4.1.1 レジスタの特徴

レジスタには、フリップフロップとラッチが含まれています。

#### フリップフロップ

フリップフロップはすべて D フリップフロップであり、定義時に初期値が割り当てられます。リセット/セットには、同期リセット/セットと非同期リセット/セットの 2 種類があります。同期リセット/セットの場合、クロック信号 CLK の立ち上がりエッジまたは立ち下がりエッジが来て **reset/set** が High になったときにのみリセット/セットを完了できます。非同期リセット/セットの場合、**reset/set** が Low から High になる限り、クロック信号 CLK に制御されずにリセットとセットを完了できます。

#### ラッチ

ラッチには、高レベルトリガーと低レベルトリガーの 2 つのトリガーモードがあります。ラッチは定義時に初期値が割り当てられます。FPGA デザインではラッチを使用しないことをお勧めします。高レベルトリガーとは、制御信号がハイのとき、ラッチがデータ信号の通過を許可することです。低レベルトリガーとは、制御信号がローのとき、ラッチがデータ信号の通過を許可することです。

### 4.1.2 レジスタの制約

ユーザーは、**preserve** 属性を使用してレジスタを制約できます。この制約があると、出力がフローティングのレジスタが最適化され、他のレジスタはそのまま合成結果に残ります。詳細については、**syn\_preserve** を参照してください。

### 4.1.3 レジスタコードの例

Gowin のチップデザインでは、同期リセットフリップフロップの初期値は 0 にのみ設定でき、同期セットフリップフロップの初期値は 1 にのみ設定できるため、ユーザーが RTL で設定した同期フリップフロップの初期値

と、同期フリップフロップの初期値が異なる場合、**GowinSynthesis** は、優先的に **RTL** の初期値に従って同期フリップフロップのタイプを変換します。非同期フリップフロップの場合、以上のように処理する必要はありません。具体的な変換手順は次のとおりです。

3. **RTL** 設計が同期リセットフリップフロップで、指定された初期値が **1** の場合、**GowinSynthesis** はそれを同期セットフリップフロップに置き換え、関連するロジックを元の同期リセット信号に追加して同期セットを実現します。
4. **RTL** 設計が同期セットフリップフロップで、指定された初期値が **0** の場合、**GowinSynthesis** はそれを通常のフリップフロップに置き換え、関連するロジックを元の同期セット信号にデータ入力として追加します。

#### フリップフロップの初期値を指定しない場合

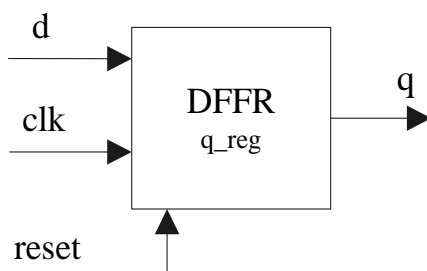
**CLK** の立ち上がりエッジでトリガーされるフリップフロップと **CLK** の立ち下がりエッジでトリガーされるフリップフロップは、**CLK** トリガー方法においてのみ異なるため、**CLK** の立ち上がりエッジトリガーするフリップフロップを合成できる例のみを以下に示します。

例 1 は、同期リセットのフリップフロップとして合成できます。

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

同期リセットのフリップフロップを図 4-1 に示します。

図 4-1 例 1 の同期リセットのフリップフロップの説明図



例 2 は、クロックイネーブル付きの同期セットフリップフロップとして合成できます。

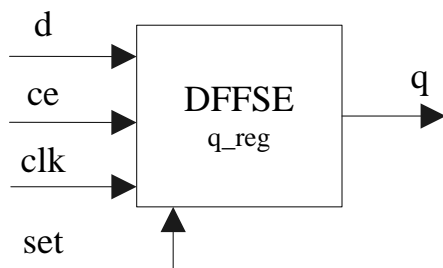
```

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

```

クロックイネーブル付きの同期セットフリップフロップを図 4-2 に示します。

図 4-2 例 2 のクロックイネーブル付きの同期セットフリップフロップの説明図



例 3 は、クロックイネーブル付きの非同期リセットフリップフロップとして合成できます。

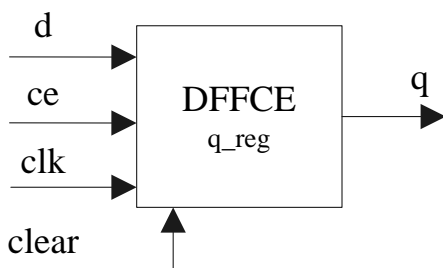
```

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

```

クロックイネーブル付きの非同期リセットフリップフロップを図 4-3 に示します。

図 4-3 例 3 のクロックイネーブル付きの非同期リセットフリップフロップの説明図



例 4 は、アクティブ High のリセット機能付きラッチとして合成できます。

```

module top(d,g,clear,q,ce);
  input d,g,clear,ce;
  output q;
  reg q_reg;

```



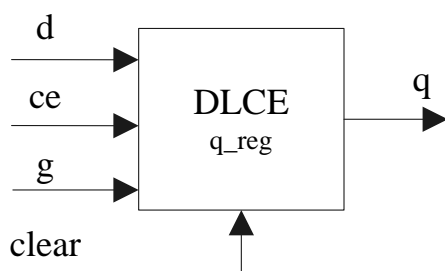
```

always @(g or d or clear or ce) begin
    if(clear)
        q_reg <= 0;
    else if(g && ce)
        q_reg <= d;
    end
assign q = q_reg;
endmodule

```

アクティブ High のリセット機能付きラッチを図 4-4 に示します。

図 4-4 例 4 のアクティブ High のリセット機能付きラッチの説明図



#### フリップフロップの初期値を指定する場合

例 5 は同期リセットのフリップフロップで、その初期値は 0 である必要がありますが、RTL での初期値が 1 です。従って、例 5 は初期値が 1 の同期リセットのフリップフロップおよび同期リセット付きの論理回路として合成されます。

```

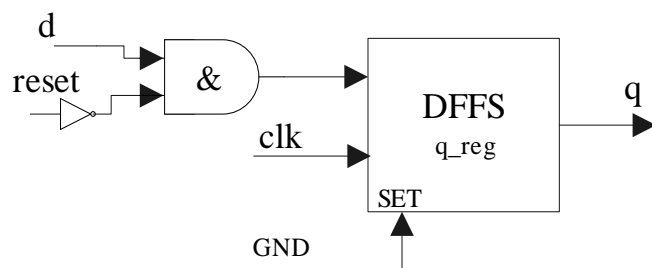
module top (q, d, clk, reset);
    input d;
    input clk;
    input reset;
    output q;
    reg q_reg = 1'b1;
    always @(posedge clk)begin
        if(reset)
            q_reg<=1'b0;
        else
            q_reg<=d;
        end
    assign q = q_reg;
end

```

*endmodule*

上記の同期リセットのフリップフロップを図 4-5 に示します。

図 4-5 例 5 の同期リセットのフリップフロップおよび論理回路の説明図



例 6 は同期セットのフリップフロップで、その初期値は 1 である必要がありますが、RTL での初期値が 0 です。従って、例 6 は初期値が 0 の通常のフリップフロップおよび同期セット付きの論理回路として合成されます。

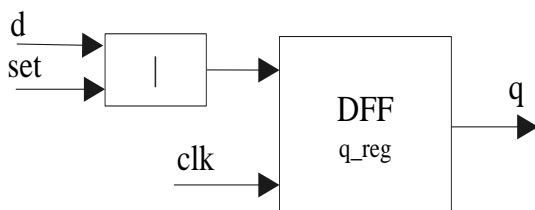
```

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

```

初期値が 0 の通常のフリップフロップおよび論理回路を図 4-6 に示します。

図 4-6 例 6 の初期値が 0 の通常のフリップフロップおよび論理回路の説明図

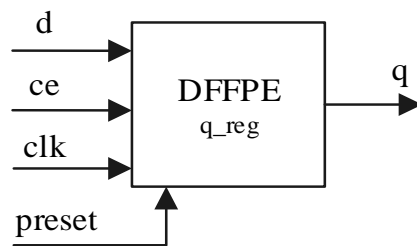


例 7 は、初期値が 1 の非同期セットのフリップフロップです。

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

上記の非同期セットのフリップフロップを図 4-7 に示します。

図 4-7 例 7 の非同期セットのフリップフロップの説明図



## 4.2 RAM の HDL コードのサポート

### 4.2.1 RAM 推論の概要

RAM の推論は、RTL 合成プロセスでユーザーデザインの一部のメモリ機能部分を、ブロックスタティックランダムアクセスメモリ (BSRAM) または分散ランダムアクセスメモリ (SSRAM) に推論し置き換えるプロセスです。RTL 設計の際、ユーザーは BSRAM または SSRAM プリミティブを直接インスタンス化するか、デバイスに依存しない RTL 形式のメモリ説明を記入できます。RTL 形式のメモリブロックの場合、GowinSynthesis は、RTL 記述に従って、対応する条件を満たす RTL 記述を、対応する RAM モジュールに置き換えます。

BSRAM を使用してロジックモジュールを実装する必要がある場合、次の条件を満たさなければなりません。

5. すべての出力レジスタには同じ制御信号があります。
6. RAM は同期メモリである必要があり、非同期制御信号を接続してはなりません。GowinSynthesis は非同期 RAM をサポートしていません。
7. 読み出しアドレスまたは出力ポートでレジスタを接続します。

### 4.2.2 RAM の特徴

#### BSRAM

BSRAM は、シングルポートモード、デュアルポートモード、セミ・デュアルポートモード、および読み出し専用モードをサポートします。読み出しは、レジスタ出力モード (pipeline) とバイパスモード (bypass) をサポートします。書き込みは、ノーマルライトモード (Normal Mode)、ライトスルーモード (write-through Mode)、およびリードビフォーライトモード (Read-before-write Mode) の 3 つのモードをサポートします。

#### SSRAM

SSRAM は、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をサポートします。

### 4.2.3 RAM 推論の制約

syn\_ramstyle はメモリの実装を指定し、syn\_romstyle は読み出し専用メモリの実装を指定します。

デザインで **SSRAM** または **BSRAM** を生成したい場合は、制約ステートメント **ram\_style**、**rom\_style**、または **syn\_srstyle** を使用して制御してください。

詳しくは、**syn\_romstyle** **syn\_srstyle** を参照してください。

#### 4.2.4 RAM 推論のコードの例

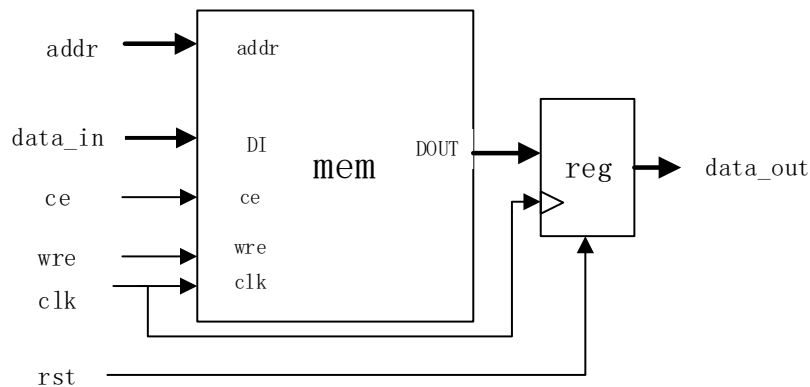
RAM の特性に応じてそれぞれ例示します。

例 1 は、アクセスアドレスが同じである 1 つの書き込みポートと 1 つの読み出しポートを持つメモリで、ノーマルモードのシングルポート **BSRAM** に合成できます。

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

上記のシングルポートの **BSRAM** 回路の説明図を図 4-8 に示します。

図 4-8 例 1 の RAM の回路図



例 2 は、アクセスアドレスが同じである 1 つの書き込みポートと 1 つの読み出しポートを持つメモリです。**wre** が 1 の場合、入力データを出力に直接転送できます。この例はライトスルーモードのシングルポート BSRAM として合成されています。

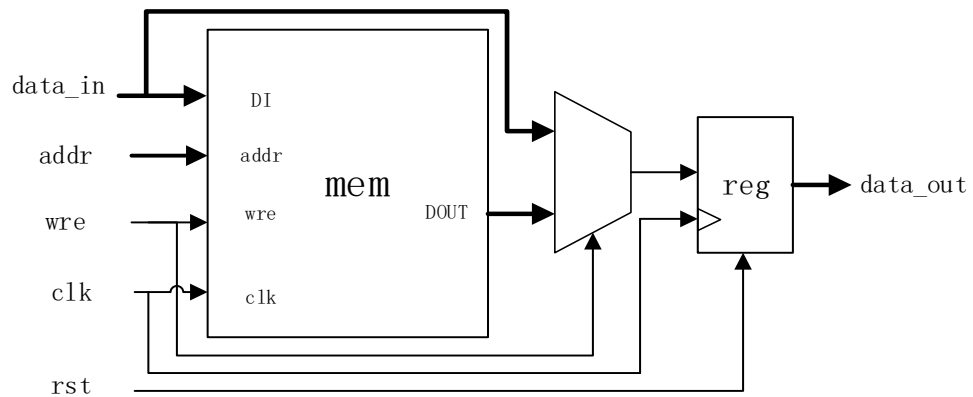
```

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

```

上記のシングルポートの BSRAM 回路の説明図を図 4-9 に示します。

図 4-9 例 2 の RAM の回路図



例 3 は、アクセスアドレスが同じである 1 つの書き込みポートと 1 つの読み出しポートを持つメモリです。**wre** が 1 の場合、入力データをメモリに書き込みできます。この例はリードビフォーライトモードのシングルポート BSRAM として合成されています。

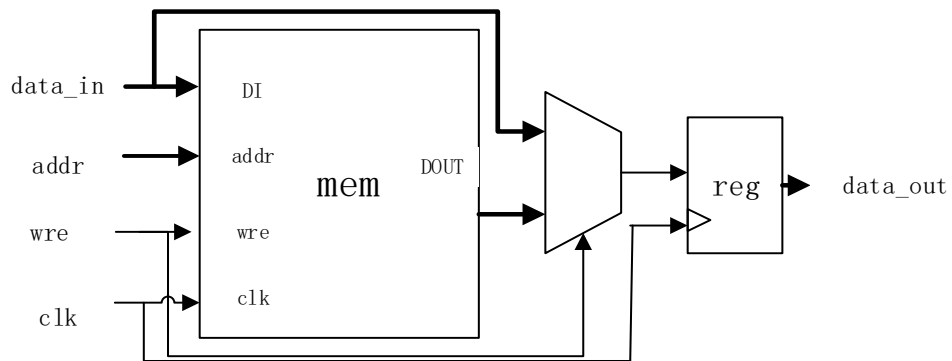
```

module read_first_01(data_out, data_in, addr, clk, wre);
output [31:0]data_out;
input [31:0]data_in;
input [6:0]addr;
input clk,wre;
reg [31:0] mem [127:0];
reg [31:0] data_out;
always @(posedge clk)
begin
    if (wre)
        mem[addr] <= data_in;
        data_out <= mem[addr];
    end
endmodule

```

上記のシングルポートの BSRAM 回路の説明図を図 4-10 に示します。

図 4-10 例 3 の RAM の回路図



例 4 は、2 つの書き込みポート、1 つの読み出しポートを備えたメモリです。1 つの書き込みポートには **wre** 信号があり、1 つの読み出しポートは非同期セットのレジスタを吸収します。この例は、書き込みモードの **A** 側がノーマルモード、**B** 側がリードビフォーライトモード、読み出しモードがレジスタ出力モードである非同期リセットのデュアルポート **BSRAM** です。

```

module read_first_02_1(data_outa, data_ina, addra, clka, rsta, cea, wrea, ocea,
data_inb, addrb, clk, ceb);
    output [17:0] data_outa;
    input [17:0] data_ina, data_inb;
    input [6:0] addra, addrb;
    input clka, rsta, cea, wrea, ocea;
    input clk, ceb;
    reg [17:0] mem [127:0];
    reg [17:0] data_outa;
    reg [17:0] data_out_rega, data_out_regb;
    always @(posedge clk)
    if (ceb)
        mem[addrb] <= data_inb;
    always @(posedge clka or posedge rsta)
    if (rsta)
        data_out_rega <= 0;
    else begin
        data_out_rega <= mem[addra];
    end
    always @(posedge clka or posedge rsta)

```



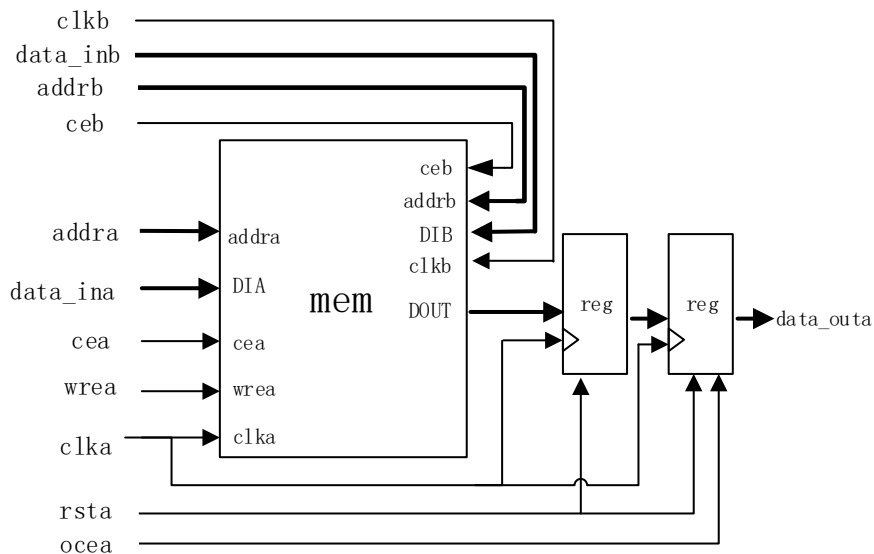
```

if(rsta)
    data_outa <= 0;
else if(ocea)
    data_outa <= data_out_rega;
always @(posedge clka)
if(cea & wrea)
    mem[addra] <= data_ina;
endmodule

```

上記のデュアルポートの BSRAM 回路の説明図を図 4-11 に示します。

図 4-11 例 4 の RAM の回路図



例 5 は、アクセスアドレスが異なる 1 つの読み出しポートと 1 つの書き込みポートを備えたメモリで、書き込みモードがノーマルモードで読み出しモードがバイパスモードのセミ・デュアルポート BSRAM に合成されています。

```

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

```

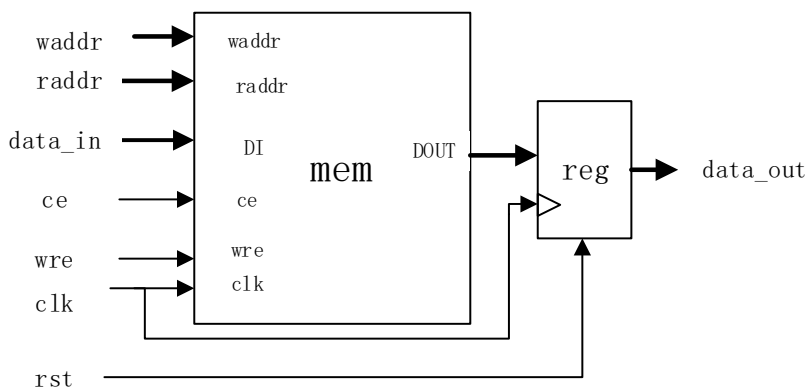
```

        data_out <= 0;
    else if(ce)
        data_out <= mem[raddr];
    always @(posedge clk)
    if (ce) mem[waddr] <= data_in;
endmodule

```

上記のセミ・デュアルポートの BSRAM 回路の説明図を図 4-12 に示します。

図 4-12 例 5 の RAM の回路図



例 6 は、1つの読み出しモードを備えた初期値ありのメモリで、読み出しモードがバイパスモードの非同期セット読み出し専用メモリとして合成されています。

```

module test_invce (clock,ce,oce,reset,addr,dataout) ;
input clock,ce,oce,reset;
input [5:0] addr;
output [7:0] dataout;
reg [7:0] dataout;
always @(posedge clock or posedge reset)
if(reset) begin
    dataout <= 0;
end else begin
    if (ce & oce) begin
        case (addr)
        6'b000000: dataout <= 32'h87654321;
        6'b000001: dataout <= 32'h18765432;
        6'b000010: dataout <= 32'h21876543;

```

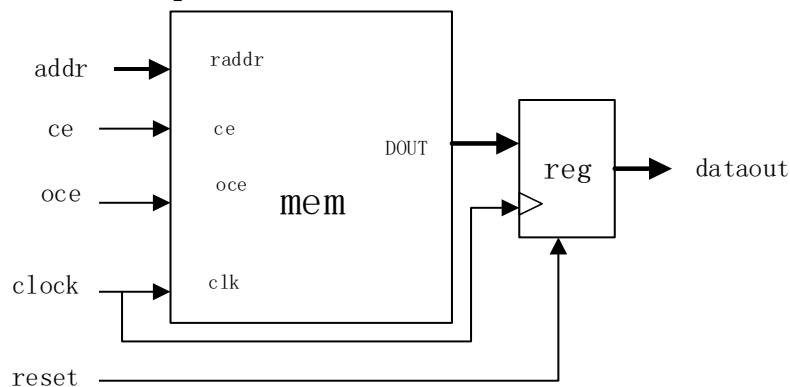
```

.....;
6'b111110: dataout <= 32'hdef89aba;
6'b111111: dataout <= 32'hcf89abce;
default: dataout <= 32'hf89abcde;
endcase
end
end
endmodule

```

上記の読み出し専用メモリ回路の説明図を図 4-13 に示します。

図 4-13 例 6 の pROM の回路図



例 7 は、shift register モードのメモリで、ノーマルモードのセミ・デュアルポート BSRAM に合成されています。

```

module seqshift_bsram (clk, din, dout) ;
parameter SRL_WIDTH = 65;
parameter SRL_DEPTH = 16;
input clk;
input [SRL_WIDTH-1:0] din;
output [SRL_WIDTH-1:0] dout;
reg [SRL_WIDTH-1:0] regBank[SRL_DEPTH-1:0] ;
integer i;
always @(posedge clk) begin
    for (i=SRL_DEPTH-1; i>0; i=i-1) begin
        regBank[i] <= regBank[i-1];
    end
    regBank[0] <= din;
end
end

```

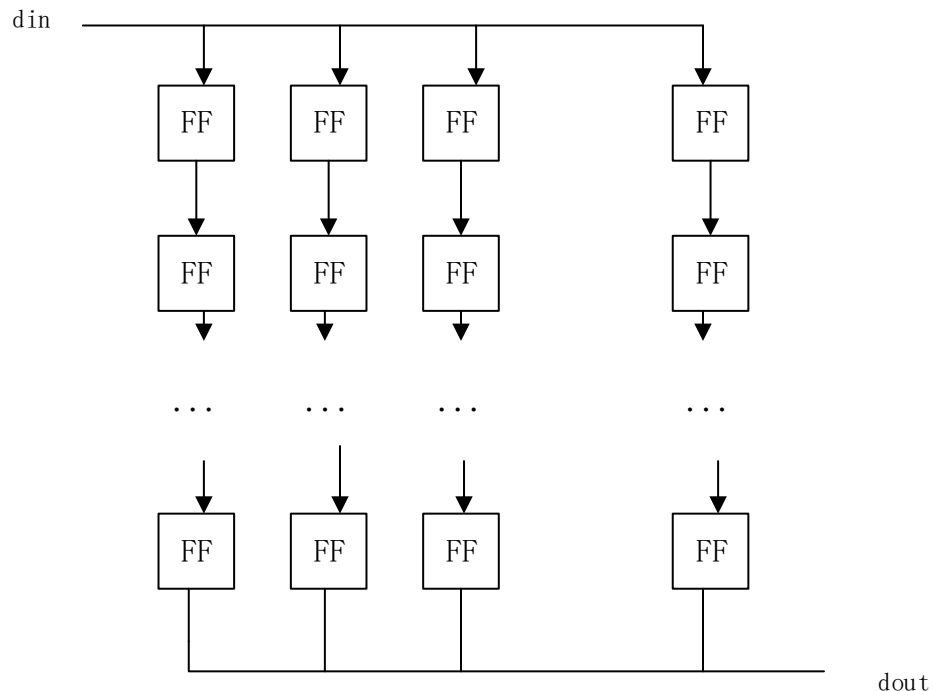
```

assign dout = regBank[SRL_DEPTH-1];
endmodule

```

上記のセミ・デュアルポートの BSRAM 回路の説明図を図 4-14 に示します。

図 4-14 例 7 の RAM の回路図



注記：

その他の例については、GOWIN の公式サイトの GowinSynthesis 推論コーディングテンプレート「[GowinSynthesis Inference Coding Template](#)」を参照してください。

## 4.3 DSP の HDL コードのサポート

### 4.3.1 DSP 推論の概要

DSP 推論は、合成プロセスでユーザーデザインの乗算および一部の加算を DSP に推論・置き換えるアルゴリズムです。RTL 設計の際、ユーザーは DSP をインスタンス化するか、RTL 形式の DSP 記述を記入できます。GowinSynthesis は、RTL 記述に従って、対応する条件を満たす RTL 記述を、対応する DSP ブロックに推論・置き換えます。

DSP ブロックは、乗算と加算、およびレジスタ機能を有しています。ユーザーの現在のデバイスが DSP ブロックをサポートしていない場合、GowinSynthesis は論理回路を使用して乗算器を実装します。

### 4.3.2 DSP の特徴

Gowin DSP には、乗算器、乗算加算器、前置加算器、およびアキュムレータがあります。その機能は次のとおりです。

1. 入力符号ビットが異なる乗算置き換えをサポート。

2. 同期モードまたは非同期モードをサポート。
3. 乗算のカスケードをサポート。
4. 乗算の累積をサポート。
5. 前置加算機能をサポート。
6. 入力レジスタ、出力レジスタ、バイパスレジスタの吸収を含む、レジスタの吸収をサポート。

### 4.3.3 DSP の制約

`syn_dspstyle` を使用して、特定のオブジェクトまたはグローバルの乗算器を DSP またはロジック回路で実装するかどうかを制御します。

`syn_perserve` は、レジスタを予約するために使用されます。DSP 周辺のレジスタにこの属性がある場合、DSP はこのレジスタを吸収できません。

制約ステートメントの特定の使用方法については、章 `syn_dspstyle` 、`syn_preserve` を参照してください。

### 4.3.4 DSP 推論のコードの例

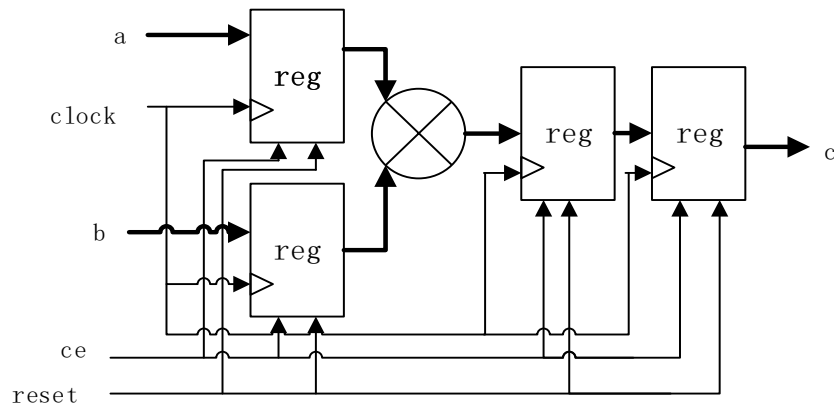
例 1 は、符号ビット付きの同期リセット乗算器として合成できます。この乗算器の入力レジスタは `ina` と `inb`、出力レジスタは `out_reg`、バイパスレジスタは `pp_reg` です。

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

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

上記の乗算器回路の説明図を図 4-15 に示します。

図 4-15 例 1 の DSP の回路図



例 2 は、非同期モードの乗算加算器として合成できます。この乗算加算器の入力レジスタは `a0_reg`、`a1_reg`、`b0_reg`、および `b1_reg` で、出力レジスタは `s_reg` で、バイパスレジスタは `p0_reg` および `p1_reg` です。

```

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

```

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



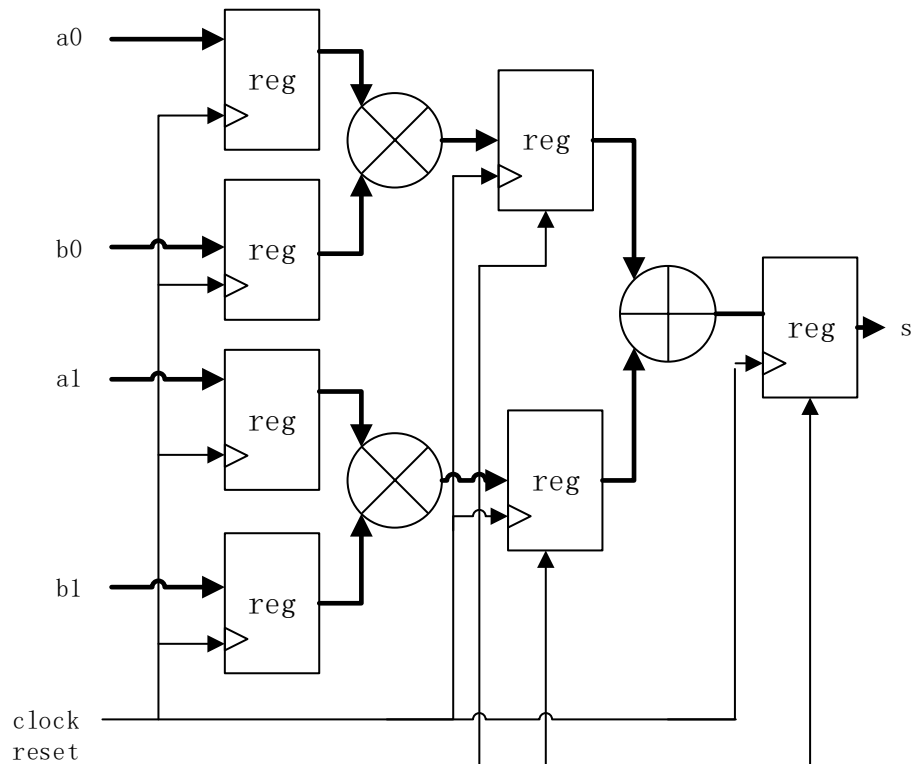
```

    end
  end
end
assign s = s_reg;
endmodule

```

上記の乗算加算器回路の説明図を図 4-16 に示します。

図 4-16 例 2 の DSP の回路図



例 3 は、カスケード関係にある 2 つの符号ビットなし乗算加算器として合成できます。

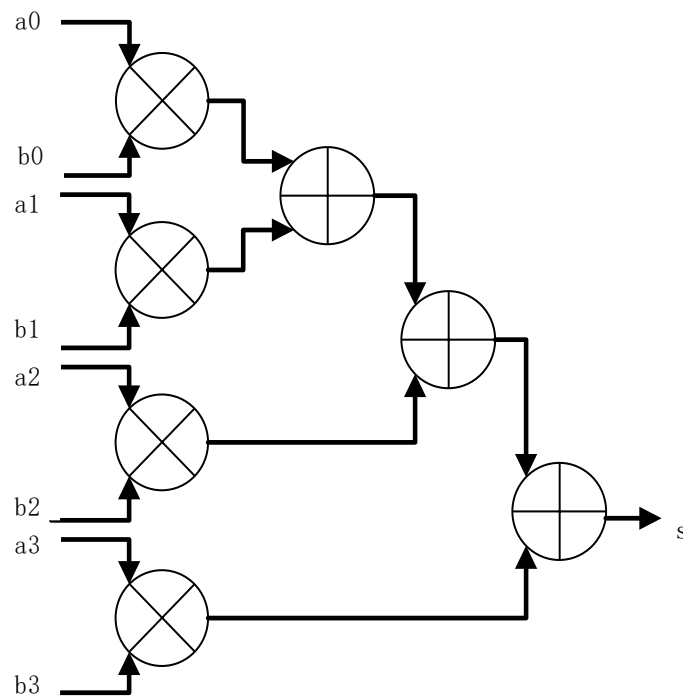
```

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

```

上記の乗算加算器回路の説明図を図 4-17 に示します。

図 4-17 例 3 の DSP の回路図



例 4 は、符号ビットが 0 の乗算器と符号ビットが 0 の前置加算器として合成できます。乗算器の 1 つの入力ポートと前置加算器の出力ポート **b** は互いに接続されます。

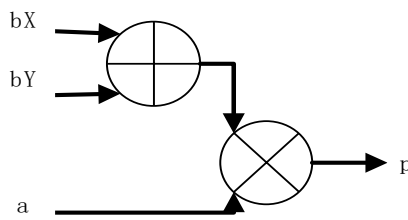
```

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

```

上記の乗算加算器回路の説明図を図 4-18 に示します。

図 4-18 例 4 の DSP の回路図



例 5 は、符号ビットが 0 の乗算アキュムレータとして合成できます。この乗算アキュムレータの出力レジスタは **s** です。

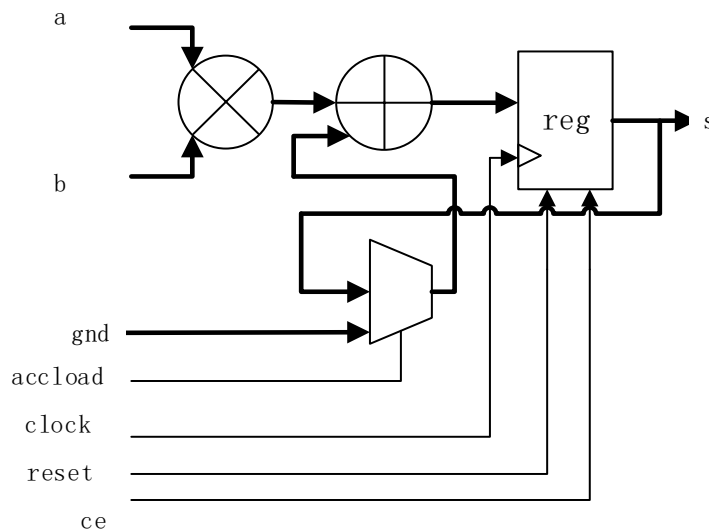
```

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

```

上記の乗算アキュムレータ回路の説明図を図 4-19 に示します。

図 4-19 例 5 の DSP の回路図



注記：

その他の例については、GOWIN の公式サイト [GowinSynthesis](#) 推論コーディングテンプレート「[GowinSynthesis Inference Coding Template](#)」を参照してください。

## 4.4 有限状態機械の合成ルール

### 4.4.1 有限状態機械の合成ルール

**GowinSynthesis** は有限状態機械(Finite State Machine,FSM)の合成をサポートするほか、ワンホットコード、グレイコード、バイナリコードなどエンコード方法をサポートしています。有限状態機械の合成結果は、状態機械のエンコード方式、エンコードの数、エンコードのビット幅、エンコードの制約、およびその他の情報に依存しています。エンコードの制約がない場合、**GowinSynthesis** は自動的にワンホットコードまたはグレイコードを選択して状態機械を実装します。コエンコードの制約がある場合、制約で指定されたエンコード方法に従って優先的に実装します。有限機械のエンコードの制約については、セクション **syn\_encoding** を参照してください。

注記：

有限状態機械の出力が出力ポートを直接駆動する場合、**GowinSynthesis** はそれを状態機械として合成せず、状態機械のエンコード制約は無視されます。

### 4.4.2 有限状態機械のコード例

有限状態機械の合成ルールを以下に説明します。

#### ワンホットコード状態機械

RTL デザインの状態機械がエンコーディングにワンホットコードを使用し、かつエンコード制約が設定されていない場合、**GowinSynthesis** はデフォルトでワンホットコードで状態機械の機能を実装します。エンコード制約がある場合、制約で指定されたエンコード方法で状態機械の機能を実装します。ワンホットコードのエンコード方法の例は次のとおりです。

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

上記の例では、RTL はエンコーディングにワンホットコードを使用し、GowinSynthesis は実装にワンホットコードを使用します。

#### グレイコード状態機械

RTL デザインの状態機械がエンコーディングにグレイコードを使用し、かつエンコード制約が設定されていない場合、合成ツールはデフォルトでグレイコードで状態機械の機能を実装します。エンコード制約がある場合、制約で指定されたエンコード方法で状態機械の機能を実装します。グレイコードのエンコード方法の例は次のとおりです。

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

上記の例では、RTL はエンコーディングにグレイコードを使用し、GowinSynthesis は実装にグレイコードを使用します。

#### バイナリコードまたはその他のエンコード方法を採用した状態機械

状態機械が RTL デザインでバイナリコードでエンコードされている場合、つまりワンホットコードでもグレイコードでもない場合。エンコード制約を設定しないと、GowinSynthesis はコードの数とビット幅に基づいて対応するエンコードを選択して実装します。選択の原則は次のとおりです。コード数がコードの有効ビット幅より大きい場合、グレイコードを使用して実装します。コードの数がコードの有効ビット幅以下の場合、ワンホットコードを使用して実装します。エンコード制約がある場合は、制約で指定されたエンコード方法に従って状態機械を実装します。

##### 例 1

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

上記の例では、コード数は 4、コードのビット幅は 6、有効ビット幅は 5 です。コード数がコードの有効ビット幅よりも小さいため、ワンホットコードが実装に使用されることになります。

## 例 2

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

上記の例では、コード数は 4、コードの有効ビット幅は 3 です。コード数がコードの有効ビット幅よりも小さいため、グレイコードが実装に使用されることになります。

## 例 3

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

上記の例では、コード数は 4 で、エンコードは 10 進数で、それを 2 進数に変換すると有効ビット幅は 4 ビットになります。コード数がコードの有効ビット幅に等しいため、ワンホットコードが実装に使用されることになります。

# 5 合成制約のサポート

属性の制約は、合成の結果が設計要件をよりよく満たすよう、合成プロジェクトの最適化の選択、機能の実装、出力ネットリストの形式などのさまざまな属性の設定に使用されています。属性の設定は、制約ファイルに書き込むか、ソースコードに書き込むことができます。

このセクションでは、RTL ファイルの制約と GowinSynthesis 制約ファイル GSC(Gowin Synthesis Constraint)制約の構文について説明します。Verilog ファイルは大文字と小文字が区別されるため、構文で説明されているとおりに命令と属性を正確に入力する必要があります。制約ステートメントの属性制約は、1 行で記述する必要があります。ステートメントの最後にセミコロンを追加する必要があります。

## RTL ファイル内の制約

RTL ファイル内の制約は、制約オブジェクトの定義ステートメント内に追加する必要があります。ステートメントの制約属性値(setting\_value)が文字列の場合、setting\_value を二重引用符で囲む必要があります。setting\_value が数値の場合、setting\_value を二重引用符で囲むではありません。

## GSC

GSC 制約は、Instance タイプ制約、Net タイプ制約、Port タイプ制約、およびグローバルオブジェクト制約に分けられます。異なるタイプを区別するために、異なる形式の構文があります。制約オブジェクトは二重引用符で囲む必要があります。attributeName(属性名)と制約属性値は二重引用符または他の記号でマークする必要はありません。等号の前後にスペースを入れることが許容されます。GSC 制約では「//」を使用したコメントがサポートされています。具体的な構文は次のとおりです：

```
INS "object" attributeName=setting_value;
```

```
NET "object" attributeName=setting_value;
```

```
PORT "object" attributeName=setting_value;
```

```
GLOBAL attributeName=setting_value;
```

制約ステートメントは INS で始まり、object は instance 名である必要が

あります。instance には module/entity instance と primitive instance が含まれています。instance 名には角かっこは不要です。

制約ステートメントは NET で始まり、制約オブジェクトは net 名でなければなりません。

制約ステートメントは PORT で始まり、制約オブジェクトは port 名でなければなりません。

制約ステートメントは GLOBAL で始まる場合、後続の属性制約がグローバル属性制約であることを示します。制約オブジェクトはグローバルです。

制約のオブジェクト名はネットリストの名前と一致する必要があります。名前にスペースがあってはなりません。オブジェクト名にはワイルドカードがサポートされています。名前間の階層関係を区別するには「/」を使用します。ワイルドカードを使用する場合、区別するためにオブジェクト名の前に w が付きます(例えば、w "object")。

制約属性値(setting\_value)の設定値は、ユーザーが直接指定した値、上部構造から継承した値、または属性のデフォルト値です。値の優先順位は、直接値>継承値>デフォルト値であり、複数の継承値がある場合、指定された名前(最も低いレベル)に最も近い値が使用されます。たとえば、A/D/C/mult1(「/」はモジュール名間の階層関係を示す)の MULT\_STYLE 属性を照会する場合、直接値 dsp があります。A/D/C の MULT\_STYLE 属性を照会する場合、直接値がなく、MULT\_STYLE 属性を継承できるため、A/D の属性値 logic を見つけて継承します。A/D/C/mult1 の MULT\_STYLE を照会する場合、A/D 継承値も直接値もあり、直接値の方が優先度が高いため、最終的に直接値 DSP が使用されます。

## 5.1 black\_box\_pad\_pin

### 説明

ブラックボックスの I/O パッドが外部から見えるように指定します。この属性は、ブラックボックスの I/O パッドでのみ機能します。

この属性は、RTL ファイルでのみ指定できます。

### 構文

Verilog 構文

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

VHDL 構文

```
attribute black_box_pad_pin : string;
```

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

注記：

- object : ブラックボックスで定義されたモジュールまたはコンポーネントです。
- 二重引用符で囲まれた portList は、ブラックボックス上のポートの名前のスペースなしのコンマ区切りのリストです。



## 例

Verilog の例

```

module top(clk, in1, in2, out1, out2,D,E);
input clk;
input [1:0]in1;
input [1:0]in2;
output [1:0]out1;
output [1:0]out2;
output D,E;
.....

black_box_add U2 (in1, in2, out2,D,E);
endmodule

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

```

VHDL の例

```

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

```

```

        q : out std_logic_vector (width1 downto 0)
    );
end component;
attribute black_box_pad_pin : string;
attribute black_box_pad_pin of test : component is "q";
begin
test123 : test generic map (width) port map (in1,in2,clk,q);
end top1_arch;
```

## 5.2 full\_case

### 説明

**full\_case** は、Verilog のデザインでのみ使用されます。**case**、**casex** または **casez** ステートメントが使用される場合、この属性は、すべての可能な値が取得されたことを示し、信号値を保持するために追加のハードウェアを使用する必要はありません。

この属性は、RTL ファイルでのみ指定でき、Verilog 言語の設計のみをサポートしています。

### 構文

Verilog 構文

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

### 例

Verilog の例

例 1 は、回路のこの部分が信号値を保持するために追加のハードウェアを必要としないことを指定します。

```

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

*endmodule*

## 5.3 parallel\_case

命令。priority-encoded(優先順位エンコード)構造の代わりに、parallel-multiplexed(並列多重化)構造を使用するように強制します。

この命令は、Verilog ファイルでのみ指定できます。

### 説明

**case** ステートメントは、デフォルトで優先順位に従って機能し、選択した値に一致する最初のステートメントのみを実行するように定義されています。**priority-encoded** では、複数の入力ポートで同時に入力信号があることが可能になります。この場合、同時に入力された複数の信号の中で優先度が最も高い信号のみがエンコードされます。

選択したバスが現在のモジュールの外部から駆動され、現在のモジュールに法的な選択値に関する情報がない場合、ソフトウェアは、後続のすべてのステートメントを無効にするためのディセーブル用のロジックチェーンを作成する必要があります。

ただし、法的な選択値がわかっている場合は、**parallel\_case** 命令を使用して、余分な優先順位エンコードのロジックを回避できます。

### 構文

Verilog 構文

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

注記：

- global support : No
- object : case、casex、または casez ステートメント。
- setting\_value : 値は不要です。

### 例

Verilog の例

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

```

        4'b1???: out = d;
    default: out = 'bx;
endcase
end
endmodule

```

## 5.4 syn\_black\_box

### 説明

モジュールまたはコンポーネントをブラックボックスとして指定します。合成する場合、ブラックボックス・モジュールはそのインターフェースのみが定義され、そのコンテンツはアクセスできず、最適化できません。モジュールが空であるかどうかに関係なく、ブラックボックスと見なされます。

この属性は、RTL ファイルでのみ指定できます。

### 構文

#### Verilog 構文

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

#### VHDL 構文

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

### 注記：

- object：オブジェクトの指定。sub module/entity に限定されます。
- objectType：オブジェクトのタイプ(通常はコンポーネント)。

### 例

#### Verilog の例

```

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

```

```
module add (clk, in1, in2, out1);
```

```

.....

begin
out1 <= in1 + in2;
end
endmodule

module black_box_add(A, B, C)/* synthesis syn_black_box */;
.....

assign C = A + B;
endmodule

```

この属性を使用する前に、モジュール `black_box_add` のコンテンツが表示されます。この属性を使用すると、モジュール `black_box_add` のコンテンツは非表示になり、ブラックボックスになります。

VHDL の例

```

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

architecture Behavioral of mux2_1_top is
.....

    attribute syn_black_box: boolean;
    attribute syn_black_box of mux2_1 : component is true;
begin
    u_mux2_1 : mux2_1
    port map(
        dina => dina,
        dinb => dinb,

```

```

        sel => sel,
        dout => dout
    );
end Behavioral;

```

## 5.5 syn\_dspstyle

### 説明

乗算器が、専用の **DSP** ハードウェアモジュールまたは論理回路の形式で実装されるかを指定します。特定の **module/entity instance** またはグローバルに適用できます。この属性は、**GSC** および **RTL** ファイルで指定できます。

### 構文

GSC 制約の構文

```

INS "object" syn_dspstyle =setting_value;
GLOBAL syn_dspstyle =setting_value;

```

Verilog 制約の構文

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

VHDL 構文

```

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

```

### 注記：

- **object**：指定されるオブジェクトは、**wire**、**register**、**module/entity** 名または **module/entity instance** 名です。
- **setting\_value**：乗算器の実装。現在、**dsp**、**logic** をサポートしています。
- **setting\_value** が **logic** の場合：**object** を論理回路にマップします。
- **setting\_value** が **dsp** の場合：**object** を **DSP** にマップします(デフォルト)。

### 例

GSC 制約の例

例 1 では、**instance** の実装を **logic** として指定しています。

```

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

```

例 2 では、グローバルでのすべての乗算器の実装を **logic** として指定しています。

```
GLOBAL syn_dspstyle=logic;
```

Verilog の例

例 1 では、**mult module** 内のすべての乗算器の実装を **logic** として指定しています。

```

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

wire [15:0] temp;
assign temp = a*b;
.....

endmodule

```

例 2 では、乗算器 **temp** の実装を **logic** として指定しています。

```

module mult(···) ;
.....

wire [15:0] temp/* synthesis syn_dspstyle = "logic" */;
assign temp = a*b;
.....

endmodule

```

VHDL の例

例 1 では、乗算器 **result** の実装を **logic** として指定しています。

```

entity Mult is
port(
.....

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

architecture Behavior of Mult is
    signal x1 : signed(11 downto 0);
    signal y1 : signed(11 downto 0);
begin
.....

    result <= x1 * y1;
end Behavior;

```

## 5.6 syn\_encoding

### 説明

状態機械コンパイラーのエンコード方法の指定。特定のオブジェクトとグローバルの両方に適用できます。

この属性は、RTL ファイルでのみ指定できます。

### 構文

Verilog 構文

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

VHDL 構文

```
attribute syn_encoding : string;
```

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

注記：

- **object** : オブジェクトの指定。**register** 名に限定されます。
- **setting\_value** : 状態機械のエンコード方法。現在、**onehot** と **gray** をサポートしています。

### 例

Verilog :

例 1 は、状態機械のエンコード方法を **gray** エンコーディングとして指定しています。

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

VHDL の例

例 1 は、状態機械のエンコード方法を **onehot** エンコーディングとして指定しています。

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



## 5.7 syn\_insert\_pad

### 説明

I/O buffer を挿入するかどうかを指定します。属性値が 1 の場合、I/O buffer が挿入されます。

この属性は、GSC ファイルでのみ指定できます。

### 構文

GSC 制約の構文

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

注記：

- setting\_value : 0 または 1。0 の場合、I/O buffer が削除され、1 の場合、I/O buffer が挿入されます。
- object : port に限定。この制約は、Input port または Output port にのみ適用され、Inout port には適用されません。

### 例

GSC の例

例 1 では、I/O buffer が挿入されます。

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

例 2 では、I/O buffer が削除されます。

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

## 5.8 syn\_keep

### 説明

wire、reg、port をプレースホルダーとして指定し、最適化しないままにします。

この属性は、RTL ファイルでのみ指定できます。

### 構文

Verilog 構文

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

VHDL 構文

```
attribute syn_keep : integer;
```

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

注記：

- object : オブジェクトの指定。wire、port、および組み合わせロジックに限定されます。
- setting\_value : 0 または 1 に限定。1 の場合、この net は最適化されません。

### 例

#### Verilog の例

例 1 では、mywire が最適化されないように指定しています

```
module test (...);  
.....  
wire mywire /* synthesis syn_keep=1 */;  
.....  
endmodule
```

#### VHDL の例

例 1 では、tmp0 が最適化されないように指定しています。

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

## 5.9 syn\_looplimit

### 説明

設計のループ反復制限を指定します。デフォルトのループ反復回数は 2000 です。設計でループ反復回数が 2000 を超え、ループ数が指定されていない場合、合成中にエラーが報告されます。

この属性は GSC でのみ設定できます。

### 構文

#### GSC 制約の構文

GLOBAL syn\_looplimit=setting\_value

#### 注記：

- setting\_value : 数字に限定。 この設計のループ反復回数の上限を表します。

### 例

GSC 制約の例

```
GLOBAL syn_looplevelimit=3000
```

## 5.10 syn\_maxfan

### 説明

最大ファンアウト値の指定。特定のオブジェクトとグローバルの両方に適用できます。

この属性は、GSC および RTL ファイルで指定できます。

### 構文

GSC 制約の構文

```
INS "object" syn_maxfan=setting_value;
```

```
NET "object" syn_maxfan=setting_value;
```

```
GLOBAL syn_maxfan=setting_value;
```

Verilog 構文

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

VHDL 構文

```
attribute syn_maxfan : integer;
```

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

### 注記：

- object : 指定されるオブジェクトは、wire、register、input、output、module/entity 名、module/entity instance 名です。
- setting\_value : 0 より大きい整数。

### 例

GSC の例

例 1 は、instance の最大ファンアウト値 10 を指定します

```
INS "d" syn_maxfan=10;
```

例 2 は、グローバルの最大ファンアウト値 100 を指定します

```
GLOBAL syn_maxfan=100;
```

例 3 は、instance の最大ファンアウト値 10 を指定します

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

例 4 は、net の最大ファンアウト値 10 を指定します

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

Verilog の例

例 1 は、module 内のすべての instance の最大ファンアウト値 3 を指定

します(CLK を除く)。

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

endmodule
```

例 2 は、instance の最大ファンアウト値 3 を指定します

```
module test (···);
reg [7:0] d /* synthesis syn_maxfan = 3*/;
.....

endmodule
```

VHDL の例

```
entity test is
.....

end test;

architecture rtl of test is
signal d : std_logic;
attribute syn_maxfan : integer;
attribute syn_maxfan of d : signal is 5;
.....

end rtl;
```

## 5.11 syn\_netlist\_hierarchy

### 説明

階層ネットリストを生成するかどうかを指定します。デフォルト値(1)の場合、階層ネットリストを生成します; 0 の場合、階層ネットリストはフラット化されて出力されます。

この属性は、GSC および RTL ファイルで指定できます。

### 構文

GSC 制約の構文

```
GLOBAL syn_netlist_hierarchy=setting_value;
```

Verilog 構文

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

VHDL 構文

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

注記 :

- **object** : オブジェクトの指定。**top module/entity** に限定されます。
- **setting\_value** : 0 または 1。1 の場合、**hierarchy** を生成できます。0 の場合、出力階層のネットリストがフラット化されます。

### 例

GSC の例

```
GLOBAL syn_netlist_hierarchy=0;
```

Verilog :

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

```
.....
```

```
endmodule
```

VHDL の例

```
entity mux4_1_top is
```

```
port(
```

```
    dina : in bit;
```

```
    dinb : in bit;
```

```
    sel  : in bit;
```

```
    dout : out bit
```

```
);
```

```
attribute syn_netlist_hierarchy: integer;
```

```
attribute syn_netlist_hierarchy of mux4_1_top: entity is 0;
```

```
end mux4_1_top;
```

## 5.12 syn\_noprune

### 説明

**module/entity instance** や **primitive instance**、またはブラックボックス(プリミティブを含む)の出力がすべてフローティング状態の場合に最適化(削除または合併)されるかどうかを指定します。これは、特定のオブジェクトとグローバルの両方に適用できます。

この属性は、RTL ファイルでのみ指定されます。

### 構文

Verilog 構文

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

VHDL 構文

```
attribute syn_noprune : integer;
```

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

注記：

- **object** : 指定されるオブジェクト名は、**module/entity instance** 名、**primitive instance** 名、またはブラックボックスです。
- **setting\_value** : 0 または 1。1 の場合、インスタンスとブラックボックスは保持され、0 の場合、インスタンスとブラックボックスは必要に応じて削除されます。

例

Verilog の例

```
module test (out1,out2,clk,in1,in2);
.....

noprune_bb u1(out1,in1)/*synthesis syn_noprune=1*/;
.....

endmodule

module noprune_bb(din,dout);
input din;
output dout;
endmodule
```

VHDL の例

```
library ieee;
use ieee.std_logic_1164.all;
entity top is
.....

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

end architecture arch;
```

## 5.13 syn\_preserve

### 説明

レジスタまたはレジスタロジックを最適化(削除または合併)するかどうかの指定。特定のオブジェクトとグローバルの両方に適用できます。

この属性は、RTL ファイルおよび GSC ファイルで指定できます。

### 構文

GSC 制約の構文

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

```
GLOBAL syn_preserve=setting_value;
```

Verilog 構文

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

VHDL 構文

```
attribute syn_preserve : integer;
```

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

### 注記：

- **object** : 指定されるオブジェクトは、**register** 名、**module /entity** 名、または **module /entity instance** 名です。
- **setting\_value** : 0 または 1。1 の場合、対応するレジスタは保持され、0 の場合、対応するレジスタは必要に応じて削除されます。

### 例

GSC の例

例 1 では、**reg1** が最適化されないように指定しています

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

例 2 では、デザイン内のすべてのレジスタを保持するように指定しています

```
GLOBAL syn_preserve =1;
```

Verilog :

例 1 では、**module** 内のすべてのレジスタを保持するように指定しています

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

```
.....
```

```
endmodule
```

例 2 では、**reg1** が最適化されないように指定しています

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

```
.....
```

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

```
.....
```

```
endmodule
```

VHDL の例

例 1 では、レジスタ **reg1** を保持するように指定しています。

```
entity syn_test is
```

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

```
    );
```

```
end syn_test;
```

```
architecture behave of syn_test is
```

```
    signal reg1 : std_logic;
```

```
    signal reg2 : std_logic;
```

```
    attribute syn_preserve : integer;
```

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

```
begin
```

```
.....
```

```
end behave;
```

## 5.14 syn\_probe

### 説明

この属性は、プローブポイントを挿入して、デザインの内部信号をテストおよびデバッグします。指定されたプローブポイントは、トップレベルのポートリストに **port** として表示されます。

この属性は、RTL ファイルでのみ指定できます。

### 構文

Verilog 構文

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

VHDL 構文

```
attribute syn_probe: string;
```

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

注記：

- **object**：オブジェクトの指定。**wire** または **register** に限定されます。
- **setting\_value** が 1 の場合：プローブポイントを挿入し、**net** の名前からプローブポートの名前を自動的に取得します。
- **setting\_value** が 0 の場合：プローブは許可されません。



- **setting\_value** が文字列の場合：指定された名前のプローブポイントを挿入します。  
**Setting\_value** で指定された名前が **bus** の場合、挿入された名前の後に番号が自動的に追加されます。
- **gowinSyn** では、**setting\_value** が **object** 名または **module** の **port** と同じであってはなりません。

### 例

Verilog 制約の例：**probe\_tmp** がこのプロパティに設定されると、**probe\_tmp** がトップレベルの出力ポートリストにリストされます。

```
module test (...);
.....
reg [7:0] probe_tmp /* synthesis syn_probe=1*/;
.....
endmodule
```

VHDL の例

```
entity halfadd is
port(.....);
end halfadd;
architecture add of halfadd is
    signal probe_tmp: std_logic;
    attribute syn_probe: string;
    attribute syn_probe of probe_tmp: signal is "probe_string";
.....
end;
```

## 5.15 syn\_ramstyle

### 説明

メモリの実装方法の指定。特定のオブジェクトとグローバルの両方に適用できます。

この属性は、**GSC** および **RTL** ファイルで指定できます。

### 構文

**GSC** 制約の構文

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

Verilog 構文

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

VHDL 構文

```
attribute syn_ramstyle:string;
```

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

注記：

- **object**：指定されるオブジェクトは、**module/entity** 名、**module/entity instance** 名、または **register** です。
- **setting\_value**：メモリの実装。現在、**block\_ram**、**distributed\_ram**、**registers**、**rw\_check**、**no\_rw\_check** をサポートしています。
- **setting\_value** が **registers** の場合：inferred RAM を専用の RAM リソースではなく、**registers**(フリップフロップおよび論理回路)にマップします。
- **setting\_value** が **block\_ram** の場合：inferred RAM を、FPGA の専用メモリリソースを使用する適切なデバイス専用のメモリにマップします。
- **setting\_value** が **distributed\_ram** の場合：inferred RAM を適切なデバイス専用のメモリ(分散メモリブロック)にマップします。
- **setting\_value** が **rw\_check/no\_rw\_check** の場合：デフォルト(**rw\_check**)では、同じアドレスで読み出しと書き込みすると、不確実な外部入力によりシミュレーションの不一致が発生します。このとき、GowinSynthesis は、このシミュレーションの不一致を回避するために、推論された RAM の周りに **bypass** ロジックを挿入します。この属性値が **no\_rw\_check** の場合、読み出しと書き込みはチェックされず、推論された RAM の周りに **bypass** ロジックは挿入されません。

## 例

### GSC 制約の例

例 1 では、**instance** の実装を BSRAM として指定しています。

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

例 2 では、グローバルでのすべてのメモリの実装を SSRAM として指定しています。

```
GLOBAL syn_ramstyle=distributed_ram;
```

### Verilog の例

例 1 では、**module** 内のすべてのメモリの実装は **block\_ram** として指定されており、読み出しと書き込みはチェックされません。

```
module test (···) /* synthesis syn_ramstyle = " no_rw_check,block_ram" */;
```

```
.....
```

```
endmodule
```

例 2 では、**instance** の実装を BSRAM ハードコアとして指定しています。

```
module test (···) ;
```

```
.....
```

```
reg [DATA_W - 1 : 0] mem [(2**ADDR_W) - 1 : 0] /* synthesis syn_ramstyle = "block_ram" */;
```

```
.....
```

```
endmodule
```

## VHDL の例

例 1 では、メモリの実装を **BSRAM** として指定しています。

```
entity ram is
```

```
  GENERIC(bits:INTEGER:=8;
```

```
    words:INTEGER:=256);
```

```
  PORT(……);
```

```
end ram;
```

```
ARCHITECTURE arch of ram IS
```

```
  TYPE vector_array IS ARRAY(0 TO words-1) OF  
  STD_LOGIC_VECTOR(bits-1 DOWNT0 0);
```

```
  SIGNAL memory:vector_array ;
```

```
    attribute syn_romstyle:string;
```

```
    attribute syn_romstyle of memory : signal is " block_ram";
```

```
BEGIN
```

```
  ……
```

```
end arch;
```

## 5.16 syn\_romstyle

### 説明

読み出し専用メモリの実装方法の指定。特定のオブジェクトとグローバルの両方に適用できます。

この属性は、**GSC** および **RTL** ファイルで指定できます。

### 構文

GSC 制約の構文

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

```
GLOBAL syn_romstyle =setting_value;
```

Verilog 構文

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

VHDL 構文

```
attribute syn_romstyle:string;
```

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

### 注記：

- object：指定されるオブジェクトは、module/entity 名、module/entity instance 名、または register です。

- **setting\_value** : 読み出し専用メモリの実装。現在、**block\_rom**、**distributed\_rom**、**logic** をサポートしています。

## 例

### GSC 制約の例

例 1 では、**instance** の実装を **BSRAM** として指定しています。

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

例 2 では、グローバルでのすべてのメモリの実装を **SSRAM** として指定しています。

```
GLOBAL syn_romstyle=distributed_rom;
```

### Verilog の例

例 1 では、**module** 内のすべてのメモリの実装を **SSRAM** として指定しています。

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

```
.....
```

```
endmodule
```

### VHDL の例

例 1 では、**module** 内のすべてのメモリの実装を **SSRAM** として指定しています。

```
ENTITY rom is
```

```
.....
```

```
end rom;
```

```
ARCHITECTURE rom OF rom IS
```

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

```
    attribute syn_romstyle:string;
```

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

```
.....
```

```
END rom;
```

## 5.17 syn\_srlstyle

### 説明

**shift registers** の実装方法の指定。特定のオブジェクトとグローバルの両方に適用できます。**shift register** は、**BSRAM**、**SSRAM**、およびレジスタによって実装できます。デフォルトでは、実装方法は **shift register** 内のレジスタの数に基づいて決定されます。デフォルト値は、**syn\_srlstyle** を使用して変更できます。

この属性は、**GSC** および **RTL** ファイルで指定できます。

## 構文

GSC 制約の構文

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

```
GLOBAL syn_srlstyle =setting_value
```

Verilog 構文

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

VHDL 構文

```
attribute syn_srlstyle:string;
```

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

**object:** 指定されるオブジェクトは、**module/entity** 名、または **module/entity instance** 名です。

注記：

- **object**：指定されるオブジェクト名は、**module/entity**、**module/entity instance**、または **register** です。**module/entity** は GSC 構文ではサポートされていません。
- **setting\_value**：メモリの実装。現在、**block\_ram**、**distributed\_ram**、**registers** をサポートしています。

## 例

GSC 制約の例

例 1 では、**instance** の実装を **BSRAM** として指定しています。

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

例 2 では、グローバルでのすべてのメモリの実装を **SSRAM** として指定しています。

```
GLOBAL syn_srlstyle=distributed_ram
```

Verilog の例

例 1 では、**module** 内のすべてのレジスタの実装を **block\_ram** として指定しています。

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

```
.....
```

```
endmodule
```

例 2 では、**instance** の実装を **BSRAM** として指定しています。

```
module test (···) ;
```

```
.....
```

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

```
.....
```

```
endmodule
```

## VHDL の例

例 1 では、**module** 内のすべてのレジスタの実装を **register** として指定しています。

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

ARCHITECTURE arch of ram IS
  ……
end ram;
```

## 5.18 syn\_tlvds\_io/syn\_elvds\_io

## 説明

差動 I/O buffer マッピングの属性の指定。特定のオブジェクトとグローバルの両方に適用できます。この属性は、**GSC** および **RTL** ファイルで指定できます。

## 構文

GSC 制約の構文

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

Verilog 構文

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

VHDL 構文

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

注記：

- **object** : オブジェクトの指定。module/entity 名または port 名です。
- **setting\_value** : 0 または 1。

## 例

GSC 制約の例

例 1 では、**buffer** の実装を **TLVDS** として指定しています。

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

例 2 では、グローバルでの **buffer** の実装を **TLVDS** として指定しています。

```
GLOBAL syn_tlvds_io =1;
```

Verilog の例

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

VHDL の例

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

## 5.19 translate\_off/translate\_on

### 説明

**translate\_off/translate\_on** はペアで現れる必要があります。**translate\_off** の後のステートメントは、**translate\_on** が表示されるまで合成中にスキップされるため、**translate\_off** は通常ステートメントの自動ブロックに使用されます。

この属性は、RTL ファイルでのみ指定できます。

### 構文

Verilog 構文

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

合成中にスキップされるステートメント

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

VHDL 構文

```
-- synthesis translate_off
```

合成中にスキップされるステートメント

```
-- synthesis translate_on
```

例

Verilog の例

例 1 */\*synthesis translate\_off\*/*と*/\*synthesis translate\_on\*/*の間の assign Nout =a\*b は合成中にスキップされます。

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

VHDL の例

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



# 6 Report ファイル

Report ファイルは、合成実行後に生成される統計レポートファイルで、ファイル名は\*\_syn.rpt.html(\*は指定された出力ネットリスト vg ファイルの名前)です。このファイルには、Synthesis Message、Design Settings、Resource、Timing、Message、および Summary の 6 つの部分が含まれています。これらの 6 つの部分を以下に紹介します。

## 6.1 Synthesis Message

Synthesis Message は合成の基本情報です。図 6-1 に示すように、主に合成の設計ファイル、現在の GowinSynthesis バージョン、構成情報、およびランタイム情報が含まれています。

図 6-1 Synthesis Message

Synthesis Messages

Report Title	GowinSynthesis Report
Design File	E:\tmp\mantis\10631\mantis10542\src\test.v
GowinSynthesis Constraints File	---
GowinSynthesis Version	GowinSynthesis V1.9.8
Part Number	GW1NSR-LX2QN48PES
Device	GW1NSR-2
Created Time	Mon Aug 16 15:00:53 2021
Legal Announcement	Copyright (C)2014-2021 Gowin Semiconductor Corporation. ALL rights reserved.

## 6.2 Synthesis Details

Synthesis Details : 設計ファイルのトップモジュール、合成の各段階の実際の実行時間とメモリ使用量、合計実行時間と合計メモリ使用量が表示されます(図 6-2)。

## 図 6-2 Synthesis Details

### Synthesis Details

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

## 6.3 Resource

**Resource** はリソース情報です。主に、リソース使用率が含まれています。

リソース使用テーブル(**Resource Usage Summary**)には、ユーザーデザインの I/O Port、I/O Buf、REG、LUT などの数がリストされています。リソース使用率テーブル(**Resource Utilization Summary**)は、ユーザーデザインの現在のデバイスにおける、CFU Logics、Register、BSRAM、DSP などのリソース使用率を推定するために使用されます(図 6-3)。

### 図 6-3 Resource

## Resource

### Resource Usage Summary

Resource	Usage
I/O Port	16
I/O Buf	16
IBUF	11
OBUF	5
Black Box	1
test	1

### Resource Utilization Summary

Resource	Usage	Utilization
Logic	0(0 LUTs, 0 ALUs) / 4608	0%
Register	0 / 4020	0%
--Register as Latch	0 / 4020	0%
--Register as FF	0 / 4020	0%
BSRAM	0 / 10	0%

## 6.4 Timing

Timing はタイミングの統計情報です。主に、Clock Summary、Max Frequency Summary、Detail Timing Paths Informations などの情報が含まれています。

Clock Summary は、主にネットリストのクロック信号について説明します。図 6-4 に示すように、クロックは、周波数が 100MHz、周期が 10ns であり、0ns の時点が立ち上がりエッジ、5ns の時点が立ち下がりエッジです。

図 6-4 Timing

### Timing

#### Clock Summary:

Clock Name	Type	Period	Frequency(MHz)	Rise	Fall	Source	Master	Object
clk	Base	10.000	100.0	0.000	5.000			clk_ibuf/l

Max Frequency Summary には、主にネットリストファイルの最大クロック周波数がリストされています。これによりネットリストファイル全体のタイミングが要件を満たしているかどうかを判断します。図 6-5 に示すとおりです。必要な周波数は 100MHz で、実際のクロック周波数は 747.2MHz であり、タイミング要件を満たしています。実際の周波数が必要な周波数に達していない場合は、タイミング要件が満たされていないことになります。この場合、特定のタイミングパスをさらに確認する必要があります。

図 6-5 Performance Summary

#### Max Frequency Summary:

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

Detail Timing Paths Information には、詳細なタイミングパス情報が表示されます。デフォルトは 5 で、すべての時間値のデフォルトの単位はナノ秒です。Path Summary は、主に、ネットリストファイル内のクリティカルパス、始点、および遅延情報について説明します(図 6-6)。Data Arrival Path と Data Require Path は、主にクリティカルパスを説明します。ここでは、詳細な接続関係、ファンアウト情報を図 6-7 に示します。Path Statistics は、図 6-8 に示すように、パスの遅延情報を示します。

## 図 6-6 Path Summary

### Detail Timing Paths Information

#### Path 1

##### Path Summary:

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

## 図 6-7 接続関係、遅延、およびファンアウト情報

##### Data Arrival Path:

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

##### Data Required Path:

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

## 図 6-8 Path Statistics

##### Path Statistics:

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

